The Rust Programming Language 日本語版

著：Steve Klabnik、Carol Nichols、貢献：Rust コミュニティ

このテキストのこの版では Rust 1.58（2022 年 1 月 13 日リリース）かそれ以降が使われていることを前提にしています。Rust をインストールしたりアップデートしたりするには第 1 章の「インストール」節を読んでください。

HTML 版は https://doc.rust-lang.org/stable/book/で公開されています。オフラインのときは、`rustup` でインストールした Rust を使って `rustup docs --book` で開けます。

> 訳注：日本語の HTML 版は https://doc.rust-jp.rs/book-ja/で公開されています。`rustup` を使ってオフラインで読むことはできません。

また、コミュニティによるいくつかの翻訳版もあります。

このテキストの（英語版の）ペーパーバック版と電子書籍版は No Starch 出版から発売されています。

まえがき

すぐにはわかりにくいかもしれませんが、Rust プログラミング言語は、エンパワーメント (empowerment) を根本原理としています: どんな種類のコードを現在書いているにせよ、Rust は幅広い領域で以前よりも遠くへ到達し、自信を持ってプログラムを組む力を与え (empower) ます。

一例を挙げると、メモリ管理やデータ表現、並行性などの低レベルな詳細を扱う「システムレベル」のプログラミングがあります。伝統的にこの分野は難解で、年月をかけてやっかいな落とし穴を回避する術を習得した選ばれし者にだけ可能と見なされています。そのように鍛錬を積んだ者でさえ注意が必要で、さもないと書いたコードがクラッキングの糸口になったりクラッシュやデータ破損を引き起こしかねないのです。

この難しさを取り除くために、Rust は、古い落とし穴を排除し、その過程で使いやすく役に立つ洗練された一連のツールを提供します。低レベルな制御に「下がる」必要があるプログラマは、お決まりのクラッシュやセキュリティホールのリスクを負わず、気まぐれなツールチェーンのデリケートな部分を学ぶ必要なく Rust で同じことができます。さらにいいことに、Rust は、スピードとメモリ使用の観点で効率的な信頼性の高いコードへと自然に導くよう設計されています。

既に低レベルコードに取り組んでいるプログラマは、Rust を使用してさらなる高みを目指せます。例えば、Rust で並列性を導入することは、比較的低リスクです: コンパイラが伝統的なミスを捕捉してくれるのです。そして、クラッシュや脆弱性の糸口を誤って導入しないという自信を持ってコードの大胆な最適化に取り組めるのです。

ですが、Rust は低レベルなシステムプログラミングに限定されているわけではありません。十分に表現力豊かでエルゴノミックなので、コマンドラインアプリや Web サーバ、その他様々な楽しいコードを書けます。この本の後半に両者の単純な例が見つかるでしょう。Rust を使うことで 1 つの領域から他の領域へと使い回せる技術を身につけられます; ウェブアプリを書いて Rust を学び、それからその同じ技術をラズベリーパイを対象に適用できるのです。

この本は、ユーザに力を与え (empower) る Rust のポテンシャルを全て含んでいます。あなたの Rust の知識のみをレベルアップさせるだけでなく、プログラマとしての全般的な能力や自信をもレベルアップさせる手助けを意図した親しみやすくわかりやすいテキストです。さあ、飛び込んで学ぶ準備をしてください。Rust コミュニティへようこそ！

- ニコラス・マットサキス (Nicholas Matsakis) とアーロン・チューロン (Aaron Turon)

はじめに

> 注釈: この本のこの版は、本として利用可能な The Rust Programming Language と、No Starch Press の ebook 形式と同じです。

The Rust Programming Language へようこそ。Rust に関する入門書です。

Rust プログラミング言語は、高速で信頼できるソフトウェアを書く手助けをしてくれます。高レベルのエルゴノミクス (訳注 : ergonomics とは、人間工学的という意味。砕いて言えば、人間に優しいということ) と低レベルの制御は、しばしばプログラミング言語の設計においてトレードオフの関係になります; Rust は、その衝突に挑戦しています。バランスのとれた強力な技術の許容量と素晴らしい開発者経験を通して、Rust は伝統的にそれらの制御と紐付いていた困難全てなしに低レベルの詳細 (メモリ使用など) を制御する選択肢を与えてくれます。

Rust は誰のためのものなの

Rust は、様々な理由により多くの人にとって理想的です。いくつか最も重要なグループを見ていきましょう。

開発者チーム

Rust は、いろんなレベルのシステムプログラミングの知識を持つ開発者の巨大なチームとコラボするのに生産的なツールであると証明してきています。低レベルコードは様々な種類の微細なバグを抱える傾向があり、そのようなバグは他の言語だと広範なテストと、経験豊富な開発者による注意深いコードレビューによってのみ捕捉されるものです。Rust においては、コンパイラが並行性のバグも含めたこのようなとらえどころのないバグのあるコードをコンパイルするのを拒むことで、門番の役割を担います。コンパイラとともに取り組むことで、チームはバグを追いかけるよりもプログラムのロジックに集中することに、時間を費やせるのです。

Rust はまた、現代的な開発ツールをシステムプログラミング世界に導入します。

- Cargo は、付属の依存関係管理ツール兼ビルドツールで、依存関係の追加、コンパイル、管理を容易にし、Rust のエコシステム全体で一貫性を持たせます。
- Rustfmt フォーマットツールは開発者の間で一貫したコーディングスタイルを保証します。

- Rust 言語サーバーは、IDE(統合開発環境) との統合により、コード補完やインラインエラーメッセージに対応しています。

これらのツールや Rust のエコシステムの他のツールを使用することで、開発者はシステムレベルのコードを書きながら生産性を高めることができます。

学生

Rust は、学生やシステムの概念を学ぶことに興味のある方向けです。Rust を使用して、多くの人が OS 開発などの話題を学んできました。コミュニティはとても暖かく、喜んで学生の質問に答えてくれます。この本のような努力を通じて、Rust チームはシステムの概念を多くの人、特にプログラミング初心者にとってアクセス可能にしたいと考えています。

企業

数百の企業が、大企業、中小企業を問わず、様々なタスクにプロダクションで Rust を使用しています。そのタスクには、コマンドラインツール、Web サービス、DevOps ツール、組み込みデバイス、オーディオとビデオの解析および変換、暗号通貨、生物情報学、サーチエンジン、IoT アプリケーション、機械学習、Firefox ウェブブラウザの主要部分さえ含まれます。

オープンソース開発者

Rust は、Rust プログラミング言語やコミュニティ、開発者ツール、ライブラリを開発したい方向けです。あなたが Rust 言語に貢献されることを心よりお待ちしております。

スピードと安定性に価値を見出す方

Rust は、スピードと安定性を言語に渇望する方向けです。ここでいうスピードとは、Rust コードの実行速度とプログラムを書くスピードのことです。Rust コンパイラのチェックにより、機能の追加とリファクタリングを通して安定性を保証してくれます。これはこのようなチェックがない言語の脆いレガシーコードとは対照的で、その場合開発者はしばしば、変更するのを恐れてしまいます。ゼロコスト抽象化を志向し、手で書いたコードと同等の速度を誇る低レベルコードにコンパイルされる高レベル機能により、Rust は安全なコードを高速なコードにもしようと努力しています。

Rust 言語は他の多くのユーザのサポートも望んでいます; ここで名前を出した方は、ただの最大の出資者の一部です。総合すると、Rust の最大の野望は、プログラマが数十年間受け入れてきた代償を、安全性**と**生産性、スピード**と**エルゴノミクスを提供することで排除することです。Rust を試してみて、その選択が自分に合っているか確かめてください。

この本は誰のためのものなの

この本は、あなたが他のプログラミング言語でコードを書いたことがあることを想定していますが、具体的にどの言語かという想定はしません。私たちは、幅広い分野のプログラミング背景からの人にとってこの資料を広くアクセスできるようにしようとしてきました。プログラミングとはなん**なのか**やそれについて考える方法について多くを語るつもりはありません。もし、完全なプログラミング初心者であれば、プログラミング入門を特に行う本を読むことでよりよく役に立つでしょう。

この本の使い方

一般的に、この本は、順番に読み進めていくことを前提にしています。後の章は、前の章の概念の上に成り立ち、前の章では、特定の話題にさほど深入りしない可能性がありますが、後ほどの章で同じ話題を再検討するでしょう。

この本には 2 種類の章があるとわかるでしょう: 概念の章とプロジェクトの章です。概念の章では、Rust の一面を学ぶでしょう。プロジェクトの章では、それまでに学んだことを適用して一緒に小さなプログラムを構築します。2、12、20 章がプロジェクトの章です。つまり、残りは概念の章です。

第 1 章は Rust のインストール方法、“Hello, world!” プログラムの書き方、Rust のパッケージマネージャ兼、ビルドツールの Cargo の使用方法を説明します。第 2 章は、数当てゲームを作りながら、実際に Rust でのプログラミングをやってもらう導入です。ここでは概念をざっくりと講義し、後ほどの章で追加の詳細を提供します。今すぐ Rust の世界に飛び込みたいなら、第 2 章こそがそのためのものです。第 3 章は他のプログラミング言語の機能に似た Rust の機能を講義し、第 4 章では Rust の所有権システムについて学びます。あなたが次に進む前に全ての詳細を学ぶことを好む特別に几帳面な学習者なら、第 2 章を飛ばして真っ先に第 3 章に行き、学んだ詳細を適用するプロジェクトに取り組みたくなった時に第 2 章に戻りたくなる可能性があります。

第 5 章は、構造体とメソッドについて議論し、第 6 章は enum、`match` 式、`if let` 制御フロー構文を講義します。構造体と enum を使用して Rust において独自の型を作成します。

第 7 章では、Rust のモジュールシステムと自分のコードとその公開された API(Application Programming Interface) を体系化するプライバシー規則について学びます。第 8 章では、ベクタ、文字列、ハッシュマップなどの標準ライブラリが提供する一般的なコレクションデータ構造の一部を議論します。第 9 章では、Rust のエラー処理哲学とテクニックを探究します。

第 10 章ではジェネリクス、トレイト、ライフタイムについて深入りし、これらは複数の型に適用されるコードを定義する力をくれます。第 11 章は、完全にテストに関してで、Rust の安全性保証があってさえ、プログラムのロジックが正しいことを保証するために、必要になります。第 12 章では、ファイル内のテキストを検索する `grep` コマンドラインツールの一部の機能を自身で構築します。このために、以前の章で議論した多くの概念を使用します。

第 13 章はクロージャとイテレータを探究します。これらは、関数型プログラミング言語由来の

Rust の機能です。第 14 章では、Cargo をより詳しく調査し、他人と自分のライブラリを共有する最善の策について語ります。第 15 章では、標準ライブラリが提供するスマートポインタとその機能を可能にするトレイトを議論します。

第 16 章では、並行プログラミングの異なるモデルを見ていき、Rust が恐れなしに複数のスレッドでプログラムする手助けをする方法を語ります。第 17 章では、馴染み深い可能性のあるオブジェクト指向プログラミングの原則と Rust のイディオムがどう比較されるかに目を向けます。

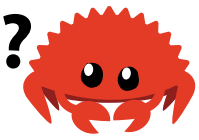
第 18 章は、パターンとパターンマッチングのリファレンスであり、これらは Rust プログラムを通して、考えを表現する強力な方法になります。第 19 章は、unsafe Rust やマクロ、ライフタイム、トレイト、型、関数、クロージャの詳細を含む、興味のある高度な話題のスモーガスボード (訳注 : 日本でいうバイキングのこと) を含みます。

第 20 章では、低レベルなマルチスレッドの Web サーバを実装するプロジェクトを完成させます！

最後に、言語についての有用な情報をよりリファレンスのような形式で含む付録があります。付録 A は Rust のキーワードを講義し、付録 B は、Rust の演算子と記号、付録 C は、標準ライブラリが提供する導出可能なトレイト、付録 D はいくつか便利な開発ツールを講義し、付録 E では Rust のエディションについて説明します。付録 F ではこの本の翻訳を見つけることができ、付録 G では Rust の作られ方、そして nightly Rust とは何かについて講義します。

この本を読む間違った方法なんてありません: 飛ばしたければ、どうぞご自由に！ 混乱したら、前の章に戻らなければならない可能性もあります。ですが、自分に合った方法でどうぞ。

Rust を学ぶ過程で重要な部分は、コンパイラが表示するエラーメッセージを読む方法を学ぶことです: それは動くコードへと導いてくれます。そのため、各場面でコンパイラが表示するエラーメッセージとともに、コンパイルできない例を多く提供します。適当に例を選んで走らせたら、コンパイルできないかもしれないことを知ってください！ 周りのテキストを読んで実行しようとしている例がエラーになることを意図しているのか確認することを確かめてください。フェリスもコードが動作するとは意図されていないコードを見分けるのを手助けしてくれます:

Ferris	Meaning
	このコードはコンパイルできません！
	このコードはパニックします！

Ferris	Meaning
	このコードは求められている振る舞いをしません。

ほとんどの場合、コンパイルできないあらゆるコードの正しいバージョンへと導きます。

ソースコード

この本が生成されるソースファイルは、GitHub で見つかります。

訳注: 日本語版はこちらです。

目次

The Rust Programming Language 日本語版 i

まえがき ii

はじめに iii
Rust は誰のためのものなの . . . iii
この本は誰のためのものなの . . . v
この本の使い方 . . . v
ソースコード . . . vii

第 1 章 事始め 1
1.1 インストール . . . 1
1.2 Hello, World! . . . 4
1.3 Hello, Cargo! . . . 8
1.4 まとめ . . . 13

第 2 章 数当てゲームのプログラミング 14
2.1 新規プロジェクトの立ち上げ . . . 14
2.2 予想を処理する . . . 15
2.3 秘密の数字を生成する . . . 22
2.4 予想と秘密の数字を比較する . . . 27
2.5 ループで複数回の予想を可能にする . . . 31
2.6 まとめ . . . 37

第 3 章 一般的なプログラミングの概念 38
3.1 変数と可変性 . . . 38
3.2 データ型 . . . 43

3.3 関数 52
3.4 コメント 58
3.5 制御フロー 59
3.6 まとめ 69

第 4 章 所有権を理解する 71
4.1 所有権とは？ 71
4.2 参照と借用 84
4.3 スライス型 93
4.4 まとめ 103

第 5 章 構造体を使用して関係のあるデータを構造化する 104
5.1 構造体を定義し、インスタンス化する 104
5.2 構造体を使ったプログラム例 113
5.3 メソッド記法 119
5.4 まとめ 127

第 6 章 Enum とパターンマッチング 128
6.1 Enum を定義する 128
6.2 match 制御フロー構造 137
6.3 if let で簡潔な制御フロー 145
6.4 まとめ 147

第 7 章 肥大化していくプロジェクトをパッケージ、クレート、モジュールを利用して管理する 148
7.1 パッケージとクレート 149
7.2 モジュールを定義して、スコープとプライバシーを制御する 150
7.3 モジュールツリーの要素を示すためのパス 154
7.4 use キーワードでパスをスコープに持ち込む 163
7.5 モジュールを複数のファイルに分割する 172
7.6 まとめ 174

第 8 章 一般的なコレクション 176
8.1 ベクタで値のリストを保持する 176
8.2 文字列で UTF-8 でエンコードされたテキストを保持する 183
8.3 キーとそれに紐づいた値をハッシュマップに格納する 192
8.4 まとめ 198

第 9 章 エラー処理 199
9.1 panic!で回復不能なエラー 199

9.2 Result で回復可能なエラー 203
9.3 panic!すべきかするまいか 215
9.4 まとめ 220

第 10 章 ジェネリック型、トレイト、ライフタイム 221
10.1 関数を抽出することで重複を取り除く 222
10.2 ジェネリックなデータ型 224
10.3 トレイト: 共通の振る舞いを定義する 234
10.4 ライフタイムで参照を検証する 246
10.5 ジェネリックな型引数、トレイト境界、ライフタイムを一度に 262
10.6 まとめ 263

第 11 章 自動テストを書く 264
11.1 テストの記述法 265
11.2 テストの実行のされ方を制御する 285
11.3 テストの体系化 292
11.4 まとめ 299

第 12 章 入出力プロジェクト: コマンドラインプログラムを構築する 300
12.1 コマンドライン引数を受け付ける 301
12.2 ファイルを読み込む 304
12.3 リファクタリングしてモジュール性とエラー処理を向上させる 306
12.4 テスト駆動開発でライブラリの機能を開発する 323
12.5 環境変数を取り扱う 334
12.6 標準出力ではなく標準エラーにエラーメッセージを書き込む 347
12.7 まとめ 349

第 13 章 関数型言語の機能: イテレータとクロージャ 350
13.1 クロージャ: 環境をキャプチャする匿名関数 350
13.2 一連の要素をイテレータで処理する 363
13.3 入出力プロジェクトを改善する 370
13.4 パフォーマンス比較: ループ VS イテレータ 383
13.5 まとめ 384

第 14 章 Cargo と Crates.io についてより詳しく 385
14.1 リリースプロファイルでビルドをカスタマイズする 385
14.2 Crates.io にクレートを公開する 387
14.3 Cargo のワークスペース 399
14.4 cargo install でバイナリをインストールする 405

14.5 独自のコマンドで Cargo を拡張する 406
14.6 まとめ 406

第 15 章 スマートポインタ 407
15.1 ヒープのデータを指す Box<T>を使用する 408
15.2 Deref トレイトでスマートポインタを普通の参照のように扱う 415
15.3 Drop トレイトで片付け時にコードを走らせる 423
15.4 Rc<T>は、参照カウント方式のスマートポインタ 427
15.5 RefCell<T>と内部可変性パターン 432
15.6 循環参照は、メモリをリークすることもある 445
15.7 まとめ 455

第 16 章 恐れるな！ 並行性 456
16.1 スレッドを使用してコードを同時に走らせる 457
16.2 メッセージ受け渡しを使ってスレッド間でデータを転送する 464
16.3 状態共有並行性 471
16.4 Sync と Send トレイトで拡張可能な並行性 479
16.5 まとめ 480

第 17 章 Rust のオブジェクト指向プログラミング機能 482
17.1 オブジェクト指向言語の特徴 482
17.2 トレイトオブジェクトで異なる型の値を許容する 486
17.3 オブジェクト指向デザインパターンを実装する 494
17.4 まとめ 511

第 18 章 パターンとマッチング 513
18.1 パターンが使用されることのある箇所全部 513
18.2 論駁可能性: パターンが合致しないかどうか 519
18.3 パターン記法 521
18.4 まとめ 538

第 19 章 高度な機能 539
19.1 Unsafe Rust 539
19.2 高度なトレイト 549
19.3 高度な型 562
19.4 高度な関数とクロージャ 568
19.5 マクロ 571
19.6 まとめ 582

第 20 章　最後のプロジェクト: マルチスレッドの Web サーバを構築する 583
20.1　シングルスレッドの Web サーバを構築する 584
20.2　シングルスレッドサーバをマルチスレッド化する 595
20.3　正常なシャットダウンと片付け 618
20.4　まとめ 629

付録 630
付録 A: キーワード 630
付録 B: 演算子と記号 633
付録 C: 導出可能なトレイト 639
付録 D - 便利な開発ツール 642
付録 E：エディション 646
付録 F: 本の翻訳 647
付録 G: Rust の作られ方と “Nightly Rust” 648

第 1 章

事始め

Rust の旅を始めましょう！ 学ぶべきことはたくさんありますが、いかなる旅もどこかから始まります。この章では、以下のことを説明します:

- Rust を Linux、macOS、Windows にインストールする
- `Hello, world!` と表示するプログラムを書く
- `cargo` という Rust のパッケージマネージャ兼ビルドシステムを使用する

1.1 インストール

最初の手順は、Rust をインストールすることです。Rust は、Rust のバージョンと関連するツールを管理する、`rustup` というコマンドラインツールを使用してダウンロードします。ダウンロードには、インターネットへの接続が必要になります。

> 注釈: なんらかの理由で `rustup` を使用したくない場合、Other Rust Installation Methods ページで、他の選択肢をご覧になってください。

以下の手順で最新の安定版の Rust コンパイラをインストールします。Rust は安定性 (stability) を保証しているので、現在この本の例でコンパイルできるものは、新しいバージョンになってもコンパイルでき続けることが保証されます。出力は、バージョンによって多少異なる可能性があります。Rust は頻繁にエラーメッセージと警告を改善しているからです。言い換えると、どんな新しいバージョンでもこの手順に従ってインストールした安定版なら、この本の内容で想定通りに動くはずです。

1.1.1　コマンドラインの記法

この章及び、本を通して、端末で使用するなんらかのコマンドを示すことがあります。読者が入力するべき行は、全て $ で始まります。ただし、読者が $ 文字を入力する必要はありません; これは各コマンドの開始を示すために表示しているコマンドラインプロンプトです。$ で始まらない行は、典型的には直前のコマンドの出力を示します。また、PowerShell 限定の例には、$ ではなく、> を使用します。

1.1.2　Linux と macOS に `rustup` をインストールする

Linux か macOS を使用しているなら、端末（ターミナル）を開き、以下のコマンドを入力してください:

```
$ curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

このコマンドはスクリプトをダウンロードし、`rustup` ツールのインストールを開始し、Rust の最新の安定版をインストールします。パスワードを求められる可能性があります。インストールがうまく行けば、以下の行が出現するでしょう:

```
Rust is installed now. Great!
```

リンカも必要になるでしょう。リンカは、コンパイルされた出力をひとつのファイルに合体させるために Rust が使用するプログラムです。リンカが既にインストールされている可能性は高いでしょう。リンカエラーが発生したときは、C コンパイラは典型的にリンカを含んでいるでしょうから、C コンパイラをインストールすべきです。一般的な Rust パッケージの中には、C コードに依存し、C コンパイラが必要になるものもあるので、この理由からも C コンパイラは有用です。

macOS では、以下を実行することで C コンパイラが手に入ります:

```
$ xcode-select --install
```

Linux ユーザは、通常はディストリビューションのドキュメントに従って、GCC または Clang をインストールするべきです。例えば Ubuntu を使用している場合は、`build-essential` パッケージをインストールすれば大丈夫です。

1.1.3　Windows で `rustup` をインストールする

Windows では、https://www.rust-lang.org/tools/install に行き、手順に従って Rust をインストールしてください。インストールの途中で、Visual Studio 2013 以降用の MSVC ビルドツールも必要になるという旨のメッセージが出るでしょう。

ビルドツールを取得するには、Visual Studio 2022 をインストールする必要があるでしょう。どのワークロード (workloads) をインストールするかと質問されたときは、以下を含めてください:

- 「C++ によるデスクトップ開発」(“Desktop Development with C++”)
- Windows 10 または 11 の SDK
- 英語の言語パック (English language pack) コンポーネント (お好みで他の任意の言語パックも)

  訳注：Windows の言語を日本語にしている場合は言語パックのところで「日本語」が選択されており、そのままの設定でインストールしても基本的に問題ないはずです。しかし、サードパーティーのツールやライブラリの中には英語の言語パックを必要とするものがあるため、「日本語」に加えて「英語」も選択することをお勧めします。

これ以降、**cmd.exe** と PowerShell の両方で動くコマンドを使用します。特段の違いがあったら、どちらを使用すべきか説明します。

1.1.4　トラブルシューティング

Rust が正常にインストールされているか確かめるには、シェルを開いて以下の行を入力してください:

```
$ rustc --version
```

リリース済みの最新の安定版のバージョンナンバー、コミットハッシュ、コミット日が以下の形式で表示されるはずです。

```
rustc x.y.z (abcabcabc yyyy-mm-dd)
```

この情報が見られたなら、Rust のインストールに成功しています！ この情報が出ない場合は、次のようにして Rust が `%PATH%` システム環境変数にあることを確認してください。

Windows CMD では:

```
> echo %PATH%
```

PowerShell では:

```
> echo $env:Path
```

Linux および macOS では:

```
$ echo $PATH
```

これらが全て正常であるのに、それでも Rust がうまく動かないなら、助力を得られる場所はたく

さんあります。他の Rustacean（Rust ユーザが自分たちのことを呼ぶ、冗談めいたニックネーム）たちと交流する方法をコミュニティページで探してください。

> 訳注 1：Rustacean について、いらないかもしれない補足です。公式 Twitter 曰く、Rustacean は crustaceans（甲殻類）から来ているそうです。そのため、Rust のマスコットは（非公式らしいですが）カニ。上の会話で C の欠点を削ぎ落としているから c を省いてるの？ みたいなことを聞いていますが、違うそうです。検索したら、堅牢性が高いから甲殻類という意見もありますが、真偽は不明です。明日使えるかもしれないトリビアでした。
>
> 訳注 2：上にあるコミュニティページはどれも英語話者のコミュニティへのリンク集です。日本語話者のためのコミュニティが Zulip rust-lang-jp にあり、こちらでも Rustacean たちが活発に議論をしています。公式 Discord 同様、初心者向けの#beginners チャンネルが存在するので、気軽に質問してみてください。

1.1.5　更新及びアンインストール

`rustup` 経由で Rust がインストールされたなら、新しくリリースされた版へ更新するのは簡単です。シェルから以下の更新スクリプトを実行してください:

```
$ rustup update
```

Rust と `rustup` をアンインストールするには、シェルから以下のアンインストールスクリプトを実行してください:

```
$ rustup self uninstall
```

1.1.6　ローカルのドキュメンテーション

インストールされた Rust には、オフラインでドキュメンテーションを閲覧できるように、ドキュメンテーションのローカルコピーが含まれています。ブラウザでローカルのドキュメンテーションを開くには、`rustup doc` を実行してください。

標準ライブラリにより提供される型や関数がなんなのかや、それをどう使えば良いのかがよくわからないときは、いつでも API のドキュメンテーションを検索してみてください！

1.2　Hello, World!

Rust をインストールしたので、最初の Rust プログラムを書きましょう。新しい言語を学ぶ際に、`Hello, world!` というテキストを画面に出力する小さなプログラムを書くことは伝統的なことなので、ここでも同じようにしましょう！

注釈: この本は、コマンドラインに基礎的な馴染みがあることを前提にしています。Rust は、編集やツール、どこにコードがあるかについて特定の要求をしないので、コマンドラインではなく IDE を使用することを好むのなら、どうぞご自由にお気に入りの IDE を使用してください。今では、多くの IDE がなんらかの形で Rust をサポートしています; 詳しくは、IDE のドキュメンテーションをご覧ください。Rust チームは `rust-analyzer` を介して優れた IDE サポートを可能にすることに注力しています。詳しくは付録 D をご覧ください。

1.2.1　プロジェクトのディレクトリを作成する

Rust コードを格納するディレクトリを作ることから始めましょう。Rust にとって、コードがどこにあるかは問題ではありませんが、この本の練習とプロジェクトのために、ホームディレクトリに **projects** ディレクトリを作成してプロジェクトを全てそこに保管することを推奨します。

端末を開いて以下のコマンドを入力し、**projects** ディレクトリと、**projects** ディレクトリ内に「Hello, world!」プロジェクトのディレクトリを作成してください。

Linux、macOS、そして Windows 上の PowerShell なら、こう入力してください:

```
$ mkdir ~/projects
$ cd ~/projects
$ mkdir hello_world
$ cd hello_world
```

Windows の cmd なら、こう:

```
> mkdir "%USERPROFILE%\projects"
> cd /d "%USERPROFILE%\projects"
> mkdir hello_world
> cd hello_world
```

1.2.2　Rust プログラムを書いて走らせる

次にソースファイルを作り、**main.rs** というファイル名にしてください。Rust のファイルは常に**.rs** という拡張子で終わります。ファイル名に 2 単語以上使っているなら、アンダースコアで区切るのが規約です。例えば、**helloworld.rs** ではなく、**hello_world.rs** を使用してください。

さて、作ったばかりの **main.rs** ファイルを開き、リスト 1-1 のコードを入力してください。

ファイル名: main.rs

```
fn main() {
    // 世界よ、こんにちは
    println!("Hello, world!");
}
```

リスト 1-1: `Hello, world!` と出力するプログラム

ファイルを保存し、**~/projects/hello_world** ディレクトリの端末ウィンドウに戻ってください。Linux か macOS なら、以下のコマンドを打ってファイルをコンパイルし、実行してください:

```
$ rustc main.rs
$ ./main
Hello, world!
```

Windows なら、`./main` の代わりに `.\main.exe` と打ちます:

```
> rustc main.rs
> .\main.exe
Hello, world!
```

OS に関わらず、`Hello, world!` という文字列が端末に出力されるはずです。この出力が見れないなら、インストールの節の「トラブルシューティング」の部分に立ち戻って、助けを得る方法を参照してください。

`Hello, world!` が確かに出力されたら、おめでとうございます！ 正式に Rust プログラムを書きました。Rust プログラマになったのです！ ようこそ！

1.2.3　Rust プログラムの解剖

この「Hello, world!」プログラムを詳しく再確認しましょう。こちらがパズルの最初のピースです:

```
fn main() {

}
```

これらの行は `main` という名前の関数を定義しています。`main` 関数は特別です: 常に全ての実行可能な Rust プログラムで走る最初のコードになります。ここで、1 行目は、引数がなく何も返さない `main` という関数を宣言しています。引数があるなら、かっこ (`()`) の内部に入ります。

関数の本体は `{}` に包まれます。Rust では、全ての関数本体の周りに波括弧が必要になります。スペースを 1 つあけて、開き波括弧を関数宣言と同じ行に配置するのがいいスタイルです。

> 注釈: 複数の Rust プロジェクトに渡って標準的なスタイルにこだわりたいなら、`rustfmt` を使うことでコードを決まったスタイルに整形できるでしょう（ `rustfmt` の詳細は付録 D で）。Rust チームは、`rustc` のように標準的な Rust の配布にこのツールを含んでいるため、既にコンピューターにインストールされているはずです！

`main` 関数の本体は、こんなコードを抱えています:

```
    println!("Hello, world!");
```

この行が、この小さなプログラムの全作業をしています: テキストを画面に出力するのです。ここで気付くべき重要な詳細が 4 つあります。

まず、Rust のスタイルは、タブではなく、4 スペースでインデントするということです。

2 番目に `println!` は Rust のマクロを呼び出すということです。代わりに関数を呼んでいたら、`println` (！なし) と入力されているでしょう。Rust のマクロについて詳しくは、第 19 章で議論します。とりあえず、`!` を使用すると、普通の関数ではなくマクロを呼んでいるのだということと、マクロは関数と同じルールには必ずしも従わないということを知っておくだけでいいでしょう。

3 番目に、`"Hello, world!"` 文字列が見えます。この文字列を引数として `println!` に渡し、この文字列が画面に表示されているのです。

4 番目にこの行をセミコロン (`;`) で終え、この式が終わり、次の式の準備ができていると示唆していることです。Rust コードのほとんどの行は、セミコロンで終わります。

1.2.4　コンパイルと実行は個別のステップ

新しく作成したプログラムをちょうど実行したので、その途中の手順を調査しましょう。

Rust プログラムを実行する前に、以下のように、`rustc` コマンドを入力し、ソースファイルの名前を渡すことで、Rust コンパイラを使用してコンパイルしなければなりません。

```
$ rustc main.rs
```

あなたに C や C++ の背景があるなら、これは `gcc` や `clang` と似ていると気付くでしょう。コンパイルに成功後、Rust はバイナリの実行可能ファイルを出力します。

Linux、macOS、Windows の PowerShell なら、シェルで `ls` コマンドを入力することで実行可能ファイルを見られます:

```
$ ls
main  main.rs
```

Linux と macOS では、2 つのファイルが見えるでしょう。Windows の PowerShell では、CMD を使ったときに見ることになるのと同じ 3 つのファイルが見えるでしょう。Windows の CMD なら、以下のように入力するでしょう:

```
> dir /B %= the /B option says to only show the file names =%
         %= /Bオプションは、ファイル名だけを表示することを宣言する =%
main.exe
main.pdb
main.rs
```

これは、**.rs** 拡張子のソースコードファイル、実行可能ファイル (Windows なら **main.exe**、他のプラットフォームでは、**main**)、そして、Windows を使用しているなら、**.pdb** 拡張子のデバッグ情報を含むファイルを表示します。ここから、**main** か **main.exe** を走らせます。このように:

```
$ ./main # or .\main.exe on Windows
         # または、Widnowsなら.\main.exe
```

main.rs が Hello, world!プログラムなら、この行は `Hello, world!` と端末に出力します。

Ruby や Python、JavaScript などの動的言語により造詣が深いなら、プログラムのコンパイルと実行を個別の手順で行うことに慣れていない可能性があります。Rust は **AOT コンパイル** (ahead-of-time; 訳注：予め) 言語です。つまり、プログラムをコンパイルし、実行可能ファイルを誰かにあげ、あげた人が Rust をインストールしていなくても実行できるわけです。誰かに**.rb**、**.py**、**.js** ファイルをあげたら、それぞれ Ruby、Python、JavaScript の処理系がインストールされている必要があります。ですが、そのような言語では、プログラムをコンパイルし実行するには、1 コマンドしか必要ないのです。全ては言語設計においてトレードオフなのです。

簡単なプログラムなら `rustc` でコンパイルするだけでも十分ですが、プロジェクトが肥大化してくると、オプションを全て管理し、自分のコードを簡単に共有したくなるでしょう。次は、Cargo ツールを紹介します。これは、現実世界の Rust プログラムを書く手助けをしてくれるでしょう。

1.3　Hello, Cargo!

Cargo は Rust のビルドシステム兼パッケージマネージャです。ほとんどの Rustacean はこのツールを使って Rust プロジェクトを管理しています。なぜなら、Cargo は多くの仕事、たとえばコードのビルド、コードが依存するライブラリのダウンロード、それらのライブラリのビルドなどを扱ってくれるからです。（コードが必要とするライブラリのことを**依存**（dependencies）と呼びます）

いままでに書いたようなごく単純な Rust プログラムには依存がありません。「Hello, world!」プロジェクトを Cargo でビルドしても、Cargo の中のコードをビルドする部分しか使わないでしょう。より複雑な Rust プログラムを書くようになると依存を追加することになりますが、Cargo を使ってプロジェクトを開始したなら、依存の追加もずっと簡単になります。

Rust プロジェクトの大多数が Cargo を使用しているので、これ以降、この本では、あなたも Cargo を使用していると想定します。もし「インストール」節で紹介した公式のインストーラを使用したなら、Cargo は Rust と共にインストールされています。Rust を他の方法でインストールした場合は、以下のコマンドをターミナルに入れて、Cargo がインストールされているか確認してください。

```
$ cargo --version
```

バージョンナンバーが表示されたならインストールされています！ `command not found` などのエラーが表示された場合は、自分がインストールした方法についてのドキュメントを参照して、Cargo を個別にインストールする方法を調べてください。

1.3.1　Cargo でプロジェクトを作成する

Cargo を使って新しいプロジェクトを作成し、元の「Hello, world!」プロジェクトとの違いを見ていきましょう。**projects** ディレクトリ（または自分がコードを保存すると決めた場所）に戻ってください。それから、OS に関係なく、以下を実行してください。

```
$ cargo new hello_cargo
$ cd hello_cargo
```

最初のコマンドは **hello_cargo** という名の新しいディレクトリとプロジェクトを作成します。プロジェクトを **hello_cargo** と名付けたので、Cargo はそれに関連するいくつかのファイルを同名のディレクトリに作成します。

hello_cargo ディレクトリに行き、ファイルの一覧を取得してください。Cargo が 2 つのファイルと 1 つのディレクトリを生成してくれたことがわかるでしょう。**Cargo.toml** ファイルと **src** ディレクトリがあり、**src** の中には **main.rs** ファイルがあります。

また、**.gitignore** ファイルと共に新しい Git リポジトリも初期化されています。もし、すでに存在する Git リポジトリの中で `cargo new` を実行したなら、Git 関連のファイルは作られません。`cargo new --vcs=git` とすることで、この振る舞いを変更できます。

> 補足：Git は一般的なバージョン管理システムです。`cargo new` コマンドに `--vcs` フラグを与えることで、別のバージョン管理システムを使用したり、何も使用しないようにもできます。利用可能なオプションを確認するには `cargo new --help` を実行します。

お気に入りのテキストエディタで **Cargo.toml** を開いてください。リスト 1-2 のコードのようになっているはずです。

ファイル名：Cargo.toml

```
[package]
name = "hello_cargo"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/
    reference/manifest.html

[dependencies]
```

リスト 1-2：`cargo new` で生成された **Cargo.toml** の内容

このファイルは **TOML**（**Tom's Obvious, Minimal Language**、トムの明確な最小限の言語）形式で、Cargo の設定フォーマットです。

最初の行の `[package]` はセクションヘッダーで、それ以降の文がパッケージを設定することを示し

ます。このファイルに情報を追加してく中で、他のセクションも追加していくことになります。

次の 3 行は Cargo がプログラムをコンパイルするのに必要となる設定情報を指定します。ここでは、名前、バージョン、使用する Rust のエディションを指定しています。`edition` キーについては付録 E で説明されています。

最後の行の `[dependencies]` は、プロジェクトの依存を列挙するためのセクションの始まりです。Rust ではコードのパッケージのことを**クレート**と呼びます。このプロジェクトでは他のクレートは必要ありませんが、第 2 章の最初のプロジェクトでは必要になるので、そのときにこの依存セクションを使用します。

では、**src/main.rs** を開いて見てみましょう。

ファイル名: src/main.rs

```
fn main() {
    println!("Hello, world!");
}
```

Cargo はリスト 1-1 で書いたような「Hello, world!」プログラムを生成してくれています。これまでのところ、私たちのプロジェクトと Cargo が生成したプロジェクトの違いは、Cargo がコードを **src** ディレクトリに配置したことと、最上位のディレクトリに **Cargo.toml** 設定ファイルがあることです。

Cargo はソースファイルが **src** ディレクトリにあることを期待します。プロジェクトの最上位のディレクトリは、README ファイル、ライセンス情報、設定ファイル、その他のコードに関係しないものだけを置きます。Cargo を使うとプロジェクトを整理することができます。すべてのものに決まった場所があり、すべてがその場所にあるのです。

「Hello, world!」プロジェクトのように Cargo を使用しないプロジェクトを開始したときでも、Cargo を使用するプロジェクトへと変換できます。プロジェクトのコードを **src** ディレクトリに移動し、適切な **Cargo.toml** ファイルを作成すればいいのです。

1.3.2　Cargo プロジェクトをビルドし、実行する

では「Hello, world!」プログラムを Cargo でビルドして実行すると、何が違うのかを見てみましょう！ **hello_cargo** ディレクトリから以下のコマンドを入力して、プロジェクトをビルドします。

```
$ cargo build
   Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 2.85 secs
```

このコマンドは実行ファイルを現在のディレクトリではなく、**target/debug/hello_cargo**（Windows では **target/debug/hello_cargo.exe**）に作成します。デフォルトのビルドはデバッグビルドなので、Cargo はバイナリを **debug** という名前のディレクトリの中に入れます。以下のコ

マンドで実行ファイルを実行できます。

```
$ ./target/debug/hello_cargo # or .\target\debug\hello_cargo.exe on Windows
                             # Windowsでは .\target\debug\hello_cargo.exe
Hello, world!
```

すべてがうまくいけば、ターミナルに Hello, world! と表示されるはずです。cargo build を初めて実行したとき、Cargo は最上位に **Cargo.lock** という新しいファイルを作成します。このファイルはプロジェクト内の依存関係の正確なバージョンを記録しています。このプロジェクトには依存がないので、このファイルの中は少しまばらです。このファイルは手動で変更する必要はありません。Cargo がその内容を管理してくれます。

先ほどは cargo build でプロジェクトをビルドし、./target/debug/hello_cargo で実行しました。cargo run を使うと、コードのコンパイルから、できた実行ファイルの実行までの全体を一つのコマンドで行えます。

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/hello_cargo`
Hello, world!
```

cargo build を実行してから、バイナリへのパス全体を使って実行する、という手順をいちいち踏むより、cargo run を使う方が便利なので、ほとんどの開発者は cargo run を使います。

今回は Cargo が hello_cargo をコンパイルしていることを示す出力がないことに注目してください。Cargo はファイルが変更されていないことに気づいたので、再ビルドせずに単にバイナリを実行したのです。もしソースコードを変更していたら、Cargo は実行前にプロジェクトを再ビルドし、以下のような出力が表示されたことでしょう。

```
$ cargo run
   Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 0.33 secs
     Running `target/debug/hello_cargo`
Hello, world!
```

Cargo は cargo check というコマンドも提供しています。このコマンドはコードがコンパイルできるか素早くチェックしますが、実行ファイルは生成しません。

```
$ cargo check
   Checking hello_cargo v0.1.0 (file:///projects/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 0.32 secs
```

なぜ実行可能ファイルが欲しくないのでしょうか？ cargo check は実行ファイルを生成するステップを省くことができるので、多くの場合、cargo build よりもずっと高速です。もし、あなたがコードを書きながら継続的にチェックするのなら、cargo check を使えば、プロジェクトがまだコンパイルできるか確認するプロセスを高速化できます！ そのため多くの Rustacean はプログラムを書きな

がら定期的に `cargo check` を実行し、コンパイルできるか確かめます。そして、実行ファイルを使う準備ができたときに `cargo build` を走らせるのです。

ここまでに Cargo について学んだことをおさらいしておきましょう。

- `cargo new` を使ってプロジェクトを作成できる
- `cargo build` を使ってプロジェクトをビルドできる
- `cargo run` を使うとプロジェクトのビルドと実行を 1 ステップで行える
- `cargo check` を使うとバイナリを生成せずにプロジェクトをビルドして、エラーがないか確認できる
- Cargo は、ビルドの成果物をコードと同じディレクトリに保存するのではなく、**target/debug** ディレクトリに格納する

Cargo を使用するもう一つの利点は、どの OS で作業していてもコマンドが同じであることです。そのため、これ以降は Linux や macOS 向けの手順と、Windows 向けの手順を分けて説明することはありません。

1.3.3　リリースに向けたビルド

プロジェクトが最終的にリリースできるようになったら、`cargo build --release` を使い、最適化した状態でコンパイルできます。このコマンドは実行ファイルを、**target/debug** ではなく、**target/release** に作成します。最適化によって Rust コードの実行速度が上がりますが、それを有効にすることでプログラムのコンパイルにかかる時間が長くなります。このため二つの異なるプロファイルがあるのです。一つは開発用で、素早く頻繁に再ビルドしたいときのもの。もう一つはユーザに渡す最終的なプログラムをビルドするためのもので、繰り返し再ビルドすることはなく、可能な限り高速に動作するようにします。コードの実行時間をベンチマークするなら、必ず `cargo build --release` を実行し、**target/release** の実行ファイルを使ってベンチマークを取ってください。

1.3.4　習慣としての Cargo

単純なプロジェクトでは、Cargo は単に `rustc` を使うことに対してあまり多くの価値を生みません。しかし、プログラムが複雑になるにつれて、その価値を証明することになるでしょう。プログラムが複数のファイルに分かれるほど大きくなったり、依存が必要になってくると、Cargo にビルドを調整させるほうがずっと簡単です。

`hello_cargo` プロジェクトは単純ではありますが、Rust のキャリアを通じて使うことになる本物のツールの多くを使用しています。実際、既存のどんなプロジェクトで作業するときも、以下のコマンドを使えば、Git でコードをチェックアウトし、そのプロジェクトのディレクトリに移動し、ビルドすることができます。

```
$ git clone example.org/someproject
```

```
$ cd someproject
$ cargo build
```

Cargo の詳細については、ドキュメントを参照してください。

1.4　まとめ

既に Rust の旅の素晴らしいスタートを切っています！ この章では以下を行う方法について学びました。

- `rustup` で最新の安定版の Rust をインストールする
- 新しい Rust のバージョンに更新する
- ローカルにインストールされたドキュメントを開く
- 「Hello, world!」プログラムを書き、`rustc` を直接使って実行する
- Cargo における習慣に従った新しいプロジェクトを作成し、実行する

いまは、より中身のあるプログラムを構築し、Rust コードの読み書きに慣れるのに良いタイミングでしょう。そこで第 2 章では、数当てゲームプログラムを構築します。もし、一般的なプログラミングの概念が Rust でどう実現されるか学ぶことから始めたいのであれば、第 3 章を読んで、それから第 2 章に戻ってください。

第 2 章

数当てゲームのプログラミング

ハンズオン形式のプロジェクトに一緒に取り組むことで、Rust の世界に飛び込んでみましょう！

この章では Rust の一般的な概念を、実際のプログラムでの使い方を示しながら紹介します。`let`、`match`、メソッド、関連関数、外部クレートなどについて学びます！　これらについての詳細は後続の章で取り上げますので、この章では基本的なところだけを練習します。

プログラミング初心者向けの定番問題である「数当てゲーム」を実装してみましょう。これは次のように動作します。プログラムは 1 から 100 までのランダムな整数を生成します。そして、プレーヤーに予想（した数字）を入力するように促します。予想が入力されると、プログラムはその予想が小さすぎるか大きすぎるかを表示します。予想が当たっているなら、お祝いのメッセージを表示し、ゲームを終了します。

2.1　新規プロジェクトの立ち上げ

新しいプロジェクトを立ち上げましょう。第 1 章で作成した **projects** ディレクトリに移動し、以下のように Cargo を使って新規プロジェクトを作成します。

```
$ cargo new guessing_game
$ cd guessing_game
```

最初のコマンド `cargo new` は、第 1 引数としてプロジェクト名 (`guessing_game`) を取ります。2 番目のコマンドは新規プロジェクトのディレクトリに移動します。

生成された **Cargo.toml** ファイルを見てみましょう。

ファイル名：Cargo.toml

```
[package]
name = "guessing_game"
version = "0.1.0"
```

```
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/
    reference/manifest.html

[dependencies]
```

第 1 章で見たように cargo new は「Hello, world!」プログラムを生成してくれます。**src/main.rs** ファイルをチェックしてみましょう。

ファイル名：src/main.rs

```
fn main() {
    println!("Hello, world!");
}
```

さて、cargo run コマンドを使って、この「Hello, world!」プログラムのコンパイルと実行を一気に行いましょう。

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 1.50s
     Running `target/debug/guessing_game`
Hello, world!
```

このゲーム（の開発）では各イテレーションを素早くテストしてから、次のイテレーションに移ります。run コマンドは、今回のようにプロジェクトのイテレーションを素早く回したいときに便利です。

> 訳注：ここでのイテレーションは、アジャイルな開発手法で用いられている用語にあたります。イテレーションとは開発工程の「一回のサイクル」のことで、サイクルには、設計、実装、テスト、改善（リリース後の振り返り）が含まれます。アジャイル開発ではイテレーションを数週間の短いスパンで一通り回し、それを繰り返すことで開発を進めていきます。
> この章では「実装」→「テスト」のごく短いサイクルを繰り返すことで、プログラムに少しずつ機能を追加していきます。

src/main.rs ファイルを開き直しましょう。このファイルにすべてのコードを書いていきます。

2.2　予想を処理する

数当てゲームプログラムの最初の部分は、ユーザに入力を求め、その入力を処理し、期待した形式になっていることを確認することです。手始めに、プレーヤーが予想を入力できるようにしましょう。リスト 2-1 のコードを **src/main.rs** に入力してください。

ファイル名：src/main.rs

```
use std::io;

fn main() {
    println!("Guess the number!");          // 数を当ててごらん

    println!("Please input your guess.");   // ほら、予想を入力してね

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");     // 行の読み込みに失敗しました

    println!("You guessed: {guess}");       // 次のように予想しました: {guess}
}
```

リスト 2-1：ユーザに予想を入力してもらい、それを出力するコード

このコードには多くの情報が詰め込まれています。行ごとに見ていきましょう。ユーザ入力を受け付け、結果を出力するためには io（入出力）ライブラリをスコープに入れる必要があります。io ライブラリは、std と呼ばれる標準ライブラリに含まれています。

```
use std::io;
#
# fn main() {
#     println!("Guess the number!");          // 数を当ててごらん
#
#     println!("Please input your guess.");   // ほら、予想を入力してね
#
#     let mut guess = String::new();
#
#     io::stdin()
#         .read_line(&mut guess)
#         .expect("Failed to read line");     // 行の読み込みに失敗しました
#
#     println!("You guessed: {guess}");       // 次のように予想しました: {guess}
# }
```

Rust はデフォルトで、標準ライブラリで定義されているアイテムの中のいくつかを、すべてのプログラムのスコープに取り込みます。このセットは **prelude**（プレリュード）と呼ばれ、標準ライブラリのドキュメントでその中のすべてを見ることができます。

使いたい型が prelude にない場合は、その型を use 文で明示的にスコープに入れる必要があります。std::io ライブラリを use すると、ユーザ入力を受け付ける機能など（入出力に関する）多くの便利な機能が利用できるようになります。

第 1 章で見た通り、main 関数がプログラムへのエントリーポイント（訳注：スタート地点）になります。

```
# use std::io;
#
fn main() {
#     println!("Guess the number!");          // 数を当ててごらん
#
#     println!("Please input your guess.");   // ほら、予想を入力してね
#
#     let mut guess = String::new();
#
#     io::stdin()
#         .read_line(&mut guess)
#         .expect("Failed to read line");     // 行の読み込みに失敗しました
#
#     println!("You guessed: {guess}");       // 次のように予想しました: {guess}
# }
```

fn 構文は関数を新しく宣言し、かっこの () は引数がないことを示し、波括弧の { は関数の本体を開始します。

また、第 1 章で学んだように、println! は画面に文字列を表示するマクロです.

```
# use std::io;
#
# fn main() {
    println!("Guess the number!");          // 数を当ててごらん

    println!("Please input your guess.");   // ほら、予想を入力してね
#
#     let mut guess = String::new();
#
#     io::stdin()
#         .read_line(&mut guess)
#         .expect("Failed to read line");     // 行の読み込みに失敗しました
#
#     println!("You guessed: {guess}");       // 次のように予想しました: {guess}
# }
```

このコードはゲームの内容などを示すプロンプトを表示し、ユーザに入力を求めています。

2.2.1　値を変数に保持する

次に、ユーザの入力を格納するための**変数**を作りましょう。こんな感じです。

```
# use std::io;
#
# fn main() {
#     println!("Guess the number!");          // 数を当ててごらん
#
#     println!("Please input your guess.");   // ほら、予想を入力してね
#
```

```
    let mut guess = String::new();
#
#     io::stdin()
#         .read_line(&mut guess)
#         .expect("Failed to read line");     // 行の読み込みに失敗しました
#
#     println!("You guessed: {guess}");       // 次のように予想しました: {guess}
# }
```

プログラムが少し興味深いものになってきました。この小さな行の中でいろいろなことが起きています。`let` 文を使って変数を作っています。別の例も見てみましょう。

```
let apples = 5;
```

この行では `apples` という名前の新しい変数を作成し `5` という値に束縛しています。Rust では変数はデフォルトで不変（immutable）で、これは一度変数に値を与えたらその値は変わらないという意味です。この概念については第 3 章の「変数と可変性」の節で詳しく説明します。変数を可変（mutable）にするには、変数名の前に `mut` をつけます。

```
let apples = 5; // immutable
                // 不変
let mut bananas = 5; // mutable
                     // 可変
```

> 注：`//` 構文は行末まで続くコメントを開始し、Rust はコメント内のすべて無視します。コメントについては第 3 章で詳しく説明します。

数当てゲームのプログラムに戻りましょう。ここまでの話で `let mut guess` が `guess` という名前の可変変数を導入することがわかったと思います。等号記号（`=`）は Rust に、いまこの変数を何かに束縛したいことを伝えます。等号記号の右側には `guess` が束縛される値があります。これは `String::new` 関数を呼び出すことで得られた値で、この関数は `String` 型の新しいインスタンスを返します。`String` は標準ライブラリによって提供される文字列型で、サイズが拡張可能な、UTF-8 でエンコードされたテキスト片になります。

`::new` の行にある `::` 構文は `new` が `String` 型の関連関数であることを示しています。**関連関数**とは、ある型（ここでは `String`）に対して実装される関数のことです。この `new` 関数は新しい空の文字列を作成します。`new` 関数は多くの型に見られます。なぜなら、何らかの新しい値を作成する関数によくある名前だからです。

つまり `let mut guess = String::new();` という行は可変変数を作成し、その変数は現時点では新しい空の `String` のインスタンスに束縛されているわけです。ふう！

2.2.2　ユーザの入力を受け取る

プログラムの最初の行に use std::io と書いて、標準ライブラリの入出力機能を取り込んだことを思い出してください。ここで io モジュールの stdin 関数を呼び出して、ユーザ入力を処理できるようにしましょう。

```
# use std::io;
#
# fn main() {
#     println!("Guess the number!");          // 数を当ててごらん
#
#     println!("Please input your guess.");   // ほら、予想を入力してね
#
#     let mut guess = String::new();
#
    io::stdin()
        .read_line(&mut guess)
#         .expect("Failed to read line");     // 行の読み込みに失敗しました
#
#     println!("You guessed: {guess}");       // 次のように予想しました: {guess}
# }
```

もし、プログラムの最初に use std::io; と書いて io ライブラリをインポートしていなかったとしても、std::io::stdin のように呼び出せば、この関数を利用できます。stdin 関数はターミナルの標準入力へのハンドルを表す型である std::io::Stdin のインスタンスを返します。

次の .read_line(&mut guess) 行は、標準入力ハンドルの read_line メソッドを呼び出し、ユーザからの入力を得ています。また、read_line の引数として &mut guess を渡し、ユーザ入力をどの文字列に格納するかを指示しています。read_line メソッドの仕事は、ユーザが標準入力に入力したものを文字列に（いまの内容を上書きせずに）追加することですので、文字列を引数として渡しているわけです。引数の文字列は、その内容をメソッドが変更できるように、可変である必要があります。

この & は、この引数が**参照**であることを示し、これによりコードの複数の部分が同じデータにアクセスしても、そのデータを何度もメモリにコピーしなくて済みます。参照は複雑な機能（訳注：一部のプログラム言語では正しく使うのが難しい機能）ですが、Rust の大きな利点の一つは参照を安全かつ簡単に使用できることです。このプログラムを完成させるのに、そのような詳細を知る必要はないでしょう。とりあえず知っておいてほしいのは、変数のように参照もデフォルトで不変であることです。したがって、&guess ではなく &mut guess と書いて可変にする必要があります。（参照については第 4 章でより詳しく説明します）

2.2.3　Result で失敗の可能性を扱う

まだ、このコードの行は終わってません。これから説明するのはテキスト上は 3 行目になりますが、まだ一つの論理的な行の一部分に過ぎません。次の部分はこのメソッドです。

```
# use std::io;
#
# fn main() {
#     println!("Guess the number!");          // 数を当ててごらん
#
#     println!("Please input your guess.");   // ほら、予想を入力してね
#
#     let mut guess = String::new();
#
#     io::stdin()
#         .read_line(&mut guess)
        .expect("Failed to read line");     // 行の読み込みに失敗しました
#
#     println!("You guessed: {guess}");       // 次のように予想しました: {guess}
# }
```

このコードは、こう書くこともできました。

```
io::stdin().read_line(&mut guess).expect("Failed to read line");
```

しかし、長い行は読みづらいので分割したほうがよいでしょう。`.method_name()` 構文でメソッドを呼び出すとき、長い行を改行と空白で分割するのが賢明なことがよくあります。それでは、この行（`expect()` メソッド）が何をするのか説明します。

前述したように、`read_line` メソッドは渡された文字列にユーザが入力したものを入れますが、同時に `Result` 値も返します。`Result` は**列挙型**、または **enum** ともよく呼ばれるもののひとつです。列挙型は、複数の取りうる状態の中からどれか一つになることができる型です。私たちはこのそれぞれの取りうる状態のことを**列挙子** (variant) と呼びます。

enum については第 6 章で詳しく説明します。これらの `Result` 型の目的は、エラー処理に関わる情報を符号化（エンコード）することです。

`Result` の列挙子は `Ok` か `Err` です。`Ok` 列挙子は処理が成功したことを示し、`Ok` の中には正常に生成された値が入っています。`Err` 列挙子は処理が失敗したことを意味し、`Err` には処理が失敗した過程や理由についての情報が含まれています。

`Result` 型の値にも、他の型と同様にメソッドが定義されています。`Result` のインスタンスには `expect` メソッドがありますので、これを呼び出せます。この `Result` インスタンスが `Err` の値の場合、`expect` メソッドはプログラムをクラッシュさせ、引数として渡されたメッセージを表示します。`read_line` メソッドが `Err` を返したら、それはおそらく基礎となるオペレーティング・システムに起因するものでしょう。もしこの `Result` オブジェクトが `Ok` 値の場合、`expect` メソッドは `Ok` 列挙子が保持する戻り値を取り出して、その値だけを返してくれます。こうして私たちはその値を使うことができるわけです。今回の場合、その値はユーザ入力のバイト数になります。

もし `expect` メソッドを呼び出さなかったら、コンパイルはできるものの警告が出るでしょう。

```
$ cargo build
```

```
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
warning: unused `Result` that must be used
(警告: 使用されなければならない`Result`が使用されていません)
  --> src/main.rs:10:5
   |
10 |     io::stdin().read_line(&mut guess);
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: this `Result` may be an `Err` variant, which should be handled
   = note: `#[warn(unused_must_use)]` on by default
help: use `let _ = ...` to ignore the resulting value
(ヘルプ: 結果の値を無視するには `let _ = ...` を使用してください)
   |
10 |     let _ = io::stdin().read_line(&mut guess);
   |     +++++++

warning: `guessing_game` (bin "guessing_game") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 0.59s
```

Rust は私たちが `read_line` から返された `Result` 値を使用していないことを警告し、これはプログラムがエラーの可能性に対処していないことを示します。

警告を抑制する正しい方法は実際にエラー処理コードを書くことです。しかし、現時点では問題が起きたときにこのプログラムをクラッシュさせたいだけなので、`expect` が使えるわけです。エラーからの回復については第 9 章で学びます。

2.2.4　println! マクロのプレースホルダーで値を表示する

閉じ波かっこを除けば、ここまでのコードで説明するのは残り 1 行だけです。

```
# use std::io;
#
# fn main() {
#     println!("Guess the number!");          // 数を当ててごらん
#
#     println!("Please input your guess.");   // ほら、予想を入力してね
#
#     let mut guess = String::new();
#
#     io::stdin()
#         .read_line(&mut guess)
#         .expect("Failed to read line");     // 行の読み込みに失敗しました
#
    println!("You guessed: {guess}");       // 次のように予想しました: {guess}
# }
```

この行はユーザの入力を現在保持している文字列を表示します。一組の波括弧の `{}` はプレースホルダーです。`{}` は値を所定の場所に保持する小さなカニのはさみだと考えてください。変数の値を表示するときは、変数名を波括弧の中に入れればよいです。式の評価結果を表示するときは、フォー

マット文字列の中に空の波括弧を置き、それぞれの空の波括弧プレースホルダに表示する式を同じ順で、カンマ区切りリストにして続けてください。一回の println! の呼び出しで変数と式の結果を表示するなら次のようになります。

```
let x = 5;
let y = 10;

println!("x = {x} and y + 2 = {}", y + 2);
```

このコードは x = 5 and y + 2 = 12 と表示するでしょう。

2.2.5　最初の部分をテストする

数当てゲームの最初の部分をテストしてみましょう。cargo run で走らせてください。

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 6.44s
     Running `target/debug/guessing_game`
Guess the number!
Please input your guess.
6
You guessed: 6
```

これで、キーボードからの入力を得て、それを表示するという、ゲームの最初の部分は完成になります。

2.3　秘密の数字を生成する

次にユーザが数当てに挑戦する秘密の数字を生成する必要があります。この数字を毎回変えることで何度やっても楽しいゲームになります。ゲームが難しくなりすぎないように 1 から 100 までの乱数を使用しましょう。Rust の標準ライブラリには、まだ乱数の機能は含まれていません。ですが、Rust の開発チームがこの機能を持つ rand クレートを提供してくれています。

2.3.1　クレートを使用して機能を追加する

クレートは Rust ソースコードを集めたものであることを思い出してください。私たちがここまで作ってきたプロジェクトは**バイナリクレート**であり、これは実行可能ファイルになります。rand クレートは**ライブラリクレート**です。他のプログラムで使用するためのコードが含まれており、単独で実行することはできません。

Cargo がその力を発揮するのは外部クレートと連携するときです。rand を使ったコードを書く前に、**Cargo.toml** ファイルを編集して rand クレートを依存関係に含める必要があります。そのファイルを開いて、Cargo が作ってくれた [dependencies] セクションヘッダの下に次の行を追加してく

ださい。バージョンナンバーを含め、ここに書かれている通り正確に rand を指定してください。そうしないと、このチュートリアルのコード例が動作しないかもしれません:

ファイル名：Cargo.toml

```
[dependencies]
rand = "0.8.5"
```

Cargo.toml ファイルでは、ヘッダに続くものはすべて、他のセクションが始まるまで続くセクションの一部になります。（訳注：Cargo.toml ファイル内には複数のセクションがあり、各セクションは [ ] で囲まれたヘッダ行から始まります）

[dependencies] はプロジェクトが依存する外部クレートと必要とするバージョンを Cargo に伝えます。今回は rand クレートを 0.8.5 というセマンティックバージョン指定子で指定します。Cargo はセマンティックバージョニング（**SemVer** と呼ばれることもあります）を理解しており、これはバージョンナンバーを記述するための標準です。0.8.5 という指定子は実際には ^0.8.5 の省略記法で、0.8.5 以上 0.9.0 未満の任意のバージョンを意味します。

Cargo はこれらのバージョンを、バージョン 0.8.5 と互換性のある公開 API を持つものとみなします。この仕様により、この章のコードが引き続きコンパイルできるようにしつつ、最新のパッチリリースを取得できるようになります。0.9.0 以降のバージョンは、以下の例で使用しているものと同じ API を持つことを保証しません。

さて、コードを一切変えずに、次のリスト 2-2 のようにプロジェクトをビルドしてみましょう。

```
$ cargo build
    Updating crates.io index
    (crates.ioインデックスを更新しています)
  Downloaded rand v0.8.5
  (rand v0.8.5をダウンロードしています)
  Downloaded libc v0.2.127
  Downloaded getrandom v0.2.7
  Downloaded cfg-if v1.0.0
  Downloaded ppv-lite86 v0.2.16
  Downloaded rand_chacha v0.3.1
  Downloaded rand_core v0.6.3
   Compiling libc v0.2.127
   (libc v0.2.127をコンパイルしています)
   Compiling getrandom v0.2.7
   Compiling cfg-if v1.0.0
   Compiling ppv-lite86 v0.2.16
   Compiling rand_core v0.6.3
   Compiling rand_chacha v0.3.1
   Compiling rand v0.8.5
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
   (guessing_game v0.1.0をコンパイルしています)
    Finished dev [unoptimized + debuginfo] target(s) in 2.53s
```

リスト 2-2：rand クレートを依存として追加した後の cargo build コマンドの出力

もしかしたら異なるバージョンナンバー（とはいえ、SemVer のおかげですべてのコードに互換性があります）や、異なる行（オペレーティングシステムに依存します）が表示されるかもしれません。また、行の順序も違うかもしれません。

外部依存を持つようになると、Cargo はその依存関係が必要とするすべてについて最新のバージョンを**レジストリ**から取得します。レジストリとは Crates.io のデータのコピーです。Crates.io は、Rust のエコシステムにいる人たちがオープンソースの Rust プロジェクトを投稿し、他の人が使えるようにする場所です。

レジストリの更新後、Cargo は `[dependencies]` セクションにリストアップされているクレートをチェックし、まだ取得していないものがあればダウンロードします。ここでは依存関係として `rand` だけを書きましたが、`rand` が動作するために依存している他のクレートも取り込まれています。クレートをダウンロードしたあと、Rust はそれらをコンパイルし、依存関係が利用できる状態でプロジェクトをコンパイルします。

何も変更せずにすぐに `cargo build` コマンドを再度実行すると、`Finished` の行以外は何も出力されないでしょう。Cargo はすでに依存関係をダウンロードしてコンパイル済みであることを認識しており、また、あなたが **Cargo.toml** ファイルを変更していないことも知っているからです。さらに、Cargo はあなたがコードを何も変更していないことも知っているので、再コンパイルもしません。何もすることがないので単に終了します。

src/main.rs ファイルを開いて些細な変更を加え、それを保存して再度ビルドすると 2 行しか表示されません。

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 2.53 secs
```

これらの行は Cargo が **src/main.rs** ファイルへの小さな変更に対して、ビルドを更新していることを示しています。依存関係は変わっていないので、Cargo は既にダウンロードしてコンパイルしたものが再利用できることを知っています。

2.3.1.1　Cargo.lock ファイルで再現可能なビルドを確保する

Cargo はあなたや他の人があなたのコードをビルドするたびに、同じ生成物をリビルドできるようにするしくみを備えています。Cargo は何も指示されない限り、指定したバージョンの依存のみを使用します。たとえば来週 `rand` クレートのバージョン 0.8.6 が出て、そのバージョンには重要なバグ修正が含まれていますが、同時にあなたのコードを破壊するリグレッションも含まれているとします。これに対応するため、Rust は `cargo build` を最初に実行したときに **Cargo.lock** ファイルを作成します。（いまの **guessing_game** ディレクトリにもあるはずです）

プロジェクトを初めてビルドするとき、Cargo は条件に合うすべての依存関係のバージョンを計算し **Cargo.lock** ファイルに書き込みます。次にプロジェクトをビルドすると、Cargo は **Cargo.lock** ファイルが存在することを確認し、バージョンを把握するすべての作業を再び行う代わりに、そこで

指定されているバージョンを使うでしょう。これにより再現性のあるビルドを自動的に行えます。言い換えれば、**Cargo.lock** ファイルのおかげで、あなたが明示的にアップグレードするまで、プロジェクトは 0.8.5 を使い続けます。**Cargo.lock** ファイルは再現性のあるビルドのために重要なので、プロジェクトの残りのコードとともにソース管理にチェックインされることが多いです。

2.3.1.2　クレートを更新して新バージョンを取得する

クレートを**本当に**アップグレードしたくなったときのために、Cargo は `update` コマンドを提供します。このコマンドは **Cargo.lock** ファイルを無視して、**Cargo.toml** ファイル内の全ての指定に適合する最新バージョンを算出します。成功したら Cargo はそれらのバージョンを **Cargo.lock** ファイルに記録します。ただし、デフォルトで Cargo は 0.8.5 以上、0.9.0 未満のバージョンのみを検索します。もし `rand` クレートの新しいバージョンとして 0.8.6 と 0.9.0 の二つがリリースされていたなら、`cargo update` を実行したときに以下のようなメッセージが表示されるでしょう。

```
$ cargo update
    Updating crates.io index
    (crates.ioインデックスを更新しています)
    Updating rand v0.8.5 -> v0.8.6
    (randクレートをv0.8.5 -> v0.8.6に更新しています)
```

Cargo は 0.9.0 リリースを無視します。またそのとき、**Cargo.lock** ファイルが変更され、`rand` クレートの現在使用中のバージョンが 0.8.6 になったことにも気づくでしょう。そうではなく、`rand` のバージョン 0.9.0 か、0.9.**x** 系のどれかを使用するには、**Cargo.toml** ファイルを以下のように変更する必要があります。

```
[dependencies]
rand = "0.9.0"
```

次に `cargo build` コマンドを実行したとき、Cargo は利用可能なクレートのレジストリを更新し、あなたが指定した新しいバージョンに従って `rand` の要件を再評価します。

Cargo とそのエコシステムについては、まだ伝えたいことが山ほどありますが、それらについては第 14 章で説明します。いまのところは、これだけ知っていれば十分です。Cargo はライブラリの再利用をとても簡単にしてくれるので、Rustacean が数多くのパッケージから構成された小さなプロジェクトを書くことが可能になっています。

2.3.2　乱数を生成する

`rand` クレートを使って予想する数字を生成しましょう。次のステップは **src/main.rs** ファイルをリスト 2-3 のように更新することです。

ファイル名：src/main.rs

```
use std::io;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1..=100);

    println!("The secret number is: {secret_number}");    //秘密の数字は次の通り
    : {secret_number}

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {guess}");
}
```

リスト 2-3：乱数を生成するコードの追加

まず `use rand::Rng;` という行を追加します。`Rng` トレイトは乱数生成器が実装すべきメソッドを定義しており、それらのメソッドを使用するには、このトレイトがスコープ内になければなりません。トレイトについて詳しくは第 10 章で解説します。

次に、途中に 2 行を追加しています。最初の行では `rand::thread_rng` 関数を呼び出して、これから使う、ある特定の乱数生成器を取得しています。なお、この乱数生成器は現在のスレッドに固有で、オペレーティングシステムからシード値を得ています。そして、この乱数生成器の `gen_range` メソッドを呼び出しています。このメソッドは `use rand::Rng;` 文でスコープに導入した `Rng` トレイトで定義されています。`gen_range` メソッドは範囲式を引数にとり、その範囲内の乱数を生成してくれます。ここで使っている範囲式の種類は `開始..=終了` という形式で、下限値と上限値をともに含みます。そのため、1 から 100 までの数をリクエストするには `1..101` と指定する必要があります。

> 注：クレートのどのトレイトを `use` するかや、どのメソッドや関数を呼び出すかを知るために、各クレートにはその使い方を説明したドキュメントが用意されています。Cargo のもう一つの素晴らしい機能は、`cargo doc --open` コマンドを走らせると、すべての依存クレートが提供するドキュメントをローカルでビルドして、ブラウザで開いてくれることです。たとえば `rand` クレートの他の機能に興味があるなら、`cargo doc --open` コマンドを実行して、左側のサイドバーにある `rand` をクリックしてください。

コードに追加した 2 行目は秘密の数字を表示します。これはプログラムを開発している間のテストに便利ですが、最終版からは削除する予定です。プログラムが始まってすぐに答えが表示されたら

ゲームになりませんからね！

試しにプログラムを何回か走らせてみてください。

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 2.53s
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 7
Please input your guess.
4
You guessed: 4

$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 83
Please input your guess.
5
You guessed: 5
```

毎回異なる乱数を取得し、それらはすべて 1 から 100 の範囲内の数字になるはずです。よくやりました！

2.4　予想と秘密の数字を比較する

さて、ユーザ入力と乱数が揃ったので両者を比較してみましょう。このステップをリスト 2-4 に示します。これから説明するように、このコードはまだコンパイルできないことに注意してください。

ファイル名：src/main.rs

```
use rand::Rng;
use std::cmp::Ordering;
use std::io;

fn main() {
    // --snip--
#     println!("Guess the number!");
#
#     let secret_number = rand::thread_rng().gen_range(1..=100);
#
#     println!("The secret number is: {secret_number}");
#
#     println!("Please input your guess.");
#
#     let mut guess = String::new();
#
#     io::stdin()
```

```
#         .read_line(&mut guess)
#         .expect("Failed to read line");

    println!("You guessed: {guess}");

    match guess.cmp(&secret_number) {
        Ordering::Less => println!("Too small!"),       //小さすぎ！
        Ordering::Greater => println!("Too big!"),      //大きすぎ！
        Ordering::Equal => println!("You win!"),        //やったね！
    }
}
```

リスト 2-4：二つの数値を比較したときに返される可能性のある値を処理する

まず use 文を追加して標準ライブラリから std::cmp::Ordering という型をスコープに導入しています。Ordering も enum の一つで Less 、Greater 、Equal という列挙子を持っています。これらは二つの値を比較したときに得られる 3 種類の結果です。

それから Ordering 型を使用する新しい 5 行をいちばん下に追加してしています。cmp メソッドは二つの値の比較を行い、比較できるものになら何に対しても呼び出せます。比較対象への参照をとり、ここでは guess と secret_number を比較しています。そして use 文でスコープに導入した Ordering 列挙型の列挙子を返します。ここでは match 式を使用しており、guess と secret_number の値に対して cmp を呼んだ結果返された Ordering の列挙子に基づき、次の動作を決定しています。

match 式は複数の**アーム**（腕）で構成されます。各アームはマッチさせる**パターン**と、match に与えられた値がそのアームのパターンにマッチしたときに実行されるコードで構成されます。Rust は match に与えられた値を受け取って、各アームのパターンを順に照合していきます。パターンと match 式は Rust の強力な機能です: コードが遭遇する可能性のあるさまざまな状況を表現し、それらすべてを確実に処理できるようにします。これらの機能については、それぞれ第 6 章と第 18 章で詳しく説明します。

ここで使われている match 式に対して、例を通して順に見ていきましょう。たとえばユーザが 50 と予想し、今回ランダムに生成された秘密の数字は 38 だったとしましょう。

コードが 50 と 38 を比較すると、50 は 38 よりも大きいので cmp メソッドは Ordering::Greater を返します。match 式は Ordering::Greater の値を取得し、各アームのパターンを吟味し始めます。まず最初のアームのパターンである Ordering::Less を見て、Ordering::Greater の値と Ordering::Less がマッチしないことがわかります。そのため、このアームのコードは無視して、次のアームに移ります。次のアームのパターンは Ordering::Greater で、これは Ordering::Greater と**マッチ**します！このアームに関連するコードが実行され、画面に Too big! と表示されます。このシナリオでは最初に成功したマッチで match 式（の評価）は終了し、最後のアームとは照合されません。

ところがリスト 2-4 のコードはまだコンパイルできません。試してみましょう。

```
$ cargo build
   Compiling libc v0.2.86
```

```
   Compiling getrandom v0.2.2
   Compiling cfg-if v1.0.0
   Compiling ppv-lite86 v0.2.10
   Compiling rand_core v0.6.2
   Compiling rand_chacha v0.3.0
   Compiling rand v0.8.5
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
error[E0308]: mismatched types          (型が合いません)
  --> src/main.rs:22:21
   |
22 |     match guess.cmp(&secret_number) {
   |                 --- ^^^^^^^^^^^^^^^ expected `&String`, found `&{integer}`
   |                 |                   (`&String`を予期したけど、`&{integer}`が
       見つかりました)
   |                 |
   |                 arguments to this method are incorrect
   |                 (このメソッドへの引数が正しくありません)
   |
   = note: expected reference `&String`
              found reference `&{integer}`
note: method defined here
  --> /rustc/07dca489ac2d933c78d3c5158e3f43beefeb02ce/library/core/src/cmp.rs
      :814:8

For more information about this error, try `rustc --explain E0308`.
error: could not compile `guessing_game` (bin "guessing_game") due to 1 previous
     error      (先の1つのエラーのため、`guessing_game` (bin "guessing_game") を
    コンパイルできませんでした)
```

このエラーの核心は**型の不一致**があると述べていることです。Rust は強い静的型システムを持ちますが、型推論も備えています。`let guess = String::new()` と書いたとき、Rust は `guess` が `String` 型であるべきと推論したので、私たちはその型を書かずに済みました。一方で `secret_number` は数値型です。Rust のいくつかの数値型は 1 から 100 までの値を表現でき、それらの型には 32 ビット数値の `i32`、符号なしの 32 ビット数値の `u32`、64 ビット数値の `i64` などがあります。Rust のデフォルトは `i32` 型で、型情報をどこかに追加して Rust に異なる数値型だと推論させない限り `secret_number` の型はこれになります。エラーの原因は Rust が文字列と数値型を比較できないためです。

最終的にはプログラムが入力として読み込んだ `String` を実数型に変換し、秘密の数字と数値として比較できるようにしたいわけです。そのためには `main` 関数の本体に次の行を追加します。

ファイル名：src/main.rs

```
# use rand::Rng;
# use std::cmp::Ordering;
# use std::io;
#
# fn main() {
#     println!("Guess the number!");
#
#     let secret_number = rand::thread_rng().gen_range(1..=100);
```

```
#
#     println!("The secret number is: {secret_number}");
#
#     println!("Please input your guess.");
#
    // --snip--

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");

    let guess: u32 = guess.trim().parse().expect("Please type a number!"); //数
    値を入力してください！

    println!("You guessed: {guess}");

    match guess.cmp(&secret_number) {
        Ordering::Less => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal => println!("You win!"),
    }
# }
```

その行とはこれのことです。

```
let guess: u32 = guess.trim().parse().expect("Please type a number!");
```

guess という名前の変数を作成しています。しかし待ってください、このプログラムには既に guess という名前の変数がありませんでしたか？　たしかにありますが、Rust では guess の前の値を新しい値で覆い隠す（shadow する）ことが許されているのです。**シャドーイング**（shadowing）は、guess_str と guess のような重複しない変数を二つ作る代わりに、guess という変数名を再利用させてくれるのです。これについては第 3 章で詳しく説明しますが、今のところ、この機能はある型から別の型に値を変換するときによく使われることを知っておいてください。

この新しい変数を guess.trim().parse() という式に束縛しています。式の中にある guess は、入力が文字列として格納されたオリジナルの guess 変数を指しています。String インスタンスの trim メソッドは文字列の先頭と末尾の空白をすべて削除します。これは数値データのみを表現できる u32 型とこの文字列を比較するために（準備として）行う必要があります。ユーザは予想を入力したあと read_line の処理を終えるために Enter キーを押す必要がありますが、これにより文字列に改行文字が追加されます。たとえばユーザが 5 と入力して Enter キーを押すと、guess は 5\n になります。この \n は「改行」を表しています。（Windows では Enter キーを押すとキャリッジリターンと改行が入り \r\n となります。）trim メソッドは \n や \r\n を削除するので、その結果 5 だけになります。

文字列の parse メソッドは文字列を別の型に変換します。ここでは、私たちは文字列を数値に変換するために使います。let guess: u32 として、Rust に欲しい数値の正確な型を伝える必要がありま

す。guess の後にコロン（ : ）を付けることで変数の型に注釈をつけることを Rust に伝えています。Rust には組み込みの数値型がいくつかあります。ここにある u32 は符号なし 32 ビット整数で、小さな正の数を表すデフォルトの型に適しています。他の数値型については第 3 章で学びます。

さらに、このサンプルプログラムでは、u32 という注釈と secret_number 変数との比較していることから、Rust は secret_number 変数も u32 型であるべきだと推論しています。つまり、いまでは二つの同じ型の値を比較することになるわけです！

parse メソッドは論理的に数値に変換できる文字にしか使えないので、よくエラーになります。たとえば文字列に A👍% が含まれていたら数値に変換する術はありません。解析に失敗する可能性があるため、parse メソッドは read_line メソッドと同様に Result 型を返します（「Result で失敗の可能性を扱う」で説明しました） 今回も expect メソッドを使用して Result 型を同じように扱います。parse メソッドが文字列から数値を作成できなかったために Result 型の Err 列挙子を返したら、expect の呼び出しはゲームをクラッシュさせ、私たちが与えたメッセージを表示します。parse が文字列をうまく数値へ変換できたときは Result 型の Ok 列挙子を返し、expect は Ok 値から欲しい数値を返してくれます。

さあ、プログラムを走らせましょう:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 0.43s
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 58
Please input your guess.
  76
You guessed: 76
Too big!
```

いい感じです！　予想の前にスペースを追加したにもかかわらず、プログラムはちゃんとユーザが 76 と予想したことを理解しました。このプログラムを何回か走らせ、数字を正しく言い当てたり、大きすぎる数字や小さすぎる数字を予想したりといった、異なる種類の入力に対する動作の違いを検証してください。

現在、ゲームの大半は動作していますが、まだユーザは 1 回しか予想できません。ループを追加して、その部分を変更しましょう！

2.5　ループで複数回の予想を可能にする

loop キーワードは無限ループを作成します。ループを追加してユーザが数字を予想する機会を増やします。

ファイル名：src/main.rs

```
# use rand::Rng;
```

```
# use std::cmp::Ordering;
# use std::io;
#
# fn main() {
#     println!("Guess the number!");
#
#     let secret_number = rand::thread_rng().gen_range(1..=100);
#
    // --snip--

    println!("The secret number is: {secret_number}");

    loop {
        println!("Please input your guess.");

        // --snip--

#
#         let mut guess = String::new();
#
#         io::stdin()
#             .read_line(&mut guess)
#             .expect("Failed to read line");
#
#         let guess: u32 = guess.trim().parse().expect("Please type a number!");
#
#         println!("You guessed: {guess}");
#
        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => println!("You win!"),
        }
    }
}
```

見ての通り予想入力のプロンプト以降をすべてループ内に移動しました。ループ内の行をさらに 4 つのスペースでインデントして、もう一度プログラムを実行してください。プログラムはいつまでも推測を求めるようになりましたが、実はこれが新たな問題を引き起こしています。これではユーザが（ゲームを）終了できません！

ユーザはキーボードショートカットの ctrl-c を使えば、いつでもプログラムを中断させられます。しかし「予想と秘密の数字を比較する」の parse で述べたように、この飽くなきモンスターから逃れる方法はもう一つあります。ユーザが数字以外の答えを入力すればプログラムはクラッシュします。それを利用して以下のようにすれば終了できます。

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 1.50s
     Running `target/debug/guessing_game`
```

```
Guess the number!
The secret number is: 59
Please input your guess.
45
You guessed: 45
Too small!
Please input your guess.
60
You guessed: 60
Too big!
Please input your guess.
59
You guessed: 59
You win!
Please input your guess.
quit
thread 'main' panicked at 'Please type a number!: ParseIntError { kind:
    InvalidDigit }', src/main.rs:28:47
(スレッド'main'は'数字を入力してください！ ：ParseIntError { kind: InvalidDigit
    }', src/libcore/result.rs:785でパニックしました)
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
(注：`RUST_BACKTRACE=1`で走らせるとバックトレースを見れます)
```

quit と入力すればゲームが終了しますが、数字以外の入力でもそうなります。これは控えめに言っても最適ではありません。私たちは正しい数字が予想されたときにゲームが停止するようにしたいのです。

2.5.1　正しい予想をした後に終了する

break 文を追加して、ユーザが勝ったらゲームが終了するようにプログラムしましょう。

ファイル名：src/main.rs

```
# use rand::Rng;
# use std::cmp::Ordering;
# use std::io;
#
# fn main() {
#     println!("Guess the number!");
#
#     let secret_number = rand::thread_rng().gen_range(1..=100);
#
#     println!("The secret number is: {secret_number}");
#
#     loop {
#         println!("Please input your guess.");
#
#         let mut guess = String::new();
#
#         io::stdin()
```

```
#             .read_line(&mut guess)
#             .expect("Failed to read line");
#
#         let guess: u32 = guess.trim().parse().expect("Please type a number!");
#
#         println!("You guessed: {guess}");
#
        // --snip--

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => {
                println!("You win!");
                break;
            }
        }
    }
}
```

`You win!` の後に `break` の行を追記することで、ユーザが秘密の数字を正確に予想したときにプログラムがループを抜けるようになりました。ループは `main` 関数の最後の部分なので、ループを抜けることはプログラムを抜けることを意味します。

2.5.2　不正な入力を処理する

このゲームの動作をさらに洗練させるために、ユーザが数値以外を入力したときにプログラムをクラッシュさせるのではなく、数値以外を無視してユーザが数当てを続けられるようにしましょう。これはリスト 2-5 のように、`String` から `u32` に `guess` を変換する行を変えることで実現できます。

ファイル名：src/main.rs

```
# use rand::Rng;
# use std::cmp::Ordering;
# use std::io;
#
# fn main() {
#     println!("Guess the number!");
#
#     let secret_number = rand::thread_rng().gen_range(1..=100);
#
#     println!("The secret number is: {secret_number}");
#
#     loop {
#         println!("Please input your guess.");
#
#         let mut guess = String::new();
#
        // --snip--
```

```
        io::stdin()
            .read_line(&mut guess)
            .expect("Failed to read line");

        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        println!("You guessed: {guess}");

        // --snip--
#
#         match guess.cmp(&secret_number) {
#             Ordering::Less => println!("Too small!"),
#             Ordering::Greater => println!("Too big!"),
#             Ordering::Equal => {
#                 println!("You win!");
#                 break;
#             }
#         }
#     }
# }
```

リスト 2-5：数値以外の予想を無視し、プログラムをクラッシュさせるのではなく、もう 1 回予想してもらう

`expect` の呼び出しから `match` 式に切り替えて、エラーによるクラッシュからエラー処理へと移行します。`parse` が `Result` 型を返すことと、`Result` が `Ok` と `Err` の列挙子を持つ列挙型であることを思い出してください。ここでは `match` 式を、`cmp` メソッドから返される `Ordering` を処理したときと同じように使っています。

もし `parse` メソッドが文字列から数値への変換に成功したなら、結果の数値を保持する `Ok` 値を返します。この `Ok` 値は最初のアームのパターンにマッチします。`match` 式は `parse` メソッドが生成して `Ok` 値に格納した `num` の値を返します。その数値は私たちが望んだように、これから作成する新しい `guess` 変数に収まります。

もし `parse` メソッドが文字列から数値への変換に**失敗**したなら、エラーに関する詳細な情報を含む `Err` 値を返します。この `Err` 値は最初の `match` アームの `Ok(num)` パターンにはマッチしませんが、2 番目のアームの `Err(_)` パターンにはマッチします。アンダースコアの `_` はすべての値を受け付けます。この例ではすべての `Err` 値に対して、その中にどんな情報があってもマッチさせたいと言っているのです。したがってプログラムは 2 番目のアームのコードである `continue` を実行します。これは `loop` の次の繰り返しに移り、別の予想を求めるようプログラムに指示します。つまり実質的にプログラムは `parse` メソッドが遭遇し得るエラーをすべて無視するようになります！

これでプログラム内のすべてが期待通りに動作するはずです。試してみましょう。

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 4.45s
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 61
Please input your guess.
10
You guessed: 10
Too small!
Please input your guess.
99
You guessed: 99
Too big!
Please input your guess.
foo
Please input your guess.
61
You guessed: 61
You win!
```

素晴らしい！　最後にほんの少し手を加えれば数当てゲームは完成です。このプログラムはまだ秘密の数字を表示していることを思い出してください。テストには便利でしたが、これではゲームが台無しです。秘密の数字を表示している `println!` を削除しましょう。最終的なコードをリスト 2-6 に示します。

ファイル名：src/main.rs

```
use rand::Rng;
use std::cmp::Ordering;
use std::io;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1..=100);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin()
            .read_line(&mut guess)
            .expect("Failed to read line");

        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };
```

```
        println!("You guessed: {guess}");

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => {
                println!("You win!");
                break;
            }
        }
    }
}
```

リスト 2-6：数当てゲームの完全なコード

数当てゲームを無事に作り上げることができました。おめでとうございます！

2.6　まとめ

このプロジェクトではハンズオンを通して、`let`、`match`、メソッド、関連関数、外部クレートの使いかたなど、多くの新しい Rust の概念に触れました。以降の章では、これらの概念についてより詳しく学びます。第 3 章では変数、データ型、関数など多くのプログラミング言語が持つ概念を取り上げ、Rust での使い方を説明します。第 4 章では Rust を他の言語とは異なるものに特徴づける、所有権について説明します。第 5 章では構造体とメソッドの構文について説明し、第 6 章では列挙型がどのように動くのかについて説明します。

第 3 章

一般的なプログラミングの概念

この章では、ほとんど全てのプログラミング言語で見られる概念を講義し、それらが Rust において、どう動作するかを見ていきます。多くのプログラミング言語は、その核心において、いろいろなものを共有しています。この章で提示する概念は、全て Rust に固有のものではありませんが、Rust の文脈で議論し、これらの概念を使用することにまつわる仕様を説明します。

具体的には、変数、基本的な型、関数、コメント、そして制御フローについて学びます。これらの基礎は全ての Rust プログラムに存在するものであり、それらを早期に学ぶことにより、強力な基礎を築くことになるでしょう。

3.0.0.1 キーワード

Rust 言語にも他の言語同様、キーワードが存在し、これらは言語だけが使用できるようになっています。これらの単語は、変数や関数名には使えないことを弁えておいてください。ほとんどのキーワードは、特別な意味を持っており、自らの Rust プログラムにおいて、様々な作業をこなすために使用することができます; いくつかは、紐付けられた機能がないものの、将来 Rust に追加されるかもしれない機能用に予約されています。キーワードの一覧は、**??**章で確認できます。

3.1 変数と可変性

「値を変数に保持する」の節で触れた通り、変数は標準で不変になります。これは、Rust が提供する安全性や簡便な並行性の利点を享受する形でコードを書くための選択の 1 つです。ところが、まだ変数を可変にするという選択肢も残されています。どのように、そしてなぜ Rust は不変性を推奨するのか、さらには、なぜそれとは違う道を選びたくなることがあるのか見ていきましょう。

変数が不変であると、値が一旦名前に束縛されたら、その値を変えることができません。これ

を具体的に説明するために、**projects** ディレクトリに `cargo new variables` コマンドを使って、**variables** という名前のプロジェクトを生成してください。

それから、新規作成した **variables** ディレクトリで、**src/main.rs** ファイルを開き、その中身を以下のコードに置き換えましょう。このコードはまだコンパイルできません:

ファイル名: src/main.rs

```
fn main() {
    let x = 5;
    println!("The value of x is: {x}");     // xの値は{x}です
    x = 6;
    println!("The value of x is: {x}");
}
```

これを保存し、`cargo run` コマンドでプログラムを走らせてください。次の出力に示されているような、不変性に関するエラーメッセージを受け取るはずです:

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
error[E0384]: cannot assign twice to immutable variable `x`
              (不変変数`x`に2回代入できません)
 --> src/main.rs:4:5
  |
2 |     let x = 5;
  |         -
  |         |
  |         first assignment to `x`
  |         (`x`への最初の代入)
  |         help: consider making this binding mutable: `mut x`
3 |     println!("The value of x is: {x}");
4 |     x = 6;
  |     ^^^^^ cannot assign twice to immutable variable

For more information about this error, try `rustc --explain E0384`.
error: could not compile `variables` (bin "variables") due to 1 previous error
```

この例では、コンパイラがプログラムに潜むエラーを見つけ出す手助けをしてくれることが示されています。コンパイルエラーは、イライラすることもあるものですが、本当はまだプログラムにしてほしいことを安全に行えていないだけということなのです。エラーが出るからといって、あなたがいいプログラマではないという意味ではあり**ません**！ 経験豊富な Rustacean でも、コンパイルエラーを出すことはあります。

`不変変数`x`に2回代入できません` というエラーメッセージを受け取りました。不変な `x` という変数に別の値を代入しようとしたからです。

不変と指定された値を変えようとした時に、コンパイルエラーが出るのは重要なことです。なぜなら、この状況はまさしく、バグに繋がるからです。コードのある部分は、値が変わることはないという前提のもとに処理を行い、別の部分がその値を変更していたら、最初の部分が目論見通りに動いて

いない可能性があるのです。このようなバグは、発生してしまってからでは原因が追いかけづらいものです。特に第 2 のコード片が、値を**時々**しか変えない場合、尚更です。Rust コンパイラは、値が不変であると宣言したら、本当に変わらないことを担保してくれるので、変更を自分で追いかける必要がなくなります。故にコードを通して正しいことを確認するのが簡単になるのです。

しかし可変性は非常に有用で、よりコードを書きやすくしてくれることもあります。変数は標準では不変ですが、第 2 章でやったように、変数名の前に `mut` キーワードを付けることで、可変にできるわけです。また `mut` を付けることで、コードの別の部分がこの変数の値を変えるだろうと示すことによって、未来の読者に対してその意図を汲ませることができるのです。

例として、**src/main.rs** ファイルを以下のように書き換えましょう:

ファイル名: src/main.rs

```
fn main() {
    let mut x = 5;
    println!("The value of x is: {x}");
    x = 6;
    println!("The value of x is: {x}");
}
```

今、このプログラムを走らせると、以下のような出力が得られます:

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
    Finished dev [unoptimized + debuginfo] target(s) in 0.30s
     Running `target/debug/variables`
The value of x is: 5   (xの値は5です)
The value of x is: 6
```

`mut` キーワードが使われると、`x` に束縛されている値を `5` から `6` に変更できます。可変性を使うかどうかは最終的にはプログラマに任せられており、どちらがより明白と思えるかは個別の状況によるでしょう。

3.1.1　定数

不変変数のように、**定数** (constants) は名前に束縛され、変更することが叶わない値のことですが、定数と変数の間にはいくつかの違いがあります。

まず、定数には `mut` キーワードは使えません: 定数は標準で不変であるだけでなく、常に不変なのです。定数は `let` キーワードの代わりに、`const` キーワードで宣言し、値の型は**必ず**注釈しなければなりません。型と型注釈については次のセクション、「データ型」で講義しますので、その詳細について気にする必要はありません。ただ単に型は常に注釈しなければならないのだと思っていてください。

定数はどんなスコープでも定義できます。グローバルスコープも含めてです。なので、いろんなところで使用される可能性のある値を定義するのに役に立ちます。

最後の違いは、定数は定数式にしかセットできないことです。実行時に評価される値にはセットできません。

これが定数定義の例です:

```
const THREE_HOURS_IN_SECONDS: u32 = 60 * 60 * 3;
```

定数の名前は `THREE_HOURS_IN_SECONDS` で、その値は 60（1 分あたりの秒数）×60（1 時間あたりの分数）×3（このプログラムで数えたい時間数）の結果にセットされています。Rust の定数の命名規則は、全て大文字でアンダースコアで単語区切りすることです。コンパイラはコンパイル時に一部の演算を評価することができるので、この定数に 10,800 という値を設定する代わりに、理解し検証しやすい方法でこの値を書き出すことを選択できます。定数宣言内でどの演算が使用できるかについてのさらなる情報は、Rust Reference の constant evaluation の節をお読みください。

定数は、プログラムが走る期間、定義されたスコープ内でずっと有効です。この性質のおかげで、定数はプログラムのいろんなところで使用される可能性のあるアプリケーション空間の値を定義するのに有用です。例えば、ゲームでプレイヤーが取得可能なポイントの最高値や、光速度などですね。

プログラム中で使用されるハードコードされた値に対して、定数として名前付けすることは、コードの将来的な管理者にとって値の意味を汲むのに役に立ちます。将来、ハードコードされた値を変える必要が出た時に、たった 1 箇所を変更するだけで済むようにもしてくれます。

3.1.2　シャドーイング

第 2 章の数当てゲームのチュートリアルで見たように、前に定義した変数と同じ名前の変数を新しく宣言できます。Rustacean はこれを、最初の変数は 2 番目の変数に**覆い隠さ**れたと言います。これはその変数名を使用した際に、コンパイラは 2 番目の変数を見るという意味です。2 番目の変数は実質的に、最初の変数にその影を投げかけ、自身が覆い隠されるかスコープが終了するまで、変数名の使用を自身へのものとして扱います。以下のようにして、同じ変数名を用いて変数を覆い隠し、`let` キーワードの使用を繰り返します:

ファイル名: src/main.rs

```
fn main() {
    let x = 5;

    let x = x + 1;

    {
        let x = x * 2;
        println!("The value of x in the inner scope is: {x}");
    }

    println!("The value of x is: {x}");
}
```

このプログラムはまず、x を 5 という値に束縛します。それから let x = を繰り返すことで新しい変数 x を作り、元の値に 1 を加えることになるので、x の値は 6 になります。次に波括弧によって作られた内側のスコープの中で、3 番目の let 文も x を覆い隠して新しい変数を作り、以前の値に 2 をかけることになるので、x の最終的な値は 12 になります。スコープが終わるとシャドーイングは終了し、x の値は元の 6 に戻ります。このプログラムを走らせたら、以下のように出力するでしょう:

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/variables`
The value of x in the inner scope is: 12
The value of x is: 6
```

シャドーイングは、変数を mut にするのとは違います。なぜなら、let キーワードを使わずに、誤ってこの変数に再代入を試みようものなら、コンパイルエラーが出るからです。let を使うことで、値にちょっとした加工は行えますが、その加工が終わったら、変数は不変になるわけです。

mut と上書きのもう一つの違いは、再度 let キーワードを使用したら、実効的には新しい変数を生成していることになるので、値の型を変えつつ、同じ変数名を使いまわせることです。例えば、プログラムがユーザに何らかのテキストに対して空白文字を入力することで何個分のスペースを表示したいかを尋ねて、そうしたらこの入力を数値として保持したいとしましょう:

```
# fn main() {
    let spaces = "   ";
    let spaces = spaces.len();
# }
```

最初の spaces 変数は文字列型であり、2 番目の spaces 変数は数値型です。故に、シャドーイングのおかげで、異なる名前を思いつく必要がなくなるわけです。spaces_str と spaces_num などですね; 代わりに、よりシンプルな spaces という名前を再利用できるわけです。一方で、この場合に mut を使おうとすると、以下に示した通りですが、コンパイルエラーになるわけです:

```
# fn main() {
    let mut spaces = "   ";
    spaces = spaces.len();
# }
```

変数の型を可変にすることは許されていないと言われているわけです:

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
error[E0308]: mismatched types          (型が合いません)
 --> src/main.rs:3:14
  |
```

```
2 |     let mut spaces = "   ";
  |                      ----- expected due to this value
3 |     spaces = spaces.len();
  |              ^^^^^^^^^^^^ expected `&str`, found `usize`
  |                            (&str型を予期しましたが、usizeが見つかりました)
  |
help: try removing the method call
  |
3 -     spaces = spaces.len();
3 +     spaces = spaces;
  |

For more information about this error, try `rustc --explain E0308`.
error: could not compile `variables` (bin "variables") due to 1 previous error
```

さあ、変数が動作する方法を見てきたので、今度は変数が取りうるデータ型について見ていきましょう。

3.2　データ型

Rust における値は全て、何らかの**データ型**になり、コンパイラがどんなデータが指定されているか知れるので、そのデータの取り扱い方も把握できるというわけです。2 種のデータ型のサブセットを見ましょう: スカラー型と複合型です。

Rust は**静的型付き**言語であることを弁えておいてください。つまり、コンパイル時に全ての変数の型が判明している必要があるということです。コンパイラは通常、値と使用方法に基づいて、使用したい型を推論してくれます。複数の型が推論される可能性がある場合、例えば、第 2 章の「予想と秘密の数字を比較する」節で `parse` メソッドを使って `String` 型を数値型に変換した時のように、複数の型が可能な場合には、型注釈をつけなければいけません。以下のようにですね:

```
let guess: u32 = "42".parse().expect("Not a number!");    // 数字ではありません！
```

上のコード中に示す `: u32` 型注釈を付けなければ、コンパイラは以下のエラーを表示し、これは可能性のある型のうち、どの型を使用したいのかを知るのに、コンパイラがプログラマからもっと情報を得る必要があることを意味します:

```
$ cargo build
   Compiling no_type_annotations v0.1.0 (file:///projects/no_type_annotations)
error[E0284]: type annotations needed
              (型注釈が必要です)
 --> src/main.rs:2:9
  |
2 |     let guess = "42".parse().expect("Not a number!");
  |         ^^^^^        ----- type must be known at this point
  |                            (型はこの時点で既知でなくてはなりません)
  |
```

```
  = note: cannot satisfy `<_ as FromStr>::Err == _`
          (`<_ as FromStr>::Err == _`を満たすことができません)
help: consider giving `guess` an explicit type
  |   (`guess`に型を与えることを検討してください)
  |
2 |     let guess: /* Type */ = "42".parse().expect("Not a number!");
  |              ++++++++++++

For more information about this error, try `rustc --explain E0284`.
error: could not compile `no_type_annotations` (bin "no_type_annotations") due
    to 1 previous error
```

他のデータ型についても、様々な型注釈を目にすることになるでしょう。

3.2.1　スカラー型

スカラー型は、単独の値を表します。Rust には主に 4 つのスカラー型があります: 整数、浮動小数点数、論理値、最後に文字です。他のプログラミング言語でも、これらの型を見かけたことはあるでしょう。Rust での動作方法に飛び込みましょう。

3.2.1.1　整数型

整数とは、小数部分のない数値のことです。第 2 章で一つの整数型を使用しましたね。`u32` 型です。この型定義は、紐付けられる値が、符号なし整数 (符号付き整数は `u` ではなく、`i` で始まります) になり、これは、32 ビット分のサイズを取ります。表 3-1 は、Rust の組み込み整数型を表示しています。これらのバリアントを使用して、整数値の型を宣言することができます。

表 3-1: Rust の整数型

大きさ	符号付き	符号なし
8-bit	`i8`	`u8`
16-bit	`i16`	`u16`
32-bit	`i32`	`u32`
64-bit	`i64`	`u64`
128-bit	`i128`	`u128`
arch	`isize`	`usize`

各バリアントは、符号付きか符号なしかを選べ、明示的なサイズを持ちます。**符号付き**と**符号なし**は、数値が負の数になり得るかどうかを示します。つまり、数値が符号を持つ必要があるかどうか (符号付き)、または、絶対に正数にしかならず符号なしで表現できるかどうか (符号なし) です。これは、数値を紙に書き下すのと似ています: 符号が問題になるなら、数値はプラス記号、またはマイナス記号とともに表示されます; しかしながら、その数値が正数であると仮定することが安全なら、符号な

しで表示できるわけです。符号付き数値は、2 の補数表現で保持されます。

各符号付きバリアントは、$-(2^{n-1})$ 以上 $2^{n-1} - 1$ 以下の数値を保持でき、ここで **n** はこのバリアントが使用するビット数です。以上から、`i8` 型は$-(2^7)$ から $2^7 - 1$ まで、つまり、-128 から 127 までを保持できます。符号なしバリアントは、0 以上 $2^n - 1$ 以下を保持できるので、`u8` 型は、0 から $2^8 - 1$ までの値、つまり、0 から 255 までを保持できることになります。

加えて、`isize` と `usize` 型は、表では「arch」と表記していますが、プログラムが動作しているコンピュータのアーキテクチャに依存します: 64 ビットアーキテクチャなら、64 ビットですし、32 ビットアーキテクチャなら、32 ビットになります。

整数リテラル (訳注：リテラルとは、見たままの値ということ) は、表 3-2 に示すどの形式でも記述することができます。複数の数値型になることができる数値リテラルは、型を指示するために型接尾辞をつけて、`57u8` のように書くことができます。数値リテラルはさらに、数値を読みやすくするために見た目の区切り記号として `_` をつけて、`1_000` のように書くこともできます。これは `1000` と指定した場合とまったく同じ値となるでしょう。

表 3-2: Rust の整数リテラル

数値リテラル	例
10 進数	`98_222`
16 進数	`0xff`
8 進数	`0o77`
2 進数	`0b1111_0000`
バイト (`u8` だけ)	`b'A'`

では、どの整数型を使うべきかはどう把握すればいいのでしょうか？ もし確信が持てないのならば、Rust の基準型は一般的にいい開始地点になります: 整数型の基準は `i32` 型です。`isize` と `usize` を使う主な状況は、何らかのコレクションにアクセスすることです。

■整数オーバーフロー　`u8` 型の変数があるとしましょう。`u8` は 0 から 255 までの間の値を取ることができます。この変数を範囲外の値、例えば 256 に変更しようとすると、**整数オーバーフロー (integer overflow)** が発生し、次の 2 つのうちのどちらかの挙動になります。デバッグモードでコンパイルしているときは、もしこの挙動が発生したときは実行時にプログラムを**パニック (panic)** させるような、整数オーバーフローのチェックをコンパイラが入れ込みます。プログラムがエラーとともに終了するとき、Rust は**パニック**という用語を使用します; パニックについては第 9 章の「`panic!` で回復不能なエラー」でより深く議論します。

`--release` フラグを付けてリリースモードでコンパイルしているときは、コンパイラはパニックを引き起こす整数オーバーフローチェックを入れ込み**ません**。代わりに、オーバーフローが

起きたときは、プログラムは **2 の補数ラップアラウンド (two's complement wrapping)** を行います。一言で言うと、その型が取ることができる最大値よりも大きい値は、その型が取ることができる最小値に「回り込む」("wrap around") のです。`u8` の場合は、値 256 は 0 になり、値 257 は 1 になり、という感じです。プログラムはパニックはしなくなるでしょうが、変数が持っている値はおそらくプログラマが期待していたものではないでしょう。整数オーバーフローのラップアラウンドの挙動に依存するのは、エラーと考えられます。

オーバーフローが発生する可能性を明示的に取り扱うためには、プリミティブ数値型に関して標準ライブラリが提供する、以下のメソッド群を使うことができます:

- `wrapping_*` メソッド（`wrapping_add` 等）で、モードを問わずラップアラウンドさせる。
- `checked_*` メソッドで、オーバーフローが発生する場合には `None` 値を返す。
- `overflowing_*` メソッドで、値と、オーバーフローが発生したかどうかを示す論理値を返す。
- `saturating_*` メソッドで、値の最小値または最大値で飽和させる。（訳注: 結果が最大値を上回る場合は最大値に、最小値を下回る場合は最小値にするという意味です）

3.2.1.2　浮動小数点型

Rust にはさらに、**浮動小数点数**に対しても、2 種類の基本型があり、浮動小数点数とは数値に小数点がついたもののことです。Rust の浮動小数点型は、`f32` と `f64` で、それぞれ 32 ビットと 64 ビットサイズです。基準型は `f64` です。なぜなら、現代の CPU では、`f32` とほぼ同スピードにもかかわらず、より精度が高くなるからです。すべての浮動小数点型は符号付きです。

実際に動作している浮動小数点数の例をご覧ください:

ファイル名: src/main.rs

```
fn main() {
    let x = 2.0; // f64

    let y: f32 = 3.0; // f32
}
```

浮動小数点数は、IEEE-754 規格に従って表現されています。`f32` が単精度浮動小数点数、`f64` が倍精度浮動小数点数です。

3.2.1.3　数値演算

Rust にも全数値型に期待されうる標準的な数学演算が用意されています: 足し算、引き算、掛け算、割り算、余りです。整数の割り算では、0 に近い方の最も近い整数に切り捨てられます。以下の例では、`let` 文での各数学演算の使用方法をご覧になれます:

ファイル名: src/main.rs

```
fn main() {
    // addition
    // 足し算
    let sum = 5 + 10;

    // subtraction
    // 引き算
    let difference = 95.5 - 4.3;

    // multiplication
    // 掛け算
    let product = 4 * 30;

    // division
    // 割り算
    let quotient = 56.7 / 32.2;
    let truncated = -5 / 3; // Results in -1
                            // 結果は-1

    // remainder
    // 余り
    let remainder = 43 % 5;
}
```

これらの文の各式は、数学演算子を使用しており、一つの値に評価され、そして、変数に束縛されます。**??**章に Rust で使える演算子の一覧が載っています。

3.2.1.4　論理値型

他の多くの言語同様、Rust の論理値型も取りうる値は二つしかありません: `true` と `false` です。論理値のサイズは 1 バイトです。Rust の論理値型は、`bool` と指定されます。例です:

ファイル名: src/main.rs

```
fn main() {
    let t = true;

    let f: bool = false; // with explicit type annotation
                         // 明示的型注釈付きで
}
```

論理値を使う主な手段は、条件式です。例えば、`if` 式などですね。`if` 式の Rust での動作方法については、「制御フロー」節で講義します。

3.2.1.5　文字型

Rust の `char` 型は、言語の最も基本的なアルファベット型です。以下は `char` 値を宣言するいくつかの例です:

ファイル名: src/main.rs

```
fn main() {
    let c = 'z';
    let z: char = 'ℤ'; // with explicit type annotation
                       // 明示的型注釈付きで
    let heart_eyed_cat = '😻';    //ハート目の猫
}
```

char リテラルは、ダブルクォーテーションマークを使用する文字列リテラルに対して、シングルクォートで指定することに注意してください。Rust の char 型は 4 バイトのサイズを持ち、ユニコードのスカラー値を表します。これはつまり、アスキーよりもずっとたくさんのものを表せるということです。アクセント記号付き文字; 中国語、日本語、韓国語の文字; 絵文字; ゼロ幅スペースはすべて、Rust では有効な char 値です。ユニコードスカラー値は、U+0000 から U+D7FF までと U+E000 から U+10FFFF までの範囲になります。ところが、「文字」は実はユニコードの概念ではないので、文字とは何かという人間としての直観は、Rust における char 値が何かとは合致しない可能性があります。この話題については、第 8 章の「文字列で UTF-8 でエンコードされたテキストを保持する」で詳しく議論しましょう。

3.2.2　複合型

複合型により、複数の値を一つの型にまとめることができます。Rust には、2 種類の基本的な複合型があります: タプルと配列です。

3.2.2.1　タプル型

タプルは、様々な型の複数の値を一つの複合型にまとめ上げる汎用的な手段です。タプルの長さは固定です: 一度宣言されたらサイズは伸縮できません。

タプルは、丸かっこの中にカンマ区切りの値リストを書くことで生成します。タプルの位置ごとに型があり、タプル内の値はそれぞれ全てが同じ型である必要はありません。今回の例では、型注釈をあえて追加しました:

ファイル名: src/main.rs

```
fn main() {
    let tup: (i32, f64, u8) = (500, 6.4, 1);
}
```

変数 tup は、タプル全体に束縛されています。なぜなら、タプルは、一つの複合要素と考えられるからです。タプルから個々の値を取り出すには、パターンマッチングを使用して分解することができます。以下のように:

ファイル名: src/main.rs

```
fn main() {
    let tup = (500, 6.4, 1);

    let (x, y, z) = tup;

    println!("The value of y is: {y}");
}
```

このプログラムは、まずタプルを生成し、それを変数 tup に束縛しています。それから let とパターンを使って tup 変数の中身を 3 つの個別の変数 (x 、y 、z ですね) に変換しています。この過程は、**分配**と呼ばれます。単独のタプルを破壊して三分割しているからです。最後に、プログラムは y 変数の値を出力し、6.4 と表示されます。

アクセスしたい値の番号をピリオド (.) に続けて書くことで、タプルの要素に直接アクセスすることもできます。例です:

ファイル名: src/main.rs

```
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let five_hundred = x.0;

    let six_point_four = x.1;

    let one = x.2;
}
```

このプログラムは、新しいタプル x を作成し、タプルの各要素にそれぞれの添え字を使ってアクセスしています。多くのプログラミング言語同様、タプルの最初の添え字は 0 です。

値をひとつも持たないタプルは**ユニット**という特別な名前を持っています。この値と、それに対応する型はともに () と書き表され、空の値や空の戻り値型を表現します。式は、特に値を返さなければ、暗黙的にユニット値を返します。

3.2.2.2　配列型

配列によっても、複数の値のコレクションを得ることができます。タプルと異なり、配列の全要素は、同じ型でなければなりません。一部の他の言語の配列と異なり、Rust の配列は固定長です。

配列内の要素は、角かっこ内にカンマ区切りリストとして記述します:

ファイル名: src/main.rs

```
fn main() {
    let a = [1, 2, 3, 4, 5];
```

```
}
```

配列は、ヒープよりもスタック (スタックとヒープについては第 4 章で詳 (つまび) らかに議論します) にデータのメモリを確保したい時、または、常に固定長の要素があることを確認したい時に有効です。ただ、配列は、ベクタ型ほど柔軟ではありません。**ベクタ**は、標準ライブラリによって提供されている配列と似たようなコレクション型で、こちらは、サイズを伸縮させることが**できます**。配列とベクタ型、どちらを使うべきか確信が持てない時は、おそらくベクタ型を使うべきです。第 8 章でベクタについて詳細に議論します。

しかしながら、要素数を変えられる必要はないだろうと分かっている場合は、配列のほうが便利です。例えば、プログラム中で月の名前を使おうとしているなら、おそらくベクタよりも配列を使うのが良いでしょう。常に 12 個要素があることもわかってますからね:

```
let months = ["January", "February", "March", "April", "May", "June", "July",
              "August", "September", "October", "November", "December"];
```

例えば次のように、配列の型は角かっこの中に要素の型とセミコロン、そして配列の要素数を与えます。

```
let a: [i32; 5] = [1, 2, 3, 4, 5];
```

ここでの `i32` は要素の型です。セミコロンのあとの `5` という数字は配列の要素が 5 つあることを表しています。

次のように、角かっこの中に初期値とセミコロン、そして配列の長さを与えることで、各要素に同じ値を持つように配列を初期化することができます。

```
let a = [3; 5];
```

この `a` という名前の配列は `3` という値が 5 つあるものです。これは `let a = [3, 3, 3, 3, 3];` と書くのと同じですが、より簡潔になります。

■配列の要素にアクセスする　配列は、あらかじめ知られた固定サイズを持ち、スタック上に確保することができる一塊のメモリです。添え字によって、配列の要素にこのようにアクセスすることができます:

ファイル名: src/main.rs

```
fn main() {
    let a = [1, 2, 3, 4, 5];

    let first = a[0];
    let second = a[1];
}
```

この例では、first という名前の変数には 1 という値が格納されます。配列の [0] 番目にある値が、それだからですね。second という名前の変数には、配列の [1] 番目の値 2 が格納されます。

■配列要素への無効なアクセス　配列の終端を越えて要素にアクセスしようとしたらどうなるか、見てみましょう。第 2 章の数当てゲームと同じようにユーザから配列の添え字を受け取る、次のコードを実行する場合を考えてみてください:

ファイル名: src/main.rs

```
use std::io;

fn main() {
    let a = [1, 2, 3, 4, 5];

    println!("Please enter an array index.");
           // 配列の何番目の要素にアクセスするか指定してください

    let mut index = String::new();

    io::stdin()
        .read_line(&mut index)
        .expect("Failed to read line");
              // 値の読み込みに失敗しました

    let index: usize = index
        .trim()
        .parse()
        .expect("Index entered was not a number");
              // 入力された値は数字ではありません

    let element = a[index];

    println!("The value of the element at index {index} is: {element}");
           // {index}番目の要素の値は{element}です
}
```

このコードはコンパイルされます。cargo run で走らせ、0，1，2，3，または 4 をこのプログラムに入力すると配列の対応する値を出力します。もし配列の末尾を超えるような、例えば 10 などの数字を与えると、次のような出力が表示されます。

```
thread 'main' panicked at src/main.rs:19:19:
index out of bounds: the len is 5 but the index is 10
(スレッド'main'はsrc/main.rs:19:19でパニックしました:
範囲外アクセス: 長さは5ですが、添え字は10でした)
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

プログラムは、添え字アクセスで無効な値を使用した時点で**実行時**エラーに陥りました。プログラ

ムはエラーメッセージとともに終了し、最後の println! 文を実行しませんでした。要素に添え字アクセスを試みると、言語は、指定されたその添え字が配列長よりも小さいかを確認してくれます。添え字が配列長と等しいかより大きければ、言語は**パニック**します。この例の場合は特にそうですが、このチェックは実行時に行われなくてはなりません。なぜならコンパイラは、ユーザが後でコードを実行したときに、ユーザがどんな値を入力するか知りようがないからです。

これは、実際に稼働している Rust のメモリ安全機構の例のひとつになります。低レベル言語の多くでは、この種のチェックは行われないため、間違った添え字を与えると、無効なメモリにアクセスできてしまいます。Rust では、メモリアクセスを許可し、処理を継続する代わりに即座にプログラムを終了することで、この種のエラーからプログラマを保護しています。第 9 章では Rust のエラー処理について、そして、可読性が高く安全で、パニックもしなければ不正なメモリアクセスも許さないコードをどうすれば書けるのか、もっと議論します。

3.3　関数

関数は、Rust のコードにおいてよく見かける存在です。既に、言語において最も重要な関数のうちの一つを目撃していますね: そう、main 関数です。これは、多くのプログラムのエントリーポイント (訳注：プログラム実行時に最初に走る関数のこと) になります。fn キーワードもすでに見かけましたね。これによって新しい関数を宣言することができます。

Rust の関数と変数の命名規則は、**スネークケース** (訳注：some_variable のような命名規則) を使うのが慣例です。スネークケースとは、全文字を小文字にし、単語区切りにアンダースコアを使うことです。以下のプログラムで、サンプルの関数定義をご覧ください:

ファイル名: src/main.rs

```
fn main() {
    println!("Hello, world!");

    another_function();
}

fn another_function() {
    println!("Another function.");  // 別の関数
}
```

Rust では、fn に続けて関数名と丸かっこの組を入力して関数を定義します。波かっこが、コンパイラに関数本体の開始と終了の位置を伝えます。

定義した関数は、名前に丸かっこの組を続けることで呼び出すことができます。another_function 関数がプログラム内で定義されているので、main 関数内から呼び出すことができるわけです。ソースコード中で another_function を main 関数の**後**に定義していることに注目してください; 勿論、main 関数の前に定義することもできます。コンパイラは、関数がどこで定義されているかは気にしません。

呼び出し元から見えるスコープ内のどこかで定義されていることのみ気にします。

functions という名前の新しいバイナリ生成プロジェクトを始めて、関数についてさらに深く探究していきましょう。another_function の例を **src/main.rs** ファイルに配置して、走らせてください。以下のような出力が得られるはずです:

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
    Finished dev [unoptimized + debuginfo] target(s) in 0.28s
     Running `target/debug/functions`
Hello, world!
Another function.
```

行出力は、main 関数内に書かれた順序で実行されています。最初に”Hello, world” メッセージが出て、それから another_function が呼ばれて、こちらのメッセージが出力されています。

3.3.1　引数

関数は、**仮引数 (parameter)** を持つよう定義することもできます。仮引数とは、関数シグニチャの一部になる特別な変数のことです。関数に仮引数があると、仮引数に対して具体的な値を与えることができます。厳密にはこの具体的な値は**実引数 (argument)** と呼ばれますが、普段の会話では、関数定義内の変数と関数呼び出し時に渡す実際の値の両方の意味で、**parameter** と **argument** を区別なく使う傾向にあります (訳注：日本語では、特別区別する意図がない限り、どちらも単に**引数**と呼ぶことが多いでしょう)。

次の版の another_function では、仮引数を追加しています:

ファイル名: src/main.rs

```
fn main() {
    another_function(5);
}

fn another_function(x: i32) {
    println!("The value of x is: {x}");   // xの値は{x}です
}
```

このプログラムを走らせてみてください; 以下のような出力が得られるはずです:

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
    Finished dev [unoptimized + debuginfo] target(s) in 1.21s
     Running `target/debug/functions`
The value of x is: 5
```

another_function の宣言には、x という名前の仮引数があります。x の型は、i32 と指定されています。値 5 を another_function に渡すと、println! マクロにより、フォーマット文字列中の x を含

む 1 組の波かっこがあった位置に値 5 が出力されます。

関数シグニチャにおいて、各仮引数の型を宣言しなければ**なりません**。これは、Rust の設計において、意図的な判断です: 関数定義で型注釈が必要不可欠ということは、コンパイラがその意図する型を推し量るのに、プログラマがコードの他の箇所で使用する必要がないということを意味します。コンパイラも、関数が期待する型を知っていれば、より役に立つエラーメッセージを与えることができます。

関数に複数の仮引数を定義したいときは、仮引数定義をカンマで区切ってください。こんな感じです:

ファイル名: src/main.rs

```
fn main() {
    print_labeled_measurement(5, 'h');
}

fn print_labeled_measurement(value: i32, unit_label: char) {
    println!("The measurement is: {value}{unit_label}");
}
```

この例では、`print_labeled_measurement` という名前の 2 引数の関数を生成しています。第 1 引数は `value` という名前で `i32` です。第 2 引数は `unit_label` という名前で `char` 型です。この関数は `value` と `unit_label` の両方を含むテキストを出力します。

このコードを走らせてみましょう。今、**function** プロジェクトの **src/main.rs** ファイルに記載されているプログラムを先ほどの例と置き換えて、`cargo run` で走らせてください:

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/functions`
The measurement is: 5h
```

`value` に対して値 `5`、`unit_label` に対して値 `'h'` を渡して関数を呼び出したので、プログラムの出力にはこれらの値が含まれます。

3.3.2　文と式

関数本体は、文が並び、最後に式を置くか文を置くという形で形成されます。今のところ、私たちが見てきた関数は式で終わることはありませんでしたが、式が文の一部になっているものなら見かけましたね。Rust は、式指向言語なので、これは理解しておくべき重要な差異になります。他の言語にこの差異はありませんので、文と式がなんなのかと、その違いが関数本体にどんな影響を与えるかを見ていきましょう。

- **文 (statement)** はなんらかの動作をして値を返さない命令です。

- **式 (expression)** は結果値に評価されます。ちょっと例を眺めてみましょう。

実のところ、もう文と式は使っています。let キーワードを使用して変数を生成し、値を代入することは文になります。リスト 3-1 で let y = 6; は文です。

ファイル名: src/main.rs

```
fn main() {
    let y = 6;
}
```

リスト 3-1: 1 文を含む main 関数宣言

関数定義も文になります。つまり、先の例は全体としても文になるわけです。
文は値を返しません。故に、let 文を他の変数に代入することはできません。以下のコードではそれを試みていますが、エラーになります:

ファイル名: src/main.rs

```
fn main() {
    let x = (let y = 6);
}
```

このプログラムを実行すると、以下のようなエラーが出るでしょう:

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
error: expected expression, found `let` statement
(エラー: 式を予期しましたが、`let`文が見つかりました)
 --> src/main.rs:2:14
  |
2 |     let x = (let y = 6);
  |              ^^^
  |
  = note: only supported directly in conditions of `if` and `while` expressions
    (注釈: `if` および `while` 式の条件部直下でのみ対応しています

warning: unnecessary parentheses around assigned value
 --> src/main.rs:2:13
  |
2 |     let x = (let y = 6);
  |             ^         ^
  |
  = note: `#[warn(unused_parens)]` on by default
help: remove these parentheses
  |
2 -     let x = (let y = 6);
2 +     let x = let y = 6;
  |
```

```
warning: `functions` (bin "functions") generated 1 warning
error: could not compile `functions` (bin "functions") due to 1 previous error;
    1 warning emitted
```

この `let y = 6` という文は値を返さないので、`x` に束縛するものがないわけです。これは、C や Ruby などの言語とは異なる動作です。C や Ruby では、代入は代入値を返します。これらの言語では、`x = y = 6` と書いて、`x` も `y` も値 6 になるようにできるのですが、Rust においては、そうは問屋が卸さないわけです。

式は値に評価され、これからあなたが書く Rust コードの多くを構成します。数学演算 (`5 + 6` など) を思い浮かべましょう。この例は、値 `11` に評価される式です。式は文の一部になりえます: リスト 3-1 において、`let y = 6` という文の `6` は値 `6` に評価される式です。関数呼び出しも式です。マクロ呼び出しも式です。波括弧で作られる新しいスコープも式です:

ファイル名: src/main.rs

```
fn main() {
    let y = {
        let x = 3;
        x + 1
    };

    println!("The value of y is: {y}");
}
```

以下の式:

```
{
    let x = 3;
    x + 1
}
```

は今回の場合、`4` に評価されるブロックです。その値が、`let` 文の一部として `y` に束縛されます。今まで見かけてきた行と異なり、`x + 1` の行には文末にセミコロンがついていないことに気をつけてください。式は終端にセミコロンを含みません。式の終端にセミコロンを付けたら、文に変えてしまいます。そして、文は値を返しません。次に関数の戻り値や式を見ていく際にこのことを肝に銘じておいてください。

3.3.3　戻り値のある関数

関数は、それを呼び出したコードに値を返すことができます。戻り値に名前を付けはしませんが、矢印 (`->`) の後に型を書いて宣言する必要があります。Rust では、関数の戻り値は、関数本体ブロックの最後の式の値と同義です。`return` キーワードで関数から早期リターンし、値を指定することもで

きますが、多くの関数は最後の式を暗黙的に返します。こちらが、値を返す関数の例です:

ファイル名: src/main.rs

```
fn five() -> i32 {
    5
}

fn main() {
    let x = five();

    println!("The value of x is: {x}");
}
```

`five` 関数内には、関数呼び出しもマクロ呼び出しも、`let` 文でさえ存在しません。数字の 5 が単独であるだけです。これは、Rust において、完璧に問題ない関数です。関数の戻り値型が `-> i32` と指定されていることにも注目してください。このコードを実行してみましょう; 出力はこんな感じになるはずです:

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
    Finished dev [unoptimized + debuginfo] target(s) in 0.30s
     Running `target/debug/functions`
The value of x is: 5
```

`five` 内の `5` が関数の戻り値です。だから、戻り値型が `i32` なのです。これについてもっと深く考察しましょう。重要な箇所は 2 つあります: まず、`let x = five()` という行は、関数の戻り値を使って変数を初期化していることを示しています。関数 `five` は `5` を返すので、この行は以下のように書くのと同義です:

```
let x = 5;
```

2 番目に、`five` 関数は仮引数をもたず、戻り値型を定義していますが、関数本体はセミコロンなしの `5` 単独です。なぜなら、これが返したい値になる式だからです。

もう一つ別の例を見ましょう:

ファイル名: src/main.rs

```
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {x}");
}

fn plus_one(x: i32) -> i32 {
    x + 1
}
```

このコードを走らせると、`The value of x is: 6` と出力されるでしょう。しかし、`x + 1` を含む行の終端にセミコロンを付けて、式から文に変えたら、エラーになるでしょう:

ファイル名: src/main.rs

```
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {x}");
}

fn plus_one(x: i32) -> i32 {
    x + 1;
}
```

このコードをコンパイルすると、以下のようにエラーが出ます:

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
error[E0308]: mismatched types
              (型が合いません)
 --> src/main.rs:7:24
  |
7 | fn plus_one(x: i32) -> i32 {
  |    --------            ^^^ expected `i32`, found `()`
  |    |
  |    implicitly returns `()` as its body has no tail or `return` expression
8 |     x + 1;
  |          - help: remove this semicolon to return this value

For more information about this error, try `rustc --explain E0308`.
error: could not compile `functions` (bin "functions") due to 1 previous error
```

メインのエラーメッセージである `mismatched types (型が合いません)` でこのコードの根本的な問題が明らかになるでしょう。関数 `plus_one` の定義では、`i32` 型を返すと言っているのに、文は値に評価されないからです。このことは、`()`、つまりユニット型として表現されています。それゆえに、何も戻り値がなく、これが関数定義と矛盾するので、結果としてエラーになるわけです。この出力内で、コンパイラは問題を修正する手助けになりそうなメッセージも出していますね: セミコロンを削除するよう提言しています。そして、そうすれば、エラーは直るわけです。

3.4　コメント

全プログラマは、自分のコードがわかりやすくなるよう努めますが、時として追加の説明が許されることもあります。このような場合、プログラマは**コメント**をソースコードに残し、コメントをコン

パイラは無視しますが、ソースコードを読む人間には有益なものと思えるでしょう。

こちらが単純なコメントです:

```
// hello, world
```

Rust の慣用的なコメントスタイルでは、コメントは 2 連スラッシュで始め、行の終わりまで続きます。コメントが複数行にまたがる場合、各行に `//` を含める必要があります。こんな感じに:

```
// So we're doing something complicated here, long enough that we need
// multiple lines of comments to do it! Whew! Hopefully, this comment will
// explain what's going on.
// ここで何か複雑なことをしていて、長すぎるから複数行のコメントが必要なんだ。
// ふう！ 願わくば、このコメントで何が起きているか説明されていると嬉しい。
```

コメントは、コードが書かれた行の末尾にも配置することができます:

ファイル名: src/main.rs

```
fn main() {
    let lucky_number = 7; // I'm feeling lucky today(今日はラッキーな気がするよ
    )
}
```

しかし、こちらの形式のコメントの方が見かける機会は多いでしょう。注釈しようとしているコードの 1 行上に書く形式です:

ファイル名: src/main.rs

```
fn main() {
    // I'm feeling lucky today
    // 今日はラッキーな気がするよ
    let lucky_number = 7;
}
```

Rust には他の種類のコメント、ドキュメントコメントもあり、それについては第 14 章の「Crates.io にクレートを公開する」節で議論します。

3.5　制御フロー

条件が `true` かどうかによってコードを走らせたり、条件が `true` の間繰り返しコードを走らせたりできることは、多くのプログラミング言語において、基本的な構成ブロックです。Rust コードの実行フローを制御する最も一般的な文法要素は、`if` 式とループです。

3.5.1　if 式

if 式によって、条件に依存して枝分かれをさせることができます。条件を与え、以下のように宣言します。「もし条件が合ったら、この一連のコードを実行しろ。条件に合わなければ、この一連のコードは実行するな」と。

projects ディレクトリに **branches** という名のプロジェクトを作って if 式について掘り下げていきましょう。**src/main.rs** ファイルに、以下のように入力してください:

ファイル名: src/main.rs

```
fn main() {
    let number = 3;

    if number < 5 {
        println!("condition was true");        // 条件は真でした
    } else {
        println!("condition was false");       // 条件は偽でした
    }
}
```

if 式は全て、キーワードの if から始め、条件式を続けます。今回の場合、条件式は変数 number が 5 未満の値になっているかどうかをチェックします。条件が true の時に実行する一連のコードを条件式の直後に波かっこで包んで配置します。if 式の条件式と紐付けられる一連のコードは、時として**アーム**と呼ばれることがあります。第 2 章の「予想と秘密の数字を比較する」の節で議論した match 式のアームと同じです。

オプションとして、else 式を含むこともでき (ここではそうしています)、これによりプログラムは、条件式が偽になった時に実行するコードを与えられることになります。仮に、else 式を与えずに条件式が偽になったら、プログラムは単に if ブロックを飛ばして次のコードを実行しにいきます。

このコードを走らせてみましょう; 以下のような出力を目の当たりにするはずです:

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/branches`
condition was true
```

number の値を条件が false になるような値に変更してどうなるか確かめてみましょう:

```
# fn main() {
    let number = 7;
#
#     if number < 5 {
#         println!("condition was true");
#     } else {
```

```
#         println!("condition was false");
#     }
# }
```

再度プログラムを実行して、出力に注目してください:

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/branches`
condition was false
```

このコード内の条件式は、bool 型で**なければならない**ことにも触れる価値があります。条件式が、bool 型でない時は、エラーになります。例えば、試しに以下のコードを実行してみてください:

ファイル名: src/main.rs

```
fn main() {
    let number = 3;

    if number {
        println!("number was three");      // 数値は3です
    }
}
```

今回、if の条件式は 3 という値に評価され、コンパイラがエラーを投げます:

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
error[E0308]: mismatched types
              (型が合いません)
 --> src/main.rs:4:8
  |
4 |     if number {
  |        ^^^^^^ expected `bool`, found integer
  |                (bool型を予期したのに、整数変数が見つかりました)

For more information about this error, try `rustc --explain E0308`.
error: could not compile `branches` (bin "branches") due to 1 previous error
```

このエラーは、コンパイラは bool 型を予期していたのに、整数だったことを示唆しています。Ruby や JavaScript などの言語とは異なり、Rust では、論理値以外の値が、自動的に論理値に変換されることはありません。明示し、必ず if には条件式として、論理値 を与えなければなりません。例えば、数値が 0 以外の時だけ if のコードを走らせたいなら、以下のように if 式を変更することができます:

ファイル名: src/main.rs

```
fn main() {
```

```
    let number = 3;

    if number != 0 {
        println!("number was something other than zero");   // 数値は0以外の何かです
    }
}
```

このコードを実行したら、number was something other than zero と表示されるでしょう。

3.5.1.1 else if で複数の条件を扱う

if と else を組み合わせて else if 式にすることで複数の条件を使うこともできます。例です:

ファイル名: src/main.rs

```
fn main() {
    let number = 6;

    if number % 4 == 0 {
        // 数値は4で割り切れます
        println!("number is divisible by 4");
    } else if number % 3 == 0 {
        // 数値は3で割り切れます
        println!("number is divisible by 3");
    } else if number % 2 == 0 {
        // 数値は2で割り切れます
        println!("number is divisible by 2");
    } else {
        // 数値は4、3、2で割り切れません
        println!("number is not divisible by 4, 3, or 2");
    }
}
```

このプログラムには、通り道が 4 つあります。実行後、以下のような出力を目の当たりにするはずです:

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/branches`
number is divisible by 3
```

このプログラムを実行すると、if 式が順番に吟味され、最初に条件が true に評価された本体が実行されます。6 は 2 で割り切れるものの、number is divisible by 2 や、else ブロックの number is not divisible by 4, 3, or 2 という出力はされないことに注目してください。それは、Rust が最初の true な条件のブロックのみを実行し、条件に合ったものが見つかったら、残りはチェックすらしないからです。

else if 式を使いすぎると、コードがめちゃくちゃになってしまうので、1 つ以上あるなら、コードをリファクタリングしたくなるかもしれません。これらのケースに有用な match と呼ばれる、強力な Rust の枝分かれ文法要素については第 6 章で解説します。

3.5.1.2　let 文内で if 式を使う

if は式なので、let 文の右辺に持ってきて結果を変数に代入することができます。リスト 3-2 のようにですね。

ファイル名: src/main.rs

```
fn main() {
    let condition = true;
    let number = if condition { 5 } else { 6 };

    // numberの値は、{number}です
    println!("The value of number is: {number}");
}
```

リスト 3-2: if 式の結果を変数に代入する

この number 変数は、if 式の結果に基づいた値に束縛されます。このコードを走らせてどうなるか確かめてください:

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
    Finished dev [unoptimized + debuginfo] target(s) in 0.30s
     Running `target/debug/branches`
The value of number is: 5
```

一連のコードは、そのうちの最後の式に評価され、数値はそれ単独でも式になることを思い出してください。今回の場合、この if 式全体の値は、どのブロックのコードが実行されるかに基づきます。これはつまり、if の各アームの結果になる可能性がある値は、同じ型でなければならないということになります; リスト 3-2 で、if アームも else アームも結果は、i32 の整数でした。以下の例のように、型が合わない時には、エラーになるでしょう:

ファイル名: src/main.rs

```
fn main() {
    let condition = true;

    let number = if condition { 5 } else { "six" };

    println!("The value of number is: {number}");
}
```

このコードをコンパイルしようとすると、エラーになります。`if` と `else` アームは互換性のない値の型になり、コンパイラがプログラム内で問題の見つかった箇所をズバリ指摘してくれます:

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
error[E0308]: `if` and `else` have incompatible types
              (ifとelseの型に互換性がありません)
 --> src/main.rs:4:44
  |
4 |     let number = if condition { 5 } else { "six" };
  |                                -          ^^^^^ expected integer, found `&
      str`
  |                                                (整数変数を予期しましたが、&
      strが見つかりました)
  |                                |
  |                                expected because of this

For more information about this error, try `rustc --explain E0308`.
error: could not compile `branches` (bin "branches") due to 1 previous error
```

`if` ブロックの式は整数に評価され、`else` ブロックの式は文字列に評価されます。これでは動作しません。変数は単独の型でなければならず、コンパイラは、コンパイル時に `number` 変数の型を確実に把握する必要があるからです。`number` の型を把握していることで、コンパイラは、コンパイル時に `number` が使われている箇所全部で型が有効であるか検証できるのです。`number` の型が実行時にしか決まらないのであれば、コンパイラはそれを実行することができなくなってしまいます; どの変数に対しても、架空の複数の型があることを追いかけなければならないのであれば、コンパイラはより複雑になり、コードに対して行える保証が少なくなってしまうでしょう。

3.5.2　ループでの繰り返し

一連のコードを 1 回以上実行できると、しばしば役に立ちます。この作業用に、Rust にはいくつかの**ループ**が用意されています。ループは、本体内のコードを最後まで実行し、直後にまた最初から処理を開始します。ループを試してみるのに、**loops** という名の新プロジェクトを作りましょう。

Rust には 3 種類のループが存在します: `loop` と `while` と `for` です。それぞれ試してみましょう。

3.5.2.1　`loop` でコードを繰り返す

`loop` キーワードを使用すると、同じコードを何回も何回も永遠に、明示的にやめさせるまで実行します。

例として、**loops** ディレクトリの **src/main.rs** ファイルを以下のような感じに書き換えてください:

ファイル名: src/main.rs

```
fn main() {
```

```
    loop {
        println!("again!");   // また
    }
}
```

このプログラムを実行すると、プログラムを手動で止めるまで、何度も何度も続けて again! と出力するでしょう。ほとんどの端末で ctrl-c というショートカットが使え、永久ループに囚われてしまったプログラムに割り込むことができます。試しにやってみましょう:

```
$ cargo run
   Compiling loops v0.1.0 (file:///projects/loops)
    Finished dev [unoptimized + debuginfo] target(s) in 0.29s
     Running `target/debug/loops`
again!
again!
again!
again!
^Cagain!
```

^C という記号が出た場所が、ctrl-c を押した場所です。^C の後には again! と表示されたり、されなかったりします。割り込みシグナルをコードが受け取った時にループのどこにいたかによります。

幸いなことに、Rust にはコードによってループを抜け出す手段もあります。ループ内に break キーワードを配置することで、プログラムに実行を終了すべきタイミングを教えることができます。第 2 章の「正しい予想をした後に終了する」節の数当てゲーム内でこれをして、ユーザが予想を的中させ、ゲームに勝った時にプログラムを終了させたことを思い出してください。

数当てゲームで continue を使用しました。continue はループの中で残っているコードをスキップして次のループに移るためのものです。

3.5.2.2　ループから値を返す

loop の使用法のひとつとして、失敗するかもしれないと分かっている操作、例えばスレッドがそのジョブを完了したかを確認する操作などを、リトライするというのがあります。さらに、コードの他の部分で使うために、ループの外に操作の結果を渡す必要があるかもしれません。これを行うには、ループを止めるために使っている break 式の後ろに、返したい値を付け加えてください; その値はループを抜けて返され、使うことができます。このように:

```
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };
```

```
    println!("The result is {result}");
}
```

ループの前で counter という変数を宣言し、0 に初期化しています。次に、ループから返された値を保持するために、result という変数を宣言しています。ループの繰り返しごとに、counter 変数に 1 を追加し、counter が 10 に等しいかチェックします。等しい場合は、値 counter * 2 とともに break キーワードを使用しています。ループの後には、値を result に代入する文を終了するためのセミコロンが使われています。最後に、result の値を出力していて、この場合は 20 です。

3.5.2.3　複数のループを区別するループラベル

ループ内にループがある場合、break と continue は最も内側のループに適用されます。**ループラベル**を使用することで、break や continue が適用されるループを指定することができます。ループラベルはシングルクオートで始める必要があります。以下に例を示します。

```
fn main() {
    let mut count = 0;
    'counting_up: loop {
        println!("count = {count}");
        let mut remaining = 10;

        loop {
            println!("remaining = {remaining}");
            if remaining == 9 {
                break;
            }
            if count == 2 {
                break 'counting_up;
            }
            remaining -= 1;
        }

        count += 1;
    }
    println!("End count = {count}");
}
```

外側のループには 'counting_up というラベルがついていて、0 から 2 まで数え上げます。内側のラベルのないループは 10 から 9 までカウントダウンします。最初のラベルの無い break は内側のループを終了させます。break 'counting_up; は外側のループを終了させます。このコードは以下のような出力をします。

```
$ cargo run
   Compiling loops v0.1.0 (file:///projects/loops)
    Finished dev [unoptimized + debuginfo] target(s) in 0.58s
     Running `target/debug/loops`
```

```
count = 0
remaining = 10
remaining = 9
count = 1
remaining = 10
remaining = 9
count = 2
remaining = 10
End count = 2
```

3.5.2.4　while で条件付きループ

プログラムは、ループ内で条件式を評価することがよく必要になるでしょう。条件が true の間、ループが走るわけです。条件が true でなくなった時にプログラムは break を呼び出し、ループを終了します。このような挙動は、loop 、if 、else 、break を組み合わせることでも実装できます; お望みなら、プログラムで今、試してみるのもいいでしょう。しかし、このパターンは頻出するので、Rust にはそれ用の文法要素が用意されていて、while ループと呼ばれます。リスト 3-3 では、プログラムを 3 回ループさせるために while を使用しています。繰り返しごとにカウントダウンして、ループ後にメッセージを表示して終了します。

ファイル名: src/main.rs

```
fn main() {
    let mut number = 3;

    while number != 0 {
        println!("{number}!");

        number -= 1;
    }

    // 発射！
    println!("LIFTOFF!!!");
}
```

リスト 3-3: 条件が真の間、コードを走らせる while ループを使用する

この文法要素により、loop 、if 、else 、break を使った時に必要になるネストがなくなり、より明確になります。条件が true に評価される間、コードは実行されます; そうでなければ、ループを抜けます.

3.5.2.5　for でコレクションを覗き見る

配列などのコレクションの要素を覗き見るために、while 要素を使うこともできます。例えば、リスト 3-4 のループは配列 a の各要素を出力します。

ファイル名: src/main.rs

```
fn main() {
    let a = [10, 20, 30, 40, 50];
    let mut index = 0;

    while index < 5 {
        // 値は{}です
        println!("the value is: {}", a[index]);

        index += 1;
    }
}
```

リスト 3-4: `while` ループでコレクションの各要素を覗き見る

ここで、コードは配列の要素を順番にカウントアップして覗いています。番号 0 から始まり、配列の最終番号に到達するまでループします (つまり、`index < 5` が `true` でなくなる時です)。このコードを走らせると、配列内の全要素が出力されます:

```
$ cargo run
   Compiling loops v0.1.0 (file:///projects/loops)
    Finished dev [unoptimized + debuginfo] target(s) in 0.32s
     Running `target/debug/loops`
the value is: 10
the value is: 20
the value is: 30
the value is: 40
the value is: 50
```

予想通り、配列の 5 つの要素が全てターミナルに出力されています。`index` 変数の値はどこかで `5` という値になるものの、配列から 6 番目の値を拾おうとする前にループは実行を終了します。

しかし、このアプローチは間違いが発生しやすいです; 添え字の値や判定条件が間違っていれば、プログラムはパニックしてしまいます。例えば、`a` 配列の定義を 4 要素を持つように変更したのに、条件を `while index < 4` に更新し忘れた場合、コードはパニックするでしょう。また遅いです。実行時にループの各回ごとに添字が配列の境界内にあるかチェックするコードを、コンパイラが追加するからです。

より簡潔な対立案として、`for` ループを使ってコレクションの各アイテムに対してコードを実行することができます。`for` ループはリスト 3-5 のコードのようになります。

ファイル名: src/main.rs

```
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a {
```

```
        println!("the value is: {element}");
    }
}
```

リスト 3-5: for ループを使ってコレクションの各要素を覗き見る

このコードを走らせたら、リスト 3-4 と同じ出力が得られるでしょう。より重要なのは、コードの安全性を向上させ、配列の終端を超えてアクセスしたり、終端に届く前にループを終えてアイテムを見逃してしまったりするバグの可能性を完全に排除したことです。

for ループを使っていれば、配列の要素数を変えても、リスト 3-4 で使った方法のように他のコードをいじることを覚えておく必要はなくなるわけです。

for ループのこの安全性と簡潔性により、Rust で使用頻度の最も高いループになっています。リスト 3-3 で while ループを使ったカウントダウンサンプルのように、一定の回数、同じコードを実行したいような状況であっても、多くの Rustacean は、for ループを使うでしょう。どうやってやるかといえば、標準ライブラリで提供される Range 型を使うのです。Range 型は、片方の数字から始まって、もう片方の数字未満の数値を順番に生成する型です。

for ループと、まだ話していない別のメソッド rev を使って範囲を逆順にしたカウントダウンはこうなります:

ファイル名: src/main.rs

```
fn main() {
    for number in (1..4).rev() {
        println!("{number}!");
    }
    println!("LIFTOFF!!!");
}
```

こちらのコードの方が少しいいでしょう？

3.6　まとめ

やりましたね！ 結構長い章でした: 変数、スカラー値と複合データ型、関数、コメント、if 式、そして、ループについて学びました！ この章で議論した概念について経験を積むために、以下のことをするプログラムを組んでみてください:

- 温度を華氏と摂氏で変換する。
- フィボナッチ数列の **n** 番目を生成する。
- クリスマスキャロルの定番、”The Twelve Days of Christmas” の歌詞を、曲の反復性を利用して出力する。

次に進む準備ができたら、他の言語にはあまり存在**しない** Rust の概念について話しましょう: 所有

権です。

第4章

所有権を理解する

所有権はRustの最もユニークな機能であり、それ以外の言語機能と深く関連しています。所有権のおかげで、Rustはガベージコレクタの必要なしにメモリ安全性を担保することができていて、そのため所有権の仕組みを理解しておくことが重要です。この章では、所有権以外にも、関連する機能をいくつか話していきます: 借用、スライス、そして、コンパイラがデータをメモリにどう配置するかです。

4.1 所有権とは？

所有権とは、Rustプログラムがメモリをどのように管理するかを支配している、規則の集合です。

全てのプログラムは、実行中にコンピュータのメモリの使用方法を管理する必要があります。プログラムが動作するにつれて、定期的に使用されていないメモリを検索するガベージコレクションを持つ言語もありますが、他の言語では、プログラマが明示的にメモリを確保したり、解放したりしなければなりません。Rustでは第3の選択肢を取っています: メモリは、コンパイラがチェックする一定の規則とともに所有権システムを通じて管理されています。規則のうちどれかひとつにでも違反すれば、プログラムはコンパイルできないでしょう。どの所有権機能も、実行中にプログラムの動作を遅くすることはないでしょう。

所有権は多くのプログラマにとって新しい概念なので、慣れるまでに時間がかかります。嬉しいことに、Rustと、所有権システムの規則の経験を積んでいくにつれて、自然に安全かつ効率的なコードを構築するのはだんだん簡単だと思えるようになるでしょう。その調子でいきましょう！

所有権を理解した時、Rustを際立たせる機能の理解に対する強固な礎を得ることになるでしょう。この章では、非常に一般的なデータ構造に着目した例を取り扱うことで所有権を学んでいきます: 文字列です。

4.1.1　スタックとヒープ

多くのプログラミング言語において、スタックとヒープについて考える機会はそう多くないでしょう。しかし、Rust のようなシステムプログラミング言語においては、値がスタックに積まれるかヒープに置かれるかは、言語の振る舞いや、特定の決断を下す理由などに影響を与えるのです。この章の後半でスタックとヒープを交えて所有権の一部が解説されるので、ここでちょっと予行演習をしておきましょう。

スタックもヒープも、実行時にコードが使用できるメモリの一部になりますが、異なる手段で構成されています。スタックは、得た順番に値を並べ、逆の順で値を取り除いていきます。これは、**last in, first out(** 訳注：あえて日本語にするなら、「最後に入れたものが最初に出てくる」といったところでしょうか**)** と呼ばれます。お皿の山を思い浮かべてください: お皿を追加する時には、山の一番上に置き、お皿が必要になったら、一番上から 1 枚を取り去りますよね。途中や一番下に追加したり、取り除いたりすることもできません。データを追加することは、**スタックに push する**といい、データを取り除くことは、**スタックから pop する**と表現します **(** 訳注：日本語では単純に英語をそのまま活用してプッシュ、ポップと表現するでしょう**)**。スタック上のデータは全て既知の固定サイズでなければなりません。コンパイル時にサイズがわからなかったり、サイズが可変のデータについては、代わりにヒープに格納しなければなりません。

ヒープは、もっとごちゃごちゃしています: ヒープにデータを置く時、あるサイズのスペースを求めます。メモリアロケータはヒープ上に十分な大きさの空の領域を見つけ、使用中にし、**ポインタ**を返します。ポインタとは、その場所へのアドレスです。この過程は、**ヒープに領域を確保する (allocating on the heap)** と呼ばれ、時としてそのフレーズを単に **allocate する**などと省略したりします **(** 訳注：こちらもこなれた日本語訳はないでしょう。allocate は「メモリを確保する」と訳したいところですが**) (**スタックに値を積むことは、メモリ確保とは考えられません**)**。ヒープへのポインタは、既知の固定サイズなので、スタックに保管することができますが、実データが必要になったら、ポインタを追いかける必要があります。レストランで席を確保することを考えましょう。入店したら、グループの人数を告げ、店員が全員座れる空いている席を探し、そこまで誘導します。もしグループの誰かが遅れて来るのなら、着いた席の場所を尋ねてあなたを発見することができます。

スタックにデータを積むほうが、ヒープ上にメモリを確保するより高速です。アロケータが新しいデータを置くための場所を探さなくていいからです。というのも、その場所は常にスタックの一番上だからですね。それと比べて、ヒープ上にメモリを確保するのはより手間の多い仕事です。アロケータはまずデータを入れるのに充分な大きさの場所を探して、そして次のメモリ確保に備えて予約をしないといけないからです。

ヒープへのデータアクセスは、スタックのデータへのアクセスよりも低速です。ポインタを追って目的の場所に到達しなければならないからです。現代のプロセッサは、メモリをあちこ

ち行き来しなければ、より速くなります。似た例えを続けましょう。レストランで多くのテーブルから注文を受ける給仕人を考えましょう。最も効率的なのは、次のテーブルに移らずに、一つのテーブルで全部の注文を受け付けてしまうことです。テーブル A で注文を受け、それからテーブル B の注文、さらにまた A、それからまた B と渡り歩くのは、かなり低速な過程になってしまうでしょう。同じ意味で、プロセッサは、データが隔離されている (ヒープではそうなっている可能性がある) よりも近くにある (スタックではこうなる) ほうが、仕事をうまくこなせるのです。

コードが関数を呼び出すと、関数に渡された値 (ヒープのデータへのポインタも含まれる可能性あり) と、関数のローカル変数がスタックに載ります。関数の実行が終了すると、それらの値はスタックから取り除かれます。

どの部分のコードがどのヒープ上のデータを使用しているか把握すること、ヒープ上の重複するデータを最小化すること、メモリ不足にならないようにヒープ上の未使用のデータを掃除することは全て、所有権が解決する問題です。一度所有権を理解したら、あまり頻繁にスタックとヒープに関して考える必要はなくなるでしょうが、所有権の主な目的はヒープデータを管理することだと知っていると、所有権がどうしてこうなっているのか説明するのに役立つこともあります。

4.1.2　所有権規則

まず、所有権の規則について見ていきましょう。この規則を具体化する例を扱っていく間もこれらの規則を肝に銘じておいてください:

- Rust の各値には、**所有者**が存在する。
- いかなる時も所有者は一つである。
- 所有者がスコープから外れたら、値は破棄される。

4.1.3　変数スコープ

もう基本的な Rust の文法は通り過ぎたので、`fn main() {` というコードはもう例に含みません。従って、例をなぞっているなら、これからの例は `main` 関数に手動で入れ込むようにしてください。結果的に、例は少々簡潔になり、定型コードよりも具体的な詳細に集中しやすくなります。

所有権の最初の例として、何らかの変数の**スコープ**について見ていきましょう。スコープとは、要素が有効になるプログラム内の範囲のことです。以下の変数を例に取ります:

```
let s = "hello";
```

変数 `s` は、文字列リテラルを参照し、ここでは、文字列の値はプログラムのテキストとしてハードコードされています。この変数は、宣言された地点から、現在の**スコープ**の終わりまで有効になりま

す。リスト 4-1 には、変数 s が有効な場所に関する注釈がコメントで付記されたプログラムを示します。

```
# fn main() {
    {                      // sは、ここでは有効ではない。まだ宣言されていない
        let s = "hello";   // sは、ここから有効になる

        // sで作業をする
    }                      // このスコープは終わり。もうsは有効ではない
# }
```

リスト 4-1: 変数と有効なスコープ

言い換えると、ここまでに重要な点は二つあります:

- s がスコープに**入る**と、有効になる。
- スコープを**抜ける**まで、有効なまま。

今のところ、スコープと、変数が有効である期間の関係は、他の言語のそれに類似しています。さて、この理解のもとに、String 型を導入して構築していきましょう。

4.1.4　String 型

所有権の規則を具体化するには、第 3 章の「データ型」節で講義したものよりも、より複雑なデータ型が必要になります。以前講義した型は、既知のサイズを持ち、スタックに置いておいてスコープが終わるとスタックから取り除かれるという扱いが可能でした。さらに、コードの別の場所で同じ値を異なるスコープを持たせて使う必要があるなら、新しい、独立したインスタンスに、高速に、そして自明にコピーすることができました。が、ヒープに確保されるデータ型を観察して、コンパイラがどうそのデータを掃除すべきタイミングを把握しているかを掘り下げていきたいと思います。String 型はその好例です。

String 型の所有権にまつわる部分に着目しましょう。またこの観点は、標準ライブラリから提供されたか自分で作成したかを問わず、他の複雑なデータ型にも適用されます。String 型については、第 8 章でより深く議論します。

既に文字列リテラルは見かけましたね。文字列リテラルでは、文字列の値はプログラムにハードコードされます。文字列リテラルは便利ですが、テキストを使いたいかもしれない場面全てに最適なわけではありません。一因は、文字列リテラルが不変であることに起因します。別の原因は、コードを書く際に、全ての文字列値が判明するわけではないからです: 例えば、ユーザ入力を受け付け、それを保持したいとしたらどうでしょうか？ このような場面用に、Rust には、2 種類目の文字列型、String 型があります。この型はヒープにメモリを確保するので、コンパイル時にはサイズが不明なテキストも保持することができるのです。from 関数を使用して、文字列リテラルから String 型を生成できます。以下のように:

```
let s = String::from("hello");
```

この二重コロン :: 演算子は、string_from などの名前を使うのではなく、String 型直下の from 関数を特定する働きをします。この記法について詳しくは、第 5 章の「メソッド記法」節と、第 7 章の「モジュールツリーの要素を示すためのパス」でモジュールを使った名前空間分けについて話をするときに議論します。

この種の文字列は、変更することが**できます**:

```
# fn main() {
    let mut s = String::from("hello");

    s.push_str(", world!"); // push_str()関数は、リテラルをStringに付け加える

    println!("{}", s); // これは`hello, world!`と出力する
# }
```

では、ここでの違いは何でしょうか？ なぜ、String 型は変更できるのに、リテラルはできないのでしょうか？ 違いは、これら二つの型がメモリを扱う方法にあります。

4.1.5　メモリと確保

文字列リテラルの場合、中身はコンパイル時に判明しているので、テキストは最終的なバイナリファイルに直接ハードコードされます。このため、文字列リテラルは、高速で効率的になるのです。しかし、これらの特性は、その文字列リテラルの不変性にのみ端を発するものです。残念なことに、コンパイル時にサイズが不明だったり、プログラム実行に合わせてサイズが可変なテキスト片用に一塊のメモリをバイナリに確保しておくことは不可能です。

String 型では、可変かつ伸長可能なテキスト破片をサポートするために、コンパイル時には不明な量のメモリをヒープに確保して内容を保持します。つまり:

- メモリは、実行時にメモリアロケータに要求される。
- String 型を使用し終わったら、アロケータにこのメモリを返還する方法が必要である。

この最初の部分は、既にしています: String::from 関数を呼んだら、その実装が必要なメモリを要求するのです。これは、プログラミング言語において、極めて普遍的です。

しかしながら、2 番目の部分は異なります。**ガベージコレクタ** (GC) 付きの言語では、GC がこれ以上、使用されないメモリを検知して片付けるため、プログラマは、そのことを考慮する必要はありません。GC を持たない言語の多くでは、メモリがもう使用されないことを見計らって、明示的に解放するコードを呼び出すのは、プログラマの責任になります。ちょうど要求の際にしたようにですね。これを正確にすることは、歴史的にも難しいプログラミング問題の一つであり続けています。もし、忘れていたら、メモリを無駄にします。タイミングが早すぎたら、無効な変数を作ってしまいます。2

回解放してしまっても、バグになるわけです。allocate と free は完璧に 1 対 1 対応にしなければならないのです。

Rust は、異なる道を歩んでいます: ひとたび、メモリを所有している変数がスコープを抜けたら、メモリは自動的に返還されます。こちらの例は、リスト 4-1 のスコープ例を文字列リテラルから String 型を使うものに変更したバージョンになります:

```
# fn main() {
    {
        let s = String::from("hello"); // sはここから有効になる

        // sで作業をする
    }                                  // このスコープはここでおしまい。sは
                                       // もう有効ではない
# }
```

String 型が必要とするメモリをアロケータに返還することが自然な地点があります: s 変数がスコープを抜ける時です。変数がスコープを抜ける時、Rust は特別な関数を呼んでくれます。この関数は、drop と呼ばれ、ここに String 型の書き手はメモリを返還するコードを配置することができます。Rust は、閉じ波括弧で自動的に drop 関数を呼び出します。

> 注釈: C++ では、要素の生存期間の終了地点でリソースを解放するこのパターンを時に、**RAII**(Resource Aquisition Is Initialization: リソースの獲得は、初期化である) と呼んだりします。Rust の drop 関数は、あなたが RAII パターンを使ったことがあれば、馴染み深いものでしょう。

このパターンは、Rust コードの書かれ方に甚大な影響をもたらします。現状は簡単そうに見えるかもしれませんが、ヒープ上に確保されたデータを複数の変数に使用させるようなもっと複雑な場面では、コードの振る舞いは、予期しないものになる可能性もあります。これから、そのような場面を掘り下げてみましょう。

4.1.5.1　ムーブによる変数とデータの相互作用

Rust においては、複数の変数が同じデータに対して異なる手段で相互作用することができます。整数を使用したリスト 4-2 の例を見てみましょう。

```
# fn main() {
    let x = 5;
    let y = x;
# }
```

リスト 4-2: 変数 x の整数値を y に代入する

もしかしたら、何をしているのか予想することができるでしょう:「値 5 を x に束縛する; それから x の値をコピーして y に束縛する。」これで、二つの変数 (x と y) が存在し、両方、値は 5 になりま

した。これは確かに起こっている現象を説明しています。なぜなら、整数は既知の固定サイズの単純な値で、これら二つの 5 という値は、スタックに積まれるからです。

では、String バージョンを見ていきましょう:

```
# fn main() {
    let s1 = String::from("hello");
    let s2 = s1;
# }
```

このコードは酷似していますので、動作方法も同じだと思い込んでしまうかもしれません: 要するに、2 行目で s1 の値をコピーし、s2 に束縛するということです。ところが、これは全く起こることを言い当てていません。

図 4-1 を見て、ベールの下で String に何が起きているかを確かめてください。String 型は、左側に示されているように、3 つの部品でできています: 文字列の中身を保持するメモリへのポインタと長さ、そして、許容量です。この種のデータは、スタックに保持されます。右側には、中身を保持したヒープ上のメモリがあります。

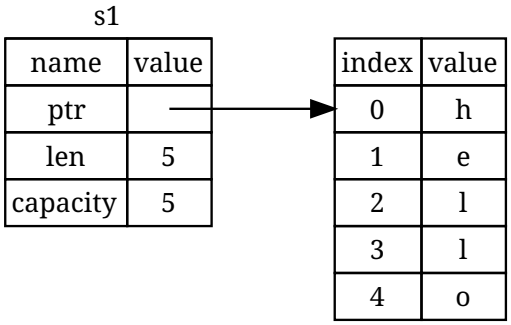


図 4-1: s1 に束縛された "hello" という値を保持する String のメモリ上の表現

長さは、String 型の中身が現在使用しているメモリ量をバイトで表したものです。許容量は、String 型がアロケータから受け取った全メモリ量をバイトで表したものです。長さと許容量の違いは問題になることですが、この文脈では違うので、とりあえずは、許容量を無視しても構わないでしょう。

s1 を s2 に代入すると、String 型のデータがコピーされます。つまり、スタックにあるポインタ、長さ、許容量をコピーするということです。ポインタが指すヒープ上のデータはコピーしません。言い換えると、メモリ上のデータ表現は図 4-2 のようになるということです。

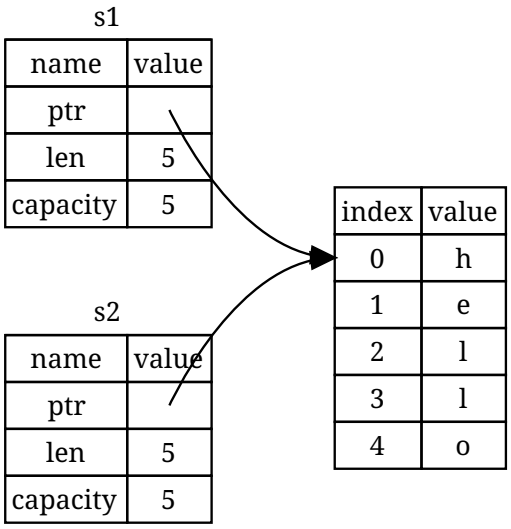


図 4-2: `s1` のポインタ、長さ、許容量のコピーを保持する変数 `s2` のメモリ上での表現

メモリ上の表現は、図 4-3 のようにはなり**ません**。これは、Rust が代わりにヒープデータもコピーするという選択をしていた場合のメモリ表現ですね。Rust がこれをしていたら、ヒープ上のデータが大きい時に `s2 = s1` という処理の実行時性能がとても悪くなっていた可能性があるでしょう。

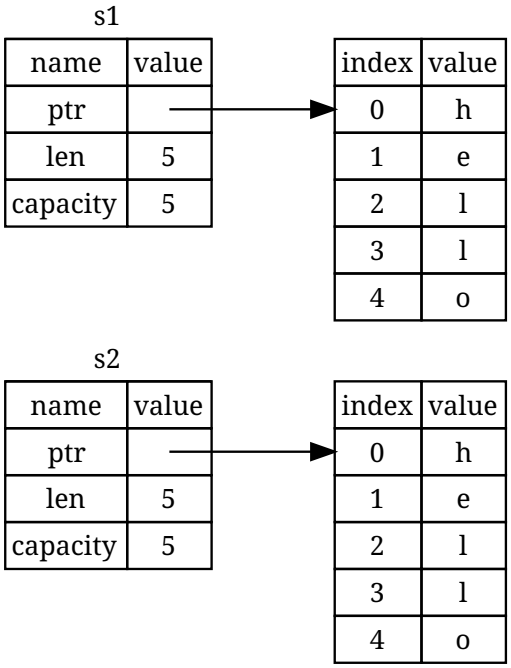


図 4-3: Rust がヒープデータもコピーしていた場合に `s2 = s1` という処理が行なった可能性のあること

先ほど、変数がスコープを抜けたら、Rust は自動的に `drop` 関数を呼び出し、その変数が使っていたヒープメモリを片付けると述べました。しかし、図 4-2 は、両方のデータポインタが同じ場所を指していることを示しています。これは問題です: `s2` と `s1` がスコープを抜けたら、両方とも同じメモリを解放しようとします。これは**二重解放**エラーとして知られ、以前触れたメモリ安全性上のバグの一つになります。メモリを 2 回解放することは、memory corruption (訳注：メモリの崩壊。意図せぬメモリの書き換え) につながり、セキュリティ上の脆弱性を生む可能性があります。

メモリ安全性を保証するために、`let s2 = s1;` の行の後、コンパイラは `s1` が最早有効ではないとみなします。故に `s1` がスコープを抜けた際に何も解放する必要がなくなるわけです。`s2` の生成後に `s1` を使用しようとしたら、どうなるかを確認してみましょう。動かないでしょう:

```
# fn main() {
    let s1 = String::from("hello");
    let s2 = s1;

    println!("{}, world!", s1);
# }
```

コンパイラが無効化された参照は使用させてくれないので、以下のようなエラーが出るでしょう:

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0382]: borrow of moved value: `s1`
              (ムーブされた値の借用: `s1`)
 --> src/main.rs:5:28
  |
2 |     let s1 = String::from("hello");
  |         -- move occurs because `s1` has type `String`, which does not
      implement the `Copy` trait
  |           (`s1`は`String`型を持ち、これは`Copy`トレイトを実装していないの
      で、ムーブが発生します)
3 |     let s2 = s1;
  |              -- value moved here
  |                 (ここで値がムーブされました)
4 |
5 |     println!("{}, world!", s1);
  |                            ^^ value borrowed here after move
  |                               (ムーブ後にここで借用されています)
  |
  = note: this error originates in the macro `$crate::format_args_nl` which
      comes from the expansion of the macro `println` (in Nightly builds, run
      with -Z macro-backtrace for more info)
help: consider cloning the value if the performance cost is acceptable
  |
3 |     let s2 = s1.clone();
  |                 ++++++++
```

```
For more information about this error, try `rustc --explain E0382`.
error: could not compile `ownership` (bin "ownership") due to 1 previous error
```

他の言語を触っている間に **shallow copy** と **deep copy** という用語を耳にしたことがあるなら、データのコピーなしにポインタと長さ、許容量をコピーするという概念は、shallow copy することのように思えるかもしれません。ですが、コンパイラは最初の変数をも無効化するので、shallow copy と呼ばれる代わりに、**ムーブ**として知られているわけです。この例では、`s1` は `s2` に**ムーブ**されたと表現するでしょう。以上より、実際に起きることを図 4-4 に示してみました。

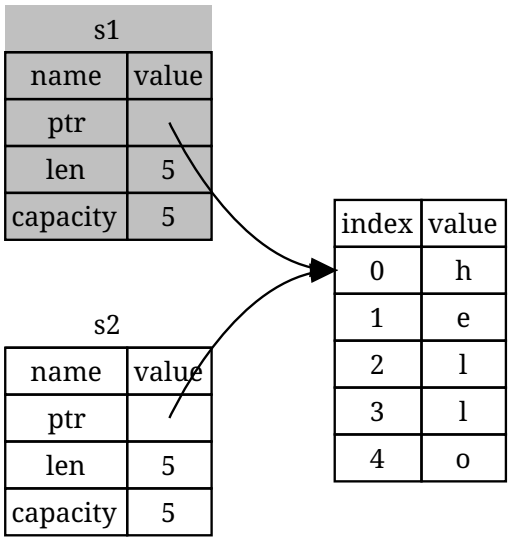


図 4-4: `s1` が無効化された後のメモリ表現

これにて一件落着です。`s2` だけが有効なので、スコープを抜けたら、それだけがメモリを解放して、終わりになります。

付け加えると、これにより暗示される設計上の選択があります: Rust では、自動的にデータの”deep copy” が行われることは絶対にないわけです。それ故に、あらゆる**自動**コピーは、実行時性能の観点で言うと、悪くないと考えてよいことになります。

4.1.5.2　クローンによる変数とデータの相互作用

仮に、スタック上のデータだけでなく、**本当に** `String` 型のヒープデータの deep copy が必要ならば、`clone` と呼ばれるよくあるメソッドを使うことができます。メソッド記法については第 5 章で議論しますが、メソッドは多くのプログラミング言語に見られる機能なので、以前に見かけたこともあるんじゃないでしょうか。

これは、`clone` メソッドの動作例です:

```
# fn main() {
    let s1 = String::from("hello");
    let s2 = s1.clone();

    println!("s1 = {}, s2 = {}", s1, s2);
# }
```

これは問題なく動作し、図 4-3 で示した動作を明示的に生み出します。ここでは、ヒープデータが**実際に**コピーされています。

clone メソッドの呼び出しを見かけたら、何らかの任意のコードが実行され、その実行コストは高いと把握できます。何か違うことが起こっているなと見た目でわかるわけです。

4.1.5.3　スタックのみのデータ: コピー

まだ話題にしていない別の問題があります。この整数を使用したコードは、一部をリスト 4-2 で示しましたが、うまく動作する有効なものです:

```
# fn main() {
    let x = 5;
    let y = x;

    println!("x = {}, y = {}", x, y);
# }
```

ですが、このコードは一見、今学んだことと矛盾しているように見えます: clone メソッドの呼び出しがないのに、x は有効で、y にムーブされませんでした。

その理由は、整数のようなコンパイル時に既知のサイズを持つ型は、スタック上にすっぽり保持されるので、実際の値をコピーするのも高速だからです。これは、変数 y を生成した後にも x を無効化したくなる理由がないことを意味します。換言すると、ここでは、shallow copy と deep copy の違いがないことになり、clone メソッドを呼び出しても、一般的な shallow copy 以上のことをしなくなり、そのまま放置しておけるということです。

Rust には Copy トレイトと呼ばれる特別な注釈があり、整数のようなスタックに保持される型に対して配置することができます (トレイトについては第 10 章でもっと詳しく話します)。型が Copy トレイトを実装していれば、その型を持つ変数はムーブされず、代わりに自明な方法でにコピーされます。そのため、他の変数に代入した後でも、元の変数は有効なままです。

コンパイラは、型やその一部分でも Drop トレイトを実装している場合、Copy による注釈をさせてくれません。型の値がスコープを外れた時に何か特別なことを起こす必要がある場合に、Copy 注釈を追加すると、コンパイルエラーが出ます。型に Copy 注釈をつけてトレイトを実装する方法について学ぶには、付録 C の「導出可能なトレイト」をご覧ください。

では、どの型が Copy トレイトを実装しているのでしょうか？ 確実に知るためには、調べたい型のドキュメントをチェックすればいいのですが、一般規則として、単純なスカラー値の集合は何でも Copy を実装でき、メモリ確保が必要だったり、何らかの形態のリソースだったりするものは Copy を

実装できません。ここに Copy を実装する型の一部を並べておきます。

- あらゆる整数型。u32 など。
- 論理値型である bool 。true と false という値がある。
- あらゆる浮動小数点型、f64 など。
- 文字型である char 。
- タプル。ただ、Copy の型だけを含む場合。例えば、(i32, i32) は Copy だが、(i32, String) は違う。

4.1.6　所有権と関数

関数に値を渡すことと、値を変数に代入することの仕組みは似ています。関数に変数を渡すと、代入のようにムーブやコピーされます。リスト 4-3 は変数がスコープに入ったり、抜けたりする地点について注釈してある例です。

ファイル名: src/main.rs

```
fn main() {
    let s = String::from("hello");  // sがスコープに入る

    takes_ownership(s);             // sの値が関数にムーブされ...
                                    // ... ここではもう有効ではない

    let x = 5;                      // xがスコープに入る

    makes_copy(x);                  // xも関数にムーブされるが、
                                    // i32はCopyなので、この後にxを使っても
                                    // 大丈夫

} // ここでxがスコープを抜け、sもスコープを抜ける。ただし、sの値はムーブされているので、
  // 何も特別なことは起こらない。

fn takes_ownership(some_string: String) { // some_stringがスコープに入る。
    println!("{}", some_string);
} // ここでsome_stringがスコープを抜け、`drop`が呼ばれる。後ろ盾してたメモリが解放される。
  // 後ろ盾してたメモリが解放される。

fn makes_copy(some_integer: i32) { // some_integerがスコープに入る
    println!("{}", some_integer);
} // ここでsome_integerがスコープを抜ける。何も特別なことはない。
```

リスト 4-3: 所有権とスコープが注釈された関数群

takes_ownership の呼び出し後に s を呼び出そうとすると、コンパイラは、コンパイルエラーを投

げるでしょう。これらの静的チェックにより、ミスを犯さないでいられます。s や x を使用するコードを main に追加してみて、どこで使えて、そして、所有権規則により、どこで使えないかを確認してください。

4.1.7　戻り値とスコープ

値を返すことでも、所有権は移動します。リスト 4-4 に値を返す関数の例を、リスト 4-3 と似た注釈を付けて示します。

ファイル名: src/main.rs

```
fn main() {
    let s1 = gives_ownership();         // gives_ownershipは、戻り値をs1に
                                        // ムーブする

    let s2 = String::from("hello");     // s2がスコープに入る

    let s3 = takes_and_gives_back(s2);  // s2はtakes_and_gives_backにムーブされ
                                        // 戻り値もs3にムーブされる
} // ここで、s3はスコープを抜け、ドロップされる。s2はムーブされているので、何も
    起きない。
  // s1もスコープを抜け、ドロップされる。

fn gives_ownership() -> String {             // gives_ownershipは、戻り値を
                                             // 呼び出した関数にムーブする

    let some_string = String::from("yours"); // some_stringがスコープに入る

    some_string                              // some_stringが返され、呼び出し元
    関数に
                                             // ムーブされる
}

// この関数は、Stringを一つ受け取り、返す。
fn takes_and_gives_back(a_string: String) -> String { // a_stringがスコープに入
    る。

    a_string  // a_stringが返され、呼び出し元関数にムーブされる
}
```

リスト 4-4: 戻り値の所有権を移動する

変数の所有権は、毎回同じパターンを辿っています: 別の変数に値を代入すると、ムーブされます。ヒープにデータを含む変数がスコープを抜けると、データの所有権が別の変数にムーブされていない限り、drop により片付けられるでしょう。

これでもいいのですが、所有権を取り、またその所有権を戻す、ということを全ての関数でしていたら、ちょっとめんどくさいですね。関数に値は使わせるものの所有権を取らないようにさせるには

どうするべきでしょうか。返したいと思うかもしれない関数本体で発生したあらゆるデータとともに、再利用したかったら、渡されたものをまた返さなきゃいけないのは、非常に煩わしいことです。

Rust では、タプルを使って複数の値を返すことができます。リスト 4-5 のようにですね。

ファイル名: src/main.rs

```
fn main() {
    let s1 = String::from("hello");

    let (s2, len) = calculate_length(s1);

    //'{}'の長さは、{}です
    println!("The length of '{}' is {}.", s2, len);
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // len()メソッドは、Stringの長さを返します

    (s, length)
}
```

リスト 4-5: 引数の所有権を返す

でも、これでは、大袈裟すぎますし、ありふれているはずの概念に対して、作業量が多すぎます。私たちにとって幸運なことに、Rust には所有権を移動することなく値を使うための機能があり、**参照**と呼ばれます。

4.2　参照と借用

リスト 4-5 のタプルコードの問題は、String 型を呼び出し元の関数に戻さないと、calculate_length を呼び出した後に、String オブジェクトが使えなくなることであり、これは String オブジェクトが calculate_length にムーブされてしまうためでした。代わりに、String 値への参照を渡すことができます。**参照**はアドレスであり、それを辿ってそのアドレスに置かれているデータにアクセスできるという点で、ポインタと似ています; データは他の変数によって所有されています。ポインタと異なる点としては、参照はその生存期間中を通して、特定の型の有効な値を指していることが保証されています。

ここで、値の所有権をもらう代わりに引数としてオブジェクトへの参照を取る calculate_length 関数を定義し、使う方法を見てみましょう:

ファイル名: src/main.rs

```
fn main() {
    let s1 = String::from("hello");
```

```
    let len = calculate_length(&s1);

    // '{}'の長さは、{}です
    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

まず、変数宣言と関数の戻り値にあったタプルコードは全てなくなったことに気付いてください。2 番目に、`&s1` を `calcuate_length` に渡し、その定義では、`String` 型ではなく、`&String` を受け取っていることに注目してください。これらのアンド記号が参照を表しており、これのおかげで所有権をもらうことなく値を参照することができるのです。図 4-5 にこの概念を描写します。

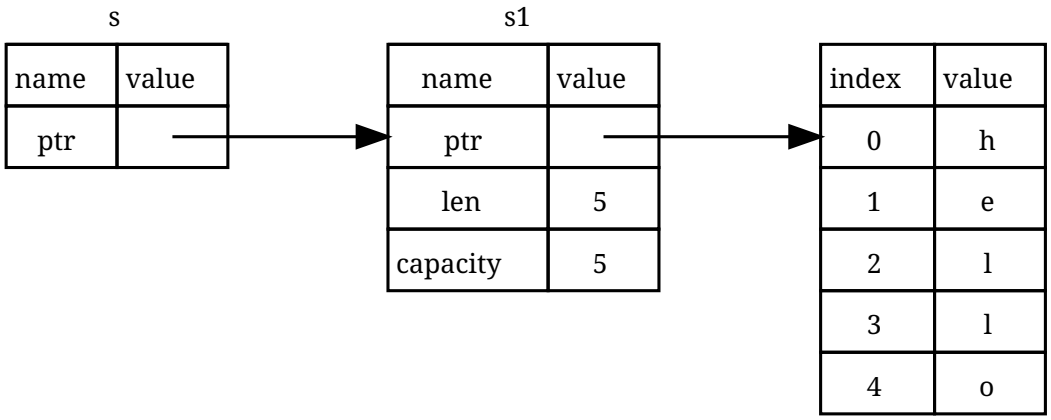


図 4-5: `String s1` を指す `&String s` の図表

> 注釈: `&` による参照の逆は、**参照外し**であり、参照外し演算子の `*` で達成できます。第 8 章で参照外し演算子の使用例を眺め、第 15 章で参照外しについて詳しく議論します。

ここの関数呼び出しについて、もっと詳しく見てみましょう:

```
# fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);
#
#     // '{}'の長さは、{}です
#     println!("The length of '{}' is {}.", s1, len);
# }
#
# fn calculate_length(s: &String) -> usize {
```

```
#     s.len()
# }
```

この `&s1` という記法により、`s1` の値を**参照する**参照を生成することができますが、これを所有することはありません。所有してないということは、指している値は、参照が使用されなくなってもドロップされないということです。

同様に、関数のシグニチャでも、`&` を使用して引数 `s` の型が参照であることを示しています。説明的な注釈を加えてみましょう:

```
# fn main() {
#     let s1 = String::from("hello");
#
#     let len = calculate_length(&s1);
#
#     println!("The length of '{}' is {}.", s1, len);
# }
#
fn calculate_length(s: &String) -> usize { // sはStringへの参照
    s.len()
} // ここで、sはスコープ外になる。けど、参照しているものの所有権を持っているわけ
    ではないので
  // ドロップはされない。
```

変数 `s` が有効なスコープは通常の関数の引数のものと同じですが、`s` はそれが指す値に対する所有権を持っていないので、`s` が使用されなくなっても指している値をドロップすることはありません。関数が実際の値の代わりに参照を引数に取ると、所有権をもらわないので、所有権を返す目的で値を返す必要はありません。

参照を作成することを**借用**と呼びます。現実生活のように、誰かが何かを所有していたら、それを借りることができます。用が済んだら、返さないといけません。持っているのとは違うのです。

では、借用した何かを変更しようとしたら、どうなるのでしょうか？ リスト 4-6 のコードを試してください。ネタバレ注意: 動きません！

ファイル名: src/main.rs

```
fn main() {
    let s = String::from("hello");

    change(&s);
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}
```

リスト 4-6: 借用した値を変更しようと試みる

これがエラーです:

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0596]: cannot borrow `*some_string` as mutable, as it is behind a `&`
    reference
              (`*some_string`は`&`参照の背後にあるため、それを可変として借用する
                  ことはできません)
 --> src/main.rs:8:5
  |
8 |     some_string.push_str(", world");
  |     ^^^^^^^^^^^ `some_string` is a `&` reference, so the data it refers to
      cannot be borrowed as mutable
  |               (`some_string`は`&`参照なので、それが指すデータを可変として借
      用することはできません)
  |
help: consider changing this to be a mutable reference
     (これを可変参照に変更することを検討してください)
  |
7 | fn change(some_string: &mut String) {
  |                         +++

For more information about this error, try `rustc --explain E0596`.
error: could not compile `ownership` (bin "ownership") due to 1 previous error
```

変数が標準で不変なのと全く同様に、参照も不変なのです。参照している何かを変更することは叶わないわけです。

4.2.1　可変参照

借用された値を変更できるようにするには、代わりに**可変参照**を使うという一捻りを加えるだけでよく、これでリスト 4-6 のコードを修正できます:

ファイル名: src/main.rs

```
fn main() {
    let mut s = String::from("hello");

    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

まず s が mut となるように変更します。次に、change 関数を呼ぶところで &mut s によって可変参照を作成し、さらに関数シグネチャを、some_string: &mut String で可変参照を受け入れるように更新します。これにより、change 関数は借用する値を変更しようとすることがとても明確になります。

可変参照には大きな制約が一つあります: ある値への可変参照が存在するなら、その値への参照を他に作ることはできません。このコードは s への可変参照を 2 個作成しようとしていますが、これは失敗します:

ファイル名: src/main.rs

```
# fn main() {
    let mut s = String::from("hello");

    let r1 = &mut s;
    let r2 = &mut s;

    println!("{}, {}", r1, r2);
# }
```

これがエラーです:

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0499]: cannot borrow `s` as mutable more than once at a time
              (一度に`s`を可変として2回以上借用することはできません)
 --> src/main.rs:5:14
  |
4 |     let r1 = &mut s;
  |              ------ first mutable borrow occurs here
  |                     (最初の可変借用はここで発生しています)
5 |     let r2 = &mut s;
  |              ^^^^^^ second mutable borrow occurs here
  |                     (2つ目の可変借用はここで発生しています)
6 |
7 |     println!("{}, {}", r1, r2);
  |                        -- first borrow later used here
  |                          (最初の参照はここで後で使用されています)

For more information about this error, try `rustc --explain E0499`.
error: could not compile `ownership` (bin "ownership") due to 1 previous error
```

このエラーによると、一度に s を可変として 2 回以上借用することはできないので、このコードは不正だ、とのことです。最初の可変借用は r1 にあり、println! で使用されるまで続かないといけませんが、この可変借用の作成から使用までの間に、r1 と同じデータを借用する別の可変借用を r2 に作成しようとしました。

同じデータへの複数の可変参照が同時に存在することを禁止する、という制約は、可変化を許可するものの、それを非常に統制の取れた形で行えます。これは、新たな Rustacean にとっては、壁です。なぜなら、多くの言語では、いつでも好きな時に可変化できるからです。この制約がある利点は、コンパイラがコンパイル時にデータ競合を防ぐことができる点です。データ競合とは、競合状態と類似していて、これら 3 つの振る舞いが起きる時に発生します:

- 2 つ以上のポインタが同じデータに同時にアクセスする。
- 少なくとも一つのポインタがデータに書き込みを行っている。
- データへのアクセスを同期する機構が使用されていない。

データ競合は未定義の振る舞いを引き起こし、実行時に追いかけようとした時に特定し解決するのが難しい問題です。しかし、Rust は、データ競合が起こるコードのコンパイルを拒否することで、この問題が発生しないようにしてくれるわけです。

いつものように、波かっこを使って新しいスコープを生成し、**同時並行**なものでなく、複数の可変な参照を作ることができます。

```
# fn main() {
    let mut s = String::from("hello");

    {
        let r1 = &mut s;
    } // r1はここでスコープを抜けるので、問題なく新しい参照を作ることができる。

    let r2 = &mut s;
# }
```

コンパイラは可変と不変な参照を組み合わせることに関しても、似たような規則を強制します。このコードはエラーになります:

```
# fn main() {
    let mut s = String::from("hello");

    let r1 = &s; // 問題なし
    let r2 = &s; // 問題なし
    let r3 = &mut s; // 大問題!

    println!("{}, {}, and {}", r1, r2, r3);
# }
```

これがエラーです:

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as
    immutable
             (`s`は不変で借用されているので、可変で借用できません)
 --> src/main.rs:6:14
  |
4 |     let r1 = &s; // 問題なし
  |              -- immutable borrow occurs here
  |                 (不変借用はここで発生しています)
5 |     let r2 = &s; // 問題なし
6 |     let r3 = &mut s; // 大問題!
  |              ^^^^^^ mutable borrow occurs here
```

```
  |                      (可変借用はここで発生しています)
7 |
8 |     println!("{}, {}, and {}", r1, r2, r3);
  |                                 -- immutable borrow later used here
  |                                    (不変借用はその後ここで使われています)

For more information about this error, try `rustc --explain E0502`.
error: could not compile `ownership` (bin "ownership") due to 1 previous error
```

ふう！ **さらに**不変な参照をしている間は、同じ値に対して可変な参照をすることはできません。不変参照の使用者は、それ以降に値が突然変わることなんて予想してません！ しかしながら、複数の不変参照をすることは許されています。データを読み込んでいるだけの人に、他人がデータを読み込むことに対して影響を与える能力はないからです。

参照のスコープは、それが導入されたところから始まり、その参照が最後に使用される時点まで続きます。例えば、次のコードはコンパイルできるでしょう。不変参照の最後の使用箇所である `println!` は、可変参照が導入されるよりも前に発生するからです:

```
# fn main() {
    let mut s = String::from("hello");

    let r1 = &s; // 問題なし
    let r2 = &s; // 問題なし
    println!("{} and {}", r1, r2);
    // r1とr2はもうこれ以降使用されません

    let r3 = &mut s; // 問題なし
    println!("{}", r3);
# }
```

不変参照 `r1` と `r2` のスコープは、それらが最後に使用される `println!` の後で終了します。これは可変参照 `r3` が作成されるより前のことです。これらのスコープは重なっていないので、このコードは許可されます: コンパイラは、スコープの終了より前の時点で参照がもはや使用されていないということを、判別できるのです。

借用エラーは、時としてイライラするものではありますが、Rust コンパイラがバグの可能性を早期に指摘してくれ (それも実行時ではなくコンパイル時に)、問題の発生箇所をズバリ示してくれるのだと覚えておいてください。そうして想定通りにデータが変わらない理由を追いかける必要がなくなります。

4.2.2　宙に浮いた参照

ポインタのある言語では、誤ってダングリングポインタを生成してしまいやすいです。ダングリングポインタとは、他人に渡されてしまった可能性のあるメモリを指すポインタのことであり、その箇所へのポインタを保持している間に、メモリを解放してしまうことで発生します。対照的に Rust では、コンパイラが、参照がダングリング参照に絶対ならないよう保証してくれます: つまり、何らか

のデータへの参照があったら、コンパイラは参照がスコープを抜けるまで、データがスコープを抜けることがないよう確認してくれるわけです。

ダングリング参照を作ってみて、コンパイラがどのようにこれをコンパイルエラーで阻止するか見てみましょう:

ファイル名: src/main.rs

```
fn main() {
    let reference_to_nothing = dangle();
}

fn dangle() -> &String {
    let s = String::from("hello");

    &s
}
```

こちらがエラーです:

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0106]: missing lifetime specifier
(エラー: ライフタイム指定子がありません)
 --> src/main.rs:5:16
  |
5 | fn dangle() -> &String {
  |                ^ expected named lifetime parameter
  |                  (名前付きのライフタイムパラメータが期待されていました)
  |
  = help: this function's return type contains a borrowed value, but there is no
       value for it to be borrowed from
  = (助言: この関数の戻り値型は、借用した値を含んでいますが、借用される値がどこ
      にもありません)
help: consider using the `'static` lifetime, but this is uncommon unless you're
    returning a borrowed value from a `const` or a `static`
(助言: `'static`ライフタイムを使うことを検討してください、ただし`const`または`
    static`から借用した値を返すのでない限り、これはあまり使われません)
  |
5 | fn dangle() -> &'static String {
  |                 +++++++
help: instead, you are more likely to want to return an owned value
(助言: または、所有権を持つ値を返す方がいいかもしれません)
  |
5 - fn dangle() -> &String {
5 + fn dangle() -> String {
  |

For more information about this error, try `rustc --explain E0106`.
error: could not compile `ownership` (bin "ownership") due to 1 previous error
```

このエラーメッセージは、まだ講義していない機能について触れています: ライフタイムです。ライフタイムについては第 10 章で詳しく議論しますが、ライフタイムに関する部分を無視すれば、このメッセージは、確かにこのコードが問題になる理由に関する鍵を握っています:

```
this function's return type contains a borrowed value, but there is no value
for it to be borrowed from
```

dangle コードの各段階で一体何が起きているのかを詳しく見ていきましょう:

ファイル名: src/main.rs

```
# fn main() {
#     let reference_to_nothing = dangle();
# }
#
fn dangle() -> &String { // dangleはStringへの参照を返す

    let s = String::from("hello"); // sは新しいString

    &s // String sへの参照を返す
} // ここで、sはスコープを抜け、ドロップされる。そのメモリは消される。
  // 危険だ
```

s は、dangle 内で生成されているので、dangle のコードが終わったら、s は解放されてしまいますが、そこへの参照を返そうとしました。つまり、この参照は無効な String を指していると思われるのです。よくないことです！ コンパイラは、これを阻止してくれるのです。

ここでの解決策は、String を直接返すことです:

```
# fn main() {
#     let string = no_dangle();
# }
#
fn no_dangle() -> String {
    let s = String::from("hello");

    s
}
```

これは何の問題もなく動きます。所有権はムーブされ、何も解放されることはありません。

4.2.3　参照の規則

参照について議論したことを再確認しましょう:

- 任意のタイミングで、一つの可変参照**か**不変な参照いくつでもの**どちらか**を行える。
- 参照は常に有効でなければならない。

次は、違う種類の参照を見ていきましょう: スライスです。

4.3　スライス型

スライスにより、コレクション全体ではなく、その内の連続した要素の列を参照することができます。スライスは参照の一種であり、そのため所有権を持っていません。

ちょっとしたプログラミングの問題を考えてみましょう: スペースで区切られた複数の単語からなる文字列を受け取って、その文字列中の最初の単語を返す関数を書いてください。関数が文字列中に空白を見つけられなかったら、文字列全体が一つの単語に違いないので、文字列全体が返されるべきです。

スライスが解決しようとしている問題を理解するために、スライスを使わずにこの関数のシグネチャをどう書くかという問題を通して、考えてみましょう。

```
fn first_word(s: &String) -> ?
```

`first_word` 関数は引数に `&String` をとります。所有権はいらないので、これで十分です。ですが、何を返すべきでしょうか？ 文字列の**一部**について語る方法が全くありません。しかし、空白によって示される単語の終端の添え字を返すことができますね。リスト 4-7 に示したように、その方法を試してみましょう。

ファイル名: src/main.rs

```
fn first_word(s: &String) -> usize {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return i;
        }
    }

    s.len()
}
#
# fn main() {}
```

リスト 4-7: `String` 引数へのバイト数で表された添え字を返す `first_word` 関数

`String` の値を要素ごとに見て、空白かどうかを確かめる必要があるので、`as_bytes` メソッドを使って、`String` オブジェクトをバイト配列に変換しています。

```
# fn first_word(s: &String) -> usize {
    let bytes = s.as_bytes();
#
```

```
#     for (i, &item) in bytes.iter().enumerate() {
#         if item == b' ' {
#             return i;
#         }
#     }
#
#     s.len()
# }
#
# fn main() {}
```

次に、そのバイト配列に対して、iter メソッドを使用してイテレータを生成しています:

```
# fn first_word(s: &String) -> usize {
#     let bytes = s.as_bytes();
#
    for (i, &item) in bytes.iter().enumerate() {
#         if item == b' ' {
#             return i;
#         }
#     }
#
#     s.len()
# }
#
# fn main() {}
```

イテレータについて詳しくは、第 13 章で議論します。今は、iter は、コレクション内の各要素を返すメソッドであること、enumerate が iter の結果をラップして、(結果をそのまま返す) 代わりにタプルの一部として各要素を返すことを知っておいてください。enumerate から返ってくるタプルの第 1 要素は、添え字であり、2 番目の要素は、(コレクションの) 要素への参照になります。これは、手動で添え字を計算するよりも少しだけ便利です。

enumerate メソッドがタプルを返すので、パターンを使って、そのタプルを分配できます。パターンについては第 6 章でさらに議論します。for ループ内で、タプルの添え字に対する i とタプルの 1 バイトに対応する &item を含むパターンを指定しています。.iter().enumerate() から要素への参照を取得するので、パターンに & を使っています。

for ループ内で、バイトリテラル表記を使用して空白を表すバイトを検索しています。空白が見つかったら、その位置を返します。それ以外の場合、s.len() を使って文字列の長さを返します。

```
# fn first_word(s: &String) -> usize {
#     let bytes = s.as_bytes();
#
#     for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return i;
        }
    }
```

```
    s.len()
# }
#
# fn main() {}
```

さて、文字列内の最初の単語の終端の添え字を見つけ出せるようになりましたが、問題があります。usize 型を単独で返していますが、これは &String の文脈でのみ意味を持つ数値です。言い換えると、String から切り離された値なので、将来的にも有効である保証がないのです。リスト 4-7 の first_word 関数を使用するリスト 4-8 のプログラムを考えてください。

ファイル名: src/main.rs

```
# fn first_word(s: &String) -> usize {
#     let bytes = s.as_bytes();
#
#     for (i, &item) in bytes.iter().enumerate() {
#         if item == b' ' {
#             return i;
#         }
#     }
#
#     s.len()
# }
#
fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s); // word will get the value 5
                               // wordの中身は、値5になる

    s.clear(); // this empties the String, making it equal to ""
               // Stringを空にする。つまり、""と等しくする

    // word still has the value 5 here, but there's no more string that
    // we could meaningfully use the value 5 with. word is now totally invalid!
    // wordはまだ値5を保持しているが、もうこの値を正しい意味で使用できる文字列は
    存在しない。
    // wordは今や完全に無効なのだ！
}
```

リスト 4-8: first_word 関数の呼び出し結果を保持し、String の中身を変更する

このプログラムは何のエラーもなくコンパイルが通り、word を s.clear() の呼び出し後に使用しても、コンパイルが通ります。word は s の状態に全く関連づけられていないので、その中身はまだ値 5 のままです。その値 5 を変数 s に使用し、最初の単語を取り出そうとすることはできますが、これはバグでしょう。というのも、s の中身は、5 を word に保存した後変わってしまったからです。

word 内の添え字が s に格納されたデータと同期されなくなるのを心配することは、面倒ですし間

違いになりやすいです！ これらの添え字の管理は、second_word 関数を書いたら、さらに難しくなります。そのシグニチャは以下のようになるはずです:

```
fn second_word(s: &String) -> (usize, usize) {
```

今、私たちは開始と終端の添え字を追うようになりました。特定の状態のデータから計算されたが、その状態に全く紐付けられていない値がさらに増えました。いつの間にか変わってしまうので、同期を取る必要のある、関連性のない変数が 3 つになってしまいました。

運のいいことに、Rust にはこの問題への解決策が用意されています: 文字列スライスです。

4.3.1　文字列スライス

文字列スライスとは、String の一部への参照で、こんな見た目をしています:

```
# fn main() {
    let s = String::from("hello world");

    let hello = &s[0..5];
    let world = &s[6..11];
# }
```

hello は String 全体への参照ではなく、追加の [0..5] という部分で指定された、String の一部への参照です。[starting_index..ending_index] と指定することで、角かっこに範囲を使い、スライスを生成します。ここで、starting_index はスライスの最初の位置、ending_index はスライスの終端位置よりも、1 大きい値です。内部的には、スライスデータ構造は、開始地点とスライスの長さを保持しており、スライスの長さは ending_index から starting_index を引いたものに対応します。以上より、let world = &s[6..11]; の場合には、world は s の添え字 6 のバイトへのポインタと 5 という長さを持つスライスになるでしょう。

図 4-6 は、これを図解しています。

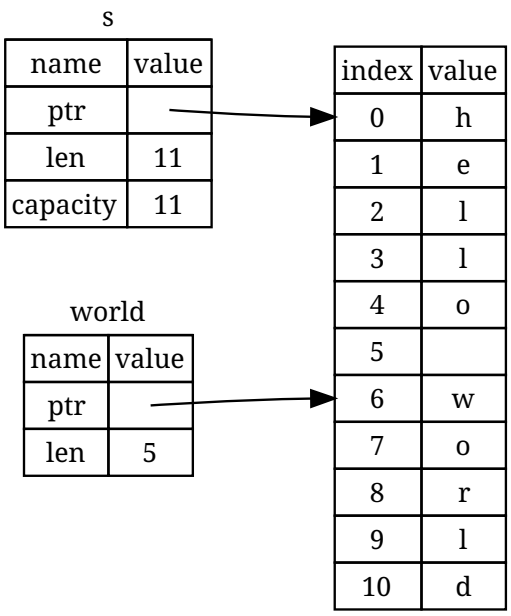


図 4-6: `String` オブジェクトの一部を参照する文字列スライス

Rust の `..` という範囲記法で、添え字 0 から始めたければ、2 連ピリオドの前に値を書かなければいいです。換言すれば、これらは等価です:

```
let s = String::from("hello");

let slice = &s[0..2];
let slice = &s[..2];
```

同様の意味で、`String` の最後のバイトをスライスが含むのならば、末尾の数値を書かなければいいです。つまり、これらは等価になります:

```
let s = String::from("hello");

let len = s.len();

let slice = &s[3..len];
let slice = &s[3..];
```

さらに、両方の値を省略すると、文字列全体のスライスを得られます。故に、これらは等価です:

```
let s = String::from("hello");

let len = s.len();

let slice = &s[0..len];
```

```
let slice = &s[..];
```

> 注釈: 文字列スライスの範囲添え字は、有効な UTF-8 文字境界に置かなければなりません。マルチバイト文字の真ん中で文字列スライスを生成しようとしたら、エラーでプログラムは落ちるでしょう。この節では文字列スライスを導入することが目的なので、ASCII のみを想定しています; UTF-8 に関するより徹底した議論は、第 8 章の「文字列で UTF-8 エンコードされたテキストを格納する」節で行います。

これらの情報を念頭に、`first_word` を書き直してスライスを返すようにしましょう。文字列スライスを意味する型は、`&str` と記述します:

ファイル名: src/main.rs

```
fn first_word(s: &String) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
#
# fn main() {}
```

リスト 4-7 で取った方法と同じように、最初の空白を探すことで単語の終端の添え字を取得しています。空白を発見したら、文字列の最初を開始地点、空白の添え字を終了地点として使用して文字列スライスを返しています。

これで、`first_word` を呼び出すと、元のデータに紐付けられた単独の値を得られるようになりました。この値は、スライスの開始地点への参照とスライス中の要素数から構成されています。

`second_word` 関数についても、スライスを返すことでうまくいくでしょう:

```
fn second_word(s: &String) -> &str {
```

これで、ずっと混乱しにくい素直な API になりました。なぜなら、`String` への参照が有効なままであることをコンパイラが、保証してくれるからです。最初の単語の終端添え字を得た時に、文字列を空っぽにして先ほどの添え字が無効になってしまったリスト 4-8 のプログラムのバグを覚えていますか？ そのコードは、論理的に正しくないのですが、即座にエラーにはなりませんでした。問題は後になってから発生し、それは空の文字列に対して、最初の単語の添え字を使用し続けようとした時でした。スライスならこんなバグはあり得ず、コードに問題があるなら、もっと迅速に判明します。スラ

イスバージョンの first_word を使用すると、コンパイルエラーが発生します:

ファイル名: src/main.rs

```
# fn first_word(s: &String) -> &str {
#     let bytes = s.as_bytes();
#
#     for (i, &item) in bytes.iter().enumerate() {
#         if item == b' ' {
#             return &s[0..i];
#         }
#     }
#
#     &s[..]
# }
#
fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s);

    s.clear(); // error! （エラー！）

    println!("the first word is: {}", word);
}
```

こちらがコンパイルエラーです:

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as
    immutable
(エラー: 不変として借用されているので、`s`を可変で借用できません)
  --> src/main.rs:18:5
   |
16 |     let word = first_word(&s);
   |                           -- immutable borrow occurs here
   |                             (不変借用はここで発生しています)
17 |
18 |     s.clear(); // error!
   |     ^^^^^^^^^ mutable borrow occurs here
   |               (可変借用はここで発生しています)
19 |
20 |     println!("the first word is: {}", word);
   |                                       ---- immutable borrow later used here
                                                (不変借用はその後ここで使われて
                                                    います)

For more information about this error, try `rustc --explain E0502`.
error: could not compile `ownership` (bin "ownership") due to 1 previous error
```

借用規則から、何かへの不変な参照がある時、さらに可変な参照を得ることはできないことを思い出してください。clear は String を切り詰める必要があるので、可変な参照を得る必要があります。clear の呼び出しの後の println! は word 中の参照を使用するので、不変参照はその時点でもまだ有効でなくてはいけません。Rust は clear 中の可変参照と word 中の不変参照が同時に存在することを認めないので、コンパイルが失敗します。Rust のおかげで API が使いやすくなるだけでなく、ある種のエラー全てを完全にコンパイル時に排除してくれるのです！

4.3.1.1　スライスとしての文字列リテラル

文字列は、バイナリに埋め込まれると話したことを思い出してください。今やスライスのことを知ったので、文字列リテラルを正しく理解することができます。

```
let s = "Hello, world!";
```

ここでの s の型は、&str です: バイナリのその特定の位置を指すスライスです。これは、文字列が不変である理由にもなっています。要するに、&str は不変な参照なのです。

4.3.1.2　引数としての文字列スライス

リテラルや String 値のスライスを得ることができると知ると、first_word に対して、もう一つ改善点を見出すことができます。シグニチャです:

```
fn first_word(s: &String) -> &str {
```

もっと経験を積んだ Rustacean なら、代わりにリスト 4-9 のようなシグニチャを書くでしょう。というのも、こうすると、同じ関数を &String 値と &str 値両方に使えるようになるからです。

```
fn first_word(s: &str) -> &str {
#     let bytes = s.as_bytes();
#
#     for (i, &item) in bytes.iter().enumerate() {
#         if item == b' ' {
#             return &s[0..i];
#         }
#     }
#
#     &s[..]
# }
#
# fn main() {
#     let my_string = String::from("hello world");
#
#     // `first_word` works on slices of `String`s, whether partial or whole
#     // `first_word`は`String`の全体または部分へのスライスに対して機能する
#     let word = first_word(&my_string[0..6]);
#     let word = first_word(&my_string[..]);
```

```
#     // `first_word` also works on references to `String`s, which are
    equivalent
#     // to whole slices of `String`s
#     // `first_word`は`String`の参照に対しても機能する。この場合は
#     // `String`全体へのスライスと同等。
#     let word = first_word(&my_string);
#
#     let my_string_literal = "hello world";
#
#     // `first_word` works on slices of string literals, whether partial or
    whole
#     // `first_word`は文字列リテラルの全体または部分へのスライスに対して機能す
    る
#     let word = first_word(&my_string_literal[0..6]);
#     let word = first_word(&my_string_literal[..]);
#
#     // Because string literals *are* string slices already,
#     // this works too, without the slice syntax!
#     // 文字列リテラルは「それ自体すでに文字列スライスなので」、
#     // スライス記法なしでも機能するのだ！
#     let word = first_word(my_string_literal);
# }
```

リスト 4-9: s 引数の型に文字列スライスを使用して first_word 関数を改善する

もし、文字列スライスがあるなら、それを直接渡せます。String があるなら、その String のスライスか、String への参照を渡せます。この柔軟性は、第 15 章の「関数やメソッドで暗黙的な参照外し型強制」で扱う機能、**参照外し型強制**の利点を活用して実現されています。

String への参照の代わりに文字列スライスを取るよう関数を定義すると、何も機能を失うことなく API をより一般的で有益なものにできるのです。

Filename: src/main.rs

```
# fn first_word(s: &str) -> &str {
#     let bytes = s.as_bytes();
#
#     for (i, &item) in bytes.iter().enumerate() {
#         if item == b' ' {
#             return &s[0..i];
#         }
#     }
#
#     &s[..]
# }
#
fn main() {
    let my_string = String::from("hello world");

    // `first_word` works on slices of `String`s, whether partial or whole
    // `first_word`は`String`の全体または部分へのスライスに対して機能する
```

```
    let word = first_word(&my_string[0..6]);
    let word = first_word(&my_string[..]);
    // `first_word` also works on references to `String`s, which are equivalent
    // to whole slices of `String`s
    // `first_word`は`String`の参照に対しても機能する。この場合は
    // `String`全体へのスライスと同等。
    let word = first_word(&my_string);

    let my_string_literal = "hello world";

    // `first_word` works on slices of string literals, whether partial or whole
    // `first_word`は文字列リテラルの全体または部分へのスライスに対して機能する
    let word = first_word(&my_string_literal[0..6]);
    let word = first_word(&my_string_literal[..]);

    // Because string literals *are* string slices already,
    // this works too, without the slice syntax!
    // 文字列リテラルは「それ自体すでに文字列スライスなので」、
    // スライス記法なしでも機能するのだ!
    let word = first_word(my_string_literal);
}
```

4.3.2　他のスライス

文字列リテラルは、ご想像通り、文字列に特化したものです。ですが、もっと一般的なスライス型も存在します。この配列を考えてください:

```
let a = [1, 2, 3, 4, 5];
```

文字列の一部を参照したくなる可能性があるのと同様、配列の一部を参照したくなる可能性もあります。以下のようにすれば、参照することができます:

```
let a = [1, 2, 3, 4, 5];

let slice = &a[1..3];

assert_eq!(slice, &[2, 3]);
```

このスライスは、`&[i32]` という型になります。これも文字列スライスと同じように動作します。つまり、最初の要素への参照と長さを保持するのです。この種のスライスは、他のすべての種類のコレクションに対して使用することになるでしょう。それらのコレクションについて、詳しくは、第 8 章でベクタについて話すときに議論します。

4.4　まとめ

所有権、借用、スライスの概念は、Rust プログラムにおいて、コンパイル時にメモリ安全性を保証します。Rust 言語も他のシステムプログラミング言語と同じように、メモリの使用法について制御させてくれるわけですが、データの所有者がスコープを抜けたときに、所有者に自動的にデータを片付けさせることは、この制御をするために、余計なコードを書いたりデバッグしたりする必要がないことを意味します。

所有権は、Rust の他のいろんな部分が動作する方法に影響を与えるので、これ以降もこれらの概念についてさらに語っていく予定です。第 5 章に移って、`struct` でデータをグループ化することについて見ていきましょう。

第 5 章

構造体を使用して関係のあるデータを構造化する

struct または、**構造体**は、意味のあるグループを形成する複数の関連した値をまとめ、名前付けできる独自のデータ型です。あなたがオブジェクト指向言語に造詣が深いなら、**struct** はオブジェクトのデータ属性みたいなものです。この章では、すでに学習したものに積み重ねる目的でタプルと構造体を対照的に比較し、データをまとめるのに構造体がより良い方法となるのはどういう場合かを示します。

構造体を定義してインスタンス化する方法を実演します。関連関数、特に**メソッド**と呼ばれる種類の関連関数を定義して、構造体型に紐付く振る舞いを指定する方法について議論します。構造体と**enum**(第 6 章で議論します) は、自分のプログラム領域で新しい型を定義し、Rust のコンパイル時型精査機能をフル活用する構成要素になります。

5.1 構造体を定義し、インスタンス化する

構造体は、「タプル型」の節で議論したタプルと、どちらも関係する複数の値を抱えるという点で似ています。タプル同様、構造体の一部を異なる型にできます。一方タプルとは違って、構造体では各データ片には名前をつけるので、値の意味が明確になります。これらの名前が付いていることで、構造体はタプルに比して、より柔軟になるわけです: データの順番に頼って、インスタンスの値を指定したり、アクセスしたりする必要がないのです。

構造体の定義は、`struct` キーワードを入れ、構造体全体に名前を付けます。構造体名は、一つにグループ化されるデータ片の意義を表すものであるべきです。そして、波かっこ内に、データ片の名前と型を定義し、これは**フィールド**と呼ばれます。例えば、リスト 5-1 では、ユーザアカウントに関する情報を保持する構造体を示しています。

ファイル名: src/main.rs

```
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}
#
# fn main() {}
```

リスト 5-1: User 構造体定義

構造体を定義した後に使用するには、各フィールドに対して具体的な値を指定して構造体の**インスタンス**を生成します。インスタンスは、構造体名を記述し、**key: value** ペアを含む波かっこを付け加えることで生成します。ここで、キーはフィールド名、値はそのフィールドに格納したいデータになります。フィールドは、構造体で宣言した通りの順番に指定する必要はありません。換言すると、構造体定義とは、型に対する一般的な雛形のようなものであり、インスタンスは、その雛形を特定のデータで埋め、その型の値を生成するわけです。例えば、リスト 5-2 で示されたように特定のユーザを宣言することができます。

ファイル名: src/main.rs

```
# struct User {
#     active: bool,
#     username: String,
#     email: String,
#     sign_in_count: u64,
# }
#
fn main() {
    let user1 = User {
        active: true,
        username: String::from("someusername123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
    };
}
```

リスト 5-2: User 構造体のインスタンスを生成する

構造体から特定の値を得るには、ドット記法を使います。例えば、このユーザの E メールアドレスにアクセスするには、user1.email を使います。インスタンスが可変であれば、ドット記法を使い特定のフィールドに代入することで値を変更できます。リスト 5-3 では、可変な User インスタンスの email フィールド値を変更する方法を示しています。

ファイル名: src/main.rs

```
# struct User {
#     active: bool,
#     username: String,
#     email: String,
#     sign_in_count: u64,
# }
#
fn main() {
    let mut user1 = User {
        active: true,
        username: String::from("someusername123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
    };

    user1.email = String::from("anotheremail@example.com");
}
```

リスト 5-3: ある `User` インスタンスの `email` フィールド値を変更する

インスタンス全体が可変でなければならないことに注意してください; Rust では、一部のフィールドのみを可変にすることはできないのです。また、あらゆる式同様、構造体の新規インスタンスを関数本体の最後の式として生成して、そのインスタンスを返すことを暗示できます。

リスト 5-4 は、与えられた email と username で `User` インスタンスを生成する `build_user` 関数を示しています。`active` フィールドには `true` 値が入り、`sign_in_count` には値 `1` が入ります。

ファイル名: src/main.rs

```
# struct User {
#     active: bool,
#     username: String,
#     email: String,
#     sign_in_count: u64,
# }
#
fn build_user(email: String, username: String) -> User {
    User {
        active: true,
        username: username,
        email: email,
        sign_in_count: 1,
    }
}
#
# fn main() {
#     let user1 = build_user(
#         String::from("someone@example.com"),
#         String::from("someusername123"),
#     );
```

```
# }
```

リスト 5-4: E メールとユーザ名を取り、User インスタンスを返す build_user 関数

構造体のフィールドと同じ名前を関数の引数にもつけることは筋が通っていますが、email と username というフィールド名と変数を繰り返さなきゃいけないのは、ちょっと面倒です。構造体にもっとフィールドがあれば、名前を繰り返すことはさらに煩わしくなるでしょう。幸運なことに、便利な省略記法があります！

5.1.1　フィールド初期化省略記法を使う

仮引数名と構造体のフィールド名がリスト 5-4 では、全く一緒なので、**フィールド初期化省略**記法を使って build_user を書き換えても、振る舞いは全く同じにしつつ、リスト 5-5 に示したように username と email を繰り返さなくてもよくなります。

ファイル名: src/main.rs

```
# struct User {
#     active: bool,
#     username: String,
#     email: String,
#     sign_in_count: u64,
# }
#
fn build_user(email: String, username: String) -> User {
    User {
        active: true,
        username,
        email,
        sign_in_count: 1,
    }
}
#
# fn main() {
#     let user1 = build_user(
#         String::from("someone@example.com"),
#         String::from("someusername123"),
#     );
# }
```

リスト 5-5: username と email 引数が構造体のフィールドと同名なので、フィールド初期化省略法を使用する build_user 関数

ここで、email というフィールドを持つ User 構造体の新規インスタンスを生成しています。email フィールドを build_user 関数の email 引数の値にセットしたいわけです。email フィールドと email 引数は同じ名前なので、email: email と書くのではなく、email と書くだけで済むのです。

5.1.2　構造体更新記法で他のインスタンスからインスタンスを生成する

他のインスタンスからの値の多くの部分を含みつつ、一部を変更する形で新しいインスタンスを生成できるとしばしば有用です。**構造体更新記法**でそうすることができます。

まず、リスト 5-6 では、更新記法なしで普通に `user2` に新しい `User` インスタンスを生成する方法を示しています。`email` には新しい値をセットしていますが、それ以外にはリスト 5-2 で生成した `user1` の値を使用しています。

ファイル名: src/main.rs

```
# struct User {
#     active: bool,
#     username: String,
#     email: String,
#     sign_in_count: u64,
# }
#
fn main() {
    // --snip--
#
#     let user1 = User {
#         email: String::from("someone@example.com"),
#         username: String::from("someusername123"),
#         active: true,
#         sign_in_count: 1,
#     };

    let user2 = User {
        active: user1.active,
        username: user1.username,
        email: String::from("another@example.com"),
        sign_in_count: user1.sign_in_count,
    };
}
```

リスト 5-6: `user1` の一部の値を使用しつつ、新しい `User` インスタンスを生成する

構造体更新記法を使用すると、リスト 5-7 に示したように、コード量を減らしつつ、同じ効果を達成できます。`..` という記法により、明示的にセットされていない残りのフィールドが、与えられたインスタンスのフィールドと同じ値になるように指定します。

ファイル名: src/main.rs

```
# struct User {
#     active: bool,
#     username: String,
#     email: String,
```

```
#     sign_in_count: u64,
# }
#
fn main() {
    // --snip--
#
#     let user1 = User {
#         email: String::from("someone@example.com"),
#         username: String::from("someusername123"),
#         active: true,
#         sign_in_count: 1,
#     };

    let user2 = User {
        email: String::from("another@example.com"),
        ..user1
    };
}
```

リスト 5-7: 構造体更新記法を使用して、新しい User インスタンス用の値に新しい email をセットしつつ、残りの値は user1 を使う

リスト 5-7 のコードも、email については user1 とは異なる値、username 、active と sign_in_count フィールドについては、user1 と同じ値になるインスタンスを user2 に生成します。..user1 は、残りのフィールドについては user1 の対応するフィールドから値を取る、ということを示すために最後に来る必要がありますが、フィールドについては好きなだけ多く、構造体定義中のフィールドの順序とは無関係に好きな順で、値を指定してかまいません。

構造体更新記法は代入と同様に = を使います; これは、「ムーブによる変数とデータの相互作用」の節で見たのと同じように、データをムーブするからです。この例で言うと、user2 を作成した後は、もう user1 をそっくりそのまま使うことはできません。user1 の username フィールド中の String が user2 の中にムーブされてしまったからです。もし user2 に、email と username のために新しい String 値を与えていたなら、つまり、user1 からは active と sign_in_count の値だけを使用していたなら、user2 を作成した後も user1 はまだ有効だったでしょう。active と sign_in_count はどちらも Copy トレイトを実装した型なので、「スタックのみのデータ: コピー」節で議論した振る舞いが適用されるからです。

5.1.3　異なる型を生成する名前付きフィールドのないタプル構造体を使用する

Rust は、構造体名により追加の意味を含むものの、フィールドに紐づけられた名前がなく、むしろフィールドの型だけの**タプル構造体**と呼ばれる、タプルに似た構造体もサポートしています。タプル構造体は、構造体名が提供する追加の意味は含むものの、フィールドに紐付けられた名前はありません; むしろ、フィールドの型だけが存在します。タプル構造体は、タプル全体に名前をつけ、そのタプルを他のタプルとは異なる型にしたいが、普通の構造体のように各フィールド名を与えるのは、冗

長、または余計という場合に有用です。

タプル構造体を定義するには、struct キーワードの後に構造体名、さらにタプルに含まれる型を続けてください。例えば、ここでは、Color と Point という 2 種類のタプル構造体の定義して使用します:

ファイル名: src/main.rs

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

fn main() {
    let black = Color(0, 0, 0);
    let origin = Point(0, 0, 0);
}
```

black と origin の値は、違う型であることに注目してください。これらは、異なるタプル構造体のインスタンスだからですね。定義された各構造体は、構造体内のフィールドが同じ型であっても、それ自身が独自の型になります。例えば、Color 型を引数に取る関数は、Point を引数に取ることはできません。たとえ、両者の型が、3 つの i32 値からできていてもです。それ以外については、タプル構造体のインスタンスは、分配して個々の部品にしたり、. と添え字を使用して個々の値にアクセスできるという点で、タプルと似ています。

5.1.4　フィールドのないユニット様 (よう) 構造体

また、一切フィールドのない構造体を定義することもできます！ これらは、() 、「タプル型」の節で言及したユニット型と似たような振る舞いをすることから、**ユニット様構造体**と呼ばれます。ユニット様構造体は、ある型にトレイトを実装するけれども、型自体に保持させるデータは一切ない場面に有効になります。トレイトについては第 10 章で議論します。以下は、AlwaysEqual という名前のユニット様構造体を宣言し、インスタンス化する例です:

ファイル名: src/main.rs

```
struct AlwaysEqual;

fn main() {
    let subject = AlwaysEqual;
}
```

AlwaysEqual を定義するためには、struct キーワード、付けたい名前、そしてセミコロンを使います。波括弧や丸括弧は不要です！ 次に、同じようにして、subject 変数に AlwaysEqual のインスタンスを得られます: 波括弧や丸括弧を付けずに、定義した名前を使います。後でこの型の振る舞いを、おそらくはテスト目的で既知の結果を得るために、AlwaysEqual のすべてのインスタンスが常に他の

任意の型と等価であるように実装することを想像してください。この挙動を実装するためにデータはまったく必要ないですね！ トレイトを定義して、ユニット様構造体も含めた任意の型にそれを実装する方法については 10 章で触れます。

<!–

5.1.5 Ownership of Struct Data

In the `User` struct definition in Listing 5-1, we used the owned `String` type rather than the `&str` string slice type. This is a deliberate choice because we want each instance of this struct to own all of its data and for that data to be valid for as long as the entire struct is valid.

It’ s also possible for structs to store references to data owned by something else, but to do so requires the use of **lifetimes**, a Rust feature that we’ ll discuss in Chapter 10. Lifetimes ensure that the data referenced by a struct is valid for as long as the struct is. Let’ s say you try to store a reference in a struct without specifying lifetimes, like the following; this won’ t work:

Filename: src/main.rs

5.1.6 構造体データの所有権

リスト 5-1 の `User` 構造体定義において、`&str` 文字列スライス型ではなく、所有権のある `String` 型を使用しました。これは意図的な選択です。というのも、この構造体の各インスタンスには自身の全データを所有してもらう必要があり、このデータは、構造体全体が有効な間はずっと有効である必要があるのです。

構造体に、他の何かに所有されたデータへの参照を保持させることもできますが、そうするには**ライフタイム**という第 10 章で議論する Rust の機能を使用しなければなりません。ライフタイムのおかげで構造体に参照されたデータが、構造体自体が有効な間、ずっと有効であることを保証してくれるのです。次のように、ライフタイムを指定せずに構造体に参照を保持させようとしたとしましょう。これは動きません:

ファイル名: src/main.rs

```
struct User {
    active: bool,
    username: &str,
    email: &str,
    sign_in_count: u64,
}
```

```
fn main() {
    let user1 = User {
        active: true,
        username: "someusername123",
        email: "someone@example.com",
        sign_in_count: 1,
    };
}
```

コンパイラは、ライフタイム指定子が必要だと怒るでしょう:

```
$ cargo run
   Compiling structs v0.1.0 (file:///projects/structs)
error[E0106]: missing lifetime specifier
(エラー: ライフタイム指定子がありません)
 --> src/main.rs:3:15
  |
3 |     username: &str,
  |               ^ expected named lifetime parameter
  |                (ライフタイム引数を予期しました)
  |
help: consider introducing a named lifetime parameter
  |
1 ~ struct User<'a> {
2 |     active: bool,
3 ~     username: &'a str,
  |

error[E0106]: missing lifetime specifier
 --> src/main.rs:4:12
  |
4 |     email: &str,
  |            ^ expected named lifetime parameter
  |
help: consider introducing a named lifetime parameter
  |
1 ~ struct User<'a> {
2 |     active: bool,
3 |     username: &str,
4 ~     email: &'a str,
  |

For more information about this error, try `rustc --explain E0106`.
error: could not compile `structs` (bin "structs") due to 2 previous errors
```

第 10 章で、これらのエラーを解消して構造体に参照を保持する方法について議論しますが、当面、今回のようなエラーは、`&str` のような参照の代わりに、`String` のような所有された型を使うことで修正します。

5.2　構造体を使ったプログラム例

構造体を使用したくなる可能性のあるケースを理解するために、長方形の面積を求めるプログラムを書きましょう。単一の変数から始め、代わりに構造体を使うようにプログラムをリファクタリングします。

Cargo で **rectangles** という新規バイナリプロジェクトを作成しましょう。このプロジェクトは、長方形の幅と高さをピクセルで指定し、その面積を求めます。リスト 5-8 に、プロジェクトの **src/main.rs** で、正にそうする一例を短いプログラムとして示しました。

ファイル名: src/main.rs

```
fn main() {
    let width1 = 30;
    let height1 = 50;

    println!(
        // 長方形の面積は、{}平方ピクセルです
        "The area of the rectangle is {} square pixels.",
        area(width1, height1)
    );
}

fn area(width: u32, height: u32) -> u32 {
    width * height
}
```

リスト 5-8: 個別の幅と高さ変数を指定して長方形の面積を求める

では、`cargo run` でこのプログラムを走らせてください:

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished dev [unoptimized + debuginfo] target(s) in 0.42s
     Running `target/debug/rectangles`
The area of the rectangle is 1500 square pixels.
(長方形の面積は、1500平方ピクセルです)
```

このコードは、各寸法を与えて `area` 関数を呼び出すことで長方形の面積を割り出すことができますが、このコードはもっと簡潔で読みやすくすることができます。

このコードの問題点は、`area` のシグニチャから明らかです:

```
# fn main() {
#     let width1 = 30;
#     let height1 = 50;
#
#     println!(
```

```
#         // 長方形の面積は、{}平方ピクセルです
#         "The area of the rectangle is {} square pixels.",
#         area(width1, height1)
#     );
# }
#
fn area(width: u32, height: u32) -> u32 {
#     width * height
# }
```

area 関数は、1 長方形の面積を求めるものと考えられますが、今書いた関数には引数が 2 つあり、そしてこのプログラム内のどこを見ても、これらの引数に関連性があることが明確になっていません。幅と高さを一緒にグループ化する方が、より読みやすく、扱いやすくなるでしょう。それをする一つの方法については、第 3 章の「タプル型」節ですでに議論しました: タプルを使うのです。

5.2.1　タプルでリファクタリングする

リスト 5-9 は、タプルを使う別バージョンのプログラムを示しています。

ファイル名: src/main.rs

```
fn main() {
    let rect1 = (30, 50);

    println!(
        "The area of the rectangle is {} square pixels.",
        area(rect1)
    );
}

fn area(dimensions: (u32, u32)) -> u32 {
    dimensions.0 * dimensions.1
}
```

リスト 5-9: タプルで長方形の幅と高さを指定する

ある意味では、このプログラムはマシです。タプルのおかげで少し構造的になり、一引数を渡すだけになりました。しかし別の意味では、このバージョンは明確性を失っています: タプルは要素に名前を付けないので、タプルの要素に添え字でアクセスする必要があり、計算が不明瞭になったのです。

面積計算では幅と高さを混同しても問題ないですが、長方形を画面に描画したいとなると、これは問題になります！ タプルの添え字 `0` が `幅` で、添え字 `1` が `高さ` であることを肝に銘じておかなければなりません。もし他人がこのコードを使用することになったら、彼らがこのことを見つけ出して肝に銘じておくのはより難しくなるでしょう。データの意味をコードに載せていないことで、エラーを招きやすくなってしまいました。

5.2.2　構造体でリファクタリングする: より意味付けする

データのラベル付けで意味を付与するために構造体を使います。現在使用しているタプルを全体と一部に名前のある構造体に、変形することができます。そう、リスト 5-10 に示したように。

ファイル名: src/main.rs

```
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "The area of the rectangle is {} square pixels.",
        area(&rect1)
    );
}

fn area(rectangle: &Rectangle) -> u32 {
    rectangle.width * rectangle.height
}
```

リスト 5-10: `Rectangle` 構造体を定義する

ここでは、構造体を定義し、`Rectangle` という名前にしています。波括弧の中で `width` と `height` というフィールドを定義し、`u32` という型にしました。それから `main` 内で `Rectangle` の特定のインスタンスを生成し、幅を `30` 、高さを `50` にしました。

これで `area` 関数は引数が一つになり、この引数は名前が `rectangle` 、型は `Rectangle` 構造体インスタンスへの不変借用になりました。第 4 章で触れたように、構造体の所有権を奪うよりも借用する必要があります。こうすることで `main` は所有権を保って、`rect1` を使用し続けることができ、そのために関数シグニチャと関数呼び出し時に `&` を使っているわけです。

`area` 関数は、`Rectangle` インスタンスの `width` と `height` フィールドにアクセスしています。(借用された構造体インスタンスのフィールドにアクセスしても、そのフィールドの値はムーブされないことに注意してください。構造体の借用をよく使うのはこのためです) これで、`area` の関数シグニチャは、我々の意図をズバリ示すようになりました: `width` と `height` フィールドを使って、`Rectangle` の面積を計算します。これにより、幅と高さが相互に関係していることが伝わり、タプルの `0` や `1` という添え字を使うよりも、これらの値に説明的な名前を与えられるのです。プログラムの意図が明瞭に

なりました。

5.2.3　トレイトの導出で有用な機能を追加する

プログラムのデバッグをしている間に、Rectangle のインスタンスを出力し、フィールドの値を確認できると便利でしょう。リスト 5-11 では、以前の章のように、println! マクロを試しに使用しようとしています。ですが、これは動きません。

ファイル名: src/main.rs

```
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    // rect1は{}です
    println!("rect1 is {}", rect1);
}
```

リスト 5-11: Rectangle のインスタンスを出力しようとする

このコードをコンパイルすると、こんな感じのエラーが出ます:

```
error[E0277]: `Rectangle` doesn't implement `std::fmt::Display`
(エラー: `Rectangle`は`std::fmt::Display`を実装していません)
```

println! マクロには、様々な整形があり、標準では、波括弧は Display として知られる整形をするよう、println! に指示するのです: 直接エンドユーザ向けの出力です。これまでに見てきた基本型は、標準で Display を実装しています。というのも、1 や他の基本型をユーザに見せる方法は一つしかないからです。しかし構造体では、println! が出力を整形する方法は自明ではなくなります。出力方法がいくつもあるからです: カンマは必要なの？ 波かっこを出力する必要はある？ 全フィールドが見えるべき？ この曖昧性のため、Rust は必要なものを推測しようとせず、構造体は println! と {} プレースホルダで使用される Display 実装を提供しないのです。

エラーを読み下すと、こんな有益な注意書きがあります:

```
    = help: the trait `std::fmt::Display` is not implemented for `Rectangle`
    = note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-
        print) instead
    (ヘルプ: `std::fmt::Display`は`Rectangle`に対して実装されていません)
    (注釈: フォーマット文字列では代わりに`{:?}`(またはpretty-printするためには
```

```
        {:#?})が使用できるかもしれません)
```

試してみましょう！ pritnln! マクロ呼び出しは、println!("rect1 is {:?}", rect1); という見た目になるでしょう。波括弧内に :? という指定子を書くと、println! に Debug と呼ばれる出力整形を使いたいと指示するのです。Debug トレイトは、開発者にとって有用な方法で構造体を出力させてくれるので、コードをデバッグしている最中に、値を確認することができます。

変更してコードをコンパイルしてください。なに！ まだエラーが出ます:

```
error[E0277]: `Rectangle` doesn't implement `Debug`
(エラー: `Rectangle`は`Debug`を実装していません)
```

しかし今回も、コンパイラは有益な注意書きを残してくれています:

```
   = help: the trait `Debug` is not implemented for `Rectangle`
   = note: add `#[derive(Debug)]` to `Rectangle` or manually `impl Debug for
       Rectangle`
   (ヘルプ: トレイト`Debug`は`Rectangle`に対して実装されていません)
   (注釈: `Rectangle`に`#[derive(Debug)]`を追加するか、手動で`impl Debug for
       Rectangle`してください)
```

確かに Rust にはデバッグ用の情報を出力する機能が備わっていますが、この機能を構造体で使えるようにするには、明示的な選択をしなければならないのです。そうするには、構造体定義の直前に #[derive(Debug)] という外部属性を追加します。そう、リスト 5-12 で示されている通りです。

ファイル名: src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!("rect1 is {:?}", rect1);
}
```

リスト 5-12: Debug トレイトを導出する属性を追加し、Rectangle インスタンスをデバッグ用整形機で出力する

これでプログラムを実行すれば、エラーは出ず、以下のような出力が得られるでしょう:

```
$ cargo run
```

```
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished dev [unoptimized + debuginfo] target(s) in 0.48s
     Running `target/debug/rectangles`
rect1 is Rectangle { width: 30, height: 50 }
```

素晴らしい！ 最善の出力ではないものの、このインスタンスの全フィールドの値を出力しているので、デバッグ中には間違いなく役に立つでしょう。より大きな構造体があるなら、もう少し読みやすい出力の方が有用です; そのような場合には、`println!` 文字列中の `{:?}` の代わりに `{:#?}` を使うことができます。この例で `{:#?}` というスタイルを使用したら、出力は以下のようになるでしょう:

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished dev [unoptimized + debuginfo] target(s) in 0.48s
     Running `target/debug/rectangles`
rect1 is Rectangle {
    width: 30,
    height: 50,
}
```

`Debug` 整形を使用して値を出力するためのもう一つの方法は、`dbg!` マクロを使用することです。`dbg!` マクロは式の所有権を奪い（参照を取る `println!` とは対照的です）、その呼び出しが発生したコード内のファイル名と行番号とともに式を評価した結果を出力して、その値の所有権を返します。

> 注釈: 標準出力コンソールストリーム（ `stdout` ）に出力する `println!` とは異なり、`dbg!` マクロの呼び出しは標準エラーコンソールストリーム（ `stderr` ）に出力します。`stderr` と `stdout` については 12 章の「標準出力ではなく標準エラーにエラーメッセージを書き込む」節でより詳しく触れます。

以下は、`width` フィールドに代入される値と、`rect1` の構造体全体の値に関心がある場合の例です:

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let scale = 2;
    let rect1 = Rectangle {
        width: dbg!(30 * scale),
        height: 50,
    };

    dbg!(&rect1);
}
```

`dbg!` は式が評価された値の所有権を返すため、式 `30 * scale` を囲うように `dbg!` を書くことで、

`width` フィールドはここで `dbg!` の呼び出しをしなかった場合とまったく同じ値になります。`rect1` の所有権は奪ってほしくないので、次の `dbg!` 呼び出しでは `rect1` への参照を使用しています。この例の出力は以下のようになります:

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished dev [unoptimized + debuginfo] target(s) in 0.61s
     Running `target/debug/rectangles`
[src/main.rs:10:16] 30 * scale = 60
[src/main.rs:14:5] &rect1 = Rectangle {
    width: 60,
    height: 50,
}
```

出力の前半は **src/main.rs** の 10 行目からの出力です。ここでは式 `30 * scale` をデバッグ出力していて、その結果の値は `60` です（整数に実装されている `Debug` 整形を使用して出力されています)。**src/main.rs** の 14 行目の `dbg!` 呼び出しは `&rect1` の値を出力し、これは `Rectangle` 構造体です。この出力は `Rectangle` 型の pretty `Debug` 整形を使用します。`dbg!` マクロは、コードが何をしているのか理解しようとするときには非常に有用です！

`Debug` トレイトの他にも、Rust では `derive` 属性で使えるトレイトが多く提供されており、独自の型に有用な振る舞いを追加することができます。そのようなトレイトとその振る舞いは、付録 C で一覧になっています。これらのトレイトを独自の動作とともに実装する方法だけでなく、独自のトレイトを生成する方法については、第 10 章で解説します。また、`derive` の他にも多数の属性が存在します; さらなる情報については Rust Reference の “Attributes” 節を参照してください。

`area` 関数は、非常に特殊です: 長方形の面積を算出するだけです。`Rectangle` 構造体とこの動作をより緊密に結び付けられると、役に立つでしょう。なぜなら、他のどんな型でもうまく動作しなくなるからです。`area` 関数を `Rectangle` 型に定義された `area` **メソッド**に変形することで、このコードをリファクタリングし続けられる方法について見ていきましょう。

5.3　メソッド記法

メソッドは関数に似ています: `fn` キーワードと名前を使って宣言され、引数と戻り値を持つことができ、メソッドがどこか別の場所で呼び出された時に実行されるコードを含みます。関数とは異なり、メソッドは構造体の文脈の中で定義されます（enum の文脈やトレイトオブジェクトの文脈の中でも定義されますが、これらについてはそれぞれ 6 章と**??**章で解説します)。最初の引数は必ず `self` になり、これはメソッドが呼び出されている構造体インスタンスを表します。

5.3.1　メソッドを定義する

`Rectangle` インスタンスを引数に取る `area` 関数を変え、代わりに `Rectangle` 構造体上に `area` メソッドを作りましょう。リスト 5-13 に示した通りですね。

ファイル名: src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        // 長方形の面積は{}平方ピクセルです。
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
```

リスト 5-13: `Rectangle` 構造体上に `area` メソッドを定義する

`Rectangle` の文脈内で関数を定義するには、`Rectangle` のための `impl` (implementation; 実装) ブロックを始めます。この `impl` ブロック内のあらゆる定義は `Rectangle` に関連付けられたものとなるでしょう。それから `area` 関数を `impl` の波かっこ内に移動させ、最初の (今回は唯一の) 引数をシグニチャ内と本体内全てで `self` に変えます。`area` 関数を呼び出し、`rect1` を引数として渡す `main` では、代替としてメソッド記法を使用して、`Rectangle` インスタンスの `area` メソッドを呼び出せます。メソッド記法は、インスタンスの後に続きます: ドット、メソッド名、かっこ、そして引数と続くわけです。

`area` のシグニチャでは、`rectangle: &Rectangle` の代わりに `&self` を使用しています。`&self` は実際のところ `self: &Self` の省略記法です。`impl` ブロック内では、型 `Self` はその `impl` ブロックの対象である型に対するエイリアスとして使えます。メソッドはその第 1 引数として `self` という名前の `Self` 型引数を持たなくてはならないので、Rust 処理系は第 1 引数の位置では `self` という名前のみに省略することを許可しているのです。ただし、`self` 省略記法の前の `&` は依然として必要であることに注意してください。これは、`rectangle: &Rectangle` と書いたときと同様に、このメソッドが `Self` のインスタンスを借用することを示しています。メソッドは、`self` の所有権を奪ったり、ここでしているように不変で `self` を借用したり、可変で `self` を借用したりできるのです。他の引数と全く同じですね。

ここで `&self` を選んでいるのは、関数バージョンで `&Rectangle` を使用していたのと同様の理由です: 所有権はいらず、構造体のデータを読み込みたいだけで、書き込む必要はないわけです。メソッドの一部でメソッドを呼び出したインスタンスを変更したかったら、第 1 引数に `&mut self` を使用するでしょう。`self` だけを第 1 引数にしてインスタンスの所有権を奪うメソッドを定義することは稀です; このテクニックは通常、メソッドが `self` を何か別のものに変形し、変形後に呼び出し元が元のインスタンスを使用できないようにしたい場合に使用されます。

関数の代替としてメソッドを使う主な理由は、メソッド記法を使用できるようにすることと、すべてのメソッドのシグニチャでいちいち `self` の型を繰り返す必要がなくなることに加えて、体系化のためです。コードの将来的な利用者に、私たちが提供するライブラリ内のあらゆる箇所から `Rectangle` の機能を探させるのではなく、この型のインスタンスでできることを一つの `impl` ブロックにまとめあげています。

メソッドには、構造体のフィールドと同じ名前を与えることもできることに注意してください。例えば、`Rectangle` に `width` という名前のメソッドを定義することができます:

ファイル名: src/main.rs

```
# #[derive(Debug)]
# struct Rectangle {
#     width: u32,
#     height: u32,
# }
#
impl Rectangle {
    fn width(&self) -> bool {
        self.width > 0
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    if rect1.width() {
        // 長方形は非ゼロの幅を持っています; それは{}です
        println!("The rectangle has a nonzero width; it is {}", rect1.width);
    }
}
```

ここで `width` メソッドは、インスタンスの `width` フィールドの値が `0` より大きい場合に `true` を返し、値が `0` の場合は `false` を返すようにしています: 同名のメソッド内でも、フィールドをあらゆる目的のために使用することができます。`main` では `rect1.width` の後に丸かっこを続けているので、処理系はメソッドの `width` を意図したものと認識します。丸かっこを使わなければ、処理系はフィールドの `width` を意図したものと認識します。

フィールドの値を返すだけで他に何もしないメソッドに、フィールドと同じ名前を与えることが、常にではありませんが、よくあります。このようなメソッドは**ゲッター**と呼ばれます。他のいくつかの言語で行われているような自動的なゲッターの実装を、Rust が構造体フィールドに対して行うことはありません。フィールドを非公開にしながらメソッドは公開するということができ、そうすることで、型の公開 API の一部としてフィールドへの読み込み専用アクセスを提供できるので、ゲッターは有用です。公開と非公開とはなにか、そしてフィールドやメソッドを公開または非公開として指定する方法については、7 章で議論します。

<!–

5.3.2　Where's the -> Operator?

In C and C++, two different operators are used for calling methods: you use `.` if you're calling a method on the object directly and `->` if you're calling the method on a pointer to the object and need to dereference the pointer first. In other words, if `object` is a pointer, `object->something()` is similar to `(*object).something()`.

Rust doesn't have an equivalent to the `->` operator; instead, Rust has a feature called **automatic referencing and dereferencing**. Calling methods is one of the few places in Rust that has this behavior.

Here's how it works: when you call a method with `object.something()`, Rust automatically adds in `&`, `&mut`, or `*` so `object` matches the signature of the method. In other words, the following are the same:

5.3.3　->演算子はどこに行ったの？

C と C++ では、メソッド呼び出しには 2 種類の異なる演算子が使用されます: オブジェクトに対して直接メソッドを呼び出すのなら、. を使用するし、オブジェクトのポインタに対してメソッドを呼び出し、先にポインタを参照外しする必要があるなら、-> を使用するわけです。言い換えると、`object` がポインタなら、`object->something()` は、`(*object).something()` と同等なのです。

Rust には -> 演算子の代わりとなるようなものはありません; その代わり、Rust には、**自動参照および参照外し**という機能があります。Rust においてメソッド呼び出しは、この動作が行われる数少ない箇所なのです。

動作方法はこうです: `object.something()` とメソッドを呼び出すと、コンパイラは `object` がメソッドのシグニチャと合致するように、自動で `&` か `&mut`、`*` を付与するのです。要するに、以下のコードは同じものです:

```
# #[derive(Debug,Copy,Clone)]
```

```
# struct Point {
#     x: f64,
#     y: f64,
# }
#
# impl Point {
#     fn distance(&self, other: &Point) -> f64 {
#         let x_squared = f64::powi(other.x - self.x, 2);
#         let y_squared = f64::powi(other.y - self.y, 2);
#
#         f64::sqrt(x_squared + y_squared)
#     }
# }
# let p1 = Point { x: 0.0, y: 0.0 };
# let p2 = Point { x: 5.0, y: 6.5 };
p1.distance(&p2);
(&p1).distance(&p2);
```

前者の方がずっと明確です。メソッドには自明な受け手 (`self` の型) がいるので、この自動参照機能は動作するのです。受け手とメソッド名が与えられれば、コンパイラは確実にメソッドが読み込み専用 (`&self`) か、書き込みもする (`&mut self`) のか、所有権を奪う (`self`) のか判断できるわけです。メソッドの受け手に関して借用が明示されないというのが、所有権を実際に使うのが Rust において簡単である大きな理由です。

5.3.4　より引数の多いメソッド

`Rectangle` 構造体に 2 番目のメソッドを実装して、メソッドを使う練習をしましょう。今回は `Rectangle` のインスタンスに別の `Rectangle` のインスタンスを受け取らせ、2 番目の `Rectangle` が `self` (1 番目の `Rectangle`) に完全に収まるなら、`true` を返すようにしたいとしましょう; そうでなければ、`false` を返すべきです。つまり、一旦 `can_hold` メソッドを定義したら、リスト 5-14 のようなプログラムを書けるようにしたいのです。

ファイル名: src/main.rs

```
fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };
    let rect2 = Rectangle {
        width: 10,
        height: 40,
    };
    let rect3 = Rectangle {
        width: 60,
        height: 45,
```

```
    };

    println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));
    println!("Can rect1 hold rect3? {}", rect1.can_hold(&rect3));
}
```

リスト 5-14: まだ書いていない `can_hold` メソッドを使用する

予期される出力は以下のようになります。なぜなら、`rect2` の各寸法は `rect1` よりも小さいものの、`rect3` は `rect1` より幅が広いからです:

```
Can rect1 hold rect2? true
Can rect1 hold rect3? false
```

メソッドを定義したいことはわかっているので、`impl Rectangle` ブロック内での話になります。メソッド名は、`can_hold` になり、引数として別の `Rectangle` を不変借用で取るでしょう。メソッドを呼び出すコードを見れば、引数の型が何になるかわかります: `rect1.can_hold(&rect2)` は、`&rect2` 、`Rectangle` のインスタンスである `rect2` への不変借用を渡しています。これは道理が通っています。なぜなら、`rect2` を読み込む (書き込みではなく。この場合、可変借用が必要になります) だけでよく、`can_hold` メソッドを呼び出した後にも `rect2` が使えるよう、所有権を `main` に残したままにしたいからです。`can_hold` の返り値は、`boolean` になり、メソッドの中身は、`self` の幅と高さがもう一つの `Rectangle` の幅と高さよりも、それぞれ大きいことを確認します。リスト 5-13 の `impl` ブロックに新しい `can_hold` メソッドを追記しましょう。リスト 5-15 に示した通りです。

ファイル名: src/main.rs

```
# #[derive(Debug)]
# struct Rectangle {
#     width: u32,
#     height: u32,
# }
#
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }

    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
#
# fn main() {
#     let rect1 = Rectangle {
#         width: 30,
#         height: 50,
#     };
```

```
#     let rect2 = Rectangle {
#         width: 10,
#         height: 40,
#     };
#     let rect3 = Rectangle {
#         width: 60,
#         height: 45,
#     };
#
#     println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));
#     println!("Can rect1 hold rect3? {}", rect1.can_hold(&rect3));
# }
```

リスト 5-15: 別の `Rectangle` のインスタンスを引数として取る `can_hold` メソッドを、`Rectangle` に実装する

このコードをリスト 5-14 の `main` 関数と合わせて実行すると、望み通りの出力が得られます。メソッドは、`self` 引数の後にシグニチャに追加した引数を複数取ることができ、その引数は、関数の引数と同様に動作するのです。

5.3.5　関連関数

`impl` ブロック内で定義されたすべての関数は、`impl` の後に書かれた型に関連付けられているので、**関連関数**と呼ばれます。対象の型のインスタンスを必要としないために、`self` を第 1 引数として持たない（つまりメソッドではない）関連関数を定義することもできます。このような関数は、すでにひとつ使用しています: `String::from` 関数は `String` 型の上に定義された関数です。

メソッドでない関連関数は、構造体の新規インスタンスを返すコンストラクタによく使用されます。これらにはしばしば `new` という名前が付けられますが、`new` は特別な名前ではなく、言語に組み込まれたものでもありません。例えば、一つの寸法を引数に取り、それを高さと幅の両方として使用する関連関数 `square` を提供してもよいでしょう。そうすれば同じ値を 2 回指定する必要なく、正方形の `Rectangle` を生成しやすくすることができます。

ファイル名: src/main.rs

```
# #[derive(Debug)]
# struct Rectangle {
#     width: u32,
#     height: u32,
# }
#
impl Rectangle {
    fn square(size: u32) -> Self {
        Self {
            width: size,
            height: size,
```

```
        }
    }
}
#
# fn main() {
#     let sq = Rectangle::square(3);
# }
```

戻り値型と関数本体内のの `Self` キーワードは `impl` キーワードの後に現れる型に対するエイリアスで、今回の場合は `Rectangle` です。

この関連関数を呼び出すために、構造体名と一緒に `::` 記法を使用します; 一例は `let sq = Rectangle::square(3);` です。この関数は、構造体によって名前空間分けされています: `::` という記法は、関連関数とモジュールによって作り出される名前空間両方に使用されます。モジュールについては第 7 章で議論します。

5.3.6　複数の `impl` ブロック

各構造体には、複数の `impl` ブロックを存在させることができます。例えば、リスト 5-15 はリスト 5-16 に示したコードと等価で、リスト 5-16 では、各メソッドごとに `impl` ブロックを用意しています。

```
# #[derive(Debug)]
# struct Rectangle {
#     width: u32,
#     height: u32,
# }
#
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
#
# fn main() {
#     let rect1 = Rectangle {
#         width: 30,
#         height: 50,
#     };
#     let rect2 = Rectangle {
#         width: 10,
#         height: 40,
#     };
```

```
#     let rect3 = Rectangle {
#         width: 60,
#         height: 45,
#     };
#
#     println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));
#     println!("Can rect1 hold rect3? {}", rect1.can_hold(&rect3));
# }
```

リスト 5-16: 複数の `impl` ブロックを使用してリスト 5-15 を書き直す

ここでこれらのメソッドを個々の `impl` ブロックに分ける理由はないのですが、合法な書き方です。複数の `impl` ブロックが有用になるケースは第 10 章で見ますが、そこではジェネリック型と、トレイトについて議論します。

5.4　まとめ

構造体により、ドメインにとって意味のある独自の型を作成することができます。構造体を使用することで、関連のあるデータ片を相互に結合させたままにし、各部品に名前を付け、コードを明確にすることができます。`impl` ブロック内では型に関連付けられた関数を定義することができ、メソッドは構造体のインスタンスが持つ振る舞いを規定することができる関連関数の一種です。

しかし、構造体だけが独自の型を作成する手段ではありません: Rust の enum 機能に目を向けて、別の道具を道具箱に追加しましょう。

第 6 章

Enum とパターンマッチング

この章では、**列挙型**について見ていきます。列挙型は、**enum** とも称されます。enum は、取りうる**列挙子**を列挙することで、型を定義させてくれます。最初に、enum を定義し、使用して、enum がデータとともに意味をコード化する方法を示します。次に、特別に有用な enum である `Option` について掘り下げていきましょう。この型は、値が何かかなんでもないかを表現します。それから、`match` 式のパターンマッチングにより、どう enum の色々な値に対して異なるコードを走らせやすくなるかを見ます。最後に、`if let` 文法要素も、如何 (いか) に enum をコードで扱う際に使用可能な便利で簡潔な慣用句であるかを解説します。

6.1 Enum を定義する

構造体は、`width` と `height` を持つ `Rectangle` のように、関連するフィールドとデータをひとつにまとめる方法を提供してくれます。一方で enum は、ある値が、とりうる値の集合のうちのいずれかひとつであることを表現する方法を提供するものです。例えば、`Rectangle` は、`Circle` や `Triangle` も含めたとりうる形の集合のいずれかひとつである、と表現したいことがあります。これを達成するために、Rust ではこれらの可能性を enum としてエンコードすることができます。

コードで表現したくなるかもしれない場面に目を向けて、enum が有用でこの場合、構造体よりも適切である理由を確認しましょう。IP アドレスを扱う必要が出たとしましょう。現在、IP アドレスの規格は二つあります: バージョン 4 とバージョン 6 です。これらはプログラムが遭遇する IP アドレスの可能性のすべてですので、取りうる列挙子をすべて**列挙**できます。これが列挙型の名前の由来です。

どんな IP アドレスも、バージョン 4 かバージョン 6 のどちらかになりますが、同時に両方にはなり得ません。IP アドレスのその特性により、enum データ構造が適切なものになります。というのも、enum の値は、その列挙子のいずれか一つにしかなり得ないからです。バージョン 4 とバージョン 6 のアドレスは、どちらも根源的には IP アドレスですから、コードがいかなる種類の IP アドレスにも適用される場面を扱う際には、同じ型として扱われるべきです。

この概念をコードでは、IpAddrKind 列挙型を定義し、IP アドレスがなりうる種類、V4 と V6 を列挙することで、表現できます。これらが enum の列挙子です:

```
enum IpAddrKind {
    V4,
    V6,
}
#
# fn main() {
#     let four = IpAddrKind::V4;
#     let six = IpAddrKind::V6;
#
#     route(IpAddrKind::V4);
#     route(IpAddrKind::V6);
# }
#
# fn route(ip_kind: IpAddrKind) {}
```

これで、IpAddrKind はコードの他の場所で使用できる独自のデータ型になります。

6.1.1　Enum の値

以下のようにして、IpAddrKind の各列挙子のインスタンスは生成できます:

```
# enum IpAddrKind {
#     V4,
#     V6,
# }
#
# fn main() {
    let four = IpAddrKind::V4;
    let six = IpAddrKind::V6;
#
#     route(IpAddrKind::V4);
#     route(IpAddrKind::V6);
# }
#
# fn route(ip_kind: IpAddrKind) {}
```

enum の列挙子は、その識別子の元に名前空間分けされていることと、2 連コロンを使ってその二つを区別していることに注意してください。これが有効な理由は、こうすることで、値 IpAddrKind::V4 と IpAddrKind::V6 という値は両方とも、同じ型 IpAddrKind になったからです。そうしたら、例えば、どんな IpAddrKind を取る関数も定義できるようになります。

```
# enum IpAddrKind {
#     V4,
#     V6,
# }
```

```
#
# fn main() {
#     let four = IpAddrKind::V4;
#     let six = IpAddrKind::V6;
#
#     route(IpAddrKind::V4);
#     route(IpAddrKind::V6);
# }
#
fn route(ip_kind: IpAddrKind) {}
```

そして、この関数をどちらの列挙子に対しても呼び出せます:

```
# enum IpAddrKind {
#     V4,
#     V6,
# }
#
# fn main() {
#     let four = IpAddrKind::V4;
#     let six = IpAddrKind::V6;
#
    route(IpAddrKind::V4);
    route(IpAddrKind::V6);
# }
#
# fn route(ip_kind: IpAddrKind) {}
```

enum の利用には、さらなる利点さえもあります。この IP アドレス型についてもっと考えてみると、現状では、実際の IP アドレスの**データ**を保持する方法がありません。つまり、どんな**種類**であるかを知っているだけです。構造体について第 5 章で学んだばっかりとすると、この問題に対して、あなたはリスト 6-1 のように構造体を使って対処したくなるかもしれません。

```
# fn main() {
    enum IpAddrKind {
        V4,
        V6,
    }

    struct IpAddr {
        kind: IpAddrKind,
        address: String,
    }

    let home = IpAddr {
        kind: IpAddrKind::V4,
        address: String::from("127.0.0.1"),
    };

    let loopback = IpAddr {
```

```
        kind: IpAddrKind::V6,
        address: String::from("::1"),
    };
# }
```

リスト 6-1: IP アドレスのデータと IpAddrKind の列挙子を struct を使って保持する

ここでは、二つのフィールドを持つ IpAddr という構造体を定義しています: IpAddrKind 型 (先ほど定義した enum ですね) の kind フィールドと、String 型の address フィールドです。この構造体のインスタンスが 2 つあります。最初のインスタンスは home で、これには kind として IpAddrKind::V4 があり、紐付けられたアドレスデータは 127.0.0.1 です。2 番目のインスタンスは loopback です。これには kind の値として、IpAddrKind のもう一つの列挙子、V6 があり、アドレス ::1 が紐付いています。構造体を使って kind と address 値を一緒に包んだので、もう列挙子は値と紐付けられています。

しかしながら、enum だけを使って同じ概念を表現するほうがより簡潔です: 構造体の中に enum を持たせるのではなく、enum の各列挙子に直接データを格納することができるのです。この新しい IpAddr の定義は、V4 と V6 列挙子両方に String 値が紐付けられていることを述べています。

```
# fn main() {
    enum IpAddr {
        V4(String),
        V6(String),
    }

    let home = IpAddr::V4(String::from("127.0.0.1"));

    let loopback = IpAddr::V6(String::from("::1"));
# }
```

enum の各列挙子にデータを直接添付できるので、余計な構造体を作る必要は全くありません。またここでは、enum がどう機能するかについての別の詳細も確認することができます: ここで定義した enum の各列挙子の名前は、その enum を構築する関数にもなるのです。つまり、IpAddr::V4() は String 引数を受け取り IpAddr 型のインスタンスを返す関数の呼び出しになるのです。enum を定義した結果として、自動でこのコンストラクタ関数が定義されます。

構造体よりも enum を使うことには、別の利点もあります: 各列挙子に紐付けるデータの型と量は、異なってもいいのです。バージョン 4 の IP アドレスには、常に 0 から 255 の値を持つ 4 つの数値があります。V4 のアドレスは、4 つの u8 型の値として格納するけれども、V6 のアドレスは引き続き、単独の String 型の値で格納したかったとしても、構造体では不可能です。enum なら、こんな場合も容易に対応できます:

```
# fn main() {
    enum IpAddr {
        V4(u8, u8, u8, u8),
        V6(String),
```

```
    }

    let home = IpAddr::V4(127, 0, 0, 1);

    let loopback = IpAddr::V6(String::from("::1"));
# }
```

バージョン 4 とバージョン 6 の IP アドレスを格納するデータ構造を定義する複数の異なる方法を示してきました。しかしながら、蓋を開けてみれば、IP アドレスを格納してその種類をコード化したくなるということは一般的なので、標準ライブラリに使用可能な定義があります！ 標準ライブラリでの `IpAddr` の定義のされ方を見てみましょう: 私たちが定義し、使用したのと全く同じ enum と列挙子がありますが、アドレスデータを二種の異なる構造体の形で列挙子に埋め込み、この構造体は各列挙子用に異なる形で定義されています。

```
struct Ipv4Addr {
    // --略--
}

struct Ipv6Addr {
    // --略--
}

enum IpAddr {
    V4(Ipv4Addr),
    V6(Ipv6Addr),
}
```

このコードは、enum 列挙子内にいかなる種類のデータでも格納できることを描き出しています: 例を挙げれば、文字列、数値型、構造体などです。他の enum を含むことさえできます！ また、標準ライブラリの型は、あなたの想像するよりも複雑ではないことがしばしばあります。

標準ライブラリに `IpAddr` に対する定義は含まれるものの、標準ライブラリの定義をまだ我々のスコープに導入していないので、干渉することなく自分自身の定義を生成して使用できることに注意してください。型をスコープに導入することについては、第 7 章でもっと詳しく言及します。

リスト 6-2 で enum の別の例を見てみましょう: 今回のコードは、幅広い種類の型が列挙子に埋め込まれています。

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
#
# fn main() {}
```

リスト 6-2: 列挙子各々が異なる型と量の値を格納する `Message` enum

この enum には、異なる型の列挙子が 4 つあります:

- `Quit` は関連付けられたデータをまったく持ちません。
- `Move` は構造体のように名前付きのフィールドを持っています。
- `Write` は `String` オブジェクトを 1 個だけ含んでいます。
- `ChangeColor` は 3 個の `i32` 値を含んでいます。

リスト 6-2 のような列挙子を含む enum を定義することは、enum の場合、`struct` キーワードを使わず、全部の列挙子が `Message` 型の元に分類される点を除いて、異なる種類の構造体定義を定義するのと類似しています。以下の構造体も、先ほどの enum の列挙子が保持しているのと同じデータを格納することができるでしょう:

```
struct QuitMessage; // ユニット構造体
struct MoveMessage {
    x: i32,
    y: i32,
}
struct WriteMessage(String); // タプル構造体
struct ChangeColorMessage(i32, i32, i32); // タプル構造体
#
# fn main() {}
```

ですがそれぞれ独自の型を持つ異なる構造体を使ってしまうと、リスト 6-2 で定義した単一の型である `Message` enum を使う場合と比較して、これらの任意の種類のメッセージを取る関数を簡単には定義できません。

enum と構造体にはもう 1 点似通っているところがあります: `impl` を使って構造体にメソッドを定義できるのと全く同様に、enum にもメソッドを定義することができるのです。こちらは、`Message` enum 上に定義できる `call` という名前のメソッドです:

```
# fn main() {
#     enum Message {
#         Quit,
#         Move { x: i32, y: i32 },
#         Write(String),
#         ChangeColor(i32, i32, i32),
#     }
#
    impl Message {
        fn call(&self) {
            // メソッド本体はここで定義されます
        }
    }

    let m = Message::Write(String::from("hello"));
```

```
    m.call();
# }
```

メソッドの本体では、`self` を使用して、メソッドを呼び出した相手の値を取得できるでしょう。この例では、`Message::Write(String::from("hello"))` という値を持つ、変数 `m` を生成したので、これが `m.call()` を走らせた時に、`call` メソッドの本体内で `self` が表す値になります。

非常に一般的で有用な別の標準ライブラリの enum を見てみましょう: `Option` です。

6.1.2　`Option` enum と Null 値に勝る利点

この節では、`Option` のケーススタディを掘り下げていきます。この型も標準ライブラリにより定義されている enum です。この `Option` 型は、値が何かかそうでないかという非常に一般的な筋書きをコード化します。

例えば、空でないリストの最初のアイテムをリクエストすると、値が得られるでしょう。空リストの最初のアイテムをリクエストすると、何も得られないでしょう。この概念を型システムの観点で表現することは、コンパイラが、プログラマが処理すべき場面全てを処理しているかどうかチェックできることを意味します; この機能は、他の言語において、究極的にありふれたバグを阻止することができます。

プログラミング言語のデザインは、しばしばどの機能を入れるかという観点で考えられるが、除いた機能も重要なのです。Rust には、他の多くの言語にはある null 機能がありません。**null** とはそこに何も値がないことを意味する値です。null のある言語において、変数は常に二者択一どちらかの状態になります: null かそうでないかです。

null の開発者であるトニー・ホーア (Tony Hoare) の 2009 年のプレゼンテーション、”Null References: The Billion Dollar Mistake”(Null 参照: 10 億ドルの間違い) では、こんなことが語られています。

> 私はそれを 10 億ドルの失敗と呼んでいます。その頃、私は、オブジェクト指向言語の参照に対する、最初のわかりやすい型システムを設計していました。私の目標は、どんな参照の使用も全て完全に安全であるべきことを、コンパイラにそのチェックを自動で行ってもらって保証することだったのです。しかし、null 参照を入れるという誘惑に打ち勝つことができませんでした。それは、単純に実装が非常に容易だったからです。これが無数のエラーや脆弱性、システムクラッシュにつながり、過去 40 年で 10 億ドルの苦痛や損害を引き起こしたであろうということなのです。

null 値の問題は、null の値を null でない値のように使用しようとしたら、何らかの種類のエラーが出ることです。この null かそうでないかという特性は広く存在するので、この種の間違いを大変犯しやすいのです。

しかしながら、null が表現しようとしている概念は、それでも役に立つものです: null は、何らかの理由で現在無効、または存在しない値のことなのです。

問題は、全く概念にあるのではなく、特定の実装にあるのです。そんな感じなので、Rust には null がありませんが、値が存在するか不在かという概念をコード化する enum ならあります。この enum が `Option<T>` で、以下のように標準ライブラリに定義されています。

```
enum Option<T> {
    None,
    Some(T),
}
```

`Option<T>` は有益すぎて、prelude にさえ含まれています; 明示的にスコープに導入する必要がないのです。そしてその列挙子もまた prelude に含まれています: `Some` と `None` を `Option::` の接頭辞なしに直接使えるわけです。ただ、`Option<T>` enum はそうは言っても、普通の enum であり、`Some(T)` と `None` も `Option<T>` 型のただの列挙子です。

`<T>` という記法は、まだ語っていない Rust の機能です。これはジェネリック型引数で、ジェネリクスについて詳しくは第 10 章で解説します。とりあえず `<T>` の意味に関して知っておく必要があることは、`Option` enum の `Some` 列挙子はあらゆる型のデータを 1 つだけ持つことができ、`T` の代わりに使用される具体的な型に応じて、`Option<T>` 型はそれぞれ異なる型になるということだけです。以下は `Option` 値を使って数値型や文字列型を保持する例です。

```
# fn main() {
    let some_number = Some(5);
    let some_char = Some('e');

    let absent_number: Option<i32> = None;
# }
```

`some_number` の型は `Option<i32>` です。`some_char` の型は `Option<char>` で、これは先ほどとは異なる型です。`Some` 列挙子の中で値を指定しているので、コンパイラはこれらの型を推論することができます。`absent_number` に関しては、コンパイラはジェネリクス適用後の `Option` 型を注釈することを要求します: `None` 値を見ただけでは、それに対応する `Some` 列挙子が保持する型をコンパイラは推論できないからです。ここでは、`absent_number` が `Option<i32>` 型を持つことをコンパイラに伝えています。

`Some` 値がある時、値が存在するとわかり、その値は、`Some` に保持されています。`None` 値がある場合、ある意味、null と同じことを意図します: 有効な値がないのです。では、なぜ `Option<T>` の方が、null よりも少しでも好ましいのでしょうか？

簡潔に述べると、`Option<T>` と `T` (ここで `T` はどんな型でもいい) は異なる型なので、コンパイラが `Option<T>` 値を確実に有効な値かのようには使用させてくれません。例えば、このコードは `i8` を `Option<i8>` に足そうとしているので、コンパイルできません。

```
# fn main() {
    let x: i8 = 5;
    let y: Option<i8> = Some(5);
```

```
    let sum = x + y;
# }
```

このコードを動かしたら、以下のようなエラーメッセージが出ます。

```
$ cargo run
   Compiling enums v0.1.0 (file:///projects/enums)
error[E0277]: cannot add `Option<i8>` to `i8`
(エラー: `Option<i8>`を`i8`に足すことはできません)
 --> src/main.rs:5:17
  |
5 |     let sum = x + y;
  |                 ^ no implementation for `i8 + Option<i8>`
  |                   (`i8 + `Option<i8>`のための実装がありません)
  |
  = help: the trait `Add<Option<i8>>` is not implemented for `i8`
    (ヘルプ: トレイト`Add<Option<i8>>`が`i8`に対して実装されていません)
  = help: the following other types implement trait `Add<Rhs>`:
    (ヘルプ: 以下の型であればトレイト`Add<Rhs>`を実装しています:)
            <i8 as Add>
            <i8 as Add<&i8>>
            <&'a i8 as Add<i8>>
            <&i8 as Add<&i8>>

For more information about this error, try `rustc --explain E0277`.
error: could not compile `enums` (bin "enums") due to 1 previous error
```

なんて強烈な！ 実際に、このエラーメッセージは、`i8` と `Option<i8>` が異なる型なので、足し合わせる方法がコンパイラにはわからないことを意味します。Rust において、`i8` のような型の値がある場合、コンパイラが常に有効な値であることを確認してくれます。この値を使う前に null であることをチェックする必要なく、自信を持って先に進むことができるのです。`Option<i8>` がある時 (あるいはどんな型を扱おうとしていても) のみ、値を保持していない可能性を心配する必要があるわけであり、コンパイラはプログラマが値を使用する前にそのような場面を扱っているか確かめてくれます。

言い換えると、`T` 型の処理を行う前には、`Option<T>` を `T` に変換する必要があるわけです。一般的に、これにより、null の最もありふれた問題の一つを捕捉する一助になります: 実際には null なのに、そうでないかのように想定することです。

値が null でないと誤って想定するリスクを削減することは、コードに確信を持つための助けになります。null になる可能性のある値を保持するには、その値の型を `Option<T>` にすることで明示的に同意しなければなりません。それからその値を使用する際には、値が null である場合を明示的に処理する必要があります。値が `Option<T>` 以外の型であるところ全てにおいて、値が null でないと安全に想定することが**できます**。これは、Rust にとって、意図的な設計上の決定であり、null の普遍性を制限し、Rust コードの安全性を向上させます。

では、`Option<T>` 型の値がある時、その値を使えるようにするには、どのように `Some` 列挙子から `T` 型の値を取り出せばいいのでしょうか？ `Option<T>` には様々な場面で有効に活用できる非常に多くの

メソッドが用意されています; ドキュメントでそれらを確認できます。Option<T> のメソッドに馴染むと、Rust の旅が極めて有益になるでしょう。

一般的に、Option<T> 値を使うには、各列挙子を処理するコードが欲しくなります。Some(T) 値がある場合だけ走る何らかのコードが欲しくなり、このコードが内部の T を使用できます。None 値がある場合だけ走る別のコードが欲しくなり、そちらのコードは T を使用できない状態になります。match 式が、enum とともに使用した時にこれだけの動作をする制御フロー文法要素になります: enum の列挙子によって、違うコードが走り、そのコードがマッチした値の中のデータを使用できるのです。

6.2　match 制御フロー構造

Rust には、一連のパターンに対して値を比較し、マッチしたパターンに応じてコードを実行させてくれる match と呼ばれる、非常に強力な制御フロー構造があります。パターンは、リテラル値、変数名、ワイルドカードやその他多数のもので構成することができます; 第 18 章で、全ての種類のパターンと、その目的については解説します。match のパワーは、パターンの表現力とコンパイラが全てのありうるパターンを処理しているかを確認してくれるという事実に由来します。

match 式をコイン並べ替え装置のようなものと考えてください: コインは、様々なサイズの穴が空いた通路を流れ落ち、各コインは、サイズのあった最初の穴に落ちます。同様に、値は match の各パターンを通り抜け、値が「適合する」最初のパターンで、値は紐付けられたコードブロックに落ち、実行中に使用されるわけです。

せっかくコインについて話したので、それを match を使用する例にとってみましょう！ 数え上げ装置と同じ要領で未知のアメリカコインを一枚取り、どの種類のコインなのか決定し、その価値をセントで返す関数をリスト 6-3 で示したように記述することができます。

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
#
# fn main() {}
```

リスト 6-3: enum とその enum の列挙子をパターンにした match 式

`value_in_cents` 関数内の `match` を噛み砕きましょう。まず、`match` キーワードに続けて式を並べています。この式は今回の場合、値 `coin` です。`if` で使用した条件式と非常に酷似しているみたいですね。しかし、大きな違いがあります: `if` では、条件は論理値に評価される必要がありますが、ここでは、どんな型でも構いません。この例における `coin` の型は、1 行目で定義した `Coin` enum です。

次は、`match` アームです。一本のアームには 2 つの部品があります: パターンと何らかのコードです。今回の最初のアームは `Coin::Penny` という値のパターンであり、パターンと動作するコードを区別する `=>` 演算子が続きます。この場合のコードは、ただの値 `1` です。各アームは次のアームとカンマで区切られています。

この `match` 式が実行されると、結果の値を各アームのパターンと順番に比較します。パターンに値がマッチしたら、そのコードに紐付けられたコードが実行されます。パターンが値にマッチしなければ、コイン並べ替え装置と全く同じように、次のアームが継続して実行されます。必要なだけパターンは存在できます: リスト 6-3 では、`match` には 4 本のアームがあります。

各アームに紐付けられるコードは式であり、マッチしたアームの式の結果が `match` 式全体の戻り値になります。

典型的に、アームのコードが短い場合、波かっこは使用しません。リスト 6-3 では、各アームが値を返すだけなので、これに倣っています。マッチのアームで複数行のコードを走らせたいのなら、波かっこを使わなくてはなりませんが、この場合アームの後のカンマは省略することができます。例えば、以下のコードは、メソッドが `Coin::Penny` とともに呼び出されるたびに「Lucky penny!」と表示しつつ、ブロックの最後の値、`1` を返します。

```
# enum Coin {
#     Penny,
#     Nickel,
#     Dime,
#     Quarter,
# }
#
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => {
            println!("Lucky penny!");
            1
        }
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
#
# fn main() {}
```

6.2.1　値に束縛されるパターン

マッチのアームの別の有益な機能は、パターンにマッチした値の一部に束縛できる点です。こうして、enum の列挙子から値を取り出すことができます。

例として、enum の列挙子の一つを中にデータを保持するように変えましょう。1999 年から 2008 年まで、アメリカは、片側に 50 の州それぞれで異なるデザインをしたクォーターコインを鋳造していました。他のコインは州のデザインがなされることはなかったので、クォーターだけがこのおまけの値を保持します。Quarter 列挙子を変更して、UsState 値が中に保持されるようにすることで enum にこの情報を追加でき、それをしたのがリスト 6-4 のコードになります。

```
#[derive(Debug)] // すぐに州を検査できるように
enum UsState {
    Alabama,
    Alaska,
    // --略--
}

enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}
#
# fn main() {}
```

リスト 6-4: Quarter 列挙子が UsState の値も保持する Coin enum

友人の一人が 50 州全部のクォーターコインを収集しようとしているところを想像しましょう。コインの種類で小銭を並べ替えつつ、友人が持っていない種類だったら、コレクションに追加できるように、各クォーターに関連した州の名前を出力します。

このコードの match 式では、Coin::Quarter 列挙子の値にマッチする state という名の変数をパターンに追加します。Coin::Quarter がマッチすると、state 変数はそのクォーターの state の値に束縛されます。それから、state をそのアームのコードで使用できます。以下のようにですね:

```
# #[derive(Debug)]
# enum UsState {
#     Alabama,
#     Alaska,
#     // --snip--
# }
#
# enum Coin {
#     Penny,
#     Nickel,
```

```
#     Dime,
#     Quarter(UsState),
# }
#
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            println!("State quarter from {:?}!", state);
            25
        }
    }
}
#
# fn main() {
#     value_in_cents(Coin::Quarter(UsState::Alaska));
# }
```

`value_in_cents(Coin::Quarter(UsState::Alaska))` と呼び出すつもりだったなら、`coin` は `Coin::Quarter(UsState::Alaska)` になります。その値を match の各アームと比較すると、`Coin::Quarter(state)` に到達するまで、どれにもマッチしません。その時に、`state` に束縛されるのは、`UsState::Alaska` という値です。そして、`println!` 式でその束縛を使用することができ、そのため、`Coin` enum の列挙子から `Quarter` に対する中身の state の値を取得できたわけです。

6.2.2　Option<T>とのマッチ

前節では、`Option<T>` を使用する際に、`Some` ケースから中身の `T` の値を取得したくなりました。要するに、`Coin` enum に対して行ったように、`match` を使って `Option<T>` を扱うこともできるというわけです！ コインを比較する代わりに、`Option<T>` の列挙子を比較するのですが、`match` 式の動作の仕方は同じままです。

`Option<i32>` を取る関数を書きたくなったとし、中に値があったら、その値に 1 を足すことにしましょう。中に値がなければ、関数は `None` 値を返し、何も処理を試みるべきではありません。

`match` のおかげで、この関数は大変書きやすく、リスト 6-5 のような見た目になります。

```
# fn main() {
    fn plus_one(x: Option<i32>) -> Option<i32> {
        match x {
            None => None,
            Some(i) => Some(i + 1),
        }
    }

    let five = Some(5);
    let six = plus_one(five);
    let none = plus_one(None);
```

```
# }
```

リスト 6-5: `Option<i32>` に `match` 式を使う関数

`plus_one` の最初の実行についてもっと詳しく検証しましょう。`plus_one(five)` と呼び出した時、`plus_one` の本体の変数 `x` は `Some(5)` になります。そして、これをマッチの各アームと比較します:

```
# fn main() {
#     fn plus_one(x: Option<i32>) -> Option<i32> {
#         match x {
            None => None,
#             Some(i) => Some(i + 1),
#         }
#     }
#
#     let five = Some(5);
#     let six = plus_one(five);
#     let none = plus_one(None);
# }
```

`Some(5)` という値は、`None` というパターンにはマッチしませんので、次のアームに処理が移ります:

```
# fn main() {
#     fn plus_one(x: Option<i32>) -> Option<i32> {
#         match x {
#             None => None,
            Some(i) => Some(i + 1),
#         }
#     }
#
#     let five = Some(5);
#     let six = plus_one(five);
#     let none = plus_one(None);
# }
```

`Some(5)` は `Some(i)` にマッチしますか？ しますね！ 列挙子が同じです。`i` は `Some` に含まれる値に束縛されるので、`i` は値 `5` になります。それから、このマッチのアームのコードが実行されるので、`i` の値に 1 を足し、合計の `6` を中身にした新しい `Some` 値を生成します。

さて、`x` が `None` になるリスト 6-5 の 2 回目の `plus_one` の呼び出しを考えましょう。`match` に入り、最初のアームと比較します:

```
# fn main() {
#     fn plus_one(x: Option<i32>) -> Option<i32> {
#         match x {
            None => None,
#             Some(i) => Some(i + 1),
#         }
#     }
```

```
#
#     let five = Some(5);
#     let six = plus_one(five);
#     let none = plus_one(None);
# }
```

マッチします！ 足し算する値がないので、プログラムは停止し、=> の右辺にある None 値が返ります。最初のアームがマッチしたため、他のアームは比較されません。

match と enum の組み合わせは、多くの場面で有効です。Rust コードにおいて、このパターンはよく見かけるでしょう: enum に対し match し、内部のデータに変数を束縛させ、それに基づいたコードを実行します。最初はちょっと巧妙ですが、一旦慣れてしまえば、全ての言語にあってほしいと願うことになるでしょう。一貫してユーザのお気に入りなのです。

6.2.3　マッチは包括的

もう一つ議論する必要のある match の観点があります: アームのパターンはすべての可能性を網羅しなくてはなりません。こんなバージョンの plus_one 関数を考えてください、これにはバグがありコンパイルできないでしょう:

```
# fn main() {
    fn plus_one(x: Option<i32>) -> Option<i32> {
        match x {
            Some(i) => Some(i + 1),
        }
    }
#
#     let five = Some(5);
#     let six = plus_one(five);
#     let none = plus_one(None);
# }
```

None の場合を扱っていないため、このコードはバグを生みます。幸い、コンパイラが捕捉できるバグです。このコードのコンパイルを試みると、こんなエラーが出ます:

```
$ cargo run
   Compiling enums v0.1.0 (file:///projects/enums)
error[E0004]: non-exhaustive patterns: `None` not covered
(エラー: 包括的でないパターン: `None`が網羅されていません)
 --> src/main.rs:3:15
  |
3 |         match x {
  |               ^ pattern `None` not covered
                    (パターン`None`が網羅されていません)
  |
note: `Option<i32>` defined here
 --> /rustc/07dca489ac2d933c78d3c5158e3f43beefeb02ce/library/core/src/option.rs
     :570:1
```

```
 ::: /rustc/07dca489ac2d933c78d3c5158e3f43beefeb02ce/library/core/src/option.rs
     :574:5
  |
  = note: not covered
  = note: the matched value is of type `Option<i32>`
help: ensure that all possible cases are being handled by adding a match arm
    with a wildcard pattern or an explicit pattern as shown
(ヘルプ: ワイルドカードパターンか、以下に示すように明示的なパターンを持つアーム
    を追加することで、すべての可能な場合が確実に処理されるようにしてください)
  |
4 ~             Some(i) => Some(i + 1),
5 ~             None => todo!(),
  |

For more information about this error, try `rustc --explain E0004`.
error: could not compile `enums` (bin "enums") due to 1 previous error
```

全可能性を網羅していないことをコンパイラは検知しています。もっと言えば、どのパターンを忘れているかさえ知っているのです。Rust におけるマッチは、**包括的**です: 全てのあらゆる可能性を網羅し尽くさなければ、コードは有効にならないのです。特に `Option<T>` の場合には、私達が明示的に `None` の場合を処理するのを忘れないようにしてくれます。null になるかもしれないのに値があると思い込まないよう、すなわち前に議論した 10 億ドルの失敗を犯すことができないよう、コンパイラが保護してくれるわけです。

6.2.4　catch-all パターンとプレースホルダー（_）

enum を使用することで、いくつかの特定の値に対して特別な操作を行うが、他のすべての値に対してはデフォルトの操作を行う、ということができます。ゲームを実装しているところを想像してください。サイコロを振って 3 の目が出たら、そのプレイヤーは移動できませんが、代わりにおしゃれな帽子をもらえます。サイコロを振って 7 の目が出たら、そのプレイヤーはおしゃれな帽子を失います。他のすべての値については、そのプレイヤーはゲーム盤上で同じ数だけマスを移動します。以下はこのロジックを実装する `match` です。ただしサイコロを振った結果はランダム値ではなくハードコードされており、また他のすべてのロジックは、それを実際に実装するのは本題ではないので、本体の無い関数によって表現されています:

```
# fn main() {
    let dice_roll = 9;
    match dice_roll {
        3 => add_fancy_hat(),
        7 => remove_fancy_hat(),
        other => move_player(other),
    }

    fn add_fancy_hat() {}
    fn remove_fancy_hat() {}
    fn move_player(num_spaces: u8) {}
```

```
# }
```

最初の 2 つのアームについては、パターンはリテラル値 `3` および `7` です。他のあらゆる可能な値を網羅する最後のアームについては、パターンは変数で、この変数には `other` と名付けることを選びました。`other` アームで実行されるコードは、`move_player` 関数にこの変数を渡すことで、この変数を使用しています。

`u8` が取りうるすべての値を列挙していないにも関わらず、このコードはコンパイルできます。最後のパターンが、個別に列挙していないすべての値にマッチするからです。この catch-all パターンのおかげで、`match` は包括的でなくてはならないという必要条件が満たされます。パターンは順に評価されるので、catch-all アームは最後に書く必要があることに注意してください。catch-all アームを先に書いてしまうと他のアームは絶対に実行されなくなってしまうため、catch-all の後にアームを追加するとコンパイラが警告を発するでしょう！

Rust には、catch-all したいが、catch-all パターン内で値を**使用**したくない時に使用できるパターンもあります: `_` は任意の値にマッチし、その値を束縛しない特別なパターンです。これはコンパイラにその値を使用しないということを伝えるので、コンパイラは未使用の変数についての警告を発しなくなるしょう。

ゲームのルールを変更しましょう: これからは、3 または 7 以外の目を出したら、もう一度サイコロを振らなくてはなりません。catch-all の値を使用する必要がなくなるので、`other` 変数の代わりに `_` を使用するようにコードを変更します:

```
# fn main() {
    let dice_roll = 9;
    match dice_roll {
        3 => add_fancy_hat(),
        7 => remove_fancy_hat(),
        _ => reroll(),
    }

    fn add_fancy_hat() {}
    fn remove_fancy_hat() {}
    fn reroll() {}
# }
```

この例もまた、最後のアームで明示的にすべての他の値を無視しているので、網羅性要件を満たしています; 見落としている場合分けはありません。

最後に、もう一度だけゲームのルールを変更することにします。3 または 7 以外の目を出したら、プレイヤーの番には何も起きません。以下の `_` アームのコードのように、ユニット値（「タプル型」節で説明した空タプル型）を使用することでこれを表現できます:

```
# fn main() {
    let dice_roll = 9;
    match dice_roll {
```

```
        3 => add_fancy_hat(),
        7 => remove_fancy_hat(),
        _ => (),
    }

    fn add_fancy_hat() {}
    fn remove_fancy_hat() {}
# }
```

このコードでは、先の方のアームのパターンにマッチしないあらゆる値は使用せず、この場合にはいかなるコードも実行したくないということを、コンパイラに明示的に伝えています。

パターンとマッチングについては第 18 章でさらに深く取り扱います。ひとまず、match 式ではちょっと長ったらしいという状況で便利かもしれない、if let 構文に進むことにましょう。

6.3　if let で簡潔な制御フロー

if let 記法で if と let をより冗長性の少ない方法で組み合わせ、残りを無視しつつ、一つのパターンにマッチする値を扱うことができます。config_max 変数の Option<u8> 値にマッチするけれど、値が Some 列挙子の時にだけコードを実行したい、リスト 6-6 のプログラムを考えてください。

```
# fn main() {
    let config_max = Some(3u8);
    match config_max {
        Some(max) => println!("The maximum is configured to be {}", max),
        _ => (),
    }
# }
```

リスト 6-6: 値が Some の時だけコードを実行する match

値が Some の場合は、その Some 列挙子の値をパターン内の変数 max に束縛することで、それを出力します。None 値に対しては何もしたくありません。match 式の要件を満たすためには、列挙子を一つだけ処理した後に _ => () を追加しなければなりませんが、これは煩わしい定型コードです。

その代わり、if let を使用してもっと短く書くことができます。以下のコードは、リスト 6-6 の match と同じように振る舞います:

```
# fn main() {
    let config_max = Some(3u8);
    if let Some(max) = config_max {
        println!("The maximum is configured to be {}", max);
    }
# }
```

if let という記法は等号記号で区切られたパターンと式を取り、式が match に与えられ、パター

ンが最初のアームになった match と同じ動作をします。この場合は、パターンは Some(max) で max は Some 内の値に束縛されます。そうすると、対応する match アームの中で max を使用したのと全く同じように、if let ブロックの本体の中で max を使用することができます。値がパターンにマッチしない場合は、if let ブロック内のコードは実行されません。

if let を使うと、タイプ数が減り、インデントも少なくなり、定型コードも減ります。しかしながら、match では強制された包括性チェックを失ってしまいます。match か if let かの選択は、特定の場面でどんなことをしたいかと簡潔性を得ることが包括性チェックを失うのに適切な代償となるかによります。

言い換えると、if let は値が一つのパターンにマッチした時にコードを走らせ、他は無視する match への糖衣構文と考えることができます。

if let では、else を含むこともできます。else に入るコードブロックは、if let と else に等価な match 式の _ の場合に入るコードブロックと同じになります。リスト 6-4 の Coin enum 定義を思い出してください。ここでは、Quarter 列挙子は、UsState の値も保持していましたね。クォーターコインの状態を告げつつ、見かけたクォーター以外のコインの枚数を数えたいなら、以下のように match 式で実現することができるでしょう:

```
# #[derive(Debug)]
# enum UsState {
#     Alabama,
#     Alaska,
#     // --略--
# }
#
# enum Coin {
#     Penny,
#     Nickel,
#     Dime,
#     Quarter(UsState),
# }
#
# fn main() {
#     let coin = Coin::Penny;
    let mut count = 0;
    match coin {
        Coin::Quarter(state) => println!("State quarter from {:?}!", state),
        _ => count += 1,
    }
# }
```

または、以下のように if let と else を使うこともできるでしょう:

```
# #[derive(Debug)]
# enum UsState {
#     Alabama,
#     Alaska,
#     // --略--
```

```
# }
#
# enum Coin {
#     Penny,
#     Nickel,
#     Dime,
#     Quarter(UsState),
# }
#
# fn main() {
#     let coin = Coin::Penny;
    let mut count = 0;
    if let Coin::Quarter(state) = coin {
        println!("State quarter from {:?}!", state);
    } else {
        count += 1;
    }
# }
```

match を使って表現するには冗長的すぎるロジックがプログラムにあるようなシチュエーションに遭遇したら、if let も Rust 道具箱にあることを思い出してください。

6.4　まとめ

これで、enum を使用してワンセットの列挙された値のどれかになりうる独自の型を生成する方法を講義しました。標準ライブラリの Option<T> が型システムを使用して、エラーを回避する際に役立つ方法についても示しました。enum の値がデータを内部に含む場合、処理すべきケースの数に応じて、match か if let を使用して値を取り出し、使用できます。

もう Rust プログラムで構造体と enum を使用して、自分の領域の概念を表現できます。API 内で使用するために独自の型を生成することで、型安全性を保証することができます: コンパイラが、各関数の予期する型の値のみを関数が得ることを確かめてくれるのです。

使用するのに率直な整理整頓された API をユーザに提供し、ユーザが必要とするものだけを公開するために、今度は、Rust のモジュールに目を向けてみましょう。

第 7 章

肥大化していくプロジェクトをパッケージ、クレート、モジュールを利用して管理する

書いているプログラムが大きいときほど、コードを整理しておくことはますます重要になります。関係した機能をまとめ、異なる特徴を持つコードを分割することにより、特定の機能を実装しているコードを見つけたり、機能を変更したりするためにどこを探せば良いのかを明確にできます。

私達がこれまでに書いてきたプログラムは、一つのファイル内の一つのモジュール内にありました。プロジェクトが大きくなるにつれて、これを複数のモジュールに、ついで複数のファイルに分割することで、プログラムを整理するべきです。パッケージは複数のバイナリクレートからなり、またライブラリクレートを 1 つもつこともできます。パッケージが大きくなるにつれて、その一部を抜き出して分離したクレートにし、外部依存とするのもよいでしょう。この章ではそれらのテクニックすべてを学びます。相互に関係し合い、同時に成長するパッケージの集まりからなる巨大なプロジェクトには、Cargo が**ワークスペース**という機能を提供します。これは 14 章の Cargo ワークスペースで解説します。

実装の詳細をカプセル化することについても議論します。カプセル化によって、コードをより高いレベルで再利用できるようになります：手続きを実装し終えてしまえば、他のコードはその公開されたインターフェースを通じて、実装の詳細を知る必要なくそのコードを呼び出すことができるのです。コードをどう書くかによって、どの部分が他のコードにも使える公開のものになるのか、それとも自分だけが変更できる非公開のものになるのかが決定されます。これもまた、記憶しておくべき細部を制限してくれる方法のひとつです。

関係する概念にスコープがあります：コードが記述されているネストされた文脈には、「スコープ内」として定義される名前の集合があります。コードを読んだり書いたりコンパイルしたりする時には、プログラマーやコンパイラは特定の場所にある特定の名前が、変数・関数・構造体・enum・モ

ジュール・定数・その他のどの要素を表すのか、そしてその要素は何を意味するのかを知る必要があります。そこでスコープを作り、どの名前がスコープ内/スコープ外にあるのかを変更することができます。同じ名前のものを 2 つ同じスコープ内に持つことはできません。そこで、名前の衝突を解決するための方法があります。

Rust には、どの詳細を公開するか、どの詳細を非公開にするか、どの名前がプログラムのそれぞれのスコープにあるか、といったコードのまとまりを保つためのたくさんの機能があります。これらの機能は、まとめて「モジュールシステム」と呼ばれることがあり、以下のようなものが含まれます。

- **パッケージ:** クレートをビルドし、テストし、共有することができる Cargo の機能
- **クレート:** ライブラリか実行可能ファイルを生成する、木構造をしたモジュール群
- **モジュールと use:** これを使うことで、パスの構成、スコープ、公開するか否かを決定できます
- **パス:** 要素（例えば構造体や関数やモジュール）に名前をつける方法

この章では、これらの機能をすべて学び、これらがどう相互作用するかについて議論し、これらをどう使ってスコープを制御するのかについて説明します。この章を読み終わる頃には、モジュールシステムをしっかりと理解し、熟練者のごとくスコープを扱うことができるようになっているでしょう！

7.1　パッケージとクレート

最初に学ぶモジュールシステムの要素は、パッケージとクレートです。

クレート (crate) は、Rust コンパイラが一度に考慮する最小限のコードです。`cargo` を実行するのではなく、（ちょうど第 1 章の「Rust プログラムを書いて走らせる」節でやったように）単一のソースコードファイルを渡して `rustc` を実行した場合でも、コンパイラはそのファイルをクレートとしてみなします。クレートはモジュールを含むことができ、モジュールは、クレートとともにコンパイルされる他のファイル内で定義することができます。このことについてはこれからの節で見ていきます。

クレートは 2 種類の形態のうちいずれかです: バイナリクレートか、ライブラリクレートです。**バイナリクレート (binary crate)** は、コマンドラインプログラムやサーバなどの実行可能形式にコンパイルされ、実行することができるプログラムです。どのバイナリクレートも、その実行可能形式が実行されたときに何が起きるかを定義する `main` と呼ばれる関数を持たなくてはなりません。今まで作ってきたクレートはすべてバイナリクレートでした。

ライブラリクレート (library crate) は `main` 関数を持たず、実行可能形式へとコンパイルされません。代わりに、複数のプロジェクトで共有されることを想定した機能を定義するものです。例えば、第 2 章で使用した `rand` クレートは乱数を生成する機能を提供します。Rustacean が「クレート」と言うときの、ほとんどの場合それはライブラリクレートのことであり、Rustacean は「クレート」を一般的なプログラミング概念の「ライブラリ」と同じ意味で使用します。

クレートルート (crate root) とは、Rust コンパイラの開始点となり、クレートのルートモジュールを作るソースファイルのことです（モジュールについて詳しくは「モジュールを定義して、スコープとプライバシーを制御する」のセクションで説明します）。

パッケージ (package) はある機能群を提供する 1 つ以上のクレートのまとまりです。パッケージは **Cargo.toml** という、それらのクレートをどのようにビルドするかを説明するファイルを持っています。Cargo は実のところパッケージで、今までコードをビルドするために使ってきたコマンドラインツールのためのバイナリクレートを含んでいます。Cargo パッケージは、このバイナリクレートが依存するライブラリクレートも含んでいます。他のプロジェクトは Cargo ライブラリクレートに依存することで、Cargo コマンドラインツールが使用するのと同じロジックを使用することもできます。

パッケージは好きなだけバイナリクレートを持つことができますが、ライブラリクレートは最大で 1 個しか持つことができません。パッケージはライブラリクレートかバイナリクレートを問わず、少なくとも 1 個のクレートを持っていないといけません。

パッケージを作る時に何が起こるか見てみましょう。まず、`cargo new` というコマンドを入力します：

```
$ cargo new my-project
     Created binary (application) `my-project` package
$ ls my-project
Cargo.toml
src
$ ls my-project/src
main.rs
```

`cargo new` を実行した後、Cargo が作成したものを確認するために `ls` を使っています。プロジェクトディレクトリ内には **Cargo.toml** ファイルがあり、これがパッケージを構成します。また、**main.rs** を含む **src** ディレクトリもあります。**Cargo.toml** をテキストエディタで開くと、**src/main.rs** については何も書いていないことに気づくでしょう。Cargo は **src/main.rs** が、パッケージと同じ名前を持つバイナリクレートのクレートルートであるという慣習に従っています。同じように、Cargo はパッケージディレクトリに **src/lib.rs** が含まれていたら、パッケージにはパッケージと同じ名前のライブラリクレートが含まれており、**src/lib.rs** がそのクレートルートなのだと判断します。Cargo はクレートルートファイルを `rustc` に渡し、ライブラリやバイナリをビルドします。

今、このパッケージには **src/main.rs** しか含まれておらず、つまりこのパッケージは `my-project` という名前のバイナリクレートのみを持っているということです。もしパッケージが **src/main.rs** と **src/lib.rs** を持っていたら、クレートは 2 つになります：どちらもパッケージと同じ名前を持つ、バイナリクレートとライブラリクレートです。ファイルを **src/bin** ディレクトリに置くことで、パッケージは複数のバイナリクレートを持つことができます。それぞれのファイルが別々のバイナリクレートになります。

7.2　モジュールを定義して、スコープとプライバシーを制御する

この節では、モジュールと、その他のモジュールシステムの要素——すなわち、要素に名前をつけるための**パス**、パスをスコープに持ち込む `use` キーワード、要素を公開する `pub` キーワード——につ

いて学びます。また、as キーワード、外部パッケージ、glob 演算子についても話します。

まずは、将来コードを整理するときの簡単なリファレンスとして、規則の一覧を示します。その後で各規則を詳細に説明します。

7.2.1　モジュールのチートシート

以下に、モジュール、パス、use キーワード、そして pub キーワードがコンパイラ内でどう機能するか、そして多くの開発者はどのようにコードを整理するかについての、クイックリファレンスを提供します。この章ではこれらの各規則の実例を見ていきますが、ここはモジュールがどう機能するか思い出すために見直すのに良い場所となるでしょう。

- **クレートルートから始める**: クレートをコンパイルするとき、コンパイラはまずコンパイル対象のコードとしてクレートルートファイル（通常は、ライブラリクレートでは **src/lib.rs**、バイナリクレートでは **src/main.rs**）の中を探します。
- **モジュールを宣言する**: クレートルートファイルの中で、新しいモジュールを宣言することができます; 例えば、`mod garden;` として “garden” モジュールを宣言したとします。コンパイラは以下の場所からモジュールのコードを探します:
  - インライン。`mod garden` の後のセミコロンが波かっこで置き換えられているとき、その中
  - **src/garden.rs** ファイルの中
  - **src/garden/mod.rs** ファイルの中
- **サブモジュールを宣言する**: クレートルート以外のすべてのファイルの中で、サブモジュールを宣言することができます。例えば、**src/garden.rs** 内で `mod vegetables;` と宣言することができます。コンパイラはサブモジュールのコードを、親モジュールに対応するディレクトリ内の以下の場所から探します:
  - インライン。`mod vegetables` のすぐ後にセミコロンの代わりに波かっこがあるとき、その中
  - **src/garden/vegetables.rs** ファイルの中
  - **src/garden/vegetables/mod.rs** ファイルの中
- **モジュール内のコードへのパス**: モジュールがクレートの一部となったら、プライバシー規則が許す限り同じクレート内のどこからでも、そのモジュール内のコードをコードへのパスを使用して参照できます。例えば、garden vegetables モジュール内の `Asparagus` 型は `crate::garden::vegetables::Asparagus` で参照できます。
- **非公開と公開**: モジュール内のコードはデフォルトでは非公開で、親モジュールからアクセスすることができません。モジュールを公開にするには、`mod` ではなく `pub mod` で宣言してください。公開モジュール内の要素も公開にするには、それらの宣言の前に `pub` を付けてください。
- **`use` キーワード**: `use` キーワードは、長いパスの繰り返しを減らすために、要素へのショートカットをスコープ内に作成します。`crate::garden::vegetables::Asparagus` を参照できるスコープ内であれば、`use crate::garden::vegetables::Asparagus;` でショートカットを作成で

き、以降はそのスコープ内ではその型を使用するためには Asparagus とだけ書けばよくなります。

これらの規則を説明するための例として、ここに backyard という名前のバイナリクレートを作成します。クレートのディレクトリは、同じく backyard と名付けられ、以下のファイルとディレクトリを含んでいます:

```
backyard
├── Cargo.lock
├── Cargo.toml
└── src
    ├── garden
    │   └── vegetables.rs
    ├── garden.rs
    └── main.rs
```

この場合のクレートルートファイルは **src/main.rs** であり、以下の内容を含んでいます:

ファイル名: src/main.rs

```
use crate::garden::vegetables::Asparagus;

pub mod garden;

fn main() {
    let plant = Asparagus {};
    println!("I'm growing {:?}!", plant);
}
```

pub mod garden; の行は、コンパイラに **src/garden.rs** で見つかるコードを含めるように指示します。その内容は:

ファイル名: src/garden.rs

```
pub mod vegetables;
```

この pub mod vegetables; は **src/garden/vegetables.rs** のコードも含めると言う意味です。そのコードは:

```
#[derive(Debug)]
pub struct Asparagus {}
```

それでは、これらのルールの詳細を知り、実際に試してみましょう！

7.2.2　関連するコードをモジュールにまとめる

モジュール (module) はクレート内のコードを整理し、可読性と再利用性を上げるのに役に立ちます。モジュール内のコードはデフォルトでは非公開なので、モジュールは要素の**プライバシー (priavacy)** の制御も可能にします。非公開の要素は、外部からの使用ができない内部的な実装の詳細です。モジュールとそれらの中の要素は公開にするかどうかを選ぶことができ、公開にすると、外部のコードがそれを使用し、それに依存できるようになります。

例えば、レストランの機能を提供するライブラリクレートを書いてみましょう。レストランの実装ではなくコードの関係性に注目したいので、関数にシグネチャをつけますが中身は空白のままにします。

レストラン業界では、レストランの一部を**接客部門 (front of house)** といい、その他を**後方部門 (back of house)** といいます。接客部門とはお客さんがいるところです。接客係がお客様を席に案内し、給仕係が注文と支払いを受け付け、バーテンダーが飲み物を作ります。後方部門とはシェフや料理人がキッチンで働き、皿洗い係が食器を片付け、マネージャが管理業務をする場所です。

私達のクレートをこのような構造にするために、関数をネストしたモジュールにまとめましょう。`restaurant` という名前の新しいライブラリを `cargo new --lib restaurant` と実行することで作成し、リスト 7-1 のコードを **src/lib.rs** に書き込み、モジュールと関数のシグネチャを定義してください。以下が接客部門です:

ファイル名: src/lib.rs

```
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}

        fn seat_at_table() {}
    }

    mod serving {
        fn take_order() {}

        fn serve_order() {}

        fn take_payment() {}
    }
}
```

リスト 7-1: `front_of_house` モジュールにその他のモジュールが含まれ、さらにそれらが関数を含んでいる

モジュールは、`mod` キーワードと、それに続くモジュールの名前（今回の場合、`front_of_house` ）によって定義されます。次にモジュールの本体が波かっこの中に入ります。モジュールの中には、今

回だと hosting と serving のように、他のモジュールをおくこともできます。モジュールにはその他の要素の定義も置くことができます。例えば、構造体、enum、定数、トレイト、そして（リスト 7-1 のように）関数です。

モジュールを使うことで、関連する定義を一つにまとめ、関連する理由を名前で示せます。このコードを使うプログラマーは、定義を全部読むことなく、グループ単位でコードを読み進められるので、欲しい定義を見つけ出すのが簡単になるでしょう。このコードに新しい機能を付け加えるプログラマーは、プログラムのまとまりを保つために、どこにその機能のコードを置けば良いのかがわかるでしょう。

以前、**src/main.rs** と **src/lib.rs** はクレートルートと呼ばれていると言いました。この名前のわけは、**モジュールツリー**と呼ばれるクレートのモジュール構造の根っこ（ルート）にこれら 2 つのファイルの中身が crate というモジュールを形成するからです。

リスト 7-2 は、リスト 7-1 の構造のモジュールツリーを示しています。

```
crate
 └── front_of_house
     ├── hosting
     │   ├── add_to_waitlist
     │   └── seat_at_table
     └── serving
         ├── take_order
         ├── serve_order
         └── take_payment
```

リスト 7-2: リスト 7-1 のコードのモジュールツリー

このツリーを見ると、どのモジュールがどのモジュールの中にネストしているのかがわかります; 例えば、hosting は front_of_house の中にネストしています。また、いくつかのモジュールはお互いに**兄弟 (siblings)** の関係にある、つまり、同じモジュール内で定義されていることもわかります; 例えば hosting と serving は front_of_house で定義されている兄弟同士です。他にも、モジュール A がモジュール B の中に入っている時、A は B の**子 (child)** であるといい、B は A の**親 (parent)** であるといいます。モジュールツリー全体が、暗黙のうちに作られた crate というモジュールの下にあることにも注目してください。

モジュールツリーを見ていると、コンピュータのファイルシステムのディレクトリツリーを思い出すかもしれません。その喩えはとても適切です！ ファイルシステムのディレクトリのように、モジュールはコードをまとめるのに使われるのです。そしてディレクトリからファイルを見つけるように、目的のモジュールを見つけ出す方法が必要になります。

7.3　モジュールツリーの要素を示すためのパス

ファイルシステムの中を移動する時と同じように、Rust にモジュールツリー内の要素を見つけるためにはどこを探せばいいのか教えるためにパスを使います。関数を呼び出すためには、そのパスを

知っていなければなりません。

パスは 2 つの形を取ることができます：

- **絶対パス**は、クレートルートを起点とするフルパスです; 絶対パスは、外部のクレートに属するコードに関してはそのクレート名で始まり、現在のクレートに属するコードに関してはリテラル `crate` で始まります。
- **相対パス**は、現在のモジュールを起点とし、`self` 、`super` 、または今のモジュール内の識別子を使います。

絶対パスも相対パスも、その後に一つ以上の識別子がダブルコロン (`::`) で仕切られて続きます。

リスト 7-1 に戻ってみて、例えば `add_to_waitlist` 関数を呼びたいとしましょう。これはこう聞くのと同じです: `add_to_waitlist` のパスは何でしょうか？ リスト 7-3 は、リスト 7-1 からいくつかのモジュールと関数を削除したものを含んでいます。

これを使って、クレートルートに定義された新しい `eat_at_restaurant` という関数から、`add_to_waitlist` 関数を呼び出す 2 つの方法を示しましょう。これらのパスは正しいものですが、この例をこのままではコンパイルできなくしている問題が他に残っています。理由はすぐに説明します。

`eat_at_restaurant` 関数はこのライブラリクレートの公開 (public) API の 1 つなので、`pub` キーワードをつけておきます。`pub` については、「パスを `pub` キーワードで公開する」の節でより詳しく学びます。

ファイル名: src/lib.rs

```
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // Absolute path
    // 絶対パス
    crate::front_of_house::hosting::add_to_waitlist();

    // Relative path
    // 相対パス
    front_of_house::hosting::add_to_waitlist();
}
```

リスト 7-3: `add_to_waitlist` 関数を絶対パスと相対パスで呼び出す

`eat_at_restaurant` で最初に `add_to_waitlist` 関数を呼び出す時、絶対パスを使っています。`add_to_waitlist` 関数は `eat_at_restaurant` と同じクレートで定義されているので、`crate` キーワードで絶対パスを始めることができます。続けて、`add_to_waitlist` にたどり着くまで、後に続

くモジュールを書き込んでいます。同じ構造のファイルシステムを想像してもよいでしょう: `/front_of_house/hosting/add_to_waitlist` とパスを指定して `add_to_waitlist` を実行していることに相当します。`crate` という名前を使ってクレートルートからスタートするというのは、`/` を使ってファイルシステムのルートからスタートするようなものです。

`eat_at_restaurant` で 2 回目に `add_to_waitlist` 関数を呼び出す時、相対パスを使っています。パスは、モジュールツリーにおいて `eat_at_restaurant` と同じ階層で定義されているモジュールである `front_of_house` からスタートします。これはファイルシステムで `front_of_house/hosting/add_to_waitlist` というパスを使っているのに相当します。モジュール名から始めるのは、パスが相対パスであることを意味します。

相対パスを使うか絶対パスを使うかは、プロジェクトによって決めましょう。要素を定義するコードを、その要素を使うコードと別々に動かすか一緒に動かすか、どちらが起こりそうかによって決めるのが良いです。例えば、`front_of_house` モジュールと `eat_at_restaurant` 関数を `customer_experience` というモジュールに移動させると、`add_to_waitlist` への絶対パスを更新しないといけませんが、相対パスは有効なままです。しかし、`eat_at_restaurant` 関数だけを `dining` というモジュールに移動させると、`add_to_waitlist` への絶対パスは同じままですが、相対パスは更新しないといけないでしょう。一般論としては、コードの定義と、その要素の呼び出しは独立に動かしたいことが多いと思われるので、絶対パスのほうが好ましいです。

では、リスト 7-3 をコンパイルしてみて、どうしてこれはまだコンパイルできないのか考えてみましょう！ エラーをリスト 7-4 に示しています。

```
$ cargo build
   Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0603]: module `hosting` is private
 --> src/lib.rs:9:28
  |
9 |     crate::front_of_house::hosting::add_to_waitlist();
  |                            ^^^^^^^  --------------- function `
      add_to_waitlist` is not publicly re-exported
  |                            |
  |                            private module
  |
note: the module `hosting` is defined here
 --> src/lib.rs:2:5
  |
2 |     mod hosting {
  |     ^^^^^^^^^^^

error[E0603]: module `hosting` is private
  --> src/lib.rs:12:21
   |
12 |     front_of_house::hosting::add_to_waitlist();
   |                     ^^^^^^^  --------------- function `add_to_waitlist` is
       not publicly re-exported
   |                     |
   |                     private module
```

```
   |
note: the module `hosting` is defined here
  --> src/lib.rs:2:5
   |
2  |     mod hosting {
   |     ^^^^^^^^^^^

For more information about this error, try `rustc --explain E0603`.
error: could not compile `restaurant` (lib) due to 2 previous errors
```

リスト 7-4: リスト 7-3 のコードをビルドしたときのコンパイルエラー

エラーメッセージは、hosting は非公開 (private) だ、と言っています。言い換えるなら、hosting モジュールと add_to_waitlist 関数へのパスは正しいが、非公開な部分へのアクセスは許可されていないので、Rust がそれを使わせてくれないということです。Rust では、すべての要素（関数、メソッド、構造体、enum、モジュール、そして定数）はデフォルトでは親モジュールに対して非公開です。関数や構造体などの要素を非公開にしたければ、モジュールの中に置いてください。

親モジュールの要素は子モジュールの非公開要素を使えませんが、子モジュールの要素はその祖先モジュールの要素を使えます。これは、子モジュールは実装の詳細を覆い隠しますが、子モジュールは自分の定義された文脈を見ることができるためです。レストランの喩えを続けるなら、レストランの後方部門になったつもりでプライバシーのルールを考えてみてください。レストランの顧客にはそこで何が起こっているのかは非公開ですが、そこを運営するオフィスマネージャには、レストランのことは何でも見えるし何でもできるのです。

Rust は、内部実装の詳細を隠すことが標準であるようにモジュールシステムを機能させることを選択しました。こうすることで、内部のコードのどの部分が、外部のコードを壊すことなく変更できるのかを知ることができます。しかし、Rust は pub キーワードを使って要素を公開することで、子モジュールの内部部品を外部の祖先モジュールに見せるという選択肢も与えています。

7.3.1　パスを pub キーワードで公開する

リスト 7-4 の、hosting モジュールが非公開だと言ってきていたエラーに戻りましょう。親モジュールの eat_at_restaurant 関数が子モジュールの add_to_waitlist 関数にアクセスできるようにしたいので、hosting モジュールに pub キーワードをつけます。リスト 7-5 のようになります。

ファイル名: src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
```

```
    // Absolute path
    // 絶対パス
    crate::front_of_house::hosting::add_to_waitlist();

    // Relative path
    // 相対パス
    front_of_house::hosting::add_to_waitlist();
}
```

リスト 7-5: hosting モジュールを pub として宣言することで eat_at_restaurant から使う

残念ながら、リスト 7-5 のコードもリスト 7-6 に示されるようにエラーとなります。

```
$ cargo build
   Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0603]: function `add_to_waitlist` is private
 --> src/lib.rs:9:37
  |
9 |     crate::front_of_house::hosting::add_to_waitlist();
  |                                     ^^^^^^^^^^^^^^^ private function
  |
note: the function `add_to_waitlist` is defined here
 --> src/lib.rs:3:9
  |
3 |         fn add_to_waitlist() {}
  |         ^^^^^^^^^^^^^^^^^^^^

error[E0603]: function `add_to_waitlist` is private
  --> src/lib.rs:12:30
   |
12 |     front_of_house::hosting::add_to_waitlist();
   |                              ^^^^^^^^^^^^^^^ private function
   |
note: the function `add_to_waitlist` is defined here
  --> src/lib.rs:3:9
   |
3  |         fn add_to_waitlist() {}
   |         ^^^^^^^^^^^^^^^^^^^^

For more information about this error, try `rustc --explain E0603`.
error: could not compile `restaurant` (lib) due to 2 previous errors
```

リスト 7-6: リスト 7-5 のコードをビルドしたときのコンパイルエラー

何が起きたのでしょう？ pub キーワードを mod hosting の前に追加したことで、このモジュールは公開されました。この変更によって、front_of_house にアクセスできるなら、hosting にもアクセスできるようになりました。しかし hosting の**中身**はまだ非公開です。モジュールを公開してもその中身は公開されないのです。モジュールに pub キーワードがついていても、祖先モジュールのコードはモジュールを参照できるようになるだけで、その内部のコードへのアクセスは許可されません。モ

ジュールはコンテナなので、モジュールを公開しただけでは、できることはあまりありません; さらに先へ進み、モジュール内のひとつまたは複数の要素も公開することを選択する必要があります。

リスト 7-6 のエラーは add_to_waitlist 関数が非公開だと言っています。プライバシーのルールは、モジュール同様、構造体、enum、関数、メソッドにも適用されるのです。

add_to_waitlist の定義の前に pub キーワードを追加して、これも公開しましょう。

ファイル名: src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // Absolute path
    // 絶対パス
    crate::front_of_house::hosting::add_to_waitlist();

    // Relative path
    // 相対パス
    front_of_house::hosting::add_to_waitlist();
}
```

リスト 7-7:　pub キーワードを mod hosting と fn add_to_waitlist に追加することで、eat_at_restaurant からこの関数を呼べるようになる

これでこのコードはコンパイルできます！ どうして pub キーワードを追加することで add_to_waitlist のそれらのパスを使えるようになるのか、プライバシールールの観点から確認するために、絶対パスと相対パスを見てみましょう。

絶対パスは、クレートのモジュールツリーのルートである crate から始まります。クレートルートの中に front_of_house が定義されています。front_of_house は公開されていませんが、eat_at_restaurant 関数は front_of_house と同じモジュール内で定義されている（つまり、eat_at_restaurant と front_of_house は兄弟な）ので、eat_at_restaurant から front_of_house を参照することができます。次は pub の付いた hosting モジュールです。hosting の親モジュールにアクセスできるので、hosting にもアクセスできます。最後に、add_to_waitlist 関数は pub が付いており、私達はその親モジュールにアクセスできるので、この関数呼び出しはうまく行くというわけです。

相対パスについても、最初のステップを除けば同じ理屈です。パスをクレートルートから始めるのではなくて、front_of_house から始めるのです。front_of_house モジュールは eat_at_restaurant と同じモジュールで定義されているので、eat_at_restaurant が定義されている場所からの相対パスが使えます。そして、hosting と add_to_waitlist は pub が付いていますから、残りのパスについても問題はなく、この関数呼び出しは有効というわけです。

他のプロジェクトからあなたのコードを利用できるようにライブラリクレートを共有するつもりなら、公開 API は、クレート利用者があなたのコードとどう相互作用できるかを決定する、クレート利用者との契約となります。人々があなたのクレートに依存しやすくするためには、公開 API への変更の管理に関する多数の考慮事項があります。これらの考慮事項はこの本のスコープ外です; このトピックに興味がある場合は、The Rust API Guidelines を参照してください。

7.3.1.1　バイナリとライブラリの両方を持つパッケージでのベストプラクティス

パッケージは **src/main.rs** バイナリクレートルートと **src/lib.rs** ライブラリクレートルートの両方を含むことができ、デフォルトでは両方のクレートがパッケージ名を持つことを説明しました。典型的には、ライブラリとバイナリのクレート両方を含むこのパターンのパッケージでは、バイナリクレートにはライブラリクレートのコードを呼び出して、実行可能形式を開始するために必要な最低限のコードを持たせます。こうすることで、ライブラリクレートのコードは共有できるので、他のプロジェクトはこのパッケージが提供するほとんどの機能を活用することができます。

モジュールツリーは **src/lib.rs** 内に定義してください。その場合、バイナリクレートからは、パッケージ名でから始まるパスを使って公開された要素を使用することができます。バイナリクレートは、そのライブラリクレートを利用する完全に外部のクレートとまったく同じように、ライブラリクレートの利用者になります: 公開 API しか使用することができません。これは良い API を設計する助けになります; あなたは作者であるだけでなく、利用者にもなるのです！第 12 章では、バイナリクレートとライブラリクレートの両方を含むコマンドラインプログラムを使って、この整理法の実践を示します。

7.3.2　相対パスを `super` で始める

現在のモジュールやクレートルートではなく、親モジュールから始まる相対パスなら、`super` を最初につけることで構成できます。ファイルシステムパスを `..` 構文で始めるのに似ています。`super` を使用することで、親モジュールにあることを知っている要素を参照することができます。これにより、そのモジュールが親と密接に関連しているが、いつか親がモジュールツリー内のどこかに移動されるかもしれないという場合に、モジュールツリーを再編成するのがより簡単になるかもしれません。

シェフが間違った注文を修正し、自分でお客さんに持っていくという状況をモデル化している、リスト 7-8 を考えてみてください。`back_of_house` モジュールで定義されている `fix_incorrect_order` 関数は、親モジュールで定義されている `deliver_order` 関数を呼び出すために、`super` から始まる `deliver_order` 関数へのパスを使っています。

ファイル名: src/lib.rs

```
fn deliver_order() {}
```

```
mod back_of_house {
    fn fix_incorrect_order() {
        cook_order();
        super::deliver_order();
    }

    fn cook_order() {}
}
```

リスト 7-8: super で始まる相対パスを使って関数を呼び出す

fix_incorrect_order 関数は back_of_house モジュールの中にあるので、super を使って back_of_house の親モジュールにいけます。親モジュールは、今回の場合ルートである crate です。そこから、deliver_order を探し、見つけ出します。成功！ もしクレートのモジュールツリーを再編成することにした場合でも、back_of_house モジュールと deliver_order 関数は同じ関係性で有り続け、一緒に動くように思われます。そのため、super を使うことで、将来このコードが別のモジュールに移動するとしても、更新する場所が少なくて済むようにしました。

7.3.3　構造体と enum を公開する

構造体や enum も pub を使って公開するよう指定できますが、構造体と enum での pub の使用に関しては追加の細目がいくつかあります。構造体定義の前に pub を使うと、構造体は公開されますが、構造体のフィールドは非公開のままなのです。それぞれのフィールドを公開するか否かを個々に決められます。リスト 7-9 では、公開の toast フィールドと、非公開の seasonal_fruit フィールドをもつ公開の back_of_house::Breakfast 構造体を定義しました。これは、例えば、レストランで、お客さんが食事についてくるパンの種類は選べるけれど、食事についてくるフルーツは季節と在庫に合わせてシェフが決める、という状況をモデル化しています。提供できるフルーツはすぐに変わるので、お客さんはフルーツを選ぶどころかどんなフルーツが提供されるのか知ることもできません。

ファイル名: src/lib.rs

```
mod back_of_house {
    pub struct Breakfast {
        pub toast: String,
        seasonal_fruit: String,
    }

    impl Breakfast {
        pub fn summer(toast: &str) -> Breakfast {
            Breakfast {
                toast: String::from(toast),
                seasonal_fruit: String::from("peaches"),
            }
        }
```

```
    }
}

pub fn eat_at_restaurant() {
    // Order a breakfast in the summer with Rye toast
    // 夏 (summer) にライ麦 (Rye) パン付き朝食を注文
    let mut meal = back_of_house::Breakfast::summer("Rye");
    // Change our mind about what bread we'd like
    // やっぱり別のパンにする
    meal.toast = String::from("Wheat");
    println!("I'd like {} toast please", meal.toast);

    // The next line won't compile if we uncomment it; we're not allowed
    // to see or modify the seasonal fruit that comes with the meal
    // 下の行のコメントを外すとコンパイルできない。食事についてくる
    // 季節のフルーツを知ることも修正することも許されていないので
    // meal.seasonal_fruit = String::from("blueberries");
}
```

リスト 7-9: 公開のフィールドと非公開のフィールドとを持つ構造体

back_of_house::Breakfast の toast フィールドは公開されているので、eat_at_restaurant において toast をドット記法を使って読み書きできます。seasonal_fruit は非公開なので、eat_at_restaurant において seasonal_fruit は使えないということに注意してください。seasonal_fruit を修正している行のコメントを外して、どのようなエラーが得られるか試してみてください！

また、back_of_house::Breakfast は非公開のフィールドを持っているので、Breakfast のインスタンスを作成 (construct) する公開された関連関数が構造体によって提供されている必要があります（ここでは summer と名付けました）。もし Breakfast にそのような関数がなかったら、eat_at_restaurant において非公開である seasonal_fruit の値を設定できないので、Breakfast のインスタンスを作成できません。

一方で、enum を公開すると、そのヴァリアントはすべて公開されます。リスト 7-10 に示されているように、pub は enum キーワードの前にだけおけばよいのです。

ファイル名: src/lib.rs

```
mod back_of_house {
    pub enum Appetizer {
        Soup,
        Salad,
    }
}

pub fn eat_at_restaurant() {
    let order1 = back_of_house::Appetizer::Soup;
    let order2 = back_of_house::Appetizer::Salad;
}
```

リスト 7-10: enum を公開に指定することはそのヴァリアントをすべて公開にする

Appetizer という enum を公開したので、Soup と Salad というヴァリアントも eat_at_restaurant で使えます。

enum はヴァリアントが公開されてないとあまり便利ではないのですが、毎回 enum のすべてのヴァリアントに pub をつけるのは面倒なので、enum のヴァリアントは標準で公開されるようになっているのです。構造体はフィールドが公開されていなくても便利なことが多いので、構造体のフィールドは、pub がついてない限り標準で非公開という通常のルールに従うわけです。

まだ勉強していない、pub の関わるシチュエーションがもう一つあります。モジュールシステムの最後の機能、use キーワードです。use 自体の勉強をした後、pub と use を組み合わせる方法についてお見せします。

7.4　use キーワードでパスをスコープに持ち込む

関数を呼び出すためにパスを略さずに書かなくてはならないのは、繰り返しも多くて不便に感じられるでしょう。リスト 7-7 においては、絶対パスを使うか相対パスを使うかにかかわらず、add_to_waitlist 関数を呼ぼうと思うたびに front_of_house と hosting も指定しないといけませんでした。ありがたいことに、この手続きを簡単化する方法があります: 一度 use キーワードを使ってショートカットを作成すれば、そのスコープ内であればどこでも、より短い名前を使用できます。

リスト 7-11 では、crate::front_of_house::hosting モジュールを eat_at_restaurant 関数のスコープに持ち込むことで、eat_at_restaurant において、hosting::add_to_waitlist と指定するだけで add_to_waitlist 関数を呼び出せるようにしています。

ファイル名: src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
}
```

リスト 7-11: use でモジュールをスコープに持ち込む

use とパスをスコープに追加することは、ファイルシステムにおいてシンボリックリンクを張

ることに似ています。use crate::front_of_house::hosting をクレートルートに追加することで、hosting はこのスコープで有効な名前となり、まるで hosting はクレートルートで定義されていたかのようになります。スコープに use で持ち込まれたパスも、他のパスと同じようにプライバシーがチェックされます。

use は、use が出現した特定のスコープでのみ使えるショートカットを作成することに注意してください。リスト 7-12 は eat_at_restaurant 関数を新しい子モジュール customer の中に移動していますが、このモジュールは use 文とは異なるスコープなので、この関数本体はコンパイルできないでしょう:

ファイル名: src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting;

mod customer {
    pub fn eat_at_restaurant() {
        hosting::add_to_waitlist();
    }
}
```

リスト 7-12: use 文はそれが属するスコープ内にのみ適用される

コンパイルエラーは customer モジュール内ではショートカットが適用されないことを示しています:

```
$ cargo build
   Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0433]: failed to resolve: use of undeclared crate or module `hosting`
(エラー: 名前解決に失敗しました: 宣言されていないクレートまたはモジュール`
    hosting`の使用)
  --> src/lib.rs:11:9
   |
11 |         hosting::add_to_waitlist();
   |         ^^^^^^^ use of undeclared crate or module `hosting`
   |                 (宣言されていないクレートまたはモジュール`hosting`の使用)
   |
help: consider importing this module through its public re-export
(ヘルプ: 公開再エクスポートからこのモジュールをインポートすることを検討してくだ
    さい)
   |
10 +     use crate::hosting;
   |
```

```
warning: unused import: `crate::front_of_house::hosting`
(警告: 未使用のインポート: `crate::front_of_house::hosting`)
 --> src/lib.rs:7:5
  |
7 | use crate::front_of_house::hosting;
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: `#[warn(unused_imports)]` on by default

For more information about this error, try `rustc --explain E0433`.
warning: `restaurant` (lib) generated 1 warning
error: could not compile `restaurant` (lib) due to 1 previous error; 1 warning
    emitted
```

use がそのスコープ内で使用されていないという警告も出ていることに気づくでしょう！ この問題を修正するには、use も customer モジュールの中に移動するか、customer 子モジュールの中で super::hosting によって親モジュールにあるショートカットを参照してください。

7.4.1　慣例に従った use パスを作る

リスト 7-11 を見て、なぜ use crate::front_of_house::hosting と書いて eat_at_restaurant 内で hosting::add_to_waitlist と呼び出したのか不思議に思っているかもしれません。リスト 7-13 のように、use で add_to_waitlist までのパスをすべて指定しても同じ結果が得られるのに、と。

ファイル名: src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting::add_to_waitlist;

pub fn eat_at_restaurant() {
    add_to_waitlist();
}
```

リスト 7-13: add_to_waitlist 関数を use でスコープに持ち込む。このやりかたは慣例的ではない

リスト 7-11 も 7-13 もおなじ仕事をしてくれますが、関数をスコープに use で持ち込む場合、リスト 7-11 のほうが慣例的なやり方です。関数の親モジュールを use で持ち込むということは、関数を呼び出す際、毎回親モジュールを指定しなければならないということです。関数を呼び出すときに親モジュールを指定することで、フルパスを繰り返して書くことを抑えつつ、関数がローカルで定義されていないことを明らかにできます。リスト 7-13 のコードではどこで add_to_waitlist が定義されたのかが不明瞭です。

一方で、構造体や enum その他の要素を use で持ち込むときは、フルパスを書くのが慣例的です。リスト 7-14 は標準ライブラリの HashMap 構造体をバイナリクレートのスコープに持ち込む慣例的なやり方を示しています。

ファイル名: src/main.rs

```
use std::collections::HashMap;

fn main() {
    let mut map = HashMap::new();
    map.insert(1, 2);
}
```

リスト 7-14: HashMap を慣例的なやり方でスコープに持ち込む

こちらの慣例の背後には、はっきりとした理由はありません。自然に発生した慣習であり、みんな Rust のコードをこのやり方で読み書きするのに慣れてしまったというだけです。

同じ名前の 2 つの要素を use でスコープに持ち込むのは Rust では許されないので、そのときこの慣例は例外的に不可能です。リスト 7-15 は、同じ名前を持つけれど異なる親モジュールを持つ 2 つの Result 型をスコープに持ち込み、それらを参照するやり方を示しています。

ファイル名: src/lib.rs

```
use std::fmt;
use std::io;

fn function1() -> fmt::Result {
    // --snip--
    //  （略）
#     Ok(())
}

fn function2() -> io::Result<()> {
    // --snip--
    //  （略）
#     Ok(())
}
```

リスト 7-15: 同じ名前を持つ 2 つの型を同じスコープに持ち込むには親モジュールを使わないといけない。

このように、親モジュールを使うことで 2 つの Result 型を区別できます。もし use std::fmt::Result と use std::io::Result と書いていたとしたら、2 つの Result 型が同じスコープに存在することになり、私達が Result を使ったときにどちらのことを意味しているのか Rust はわからなくなってしまいます。

7.4.2　新しい名前を as キーワードで与える

同じ名前の 2 つの型を use を使って同じスコープに持ち込むという問題には、もう一つ解決策があります。パスの後に、as と型の新しいローカル名、即ち**エイリアス**を指定すればよいのです。リスト 7-16 は、リスト 7-15 のコードを、2 つの Result 型のうち一つを as を使ってリネームするという別のやり方で書いたものを表しています。

ファイル名: src/lib.rs

```
use std::fmt::Result;
use std::io::Result as IoResult;

fn function1() -> Result {
    // --snip--
#     Ok(())
}

fn function2() -> IoResult<()> {
    // --snip--
#     Ok(())
}
```

リスト 7-16: 型がスコープに持ち込まれた時、as キーワードを使ってその名前を変えている

2 つめの use 文では、std::io::Result に、IoResult という新たな名前を選んでやります。std::fmt の Result もスコープに持ち込んでいますが、この名前はこれとは衝突しません。リスト 7-15 もリスト 7-16 も慣例的とみなされているので、どちらを使っても構いませんよ！

7.4.3　pub use を使って名前を再公開する

use キーワードで名前をスコープに持ちこんだ時、新しいスコープで使用できるその名前は非公開です。私達のコードを呼び出すコードが、まるでその名前が私達のコードのスコープで定義されていたかのように参照できるようにするためには、pub と use を組み合わせればいいです。このテクニックは、要素を自分たちのスコープに持ち込むだけでなく、他の人がその要素をその人のスコープに持ち込むことも可能にすることから、**再公開 (re-exporting)** と呼ばれています。

リスト 7-17 はリスト 7-11 のコードのルートモジュールでの use を pub use に変更したものを示しています。

ファイル名: src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
```

```
    }
}

pub use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
}
```

リスト 7-17: pub use で、新たなスコープのコードがその名前を使えるようにする

この変更を行う前の状態では、外部のコードはパス restaurant::front_of_house::hosting::add_to_waitlist() を使用して add_to_wishlist 関数を呼ばなくてはなりませんでした。この pub use によってルートモジュールから hosting モジュールを再公開された今、外部のコードはパス restaurant::hosting::add_to_waitlist() を使用できるようになりました。

再公開は、あなたのコードの内部構造と、あなたのコードを呼び出すプログラマーたちのその領域に関しての見方が異なるときに有用です。例えば、レストランの比喩では、レストランを経営している人は「接客部門 (front of house)」と「後方部門 (back of house)」のことについて考えるでしょう。しかし、レストランを訪れるお客さんは、そのような観点からレストランの部門について考えることはありません。pub use を使うことで、ある構造でコードを書きつつも、別の構造で公開するということが可能になります。こうすることで、私達のライブラリを、ライブラリを開発するプログラマにとっても、ライブラリを呼び出すプログラマにとっても、よく整理されたものとすることができます。第 14 章の「pub use で便利な公開 API をエクスポートする」の節で、pub use の別の例と、それがクレートのドキュメンテーションにどう影響するかを見ることにしましょう。

7.4.4　外部のパッケージを使う

2 章で、乱数を得るために rand という外部パッケージを使って、数当てゲームをプログラムしました。rand を私達のプロジェクトで使うために、次の行を **Cargo.toml** に書き加えましたね：

ファイル名: Cargo.toml

```
rand = "0.8.5"
```

rand を依存 (dependency) として **Cargo.toml** に追加すると、rand パッケージとそのすべての依存を crates.io からダウンロードして、私達のプロジェクトで rand が使えるようにするよう Cargo に命令します。

そして、rand の定義を私達のパッケージのスコープに持ち込むために、クレートの名前である rand から始まる use の行を追加し、そこにスコープに持ち込みたい要素を並べました。2 章の乱数を生成するの節で、Rng トレイトをスコープに持ち込み rand::thread_rng 関数を呼び出したことを思い出してください。

```
# use std::io;
use rand::Rng;

fn main() {
#     println!("Guess the number!");
#
    let secret_number = rand::thread_rng().gen_range(1..=100);
#
#     println!("The secret number is: {secret_number}");    //秘密の数字は次の通
    り: {secret_number}
#
#     println!("Please input your guess.");
#
#     let mut guess = String::new();
#
#     io::stdin()
#         .read_line(&mut guess)
#         .expect("Failed to read line");
#
#     println!("You guessed: {guess}");
}
```

Rust コミュニティに所属する人々が crates.io でたくさんのパッケージを利用できるようにしてくれており、上と同じステップを踏めばそれらをあなたのパッケージに取り込むことができます：あなたのパッケージの **Cargo.toml** ファイルにそれらを書き並べ、use を使って要素をクレートからスコープへと持ち込めばよいのです。

標準 std ライブラリも、私達のパッケージの外部にあるクレートだということに注意してください。標準ライブラリは Rust 言語に同梱されているので、**Cargo.toml** を std を含むように変更する必要はありません。しかし、その要素をそこから私達のパッケージのスコープに持ち込むためには、use を使って参照する必要はあります。例えば、HashMap には次の行を使います。

```
use std::collections::HashMap;
```

これは標準ライブラリクレートの名前 std から始まる絶対パスです。

7.4.5　巨大な use のリストをネストしたパスを使って整理する

同じクレートか同じモジュールで定義された複数の要素を使おうとする時、それぞれの要素を一行一行並べると、縦に大量のスペースを取ってしまいます。例えば、リスト 2-4 の数当てゲームで使った次の 2 つの use 文が std からスコープへ要素を持ち込みました。

ファイル名: src/main.rs

```
# use rand::Rng;
// --snip--
```

```
// （略）
use std::cmp::Ordering;
use std::io;
// --snip--
// （略）
#
# fn main() {
#     println!("Guess the number!");
#
#     let secret_number = rand::thread_rng().gen_range(1..=100);
#
#     println!("The secret number is: {secret_number}");
#
#     println!("Please input your guess.");
#
#     let mut guess = String::new();
#
#     io::stdin()
#         .read_line(&mut guess)
#         .expect("Failed to read line");
#
#     println!("You guessed: {guess}");
#
#     match guess.cmp(&secret_number) {
#         Ordering::Less => println!("Too small!"),
#         Ordering::Greater => println!("Too big!"),
#         Ordering::Equal => println!("You win!"),
#     }
# }
```

代わりに、ネストしたパスを使うことで、同じ一連の要素を 1 行でスコープに持ち込めます。これをするには、リスト 7-18 に示されるように、パスの共通部分を書き、2 つのコロンを続け、そこで波括弧で互いに異なる部分のパスのリストを囲みます。

ファイル名: src/main.rs

```
# use rand::Rng;
// --snip--
// （略）
use std::{cmp::Ordering, io};
// --snip--
// （略）
#
# fn main() {
#     println!("Guess the number!");
#
#     let secret_number = rand::thread_rng().gen_range(1..=100);
#
#     println!("The secret number is: {secret_number}");
#
#     println!("Please input your guess.");
```

```
#
#     let mut guess = String::new();
#
#     io::stdin()
#         .read_line(&mut guess)
#         .expect("Failed to read line");
#
#     let guess: u32 = guess.trim().parse().expect("Please type a number!");
#
#     println!("You guessed: {guess}");
#
#     match guess.cmp(&secret_number) {
#         Ordering::Less => println!("Too small!"),
#         Ordering::Greater => println!("Too big!"),
#         Ordering::Equal => println!("You win!"),
#     }
# }
```

リスト 7-18: 同じプレフィックスをもつ複数の要素をスコープに持ち込むためにネストしたパスを指定する

大きなプログラムにおいては、同じクレートやモジュールからのたくさんの要素をネストしたパスで持ち込むようにすれば、独立した use 文の数を大きく減らすことができます！

ネストしたパスはパスのどの階層においても使うことができます。これはサブパスを共有する 2 つの use 文を合体させるときに有用です。例えば、リスト 7-19 は 2 つの use 文を示しています：1 つは std::io をスコープに持ち込み、もう一つは std::io::Write をスコープに持ち込んでいます。

ファイル名: src/lib.rs

```
use std::io;
use std::io::Write;
```

リスト 7-19: 片方がもう片方のサブパスである 2 つの use 文

これらの 2 つのパスの共通部分は std::io であり、そしてこれは最初のパスにほかなりません。これらの 2 つのパスを 1 つの use 文へと合体させるには、リスト 7-20 に示されるように、ネストしたパスに self を使いましょう。

ファイル名: src/lib.rs

```
use std::io::{self, Write};
```

リスト 7-20: リスト 7-19 のパスを一つの use 文に合体させる

この行は std::io と std::io::Write をスコープに持ち込みます。

7.4.6　glob 演算子

パスにおいて定義されているすべての公開要素をスコープに持ち込みたいときは、glob 演算子 `*` をそのパスの後ろに続けて書きましょう：

```
use std::collections::*;
```

この use 文は std::collections のすべての公開要素を現在のスコープに持ち込みます。glob 演算子を使う際にはご注意を！ glob をすると、どの名前がスコープ内にあり、プログラムで使われている名前がどこで定義されたのか分かりづらくなります。

glob 演算子はしばしば、テストの際、テストされるあらゆるものを tests モジュールに持ち込むために使われます。これについては 11 章テストの書き方の節で話します。glob 演算子はプレリュードパターンの一部としても使われることがあります：そのようなパターンについて、より詳しくは標準ライブラリのドキュメントをご覧ください。

7.5　モジュールを複数のファイルに分割する

この章のすべての例において、今までのところ、複数のモジュールを一つのファイルに定義していました。モジュールが大きくなる時、コードを読み進めやすくするため、それらの定義を別のファイルへ移動させたくなるかもしれません。

例として、複数のレストランモジュールを持つリスト 7-17 のコードからはじめましょう。すべてのモジュールをクレートルートファイルで定義するのをやめて、複数のファイルにモジュールを抽出することにします。今回、クレートルートファイルは src/lib.rs ですが、この手続きはクレートルートファイルが src/main.rs であるバイナリクレートでもうまく行きます。

まず、front_of_house モジュールをそれ専用のファイルに抽出しましょう。front_of_house モジュールの波かっこの中のコードを削除し、mod front_of_house; 宣言だけを残して、**src/lib.rs** がリスト 7-21 に示すコードを含むようにしてください。リスト 7-22 で **src/front_of_house.rs** ファイルを作成するまで、このコードはコンパイルできないことに注意してください。

ファイル名: src/lib.rs

```
mod front_of_house;

pub use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
}
```

リスト 7-21:　front_of_house モジュールを宣言する。その中身は src/front_of_house.rs 内に

ある

次に、リスト 7-22 に示すように、**src/front_of_house.rs** という名前の新しいファイルに、波かっこの中にあったコードを配置してください。コンパイラは、クレートルートで front_of_house という名前のモジュール宣言を見つけたときは、このファイルを探せばいいということを知っています。

ファイル名: src/front_of_house.rs

```
pub mod hosting {
    pub fn add_to_waitlist() {}
}
```

リスト 7-22: **src/front_of_house.rs** における、front_of_house モジュール内部の定義

mod 宣言を使用したファイルのロードは、モジュールツリー内で**一回**のみ行う必要があることに注意してください。一度コンパイラが、そのファイルがプロジェクトの一部であることを認識したら（そして、mod 文を書いた場所に応じてモジュールツリー内のどこにコードがあることになるかを認識したら）、「モジュールツリーの要素を示すためのパス」節で扱ったように、プロジェクト内の他のファイルは、モジュールが宣言された場所へのパスを使用してロードされたファイルのコードを参照するべきです。別の言い方をすれば、mod は他のプログラミング言語で見られるような “include” 操作では**ありません**。

つづけて hosting モジュールをそれ専用のファイルに抽出します。hosting はルートモジュールの子ではなく front_of_house の子モジュールなので、このプロセスは少し異なります。hosting のためのファイルを、モジュールツリー上での祖先に対応して名付けられた新しいディレクトリ（今回の場合は **src/front_of_house/**）内に配置しましょう。

hosting の移動を開始するために、src/front_of_house.rs が hosting モジュールの宣言のみを含むように変更します：

ファイル名: src/front_of_house.rs

```
pub mod hosting;
```

さらに **src/front_of_house** ディレクトリと **hosting.rs** ファイルを作って、hosting モジュール内でなされていた定義を持つようにします。

ファイル名: src/front_of_house/hosting.rs

```
pub fn add_to_waitlist() {}
```

もし **src** ディレクトリ内に **hosting.rs** を置いた場合は、コンパイラは **hosting.rs** のコードは

front_of_house モジュールの子としてではなく、クレートルート内で宣言された hosting モジュールにあると期待するでしょう。コンパイラがどのモジュールのコードのためにどのファイルをチェックするかの規則は、ディレクトリとファイルがモジュールツリーと密接に一致することを意味します。

7.5.1　別のファイルパス

ここまで Rust コンパイラが使用するもっとも慣例的なファイルパスについて扱ってきましたが、Rust は古いスタイルのファイルパスもサポートしています。クレートルート内で宣言された front_of_house という名前のモジュールに対して、コンパイラは以下の場所からコードを探します:

- **src/front_of_house.rs**（ここまで扱ってきたもの）
- **src/front_of_house/mod.rs**（古いスタイルの、今もサポートされているパス）

front_of_house のサブモジュールである、hosting という名前のモジュールに対しては、コンパイラは以下の場所からコードを探します:

- **src/front_of_house/hosting.rs**（ここまで扱ってきたもの）
- **src/front_of_house/hosting/mod.rs**（古いスタイルの、今もサポートされているパス）

同一のモジュールに対して両方のスタイルを使用するとコンパイルエラーになります。異なるモジュールに対して両方のスタイルを混ぜて使用することは許可されていますが、プロジェクトを見て回る人たちにとって混乱を招くかもしれません。

mod.rs という名前のファイルを使用するスタイルの主な欠点は、プロジェクト内に **mod.rs** という名前のファイルが大量にできることになり、エディタで同時に開いたときに混乱を招きやすいことです。

各モジュールのコードを独立したファイルに移動しましたが、モジュールツリーは同じままです。定義が別のファイルにあるにもかかわらず、eat_at_restaurant 内での関数呼び出しもなんの変更もなくうまく行きます。このテクニックのおかげで、モジュールが大きくなってきた段階で新しいファイルへ動かす、ということができます。

src/lib.rs における pub use crate::front_of_house::hosting という文も変わっていないし、use はどのファイルがクレートの一部としてコンパイルされるかになんの影響も与えないということに注意してください。mod キーワードがモジュールを宣言したなら、Rust はそのモジュールに挿入するためのコードを求めて、モジュールと同じ名前のファイルの中を探すというわけです。

7.6　まとめ

Rust では、パッケージを複数のクレートに、そしてクレートを複数のモジュールに分割して、あるモジュールで定義された要素を他のモジュールから参照することができます。これは絶対パスか相対パスを指定することで行なえます。これらのパスは use 文でスコープに持ち込むことができ、こう

すると、そのスコープで要素を複数回使う時に、より短いパスで済むようになります。モジュールのコードは標準では非公開ですが、pub キーワードを追加することで定義を公開することができます。

次の章では、きちんと整理されたあなたのコードで使うことができる、標準ライブラリのいくつかのコレクションデータ構造を見ていきます。

第 8 章

一般的なコレクション

Rust の標準ライブラリは、**コレクション**と呼ばれる多くの非常に有益なデータ構造を含んでいます。他の多くのデータ型は、ある一つの値を表しますが、コレクションは複数の値を含むことができます。組み込みの配列とタプル型とは異なり、これらのコレクションが指すデータはヒープに確保され、データ量はコンパイル時にわかる必要はなく、プログラムの実行にあわせて、伸縮可能であることになります。各種のコレクションには異なる能力とコストが存在し、自分の現在の状況に最適なものを選び取るスキルは、時間とともに育っていきます。この章では、Rust のプログラムにおいて、非常に頻繁に使用される 3 つのコレクションについて議論しましょう。

- **ベクタ型**は、可変長の値を並べて保持できる。
- **文字列**は、文字のコレクションである。以前、`String` 型について触れたが、この章ではより掘り下げていく。
- **ハッシュマップ**は、値を特定のキーと紐付けさせてくれる。より一般的なデータ構造である、**マップ**の特定の実装である。

標準ライブラリで提供されている他の種のコレクションについて学ぶには、ドキュメントを参照されたし。

ベクタ型、文字列、ハッシュマップの生成と更新方法や、各々が特別な点について議論していきましょう。

8.1 ベクタで値のリストを保持する

最初に見るコレクション型は `Vec<T>` です。**ベクタ**とも呼ばれます。ベクタを使用することで、複数の値をメモリ上で互いに隣り合うように単一のデータ構造に保持することができます。ベクタは同じ型の値のみ保持することができます。テキストファイルの各行や、ショッピングカートのアイテムの価格などのような、要素のリストがある場合にベクタは有用です。

8.1.1　新しいベクタを生成する

空のベクタを新たに作成するには、リスト 8-1 に示すように `Vec::new` 関数を呼びます。

```
# fn main() {
    let v: Vec<i32> = Vec::new();
# }
```

リスト 8-1：`i32` 型の値を保持する新しい空のベクタを作成する

ここで型注釈を付けていることに注目してください。このベクタには何も値を挿入していないので、コンパイラには私たちがどんなデータを保持させるつもりか分かりません。これは重要な点です。ベクタはジェネリクスを使用して実装されています; ジェネリクスを使用する自身の型をどう使用するかついては第 10 章で解説します。今のところは、標準ライブラリで提供される `Vec<T>` 型は、どんな型でも保持できると理解しておいてください。特定の型を保持するベクタを作成するときには、その型を山かっこ内に指定すればよいです。リスト 8-1 では、コンパイラに `v` の `Vec<T>` は `i32` 型の要素を保持すると指示していたのです。

`Vec<T>` を作成する場合は初期値を持たせることのほうが多いでしょう。その場合コンパイラは保持したい値の型を推論するでしょうから、このように型注釈を付ける必要はほとんどありません。Rust には `vec!` という便利なマクロが用意されていて、これは与えた値を保持する新しいベクタを作成します。リスト 8-2 では、`1`、`2`、`3` という値を持つ新しい `Vec<i32>` を作成しています。整数の型が `i32` になっているのは、3 章の「データ型」節で学んだように、これがデフォルトの整数型だからです。

```
# fn main() {
    let v = vec![1, 2, 3];
# }
```

リスト 8-2: 値を含む新しいベクタを作成する

初期値の `i32` 値を与えたので、コンパイラは `v` の型が `Vec<i32>` であると推論でき、型注釈は不要になりました。次はベクタを変更する方法を見ましょう。

8.1.2　ベクタを更新する

ベクタを作成し、それから要素を追加するには、リスト 8-3 に示すように `push` メソッドを使います。

```
# fn main() {
    let mut v = Vec::new();

    v.push(5);
    v.push(6);
```

```
    v.push(7);
    v.push(8);
# }
```

リスト 8-3：push メソッドを使用してベクタに値を追加する

第 3 章で説明したとおり、どんな変数でも、その値を変更したかったら mut キーワードで可変にする必要があります。中に配置する数値は全て i32 型であり、Rust はこのことをデータから推論するので、Vec<i32> という注釈は不要です。

8.1.3　ベクタの要素を読む

ベクタに保持された値を参照する方法は 2 つあります: 添え字を使用する方法と、get メソッドを使用する方法です。これから示す例では、理解を助けるために、それらの関数からの戻り値型を注釈しています。

リスト 8-4 はベクタの値にアクセスする両方の方法として、添え字記法と get メソッドが示されています。

```
# fn main() {
    let v = vec![1, 2, 3, 4, 5];

    let third: &i32 = &v[2];
    //       "3つ目の要素は{third}です"
    println!("The third element is {third}");

    let third: Option<&i32> = v.get(2);
    match third {
        //                      "3つ目の要素は{third}です"
        Some(third) => println!("The third element is {third}"),
        //               "3つ目の要素はありません。"
        None => println!("There is no third element."),
    }
# }
```

リスト 8-4：添え字記法か get メソッドを使用してベクタの要素にアクセスする

ここでいくつか注意があります。ベクタは 0 から始まる番号で添え字アクセスされるので、3 番目の要素を得るのに 2 という添え字の値を使用しています。& と [] を使用することで、その添え字の値にある要素への参照が得られます。get メソッドに引数として添え字を渡して使用すると、Option<&T> が得られ、これは match で使用することができます。

Rust が要素を参照するためにこれらの 2 通りの方法を提供している理由は、存在する要素の範囲外の添え字の値を使おうとしたときのプログラムの振る舞いを選択できるようにするためです。例として、5 要素のベクタがあるとして、それぞれの手法で添え字 100 の要素にアクセスを試みた場合、何が起こるのか確認しましょう。リスト 8-5 に示します。

```
# fn main() {
    let v = vec![1, 2, 3, 4, 5];

    let does_not_exist = &v[100];
    let does_not_exist = v.get(100);
# }
```

リスト 8-5：5 つの要素を含むベクタの添え字 100 の要素にアクセスしようとする

このコードを走らせると、最初の `[]` メソッドはプログラムをパニックさせます。なぜなら存在しない要素を参照しているからです。このメソッドは、ベクタの終端を超えて要素にアクセスしようとしたときにプログラムをクラッシュさせたい場合に最適です。

`get` メソッドにベクタ外の添え字を渡すと、パニックすることなく `None` を返します。普通の状況でもベクタの範囲外にアクセスする可能性があるなら、このメソッドを使用することになるでしょう。その場合、第 6 章で説明したように、コードは `Some(&element)` か `None` を扱うロジックを持つことになります。例えば、誰かが入力した数値が添え字になるかもしれません。もし誤って大きすぎる値を入力し、プログラムが `None` 値を得たなら、いまベクタに何要素あるかをユーザに教え、正しい値を再入力してもらうこともできます。その方が、ただのタイプミスでプログラムをクラッシュさせるより、ユーザに優しいといえそうです。

プログラムに有効な参照がある場合、借用チェッカー (borrow checker) は、（第 4 章で解説しましたが）所有権と借用規則を強制し、ベクタの中身へのこの参照や他のいかなる参照も有効であり続けることを保証してくれます。同一スコープ上では、可変と不変な参照を同時には存在させられないというルールを思い出してください。このルールはリスト 8-6 でも適用されています。リスト 8-6 ではベクタの最初の要素への不変参照を保持しつつ、終端に要素を追加しようとしています。このプログラムは、関数内のここ以降で、この要素（訳注：`first` のこと）を参照しようとすると失敗します。

```
# fn main() {
    let mut v = vec![1, 2, 3, 4, 5];

    let first = &v[0];

    v.push(6);

    //       "最初の要素は: {first}"
    println!("The first element is: {first}");
# }
```

リスト 8-6：要素への参照を保持しつつ、ベクタに要素を追加しようとする

このコードをコンパイルすると、こんなエラーになります。

```
$ cargo run
   Compiling collections v0.1.0 (file:///projects/collections)
```

```
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as
    immutable
(エラー: 不変としても借用されているので、`v`を可変で借用できません)
 --> src/main.rs:6:5
  |
4 |     let first = &v[0];
  |                  - immutable borrow occurs here
  |                   (不変借用はここで発生しています)
5 |
6 |     v.push(6);
  |     ^^^^^^^^^ mutable borrow occurs here
  |               (可変借用はここで発生しています)
7 |
8 |     println!("The first element is: {first}");
  |                                     ------- immutable borrow later used here
  |                                            (その後、不変借用はここで使われて
      います)

For more information about this error, try `rustc --explain E0502`.
error: could not compile `collections` (bin "collections") due to 1 previous
    error
```

リスト 8-6 のコードは、一見動きそうに見えるかもしれません。なぜ最初の要素への参照が、ベクタの終端への変更を気にかける必要があるのでしょうか？ このエラーはベクタが動作するしくみによるものです: ベクタはメモリ上に値同士を隣り合うように配置するため、新たな要素をベクタの終端に追加するとき、いまベクタが置かれている場所に全要素を隣り合わせに配置するだけのスペースがないなら、新しいメモリを割り当て、古い要素を新しいスペースにコピーする必要があります。その場合、最初の要素を指す参照は、解放されたメモリを指すことになるでしょう。借用規則がそのような状況に陥らないよう防いでくれるのです。

> 注釈: `Vec<T>` の実装に関する詳細については、“The Rustonomicon” を参照してください（訳注：日本語版はこちらです）。

8.1.4　ベクタ内の値を順に処理する

ベクタの要素に順番にアクセスするには、添え字で 1 要素ごとにアクセスするのではなく、全要素を走査することになるでしょう。リスト 8-7 で `for` ループを使い、`i32` のベクタの各要素に対する不変な参照を得て、それらを表示する方法を示します。

```
# fn main() {
    let v = vec![100, 32, 57];
    for i in &v {
        println!("{i}");
    }
# }
```

リスト 8-7：for ループで要素を走査し、ベクタの各要素を表示する

また、全要素に変更を加えるために、可変なベクタの各要素への可変な参照を走査することもできます。リスト 8-9 の for ループでは各要素に 50 を足しています。

```
# fn main() {
    let mut v = vec![100, 32, 57];
    for i in &mut v {
        *i += 50;
    }
# }
```

リスト 8-8：ベクタの要素への可変な参照を走査する

可変参照が参照している値を変更するには、+= 演算子を使用する前に、* 参照外し演算子を使用して i の値にたどり着かないといけません。参照外し演算子については、第 15 章の「参照外し演算子で値までポインタを追いかける」節でより詳しく扱います。

借用チェッカーの規則のおかげで、ベクタを走査するのは不変であっても可変であっても安全です。もしリスト 8-7 やリスト 8-8 の for ループ本体の中で要素を挿入または削除しようとすると、リスト 8-6 のコードで発生したのと同じようなコンパイルエラーになるでしょう。for ループがベクタへの参照をつかんでいるために、それと同時にベクタ全体の変更が起こらないようになっています。

8.1.5　Enum を使って複数の型を保持する

ベクタは同じ型の値しか保持できません。これは不便なこともあります。異なる型の要素を保持する必要のあるユースケースは必ず存在します。幸運なことに、enum の列挙子は同じ enum の型の中に定義されるので、異なる型の要素を表現する単一の型が必要になったら、enum を定義して使用すればよいのです！

例えば、スプレッドシートのある行から値を得ることを考えます。ここで、その行の中の列には、整数を含むもの、浮動小数点数を含むもの、文字列を含むものがあるとします。列挙子ごとに異なる値の型を保持する enum が定義できます。そして、この enum の列挙子は全て同じ型、つまり enum の型、と考えられるわけです。ですから、その enum を保持するためのベクタを作成でき、結果的に異なる型を保持できるようになるわけです。リスト 8-9 でこれを実演しています。

```
# fn main() {
    enum SpreadsheetCell {
        Int(i32),
        Float(f64),
        Text(String),
    }

    let row = vec![
        SpreadsheetCell::Int(3),
```

```
        SpreadsheetCell::Text(String::from("blue")),
        SpreadsheetCell::Float(10.12),
    ];
# }
```

リスト 8-9：enum を定義して、一つのベクタに異なる型の値を保持する

個々の要素を格納するのにヒープ上で必要となるメモリの量を正確に把握するために、Rust コンパイラはコンパイル時にベクタに入る型を知る必要があります。また、このベクタではどんな型が許容されるのか明示しなくてはなりません。もし Rust が、ベクタにどんな型でも保持できることを許していたら、ベクタの要素に対して行われる処理に対して、いくつかの型がエラーを引き起こすかもしれません。enum に加えて match 式を使うことで、第 6 章で説明したとおり、あらゆるケースが処理できることを、Rust がコンパイル時に保証することになります。

プログラムが実行時に取得し、ベクタに格納し得る全ての型を網羅できない場合には、この enum を使ったテクニックはうまくいかないでしょう。代わりにトレイトオブジェクトを使用できます。こちらは第 17 章で取り上げます。

これまでにベクタの代表的な使い方をいくつか紹介しました。標準ライブラリで Vec<T> に定義されている多くの有益なメソッドについて、API ドキュメントを必ず確認するようにしてください。例えば、push に加えて、pop というメソッドがあり、これは最後の要素を削除して返します。

8.1.6　ベクタをドロップすれば、要素もドロップする

他のあらゆる struct（構造体）と同様に、ベクタもスコープを抜ければ解放されます。その様子をリスト 8-10 に示します。

```
# fn main() {
    {
        let v = vec![1, 2, 3, 4];

        // vで作業をする
    } // <- vはここでスコープを抜け、解放される
# }
```

リスト 8-10：ベクタとその要素がドロップされる箇所を示す

ベクタがドロップされると、その中身もドロップされます。つまり、保持されていた整数値が片付けられるということです。借用チェッカーは、ベクタの内容へのいかなる参照も、ベクタ自体が有効な間のみ使用されることを保証します。

それでは次のコレクション型である String に移りましょう！

8.2　文字列で UTF-8 でエンコードされたテキストを保持する

第 4 章で文字列について語りましたが、今度はより掘り下げていきましょう。新参者の Rustacean は、3 つの理由の組み合わせにより、文字列でよく行き詰まります: Rust のありうるエラーを晒す性質、多くのプログラマが思っている以上に文字列が複雑なデータ構造であること、そして UTF-8 です。これらの要因が、他のプログラミング言語から移ってきた場合、一見困難に見えるように絡み合うわけです。

文字列は、バイトのコレクションに、そのバイトをテキストとして解釈するときに便利な機能を提供するメソッドを足したものとして実装されているため、コレクションの文脈で文字列を議論していきます。この節では、生成、更新、読み込みのような全コレクションが持つ `String` の処理について語ります。また、`String` が他のコレクションと異なる点についても議論します。具体的には、人間とコンピュータが `String` データを解釈する方法の差異により、`String` に添え字アクセスする方法がどう複雑なのかということです。

8.2.1　文字列とは？

まずは、**文字列**という用語の意味を定義しましょう。Rust には、言語の核として 1 種類しか文字列型が存在しません。文字列スライスの `str` で、通常借用された形態 `&str` で見かけます。第 4 章で、**文字列スライス**について語りました。これは、別の場所に格納された UTF-8 エンコードされた文字列データへの参照です。例えば、文字列リテラルは、プログラムのバイナリに格納されるので、文字列スライスになります。

`String` 型は、言語の核として組み込まれるのではなく、Rust の標準ライブラリによって提供されますが、伸長可能、可変、所有権のある UTF-8 エンコードされた文字列型です。Rustacean が Rust において「文字列」と言うとき、`String` 型または文字列スライスの `&str` 型のいずれかを指しているかもしれません。この節は、大方、`String` についてですが、どちらの型も Rust の標準ライブラリで重宝されており、どちらも UTF-8 エンコードされています。

8.2.2　新規文字列を生成する

`String` は実際のところ、いくつか追加の保障と制限と機能を持つバイトのベクタのラッパーとして実装されているので、`Vec<T>` で使用可能な操作の多くが `String` でも使用できます。`Vec<T>` と `String` で同様に機能する関数の例としては、それぞれのインスタンスを作成する `new` 関数があります。リスト 8-11 に示したようにですね。

```
# fn main() {
    let mut s = String::new();
# }
```

リスト 8-11: 新しい空の String を生成する

この行は、新しい空の s という文字列を生成しています。それからここにデータを読み込むことができるわけです。だいたい、文字列の初期値を決めるデータがあるでしょう。そのために、to_string メソッドを使用します。このメソッドは、文字列リテラルのように、Display トレイトを実装する型ならなんでも使用できます。リスト 8-12 に 2 例、示しています。

```
# fn main() {
    let data = "initial contents";

    let s = data.to_string();

    // the method also works on a literal directly:
    let s = "initial contents".to_string();
# }
```

リスト 8-12: to_string メソッドを使用して文字列リテラルから String を生成する

このコードは、initial contents (初期値) を含む文字列を生成します。

さらに、String::from 関数を使っても、文字列リテラルから String を生成することができます。リスト 8-13 のコードは、to_string を使用するリスト 8-12 のコードと等価です。

```
# fn main() {
    let s = String::from("initial contents");
# }
```

リスト 8-13: String::from 関数を使って文字列リテラルから String を作る

文字列は、非常に多くのものに使用されるので、多くの異なる一般的な API を使用でき、たくさんの選択肢があるわけです。冗長に思われるものもありますが、適材適所です！ 今回の場合、String::from と to_string は全く同じことをします。従って、どちらを選ぶかは、スタイルと可読性次第です。

文字列は UTF-8 エンコードされていることを覚えていますか？ 要するに文字列には、適切にエンコードされていればどんなものでも含めます。リスト 8-14 に示したように。

```
# fn main() {
    let hello = String::from("السلام عليكم");
    let hello = String::from("Dobrý den");
    let hello = String::from("Hello");
    let hello = String::from("שָׁלוֹם");
    let hello = String::from("नमस्ते");
    let hello = String::from("こんにちは");
    let hello = String::from("안녕하세요");
    let hello = String::from("你好");
    let hello = String::from("Olá");
    let hello = String::from("Здравствуйте");
```

```
    let hello = String::from("Hola");
# }
```

リスト 8-14: いろんな言語の挨拶を文字列に保持する

これらは全て、有効な String の値です。

8.2.3　文字列を更新する

String は、サイズを伸ばすことができ、Vec<T> の中身のように、追加のデータをプッシュすれば、中身も変化します。付け加えると、String 値を連結する + 演算子や、format! マクロを便利に使用することができます。

8.2.3.1　push_str と push で文字列に追加する

push_str メソッドで文字列スライスを追記することで、String を伸ばすことができます。リスト 8-15 の通りです。

```
# fn main() {
    let mut s = String::from("foo");
    s.push_str("bar");
# }
```

リスト 8-15: push_str メソッドで String に文字列スライスを追記する

この 2 行の後、s は foobar を含むことになります。push_str メソッドは、必ずしも引数の所有権を得なくていいので、文字列スライスを取ります。例えば、リスト 8-16 のコードで、中身を s1 に追加した後も s2 を使いたいとします。

```
# fn main() {
    let mut s1 = String::from("foo");
    let s2 = "bar";
    s1.push_str(s2);
    println!("s2 is {s2}");
# }
```

リスト 8-16: 中身を String に追加した後に、文字列スライスを使用する

もし、push_str メソッドが s2 の所有権を奪っていたら、最後の行でその値を出力することは不可能でしょう。ところが、このコードは予想通りに動きます！

push メソッドは、1 文字を引数として取り、String に追加します。リスト 8-15 は、push メソッドで “l” を String に追加しています。

```
# fn main() {
```

```
    let mut s = String::from("lo");
    s.push('l');
# }
```

リスト 8-17: push で String 値に 1 文字を追加する

この結果、s は lol を含むことになるでしょう。

> 編者注: lol は laughing out loud (大笑いする) の頭文字からできたスラングです。日本語の www みたいなものですね。

8.2.3.2 + 演算子、または format! マクロで連結

2 つのすでにある文字列を組み合わせたくなることがよくあります。これを行うための方法の一つは、リスト 8-18 に示したように、+ 演算子を使用することです。

```
# fn main() {
    let s1 = String::from("Hello, ");
    let s2 = String::from("world!");
    let s3 = s1 + &s2; // s1はムーブされ、もう使用できないことに注意
# }
```

リスト 8-18: + 演算子を使用して二つの String 値を新しい String 値にする

s3 という文字列は、Hello, world! を含むことになるでしょう。追記の後、s1 がもう有効でなくなった理由と、s2 への参照を使用した理由は、+ 演算子を使用した時に呼ばれるメソッドのシグニチャと関係があります。+ 演算子は、add メソッドを使用し、そのシグニチャは以下のような感じです:

```
fn add(self, s: &str) -> String {
```

標準ライブラリでは、add はジェネリクスと関連型を使用して定義されているのがわかるでしょう。ここでは具体的な型で置き換えていますが、これはこのメソッドを String 値とともに呼び出した時に起こることです。ジェネリクスについては、第 10 章で議論します。このシグニチャが、+ 演算子の巧妙な部分を理解するのに必要な手がかりになるのです。

まず、s2 には & がついてます。つまり、最初の文字列に 2 番目の文字列の参照を追加するということです。これは add 関数の s 引数のためのものです: String には &str を追加することしかできません。要するに 2 つの String 値を追加することはできないのです。でも待ってください。add の第 2 引数で指定されているように、&s2 の型は、&str ではなく、&String ではないですか。では、なぜ、リスト 8-18 は、コンパイルできるのでしょうか？

add 呼び出しで &s2 を使える理由は、コンパイラが &String 引数を &str に**型強制**してくれるためです。add メソッド呼び出しの際、コンパイラは、**参照外し型強制**というものを使用し、ここでは、&s2 を &s2[..] に変えるものと考えることができます。参照外し型強制について詳しくは、第 15 章で議

論します。add が s 引数の所有権を奪わないので、この処理後も s2 が有効な String になるわけです。

2 番目に、シグニチャから add は self の所有権をもらうことがわかります。self には & がついて**いない**からです。これはつまり、リスト 8-18 において s1 は add 呼び出しにムーブされ、その後は有効ではなくなるということです。故に、s3 = s1 + &s2; は両文字列をコピーして新しいものを作るように見えますが、この文は実際には s1 の所有権を奪い、s2 の中身のコピーを追記し、結果の所有権を返すのです。言い換えると、たくさんのコピーをしているように見えますが、違います; 実装は、コピーよりも効率的です。

複数の文字列を連結する必要が出ると、+ 演算子の振る舞いは扱いにくくなります:

```
# fn main() {
    let s1 = String::from("tic");
    let s2 = String::from("tac");
    let s3 = String::from("toe");

    let s = s1 + "-" + &s2 + "-" + &s3;
# }
```

ここで、s は tic-tac-toe になるでしょう。+ と " 文字のせいで何が起きているのかわかりにくいです。もっと複雑な文字列の連結には、代わりに format! マクロを使用することができます:

```
# fn main() {
    let s1 = String::from("tic");
    let s2 = String::from("tac");
    let s3 = String::from("toe");

    let s = format!("{s1}-{s2}-{s3}");
# }
```

このコードでも、s は tic-tac-toe になります。format! マクロは、println! と似た動作をしますが、出力をスクリーンに行う代わりに、中身を String で返すのです。format! を使用したコードの方がはるかに読みやすく、format! マクロによって生成されたコードはは参照を使用するので、この呼び出しは引数の所有権を奪いません。

8.2.4　文字列に添え字アクセスする

他の多くのプログラミング言語では、文字列中の文字に、添え字で参照してアクセスすることは、有効なコードであり、一般的な処理です。しかしながら、Rust において、添え字記法で String の一部にアクセスしようとすると、エラーが発生するでしょう。リスト 8-19 の非合法なコードを考えてください。

```
# fn main() {
    let s1 = String::from("hello");
    let h = s1[0];
# }
```

リスト 8-19: 文字列に対して添え字記法を試みる

このコードは、以下のようなエラーに落ち着きます:

```
$ cargo run
   Compiling collections v0.1.0 (file:///projects/collections)
error[E0277]: the type `String` cannot be indexed by `{integer}`
(エラー: 型`String`は`{Integer}`で添え字アクセスできません)
 --> src/main.rs:3:16
  |
3 |     let h = s1[0];
  |                ^ `String` cannot be indexed by `{integer}`
  |                 (`String`は`{Integer}`で添え字アクセスできません)
  |
  = help: the trait `Index<{integer}>` is not implemented for `String`
    (ヘルプ: `Index<{Integer}>`というトレイトが`String`に対して実装されていません)
  = help: the following other types implement trait `Index<Idx>`:
    (ヘルプ: 以下の型であればトレイト`Index<Idx>`を実装しています:)
            <String as Index<RangeFull>>
            <String as Index<std::ops::Range<usize>>>
            <String as Index<RangeFrom<usize>>>
            <String as Index<RangeTo<usize>>>
            <String as Index<RangeInclusive<usize>>>
            <String as Index<RangeToInclusive<usize>>>

For more information about this error, try `rustc --explain E0277`.
error: could not compile `collections` (bin "collections") due to 1 previous
    error
```

エラーと注釈が全てを物語っています: Rust の文字列は、添え字アクセスをサポートしていないのです。でも、なぜでしょうか？ その疑問に答えるには、Rust がメモリにどのように文字列を保持しているかについて議論する必要があります。

8.2.4.1 内部表現

`String` は `Vec<u8>` のラッパです。リスト 8-14 から適切に UTF-8 でエンコードされた文字列の例をご覧ください。まずは、これ:

```
# fn main() {
#     let hello = String::from("السلام عليكم");
#     let hello = String::from("Dobrý den");
#     let hello = String::from("Hello");
#     let hello = String::from("שָׁלוֹם");
#     let hello = String::from("नमस्ते");
#     let hello = String::from("こんにちは");
#     let hello = String::from("안녕하세요");
#     let hello = String::from("你好");
```

```
#     let hello = String::from("Olá");
#     let hello = String::from("Здравствуйте");
    let hello = String::from("Hola");
# }
```

この場合、len は 4 になり、これは、文字列”Hola” を保持するベクタの長さが 4 バイトであることを意味します。これらの各文字は、UTF-8 でエンコードすると、1 バイトになるのです。しかし、次の行にはびっくりするかもしれません。(この文字列は大文字のキリル文字ゼーで始まり、数字の 3 では始まっていないことに注意してください)

```
# fn main() {
#     let hello = String::from("السلام عليكم");
#     let hello = String::from("Dobrý den");
#     let hello = String::from("Hello");
#     let hello = String::from("שָׁלוֹם");
#     let hello = String::from("नमस्ते");
#     let hello = String::from("こんにちは");
#     let hello = String::from("안녕하세요");
#     let hello = String::from("你好");
#     let hello = String::from("Olá");
    let hello = String::from("Здравствуйте");
#     let hello = String::from("Hola");
# }
```

文字列の長さはと問われたら、あなたは 12 と答えるかもしれません。実際には、Rust の答えは、24 です: “Здравствуйте” を UTF-8 でエンコードすると、この長さになります。この文字列に含まれる各 Unicode スカラー値は、2 バイトの領域を取るからです。それ故に、文字列のバイトの添え字は、必ずしも有効な Unicode のスカラー値とは相互に関係しないのです。デモ用に、こんな非合法な Rust コードを考えてください:

```
let hello = "Здравствуйте";
let answer = &hello[0];
```

answer の値は、最初の文字の 3 にはならないことを知っているでしょう。UTF-8 エンコードされた時、3 の最初のバイトは 208 、2 番目は 151 になるので、answer は実際には 208 になりそうですが、208 は単独では有効な文字ではありません。この文字列の最初の文字を求めている場合、208 を返すことは、ユーザの望んでいるものではないでしょう; しかしながら、Rust には、バイト添え字 0 の位置には、そのデータしかないのです。文字列がラテン文字のみを含む場合でも、ユーザは一般的にバイト値が返ることを望みません: &"hello"[0] がバイト値を返す有効なコードだったら、h ではなく、104 を返すでしょう。

答えは、予期しない値を返し、すぐには判明しないバグを引き起こさないために、Rust はこのコードを全くコンパイルせず、開発過程の早い段階で誤解を防いでくれるのです。

8.2.4.2　バイトとスカラー値と書記素クラスタ！ なんてこった！

UTF-8 について別の要点は、実際 Rust の観点から文字列を見るには 3 つの関連した方法があるということです: バイトとして、スカラー値として、そして、書記素クラスタ (人間が**文字**と呼ぶものに一番近い) としてです。

ヒンディー語の単語、“नमस्ते” をデーヴァナーガリー (訳注：サンスクリット語とヒンディー語を書くときに使われる書記法) で表記したものを見たら、以下のような見た目の `u8` 値のベクタとして保持されます:

```
[224, 164, 168, 224, 164, 174, 224, 164, 184, 224, 165, 141, 224, 164, 164,
224, 165, 135]
```

18 バイトになり、このようにしてコンピュータは最終的にこのデータを保持しているわけです。これを Unicode スカラー値として見たら（Rust の `char` 型はこれなのですが）このバイトは以下のような見た目になります:

```
['न', 'म', 'स', '्', 'त', 'े']
```

ここでは、6 つ `char` 値がありますが、4 番目と 6 番目は文字ではありません: 単独では意味をなさないダイアクリティックです。最後に、書記素クラスタとして見たら、このヒンディー語の単語を作り上げる人間が 4 文字と呼ぶであろうものが得られます:

```
["न", "म", "स्", "ते"]
```

Rust には、データが表す自然言語に関わらず、各プログラムが必要な解釈方法を選択できるように、コンピュータが保持する生の文字列データを解釈する方法がいろいろ用意されています。

Rust で文字を得るのに `String` に添え字アクセスすることが許されない最後の理由は、添え字アクセスという処理が常に定数時間 (O(1)) になると期待されるからです。しかし、`String` でそのパフォーマンスを保証することはできません。というのも、合法な文字がいくつあるか決定するのに、最初から添え字まで中身を走査する必要があるからです。

8.2.5　文字列をスライスする

文字列に添え字アクセスするのは、しばしば悪い考えです。文字列添え字処理の戻り値の型が明瞭ではないからです: バイト値、文字、書記素クラスタ、あるいは文字列スライスにもなります。故に、文字列スライスを生成するのに、添え字を使う必要が本当に出た場合にコンパイラは、もっと特定するよう求めてきます。

`[]` で 1 つの数値により添え字アクセスするのではなく、範囲とともに `[]` を使って、特定のバイトを含む文字列スライスを作ることができます:

```
let hello = "Здравствуйте";
```

```
let s = &hello[0..4];
```

ここで、s は文字列の最初の 4 バイトを含む &str になります。先ほど、これらの文字は各々 2 バイトになると指摘しましたから、s は Зд になります。

&hello[0..1] のようにして、文字を構成するバイトの一部のみをスライスしようとすると、Rust はベクタの非合法な添え字にアクセスしたかのように実行時にパニックするでしょう:

```
$ cargo run
   Compiling collections v0.1.0 (file:///projects/collections)
    Finished dev [unoptimized + debuginfo] target(s) in 0.43s
     Running `target/debug/collections`
thread 'main' panicked at src/main.rs:4:19:
byte index 1 is not a char boundary; it is inside 'З' (bytes 0..2) of `
    Здравствуйте`
('main'スレッドはsrc/main:4:19でパニックしました:
バイト添え字1は文字の境界ではありません; `Здравствуйте`の'З'(バイト0..2)の中です
    )
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

範囲を使用して文字列スライスを作る際にはプログラムをクラッシュさせることがあるので、気をつけるべきです。

8.2.6　文字列を走査するメソッド群

文字列の部分に対して操作を行うための最良の方法は、文字に対して操作したいのかバイトに対して操作したいのかを明示することです。個々の Unicode スカラー値に対しては、chars メソッドを使用してください。“Зд” に対して chars を呼び出したら、分解して 2 つの char 型の値を返すので、各要素にアクセスするには、その結果を走査すればいいわけです:

```
for c in "Зд".chars() {
    println!("{c}");
}
```

このコードは、以下のように出力します:

```
З
д
```

あるいは、bytes メソッドは各バイトをそのまま返すので、ドメインによってはこちらが適切かもしれません:

```
for b in "Зд".bytes() {
    println!("{b}");
}
```

このコードは、この文字列をなす 4 バイトを出力します:

```
208
151
208
180
```

ですが、合法な Unicode スカラー値は、2 バイト以上からなる場合もあることは心得ておいてください。

デーヴァナーガリー文字を含むような文字列から書記素クラスタを得る方法は複雑なので、この機能は標準ライブラリでは提供されていません。この機能が必要なら、crates.io でクレートを入手可能です。

8.2.7　文字列はそう単純じゃない

まとめると、文字列は込み入っています。プログラミング言語ごとにこの複雑性をプログラマに提示する方法は違います。Rust では、`String` データを正しく扱うことが、全ての Rust プログラムにとっての既定動作になっているわけであり、これは、プログラマが UTF-8 データを素直に扱う際に、よりしっかり考えないといけないことを意味します。このトレードオフにより、他のプログラミング言語で見えるよりも文字列の複雑性がより露出していますが、ASCII 以外の文字に関するエラーを開発の後半で扱わなければならない可能性が排除されているのです。

一方で良い面としては、こうした複雑な状況に正しく対処するために、標準ライブラリが `String` と `&str` 型の上に構築された多数の機能を提供していることです。文字列内の検索のための `contains` や、文字列の一部を別の文字列で置換するための `replace` などの便利なメソッドについて、ドキュメントを確認してみてください。

もう少し複雑でないものに切り替えていきましょう: ハッシュマップです！

8.3　キーとそれに紐づいた値をハッシュマップに格納する

一般的なコレクションのトリを飾るのは、**ハッシュマップ**です。型 `HashMap<K, V>` は、`K` 型のキーと `V` 型の値の対応関係を**ハッシュ関数**を使用して保持します。ハッシュ関数は、キーと値のメモリ配置方法を決めるものです。多くのプログラミング言語でもこの種のデータ構造はサポートされていますが、しばしば名前が違います。hash、map、object、ハッシュテーブル、辞書、連想配列など、枚挙に暇 (いとま) はありません。

ハッシュマップは、ベクタのように番号ではなく、どんな型にもなりうるキーを使ってデータを参照したいときに有用です。例えば、ゲームにおいて、各チームのスコアをハッシュマップで追いかけることができます。ここで、各キーはチーム名、値が各チームのスコアになります。チーム名が与えられれば、スコアを扱うことができるわけです。

この節でハッシュマップの基礎的な API を見ていきますが、より多くのグッズが標準ライブラリに

より、HashMap<K, V> 上に定義された関数に隠されています。いつものように、もっと情報が欲しければ、標準ライブラリのドキュメントをチェックしてください。

8.3.1　新規ハッシュマップを生成する

空のハッシュマップを作成する方法のひとつは new を使用する方法で、その後要素を insert で追加することができます。リスト 8-20 では、名前が**ブルー**と**イエロー**の 2 チームのスコアを追いかけています。ブルーチームは 10 点から、イエローチームは 50 点から始まります。

```
# fn main() {
    use std::collections::HashMap;

    let mut scores = HashMap::new();

    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Yellow"), 50);
# }
```

リスト 8-20: ハッシュマップを生成してキーと値を挿入する

最初に標準ライブラリのコレクション部分から HashMap を use する必要があることに注意してください。今までの 3 つの一般的なコレクションの内、これが最も使用頻度が低いので、初期化処理で自動的にスコープに導入される機能には含まれていません。また、標準ライブラリからのサポートもハッシュマップは少ないです; 例えば、生成するための組み込みマクロがありません。

ベクタと全く同様に、ハッシュマップはデータをヒープに保持します。この HashMap はキーが String 型、値は i32 型です。ベクタのように、ハッシュマップは均質です: すべてのキーは互いに同じ型でなければならず、値も全て同じ型でなければなりません。

8.3.2　ハッシュマップの値にアクセスする

リスト 8-21 に示したように、キーを get メソッドに提供することで、ハッシュマップから値を取り出すことができます。

```
# fn main() {
    use std::collections::HashMap;

    let mut scores = HashMap::new();

    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Yellow"), 50);

    let team_name = String::from("Blue");
    let score = scores.get(&team_name).copied().unwrap_or(0);
# }
```

リスト 8-21: ハッシュマップに保持されたブルーチームのスコアにアクセスする

ここで、score はブルーチームに紐づけられた値になり、結果は 10 となるでしょう。get メソッドは Option<&V> を返します; キーに対応する値がハッシュマップになかったら、get は None を返すでしょう。このプログラムはこの Option に対して、Option<&i32> ではなく Option<i32> を得るために copied を呼び出し、その後、scores がそのキーに対応するエントリを持たない場合は score を 0 に設定するために、unwrap_or を呼び出して対処します。

ベクタのように、for ループでハッシュマップのキーと値のペアを走査することができます:

```
# fn main() {
    use std::collections::HashMap;

    let mut scores = HashMap::new();

    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Yellow"), 50);

    for (key, value) in &scores {
        println!("{key}: {value}");
    }
# }
```

このコードは、各ペアを任意の順番で出力します:

```
Yellow: 50
Blue: 10
```

8.3.3　ハッシュマップと所有権

i32 のような Copy トレイトを実装する型について、値はハッシュマップにコピーされます。String のような所有権のある値なら、値はムーブされ、リスト 8-22 でデモされているように、ハッシュマップはそれらの値の所有者になるでしょう。

```
# fn main() {
    use std::collections::HashMap;

    let field_name = String::from("Favorite color");
    let field_value = String::from("Blue");

    let mut map = HashMap::new();
    map.insert(field_name, field_value);
    // field_name and field_value are invalid at this point, try using them and
    // see what compiler error you get!
    // field_nameとfield_valueはこの時点で無効になる。試しに使ってみて
    // どんなコンパイルエラーが出るか確認してみて!
# }
```

リスト 8-22: 一旦挿入されたら、キーと値はハッシュマップに所有されることを示す

`insert` を呼び出して `field_name` と `field_value` がハッシュマップにムーブされた後は、これらの変数を使用することは叶いません。

値への参照をハッシュマップに挿入したら、値はハッシュマップにムーブされません。参照が指している値は、最低でもハッシュマップが有効な間は、有効でなければなりません。これらの問題について詳細には、第 10 章の「ライフタイムで参照を有効化する」節で語ります。

8.3.4　ハッシュマップを更新する

キーと値のペアの数は伸長可能ですが、それぞれの一意なキーには同時に 1 つの値しか紐づけることができません（ただし逆は可能です: 例えば、`scores` ハッシュマップ内にブルーチームとイエローチームはともに値 10 を保存することができます）。

ハッシュマップ内のデータを変えたい時は、すでにキーに値が紐づいている場合の扱い方を決めなければなりません。古い値を新しい値で置き換えて、古い値を完全に無視することもできます。古い値を保持して、新しい値を無視し、キーにまだ値が**ない**場合に新しい値を追加するだけにすることもできます。あるいは、古い値と新しい値を組み合わせることもできます。各方法について見ていきましょう！

8.3.4.1　値を上書きする

キーと値をハッシュマップに挿入し、同じキーを異なる値で挿入したら、そのキーに紐づけられている値は置換されます。リスト 8-23 のコードは、`insert` を二度呼んでいるものの、ハッシュマップには一つのキーと値の組しか含まれません。なぜなら、ブルーチームキーに対する値を 2 回とも挿入しているからです。

```
# fn main() {
    use std::collections::HashMap;

    let mut scores = HashMap::new();

    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Blue"), 25);

    println!("{:?}", scores);
# }
```

リスト 8-23: 特定のキーで保持された値を置き換える

このコードは、`{"Blue": 25}` と出力するでしょう。`10` という元の値は上書きされたのです。

8.3.4.2　キーが存在しない場合のみキーと値を追加する

特定のキーがハッシュマップ内に値とともに存在しているか確認して、以下のアクションを取ることは一般的でしょう: キーがハッシュマップに存在する場合は、既存の値はそのままにします。キーが存在しない場合は、そのキーとそれに対応する値を挿入します。

ハッシュマップには、これを行う `entry` と呼ばれる特別な API があり、これは、引数としてチェックしたいキーを取ります。この `entry` メソッドの戻り値は、`Entry` と呼ばれる enum であり、これは存在したりしなかったりする可能性のある値を表します。イエローチームに対するキーに値が紐づけられているか否か確認したくなったとしましょう。存在しなかったら、50 という値を挿入したく、ブルーチームに対しても同様です。`entry` API を使用すれば、コードはリスト 8-24 のようになります。

```
# fn main() {
    use std::collections::HashMap;

    let mut scores = HashMap::new();
    scores.insert(String::from("Blue"), 10);

    scores.entry(String::from("Yellow")).or_insert(50);
    scores.entry(String::from("Blue")).or_insert(50);

    println!("{:?}", scores);
# }
```

リスト 8-24: `entry` メソッドを使ってキーに値がない場合だけ挿入する

`Entry` 上の `or_insert` メソッドは、対応する `Entry` キーが存在した時にそのキーに対する値への可変参照を返すために定義されており、もしなかったら、引数をこのキーの新しい値として挿入し、新しい値への可変参照を返します。このテクニックの方が、そのロジックを自分で書くよりもはるかに綺麗な上に、borrow checker とも親和性が高くなります。

リスト 8-24 のコードを実行すると、`{"Yellow": 50, "Blue": 10}` と出力するでしょう。最初の `entry` 呼び出しは、まだイエローチームに対する値がないので、値 50 でイエローチームのキーを挿入します。`entry` の 2 回目の呼び出しはハッシュマップを変更しません。なぜなら、ブルーチームには既に 10 という値があるからです。

8.3.4.3　古い値に基づいて値を更新する

ハッシュマップの別の一般的なユースケースは、キーの値を探し、古い値に基づいてそれを更新することです。例えば、リスト 8-25 は、各単語があるテキストに何回出現するかを数え上げるコードを示しています。キーに単語を入れたハッシュマップを使用し、その単語を何回見かけたか追いかけるために値を増やします。ある単語を見かけたのが最初だったら、まず 0 という値を挿入します。

```
# fn main() {
    use std::collections::HashMap;
```

```
    let text = "hello world wonderful world";

    let mut map = HashMap::new();

    for word in text.split_whitespace() {
        let count = map.entry(word).or_insert(0);
        *count += 1;
    }

    println!("{:?}", map);
# }
```

リスト 8-25: 単語とカウントを保持するハッシュマップを使って単語の出現数をカウントする

このコードは、`{"world": 2, "hello": 1, "wonderful": 1}` と出力するでしょう。同じキー/値ペアが、異なる順で印字されるかもしれません:「ハッシュマップの値にアクセスする」の節で説明した、ハッシュマップの走査は任意の順で起こるということを思い出してください。

`split_whitespace` メソッドは、`text` の値から、ホワイトスペースによって区切られた部分スライスを走査するイテレータを返します。`or_insert` 関数は、指定されたキーに対する値への可変参照 (`&mut V`) を返すのです。ここでその可変参照を `count` 変数に保持しているので、その値に代入するには、まずアスタリスク (`*`) で `count` を参照外ししなければならないのです。この可変参照は、`for` ループの終端でスコープを抜けるので、これらの変更は全て安全であり、借用規則により許可されるのです。

8.3.5　ハッシュ関数

`HashMap` はデフォルトでは、ハッシュテーブルに関するサービス拒否 (DoS) 攻撃に対する耐性を提供する、**SipHash** と呼ばれるハッシュ関数を使用します[*1]。これは、利用可能な最速のハッシュアルゴリズムではありませんが、パフォーマンスの欠落と引き換えに安全性を得るというトレードオフは、価値があります。自分のコードをプロファイリングして、自分の目的では標準のハッシュ関数は遅すぎると判明したら、異なる hasher を指定することで別の関数に切り替えることができます。**hasher** とは、`BuildHasher` トレイトを実装する型のことです。トレイトについてとその実装方法については、第 10 章で語ります。必ずしも独自の hasher を 1 から作り上げる必要はありません; crates.io には、他の Rust ユーザによって共有された多くの一般的なハッシュアルゴリズムを実装した hasher を提供するライブラリがあります。

[*1] https://en.wikipedia.org/wiki/SipHash

8.4　まとめ

ベクタ、文字列、ハッシュマップはデータを保持し、アクセスし、変更する必要のあるプログラムで必要になる、多くの機能を提供してくれるでしょう。今なら解決可能なはずの練習問題を用意しました:

- 整数のリストが与えられ、ベクタを使って median(ソートされた時に真ん中に来る値)、mode(最も頻繁に出現する値; ハッシュマップがここでは有効活用できるでしょう) を返してください。
- 文字列をピッグ・ラテン (訳注：英語の言葉遊びの一つ) に変換してください。各単語の最初の子音は、単語の終端に移り、”ay” が足されます。従って、”first” は”irst-fay” になります。ただし、母音で始まる単語には、お尻に”hay” が付け足されます (”apple” は”apple-hay” になります)。UTF-8 エンコードに関する詳細を心に留めておいてください！
- ハッシュマップとベクタを使用して、ユーザに会社の部署に雇用者の名前を追加させられるテキストインターフェイスを作ってください。例えば、”Add Sally to Engineering”(開発部門にサリーを追加) や”Add Amir to Sales”(販売部門にアミールを追加) などです。それからユーザに、ある部署にいる人間の一覧や部署ごとにアルファベット順で並べ替えられた会社の全人間の一覧を扱わせてあげてください。

標準ライブラリの API ドキュメントには、この練習問題に有用な、ベクタ、文字列、ハッシュマップのメソッドが解説されています。

処理が失敗することもあるような、より複雑なプログラムに入り込んできています。ということは、エラーの処理法について議論するのにぴったりということです。次はそれをします！

第9章

エラー処理

ソフトウェアにおいて、エラーは避けられない現実です。そのため、Rust には何かがおかしくなる場面に対処するための機能がたくさんあります。多くの場面で、コンパイラは、プログラマにエラーの可能性を知り、コードのコンパイルが通るまでに何かしら対応を行うことを要求してきます。この要求により、エラーを発見し、コードを実用に供する前に適切に対処していることを確認することでプログラムを頑健なものにしてくれるのです！

Rust では、エラーは大きく二つに分類されます: **回復可能**と**回復不能**なエラーです。**ファイルが見つからない**などの回復可能なエラーには、ほとんどの場合は単に問題をユーザに報告し、処理を再試行したいことでしょう。回復不能なエラー、例えば配列の境界を超えた箇所にアクセスしようとすることなどは、常にバグの兆候であり、直ちにプログラムを停止したいことでしょう。

多くの言語では、この 2 種のエラーを区別することはなく、例外などの機構を使用して同様に扱います。Rust には例外が存在しません。代わりに、回復可能なエラーには `Result<T, E>` 型があり、プログラムが回復不能なエラーに遭遇した時には、実行を中止する `panic!` マクロがあります。この章では、まず `panic!` の呼び出しを講義し、それから `Result<T, E>` を戻り値にする話をします。加えて、エラーからの回復を試みるか、実行を中止するか決定する際に考慮すべき事項についても、探究しましょう。

9.1 `panic!`で回復不能なエラー

時として、コード内で悪いことが起き、それに対してできることは何も無い、ということがあるでしょう。このような場面で、Rust には `panic!` マクロが用意されています。実際にパニックを発生させる方法は 2 つあります: コードをパニックさせるような操作（配列にその終端を超えてアクセスするなど）を行う方法と、明示的に `panic!` マクロを呼び出す方法です。いずれの場合でも、プログラムでパニックが発生します。デフォルトでは、これらのパニックは失敗メッセージを出力し、スタックを巻き戻し、片付け、プログラムを終了させます。パニックが発生したときにはそのパニックの発

生源を特定しやすくするために、環境変数を介してコールスタックを表示するように指示することもできます。

9.1.1　パニックに対してスタックを巻き戻すかアボートするか

デフォルトでは、パニックが発生すると、プログラムは**巻き戻し**を始めます。つまり、コンパイラ生成コードがスタックを遡り、遭遇した各関数のデータを片付けるということです。しかし、この遡行と片付けはすべきことが多くなります。そのため Rust では、即座に**アボート**し、片付けをせずにプログラムを終了させることを、代替として選択できるようになっています。そうなると、プログラムが使用していたメモリは、OS が片付ける必要があります。プロジェクトにおいて、実行可能ファイルを極力小さくする必要があれば、**Cargo.toml** ファイルの適切な `[profile]` 欄に `panic = 'abort'` を追記することで、パニック時に巻き戻しからアボートするように切り替えることができます。例として、リリースモード時にアボートするようにしたければ、以下を追記してください:

```
[profile.release]
panic = 'abort'
```

単純なプログラムで `panic!` の呼び出しを試してみましょう:

ファイル名: src/main.rs

```
fn main() {
    //     "クラッシュして炎上"
    panic!("crash and burn");
}
```

このプログラムを実行すると、以下のような出力を目の当たりにするでしょう:

```
$ cargo run
   Compiling panic v0.1.0 (file:///projects/panic)
    Finished dev [unoptimized + debuginfo] target(s) in 0.25s
     Running `target/debug/panic`
thread 'main' panicked at src/main.rs:2:5:
crash and burn
('main'スレッドはsrc/main.rs:2:5でパニックしました:
クラッシュして炎上)
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

`panic!` の呼び出しが、最後の 2 行に含まれるエラーメッセージを発生させているのです。1 行目にパニックメッセージとソースコード中でパニックが発生した箇所を示唆しています: **src/main.rs:2:5** は、**src/main.rs** ファイルの 2 行目 5 文字目であることを示しています。

この場合、示唆される行は、自分のコードの一部で、その箇所を見に行けば、`panic!` マクロ呼び出

しがあるわけです。それ以外では、panic! 呼び出しが、自分のコードが呼び出しているコードの一部になっている可能性もあるわけです。エラーメッセージで報告されるファイル名と行番号が、結果的に panic! 呼び出しに導いた自分のコードの行ではなく、panic! マクロが呼び出されている他人のコードになるでしょう。panic! 呼び出しの発生元である関数のバックトレースを使用して、問題を起こしている自分のコードの箇所を割り出すことができます。次はバックトレースについてはより詳しく議論しましょう。

9.1.2　panic! バックトレースを使用する

別の例を眺めて、自分のコードでマクロを直接呼び出す代わりに、コードに存在するバグにより、ライブラリで panic! 呼び出しが発生するとどんな感じなのか確かめてみましょう。リスト 9-1 は、ベクタに有効な添え字の範囲の外の添え字でアクセスを試みるコードです。

ファイル名: src/main.rs

```
fn main() {
    let v = vec![1, 2, 3];

    v[99];
}
```

リスト 9-1: ベクタの境界を超えて要素へのアクセスを試み、panic! の呼び出しを発生させる

ここでは、ベクタの 100 番目の要素 (添え字は 0 始まりなので添え字 99) にアクセスを試みていますが、ベクタには 3 つしか要素がありません。この場面では、Rust はパニックします。[] の使用は、要素を返すと想定されるものの、無効な添え字を渡せば、ここで Rust が返せて正しいと思われる要素は何もないわけです。

C では、データ構造の終端を超えて読み込みを行おうとすることは未定義動作です。メモリがデータ構造に属していないにもかかわらず、そのデータ構造内のその要素に対応するメモリ上の箇所にある何かを返してくるかもしれません。これは、**バッファオーバーリード (buffer overread)** と呼ばれ、攻撃者が、データ構造の後に格納された読めるべきでないデータを読み出せるように添え字を操作できたら、セキュリティ脆弱性につながる可能性があります。

この種の脆弱性からプログラムを保護するために、存在しない添え字の要素を読もうとしたら、Rust は実行を中止し、継続を拒みます。試して確認してみましょう:

```
$ cargo run
   Compiling panic v0.1.0 (file:///projects/panic)
    Finished dev [unoptimized + debuginfo] target(s) in 0.27s
     Running `target/debug/panic`
thread 'main' panicked at src/main.rs:4:6:
index out of bounds: the len is 3 but the index is 99
('main'スレッドはsrc/main.rs:4:6でパニックしました:
境界外番号: 長さは3なのに、添え字は99です)
```

```
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

このエラーは **main.rs** の 4 行目を指していて、ここではインデックス 99 にアクセスを試みています。その次の注釈行は、RUST_BACKTRACE 環境変数をセットして、まさしく何が起き、エラーが発生したのかのバックトレースを得られることを教えてくれています。**バックトレース**とは、ここに至るまでに呼び出された全関数の一覧です。Rust のバックトレースも、他の言語同様に動作します: バックトレースを読むコツは、頭からスタートして自分のファイルを見つけるまで読むことです。そこが、問題が発生した場所です。この場所より上の行は、自分のコードが呼び出したコードになります; それより下の行は、自分のコードを呼び出しているコードになります。これらの前後の行には、Rust の核となるコード、標準ライブラリのコード、使用しているクレートなどが含まれるかもしれません。RUST_BACKTRACE 環境変数を 0 以外の値にセットして、バックトレースを出力してみましょう。リスト 9-2 のような出力が得られるでしょう。

```
$ RUST_BACKTRACE=1 cargo run
thread 'main' panicked at src/main.rs:4:6:
index out of bounds: the len is 3 but the index is 99
stack backtrace:
   0: rust_begin_unwind
             at /rustc/07dca489ac2d933c78d3c5158e3f43beefeb02ce/library/std/src/
                 panicking.rs:645:5
   1: core::panicking::panic_fmt
             at /rustc/07dca489ac2d933c78d3c5158e3f43beefeb02ce/library/core/src
                 /panicking.rs:72:14
   2: core::panicking::panic_bounds_check
             at /rustc/07dca489ac2d933c78d3c5158e3f43beefeb02ce/library/core/src
                 /panicking.rs:208:5
   3: <usize as core::slice::index::SliceIndex<[T]>>::index
             at /rustc/07dca489ac2d933c78d3c5158e3f43beefeb02ce/library/core/src
                 /slice/index.rs:255:10
   4: core::slice::index::<impl core::ops::index::Index<I> for [T]>::index
             at /rustc/07dca489ac2d933c78d3c5158e3f43beefeb02ce/library/core/src
                 /slice/index.rs:18:9
   5: <alloc::vec::Vec<T,A> as core::ops::index::Index<I>>::index
             at /rustc/07dca489ac2d933c78d3c5158e3f43beefeb02ce/library/alloc/
                 src/vec/mod.rs:2770:9
   6: panic::main
             at ./src/main.rs:4:6
   7: core::ops::function::FnOnce::call_once
             at /rustc/07dca489ac2d933c78d3c5158e3f43beefeb02ce/library/core/src
                 /ops/function.rs:250:5
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose
    backtrace.
```

リスト 9-2: RUST_BACKTRACE 環境変数をセットした時に表示される、panic! 呼び出しが生成するバックトレース

出力が多いですね！ OS や Rust のバージョンによって、出力の詳細は変わる可能性があります。

この情報とともに、バックトレースを得るには、デバッグシンボルを有効にしなければなりません。デバッグシンボルは、`--release` オプションなしで `cargo build` や `cargo run` を使用していれば、標準で有効になり、ここではそうなっています。

リスト 9-2 の出力で、バックトレースの 6 行目が問題発生箇所を指し示しています: **src/main.rs** の 4 行目です。プログラムにパニックしてほしくなければ、自分のファイルについて言及している最初の行で示されている箇所から調査を開始すべきです。わざとパニックするコードを書いたリスト 9-1 において、パニックを解消する方法は、ベクタの添え字の範囲を超えた要素を要求しないようにすることです。将来コードがパニックしたら、パニックを引き起こすどんな値でコードがどんな動作をしているのかと、代わりにコードは何をすべきなのかを算出する必要があるでしょう。

この章の後ほど、「`panic!` すべきかするまいか」節で `panic!` とエラー状態を扱うのに `panic!` を使うべき時と使わぬべき時に戻ってきます。次は、`Result` を使用してエラーから回復する方法を見ましょう。

9.2　`Result` で回復可能なエラー

多くのエラーは、プログラムを完全にストップさせるほど深刻ではありません。時々、関数が失敗した時に、容易に解釈し、対応できる理由によることがあります。例えば、ファイルを開こうとして、ファイルが存在しないために処理が失敗したら、プロセスを停止するのではなく、ファイルを作成したいことがあります。

第 2 章の「`Result` で失敗の可能性を扱う」で `Result` enum が以下のように、`Ok` と `Err` の 2 列挙子からなるよう定義されていることを思い出してください:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

`T` と `E` は、ジェネリックな型引数です: ジェネリクスについて詳しくは、第 10 章で議論します。たった今知っておく必要があることは、`T` が成功した時に `Ok` 列挙子に含まれて返される値の型を表すことと、`E` が失敗した時に `Err` 列挙子に含まれて返されるエラーの型を表すことです。`Result` はこのようなジェネリックな型引数を含むので、`Result` 型とその上に定義されている関数を、成功した時とエラーの時に返したい値が異なるような様々な場面で使用できるのです。

関数が失敗する可能性があるために `Result` 値を返す関数を呼び出しましょう: リスト 9-3 では、ファイルを開こうとしています。

ファイル名: src/main.rs

```
use std::fs::File;

fn main() {
```

```
    let greeting_file_result = File::open("hello.txt");
}
```

リスト 9-3: ファイルを開く

File::open の戻り値の型は、Result<T, E> です。ジェネリック引数の T は、File::open の実装によって成功値の型 std::fs::File で埋められていて、これはファイルハンドルです。エラー値で使用されている E の型は、std::io::Error です。この戻り値型は、File::open の呼び出しが成功し、読み込みと書き込みを行えるファイルハンドルを返す可能性があることを意味します。また、関数呼び出しは失敗もする可能性があります: 例えば、ファイルが存在しない可能性、ファイルへのアクセス権限がない可能性です。File::open には成功したか失敗したかを知らせる方法とファイルハンドルまたは、エラー情報を与える方法が必要なのです。この情報こそが Result enum が伝達するものなのです。

File::open が成功した場合、変数 greeting_file_result の値はファイルハンドルを含む Ok インスタンスになります。失敗した場合には、発生したエラーの種類に関する情報をより多く含む Err インスタンスが greeting_file_result の値になります。

リスト 9-3 のコードに追記をして File::open が返す値に応じて異なる動作をする必要があります。リスト 9-4 に基礎的な道具を使って Result を扱う方法を一つ示しています。第 6 章で議論した match 式です。

ファイル名: src/main.rs

```
use std::fs::File;

fn main() {
    let greeting_file_result = File::open("hello.txt");

    let greeting_file = match greeting_file_result {
        Ok(file) => file,
        //                  "ファイルを開くのに問題がありました: {:?}"
        Err(error) => panic!("Problem opening the file: {:?}", error),
    };
}
```

リスト 9-4: match 式を使用して返却される可能性のある Result 列挙子を処理する

Option enum のように、Result enum とその列挙子は、prelude でスコープ内に持ち込まれているので、match アーム内で Ok と Err 列挙子の前に Result:: を指定する必要がないことに注目してください。

このコードは、結果が Ok の場合は、Ok 列挙子から中身の file 値を返し、それからそのファイルハンドル値を変数 greeting_file に代入しています。match の後には、ファイルハンドルを使用して読み込んだり書き込むことができるわけです。

match のもう一つのアームは、File::open から Err 値が得られたケースを処理しています。この例では、panic! マクロを呼び出すことを選択しています。カレントディレクトリに **hello.txt** というファイルがなく、このコードを走らせたら、panic! マクロからの以下のような出力を目の当たりにするでしょう:

```
$ cargo run
   Compiling error-handling v0.1.0 (file:///projects/error-handling)
    Finished dev [unoptimized + debuginfo] target(s) in 0.73s
     Running `target/debug/error-handling`
thread 'main' panicked at src/main.rs:8:23:
Problem opening the file: Os { code: 2, kind: NotFound, message: "No such file
    or directory" }
('main'スレッドは、src/main.rs:8:23でパニックしました:
ファイルを開く際に問題がありました: Os { code: 2, kind: NotFound, message: "その
    ようなファイルやディレクトリはありません" })
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

通常通り、この出力は、一体何がおかしくなったのかを物語っています。

9.2.1　色々なエラーにマッチする

リスト 9-4 のコードは、File::open が失敗した理由にかかわらず panic! します。ですが、失敗理由によって動作を変えたいとしましょう: ファイルが存在しないために File::open が失敗したら、ファイルを作成し、その新しいファイルへのハンドルを返したいです。他の理由 (例えばファイルを開く権限がなかったなど) で、File::open が失敗したら、リスト 9-4 のようにコードには panic! してほしいのです。このために、リスト 9-5 で示すように内側の match 式を追加します。

ファイル名: src/main.rs

```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let greeting_file_result = File::open("hello.txt");

    let greeting_file = match greeting_file_result {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("hello.txt") {
                Ok(fc) => fc,
                //                "ファイルを作成するのに問題がありました: {:?}"
                Err(e) => panic!("Problem creating the file: {:?}", e),
            },
            other_error => {
                //     "ファイルを開くのに問題がありました: {:?}"
                panic!("Problem opening the file: {:?}", other_error);
            }
        },
```

```
    };
}
```

リスト 9-5: 色々な種類のエラーを異なる方法で扱う

`File::open` が `Err` 列挙子に含めて返す値の型は、`io::Error` であり、これは標準ライブラリで提供されている構造体です。この構造体には、呼び出すと `io::ErrorKind` 値が得られる `kind` メソッドがあります。`io::ErrorKind` という enum は、標準ライブラリで提供されていて、`io` 処理の結果発生する可能性のある色々な種類のエラーを表す列挙子があります。使用したい列挙子は、`ErrorKind::NotFound` で、これは開こうとしているファイルがまだ存在しないことを示唆します。そこで、`greeting_file_result` に対してマッチし、さらに `error.kind()` に対する内側のマッチも持たせています。

内側のマッチで精査したい条件は、`error.kind()` により返る値が、`ErrorKind` enum の `NotFound` 列挙子であるかということです。もしそうなら、`File::create` でファイル作成を試みます。ところが、`File::create` も失敗する可能性があるので、内側の `match` 式の 2 番目のアームが必要なのです。ファイルを作成できない場合、異なるエラーメッセージが出力されます。外側の `match` の 2 番目のアームは同じままなので、ファイルが存在しないエラー以外ならプログラムはパニックします。

<!–

9.2.2 Alternatives to Using `match` with `Result<T, E>`

That’s a lot of `match`! The `match` expression is very useful but also very much a primitive. In Chapter 13, you’ll learn about closures, which are used with many of the methods defined on `Result<T, E>`. These methods can be more concise than using `match` when handling `Result<T, E>` values in your code.

For example, here’s another way to write the same logic as shown in Listing 9-5, this time using closures and the `unwrap_or_else` method:

9.2.3 `Result<T, E>`に対する `match` の使用の代わりとなる方法

`match` がたくさん出てきましたね！ `match` 式は非常に有用ですが、非常に原始的でもあります。第 13 章でクロージャについて学習しますが、これは `Result<T, E>` 上に定義されているメソッドの多くで使用することができます。コード内で `Result<T, E>` 値を扱うときには、こうしたメソッドを使ったほうがより簡潔になります。

例えば、以下はリスト 9-5 に示したものと同じロジックを書く例ですが、今度はクロージャと `unwrap_or_else` メソッドを使用しています:

```
use std::fs::File;
```

```
use std::io::ErrorKind;

fn main() {
    let greeting_file = File::open("hello.txt").unwrap_or_else(|error| {
        if error.kind() == ErrorKind::NotFound {
            File::create("hello.txt").unwrap_or_else(|error| {
                panic!("Problem creating the file: {:?}", error);
            })
        } else {
            panic!("Problem opening the file: {:?}", error);
        }
    });
}
```

このコードはリスト 9-5 と同じ動作をしますが、match 式をまったく含んでおらず、より読みやすいです。第 13 章を読み終えたら、この例に戻ってきて、標準ライブラリドキュメント内で unwrap_or_else を探してみてください。エラーに対処するときには、これらのメソッドを多用することで、巨大なネストされた match 式を整理することができます。

9.2.4　エラー時にパニックするショートカット: unwrap と expect

match の使用は、十分に仕事をしてくれますが、いささか冗長になり得る上、必ずしも意図をよく伝えるとは限りません。Result<T, E> 型には、様々な特定の作業をするヘルパーメソッドが多く定義されています。unwrap メソッドは、リスト 9-4 で書いた match 式と同じように実装された短絡メソッドです。Result 値が Ok 列挙子なら、unwrap は Ok の中身を返します。Result が Err 列挙子なら、unwrap は panic! マクロを呼んでくれます。こちらが実際に動作している unwrap の例です:

ファイル名: src/main.rs

```
use std::fs::File;

fn main() {
    let greeting_file = File::open("hello.txt").unwrap();
}
```

このコードを **hello.txt** ファイルなしで走らせたら、unwrap メソッドが行う panic! 呼び出しからのエラーメッセージを目の当たりにするでしょう:

```
thread 'main' panicked at src/main.rs:4:49:
called `Result::unwrap()` on an `Err` value: Os { code: 2, kind: NotFound,
    message: "No such file or directory" }
('main'スレッドは、src/main.rs:4:49でパニックしました:
`Err`値に対して`Result::unwrap()`が呼び出されました: Os { code: 2, kind:
    NotFound, message: "そのようなファイルやディレクトリはありません" })
```

同様に、expect メソッドは、panic! のエラーメッセージも選択させてくれます。unwrap の代わりに expect を使用して、いいエラーメッセージを提供すると、意図を伝え、パニックの原因をたどりやすくしてくれます。expect の表記はこんな感じです:

ファイル名: src/main.rs

```
use std::fs::File;

fn main() {
    let greeting_file = File::open("hello.txt")
        //      "hello.txtがこのプロジェクトに含まれているべきです"
        .expect("hello.txt should be included in this project");
}
```

expect を unwrap と同じように使用してます: ファイルハンドルを返したり、panic! マクロを呼び出しています。expect が panic! 呼び出しで使用するエラーメッセージは、unwrap が使用するデフォルトの panic! メッセージではなく、expect に渡した引数になります。以下のようになります:

```
thread 'main' panicked at src/main.rs:5:10:
hello.txt should be included in this project: Os { code: 2, kind: NotFound,
    message: "No such file or directory" }
('main'スレッドは、src/main.rs:5:10でパニックしました:
hello.txtがこのプロジェクトに含まれているべきです: Os { code: 2, kind: NotFound,
     message: "そのようなファイルやディレクトリはありません" })
```

製品レベルの品質のコードでは、多くの Rustacean は unwrap よりむしろ expect を選択し、なぜその操作が常に成功すると想定されるのかについてより多くの文脈情報を提供します。そうすることで、万一あなたの仮定が誤っていたと判明した場合には、デバッグ時に利用するためのより多くの情報が得られます。

9.2.5　エラーを委譲する

関数の実装が失敗する可能性のある何かを呼び出す際、その関数自体の中でエラーを処理する代わりに、呼び出し元がどうするかを決められるようにエラーを返すことができます。これはエラーの**委譲**として認知され、自分のコードの文脈で利用可能なものよりも、エラーの処理法を規定する情報やロジックがより多くある呼び出し元のコードに制御を明け渡します。

例えば、リスト 9-6 の関数は、ファイルからユーザ名を読み取ります。ファイルが存在しなかったり、読み込みできなければ、この関数はそのようなエラーを呼び出し元のコードに返します。

ファイル名: src/main.rs

```
use std::fs::File;
use std::io::{self, Read};
```

```
fn read_username_from_file() -> Result<String, io::Error> {
    let username_file_result = File::open("hello.txt");

    let mut username_file = match username_file_result {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut username = String::new();

    match username_file.read_to_string(&mut username) {
        Ok(_) => Ok(username),
        Err(e) => Err(e),
    }
}
```

リスト 9-6: match でエラーを呼び出し元のコードに返す関数

この関数はもっと短く書くことができますが、エラー処理について詳しく見るために、まずはエラー処理をたくさん手動で書くことから始めましょう; より短い方法は最後に示します。まずは、関数の戻り値型に注目しましょう: Result<String, io::Error> です。つまり、この関数は、Result<T, E> 型の値を返しているということです。ここでジェネリック引数の T は、具体型 String で埋められ、ジェネリック引数の E は具体型 io::Error で埋められています。

この関数が何の問題もなく成功すれば、この関数を呼び出したコードは、String (関数がファイルから読み取ったユーザ名) を保持する Ok 値を受け取ります。この関数が何か問題に行き当たったら、呼び出し元のコードは io::Error のインスタンスを保持する Err 値を受け取り、この io::Error は問題の内容に関する情報をより多く含んでいます。関数の戻り値の型に io::Error を選んだのは、この関数本体で呼び出している失敗する可能性のある処理が両方とも偶然この型をエラー値として返すからです: File::open 関数と read_to_string メソッドです。

関数の本体は、File::open 関数を呼び出すところから始まります。そして、リスト 9-4 の match に似た match で Result 値を扱います。File::open が成功すれば、パターン変数 file にあるファイルハンドルは可変変数 username_file 内の値となり、関数は継続します。Err ケースでは panic! を呼び出す代わりに、関数から完全に早期リターンしてこの関数のエラー値として、File::open から得たエラー値、これはパターン変数 e 内にありますが、これを呼び出し元に渡し戻すために return キーワードを使用します。

username_file にファイルハンドルが得られたら、関数は次に変数 username に新規 String を生成し、username_file のファイルハンドルに対して read_to_string を呼び出して、ファイルの中身を username に読み出します。File::open が成功しても、失敗する可能性があるので、read_to_string メソッドも、Result を返却します。その Result を処理するために別の match が必要になります: read_to_string が成功したら、関数は成功し、今は Ok に包まれた username に入っているファイルのユーザ名を返却します。read_to_string が失敗したら、File::open の戻り値を扱った match でエラー値を返したように、エラー値を返します。しかし、明示的に return を述べる必要はありません。

これが関数の最後の式だからです。

そうしたら、呼び出し元のコードは、ユーザ名を含む Ok 値か、io::Error を含む Err 値を得て扱います。それらの値をどうするかを決めるのは、呼び出し元のコードに委ねられます。呼び出しコードが Err 値を得たら、例えば、panic! を呼び出してプログラムをクラッシュさせたり、デフォルトのユーザ名を使ったり、ファイル以外の場所からユーザ名を検索したりできるでしょう。呼び出し元のコードが実際に何をしようとするかについて、十分な情報がないので、成功や失敗情報を全て委譲して適切に扱えるようにするのです。

Rust において、この種のエラー委譲は非常に一般的なので、Rust にはこれをしやすくする ? 演算子が用意されています。

9.2.5.1　エラー委譲のショートカット: ?演算子

リスト 9-7 はリスト 9-6 にあったのと同じ機能を持つ read_username_from_file の実装ですが、こちらは ? 演算子を使用しています。

ファイル名: src/main.rs

```
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let mut username_file = File::open("hello.txt")?;
    let mut username = String::new();
    username_file.read_to_string(&mut username)?;
    Ok(username)
}
```

リスト 9-7: ? 演算子でエラーを呼び出し元に返す関数

Result 値の直後に置かれた ? は、リスト 9-6 で Result 値を処理するために定義した match 式とほぼ同じように動作します。Result の値が Ok なら、Ok の中身がこの式から返ってきて、プログラムは継続します。値が Err なら、return キーワードを使ったかのように関数全体から Err が返ってくるので、エラー値は呼び出し元のコードに委譲されます。

リスト 9-6 の match 式がやっていることと ? 演算子がやっていることには、違いがあります: ? 演算子が呼ばれる対象となったエラー値は、標準ライブラリの From トレイトで定義され、値を別の型に変換する from 関数を通ることです。? 演算子が from 関数を呼び出すと、受け取ったエラー型が現在の関数の戻り値型で定義されているエラー型に変換されます。これは、個々がいろんな理由で失敗する可能性があるのにも関わらず、関数が失敗する可能性を全て一つのエラー型で表現して返す時に有用です。

例えば、リスト 9-7 の read_username_from_file 関数を、OurError という名前の自分で定義したカスタムエラー型を返すように変更することができます。さらに、io::Error から OutError のインスタンスを構築するための impl From<io::Error> for OurError を定義すれば、

`read_username_from_file` の本体の中の `?` 演算子呼び出しは `from` を呼び出してくれるので、関数にコードを追加する必要なくエラー型を変換してくれます。

リスト 9-7 の文脈では、`File::open` 呼び出し末尾の `?` は `Ok` の中身を変数 `username_file` に返します。エラーが発生したら、`?` 演算子により関数全体から早期リターンし、あらゆる `Err` 値を呼び出し元に与えます。同じ法則が `read_to_string` 呼び出し末尾の `?` にも適用されます。

`?` 演算子により定型コードの多くが排除され、この関数の実装を単純にしてくれます。リスト 9-8 で示したように、`?` の直後のメソッド呼び出しを連結することでさらにこのコードを短くすることさえもできます。

ファイル名: src/main.rs

```
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let mut username = String::new();

    File::open("hello.txt")?.read_to_string(&mut username)?;

    Ok(username)
}
```

リスト 9-8: `?` 演算子の後のメソッド呼び出しを連結する

`username` の新規 `String` の生成を関数の冒頭に移動しました; その部分は変化していません。変数 `username_file` を生成する代わりに、`read_to_string` の呼び出しを直接 `File::open("hello.txt")?` の結果に連結させました。それでも、`read_to_string` 呼び出しの末尾には `?` があり、`File::open` と `read_to_string` 両方が成功したら、エラーを返すというよりもそれでも、`username` を含む `Ok` 値を返します。機能もまたリスト 9-6 及び、9-7 と同じです; ただ単に異なるバージョンのよりエルゴノミックな書き方なのです。

これを `fs::read_to_string` を使用してさらに短くする方法をリスト 9-9 に示します。

ファイル名: src/main.rs

```
use std::fs;
use std::io;

fn read_username_from_file() -> Result<String, io::Error> {
    fs::read_to_string("hello.txt")
}
```

リスト 9-9: ファイルを開いて読む代わりに `fs::read_to_string` を使用する

ファイルを文字列に読み込むことは非常によくある操作なので、標準ライブラリは、ファイルを開

き、新しい `String` を作成し、ファイルの内容を読み込み、内容をその `String` に格納し、それを返す、便利な `fs::read_to_string` 関数を提供しています。そうそう、`fs::read_to_string` を使用してしまうと、エラー処理のすべてについて説明する機会がなくなってしまうので、長ったらしい方法を先に使ったのでした。

9.2.5.2　?演算子が使用できる場所

`?` 演算子は、`?` を使用する対象の値と戻り値の型に互換性がある関数でしか使用できません。というのも `?` 演算子は、リスト 9-6 で定義した `match` 式と同様に関数から値の早期リターンを実行するように定義されているからです。リスト 9-6 では、`match` は `Result` 値を使用して、早期リターンする側のアームは `Err(e)` 値を返していました。関数の戻り値型はこの `return` と互換性を保てるように、`Result` でなければならないのです。

リスト 9-10 で、`?` を使用する対象の値の型と戻り値の型に互換性がない `main` 関数で、`?` 演算子を使用した場合に得られるエラーを見てみましょう:

ファイル名: src/main.rs

```
use std::fs::File;

fn main() {
    let greeting_file = File::open("hello.txt")?;
}
```

リスト 9-10: `()` を返す `main` 関数内で `?` を使用しようとするとコンパイルが通らない

このコードはファイルを開きますが、これは失敗する可能性があります。`?` 演算子は `File::open` によって返される `Result` 値を確認して対処しますが、この `main` 関数は `Result` ではなく `()` の戻り値型を持っています。このコードをコンパイルすると、以下のようなエラーメッセージが得られます:

```
$ cargo run
   Compiling error-handling v0.1.0 (file:///projects/error-handling)
error[E0277]: the `?` operator can only be used in a function that returns `
    Result` or `Option` (or another type that implements `FromResidual`)
(エラー: `?`演算子は`Result`または`Option`(あるいはその他`FromResidual`を実装す
    る型)を返す関数内でのみ使用できます
 --> src/main.rs:4:48
  |
3 | fn main() {
  | --------- this function should return `Result` or `Option` to accept `?`
            (`?`を使えるようにするには、この関数は`Result`または`Option`を返す
                べきです)
4 |     let greeting_file = File::open("hello.txt")?;
  |                                                ^ cannot use the `?` operator
       in a function that returns `()`
                                                    (`()`を返す関数内では`?`演算
                                                        子を使用できません)
```

```
  |
  = help: the trait `FromResidual<Result<Infallible, std::io::Error>>` is not
      implemented for `()`
   (ヘルプ: トレイト`FromResidual<Result<Infallible, std::io::Error>>`は`()`に対
       して実装されていません)

For more information about this error, try `rustc --explain E0277`.
error: could not compile `error-handling` (bin "error-handling") due to 1
    previous error
```

このエラーは、? 演算子は Result 、Option 、またはその他 FromResidual を実装する型を返す関数でしか使用が許可されないと指摘しています。

このエラーを修正する方法としては、二つの選択肢があります。一つは、それを阻害する制限がない場合は、関数の戻り値型を、? 演算子を使用する対象の値と互換性があるような型に変更することです。もう一つの手法は、match または Result<T, E> のメソッドのいずれかを使用して、何らかの適切な方法で Result<T, E> を処理することです。

エラーメッセージは、? は Option<T> 値にも使用できることに言及しています。Result に ? を使用する場合と同様に、Option を返す関数の中でのみ、Option に ? を使用することができます。Option<T> に対して呼ばれた場合の ? 演算子の挙動は、Result<T, E> に対して呼ばれた場合の挙動と似ています: もしその値が None ならば、その時点で None が関数から早期リターンされます。もしその値が Some ならば、Some の内側の値がその式の結果の値となり、関数は継続します。リスト 9-11 は、与えられたテキストの最初の行の最後の文字を探索する関数の例です:

```
fn last_char_of_first_line(text: &str) -> Option<char> {
    text.lines().next()?.chars().last()
}
#
# fn main() {
#     assert_eq!(
#         last_char_of_first_line("Hello, world\nHow are you today?"),
#         Some('d')
#     );
#
#     assert_eq!(last_char_of_first_line(""), None);
#     assert_eq!(last_char_of_first_line("\nhi"), None);
# }
```

リスト 9-11: Option<T> 値に対する ? 演算子の使用

この関数は、文字はあるかもしれないし、ないかもしれないので、Option<char> を返します。このコードは text 文字列スライス引数を受け取り、それに対して lines メソッドを呼び出します。これは文字列内の行を走査するイテレータを返します。この関数では最初の行を確認したいので、イテレータに対して next を呼び出してイテレータから最初の値を取得します。text が空文字列の場合は、この next 呼び出しは None を返すでしょう。その場合は ? を使用して停止し、last_char_of_first_line

から None を返します。text が空文字列でない場合は、next は text 内の最初の行の文字列スライスを含む Some 値を返すでしょう。

? が文字列スライスを抽出するので、その文字列スライスに対して chars を呼び出して、含まれる文字からなるイテレータを得ることができます。最初の行の最後の文字に関心があるので、このイテレータの最後の要素を返すために last を呼び出します。例えば text が空行で始まるが他の行には文字が含まれる "\nhi" のような文字列の場合、最初の行が空文字列である可能性があるので、この戻り値は Option になっています。しかしながら、最初の行の最後の文字がある場合は、それを Some 列挙子に入れて返します。中間の ? 演算子はこのロジックを簡潔に表現する方法を提供し、そのためこの関数を一行で実装することができています。もし ? 演算子が Option に使用できなかったなら、より多くのメソッド呼び出しや match 式を使用してこのロジックを実装する必要があったでしょう。

Result を返す関数内では Result に対して ? 演算子を使用でき、Option を返す関数内では Option に対して ? 演算子を使用できますが、混ぜて組み合わせることはできないことに注意してください。? 演算子は、Result を Option に、またはその逆に、自動的に変換することはしません; そのような場合には明示的に変換を行うために、Result の ok メソッドや、Option の ok_or メソッドのようなメソッドを使用することができます。

今までに使ってきた main 関数はすべて () を返してきました。main 関数は、実行可能なプログラムの開始および終了点であることから特別扱いされており、プログラムが期待通りに振る舞うために、その戻り値型として指定できる型に制限があります。

幸運なことに、main は Result<(), E> を返すこともできます。リスト 9-12 はリスト 9-10 のコードを含んでいますが、main の戻り値型を Result<(), Box<dyn Error>> に変更し、最後に戻り値 Ok(()) を追加しています。これでこのコードはコンパイルできるでしょう:

```
use std::error::Error;
use std::fs::File;

fn main() -> Result<(), Box<dyn Error>> {
    let greeting_file = File::open("hello.txt")?;

    Ok(())
}
```

リスト 9-12: Result<(), E> を返すように main を変更することで、Result 値に対する ? 演算子が使用可能になる

Box<dyn Error> 型は**トレイトオブジェクト**ですが、これについては第 17 章の「トレイトオブジェクトで異なる型の値を許容する」節で話します。今のところは、Box<dyn Error> は「任意の種類のエラー」を意味するものと理解してください。エラー型 Box<dyn Error> を持つ main 関数の中では、任意の Err 値を早期リターンすることができるので、Result 値に対する ? の使用が許可されます。この main 関数の本体は型 std::io::Error のエラーしか返しませんが、それでも Box<dyn Error> を指定することにより、他のエラーを返すコードが main の本体に追加された場合であっても、このシグネ

チャは正しいままであり続けるでしょう。

`main` 関数が `Result<(), E>` を返す場合、その実行可能ファイルは、`main` が `Ok(())` を返す場合には値 `0` で終了し、`main` が `Err` 値を返す場合は非ゼロ値で終了するでしょう。C で書かれた実行可能ファイルは終了時に整数を返します: 正常終了するプログラムは整数 `0` を返し、エラーが発生したプログラムは `0` 以外の整数を返します。Rust もこの慣例に従い、実行可能ファイルから整数を返します。

`main` 関数は `std::process::Termination` トレイトを実装する任意の型を返すことができます。このトレイトには `ExitCode` を返す `report` 関数が含まれます。自身の型に対する `Termination` トレイトの実装に関するさらなる情報については、標準ライブラリドキュメントを確認してください。

さて、`panic!` 呼び出しや `Result` を返す詳細について議論し終えたので、どんな場合にどちらを使うのが適切か決める方法についての話に戻りましょう。

9.3　panic! すべきかするまいか

では、`panic!` を呼ぶべき時と `Result` を返すべき時はどう決定すればいいのでしょうか？ コードがパニックしたら、回復する手段はありません。回復する可能性のある手段の有る無しに関わらず、どんなエラー場面でも `panic!` を呼ぶことはできますが、そうすると、呼び出す側のコードに代わってこの場面は回復不能だという決定を下すことになります。`Result` 値を返す決定をすると、呼び出し側に選択肢を与えることになります。呼び出し側は、場面に合わせて回復を試みることを決定したり、この場合の `Err` 値は回復不能と断定して、`panic!` を呼び出し、回復可能だったエラーを回復不能に変換することもできます。故に、`Result` を返却することは、失敗する可能性のある関数を定義する際には、いい第一選択肢になります。

例やプロトタイプコード、テストなどの場面では、`Result` を返すよりもパニックするコードを書く方がより適切になることもあります。その理由を探ってみて、それからコンパイラではありえない失敗だと気づけなくとも、人間なら気づける場面を議論しましょう。そして、ライブラリコードでパニックするか決定する方法についての一般的なガイドラインで結論づけましょう。

9.3.1　例、プロトタイプコード、テスト

例を記述して何らかの概念を具体化している時、頑健なエラー処理コードを含めることは、例の明瞭さを欠くことになりかねません。例において、`unwrap` などのパニックする可能性のあるメソッド呼び出しは、アプリケーションにエラーを処理してほしい方法へのプレースホルダーを意味していると理解され、これは残りのコードがしていることによって異なる可能性があります。

同様に、`unwrap` や `expect` メソッドは、エラーの処理法を決定する準備ができる前、プロトタイプの段階では、非常に便利です。それらにより、コードにプログラムをより頑健にする時の明らかなマーカーが残されるわけです。

メソッド呼び出しがテスト内で失敗したら、そのメソッドがテスト下に置かれた機能ではなかったとしても、テスト全体が失敗してほしいでしょう。`panic!` が、テストが失敗と印づけられる手段なの

で、unwrap や expect の呼び出しはズバリ起こるべきことです。

9.3.2　コンパイラよりもプログラマがより情報を持っている場合

Result が Ok 値であることを保証する何らかの別のロジックがある場合、unwrap または except を呼び出すことは適切でしょうが、コンパイラは、そのロジックを理解はしません。それでも、処理する必要のある Result は存在するでしょう: 呼び出している処理が何であれ、自分の特定の場面では論理的に起こり得なくても、一般的にまだ失敗する可能性はあるわけです。手動でコードを調査して Err 列挙子は存在しないと確認できたら、unwrap を呼び出すことは完全に受容できることです。決して Err 列挙子にならないと考える理由を expect のテキストに文書化すると、さらに良いでしょう。こちらが例です:

```
# fn main() {
    use std::net::IpAddr;

    let home: IpAddr = "127.0.0.1"
        .parse()
        //      "ハードコードされたIPアドレスは有効であるべきです"
        .expect("Hardcoded IP address should be valid");
# }
```

ハードコードされた文字列を構文解析することで IpAddr インスタンスを生成しています。プログラマには 127.0.0.1 が合法な IP アドレスであることがわかるので、ここで expect を使用することは、受容可能なことです。しかしながら、ハードコードされた合法な文字列が存在することは、parse メソッドの戻り値型を変えることにはなりません: それでも得られるのは、Result 値であり、コンパイラは Err 列挙子になる可能性があるかのように Result を処理することを強制してきます。コンパイラは、この文字列が常に合法な IP アドレスであると把握できるほど利口ではないからです。プログラムにハードコードされるのではなく、IP アドレス文字列がユーザ起源でそれ故に**確かに**失敗する可能性がある場合、Result をもっと頑健な方法で処理したほうが絶対にいいでしょう。この IP アドレスはハードコードされたものだという想定に言及することで、もし将来 IP アドレスを他の情報源から取得する必要がでてきた場合、expect の部分をよりよいエラー処理コードに変更することが促されるでしょう。

9.3.3　エラー処理のガイドライン

コードが悪い状態に陥る可能性があるときには、パニックさせることが推奨されます。この文脈での**悪い状態**とは、無効な値、矛盾する値、欠けた値が自身のコードに渡されるなど、何らかの仮定、保証、契約、不変条件が破られることに加え、以下のいずれか一つ以上に当てはまる状態のことです:

- 悪い状態は、予期されていない何かです。ユーザが誤ったフォーマットでデータを入力することなどの、場合によっては起こりうることとは異なります。

- この時点以降の自身のコードは、各段階で問題が無いか確認するのではなく、この悪い状態にないことに依存する必要があります。
- 使用している型にこの情報をコード化するいい手段がありません。これが意味するところの例は、第 17 章の「状態と振る舞いを型としてコード化する」節で取り組みます。

誰かが自分のコードを呼び出して意味をなさない値を渡してきた場合は、可能であれば、ライブラリの利用者がその場合にどうしたいのかを決定できるように、エラーを返すのが最もよいでしょう。ですが、実行を継続するのが安全でなかったり有害かもしれない場合は、最善の選択肢は `panic!` を呼び出し、開発段階で修正できるように自分たちのコードにバグがあることをライブラリ使用者に通知することかもしれません。同様に自分の制御下にない外部コードを呼び出し、修正しようのない無効な状態を返すときに `panic!` はしばしば適切です。

しかしながら、失敗が予測できる場合は、`panic!` 呼び出しをするよりも `Result` を返すほうがより適切です。その例には、不正なデータを渡されたパーサとか、レートリミットに引っかかったことを示唆するステータスを返す HTTP リクエストなどが挙げられます。このような場合では `Result` を返すことで、失敗は、呼び出し側が処理方法を決定しなくてはならない、予測できる可能性であることが示されます。

あなたのコードが、不正な値を使用して呼ばれるとユーザを危険に晒すおそれのある処理を行う場合、コードはまず値が合法であることを確認し、値が合法でなければパニックするべきです。これはほぼ安全性上の理由によるものです: 不正なデータの処理を試みると、コードを脆弱性に晒す可能性があります。これが、境界外へのメモリアクセスを試みたときに標準ライブラリが `panic!` を呼び出す主な理由です: 現在のデータ構造に属しないメモリにアクセスを試みることは、ありふれたセキュリティ問題なのです。関数にはしばしば**契約**が伴います: 入力が特定の条件を満たすときのみ、振る舞いが保証されるのです。契約が侵されたときにパニックすることは、道理が通っています。なぜなら、契約侵害は常に呼び出し側のバグを示唆し、呼び出し側に明示的に処理してもらう必要のある種類のエラーではないからです。実際に、呼び出し側が回復する合理的な手段はありません; 呼び出し側の**プログラマ**がコードを修正する必要があるのです。関数の契約は、特に侵害がパニックを引き起こす際には、関数の API ドキュメント内で説明されているべきです。

ですが、全ての関数でたくさんのエラーチェックを行うことは冗長で煩わしいことでしょう。幸運にも、Rust の型システム (故にコンパイラによって行われる型精査) を使用して多くの検査を行ってもらうことができます。関数の引数に特定の型があるなら、合法な値があるとコンパイラがすでに確認していることを把握して、コードのロジックに進むことができます。例えば、`Option` 以外の型がある場合、プログラムは、**何もない**ではなく**何かある**と想定します。そうしたらコードは、`Some` と `None` 列挙子の 2 つの場合を処理する必要がなくなるわけです: 確実に値があるという可能性しかありません。関数に何もないことを渡そうとしてくるコードは、コンパイルが通りもしませんので、その場合を実行時に検査する必要はないわけです。別の例は、`u32` のような符号なし整数を使うことであり、この場合、引数は負には絶対にならないことが確認されます。

9.3.4　検証のために独自の型を作る

Rust の型システムを使用して合法な値があると確認するというアイディアを一歩先に進め、検証のために独自の型を作ることに目を向けましょう。第 2 章の数当てゲームで、コードがユーザに 1 から 100 までの数字を推測するよう求めたことを思い出してください。秘密の数字と照合する前にユーザの推測がそれらの値の範囲にあることを全く確認しませんでした; 推測が正であることしか確認しませんでした。この場合、結果はそれほど悲惨なものではありませんでした:「大きすぎ」、「小さすぎ」という出力は、それでも正しかったでしょう。しかし、ユーザを合法な推測に導き、ユーザが範囲外の数字を推測したり、例えばユーザが文字を代わりに入力したりしたときに別の挙動をするようにしたら、有益な改善になるでしょう。

これをする一つの方法は、ただの `u32` の代わりに `i32` として推測をパースし、負の数になる可能性を許可し、それから数字が範囲に収まっているというチェックを追加することでしょう。そう、以下のように:

```
# use rand::Rng;
# use std::cmp::Ordering;
# use std::io;
#
# fn main() {
#     println!("Guess the number!");
#
#     let secret_number = rand::thread_rng().gen_range(1..=100);
#
    loop {
        // --snip--

#         println!("Please input your guess.");
#
#         let mut guess = String::new();
#
#         io::stdin()
#             .read_line(&mut guess)
#             .expect("Failed to read line");
#
        let guess: i32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        if guess < 1 || guess > 100 {
            println!("The secret number will be between 1 and 100.");
            continue;
        }

        match guess.cmp(&secret_number) {
            // --snip--
#             Ordering::Less => println!("Too small!"),
```

```
#             Ordering::Greater => println!("Too big!"),
#             Ordering::Equal => {
#                 println!("You win!");
#                 break;
#             }
#         }
#     }
# }
```

この if 式が、値が範囲外かどうかをチェックし、ユーザに問題を告知し、continue を呼び出してループの次の繰り返しを始め、別の推測を求めます。if 式の後、guess は 1 から 100 の範囲にあると把握して、guess と秘密の数字の比較に進むことができます。

ところが、これは理想的な解決策ではありません: プログラムが 1 から 100 の範囲の値しか処理しないことが間違いなく、肝要であり、この要求がある関数の数が多ければ、このようなチェックを全関数で行うことは、面倒でパフォーマンスにも影響を及ぼす可能性があるでしょう。

代わりに、新しい型を作って検証を関数内に閉じ込め、検証を全箇所で繰り返すのではなく、その型のインスタンスを生成することができます。そうすれば、関数がその新しい型をシグニチャに用い、受け取った値を自信を持って使用することは安全になります。リスト 9-13 に、new 関数が 1 から 100 までの値を受け取った時のみ、Guess のインスタンスを生成する Guess 型を定義する一つの方法を示しました。

```
pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            //     "予想の値は1から100の範囲でなければなりませんが、{}でした。"
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess { value }
    }

    pub fn value(&self) -> i32 {
        self.value
    }
}
```

リスト 9-13: 値が 1 から 100 の場合のみ処理を継続する Guess 型

まず、i32 型の value をフィールドに持つ Guess という名前の構造体を定義しています。ここに数値が保管されます。

それから Guess に Guess 値のインスタンスを生成する new という名前の関連関数を実装していま

す。new 関数は、i32 型の value という引数を取り、Guess を返すように定義されています。new 関数の本体のコードは、value をふるいにかけ、1 から 100 の範囲であることを確かめます。value がふるいに引っかかったら、panic! 呼び出しを行います。これにより、呼び出しコードを書いているプログラマに、修正すべきバグがあると警告します。というのも、この範囲外の value で Guess を生成することは、Guess::new が頼りにしている契約を侵害するからです。Guess::new がパニックするかもしれない条件は、公開されている API ドキュメントで議論されるべきでしょう; あなたが作成する API ドキュメントで panic! の可能性を示唆する、ドキュメントの規約は、第 14 章で講義します。value が確かにふるいを通ったら、value フィールドが value 引数にセットされた新しい Guess を作成して返します。

次に、self を借用し、他に引数はなく、i32 を返す value というメソッドを実装します。この類のメソッドは時に**ゲッター**と呼ばれます。目的がフィールドから何らかのデータを得て返すことだからです。この公開メソッドは、Guess 構造体の value フィールドが非公開なので、必要になります。value フィールドが非公開なことは重要であり、そのために Guess 構造体を使用するコードは、直接 value をセットすることが叶わないのです: モジュール外のコードは、Guess::new 関数を使用して Guess のインスタンスを生成し**なければならず**、それにより、Guess::new 関数の条件式でチェックされていない value が Guess に存在する手段はないことが保証されるわけです。

そうしたら、引数を一つ持つか、1 から 100 の範囲の数値のみを返す関数は、シグニチャで i32 ではなく、Guess を取るか返し、本体内で追加の確認を行う必要はなくなると宣言できるでしょう。

9.4　まとめ

Rust のエラー処理機能は、プログラマがより頑健なコードを書く手助けをするように設計されています。panic! マクロは、プログラムが処理できない状態にあり、無効だったり不正な値で処理を継続するのではなく、プロセスに処理を中止するよう指示することを通知します。Result enum は、Rust の型システムを使用して、コードが回復可能な方法で処理が失敗するかもしれないことを示唆します。Result を使用して、呼び出し側のコードに成功や失敗する可能性を処理する必要があることも教えます。適切な場面で panic! や Result を使用することで、必然的な問題の眼前でコードの信頼性を上げてくれます。

今や、標準ライブラリが Option や Result enum などでジェネリクスを有効活用するところを目の当たりにしたので、ジェネリクスの動作法と自分のコードでの使用方法について語りましょう。

第10章

ジェネリック型、トレイト、ライフタイム

全てのプログラミング言語には、概念の重複を効率的に扱う道具があります。Rust において、そのような道具の一つが**ジェネリクス**です。ジェネリクスは、具体型や他のプロパティの抽象的な代役です。コンパイルやコード実行時に、ジェネリクスの位置に何が入るかを知ることなく、ジェネリクスの振る舞いや他のジェネリクスとの関係を表現できるのです。

関数が未知の値の引数を取り、同じコードを複数の具体的な値に対して走らせるように、`i32` や `String` などの具体的な型の代わりに何かジェネリックな型の引数を取ることができます。実際、第 6 章で `Option<T>` 、第 8 章で `Vec<T>` と `HashMap<K, V>` 、第 9 章で `Result<T, E>` を既に使用しました。この章では、独自の型、関数、メソッドをジェネリクスとともに定義する方法を探究します！

まず、関数を抽出して、コードの重複を減らす方法を確認しましょう。次に同じテクニックを活用して、引数の型のみが異なる 2 つの関数からジェネリックな関数を生成します。また、ジェネリックな型を構造体や enum 定義で使用する方法も説明します。

それから、トレイトを使用して、ジェネリックな方法で振る舞いを定義する方法を学びます。ジェネリックな型にトレイトを組み合わせることで、ジェネリックな型を、単にあらゆる型に対してではなく、特定の振る舞いのある型のみを受け付けるように制限できます。

最後に、ライフタイムを議論します。ライフタイムとは、コンパイラに参照がお互いにどう関係しているかの情報を与える一種のジェネリクスです。ライフタイムを利用することで、借用された値についての十分な情報をコンパイラに渡すことができ、プログラマの支援がなくてもできる範囲よりもより多くの場面で、参照が有効であることを保証できます。

10.1　関数を抽出することで重複を取り除く

ジェネリクスを使用すると、複数の型を表現するプレースホルダで特定の型を置き換えることで、コードの重複を取り除くことができます。ジェネリクスの記法に飛び込む前にまずは、複数の値を表現するプレースホルダで特定の値を置き換えて関数を抽出することで、ジェネリックな型が関わらない重複を取り除く方法を見ましょう。そして、同じテクニックを適用してジェネリックな関数を抽出するのです！ 関数に抽出できる重複したコードを認識する方法を見てみることで、ジェネリクスを使用できる重複コードも認識できるようになるでしょう。

リスト 10-1 の、リスト内の最大値を求める短いプログラムで始めましょう。

ファイル名: src/main.rs

```
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let mut largest = &number_list[0];

    for number in &number_list {
        if number > largest {
            largest = number;
        }
    }

    //       "最大値は{}です"
    println!("The largest number is {}", largest);
#     assert_eq!(*largest, 100);
}
```

リスト 10-1: 数字のリストから最大値を求める

このコードは、整数のリストを変数 `number_list` に格納し、リストの最初の数への参照を `largest` という変数に配置しています。それからリストの数全部を走査し、現在の数が `largest` に格納された数値よりも大きければ、その変数に代入されている参照を置き換えます。ですが、現在の数値が今まで見た最大値以下であれば、変数は変わらず、コードはリストの次の数値に移っていきます。リストの数値全てを吟味した後、`largest` は最大値を参照しているはずで、今回は 100 になります。

次は 2 つの異なる数値のリストから最大値を発見するタスクが課されました。これを行うために、リスト 10-1 のコードを複製し、プログラムの異なる 2 箇所で同じロジックを使用するという手があります。リスト 10-2 のようにですね。

ファイル名: src/main.rs

```
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];
```

```
    let mut largest = &number_list[0];

    for number in &number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {}", largest);

    let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let mut largest = &number_list[0];

    for number in &number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {}", largest);
}
```

リスト 10-2: 2 つの数値のリストから最大値を探すコード

このコードは動くものの、コードを複製することは退屈ですし、間違いも起きやすいです。また、コードを変更したい時には、複数箇所更新することを覚えておかなくてはなりません。

この重複を排除するには、引数で与えられた整数のどんなリストに対しても処理が行える関数を定義して抽象化しましょう。この解決策によりコードがより明確になり、リストの最大値を探すという概念を抽象的に表現させてくれます。

リスト 10-3 では、最大値を探すコードを `largest` という関数に抽出しています。そして、リスト 10-2 の 2 つのリストから最大値を探すために、この関数を呼んでいます。将来持つことになるかもしれない他のどんな `i32` 値のリストに対しても、この関数を使用することができます。

ファイル名: src/main.rs

```
fn largest(list: &[i32]) -> &i32 {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

```
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {}", result);
#     assert_eq!(*result, 100);

    let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let result = largest(&number_list);
    println!("The largest number is {}", result);
#     assert_eq!(*result, 6000);
}
```

リスト 10-3: 2 つのリストから最大値を探す抽象化されたコード

`largest` 関数には `list` と呼ばれる引数があり、これは、関数に渡す可能性のある、あらゆる `i32` 値の具体的なスライスを示します。結果的に、関数呼び出しの際、コードは渡した特定の値に対して走るのです。

まとめとして、こちらがリスト 10-2 のコードからリスト 10-3 に変更するのに要したステップです:

1. 重複したコードを見分ける。
2. 重複コードを関数本体に抽出し、コードの入力と戻り値を関数シグニチャで指定する。
3. 重複したコードの 2 つの実体を代わりに関数を呼び出すように更新する。

次は、この同じ手順をジェネリクスでも踏んでコードの重複を減らします。関数本体が特定の値ではなく抽象的な `list` に対して処理できたのと同様に、ジェネリクスは抽象的な型に対して処理するコードを可能にしてくれます。

例えば、関数が 2 つあるとしましょう: 1 つは `i32` 値のスライスから最大の要素を探し、1 つは `char` 値のスライスから最大要素を探します。この重複はどう排除するのでしょうか？ 答えを見つけましょう！

10.2　ジェネリックなデータ型

関数シグニチャや構造体などの要素の定義を生成するのにジェネリクスを使用することができ、それはさらに他の多くの具体的なデータ型と使用することもできます。まずは、ジェネリクスで関数、構造体、enum、メソッドを定義する方法を見ましょう。それから、ジェネリクスがコードのパフォーマンスに与える影響を議論します。

10.2.1　関数定義では

ジェネリクスを使用する関数を定義する時、通常、引数や戻り値のデータ型を指定する関数のシグニチャにジェネリクスを配置します。そうすることでコードがより柔軟になり、コードの重複を阻止しつつ、関数の呼び出し元により多くの機能を提供します。

largest 関数を続けます。リスト 10-4 はどちらもスライスから最大値を探す 2 つの関数を示しています。ここから、これらをジェネリクスを使用する単一の関数にまとめます。

ファイル名: src/main.rs

```
fn largest_i32(list: &[i32]) -> &i32 {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn largest_char(list: &[char]) -> &char {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest_i32(&number_list);
    println!("The largest number is {}", result);
#     assert_eq!(*result, 100);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest_char(&char_list);
    println!("The largest char is {}", result);
#     assert_eq!(*result, 'y');
}
```

リスト 10-4: 名前とシグニチャの型のみが異なる 2 つの関数

`largest_i32` 関数は、リスト 10-3 で抽出したスライスから最大の `i32` を探す関数です。`largest_char` 関数は、スライスから最大の `char` を探します。関数本体には同じコードがあるので、単独の関数にジェネリックな型引数を導入してこの重複を排除しましょう。

新しい単一の関数の型を引数にするには、ちょうど関数の値引数のように型引数に名前をつける必要があります。型引数の名前にはどんな識別子も使用できますが、`T` を使用します。というのも、慣習では、Rust の型引数名は短く (しばしばたった 1 文字になります)、Rust の型の命名規則がアッパーキャメルケースだからです。"type" の省略形なので、`T` が多くの Rust プログラマの既定の選択なのです。

関数の本体で引数を使用するとき、コンパイラがその名前の意味を把握できるようにシグニチャでその引数名を宣言しなければなりません。同様に、型引数名を関数シグニチャで使用する際には、使用する前に型引数名を宣言しなければなりません。ジェネリックな `largest` 関数を定義するために、型名宣言を山カッコ (`<>`) 内、関数名と引数リストの間に配置してください。こんな感じに:

```
fn largest<T>(list: &[T]) -> &T {
```

この定義は以下のように解読します: 関数 `largest` は、なんらかの型 `T` に関してジェネリックであると。この関数には `list` という引数が 1 つあり、これは型 `T` の値のスライスです。`largest` 関数は同じ `T` 型の値への参照を返します。

リスト 10-5 は、シグニチャにジェネリックなデータ型を使用して `largest` 関数定義を組み合わせたものを示しています。このリストはさらに、この関数を `i32` 値か `char` 値のどちらかで呼べる方法も表示しています。このコードはまだコンパイルできないことに注意してください。ですが、この章の後ほど修正します。

ファイル名: src/main.rs

```
fn largest<T>(list: &[T]) -> &T {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {}", result);
```

```
    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest(&char_list);
    println!("The largest char is {}", result);
}
```

リスト 10-5: ジェネリックな型引数を使用する `largest` 関数; まだコンパイルできません

直ちにこのコードをコンパイルしたら、以下のようなエラーが出ます:

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0369]: binary operation `>` cannot be applied to type `&T`
(エラー: 2項演算`>`は、型`&T`に適用できません)
 --> src/main.rs:5:17
  |
5 |         if item > largest {
  |            ---- ^ ------- &T
  |            |
  |            &T
  |
help: consider restricting type parameter `T`
(ヘルプ: 型引数`T`を制約することを考慮してください)
  |
1 | fn largest<T: std::cmp::PartialOrd>(list: &[T]) -> &T {
  |             ++++++++++++++++++++++

For more information about this error, try `rustc --explain E0369`.
error: could not compile `chapter10` (bin "chapter10") due to 1 previous error
```

ヘルプメッセージが `std::cmp::PartialOrd` に触れています。これは、**トレイト**です。トレイトについては、次の節で語ります。今のところは、このエラーは、`largest` の本体は、`T` がなりうる可能性のある全ての型に対して動作するとは限らないと主張している、と理解してください。本体で型 `T` の値を比較したいので、値が順序付け可能な型のみしか使用できないのです。比較を可能にするために、標準ライブラリには型に実装できる `std::cmp::PartialOrd` トレイトがあります (このトレイトについて詳しくは付録 C を参照されたし)。ヘルプメッセージの提案に従うことで、`T` として有効な型を `PartialOrd` を実装するもののみに制限することができ、そうすると、標準ライブラリは `i32` と `char` の両方に `PartialOrd` を実装しているので、この例はコンパイルできるようになるでしょう。

10.2.2　構造体定義では

構造体を定義して `<>` 記法で 1 つ以上のフィールドにジェネリックな型引数を使用することもできます。リスト 10-6 は、`Point<T>` 構造体を定義してあらゆる型の `x` と `y` 座標を保持しています。

ファイル名: src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}
```

リスト 10-6: 型 T の x と y 値を保持する Point<T> 構造体

構造体定義でジェネリクスを使用する記法は、関数定義のものと似ています。まず、山カッコ内に型引数の名前を構造体名の直後に宣言します。そして、本来具体的なデータ型を記述する構造体定義の箇所に、ジェネリックな型を使用します。

ジェネリックな型を 1 つだけ使用して Point<T> を定義したので、この定義は、Point<T> 構造体がなんらかの型 T に関して、ジェネリックであると述べていて、その型がなんであれ、x と y のフィールドは**両方**その同じ型になっていることに注意してください。リスト 10-7 のように、異なる型の値のある Point<T> のインスタンスを生成すれば、コードはコンパイルできません。

ファイル名: src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let wont_work = Point { x: 5, y: 4.0 };
}
```

リスト 10-7: どちらも同じジェネリックなデータ型 T なので、x と y というフィールドは同じ型でなければならない

この例で、x に整数値 5 を代入すると、この Point<T> のインスタンスに対するジェネリックな型 T は整数になるとコンパイラに知らせます。それから y に 4.0 を指定する時に、このフィールドは x と同じ型と定義したはずなので、このように型不一致エラーが出ます:

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0308]: mismatched types
(エラー: 型が合いません)
 --> src/main.rs:7:38
  |
7 |     let wont_work = Point { x: 5, y: 4.0 };
```

```
  |                                         ^^^ expected integer, found floating-
      point number

For more information about this error, try `rustc --explain E0308`.
error: could not compile `chapter10` (bin "chapter10") due to 1 previous error
```

x と y が両方ジェネリックだけれども、異なる型になり得る Point 構造体を定義するには、複数のジェネリックな型引数を使用できます。例えば、リスト 10-8 では、Point の定義を変更して、型 T と U に関してジェネリックにし、x が型 T で、y が型 U になります。

ファイル名: src/main.rs

```
struct Point<T, U> {
    x: T,
    y: U,
}

fn main() {
    let both_integer = Point { x: 5, y: 10 };
    let both_float = Point { x: 1.0, y: 4.0 };
    let integer_and_float = Point { x: 5, y: 4.0 };
}
```

リスト 10-8: Point<T, U> は 2 つの型に関してジェネリックなので、x と y は異なる型の値になり得る

これで、示された Point インスタンスは全部使用可能です！ 所望の数だけ定義でジェネリックな型引数を使用できますが、数個以上使用すると、コードが読みづらくなります。コードで多くのジェネリックな型が必要になることに気づいた時は、コードの小分けが必要なサインかもしれません。

10.2.3　enum 定義では

構造体のように、列挙子にジェネリックなデータ型を保持する enum を定義することができます。標準ライブラリが提供している Option<T> enum をもう一度見ましょう。この enum は第 6 章で使用しました:

```
enum Option<T> {
    Some(T),
    None,
}
```

この定義はもう、あなたにとってより道理が通っているはずです。ご覧の通り、Option<T> enum は、型 T に関してジェネリックで 2 つの列挙子のある enum です: その列挙子は、型 T の値を保持する Some と、値を何も保持しない None です。Option<T> enum を使用することで、オプショナルな値があるという抽象的な概念を表現でき、Option<T> はジェネリックなので、オプショナルな値の型に

関わらず、この抽象を使用できます。

enum も複数のジェネリックな型を使用できます。第 9 章で使用した `Result` enum の定義が一例です:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

`Result` enum は 2 つの型 `T` 、`E` に関してジェネリックで、2 つの列挙子があります: 型 `T` の値を保持する `Ok` と、型 `E` の値を保持する `Err` です。この定義により、`Result` enum を、成功する (なんらかの型 `T` の値を返す) か、失敗する (なんらかの型 `E` のエラーを返す) 可能性のある処理がある、あらゆる箇所に使用するのが便利になります。事実、ファイルを開くのに成功した時に `T` に型 `std::fs::File` が入り、ファイルを開く際に問題があった時に `E` に型 `std::io::Error` が入ったものが、リスト 9-3 でファイルを開くのに使用したものです。

自分のコード内で、保持している値の型のみが異なる構造体や enum 定義の場面を認識したら、代わりにジェネリックな型を使用することで重複を避けることができます。

10.2.4　メソッド定義では

(第 5 章のように、) 定義にジェネリックな型を使うメソッドを構造体や enum に実装することもできます。リスト 10-9 は、リスト 10-6 で定義した `Point<T>` 構造体に `x` というメソッドを実装したものを示しています。

ファイル名: src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}

fn main() {
    let p = Point { x: 5, y: 10 };

    println!("p.x = {}", p.x());
}
```

リスト 10-9: 型 `T` の `x` フィールドへの参照を返す `x` というメソッドを `Point<T>` 構造体に実装する

ここで、フィールド x のデータへの参照を返す x というメソッドを Point<T> に定義しました。

impl の直後に T を宣言しなければならないことに注意してください。こうすることで、型 Point<T> にメソッドを実装していることを指定するために、T を使用することができます。impl の後に T をジェネリックな型として宣言することで、コンパイラは、Point の山カッコ内の型が、具体的な型ではなくジェネリックな型であることを認識できるのです。このジェネリック引数には、構造体定義内で宣言したジェネリック引数とは異なる名前を選択することもできますが、同じ名前を使用するのが慣用的です。ジェネリック型を宣言する impl 内に書かれたメソッドは、そのジェネリック型を置き換えることになる具体型が何であっても、その型のインスタンスの任意の値に対して定義されます。

型にメソッドを定義するときに、ジェネリック型に制約を指定することもできます。例えば、ジェネリックな型を持つ Point<T> インスタンスではなく、Point<f32> だけにメソッドを実装することもできるでしょう。リスト 10-10 では、具体的な型 f32 を使用しています。つまり、impl の後に型を宣言しません。

ファイル名: src/main.rs

```
# struct Point<T> {
#     x: T,
#     y: T,
# }
#
# impl<T> Point<T> {
#     fn x(&self) -> &T {
#         &self.x
#     }
# }
#
impl Point<f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}
#
# fn main() {
#     let p = Point { x: 5, y: 10 };
#
#     println!("p.x = {}", p.x());
# }
```

リスト 10-10: ジェネリックな型引数 T に対して特定の具体的な型がある構造体にのみ適用される impl ブロック

このコードは、Point<f32> には distance_from_origin メソッドが存在するが、T が f32 ではない Point<T> の他のインスタンスにはこのメソッドが定義されないことを意味します。このメソッドは、この点が座標 (0.0, 0.0) の点からどれだけ離れているかを測定し、浮動小数点数にのみ利用可能な数学的処理を使用します。

構造体定義のジェネリックな型引数は、必ずしもその構造体のメソッドシグニチャで使用するものと同じにはなりません。リスト 10-11 では、例をより見通しよくするために、Point 構造体に対してはジェネリック型 X1 と Y1 を使用し、mixup メソッドのシグネチャに対しては X2 と Y2 を使用しています。このメソッドは、(型 X1 の) self の Point の x 値と渡した (型 Y2 の) Point の y 値から新しい Point インスタンスを生成します。

ファイル名: src/main.rs

```
struct Point<X1, Y1> {
    x: X1,
    y: Y1,
}

impl<X1, Y1> Point<X1, Y1> {
    fn mixup<X2, Y2>(self, other: Point<X2, Y2>) -> Point<X1, Y2> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}

fn main() {
    let p1 = Point { x: 5, y: 10.4 };
    let p2 = Point { x: "Hello", y: 'c' };

    let p3 = p1.mixup(p2);

    println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
}
```

リスト 10-11: 構造体定義とは異なるジェネリックな型を使用するメソッド

main で、x (値は 5) に i32 、y (値は 10.4) に f64 を持つ Point を定義しました。p2 変数は、x (値は "Hello") に文字列スライス、y (値は c) に char を持つ Point 構造体です。引数 p2 で p1 に mixup を呼び出すと、p3 が得られ、x は i32 になります。x は p1 由来だからです。p3 変数の y は、char になります。y は p2 由来だからです。println! マクロの呼び出しは、p3.x = 5, p3.y = c と出力するでしょう。

この例の目的は、一部のジェネリックな引数は impl で宣言され、他の一部はメソッド定義で宣言される場面をデモすることです。ここで、ジェネリックな引数 X1 と Y1 は impl の後に宣言されています。構造体定義にはまるからです。ジェネリックな引数 X2 と Y2 は fn mixup の後に宣言されています。何故なら、このメソッドにしか関係ないからです。

10.2.5　ジェネリクスを使用したコードのパフォーマンス

ジェネリックな型引数を使用すると、実行時にコストが発生するのかな、と思うかもしれません。嬉しいことに、ジェネリックな型を使用したからといって、具体的な型を使って同じことを行う場合よりもプログラムの実行が遅くなることはありません。

コンパイラはこれを、ジェネリクスを使用しているコードの単相化をコンパイル時に行うことで達成しています。**単相化 (monomorphization)** は、コンパイル時に使用されている具体的な型を入れることで、ジェネリックなコードを特定のコードに変換する過程のことです。この過程において、コンパイラは、リスト 10-5 でジェネリックな関数を生成するために使用した手順と真逆のことをしています: コンパイラは、ジェネリックなコードが呼び出されている箇所全部を見て、ジェネリックなコードが呼び出されている具体的な型のコードを生成するのです。

標準ライブラリのジェネリックな `Option<T>` enum を使用して、これがどう機能するのか見てみましょう:

```
let integer = Some(5);
let float = Some(5.0);
```

コンパイラがこのコードをコンパイルすると、単相化を行います。その過程で、コンパイラは `Option<T>` のインスタンスに使用された値を読み取り、2 種類の `Option<T>` を識別します: 一方は `i32` で、もう片方は `f64` です。そのように、コンパイラは、`Option<T>` のジェネリックな定義を `i32` と `f64` に特化した 2 つの定義に展開し、それにより、ジェネリックな定義を特定の定義と置き換えます。

単相化されたバージョンのコードは、以下のようなものになります (コンパイラは、ここで説明のために使用しているものとは異なる名前を使用します):

ファイル名: src/main.rs

```
enum Option_i32 {
    Some(i32),
    None,
}

enum Option_f64 {
    Some(f64),
    None,
}

fn main() {
    let integer = Option_i32::Some(5);
    let float = Option_f64::Some(5.0);
}
```

ジェネリックな `Option<T>` は、コンパイラによって生成される特化された定義で置き換えられま

す。Rust では、ジェネリックなコードを各インスタンスで型を指定したコードにコンパイルするので、ジェネリクスを使用することに対して実行時コストを払うことはありません。コードを実行すると、それぞれの定義を手作業で複製した時のように振る舞います。単相化の過程により、Rust のジェネリクスは実行時に究極的に効率的になるのです。

10.3　トレイト: 共通の振る舞いを定義する

トレイトは、特定の型に存在し、他の型と共有できる機能を定義します。トレイトを使用すると、共通の振る舞いを抽象的に定義できます。**トレイト境界**を使用すると、あるジェネリック型が、特定の振る舞いをもつあらゆる型になり得ることを指定できます。

> 注釈: 違いはあるものの、トレイトは他の言語でよくインターフェイスと呼ばれる機能に類似しています。

10.3.1　トレイトを定義する

型の振る舞いは、その型に対して呼び出せるメソッドから構成されます。異なる型は、それらの型全てに対して同じメソッドを呼び出せるなら、同じ振る舞いを共有することになります。トレイト定義は、メソッドシグニチャをあるグループにまとめ、なんらかの目的を達成するのに必要な一連の振る舞いを定義する手段です。

例えば、いろんな種類や量のテキストを保持する複数の構造体があるとしましょう: 特定の場所から送られる新しいニュースを保持する `NewsArticle` と、新規ツイートか、リツイートか、はたまた他のツイートへのリプライなのかを示すメタデータを伴う最大で 280 文字までの `Tweet` です。

`NewsArticle` または `Tweet` インスタンスに保存されているデータのサマリーを表示できる `aggregator` という名前のメディアアグリゲータライブラリクレートを作成します。これをするには、各型のサマリーが必要で、インスタンスで `summarize` メソッドを呼び出してサマリーを要求することになるでしょう。リスト 10-12 は、この振る舞いを表現する公開の `Summary` トレイトの定義を表示しています。

ファイル名: src/lib.rs

```
pub trait Summary {
    fn summarize(&self) -> String;
}
```

リスト 10-12: `summarize` メソッドで提供される振る舞いからなる `Summary` トレイト

ここでは、`trait` キーワード、それからトレイト名を使用してトレイトを宣言していて、その名前は今回の場合、`Summary` です。また、いくつかの例で見ていきますが、このクレートに依存する

クレートがこのトレイトを利用できるように、トレイトを pub として宣言しています。波括弧の中にこのトレイトを実装する型の振る舞いを記述するメソッドシグニチャを定義し、今回の場合は、fn summarize(&self) -> String です。

メソッドシグニチャの後に、波括弧内に実装を提供する代わりに、セミコロンを使用しています。このトレイトを実装する型はそれぞれ、メソッドの本体に独自の振る舞いを提供しなければなりません。コンパイラにより、Summary トレイトを保持するあらゆる型に、このシグニチャと全く同じメソッド summarize が定義されていることが強制されます。

トレイトには、本体に複数のメソッドを含むことができます: メソッドシグニチャは行ごとに並べられ、各行はセミコロンで終わります。

10.3.2　トレイトを型に実装する

これで Summary トレイトのメソッドのシグネチャを希望通りに定義できたので、メディアアグリゲータ内の型に対してこれを実装できます。リスト 10-13 は、Summary トレイトを NewsArticle 構造体上に実装したもので、ヘッドライン、著者、そして地域情報を使って summarize の戻り値を作っています。Tweet 構造体に関しては、ツイートの内容が既に 280 文字に制限されていると仮定して、ユーザー名の後にツイートのテキスト全体が続くものとして summarize を定義します。

ファイル名: src/lib.rs

```
# pub trait Summary {
#     fn summarize(&self) -> String;
# }
#
pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}, by {} ({})", self.headline, self.author, self.location)
    }
}

pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
```

```
        format!("{}: {}", self.username, self.content)
    }
}
```

リスト 10-13: Summary トレイトを NewsArticle と Tweet 型に実装する

型にトレイトを実装することは、普通のメソッドを実装することに似ています。違いは、impl の後に、実装したいトレイトの名前を置き、それから for キーワード、さらにトレイトの実装対象の型の名前を指定することです。impl ブロック内に、トレイト定義で定義したメソッドシグニチャを置きます。各シグニチャの後にセミコロンを追記するのではなく、波括弧を使用し、メソッド本体に特定の型のトレイトのメソッドに欲しい特定の振る舞いを入れます。

これでライブラリは NewsArticle と Tweet に対して Summary トレイトを実装できたので、クレートの利用者は普通のメソッド同様に NewsArticle や Tweet のインスタンスに対してこのトレイトメソッドを呼び出せます。唯一の違いは、ユーザは型だけではなくトレイトもスコープ内に持ち込まなくてはならないということです。以下は、バイナリクレートが私たちの aggregator ライブラリクレートをどうやって使用できるかの例です:

```
use aggregator::{Summary, Tweet};

fn main() {
    let tweet = Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            // もちろん、ご存知かもしれませんがね、みなさん
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    };

    println!("1 new tweet: {}", tweet.summarize());
}
```

このコードは、1 new tweet: horse_ebooks: of course, as you probably already know, people と出力します。

aggregator クレートに依存する他のクレートも、自身の型に対して Summary を実装するために、Summary トレイトをスコープに持ち込むことができます。注意すべき制限の 1 つは、トレイトか、対象の型のうち、少なくとも一方が自分のクレートに固有 (local) である時のみ、型に対してトレイトを実装できるということです。例えば、Display のような標準ライブラリのトレイトを aggregator クレートの機能の一部として、Tweet のような独自の型に実装できます。型 Tweet が aggregator クレートに固有だからです。また、Summary を aggregator クレートで Vec<T> に対して実装することもできます。トレイト Summary は、aggregator クレートに固有だからです。

しかし、外部のトレイトを外部の型に対して実装することはできません。例として、aggregator ク

レート内で Vec<T> に対して Display トレイトを実装することはできません。Display と Vec<T> はどちらも標準ライブラリで定義され、aggregator クレートに固有ではないからです。この制限は、**コヒーレンス** (coherence)、特に**孤児のルール** (orphan rule) と呼ばれる特性の一部で、親の型が存在しないためにそう命名されました。この規則により、他の人のコードが自分のコードを壊したり、その逆が起きないことを保証してくれます。この規則がなければ、2 つのクレートが同じ型に対して同じトレイトを実装できてしまい、コンパイラはどちらの実装を使うべきかわからなくなってしまうでしょう。

10.3.3　デフォルト実装

時として、全ての型の全メソッドに対して実装を要求するのではなく、トレイトの全てあるいは一部のメソッドに対してデフォルトの振る舞いがあると有用です。そうすれば、特定の型にトレイトを実装する際、各メソッドのデフォルト実装を保持するかオーバーライドするか選べるわけです。

リスト 10-14 では、リスト 10-12 のように、メソッドシグニチャだけを定義するのではなく、Summary トレイトの summarize メソッドにデフォルトの文字列を指定しています。

ファイル名: src/lib.rs

```
pub trait Summary {
    fn summarize(&self) -> String {
        // "（もっと読む）"
        String::from("(Read more...)")
    }
}
#
# pub struct NewsArticle {
#     pub headline: String,
#     pub location: String,
#     pub author: String,
#     pub content: String,
# }
#
# impl Summary for NewsArticle {}
#
# pub struct Tweet {
#     pub username: String,
#     pub content: String,
#     pub reply: bool,
#     pub retweet: bool,
# }
#
# impl Summary for Tweet {
#     fn summarize(&self) -> String {
#         format!("{}: {}", self.username, self.content)
#     }
# }
```

リスト 10-14: summarize メソッドのデフォルト実装がある Summary トレイトを定義する

デフォルト実装を利用して NewsArticle のインスタンスをまとめるには、impl Summary for NewsArticle {} と空の impl ブロックを指定します。

もはや NewsArticle に直接 summarize メソッドを定義してはいませんが、私達はデフォルト実装を提供しており、NewsArticle は Summary トレイトを実装すると指定しました。そのため、NewsArticle のインスタンスに対して summarize メソッドを同じように呼び出すことができます。このように:

```
# use aggregator::{self, NewsArticle, Summary};
#
# fn main() {
    let article = NewsArticle {
        // ペンギンチームがスタンレーカップチャンピオンシップを勝ち取る!
        headline: String::from("Penguins win the Stanley Cup Championship!"),
        // アメリカ、ペンシルベニア州、ピッツバーグ
        location: String::from("Pittsburgh, PA, USA"),
        // アイスバーグ
        author: String::from("Iceburgh"),
        // ピッツバーグ・ペンギンが再度NHL(National Hockey League)で最強のホッ
    ケーチームになった
        content: String::from(
            "The Pittsburgh Penguins once again are the best \
             hockey team in the NHL.",
        ),
    };

    println!("New article available! {}", article.summarize());
# }
```

このコードは、New article available! (Read more...)（新しい記事があります!（もっと読む））と出力します。

デフォルト実装を用意しても、リスト 10-13 の Tweet の Summary 実装を変える必要はありません。理由は、デフォルト実装をオーバーライドする記法はデフォルト実装のないトレイトメソッドを実装する記法と同じだからです。

デフォルト実装は、自らのトレイトのデフォルト実装を持たない他のメソッドを呼び出すことができます。このようにすれば、トレイトは多くの有用な機能を提供しつつ、実装者は僅かな部分しか指定しなくて済むようになります。例えば、Summary トレイトを、（実装者が）内容を実装しなければならない summarize_author メソッドを持つように定義し、それから summarize_author メソッドを呼び出すデフォルト実装を持つ summarize メソッドを定義することもできます:

```
pub trait Summary {
    fn summarize_author(&self) -> String;

    fn summarize(&self) -> String {
        // "（{}さんの文章をもっと読む）"
        format!("(Read more from {}...)", self.summarize_author())
```

```
    }
}
#
# pub struct Tweet {
#     pub username: String,
#     pub content: String,
#     pub reply: bool,
#     pub retweet: bool,
# }
#
# impl Summary for Tweet {
#     fn summarize_author(&self) -> String {
#         format!("@{}", self.username)
#     }
# }
```

このバージョンの Summary を使用するために、型にトレイトを実装する際、実装する必要があるのは summarize_author だけです:

```
# pub trait Summary {
#     fn summarize_author(&self) -> String;
#
#     fn summarize(&self) -> String {
#         // "（{}さんの文章をもっと読む）"
#         format!("(Read more from {}...)", self.summarize_author())
#     }
# }
#
# pub struct Tweet {
#     pub username: String,
#     pub content: String,
#     pub reply: bool,
#     pub retweet: bool,
# }
#
impl Summary for Tweet {
    fn summarize_author(&self) -> String {
        format!("@{}", self.username)
    }
}
```

summarize_author 定義後、Tweet 構造体のインスタンスに対して summarize を呼び出せ、summarize のデフォルト実装は、私達が提供した summarize_author の定義を呼び出すでしょう。summarize_author を実装したので、追加のコードを書く必要なく、Summary トレイトは、summarize メソッドの振る舞いを与えてくれました。

```
# use aggregator::{self, Summary, Tweet};
#
# fn main() {
    let tweet = Tweet {
```

```
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    };

    println!("1 new tweet: {}", tweet.summarize());
# }
```

このコードは、`1 new tweet: (Read more from @horse_ebooks...)`（1つの新しいツイート：(@horse_ebooksさんの文章をもっと読む)）と出力します。

デフォルト実装を、そのメソッドをオーバーライドしている実装から呼び出すことはできないことに注意してください。

10.3.4　引数としてのトレイト

トレイトを定義し実装する方法はわかったので、トレイトを使っていろんな種類の型を受け付ける関数を定義する方法を学んでいきましょう。リスト 10-13 で `NewsArticle` と `Tweet` に対して実装した `Summary` トレイトを使用して、`notify` 関数を定義しましょう。この関数は、`Summary` トレイトを実装する何らかの型を持つ引数 `item` を持ち、それに対して `summarize` メソッドを呼び出します。これを行うためには、このように `impl Trait` 構文を使います:

```
# pub trait Summary {
#     fn summarize(&self) -> String;
# }
#
# pub struct NewsArticle {
#     pub headline: String,
#     pub location: String,
#     pub author: String,
#     pub content: String,
# }
#
# impl Summary for NewsArticle {
#     fn summarize(&self) -> String {
#         format!("{}, by {} ({})", self.headline, self.author, self.location)
#     }
# }
#
# pub struct Tweet {
#     pub username: String,
#     pub content: String,
#     pub reply: bool,
#     pub retweet: bool,
# }
#
```

```
# impl Summary for Tweet {
#     fn summarize(&self) -> String {
#         format!("{}: {}", self.username, self.content)
#     }
# }
#
pub fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}
```

引数の item には、具体的な型の代わりに、impl キーワードとトレイト名を指定します。この引数は、指定されたトレイトを実装しているあらゆる型を受け付けます。notify の中身では、summarize のような、Summary トレイトに由来する item のあらゆるメソッドを呼び出すことができます。私達は、notify を呼びだし、NewsArticle か Tweet のどんなインスタンスでも渡すことができます。この関数を呼び出すときに、String や i32 のような他の型を渡すようなコードはコンパイルできません。なぜなら、これらの型は Summary を実装していないからです。

10.3.4.1　トレイト境界構文

impl Trait 構文は単純なケースを解決しますが、実はより長い**トレイト境界 (trait bound)** として知られる姿の糖衣構文 (syntax sugar) なのです。それは以下のようなものです：

```
pub fn notify<T: Summary>(item: &T) {
    // 速報！  {}
    println!("Breaking news! {}", item.summarize());
}
```

この「より長い」姿は前節の例と等価ですが、より冗長です。山カッコの中にジェネリックな型引数の宣言を書き、型引数の後ろにコロンを挟んでトレイト境界を置いています。

簡単なケースでは impl Trait 構文は便利で、コードを簡潔にしてくれます。一方でそうでないケースでは、完全なトレイト境界構文を使えばより複雑な制約を表現できます。たとえば、Summary を実装する 2 つのパラメータを持つような関数を考えることができます。impl Trait 構文を使ってそうするのはこのようになるでしょう：

```
pub fn notify(item1: &impl Summary, item2: &impl Summary) {
```

この関数が受け取る item1 と item2 の型が（どちらも Summary を実装する限り）異なっても良い場合、impl Trait の使用は適切です。しかし、両方の引数が同じ型であることを強制したい場合は、次のようにトレイト境界を使用しなくてはなりません:

```
pub fn notify<T: Summary>(item1: &T, item2: &T) {
```

引数である item1 と item2 の型としてジェネリックな型 T を指定しました。これにより、item1 と

item2 として関数に渡される値の具体的な型が同一でなければならない、という制約を与えています。

10.3.4.2　複数のトレイト境界を + 構文で指定する

複数のトレイト境界も指定できます。たとえば、notify に、item に対する summarize に加えて画面出力形式（ディスプレイフォーマット）も使わせたいとします。その場合は、notify の定義に item は Display と Summary の両方を実装していなくてはならないと指定することになります。これは、以下のように + 構文で行うことができます：

```
pub fn notify(item: &(impl Summary + Display)) {
```

+ 構文はジェネリック型につけたトレイト境界に対しても使えます：

```
pub fn notify<T: Summary + Display>(item: &T) {
```

これら 2 つのトレイト境界が指定されていれば、notify の中では summarize を呼び出すことと、{} を使って item をフォーマットすることの両方が行なえます。

10.3.4.3　where 句を使ったより明確なトレイト境界

あまりたくさんのトレイト境界を使うことには欠点もあります。それぞれのジェネリック（な型）がそれぞれのトレイト境界をもつので、複数のジェネリック型の引数をもつ関数は、関数名と引数リストの間に大量のトレイト境界に関する情報を含むことがあります。これでは関数のシグネチャが読みにくくなってしまいます。このため、Rust はトレイト境界を関数シグネチャの後の where 句の中で指定するという別の構文を用意しています。なので、このように書く：

```
fn some_function<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) -> i32 {
```

代わりに、where 句を使い、このように書くことができます：

```
fn some_function<T, U>(t: &T, u: &U) -> i32
where
    T: Display + Clone,
    U: Clone + Debug,
{
#     unimplemented!()
# }
```

この関数シグニチャは、よりさっぱりとしています。トレイト境界を多く持たない関数と同じように、関数名、引数リスト、戻り値の型が一緒になって近くにあるからですね。

10.3.5　トレイトを実装している型を返す

以下のように、`impl Trait` 構文を戻り値型のところで使うことにより、あるトレイトを実装する何らかの型を返すことができます。

```
# pub trait Summary {
#     fn summarize(&self) -> String;
# }
#
# pub struct NewsArticle {
#     pub headline: String,
#     pub location: String,
#     pub author: String,
#     pub content: String,
# }
#
# impl Summary for NewsArticle {
#     fn summarize(&self) -> String {
#         format!("{}, by {} ({})", self.headline, self.author, self.location)
#     }
# }
#
# pub struct Tweet {
#     pub username: String,
#     pub content: String,
#     pub reply: bool,
#     pub retweet: bool,
# }
#
# impl Summary for Tweet {
#     fn summarize(&self) -> String {
#         format!("{}: {}", self.username, self.content)
#     }
# }
#
fn returns_summarizable() -> impl Summary {
    Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    }
}
```

戻り値の型として `impl Summary` を使うことにより、具体的な型が何かを言うことなく、`returns_summarizable` 関数は `Summary` トレイトを実装している何らかの型を返すのだ、と指定することができます。今回 `returns_summarizable` は `Tweet` を返しますが、この関数を呼び出すコードは

そのことを知る必要はありません。

実装しているトレイトだけで戻り値型を指定できることは、13 章で学ぶ、クロージャとイテレータを扱うときに特に便利です。クロージャとイテレータの作り出す型は、コンパイラだけが知っているものであったり、指定するには長すぎるものであったりします。`impl Trait` 構文を使えば、非常に長い型を書くことなく、ある関数は `Iterator` トレイトを実装するある型を返すのだ、と簡潔に指定することができます。

ただし、`impl Trait` は一種類の型を返す場合にのみ使えます。たとえば、以下のように、戻り値の型は `impl Summary` で指定しつつ、`NewsArticle` か `Tweet` を返すようなコードは失敗します：

```
# pub trait Summary {
#     fn summarize(&self) -> String;
# }
#
# pub struct NewsArticle {
#     pub headline: String,
#     pub location: String,
#     pub author: String,
#     pub content: String,
# }
#
# impl Summary for NewsArticle {
#     fn summarize(&self) -> String {
#         format!("{}, by {} ({})", self.headline, self.author, self.location)
#     }
# }
#
# pub struct Tweet {
#     pub username: String,
#     pub content: String,
#     pub reply: bool,
#     pub retweet: bool,
# }
#
# impl Summary for Tweet {
#     fn summarize(&self) -> String {
#         format!("{}: {}", self.username, self.content)
#     }
# }
#
fn returns_summarizable(switch: bool) -> impl Summary {
    if switch {
        NewsArticle {
            headline: String::from(
                "Penguins win the Stanley Cup Championship!",
            ),
            location: String::from("Pittsburgh, PA, USA"),
            author: String::from("Iceburgh"),
            content: String::from(
                "The Pittsburgh Penguins once again are the best \
                 hockey team in the NHL.",
```

```
            ),
        }
    } else {
        Tweet {
            username: String::from("horse_ebooks"),
            content: String::from(
                "of course, as you probably already know, people",
            ),
            reply: false,
            retweet: false,
        }
    }
}
```

NewsArticle か Tweet を返すというのは、コンパイラの impl Trait 構文の実装まわりの制約により許されていません。このような振る舞いをする関数を書く方法は、17 章の「トレイトオブジェクトで異なる型の値を許容する」節で学びます。

10.3.6　トレイト境界を使用して、メソッド実装を条件分けする

ジェネリックな型引数を持つ impl ブロックにトレイト境界を与えることで、特定のトレイトを実装する型に対するメソッド実装を条件分けできます。例えば、リスト 10-15 の型 Pair<T> は、Pair<T> の新しいインスタンスを返す new 関数を常に実装します（第 5 章の「メソッドを定義する」節で学んだ、Self は impl ブロックの型に対する型エイリアスだということを思い出してください、今回は Pair<T> です）。しかし次の impl ブロック内では、Pair<T> は、内部の型 T が比較を可能にする PartialOrd トレイトと出力を可能にする Display トレイトを実装している時のみ、cmp_display メソッドを実装します。

ファイル名: src/lib.rs

```
use std::fmt::Display;

struct Pair<T> {
    x: T,
    y: T,
}

impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self { x, y }
    }
}

impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("The largest member is x = {}", self.x);
```

```
        } else {
            println!("The largest member is y = {}", self.y);
        }
    }
}
```

リスト 10-15: トレイト境界によってジェネリックな型に対するメソッド実装を条件分けする

また、別のトレイトを実装するあらゆる型に対するトレイト実装を条件分けすることもできます。トレイト境界を満たすあらゆる型にトレイトを実装することは、**ブランケット実装** (blanket implementation) と呼ばれ、Rust の標準ライブラリで広く使用されています。例を挙げれば、標準ライブラリは、`Display` トレイトを実装するあらゆる型に `ToString` トレイトを実装しています。標準ライブラリの `impl` ブロックは以下のような見た目です:

```
impl<T: Display> ToString for T {
    // --snip--
}
```

標準ライブラリにはこのブランケット実装があるので、`Display` トレイトを実装する任意の型に対して、`ToString` トレイトで定義された `to_string` メソッドを呼び出せるのです。例えば、整数は `Display` を実装するので、このように整数値を対応する `String` 値に変換できます:

```
let s = 3.to_string();
```

ブランケット実装は、トレイトのドキュメンテーションの「実装したもの」節に出現します。

トレイトとトレイト境界により、ジェネリックな型引数を使用して重複を減らしつつ、コンパイラに対して、そのジェネリックな型に特定の振る舞いが欲しいことを指定するコードを書くことができます。それからコンパイラは、トレイト境界の情報を活用してコードに使用された具体的な型が正しい振る舞いを提供しているか確認できます。動的型付き言語では、その型に定義されていないメソッドを呼び出せば、実行時 (runtime) にエラーが出るでしょう。しかし、Rust はこの種のエラーをコンパイル時に移したので、コードが動かせるようになる以前に問題を修正することを強制されるのです。加えて、コンパイル時に既に確認したので、実行時の振る舞いを確認するコードを書かなくても済みます。そうすることで、ジェネリクスの柔軟性を諦めることなくパフォーマンスを向上させます。

10.4　ライフタイムで参照を検証する

ライフタイムは、すでに使ってきましたが、もう一つの種類のジェネリクスです。ライフタイムは、型が欲しい振る舞いを持つことを保証するのではなく、必要な間だけ参照が有効であることを保証します。

第 4 章の「参照と借用」節で議論しなかった詳細の一つに、Rust において参照は全てライフタイムを保持するということがあります。ライフタイムとは、その参照が有効になるスコープのことです。

多くの場合、型が推論されるように、大体の場合、ライフタイムも暗黙的に推論されます。複数の型の可能性があるときにのみ、型を注釈しなければなりません。同様に、参照のライフタイムがいくつか異なる方法で関係することがある場合には注釈しなければなりません。コンパイラは、ジェネリックライフタイム引数を使用して関係を注釈し、実行時に実際の参照が確かに有効であることを保証することを要求するのです。

ライフタイムの注釈は、他の多くのプログラミング言語にはない概念ですので、馴染みのない気分になるでしょう。この章では、ライフタイムの全体を解説することはしませんが、ライフタイム記法に遭遇するよくある場合について議論しますので、ライフタイムの概念について馴染むことができるでしょう。

10.4.1　ライフタイムでダングリング参照を回避する

ライフタイムの主な目的は、**ダングリング参照**を回避することです。ダングリング参照によりプログラムは、参照するつもりだったデータ以外のデータを参照してしまいます。リスト 10-17 のプログラムを考えてください。これには、外側のスコープと内側のスコープが含まれています。

```
fn main() {
    let r;

    {
        let x = 5;
        r = &x;
    }

    println!("r: {}", r);
}
```

リスト 10-16: 値がスコープを抜けてしまった参照を使用しようとする

> 注釈: リスト 10-16 や 10-17、10-23 では、変数に初期値を与えずに宣言しているので、変数名は外側のスコープに存在します。初見では、これは Rust には null 値が存在しないということと衝突しているように見えるかもしれません。しかしながら、値を与える前に変数を使用しようとすれば、コンパイルエラーになり、確かに Rust では null 値は許可されていないことがわかります。

外側のスコープで初期値なしの `r` という変数を宣言し、内側のスコープで初期値 5 の `x` という変数を宣言しています。内側のスコープ内で、`r` の値を `x` への参照にセットしようとしています。それから内側のスコープが終わり、`r` の値を出力しようとしています。`r` が参照している値が使おうとする前にスコープを抜けるので、このコードはコンパイルできません。こちらがエラーメッセージです:

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
```

```
error[E0597]: `x` does not live long enough
(エラー[E0597]: `x`の生存期間が短すぎます)
 --> src/main.rs:6:13
  |
5 |         let x = 5;
  |             - binding `x` declared here
  |               (束縛`x`はここで宣言されています)
6 |         r = &x;
  |             ^^ borrowed value does not live long enough
  |               (借用された値の生存期間が短すぎます)
7 |     }
  |     - `x` dropped here while still borrowed
  |      (`x`は借用されている間にここでドロップされました)
8 |
9 |     println!("r: {}", r);
  |                        - borrow later used here
  |                         (その後、借用はここで使われています)

For more information about this error, try `rustc --explain E0597`.
error: could not compile `chapter10` (bin "chapter10") due to 1 previous error
```

変数 x の「生存期間が短すぎます」。原因は、内側のスコープが 7 行目で終わった時点で x がスコープを抜けるからです。ですが、r はまだ、外側のスコープに対して有効です; スコープが大きいので、「長生きする」と言います。Rust で、このコードが動くことを許可していたら、r は x がスコープを抜けた時に解放されるメモリを参照していることになり、r で行おうとするいかなることもちゃんと動作しないでしょう。では、どうやってコンパイラはこのコードが無効であると決定しているのでしょうか？ それは、借用チェッカーを使用しているのです。

10.4.2　借用精査機

Rust コンパイラには、スコープを比較して全ての借用が有効であるかを決定する**借用チェッカー**があります。リスト 10-17 は、リスト 10-16 と同じコードを示していますが、変数のライフタイムを表示する注釈が付いています。

```
fn main() {
    let r;                // ---------+-- 'a
                          //          |
    {                     //          |
        let x = 5;        // -+-- 'b  |
        r = &x;           //  |       |
    }                     // -+       |
                          //          |
    println!("r: {}", r); //          |
}                         // ---------+
```

リスト 10-17: それぞれ 'a と 'b と名付けられた r と x のライフタイムの注釈

ここで、r のライフタイムは 'a 、x のライフタイムは 'b で注釈しました。ご覧の通り、内側の 'b ブロックの方が、外側の 'a ライフタイムブロックよりはるかに小さいです。コンパイル時に、コンパイラは 2 つのライフタイムのサイズを比較し、r は 'a のライフタイムだけれども、'b のライフタイムのメモリを参照していると確認します。'b は 'a よりも短いので、プログラムは拒否されます: 参照の対象が参照ほど長生きしないのです。

リスト 10-18 でコードを修正したので、ダングリング参照はなくなり、エラーなくコンパイルできます。

```
fn main() {
    let x = 5;            // ----------+-- 'b
                          //           |
    let r = &x;           // --+-- 'a  |
                          //   |       |
    println!("r: {}", r); //   |       |
                          // --+       |
}                         // ----------+
```

リスト 10-18: データのライフタイムが参照より長いので、有効な参照

ここで x のライフタイムは 'b になり、今回の場合 'a よりも大きいです。つまり、コンパイラは x が有効な間、r の参照も常に有効になることを把握しているので、r は x を参照できます。

今や、参照のライフタイムがどれだけであるかと、コンパイラがライフタイムを解析して参照が常に有効であることを保証する仕組みがわかったので、関数における引数と戻り値のジェネリックなライフタイムを探究しましょう。

10.4.3　関数のジェネリックなライフタイム

2 つの文字列スライスのうち、長い方を返す関数を書きましょう。この関数は、2 つの文字列スライスを取り、1 つの文字列スライスを返します。longest 関数の実装完了後、リスト 10-19 のコードは、The longest string is abcd と出力するはずです。

ファイル名: src/main.rs

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    // 最長の文字列は、{}です
    println!("The longest string is {}", result);
}
```

リスト 10-19: longest 関数を呼び出して 2 つの文字列スライスのうち長い方を探す main 関数

関数に取ってほしい引数が文字列ではなく文字列スライス、つまり参照であることに注意してください。何故なら、longest 関数に引数の所有権を奪ってほしくないからです。リスト 10-19 で使用している引数が、我々が必要としているものである理由についてもっと詳しい議論は、第 4 章の「引数としての文字列スライス」節をご参照ください。

リスト 10-20 に示すように longest 関数を実装しようとしたら、コンパイルできないでしょう。

ファイル名: src/main.rs

```
# fn main() {
#     let string1 = String::from("abcd");
#     let string2 = "xyz";
#
#     let result = longest(string1.as_str(), string2);
#     println!("The longest string is {}", result);
# }
#
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

リスト 10-20: 2 つの文字列スライスのうち長い方を返すけれども、コンパイルできない longest 関数の実装

代わりに、以下のようなライフタイムに言及するエラーが出ます:

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0106]: missing lifetime specifier
(エラー[E0106]: ライフタイム指定子が不足しています)
 --> src/main.rs:9:33
  |
9 | fn longest(x: &str, y: &str) -> &str {
  |               ----     ----     ^ expected named lifetime parameter
  |                                  (名前付きライフタイム引数があるべきです)
  |
  = help: this function's return type contains a borrowed value, but the
      signature does not say whether it is borrowed from `x` or `y`
   (ヘルプ: この関数の戻り値型は借用された値を含んでいますが、シグネチャは、それ
       が`x`と`y`どちらから借用されたものなのか宣言していません)
help: consider introducing a named lifetime parameter
(ヘルプ: 名前付きライフライム引数の導入を検討してください)
  |
9 | fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
  |           ++++     ++          ++          ++
```

```
For more information about this error, try `rustc --explain E0106`.
error: could not compile `chapter10` (bin "chapter10") due to 1 previous error
```

助言テキストが、戻り値の型はジェネリックなライフタイム引数である必要があると明かしています。というのも、返している参照が x か y のどちらを参照しているか、コンパイラにはわからないからです。実際のところ、この関数の本体の if ブロックは x への参照を返し、else ブロックは y への参照を返すので、どちらなのか私たちにもわかりません！

この関数を定義する際、この関数に渡される具体的な値がわからないので、if ケースと else ケースのどちらが実行されるかわからないのです。また、リスト 10-17 と 10-18 で、返す参照が常に有効であるかを決定したときのようにスコープを見ることも、渡される参照の具体的なライフタイムがわからないのでできないのです。借用チェッカーもこれを決定することはできません。x と y のライフタイムがどう戻り値のライフタイムと関係するかわからないからです。このエラーを修正するために、借用チェッカーが解析を実行できるように、参照間の関係を定義するジェネリックなライフタイム引数を追加しましょう。

10.4.4　ライフタイム注釈記法

ライフタイム注釈は、いかなる参照の生存期間も変えることはありません。むしろ、ライフタイムに影響することなく、複数の参照のライフタイムのお互いの関係を記述します。シグニチャにジェネリックな型引数を指定された関数が、あらゆる型を受け取ることができるのと同様に、ジェネリックなライフタイム引数を指定された関数は、あらゆるライフタイムの参照を受け取ることができます。

ライフタイム注釈は、少し不自然な記法です: ライフタイム引数の名前はアポストロフィー (') で始まらなければならず、通常全部小文字で、ジェネリック型のようにとても短いです。多くの人は、最初のライフライム注釈に対して 'a という名前を使います。ライフタイム引数注釈は、参照の & の後に配置し、注釈と参照の型を区別するために空白を 1 つ使用します。

例を挙げましょう: ライフタイム引数なしの i32 への参照、'a というライフタイム引数付きの i32 への参照、そして同じくライフタイム 'a を持つ i32 への可変参照です。

```
&i32        // a reference
            // (ただの)参照
&'a i32     // a reference with an explicit lifetime
            // 明示的なライフタイム付きの参照
&'a mut i32 // a mutable reference with an explicit lifetime
            // 明示的なライフタイム付きの可変参照
```

1 つのライフタイム注釈それだけでは、大して意味はありません。注釈は、複数の参照のジェネリックなライフタイム引数が、お互いにどう関係するかをコンパイラに指示することを意図しているからです。longest 関数を例に、ライフタイム注釈が互いにどう関連していくのか、詳しく見ていきましょう。

10.4.5　関数シグニチャにおけるライフタイム注釈

関数シグネチャでライフタイム注釈を使用するには、ジェネリック**型**引数でやったのとまったく同じように、関数名と引数リストの間の山カッコの中にジェネリックな**ライフタイム**引数を宣言します。

このシグニチャで以下の制約を表現したいです: 返される参照は、両引数が有効である限り、有効になります。これが引数と戻り値の間のライフタイムの関係です。リスト 10-21 に示すように、ライフタイムを `'a` と名付け、それを各参照に付与します。

ファイル名: src/main.rs

```
# fn main() {
#     let string1 = String::from("abcd");
#     let string2 = "xyz";
#
#     let result = longest(string1.as_str(), string2);
#     println!("The longest string is {}", result);
# }
#
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

リスト 10-21: シグニチャの全参照が同じライフタイム `'a` を持つと指定した `longest` 関数の定義

このコードはコンパイルでき、リスト 10-19 の `main` 関数とともに使用したら、欲しい結果になるはずです。

これで関数シグニチャは、何らかのライフタイム `'a` に対して、関数は 2 つの引数を取り、どちらも少なくともライフタイム `'a` と同じだけ生きる文字列スライスであるとコンパイラに教えるようになりました。また、この関数シグニチャは、関数から返る文字列スライスも少なくともライフタイム `'a` と同じだけ生きると、コンパイラに教えています。実際には、`longest` 関数が返す参照のライフタイムは、関数実引数によって参照される値のうち、小さい方のライフタイムと同じであるという事です。これらの関係は、まさに私たちがコンパイラにこのコードを解析するときに使用してほしかったものです。

この関数シグニチャでライフタイム引数を指定する時、渡されたり、返したりした、いかなる値のライフタイムも変更していないことを思い出してください。むしろ、借用チェッカーは、これらの制約を守らない値全てを拒否するべきと指定しています。`longest` 関数は、`x` と `y` の正確な生存期間を知っている必要はなく、このシグニチャを満たすようなスコープを `'a` に代入できることを知っているだけであることに注意してください。

関数にライフタイムを注釈するときは、注釈は関数の本体ではなくシグニチャに付与します。ライフタイム注釈は、シグニチャ内の型がそうであるように、その関数の契約の一部を構成します。関数シグニチャにライフタイムの契約を含めることで、コンパイラが行う解析をより簡潔にすることができます。関数の注釈や呼ばれる状況に問題がある場合に、コンパイラのエラーはより正確にコードと制約の問題のある部分を示すことができます。もし Rust コンパイラが、プログラムが意図するライフタイムの関係についてこれより深く推論を行うとしたら、コンパイラは、問題の原因から何ステップも離れたコードを指し示すことしかできなくなるかもしれません。

具体的な参照を `longest` に渡すと、`'a` に代入される具体的なライフタイムは、`x` のスコープの一部であって `y` のスコープと重なる部分となります。言い換えると、ジェネリックなライフタイム `'a` は、`x` と `y` のライフタイムのうち、小さい方に等しい具体的なライフタイムになるのです。返却される参照を同じライフタイム引数 `'a` で注釈したので、返却される参照も `x` か `y` のライフタイムの小さい方と同じだけ有効になるでしょう。

ライフタイム注釈が異なる具体的なライフタイムを持つ参照を渡すことで `longest` 関数を制限する方法を見ましょう。リスト 10-22 はそのシンプルな例です。

ファイル名: src/main.rs

```
fn main() {
    // 長い文字列は長い
    let string1 = String::from("long string is long");
    // （訳注：この言葉自体に深い意味はない。下の"xyz"より長いということだけが重
    要）

    {
        let string2 = String::from("xyz");
        let result = longest(string1.as_str(), string2.as_str());
        // 一番長い文字列は{}
        println!("The longest string is {}", result);
    }
}
#
# fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
#     if x.len() > y.len() {
#         x
#     } else {
#         y
#     }
# }
```

リスト 10-22: 異なる具体的なライフタイムを持つ `String` 値への参照で `longest` 関数を使用する

この例において、`string1` は外側のスコープの終わりまで有効で、`string2` は内側のスコープの終わりまで有効、そして `result` は内側のスコープの終わりまで有効な何かを参照しています。このコードを実行すると、借用チェッカーが良しとするのがわかるでしょう。要するに、コンパイルでき、

`The longest string is long string is long` と出力するのです。

次に、`result` の参照のライフタイムが 2 つの引数の小さい方のライフタイムになることを示す例を試しましょう。`result` 変数の宣言を内側のスコープの外に移すものの、`result` 変数への代入は `string2` のスコープ内に残したままにします。それから `result` を使用する `println!` を内側のスコープの外、内側のスコープが終わった後に移動します。リスト 10-23 のコードはコンパイルできません。

ファイル名: src/main.rs

```
fn main() {
    let string1 = String::from("long string is long");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
    println!("The longest string is {}", result);
}
#
# fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
#     if x.len() > y.len() {
#         x
#     } else {
#         y
#     }
# }
```

リスト 10-23: `string2` がスコープを抜けてから `result` を使用しようとする

このコードのコンパイルを試みると、こんなエラーになります:

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0597]: `string2` does not live long enough
 --> src/main.rs:6:44
  |
5 |         let string2 = String::from("xyz");
  |             ------- binding `string2` declared here
6 |         result = longest(string1.as_str(), string2.as_str());
  |                                             ^^^^^^^ borrowed value does not
      live long enough
7 |     }
  |     - `string2` dropped here while still borrowed
8 |     println!("The longest string is {}", result);
  |                                          ------ borrow later used here

For more information about this error, try `rustc --explain E0597`.
error: could not compile `chapter10` (bin "chapter10") due to 1 previous error
```

このエラーは、`result` が `println!` 文に対して有効であるためには、`string2` が外側のスコープの

終わりまで有効である必要があることを示しています。関数引数と戻り値のライフタイムを同じライフタイム引数 `'a` で注釈したので、コンパイラはこのことを知っています。

人間からしたら、`string1` は `string2` よりも長く、それ故に `result` が `string1` への参照を含んでいることはコードから明らかです。まだ `string1` はスコープを抜けていないので、`string1` への参照は `println!` にとって有効でしょう。ですが、コンパイラはこの場合、参照が有効であると見なせません。`longest` 関数から返ってくる参照のライフタイムは、渡した参照のうちの小さい方と同じだとコンパイラに指示しました。したがって、借用チェッカーは、リスト 10-23 のコードを無効な参照がある可能性があるとして許可しないのです。

試しに、値や、`longest` 関数に渡される参照のライフタイムや、返される参照の使われかたが異なる実験をもっとしてみてください。コンパイル前に、その実験が借用チェッカーを通るかどうか仮説を立ててください; そして、正しいか確かめてください！

10.4.6　ライフタイムの観点で思考する

何にライフタイム引数を指定する必要があるかは、関数が行っていることに依存します。例えば、`longest` 関数の実装を最長の文字列スライスではなく、常に最初の引数を返すように変更したら、`y` 引数に対してライフタイムを指定する必要はなくなるでしょう。以下のコードはコンパイルできます:

ファイル名: src/main.rs

```
# fn main() {
#     let string1 = String::from("abcd");
#     let string2 = "efghijklmnopqrstuvwxyz";
#
#     let result = longest(string1.as_str(), string2);
#     println!("The longest string is {}", result);
# }
#
fn longest<'a>(x: &'a str, y: &str) -> &'a str {
    x
}
```

引数 `x` と戻り値に対してライフタイム引数 `'a` を指定しましたが、引数 `y` には指定していません。`y` のライフタイムは `x` や戻り値のライフタイムとは何の関係もないからです。

関数から参照を返す際、戻り値型のライフタイム引数は、引数のうちどれかのライフタイム引数と一致する必要があります。返される参照が引数のどれかを参照**していない**ならば、この関数内で生成された値を参照しているはずです。しかしながら、その値は関数の末端でスコープを抜けるので、これはダングリング参照になるでしょう。以下に示す、コンパイルできない `longest` 関数の未完成の実装を考えてください:

ファイル名: src/main.rs

```
# fn main() {
#     let string1 = String::from("abcd");
#     let string2 = "xyz";
#
#     let result = longest(string1.as_str(), string2);
#     println!("The longest string is {}", result);
# }
#
fn longest<'a>(x: &str, y: &str) -> &'a str {
    // 本当に長い文字列
    let result = String::from("really long string");
    result.as_str()
}
```

ここでは、たとえ、戻り値型にライフタイム引数 `'a` を指定していても、戻り値のライフタイムは、引数のライフタイムと全く関係がないので、この実装はコンパイルできないでしょう。こちらが、得られるエラーメッセージです:

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0515]: cannot return value referencing local variable `result`
(エラー[E0515]: ローカル変数`result`を参照する値を返すことはできません)
  --> src/main.rs:11:5
   |
11 |     result.as_str()
   |     ------^^^^^^^^^^
   |     |
   |     returns a value referencing data owned by the current function
   |    (現在の関数に所有されているデータを参照する値を返しています)
   |     `result` is borrowed here
   |    (`result`はここで借用されています)

For more information about this error, try `rustc --explain E0515`.
error: could not compile `chapter10` (bin "chapter10") due to 1 previous error
```

問題は、`result` が `longest` 関数の末端でスコープを抜け、片付けられてしまうことです。かつ、関数から `result` への参照を返そうともしています。ダングリング参照を変えてくれるようなライフタイム引数を指定する手段はなく、コンパイラは、ダングリング参照を生成させてくれません。今回の場合、最善の修正案は、(呼び出し先ではなく) 呼び出し元の関数に値の片付けをさせるために、参照ではなく所有されたデータ型を返すことでしょう。

究極的にライフタイム記法は、関数のいろんな引数と戻り値のライフタイムを接続することに関するものです。一旦それらが繋がれば、メモリ安全な処理を許可し、ダングリングポインタを生成したりメモリ安全性を侵害したりする処理を禁止するのに十分な情報をコンパイラは得たことになります。

10.4.7　構造体定義のライフタイム注釈

ここまで定義してきた構造体はすべて所有された型を保持しています。参照を保持する構造体を定義することもできますが、その場合、構造体定義の全参照にライフタイム注釈を付け加える必要があるでしょう。リスト 10-24 には、文字列スライスを保持する `ImportantExcerpt` (重要な一節) という構造体があります。

ファイル名: src/main.rs

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

fn main() {
    let novel = String::from("Call me Ishmael. Some years ago...");
    let first_sentence = novel.split('.').next().expect("Could not find a '.'");
    let i = ImportantExcerpt {
        part: first_sentence,
    };
}
```

リスト 10-24: ライフタイム注釈を必要とする、参照を含む構造体

この構造体には文字列スライスを保持する単一のフィールド、`part` があり、これは参照です。ジェネリックなデータ型同様、構造体名の後、山カッコの中にジェネリックなライフタイム引数の名前を宣言するので、構造体定義の本体でライフタイム引数を使用できます。この注釈は、`ImportantExcerpt` のインスタンスが、`part` フィールドに保持している参照よりも長生きしないことを意味します。

ここの `main` 関数は、変数 `novel` に所有される `String` の、最初の文への参照を保持する `ImportantExcerpt` インスタンスを生成しています。`novel` のデータは、`ImportantExcerpt` インスタンスが作られる前に存在しています。加えて、`ImportantExcerpt` がスコープを抜けるまで `novel` はスコープを抜けないので、`ImportantExcerpt` インスタンスの参照は有効なのです。

10.4.8　ライフタイム省略

全参照にはライフタイムがあり、参照を使用する関数や構造体にはライフタイム引数を指定する必要があることを学びました。しかし、リスト 4-9 にあった関数（リスト 10-25 に再度示しました）はライフタイム注釈なしでコンパイルできました。

ファイル名: src/lib.rs

```
fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();
```

```
    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
#
# fn main() {
#     let my_string = String::from("hello world");
#
#     // first_word works on slices of `String`s
#     let word = first_word(&my_string[..]);
#
#     let my_string_literal = "hello world";
#
#     // first_word works on slices of string literals
#     let word = first_word(&my_string_literal[..]);
#
#     // Because string literals *are* string slices already,
#     // this works too, without the slice syntax!
#     let word = first_word(my_string_literal);
# }
```

リスト 10-25: リスト 4-9 で定義した、引数と戻り値型が参照であるにも関わらず、ライフタイム注釈なしでコンパイルできた関数

この関数がライフタイム注釈なしでコンパイルできる理由には、Rust の歴史が関わっています: 昔のバージョンの Rust(1.0 以前) では、全参照に明示的なライフタイムが必要だったので、このコードはコンパイルできませんでした。その頃、関数シグニチャはこのように記述されていたのです:

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

多くの Rust コードを書いた後、Rust チームは、Rust プログラマが、特定の場面で何度も同じライフタイム注釈を入力していることを発見しました。これらの場面は予測可能で、いくつかの決まりきったパターンに従っていました。開発者はこのパターンをコンパイラのコードに落とし込んだので、このような場面には借用チェッカーがライフタイムを推論できるようになり、明示的な注釈を必要としなくなったのです。

ここで、この Rust の歴史話が関係しているのは、他にも決まりきったパターンが出現し、コンパイラに追加されることもあり得るからです。将来的に、さらに少数のライフタイム注釈しか必要にならない可能性もあります。

コンパイラの参照解析に落とし込まれたパターンは、**ライフタイム省略規則**と呼ばれます。これらはプログラマが従う規則ではありません; コンパイラが考慮する一連の特定のケースであり、自分のコードがこのケースに当てはまれば、ライフタイムを明示的に書く必要はなくなります。

省略規則は、完全な推論を提供しません。コンパイラが決定的に規則を適用できるけれども、参照が保持するライフタイムに関してそれでも曖昧性があるなら、コンパイラは、残りの参照がなるべきライフタイムを推測しません。コンパイラはそれらを推測するのではなく、ライフタイム注釈を追記することで解決できるようなエラーを与えます。

関数やメソッドの引数のライフタイムは、**入力ライフタイム**と呼ばれ、戻り値のライフタイムは**出力ライフタイム**と称されます。

コンパイラは 3 つの規則を活用し、明示的な注釈がない時に、参照のライフタイムを計算します。最初の規則は入力ライフタイムに適用され、2 番目と 3 番目の規則は出力ライフタイムに適用されます。コンパイラが 3 つの規則の最後まで到達し、それでもライフタイムを割り出せない参照があったら、コンパイラはエラーで停止します。これらのルールは `fn` の定義にも `impl` ブロックにも適用されます。

最初の規則は、コンパイラは参照である各引数に、ライフタイム引数を割り当てるというものです。換言すれば、1 引数の関数は、1 つのライフタイム引数を得るということです: `fn foo<'a>(x: &'a i32)` ; 2 つ引数のある関数は、2 つの個別のライフタイム引数を得ます: `fn foo<'a, 'b>(x: &'a i32, y: &'b i32)` ; 以下同様。

2 番目の規則は、1 つだけ入力ライフタイム引数があるなら、そのライフタイムが全ての出力ライフタイム引数に代入されるというものです: `fn foo<'a>(x: &'a i32) -> &'a i32` 。

3 番目の規則は、複数の入力ライフタイム引数があるけれども、メソッドなのでそのうちの一つが `&self` や `&mut self` だったら、`self` のライフタイムが全出力ライフタイム引数に代入されるというものです。この 3 番目の規則により、必要なシンボルの数が減るので、メソッドが遥かに読み書きしやすくなります。

コンパイラの立場になってみましょう。これらの規則を適用して、リスト 10-25 の `first_word` 関数のシグニチャの参照のライフタイムを計算します。シグニチャは、参照に紐づけられるライフタイムがない状態から始まります:

```
fn first_word(s: &str) -> &str {
```

そうして、コンパイラは最初の規則を適用し、各引数が独自のライフタイムを得ると指定します。それを通常通り `'a` と呼ぶので、シグニチャはこうなります:

```
fn first_word<'a>(s: &'a str) -> &str {
```

1 つだけ入力ライフタイムがあるので、2 番目の規則を適用します。2 番目の規則は、1 つの入力引数のライフタイムが、出力引数に代入されると指定するので、シグニチャはこうなります:

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

もうこの関数シグニチャの全ての参照にライフタイムが付いたので、コンパイラは、プログラマにこの関数シグニチャのライフタイムを注釈してもらう必要なく、解析を続行できます。

別の例に目を向けましょう。今回は、リスト 10-20 で取り掛かったときにはライフタイム引数がなかった `longest` 関数です:

```
fn longest(x: &str, y: &str) -> &str {
```

最初の規則を適用しましょう: 各引数が独自のライフタイムを得るのです。今回は、1 つではなく 2 つ引数があるので、ライフタイムも 2 つです:

```
fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &str {
```

2 つ以上入力ライフタイムがあるので、2 番目の規則は適用されないとわかります。また 3 番目の規則も適用されません。`longest` はメソッドではなく関数なので、どの引数も `self` ではないのです。3 つの規則全部を適用した後でも、まだ戻り値型のライフタイムが判明していません。このために、リスト 10-20 でこのコードをコンパイルしようとしてエラーになったのです: コンパイラは、ライフタイム省略規則全てを適用したけれども、シグニチャの参照全部のライフタイムを計算できなかったのです。

実際のところ、3 番目の規則はメソッドのシグニチャにしか適用されません。ですので、次はその文脈においてライフタイムを観察し、3 番目の規則のおかげで、メソッドシグニチャであまり頻繁にライフタイムを注釈しなくても済む理由を確認します。

10.4.9　メソッド定義におけるライフタイム注釈

構造体にライフタイムのあるメソッドを実装する際、リスト 10-11 で示したジェネリックな型引数と同じ記法を使用します。ライフタイム引数を宣言し使用する場所は、構造体フィールドかメソッド引数と戻り値に関係するかによります。

構造体のフィールド用のライフタイム名は、`impl` キーワードの後に宣言する必要があり、それから構造体名の後に使用されます。そのようなライフタイムは構造体の型の一部になるからです。

`impl` ブロック内のメソッドシグニチャでは、参照は構造体のフィールドの参照のライフタイムに紐づいている可能性と、独立している可能性があります。加えて、ライフタイム省略規則により、メソッドシグニチャでライフタイム注釈が必要なくなることがよくあります。リスト 10-24 で定義した `ImportantExcerpt` という構造体を使用した例をいくつか見てみましょう。

まず、唯一の引数が `self` への参照で戻り値が `i32` という何かへの参照ではない `level` というメソッドを使用します:

```
# struct ImportantExcerpt<'a> {
#     part: &'a str,
# }
#
impl<'a> ImportantExcerpt<'a> {
    fn level(&self) -> i32 {
        3
```

```
    }
}
#
# impl<'a> ImportantExcerpt<'a> {
#     fn announce_and_return_part(&self, announcement: &str) -> &str {
#         //       "お知らせします: {}"
#         println!("Attention please: {}", announcement);
#         self.part
#     }
# }
#
# fn main() {
#     let novel = String::from("Call me Ishmael. Some years ago...");
#     let first_sentence = novel.split('.').next().expect("Could not find a '.'"
    );
#     let i = ImportantExcerpt {
#         part: first_sentence,
#     };
# }
```

`impl` 後のライフタイム引数宣言と型名の後にそれを使用するのは必須ですが、最初の省略規則のため、`self` への参照のライフタイムを注釈する必要はありません。

3 番目のライフタイム省略規則が適用される例はこちらです:

```
# struct ImportantExcerpt<'a> {
#     part: &'a str,
# }
#
# impl<'a> ImportantExcerpt<'a> {
#     fn level(&self) -> i32 {
#         3
#     }
# }
#
impl<'a> ImportantExcerpt<'a> {
    fn announce_and_return_part(&self, announcement: &str) -> &str {
        //       "お知らせします: {}"
        println!("Attention please: {}", announcement);
        self.part
    }
}
#
# fn main() {
#     let novel = String::from("Call me Ishmael. Some years ago...");
#     let first_sentence = novel.split('.').next().expect("Could not find a '.'"
    );
#     let i = ImportantExcerpt {
#         part: first_sentence,
#     };
# }
```

2 つ入力ライフタイムがあるので、コンパイラは最初のライフタイム省略規則を適用し、`&self` と announcement に独自のライフタイムを与えます。それから、引数の 1 つが `&self` なので、戻り値型は `&self` のライフタイムを得て、全てのライフタイムが説明されました。

10.4.10　静的ライフタイム

議論する必要のある 1 種の特殊なライフタイムが `'static` であり、これは影響を受ける参照がプログラムの全期間生存**できる**ことを示します。文字列リテラルは全て `'static` ライフタイムになり、次のように注釈できます:

```
// 僕は静的ライフタイムを持ってるよ
let s: &'static str = "I have a static lifetime.";
```

この文字列のテキストは、プログラムのバイナリに直接格納され、常に利用可能です。故に、全文字列リテラルのライフタイムは、`'static` なのです。

エラーメッセージで、`'static` ライフタイムを使用するよう勧める提言を見かける可能性があります。ですが、参照に対してライフタイムとして `'static` を指定する前に、今ある参照が本当にプログラムの全期間生きるかどうか、そして生きてほしいのかどうか、考えてください。ほとんどの場合、`'static` ライフタイムを提案するエラーメッセージは、ダングリング参照を生成しようとしているか、利用可能なライフタイムの不一致が原因です。そのような場合、解決策はその問題を修正することであり、`'static` ライフタイムを指定することではありません。

10.5　ジェネリックな型引数、トレイト境界、ライフタイムを一度に

ジェネリックな型引数、トレイト境界、ライフタイム指定の構文のすべてを 1 つの関数で簡単に見てみましょう！

```
# fn main() {
#     let string1 = String::from("abcd");
#     let string2 = "xyz";
#
#     let result = longest_with_an_announcement(
#         string1.as_str(),
#         string2,
#         "Today is someone's birthday!",
#     );
#     println!("The longest string is {}", result);
# }
#
use std::fmt::Display;

fn longest_with_an_announcement<'a, T>(
    x: &'a str,
    y: &'a str,
```

```
    ann: T,
) -> &'a str
where
    T: Display,
{
    //       "アナウンス！  {}"
    println!("Announcement! {}", ann);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

これがリスト 10-21 からの 2 つの文字列のうち長い方を返す `longest` 関数ですが、ジェネリックな型 `T` の `ann` という追加の引数があり、これは `where` 節で指定されているように、`Display` トレイトを実装するあらゆる型で埋めることができます。この追加の引数は `{}` を使用して出力されるので、`Display` トレイト境界が必要なのです。ライフタイムは一種のジェネリックなので、ライフタイム引数 `'a` とジェネリックな型引数 `T` が関数名の後、山カッコ内の同じリストに収まっています。

10.6　まとめ

たくさんのことをこの章では講義しましたね！ 今やジェネリックな型引数、トレイトとトレイト境界、そしてジェネリックなライフタイム引数を知ったので、多くの異なる場面で動くコードを繰り返すことなく書く準備ができました。ジェネリックな型引数により、コードを異なる型に適用させてくれます。トレイトとトレイト境界は、型がジェネリックであっても、コードが必要とする振る舞いを持つことを保証します。ライフタイム注釈を活用して、この柔軟なコードにダングリング参照が存在しないことを保証する方法を学びました。さらにこの解析は全てコンパイル時に起こり、実行時のパフォーマンスには影響しません！

信じられないかもしれませんが、この章で議論した話題にはもっともっと学ぶべきことがあります: 第 17 章ではトレイトオブジェクトを議論します。これはトレイトを使用する別の手段です。非常に高度な筋書きの場合でのみ必要になる、ライフタイム注釈が関わる、もっと複雑な筋書きもあります。それらについては、Rust Reference をお読みください。ですが次は、コードがあるべき通りに動いていることを確かめられるように、Rust でテストを書く方法を学びます。

第 11 章

自動テストを書く

1972 年のエッセイ「謙虚なプログラマ」でエドガー・W・ダイクストラは以下のように述べています。「プログラムのテストは、バグの存在を示すには非常に効率的な手法であるが、バグの不在を示すには望み薄く不適切である」と。これは、できるだけテストを試みるべきではないということではありません。

プログラムの正当性は、どこまで自分のコードが意図していることをしているかなのです。Rust は、プログラムの正当性に重きを置いて設計されていますが、正当性は複雑で、単純に証明することはありません。Rust の型システムは、この重荷の多くの部分を肩代わりしてくれますが、型システムはすべてを捕捉してはくれません。ゆえに、Rust では、自動化されたソフトウェアテストを書くことをサポートしているのです。

渡された何かの数値に 2 を足す `add_two` 関数を書くとしましょう。この関数のシグニチャは、引数に整数を取り、結果として整数を返します。この関数を実装してコンパイルすると、コンパイラはこれまでに学んできた型チェックと借用チェックを全て行い、例えば、`String` の値や無効な参照をこの関数に渡していないかなどを確かめるのです。ところが、コンパイラはプログラマがまさしく意図したことを関数が実行しているかどうかは確かめ**られません**。つまり、そうですね、引数に 10 を足したり、50 を引いたりするのではなく、引数に 2 を足していることです。そんな時に、テストは必要になるのです。

例えば、`add_two` 関数に `3` を渡した時に、戻り値は 5 であることをアサーションするようなテストを書くことができます。コードに変更を加えた際にこれらのテストを走らせ、既存の正当な振る舞いが変わっていないことを確認できます。

テストは、複雑なスキルです: いいテストの書き方をあらゆる方面から講義することは 1 章だけではできないのですが、Rust のテスト機構のメカニズムについて議論します。テストを書く際に利用可能になるアノテーションとマクロについて、テストを実行するのに提供されているオプションと標準の動作、さらにテストをユニットテストや統合テストに体系化する方法について語ります。

11.1　テストの記述法

テストは、テスト以外のコードが想定された方法で機能していることを実証する Rust の関数です。テスト関数の本体は、典型的には以下の 3 つの動作を行います:

1. 必要なデータや状態をセットアップする。
2. テスト対象のコードを走らせる。
3. 結果が想定通りであることを断定（以下、アサーションという）する。

Rust が、特にこれらの動作を行うテストを書くために用意している機能を見ていきましょう。これには、`test` 属性、いくつかのマクロ、`should_panic` 属性が含まれます。

11.1.1　テスト関数の構成

最も単純には、Rust におけるテストは `test` 属性で注釈された関数のことです。属性とは、Rust コードの部品に関するメタデータです; 一例を挙げれば、構造体とともに第 5 章で使用した `derive` 属性です。関数をテスト関数に変えるには、`fn` の前に `#[test]` を付け加えてください。`cargo test` コマンドでテストを実行したら、コンパイラは注釈された関数を走らせるテスト用バイナリをビルドし、各テスト関数が通過したか失敗したかを報告します。

新しいライブラリプロジェクトを Cargo で作ると、テスト関数付きのテストモジュールが自動的に生成されます。このモジュールがテストを書くためのテンプレートを提供してくれるので、新しいプロジェクトを始めるたびに正しい構造とか文法をいちいち検索しなくてすみます。ここに好きな数だけテスト関数やテストモジュールを追加すればいいというわけです！

まずは実際にコードをテストする前に、自動生成されたテンプレートのテストで実験して、テストの動作の性質をいくらか学びましょう。その後で、以前書いたコードを呼び出し、振る舞いが正しいことをアサーションする、ホンモノのテストを書きましょう。

2 つの数を足す、`adder` という新しいライブラリプロジェクトを生成しましょう:

```
$ cargo new adder --lib
     Created library `adder` project
$ cd adder
```

`adder` ライブラリの **src/lib.rs** ファイルの中身は、リスト 11-1 のような見た目のはずです。

ファイル名: src/lib.rs

```
pub fn add(left: usize, right: usize) -> usize {
    left + right
}

#[cfg(test)]
```

```
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        let result = add(2, 2);
        assert_eq!(result, 4);
    }
}
```

リスト 11-1: `cargo new` で自動生成されたテストモジュールと関数

とりあえず、最初の 2 行は無視し、関数に集中しましょう。`#[test]` 注釈に注目してください: この属性は、これがテスト関数であることを示すので、テスト実行機はこの関数をテストとして扱うとわかるのです。さらに、`tests` モジュール内にはテスト関数以外の関数を入れ、一般的なシナリオをセットアップしたり、共通の処理を行う手助けをしたりもできるので、必ずどの関数がテストかを示す必要があるのです。

例の関数本体は、`assert_eq!` マクロを使用して、2 と 2 を足した結果を含む `result` が、4 に等しいことをアサーションしています。このアサーションは、典型的なテストのフォーマット例をなしているわけです。走らせてこのテストが通る（訳注：テストが成功する、の意味。英語で pass ということから、このように表現される）ことを確かめましょう。

`cargo test` コマンドでプロジェクトにあるテストが全て実行されます。リスト 11-2 に示したようにですね。

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.57s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s
```

リスト 11-2: 自動生成されたテストを走らせた出力

Cargo がテストをコンパイルし、走らせました。`running 1 test` という行が見えます。その次の行は、生成されたテスト関数の `it_works` という名前と、このテストの実行結果が `ok` であることを

示しています。テスト全体のまとめである `test result:ok.` は、全テストが通ったことを意味し、`1 passed; 0 failed` と読める部分は、通過または失敗したテストの数を合計しているのです。

特定の場合にテスト実行しないように、テストを無視するように指定することができます; これについては後でこの章の「特に要望のない限りテストを無視する」節で扱います。ここではそれを行っていないので、まとめは `0 ignored` と示しています。`cargo test` コマンドに引数を渡すことで、名前が文字列にマッチするテストのみを実行することもできます; **フィルタリング**と呼ばれますが、これについては「名前でテストの一部を実行する」節で扱います。また、実行するテストにフィルタをかけもしなかったので、まとめの最後に `0 filtered out` と表示されています。

`0 measured` という統計は、パフォーマンスを測定するベンチマークテスト用です。ベンチマークテストは、本書記述の時点では、nightly 版の Rust でのみ利用可能です。詳しくは、ベンチマークテストのドキュメンテーションを参照されたし。

テスト出力の次の部分、つまり `Doc-tests adder` で始まる部分は、ドキュメンテーションテストの結果用のものです。まだドキュメンテーションテストは何もないものの、コンパイラは、API ドキュメントに現れるどんなコード例もコンパイルできます。この機能により、ドキュメントとコードを同期することができるわけです。ドキュメンテーションテストの書き方については、第 14 章のテストとしてのドキュメンテーションコメント節で議論しましょう。今は、`Doc-tests` 出力は無視します。

それでは必要に応じてテストをカスタマイズしていきましょう。まずは `it_works` 関数の名前を違う名前に、例えば以下の `exploration` のように変更してください:

ファイル名: src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn exploration() {
        assert_eq!(2 + 2, 4);
    }
}
```

そして、`cargo test` を再度走らせます。これで出力が `it_works` の代わりに `exploration` と表示しています:

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.59s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::exploration ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

   Doc-tests adder
```

```
running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s
```

それでは別のテストを追加していきます。ただし、今回は失敗するテストにしましょう！ テスト関数内の何かがパニックすると、テストは失敗します。各テストは、新規スレッドで実行され、メインスレッドが、テストスレッドが死んだと確認した時、テストは失敗と印づけられます。第 9 章で、パニックを引き起こす最も単純な方法は `panic!` マクロを呼び出すことだと語りました。**src/lib.rs** ファイルがリスト 11-3 のような見た目になるよう、`another` という名前の関数として新しいテストを入力してください。

ファイル名: src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn exploration() {
        assert_eq!(2 + 2, 4);
    }

    #[test]
    fn another() {
        //このテストを失敗させる
        panic!("Make this test fail");
    }
}
```

リスト 11-3: `panic!` マクロを呼び出したために失敗する 2 番目のテストを追加する

`cargo test` で再度テストを走らせてください。出力はリスト 11-4 のようになるはずであり、`exploration` テストは通り、`another` は失敗したと表示されます。

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.72s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 2 tests
test tests::another ... FAILED
test tests::exploration ... ok

failures:

---- tests::another stdout ----
thread 'tests::another' panicked at src/lib.rs:10:9:
Make this test fail
```

```
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::another

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

error: test failed, to rerun pass `--lib`
```

リスト 11-4: 一つのテストが通り、一つが失敗するときのテスト結果

ok の代わりに test test::another の行は、FAILED を表示しています。個々の結果とまとめの間に、2 つ新たな区域ができました: 最初の区域は、失敗したテスト各々の具体的な理由を表示しています。今回の場合、another は **src/lib.rs** ファイルの 10 行目で 'Make this test fail'でパニックしたために失敗した、という詳細が得られました。次の区域は失敗したテストの名前だけを列挙しています。これは、テストがたくさんあり、失敗したテストの詳細がたくさん表示されるときに有用になります。失敗したテストの名前を使用してそのテストだけを実行し、より簡単にデバッグすることができます。テストの実行方法については、テストの実行され方を制御する節でもっと語りましょう。

サマリー行が最後に出力されています: 総合的に言うと、テスト結果は FAILED でした。一つのテストが通り、一つが失敗したわけです。

様々な状況でのテスト結果がどんな風になるか見てきたので、テストを行う際に有用になる panic! 以外のマクロに目を向けましょう。

11.1.2　assert!マクロで結果を確認する

assert! マクロは、標準ライブラリで提供されていますが、テスト内の何らかの条件が true と評価されることを確かめたいときに有効です。assert! マクロには、論理値に評価される引数を与えます。その値が true なら、何も起こらずにテストは通ります。その値が false なら、assert! マクロは panic! を呼び出し、テストは失敗します。assert! マクロを使用することで、コードが意図した通りに機能していることを確認する助けになるわけです。

第 5 章のリスト 5-15 で、Rectangle 構造体と can_hold メソッドを使用しました。リスト 11-5 でもそれを繰り返しています。このコードを **src/lib.rs** ファイルに放り込み、assert! マクロでそれ用のテストを何か書いてみましょう。

ファイル名: src/lib.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}
```

```
impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

リスト 11-5: 第 5 章から Rectangle 構造体とその can_hold メソッドを使用する

can_hold メソッドは論理値を返すので、assert! マクロの完璧なユースケースになるわけです。リスト 11-6 で、幅が 8、高さが 7 の Rectangle インスタンスを生成し、これが幅 5、高さ 1 の別の Rectangle インスタンスを保持できるとアサーションすることで can_hold を用いるテストを書きます。

ファイル名: src/lib.rs

```
# #[derive(Debug)]
# struct Rectangle {
#     width: u32,
#     height: u32,
# }
#
# impl Rectangle {
#     fn can_hold(&self, other: &Rectangle) -> bool {
#         self.width > other.width && self.height > other.height
#     }
# }
#
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn larger_can_hold_smaller() {
        let larger = Rectangle {
            width: 8,
            height: 7,
        };
        let smaller = Rectangle {
            width: 5,
            height: 1,
        };

        assert!(larger.can_hold(&smaller));
    }
}
```

リスト 11-6: より大きな長方形がより小さな長方形を確かに保持できるかを確認する can_hold 用のテスト

tests モジュール内に新しい行を加えたことに注目してください： use super::* です。tests モジュールは、第 7 章のモジュールツリーの要素を示すためのパス節で講義した通常の公開ルールに従う普通のモジュールです。tests モジュールは、内部モジュールなので、外部モジュール内のテスト配下にあるコードを内部モジュールのスコープに持っていく必要があります。ここでは glob を使用して、外部モジュールで定義したもの全てがこの tests モジュールでも使用可能になるようにしています。

テストは larger_can_hold_smaller と名付け、必要な Rectangle インスタンスを 2 つ生成しています。そして、assert! マクロを呼び出し、larger.can_hold(&smaller) の呼び出し結果を渡しました。この式は、true を返すと考えられるので、テストは通るはずです。確かめましょう！

```
$ cargo test
   Compiling rectangle v0.1.0 (file:///projects/rectangle)
    Finished test [unoptimized + debuginfo] target(s) in 0.66s
     Running unittests src/lib.rs (target/debug/deps/rectangle-6584c4561e48942e)

running 1 test
test tests::larger_can_hold_smaller ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

   Doc-tests rectangle

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s
```

通ります！ 別のテストを追加しましょう。今回は、小さい長方形は、より大きな長方形を保持できないことをアサーションします。

ファイル名: src/lib.rs

```
# #[derive(Debug)]
# struct Rectangle {
#     width: u32,
#     height: u32,
# }
#
# impl Rectangle {
#     fn can_hold(&self, other: &Rectangle) -> bool {
#         self.width > other.width && self.height > other.height
#     }
# }
#
#[cfg(test)]
mod tests {
    use super::*;
```

```
    #[test]
    fn larger_can_hold_smaller() {
        // --snip--
#         let larger = Rectangle {
#             width: 8,
#             height: 7,
#         };
#         let smaller = Rectangle {
#             width: 5,
#             height: 1,
#         };
#
#         assert!(larger.can_hold(&smaller));
    }

    #[test]
    fn smaller_cannot_hold_larger() {
        let larger = Rectangle {
            width: 8,
            height: 7,
        };
        let smaller = Rectangle {
            width: 5,
            height: 1,
        };

        assert!(!smaller.can_hold(&larger));
    }
}
```

今回の場合、can_hold 関数の正しい結果は false なので、その結果を assert! マクロに渡す前に反転させる必要があります。結果として、can_hold が false を返せば、テストは通ります。

```
$ cargo test
   Compiling rectangle v0.1.0 (file:///projects/rectangle)
    Finished test [unoptimized + debuginfo] target(s) in 0.66s
     Running unittests src/lib.rs (target/debug/deps/rectangle-6584c4561e48942e)

running 2 tests
test tests::larger_can_hold_smaller ... ok
test tests::smaller_cannot_hold_larger ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

   Doc-tests rectangle

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s
```

通るテストが 2 つ！ さて、コードにバグを導入したらテスト結果がどうなるか確認してみましょう。幅を比較する大なり記号を小なり記号で置き換えて `can_hold` メソッドの実装を変更しましょう:

```
# #[derive(Debug)]
# struct Rectangle {
#     width: u32,
#     height: u32,
# }
#
// --snip--
impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width < other.width && self.height > other.height
    }
}
#
# #[cfg(test)]
# mod tests {
#     use super::*;
#
#     #[test]
#     fn larger_can_hold_smaller() {
#         let larger = Rectangle {
#             width: 8,
#             height: 7,
#         };
#         let smaller = Rectangle {
#             width: 5,
#             height: 1,
#         };
#
#         assert!(larger.can_hold(&smaller));
#     }
#
#     #[test]
#     fn smaller_cannot_hold_larger() {
#         let larger = Rectangle {
#             width: 8,
#             height: 7,
#         };
#         let smaller = Rectangle {
#             width: 5,
#             height: 1,
#         };
#
#         assert!(!smaller.can_hold(&larger));
#     }
# }
```

テストを実行すると、以下のような出力をします:

```
$ cargo test
   Compiling rectangle v0.1.0 (file:///projects/rectangle)
    Finished test [unoptimized + debuginfo] target(s) in 0.66s
     Running unittests src/lib.rs (target/debug/deps/rectangle-6584c4561e48942e)

running 2 tests
test tests::larger_can_hold_smaller ... FAILED
test tests::smaller_cannot_hold_larger ... ok

failures:

---- tests::larger_can_hold_smaller stdout ----
thread 'tests::larger_can_hold_smaller' panicked at src/lib.rs:28:9:
assertion failed: larger.can_hold(&smaller)
(スレッド'tests::larger_can_hold_smaller'はsrc/lib.rs:28:9でパニックしました:
アサーション失敗: larger.can_hold(&smaller))
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::larger_can_hold_smaller

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

error: test failed, to rerun pass `--lib`
```

テストによりバグが捕捉されました！ `larger.width` が 8、`smaller.width` が 5 なので、`can_hold` 内の幅の比較が今は `false` を返すようになったのです: 8 は 5 より小さくないですからね。

11.1.3　`assert_eq!` と `assert_ne!` マクロで等値性をテストする

機能性を検証する一般的な方法は、テスト下にあるコードの結果と、コードが返すと期待される値との、等値性を確かめることです。これを `assert` マクロを使用して `==` 演算子を使用した式を渡すことで行うこともできます。しかしながら、これはありふれたテストなので、標準ライブラリには 1 組のマクロ (`assert_eq!` と `assert_ne!`) が提供され、このテストをより便利に行うことができます。これらのマクロはそれぞれ、二つの引数を比べ、等しいかと等しくないかを確かめます。また、アサーションが失敗したら二つの値の出力もし、テストが失敗した**原因**を確認しやすくなります。一方で `assert!` マクロは、`==` 式の値が `false` になったことしか示さず、`false` になった原因の値は出力しません。

リスト 11-7 では、引数に `2` を加える `add_two` という名前の関数を書いて、この関数を `assert_eq!` マクロでテストしています。

ファイル名: src/lib.rs

```
pub fn add_two(a: i32) -> i32 {
```

```
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_adds_two() {
        assert_eq!(4, add_two(2));
    }
}
```

リスト 11-7: assert_eq! マクロで add_two 関数をテストする

これが通ることを確認しましょう！

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.58s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s
```

assert_eq! マクロに引数として 4 を渡していますが、これは add_two(2) の呼び出し結果と等しいです。このテストの行は test tests::it_adds_two ... ok であり、ok というテキストはテストが通ったことを示しています！

コードにバグを仕込んで、assert_eq! が失敗した時にそれがどうなるのか確認してみましょう。add_two 関数の実装を代わりに 3 を足すように変えてください:

```
pub fn add_two(a: i32) -> i32 {
    a + 3
}
#
# #[cfg(test)]
# mod tests {
#     use super::*;
#
#     #[test]
```

```
#     fn it_adds_two() {
#         assert_eq!(4, add_two(2));
#     }
# }
```

テストを再度実行します:

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.61s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::it_adds_two ... FAILED

failures:

---- tests::it_adds_two stdout ----
thread 'tests::it_adds_two' panicked at src/lib.rs:11:9:
assertion `left == right` failed
  left: 4
 right: 5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::it_adds_two

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

error: test failed, to rerun pass `--lib`
```

テストがバグを捕捉しました！ `it_adds_two` のテストは失敗し、そのメッセージは、失敗したアサーションが `assertion failed: `(left == right)`` であったこと、そして `left` と `right` の値が何だったかを示しています。このメッセージはデバッグを開始する助けになります: `left` 引数は `4` だったが、`add_two(2)` がある `right` 引数は `5` でした。実行中のテストが多数あるときは特に、この出力は役に立つだろうと想像できるでしょう。

言語やテストフレームワークによっては、等値性アサーション関数の引数を `expected` と `actual` と呼び、引数を指定する順序が重要であることに注意してください。ですが Rust では、これらは `left` と `right` と呼ばれ、期待する値とコードが生成する値を指定する順序は重要ではありません。今回のテストのアサーションを `assert_eq!(add_two(2), 4)` と書くこともでき、そうすると失敗メッセージは、同じく `assertion failed: `(left == right)`` を表示するでしょう。

`assert_ne!` マクロは、与えた 2 つの値が等しくなければ通り、等しければ失敗します。このマクロは、値が何になる**だろう**か確信が持てないけれども、確実にこの値にはなる**べきでない**とわかっているような場合に最も有用になります。例えば、入力を何らかの手段で変え（て出力す）ることが保

証されているけれども、入力の変え方がテストを実行する曜日に依存する関数をテストしているなら、アサーションすべき最善の事柄は、関数の出力が入力と等しくないことかもしれません。

内部的には、assert_eq! と assert_ne! マクロは、それぞれ == と != 演算子を使用しています。アサーションが失敗すると、これらのマクロは引数をデバッグフォーマットを使用してプリントするので、比較対象の値は PartialEq と Debug トレイトを実装していなければなりません。すべての組み込み型と、ほぼすべての標準ライブラリの型はこれらのトレイトを実装しています。自分で定義した構造体や enum については、その型の値の等値性をアサーションするために、PartialEq を実装する必要があるでしょう。それが失敗した時にその値をプリントできるように、Debug も実装する必要もあるでしょう。第 5 章のリスト 5-12 で触れたように、どちらのトレイトも導出可能なトレイトなので、これは通常、単純に構造体や enum 定義に #[derive(PartialEq, Debug)] という注釈を追加するだけですみます。これらやその他の導出可能なトレイトに関する詳細については、付録 C、導出可能なトレイトをご覧ください。

11.1.4　カスタムの失敗メッセージを追加する

さらに、assert! 、assert_eq! 、assert_ne! の追加引数として、失敗メッセージと共にカスタムのメッセージが表示されるよう、追加することもできます。必須引数の後に指定された引数はすべて format! マクロに渡されるので、(format!マクロについては第 8 章の + 演算子、または format! マクロで連結節で議論しました)、{} プレースホルダーを含むフォーマット文字列とこのプレースホルダーに置き換えられる値を渡すことができます。カスタムメッセージは、アサーションがどんな意味を持つかドキュメント化するのに役に立ちます; もしテストが失敗した時、コードにどんな問題があるのかをよりしっかり把握できるはずです。

例として、人々に名前で挨拶をする関数があり、関数に渡した名前が出力に出現することをテストしたいとしましょう:

ファイル名: src/lib.rs

```
pub fn greeting(name: &str) -> String {
    format!("Hello {}!", name)
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn greeting_contains_name() {
        let result = greeting("Carol");
        assert!(result.contains("Carol"));
    }
}
```

このプログラムの要件はまだ取り決められておらず、挨拶の先頭の `Hello` というテキストはおそらく変わります。要件が変わった時にテストを更新しなくてもよいようにしたいと考え、`greeting` 関数から返る値と正確な等値性を確認するのではなく、出力が入力引数のテキストを含むことをアサーションするだけにします。

それでは `greeting` が `name` を含まないように変更してこのコードにバグを仕込み、テストの失敗がデフォルトでどんな風になるのか確かめましょう:

```
pub fn greeting(name: &str) -> String {
    String::from("Hello!")
}
#
# #[cfg(test)]
# mod tests {
#     use super::*;
#
#     #[test]
#     fn greeting_contains_name() {
#         let result = greeting("Carol");
#         assert!(result.contains("Carol"));
#     }
# }
```

このテストを実行すると、以下のように出力されます:

```
$ cargo test
   Compiling greeter v0.1.0 (file:///projects/greeter)
    Finished test [unoptimized + debuginfo] target(s) in 0.91s
     Running unittests src/lib.rs (target/debug/deps/greeter-170b942eb5bf5e3a)

running 1 test
test tests::greeting_contains_name ... FAILED

failures:

---- tests::greeting_contains_name stdout ----
thread 'tests::greeting_contains_name' panicked at src/lib.rs:12:9:
assertion failed: result.contains("Carol")
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::greeting_contains_name

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

error: test failed, to rerun pass `--lib`
```

この結果は、アサーションが失敗し、どの行にアサーションがあるかを示しているだけです。失敗

メッセージが greeting 関数からの値を出力していればより有用でしょう。greeting 関数から得た実際の値で埋められるプレースホルダーを含むフォーマット文字列からなるカスタムの失敗メッセージを追加してみましょう:

```
# pub fn greeting(name: &str) -> String {
#     String::from("Hello!")
# }
#
# #[cfg(test)]
# mod tests {
#     use super::*;
#
    #[test]
    fn greeting_contains_name() {
        let result = greeting("Carol");
        assert!(
            result.contains("Carol"),
            //挨拶(greeting)は名前を含んでいません。その値は`{}`でした
            "Greeting did not contain name, value was `{}`",
            result
        );
    }
# }
```

これでテストを実行したら、より有益なエラーメッセージが得られるでしょう:

```
$ cargo test
   Compiling greeter v0.1.0 (file:///projects/greeter)
    Finished test [unoptimized + debuginfo] target(s) in 0.93s
     Running unittests src/lib.rs (target/debug/deps/greeter-170b942eb5bf5e3a)

running 1 test
test tests::greeting_contains_name ... FAILED

failures:

---- tests::greeting_contains_name stdout ----
thread 'tests::greeting_contains_name' panicked at src/lib.rs:12:9:
Greeting did not contain name, value was `Hello!`
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::greeting_contains_name

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

error: test failed, to rerun pass `--lib`
```

実際に得られた値がテスト出力に表示されているので、起こると想定していたものではなく、起

こったものをデバッグするのに役に立ちます。

11.1.5　should_panic でパニックを確認する

戻り値を確認することに加えて、想定通りにコードがエラー状態を扱っていることを確認することが重要です。例えば、第 9 章のリスト 9-10 で生成した Guess 型を考えてください。Guess を使用する他のコードは、Guess のインスタンスは 1 から 100 の範囲の値しか含まないという保証に依存しています。その範囲外の値で Guess インスタンスを生成しようとするとパニックすることを確認するテストを書くことができます。

これは、テスト関数に should_panic という属性を追加することで達成できます。このテストは、関数内のコードがパニックする場合に通過します。つまり、関数内のコードがパニックしなかったら、テストは失敗するわけです。

リスト 11-8 は、予想どおりに Guess::new のエラー条件が発生していることを確認するテストを示しています。

ファイル名: src/lib.rs

```
pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            //予想値は1から100の間でなければなりませんが、{}でした。
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess { value }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    #[should_panic]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

リスト 11-8: 状況が panic! を引き起こすとテストする

#[test] 属性の後、適用するテスト関数の前に #[should_panic] 属性を配置しています。このテス

トが通るときの結果を見ましょう:

```
$ cargo test
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished test [unoptimized + debuginfo] target(s) in 0.58s
     Running unittests src/lib.rs (target/debug/deps/guessing_game-57
         d70c3acb738f4d)

running 1 test
test tests::greater_than_100 - should panic ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

   Doc-tests guessing_game

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s
```

よさそうですね！ では、値が 100 より大きいときに new 関数がパニックするという条件を除去することでコードにバグを導入しましょう:

```
# pub struct Guess {
#     value: i32,
# }
#
// --snip--
impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 {
            //予想値は1から100の間でなければなりませんが、{}でした。
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess { value }
    }
}
#
# #[cfg(test)]
# mod tests {
#     use super::*;
#
#     #[test]
#     #[should_panic]
#     fn greater_than_100() {
#         Guess::new(200);
#     }
# }
```

リスト 11-8 のテストを実行すると、失敗するでしょう:

```
$ cargo test
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished test [unoptimized + debuginfo] target(s) in 0.62s
     Running unittests src/lib.rs (target/debug/deps/guessing_game-57
         d70c3acb738f4d)

running 1 test
test tests::greater_than_100 - should panic ... FAILED

failures:

---- tests::greater_than_100 stdout ----
note: test did not panic as expected

failures:
    tests::greater_than_100

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

error: test failed, to rerun pass `--lib`
```

この場合、それほど役に立つメッセージは得られませんが、テスト関数に目を向ければ、`#[should_panic]` で注釈されていることがわかります。得られた失敗は、テスト関数のコードがパニックを引き起こさなかったことを意味するのです。

`should_panic` を使用するテストは不正確なこともあります。`should_panic` のテストは、想定していたもの以外の理由でテストがパニックしても通ってしまうのです。`should_panic` のテストの正確を期すために、`should_panic` 属性に `expected` 引数を追加することもできます。このテストハーネスは、失敗メッセージに与えられたテキストが含まれていることを確かめてくれます。例えば、リスト 11-9 の修正された `Guess` のコードを考えてください。ここでは、`new` 関数は、値が大きすぎるか小さすぎるかによって異なるメッセージでパニックします。

ファイル名: src/lib.rs

```
# pub struct Guess {
#     value: i32,
# }
#
// --snip--

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 {
            panic!(
                //予想値は1以上でなければなりませんが、{}でした。
                "Guess value must be greater than or equal to 1, got {}.",
                value
```

```
            );
        } else if value > 100 {
            panic!(
                //予想値は100以下でなければなりませんが、{}でした。
                "Guess value must be less than or equal to 100, got {}.",
                value
            );
        }

        Guess { value }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    //100以下
    #[should_panic(expected = "less than or equal to 100")]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

リスト 11-9: 指定された部分文字列を含むパニックメッセージで panic! することをテストする

should_panic 属性の expected 引数に置いた値が Guess::new 関数がパニックしたメッセージの一部になっているので、このテストは通ります。予想されるパニックメッセージ全体を指定することもでき、今回の場合、Guess value must be less than or equal to 100, got 200. となります。何を指定するかは、パニックメッセージのどこが固有でどこが動的か、またテストをどの程度正確に行いたいかによります。今回の場合、パニックメッセージの一部でも、テスト関数内のコードが、else if value > 100 の場合を実行していると確認するのに事足りるのです。

expected メッセージありの should_panic テストが失敗すると何が起きるのが確かめるために、if value < 1 と else if value > 100 ブロックの本体を入れ替えることで再度コードにバグを仕込みましょう:

```
# pub struct Guess {
#     value: i32,
# }
#
# impl Guess {
#     pub fn new(value: i32) -> Guess {
        if value < 1 {
            panic!(
                //予想値は100以下でなければなりませんが、{}でした。
                "Guess value must be less than or equal to 100, got {}.",
                value
```

```
            );
        } else if value > 100 {
            panic!(
                //予想値は1以上でなければなりませんが、{}でした。
                "Guess value must be greater than or equal to 1, got {}.",
                value
            );
        }
#
#         Guess { value }
#     }
# }
#
# #[cfg(test)]
# mod tests {
#     use super::*;
#
#     #[test]
#     #[should_panic(expected = "less than or equal to 100")]
#     fn greater_than_100() {
#         Guess::new(200);
#     }
# }
```

should_panic テストを実行すると、今回は失敗するでしょう:

```
$ cargo test
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished test [unoptimized + debuginfo] target(s) in 0.66s
     Running unittests src/lib.rs (target/debug/deps/guessing_game-57
         d70c3acb738f4d)

running 1 test
test tests::greater_than_100 - should panic ... FAILED

failures:

---- tests::greater_than_100 stdout ----
thread 'tests::greater_than_100' panicked at src/lib.rs:13:13:
Guess value must be greater than or equal to 1, got 200.
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
note: panic did not contain expected string
      panic message: `"Guess value must be greater than or equal to 1, got
          200."`,
 expected substring: `"less than or equal to 100"`

failures:
    tests::greater_than_100

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s
```

```
error: test failed, to rerun pass `--lib`
```

この失敗メッセージは、このテストが確かに予想通りパニックしたことを示していますが、パニックメッセージは、予想される文字列の `'Guess value must be less than or equal to 100'` を含んでいませんでした。実際に得られたパニックメッセージは今回の場合、`Guess value must be greater than or equal to 1, got 200.` でした。そうしてバグの所在地を割り出し始めることができるわけです！

11.1.6　Result<T, E>をテストで使う

これまで書いてきたテストは失敗するとパニックしていましたが、`Result<T, E>` を使うようなテストを書くこともできます！ 以下は、リスト 11-1 のテストを、`Result<T, E>` を使い、パニックする代わりに `Err` を返すように書き直したものです：

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() -> Result<(), String> {
        if 2 + 2 == 4 {
            Ok(())
        } else {
            Err(String::from("two plus two does not equal four"))
        }
    }
}
```

`it_works` 関数の戻り値の型は `Result<(), String>` になりました。関数内で `assert_eq!` マクロを呼び出す代わりに、テストが成功すれば `Ok(())` を、失敗すれば `Err` に `String` を入れて返すようにします。

`Result<T, E>` を返すようなテストを書くと、`?` 演算子をテストの中で使えるようになります。これは、テスト内で何らかの工程が `Err` ヴァリアントを返したときに失敗するべきテストを書くのに便利です。

`Result<T, E>` を使うテストに `#[should_panic]` 注釈を使うことはできません。操作が `Err` 列挙子を返すことをアサーションするためには、`Result<T, E>` 値に対して `?` 演算子を使用**しないでください**。代わりに、`assert!(value.is_err())` を使用してください。

今やテスト記法を複数知ったので、テストを走らせる際に起きていることに目を向け、`cargo test` で使用できるいろんなオプションを探究しましょう。

11.2　テストの実行のされ方を制御する

`cargo run` がコードをコンパイルし、出来上がったバイナリを走らせるのと全く同様に、`cargo test` はコードをテストモードでコンパイルし、出来上がったテストバイナリを実行します。`cargo test` で

生成されるバイナリの既定動作は、テストを全て並行に実行し、テスト実行中に生成された出力をキャプチャして出力が表示されるのを防ぎ、テスト結果に関係する出力を読みやすくすることです。ですが、コマンドラインオプションを指定してこの既定動作を変更することができます。

コマンドラインオプションの中には `cargo test` にかかるものや、出来上がったテストバイナリにかかるものがあります。この 2 種の引数を区別するために、`cargo test` にかかる引数を `--` という区分記号の後に列挙し、それからテストバイナリにかかる引数を列挙します。`cargo test --help` を走らせると、`cargo test` で使用できるオプションが表示され、`cargo test -- --help` を走らせると、区分記号の後に使えるオプションが表示されます。

11.2.1　テストを並行または連続して実行する

複数のテストを実行するとき、デフォルトでは、テストは複数のスレッドを使用して並行に走ります。つまり、テストが早く実行し終わり、反応をより早く得られることを意味します。テストは同時に実行されるので、テストが他のテストや、共有された環境を含む他の共有状態に依存しないようにする必要があります。現在の作業対象ディレクトリや環境変数などですね。

例えば、各テストがディスクに **test_output.txt** というファイルを作成し、何らかのデータを書き込むコードを走らせるとしてください。そして、各テストはそのファイルのデータを読み取り、ファイルが特定の値を含んでいるとアサーションし、その値は各テストで異なります。テストが同時に走るので、あるテストが、他のテストが書き込んだり読み込んだりする間隙にファイルを上書きするかもしれません。それから 2 番目のテストが失敗します。コードが不正だからではなく、並行に実行されている間にテストがお互いに邪魔をしてしまったせいです。各テストが異なるファイルに書き込むことを確かめるのが一つの解決策です; 別の解決策では、一度に一つのテストを実行します。

並行にテストを実行したくなかったり、使用されるスレッド数をよりきめ細かく制御したい場合、`--test-threads` フラグと使用したいスレッド数をテストバイナリに送ることができます。以下の例に目を向けてください:

```
$ cargo test -- --test-threads=1
```

テストスレッドの数を `1` にセットし、並行性を使用しないようにプログラムに指示しています。1 スレッドのみを使用してテストを実行すると、並行に実行するより時間がかかりますが、状態を共有していても、お互いに邪魔をすることはありません。

11.2.2　関数の出力を表示する

標準では、テストが通ると、Rust のテストライブラリは標準出力に出力されたものを全てキャプチャします。例えば、テストで `println!` を呼び出してテストが通ると、`println!` の出力は、端末に表示されません; テストが通ったことを示す行しか見られないでしょう。テストが失敗すれば、残りの失敗メッセージと共に、標準出力に出力されたものが全て見えるでしょう。

例として、リスト 11-10 は引数の値を出力し、10 を返す馬鹿げた関数と通過するテスト 1 つ、失

敗するテスト 1 つです。

ファイル名: src/lib.rs

```
fn prints_and_returns_10(a: i32) -> i32 {
    //{}という値を得た
    println!("I got the value {}", a);
    10
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn this_test_will_pass() {
        let value = prints_and_returns_10(4);
        assert_eq!(10, value);
    }

    #[test]
    fn this_test_will_fail() {
        let value = prints_and_returns_10(8);
        assert_eq!(5, value);
    }
}
```

リスト 11-10: `println!` を呼び出す関数用のテスト

これらのテストを `cargo test` で実行すると、以下のような出力を目の当たりにするでしょう:

```
$ cargo test
   Compiling silly-function v0.1.0 (file:///projects/silly-function)
    Finished test [unoptimized + debuginfo] target(s) in 0.58s
     Running unittests src/lib.rs (target/debug/deps/silly_function-160869
         f38cff9166)

running 2 tests
test tests::this_test_will_fail ... FAILED
test tests::this_test_will_pass ... ok

failures:

---- tests::this_test_will_fail stdout ----
I got the value 8
thread 'tests::this_test_will_fail' panicked at src/lib.rs:19:9:
assertion `left == right` failed
  left: 5
 right: 10
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

```
failures:
    tests::this_test_will_fail

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

error: test failed, to rerun pass `--lib`
```

この出力のどこにも `I got the value 4` と表示されていないことに注意してください。これは、テストに合格した場合に出力されるものです。その出力はキャプチャされてしまいました。失敗したテストのからの出力 `I got the value 8` がテストサマリー出力のセクションに表示され、テストが失敗した原因も示されます。

通過するテストについても出力される値が見たかったら、`--show-output` で成功するテストの出力も表示するように Rust に指示することができます:

```
$ cargo test -- --show-output
```

リスト 11-10 のテストを `--show-output` フラグと共に再度実行したら、以下のような出力を目の当たりにします:

```
$ cargo test -- --show-output
   Compiling silly-function v0.1.0 (file:///projects/silly-function)
    Finished test [unoptimized + debuginfo] target(s) in 0.60s
     Running unittests src/lib.rs (target/debug/deps/silly_function-160869
         f38cff9166)

running 2 tests
test tests::this_test_will_fail ... FAILED
test tests::this_test_will_pass ... ok

successes:

---- tests::this_test_will_pass stdout ----
I got the value 4


successes:
    tests::this_test_will_pass

failures:

---- tests::this_test_will_fail stdout ----
I got the value 8
thread 'tests::this_test_will_fail' panicked at src/lib.rs:19:9:
assertion `left == right` failed
  left: 5
 right: 10
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

```
failures:
    tests::this_test_will_fail

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

error: test failed, to rerun pass `--lib`
```

11.2.3　名前でテストの一部を実行する

時々、全テストを実行すると時間がかかってしまうことがあります。特定の部分のコードしか対象にしていない場合、そのコードに関わるテストのみを走らせたいかもしれません。`cargo test` に走らせたいテストの名前を引数として渡すことで、実行するテストを選ぶことができます。

テストの一部を走らせる方法を模擬するために、まずリスト 11-11 に示したように `add_two` 関数用に 3 つテストを作成し、そして走らせるテストを選択します。

ファイル名: src/lib.rs

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn add_two_and_two() {
        assert_eq!(4, add_two(2));
    }

    #[test]
    fn add_three_and_two() {
        assert_eq!(5, add_two(3));
    }

    #[test]
    fn one_hundred() {
        assert_eq!(102, add_two(100));
    }
}
```

リスト 11-11: 異なる名前の 3 つのテスト

以前見かけたように、引数なしでテストを走らせたら、全テストが並行に走ります:

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.62s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 3 tests
test tests::add_three_and_two ... ok
test tests::add_two_and_two ... ok
test tests::one_hundred ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s
```

11.2.3.1　単独のテストを走らせる

あらゆるテスト関数の名前を `cargo test` に渡して、そのテストのみを実行することができます:

```
$ cargo test one_hundred
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.69s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::one_hundred ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 2 filtered out;
    finished in 0.00s
```

`one_hundred` という名前のテストだけが走りました; 他の 2 つのテストはその名前に合致しなかったのです。このテスト出力は、最後に `2 filtered out` と表示することによって、実行されなかったテストがあったことを知らせてくれています。

この方法では、複数のテストの名前を指定することはできません; `cargo test` に与えられた最初の値のみが使われるのです。ですが、複数のテストを走らせる方法もあります。

11.2.3.2　複数のテストを実行するようフィルターをかける

テスト名の一部を指定でき、その値に合致するあらゆるテストが走ります。例えば、我々のテストの 2 つが `add` という名前を含むので、`cargo test add` を実行することで、その二つを走らせることができます:

```
$ cargo test add
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.61s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 2 tests
test tests::add_three_and_two ... ok
test tests::add_two_and_two ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out;
    finished in 0.00s
```

このコマンドは名前に add を含むテストを全て実行し、one_hundred という名前のテストを除外しました。また、テストが出現するモジュールがテスト名の一部になっていて、モジュール名でフィルターをかけることで、あるモジュール内のテスト全てを実行できることに注目してください。

11.2.4　特に要望のない限りテストを無視する

時として、いくつかの特定のテストが実行するのに非常に時間がかかることがあり、cargo test の実行のほとんどで除外したくなるかもしれません。引数として確かに実行したいテストを全て列挙するのではなく、ここに示したように代わりに時間のかかるテストを ignore 属性で除外すると注釈することができます。

ファイル名: src/lib.rs

```
#[test]
fn it_works() {
    assert_eq!(2 + 2, 4);
}

#[test]
#[ignore]
fn expensive_test() {
    // 実行に1時間かかるコード
    // code that takes an hour to run
}
```

#[test] の後の除外したいテストに #[ignore] 行を追加しています。これで、テストを実行したら、it_works は実行されるものの、expensive_test は実行されません:

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.60s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 2 tests
test expensive_test ... ignored
test it_works ... ok
```

```
test result: ok. 1 passed; 0 failed; 1 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s
```

expensive_test 関数は、ignored と列挙されています。無視されるテストのみを実行したかったら、cargo test -- --ignored を使うことができます:

```
$ cargo test -- --ignored
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.61s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test expensive_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out;
    finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s
```

どのテストを走らせるか制御することで、結果が早く出ることを確かめることができるのです。ignored テストの結果を確認することが道理に合い、結果を待つだけの時間ができたときに、代わりに cargo test -- --ignored を走らせることができます。無視されているかどうかにかかわらず、すべてのテストを実行したい場合は、cargo test -- --include-ignored で実行することができます。

11.3　テストの体系化

章の初めで触れたように、テストは複雑な鍛錬であり、人によって専門用語や体系化が異なります。Rust のコミュニティでは、テストを 2 つの大きなカテゴリで捉えています: 単体テストと結合テストです。**単体テスト**は小規模でより集中していて、個別に 1 回に 1 モジュールをテストし、非公開のインターフェイスもテストすることがあります。**結合テスト**は、完全にライブラリ外になり、他の外部コード同様に自分のコードを使用し、公開インターフェイスのみ使用し、1 テストにつき複数のモジュールを用いることもあります。

どちらのテストを書くのも、ライブラリの一部が個別かつ共同でしてほしいことをしていることを

確認するのに重要なのです。

11.3.1　単体テスト

単体テストの目的は、残りのコードから切り離して各単位のコードをテストし、コードが想定通り、動いたり動いていなかったりする箇所を迅速に特定することです。単体テストは、テスト対象となるコードと共に、**src** ディレクトリの各ファイルに置きます。慣習は、各ファイルに tests という名前のモジュールを作り、テスト関数を含ませ、そのモジュールを cfg(test) で注釈することです。

11.3.1.1　テストモジュールと#[cfg(test)]

tests モジュールの #[cfg(test)] という注釈は、コンパイラに cargo build を走らせた時ではなく、cargo test を走らせた時にだけ、テストコードをコンパイルし走らせるよう指示します。これにより、ライブラリをビルドしたいだけの時にはコンパイルタイムを節約し、テストが含まれないので、コンパイル後の成果物のサイズも節約します。結合テストは別のディレクトリに存在することになるので、#[cfg(test)] 注釈は必要ないとわかるでしょう。しかしながら、単体テストはコードと同じファイルに存在するので、#[cfg(test)] を使用してコンパイル結果に含まれないよう指定するのです。

この章の最初の節で新しい adder プロジェクトを生成した時に、Cargo がこのコードも生成してくれたことを思い出してください:

ファイル名: src/lib.rs

```
pub fn add(left: usize, right: usize) -> usize {
    left + right
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        let result = add(2, 2);
        assert_eq!(result, 4);
    }
}
```

このコードが自動生成されたテストモジュールです。cfg という属性は、**configuration** を表していて、コンパイラに続く要素が、ある特定の設定オプションを与えられたら、含まれるように指示します。今回の場合、設定オプションは、test であり、言語によって提供されているテストをコンパイルし、走らせるためのものです。cfg 属性を使用することで、cargo test で積極的にテストを実行した場合のみ、Cargo がテストコードをコンパイルします。これには、このモジュールに含まれるかも

しれないヘルパー関数全ても含まれ、`#[test]` で注釈された関数だけにはなりません。

11.3.1.2　非公開関数をテストする

テストコミュニティ内で非公開関数を直接テストするべきかについては議論があり、他の言語では非公開関数をテストするのは困難だったり、不可能だったりします。あなたがどちらのテストイデオロギーを支持しているかに関わらず、Rust の公開性規則により、非公開関数をテストすることが確かに可能です。リスト 11-12 の非公開関数 `internal_adder` を含むコードを考えてください。

ファイル名: src/lib.rs

```
pub fn add_two(a: i32) -> i32 {
    internal_adder(a, 2)
}

fn internal_adder(a: i32, b: i32) -> i32 {
    a + b
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn internal() {
        assert_eq!(4, internal_adder(2, 2));
    }
}
```

リスト 11-12: 非公開関数をテストする

`internal_adder` 関数は `pub` とマークされていないことに注意してください。テストも単なる Rust のコードであり、`tests` モジュールもただのモジュールでしかありません。「モジュールツリーの要素を示すためのパス」節で議論したように、子モジュール内の要素はその祖先モジュール内の要素を使用することができます。このテストでは、`use super::*` によって `test` モジュールの親のすべての要素をスコープ内に持ち込み、そしてテストが `internal_adder` を呼び出しています。非公開関数はテストするべきではないとお考えなら、Rust にはそれを強制するものは何もありません。

11.3.2　結合テスト

Rust において、結合テストは完全にライブラリ外のものです。他のコードと全く同様にあなたのライブラリを使用するので、ライブラリの公開 API の一部である関数しか呼び出すことはできません。その目的は、ライブラリのいろんな部分が共同で正常に動作しているかをテストすることです。単体では正常に動くコードも、結合した状態だと問題を孕む可能性もあるので、結合したコードのテストの範囲も同様に重要になるのです。結合テストを作成するには、まず **tests** ディレクトリが必要にな

ります。

11.3.2.1　tests ディレクトリ

プロジェクトディレクトリのトップ階層、**src** の隣に **tests** ディレクトリを作成します。Cargo は、このディレクトリに結合テストのファイルを探すことを把握しています。テストファイルは好きなだけ作成することができ、Cargo はそれぞれのファイルを個別のクレートとしてコンパイルします。

結合テストを作成しましょう。リスト 11-12 のコードが **src/lib.rs** ファイルにあるまま、**tests** ディレクトリを作成し、**tests/integration_test.rs** という名前の新しいファイルを生成してください。ディレクトリ構造は次のようになるはずです:

```
adder
├── Cargo.lock
├── Cargo.toml
├── src
│   └── lib.rs
└── tests
    └── integration_test.rs
```

リスト 11-13 のコードを **tests/integration_test.rs** ファイルに入力してください:

ファイル名: tests/integration_test.rs

```
use adder;

#[test]
fn it_adds_two() {
    assert_eq!(4, adder::add_two(2));
}
```

リスト 11-13: `adder` クレートの関数の結合テスト

`tests` ディレクトリのファイルはそれぞれ個別のクレートであるため、ライブラリを各テストのスコープ内に持ち込む必要があります。そのため、コードの先頭に `use adder` を追記していますが、これは単体テストでは必要なかったものです。

tests/integration_test.rs のどんなコードも `#[cfg(test)]` で注釈する必要はありません。Cargo は `tests` ディレクトリを特別に扱い、`cargo test` を走らせた時にのみこのディレクトリのファイルをコンパイルするのです。さあ、`cargo test` を実行してください:

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 1.31s
     Running unittests src/lib.rs (target/debug/deps/adder-1082c4b063a8fbe6)

running 1 test
test tests::internal ... ok
```

```
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

     Running tests/integration_test.rs (target/debug/deps/integration_test-1082
         c4b063a8fbe6)

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s
```

3 つの区域の出力が単体テスト、結合テスト、doc テストを含んでいます。ある区域内のテストがひとつでも失敗していれば、それ以降の区域は実行されないでしょう。例えば、単体テストが失敗したら、結合テストと doc テストについての出力はされないでしょう。すべての単体テストが通過する場合のみ、これらのテストが実行されるからです。

単体テスト用の最初の区域は、今まで見てきたものと同じです: 各単体テストに 1 行 (リスト 11-12 で追加した `internal` という名前のもの) と、単体テストのサマリー行です。

結合テストの区域は、`Running tests/integration_test.rs` という行で始まっています。次に、この結合テストの各テスト関数用の行があり、`Doc-tests adder` 区域が始まる直前に、結合テストの結果用のサマリー行があります。

結合テストファイルはそれぞれ独自の区域があるため、**tests** ディレクトリにさらにファイルを追加すれば、結合テストの区域が増えることになるでしょう。

それでも、テスト関数の名前を引数として `cargo test` に指定することで、特定の結合テスト関数を走らせることができます。特定の結合テストファイルにあるテストを全て走らせるには、`cargo test` に `--test` 引数、その後にファイル名を続けて使用してください:

```
$ cargo test --test integration_test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.64s
     Running tests/integration_test.rs (target/debug/deps/integration_test-82
         e7799c1bc62298)

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s
```

このコマンドは、**tests/integration_test.rs** ファイルにあるテストのみを実行します。

11.3.2.2　結合テスト内のサブモジュール

結合テストを追加するにつれて、**tests** ディレクトリにファイルを作成して体系化したくなるかもしれません; 例えば、テスト対象となる機能でテスト関数をグループ化することができます。前述したように、**tests** ディレクトリの各ファイルは、個別のクレートとしてコンパイルされます。これは、エンドユーザがあなたのクレートを使用する場合をより模倣するように個別のスコープを生成するのに役立ちます。ですが、これは **tests** ディレクトリのファイルが、コードをモジュールとファイルに分ける方法に関して第 7 章で学んだように、**src** のファイルとは同じ振る舞いを共有しないことを意味します。

tests ディレクトリのファイルの異なる振る舞いは、複数の結合テストファイルで使用するヘルパー関数ができ、第 7 章の「モジュールを複数のファイルに分割する」節の手順に従って共通モジュールに抽出しようとした時に最も気付きやすくなります。例えば、**tests/common.rs** を作成し、そこに `setup` という名前の関数を配置したら、複数のテストファイルの複数のテスト関数から呼び出したい `setup` に何らかのコードを追加することができます:

ファイル名: tests/common.rs

```
pub fn setup() {
    // ここにライブラリテスト固有のコードが来る
    // setup code specific to your library's tests would go here
}
```

再度テストを実行すると、**common.rs** ファイルは何もテスト関数を含んだり、`setup` 関数をどこかから呼んだりしてないのに、テスト出力に **common.rs** 用の区域が見えるでしょう。

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.89s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::internal ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

     Running tests/common.rs (target/debug/deps/common-92948b65e88960b4)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

     Running tests/integration_test.rs (target/debug/deps/integration_test-92948
```

```
         b65e88960b4)

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s
```

common が running 0 tests とテスト結果に表示されるのは、望んだ結果ではありません。ただ単に他の結合テストファイルと何らかのコードを共有したかっただけです。

common がテスト出力に出現するのを防ぐには、**tests/common.rs** を作成する代わりに、**tests/common/mod.rs** を作成します。これでプロジェクトディレクトリは次のようになります:

```
├── Cargo.lock
├── Cargo.toml
├── src
│   └── lib.rs
└── tests
    ├── common
    │   └── mod.rs
    └── integration_test.rs
```

これは第 7 章の「別のファイルパス」節で言及した、コンパイラが理解するもう一つの古い命名規則です。このように命名することでコンパイラに common モジュールを結合テストファイルとして扱わないように指示します。setup 関数のコードを **tests/common/mod.rs** に移動し、**tests/common.rs** ファイルを削除すると、テスト出力に区域はもう表示されなくなります。**tests** ディレクトリのサブディレクトリ内のファイルは個別クレートとしてコンパイルされたり、テスト出力に区域が表示されることがないのです。

tests/common/mod.rs を作成した後、それをどの結合テストファイルからもモジュールとして使用することができます。こちらは、**tests/integration_test.rs** 内の it_adds_two テストから setup 関数を呼び出す例です:

ファイル名: tests/integration_test.rs

```
use adder;

mod common;

#[test]
fn it_adds_two() {
```

```
    common::setup();
    assert_eq!(4, adder::add_two(2));
}
```

`mod common;` という宣言は、リスト 7-21 で模擬したモジュール宣言と同じであることに注意してください。それから、テスト関数内で `common::setup()` 関数を呼び出すことができます。

11.3.2.3　バイナリクレート用の結合テスト

もしもプロジェクトが **src/main.rs** ファイルのみを含み、**src/lib.rs** ファイルを持たないバイナリクレートだったら、**tests** ディレクトリに結合テストを作成し、use 文で **src/main.rs** ファイルに定義された関数をスコープ内に持ち込むことはできません。ライブラリクレートのみが、他のクレートが使用できる関数を晒せるのです; バイナリクレートはそれ単体で実行することを意味しています。

これは、バイナリを提供する Rust のプロジェクトに、**src/lib.rs** ファイルに存在するロジックを呼び出す単純な **src/main.rs** ファイルがある一因になっています。この構造を使用して結合テストは、use で重要な機能を利用できるようにすることでライブラリクレートをテストすることが**できます**。この重要な機能が動作すれば、**src/main.rs** ファイルの少量のコードも動作し、その少量のコードはテストする必要がないわけです。

11.4　まとめ

Rust のテスト機能は、変更を加えた後でさえ想定通りにコードが機能し続けることを保証して、コードが機能すべき方法を指定する手段を提供します。単体テストはライブラリの異なる部分を個別に用い、非公開の実装詳細をテストすることができます。結合テストは、ライブラリのいろんな部分が共同で正常に動作することを確認し、ライブラリの公開 API を使用して外部コードが使用するのと同じ方法でコードをテストします。Rust の型システムと所有権ルールにより防がれるバグの種類もあるものの、それでもテストは、コードが振る舞うと予想される方法に関するロジックのバグを減らすのに重要なのです。

この章と以前の章で学んだ知識を結集して、とあるプロジェクトに取り掛かりましょう！

第 12 章

入出力プロジェクト: コマンドラインプログラムを構築する

この章は、ここまでに学んできた多くのスキルを思い出すきっかけであり、もういくつか標準ライブラリの機能も探究します。ファイルやコマンドラインの入出力と相互作用するコマンドラインツールを構築し、今やあなたの支配下にある Rust の概念の一部を練習していきます。

Rust の速度、安全性、単バイナリ出力、クロスプラットフォームサポートにより、コマンドラインツールを作るのにふさわしい言語なので、このプロジェクトでは、独自の伝統的なコマンドライン検索ツールの `grep` (**g**lobally search a **r**egular **e**xpression and **p**rint: 正規表現をグローバルで検索し表示する) を作成していきます。最も単純な使用法では、`grep` は指定したファイルから指定した文字列を検索します。そうするには、`grep` は引数としてファイルパスと文字列を受け取ります。それからファイルを読み込んでそのファイル内で文字列引数を含む行を探し、検索した行を出力するのです。

その過程で、他の多くのコマンドラインツールが使用している端末の機能を使用させる方法を示します。環境変数の値を読み取ってユーザがこのツールの振る舞いを設定できるようにします。また、エラーメッセージを標準出力 (`stdout`) の代わりに標準エラー (`stderr`) に出力するので、例えば、ユーザはエラーメッセージは画面上で確認しつつ、成功した出力はファイルにリダイレクトできます。

Rust コミュニティのあるメンバであるアンドリュー・ガラント (Andrew Gallant) が既に全機能装備の非常に高速な `grep` 、`ripgrep` と呼ばれるものを作成しました。比較対象として、我々のバージョンはとても単純ですが、この章により、`ripgrep` のような現実世界のプロジェクトを理解するのに必要な背景知識の一部を身に付けられるでしょう。

この `grep` プロジェクトは、ここまでに学んできた多くの概念を集結させます:

- コードを体系化する (第 7 章でモジュールについて学んだことを使用)
- ベクタと文字列を使用する (コレクション、第 8 章)
- エラーを処理する (第 9 章)

- 適切な箇所でトレイトとライフタイムを使用する (第 10 章)
- テストを記述する (第 11 章)

さらに、クロージャ、イテレータ、トレイトオブジェクトなど、第 13 章、第 17 章で詳しく講義するものもちょっとだけ紹介します。

12.1　コマンドライン引数を受け付ける

いつものように、cargo new で新しいプロジェクトを作りましょう。プロジェクトを minigrep と名付けて、既に自分のシステムに存在するかもしれない grep ツールと区別しましょう。

```
$ cargo new minigrep
     Created binary (application) `minigrep` project
$ cd minigrep
```

最初の仕事は、minigrep を二つの引数を受け付けるようにすることです: ファイルパスと検索する文字列ですね。つまり、cargo run 、以降の引数が cargo に対するものではなく自分のプログラムに対するものであることを示すハイフン 2 個、検索文字列、そして検索を行うファイルへのパスを渡して、プログラムを実行できるようになりたいということです。こんな感じにね:

```
$ cargo run -- searchstring example-filename.txt
```

今現在は、cargo new で生成されたプログラムは、与えた引数を処理できません。crates.io に存在する既存のライブラリには、コマンドライン引数を受け付けるプログラムを書く手助けをしてくれるものもありますが、ちょうどこの概念を学んでいる最中なので、この能力を自分で実装しましょう。

12.1.1　引数の値を読み取る

minigrep が渡したコマンドライン引数の値を読み取れるようにするために、Rust の標準ライブラリで提供されている std::env::args 関数が必要になります。この関数は、minigrep に渡されたコマンドライン引数のイテレータを返します。イテレータについては第 13 章で詳しく講義します。とりあえずイテレータに関しては、2 つの詳細のみ知っていればいいです: イテレータは一連の値を生成することと、イテレータに対して collect メソッドを呼び出し、イテレータが生成する要素全部を含むベクタなどのコレクションに変えられることです。

リスト 12-1 のコードを使用することで、minigrep プログラムに渡されたあらゆるコマンドライン引数を読み取り、それからその値をベクタに集めることができます。

ファイル名: src/main.rs

```
use std::env;

fn main() {
```

```
    let args: Vec<String> = env::args().collect();
    dbg!(args);
}
```

リスト 12-1: コマンドライン引数をベクタに集結させ、出力する

まず、std::env モジュールを use 文でスコープに導入したので、args 関数が使用できます。std::env::args 関数は、2 レベルモジュールがネストされていることに注目してください。第 7 章で議論したように、希望の関数が 2 モジュール以上ネストされている場合、関数ではなく親モジュールをスコープに導入しています。そうすることで、std::env から別の関数も容易に使用することができます。また、use std::env::args を追記し、関数を args とするだけで呼び出すのに比べて曖昧でもありません。というのも、args は現在のモジュールに定義されている関数と容易に見間違えられるかもしれないからです。

12.1.2　args 関数と不正なユニコード

> 引数のどれかが不正なユニコードを含んでいたら、std::env::args はパニックすることに注意してください。プログラムが不正なユニコードを含む引数を受け付ける必要があるなら、代わりに std::env::args_os を使用してください。この関数は、String 値ではなく、OsString 値を生成するイテレータを返します。ここでは、簡潔性のために std::env::args を使うことを選択しました。なぜなら、OsString 値はプラットフォームごとに異なり、String 値に比べて取り扱いが煩雑だからです。

main の最初の行で env::args を呼び出し、そして即座に collect を使用して、イテレータをイテレータが生成する値全てを含むベクタに変換しています。collect 関数を使用して多くの種類のコレクションを生成することができるので、args の型を明示的に注釈して文字列のベクタが欲しいのだと指定しています。Rust において、型を注釈しなければならない頻度は非常に少ないのですが、collect はよく確かに注釈が必要になる一つの関数なのです。コンパイラには、あなたが欲しているコレクションの種類が推論できないからです。

最後に、デバッグマクロを使用してベクタを出力しています。まずは引数なしでコードを走らせてみて、それから引数二つで試してみましょう:

```
$ cargo run
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.61s
     Running `target/debug/minigrep`
[src/main.rs:5:5] args = [
    "target/debug/minigrep",
]
```

```
$ cargo run -- needle haystack
```

```
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 1.57s
     Running `target/debug/minigrep needle haystack`
[src/main.rs:5:5] args = [
    "target/debug/minigrep",
    "needle",
    "haystack",
]
```

ベクタの最初の値は `"target/debug/minigrep"` であることに注目してください。これはバイナリの名前です。これは C の引数リストの振る舞いと合致し、実行時に呼び出された名前をプログラムに使わせてくれるわけです。メッセージで出力したり、プログラムを起動するのに使用されたコマンドラインエイリアスによってプログラムの振る舞いを変えたい場合に、プログラム名にアクセスするのにしばしば便利です。ですが、この章の目的には、これを無視し、必要な二つの引数のみを保存します。

12.1.3　引数の値を変数に保存する

現時点でプログラムはコマンドライン引数として指定された値にアクセスできます。さて、プログラムの残りを通して使用できるように、二つの引数の値を変数に保存する必要があります。それをしているのがリスト 12-2 です。

ファイル名: src/main.rs

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();

    let query = &args[1];
    let file_path = &args[2];

    //       "{}を検索します"
    println!("Searching for {}", query);
    //       "ファイル{}の中で"
    println!("In file {}", file_path);
}
```

リスト 12-2: クエリ引数とファイルパス引数を保持する変数を生成

ベクタを出力した時に確認したように、プログラム名がベクタの最初の値、`args[0]` を占めているので、引数を添え字 `1` から始めます。`minigrep` が取る最初の引数は、検索する文字列なので、最初の引数への参照を変数 `query` に置きました。2 番目の引数はファイルパスでしょうから、2 番目の引数への参照は変数 `file_path` に置きました。

一時的にこれらの変数の値を出力して、コードが意図通りに動いていることを証明しています。再度このプログラムを `test` と `sample.txt` という引数で実行しましょう:

```
$ cargo run -- test sample.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/minigrep test sample.txt`
Searching for test
In file sample.txt
```

素晴らしい、プログラムは動作しています！ 必要な引数の値が、正しい変数に保存されています。後ほど、何らかのエラー処理を加えて、ユーザが引数を提供しなかった場合など、可能性のある特定のエラー状況に対処します; 今は、そのような状況はないものとし、代わりにファイル読み取り能力を追加することに取り組みます。

12.2　ファイルを読み込む

では、`file_path` 引数で指定されたファイルを読み込む機能を追加しましょう。まず、テスト実行するためのサンプルファイルが必要ですね: 複数行にわたって、同じ単語の繰り返しのある、少量のテキストを含むようなファイルを使いましょう。リスト 12-3 は、ちょうど良さそうなエミリー・ディキンソン (Emily Dickinson) の詩です！ プロジェクトのルート階層に **poem.txt** というファイルを作成し、この詩 “I’ m Nobody! Who are you?” を入力してください。

ファイル名: poem.txt

```
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us - don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

リスト 12-3: エミリー・ディキンソンの詩は、いいテストケースになる

テキストを適当な場所に置いて、**src/main.rs** を編集し、ファイルを読むコードを追加してください。リスト 12-4 に示したようにですね。

ファイル名: src/main.rs

```
use std::env;
use std::fs;

fn main() {
    // --snip--
#     let args: Vec<String> = env::args().collect();
```

```
#
#     let query = &args[1];
#     let file_path = &args[2];
#
#     println!("Searching for {}", query);
    println!("In file {}", file_path);

    let contents = fs::read_to_string(file_path)
        //      "ファイルを読み込むことができるはずでした"
        .expect("Should have been able to read the file");

    //       "テキスト:\n{contents}"
    println!("With text:\n{contents}");
}
```

リスト 12-4: 第 2 引数で指定されたファイルの中身を読み込む

最初に、use 文で、標準ライブラリの関係のある箇所を持ってきています: ファイルを扱うのには std::fs が必要です。

main に追加した fs::read_to_string 文は、file_path を受け取り、そのファイルを開き、そのファイルの内容を含む std::io::Result<String> を返します。

その後、今回もファイル読み込み後に contents の値を出力する一時的な println! 文を追記しているので、ここまでプログラムがきちんと動作していることを確認できます。

第 1 コマンドライン引数には適当な文字列 (まだ検索する箇所は実装してませんからね) を、第 2 引数に **poem.txt** ファイルを入れて、このコードを実行しましょう:

```
$ cargo run -- the poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/minigrep the poem.txt`
Searching for the
In file poem.txt
With text:
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us - don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

素晴らしい！ コードがファイルの中身を読み込み、出力するようになりました。しかし、このコードにはいくつか欠陥があります。現時点で、main 関数は複数の責任を受け持っています: 一般に、各関数がただ一つの責任だけを持つようになれば、関数は明確かつ、管理しやすくなります。もう一つの問題点は、できうる限りのエラー処理を怠っていることです。まだプログラムが小規模なので、こ

れらの欠陥は大きな問題にはなりませんが、プログラムが大規模になるにつれ、それを綺麗に解消するのは困難になっていきます。プログラムを開発する際に早い段階でリファクタリングを行うのは、良い戦術です。リファクタリングするコードの量が少なければ、はるかに簡単になりますからね。次は、それを行いましょう。

12.3　リファクタリングしてモジュール性とエラー処理を向上させる

プログラムを改善するために、プログラムの構造と起こりうるエラーに対処する方法に関連する 4 つの問題を修正していきましょう。1 番目は、`main` 関数が 2 つの仕事を受け持っていることです: 引数を解析し、ファイルを読んでいます。プログラムが大きくなるにつれて、`main` 関数が扱う個別の仕事の数も増えていきます。関数が責任を受け持つごとに、正しいことを確認しにくくなり、テストも行いづらくなり、機能を壊さずに変更するのも困難になっていきます。機能を小分けして、各関数が 1 つの仕事のみに責任を持つようにするのが最善です。

この問題は、2 番目の問題にも結びついています: `query` と `file_path` はプログラムの設定用変数ですが、`contents` などの変数は、プログラムのロジックを担っています。`main` が長くなるほど、スコープに入れるべき変数も増えます。そして、スコープにある変数が増えれば、各々の目的を追うのも大変になるわけです。設定用変数を一つの構造に押し込め、目的を明瞭化するのが最善です。

3 番目の問題は、ファイルを読み込み損ねた時に `expect` を使ってエラーメッセージを出力しているのに、エラーメッセージが `Should have been able to read the file` としか表示しないことです。ファイルを読む行為は、いろんな方法で失敗することがあります: 例えば、ファイルが無いのかもしれないし、それを開く権限が無いのかもしれません。現時点では、状況にかかわらず、すべてに対して同一のエラーメッセージを出力していて、これではユーザに何の情報も与えていないでしょう！

4 番目は、異なるエラーを処理するのに `expect` を繰り返し使用しているので、ユーザが十分な数の引数を渡さずにプログラムを起動した時に、問題を明確に説明しない「範囲外アクセス (index out of bounds)」というエラーが Rust から得られることです。エラー処理のコードが全て 1 箇所に存在し、将来エラー処理ロジックが変更になった時に、メンテナンス者が 1 箇所のコードのみを考慮すればいいようにするのが最善でしょう。エラー処理コードが 1 箇所にあれば、エンドユーザにとって意味のあるメッセージを出力していることを確認することにもつながります。

プロジェクトをリファクタリングして、これら 4 つの問題を扱いましょう。

12.3.1　バイナリプロジェクトの責任の分離

`main` 関数に複数の仕事の責任を割り当てるという構造上の問題は、多くのバイナリプロジェクトでありふれています。結果として、`main` が肥大化し始めた際にバイナリプログラムの個別の責任を分割するためのガイドラインを Rust コミュニティは、開発しました。この工程は、以下のような手順になっています:

- プログラムを **main.rs** と **lib.rs** に分け、ロジックを **lib.rs** に移動する。

- コマンドライン引数の解析ロジックが小規模な限り、**main.rs** に置いても良い。
- コマンドライン引数の解析ロジックが複雑化の様相を呈し始めたら、**main.rs** から抽出して **lib.rs** に移動する。

この工程の後に `main` 関数に残る責任は以下に限定される:

- 引数の値でコマンドライン引数の解析ロジックを呼び出す
- 他のあらゆる設定を行う
- **lib.rs** の `run` 関数を呼び出す
- `run` がエラーを返した時に処理する

このパターンは、責任の分離についてです: **main.rs** はプログラムの実行を行い、そして、**lib.rs** が手にある仕事のロジック全てを扱います。`main` 関数を直接テストすることはできないので、この構造により、プログラムのロジック全てを **lib.rs** の関数に移すことでテストできるようになります。**main.rs** に残るコードは、読めばその正当性が評価できるだけ小規模になるでしょう。この工程に従って、プログラムのやり直しをしましょう。

12.3.1.1　引数解析器を抽出する

引数解析の機能を `main` が呼び出す関数に抽出して、コマンドライン引数解析ロジックを **src/lib.rs** に移動する準備をします。リスト 12-5 に新しい関数 `parse_config` を呼び出す `main` の冒頭部を示し、この新しい関数は今だけ **src/main.rs** に定義します。

ファイル名: src/main.rs

```
# use std::env;
# use std::fs;
#
fn main() {
    let args: Vec<String> = env::args().collect();

    let (query, file_path) = parse_config(&args);

    // --snip--
#
#     println!("Searching for {}", query);
#     println!("In file {}", file_path);
#
#     let contents = fs::read_to_string(file_path)
#         .expect("Should have been able to read the file");
#
#     println!("With text:\n{contents}");
}

fn parse_config(args: &[String]) -> (&str, &str) {
    let query = &args[1];
    let file_path = &args[2];
```

```
    (query, file_path)
}
```

リスト 12-5: main から parse_config 関数を抽出する

それでもまだ、コマンドライン引数をベクタに集結させていますが、main 関数内で引数の値の添え字 1 を変数 query に、添え字 2 を変数 file_path に代入する代わりに、ベクタ全体を parse_config 関数に渡しています。そして、parse_config 関数にはどの引数がどの変数に入り、それらの値を main に返すというロジックが存在します。まだ main 内に query と file_path という変数を生成していますが、もう main は、コマンドライン引数と変数がどう対応するかを決定する責任は持ちません。

このやり直しは、私たちの小規模なプログラムにはやりすぎに思えるかもしれませんが、少しずつ段階的にリファクタリングしているのです。この変更後、プログラムを再度実行して、引数解析がまだ動作していることを実証してください。問題が発生した時に原因を特定する助けにするために頻繁に進捗を確認するのはいいことです。

12.3.1.2　設定値をまとめる

もう少し parse_config 関数を改善することができます。現時点では、タプルを返していますが、即座にタプルを分解して再度個別の値にしています。これは、正しい抽象化をまだできていないかもしれない兆候です。

まだ改善の余地があると示してくれる他の徴候は、parse_config の config の部分であり、返却している二つの値は関係があり、一つの設定値の一部にどちらもなることを暗示しています。現状では、一つのタプルにまとめていること以外、この意味をデータの構造に載せていません; この二つの値を 1 構造体に置き換え、構造体のフィールドそれぞれに意味のある名前をつけましょう。そうすることで将来このコードのメンテナンス者が、異なる値が相互に関係する仕方や、目的を理解しやすくできるでしょう。

リスト 12-6 は、parse_config 関数の改善を示しています。

ファイル名: src/main.rs

```
# use std::env;
# use std::fs;
#
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = parse_config(&args);

    println!("Searching for {}", config.query);
    println!("In file {}", config.file_path);

    let contents = fs::read_to_string(config.file_path)
```

```
        .expect("Should have been able to read the file");

    // --snip--
#
#     println!("With text:\n{contents}");
}

struct Config {
    query: String,
    file_path: String,
}

fn parse_config(args: &[String]) -> Config {
    let query = args[1].clone();
    let file_path = args[2].clone();

    Config { query, file_path }
}
```

リスト 12-6: `parse_config` をリファクタリングして `Config` 構造体のインスタンスを返す

`query` と `file_path` というフィールドを持つよう定義された `Config` という構造体を追加しました。`parse_config` のシグニチャは、これで `Config` 値を返すと示すようになりました。`parse_config` の本体では、以前は `args` の `String` 値を参照する文字列スライスを返していましたが、今では所有する `String` 値を含むように `Config` を定義しています。`main` の `args` 変数は引数値の所有者であり、`parse_config` 関数だけに借用させていますが、これは `Config` が `args` の値の所有権を奪おうとしたら、Rust の借用規則に違反してしまうことを意味します。

`String` のデータを管理する方法は複数あります; 最も単純だけれども多少非効率的な手段は、値に対して `clone` メソッドを呼び出すことです。これにより、`Config` インスタンスが所有するデータの総コピーが生成されるので、文字列データへの参照を保持するよりも時間とメモリを消費します。ですが、データをクローンすることで、コードがとても素直にもなります。というのも、参照のライフタイムを管理する必要がないからです。つまり、この場面において、少々のパフォーマンスを犠牲にして単純性を得るのは、価値のある代償です。

12.3.2　`clone` を使用する代償

実行時コストのために `clone` を使用して所有権問題を解消するのを避ける傾向が多くの Rustacean にあります。第 13 章で、この種の状況においてより効率的なメソッドの使用法を学ぶでしょう。ですがとりあえずは、これらのコピーをするのは 1 回だけですし、ファイルパスとクエリ文字列は非常に小さなものなので、いくつかの文字列をコピーして進捗するのは良しとしましょう。最初の通り道でコードを究極的に効率化しようとするよりも、ちょっと非効率的でも動くプログラムを用意する方がいいでしょう。もっと Rust の経験を積めば、最も効

> 率的な解決法から開始することも簡単になるでしょうが、今は、clone を呼び出すことは完璧に受け入れられることです。

main を更新したので、parse_config から返された Config のインスタンスを config という変数に置くようになり、以前は個別の query と file_path 変数を使用していたコードを更新したので、代わりに Config 構造体のフィールドを使用するようになりました。

これでコードは query と file_path が関連していることと、その目的がプログラムの振る舞い方を設定するということをより明確に伝えます。これらの値を使用するあらゆるコードは、config インスタンスの目的の名前を冠したフィールドにそれらを発見することを把握しています。

12.3.2.1　Config のコンストラクタを作成する

ここまでで、コマンドライン引数を解析する責任を負ったロジックを main から抽出し、parse_config 関数に配置しました。そうすることで query と file_path の値が関連し、その関係性がコードに載っていることを確認する助けになりました。それから Config 構造体を追加して query と file_path の関係する目的を名前付けし、構造体のフィールド名として parse_config 関数からその値の名前を返すことができています。

したがって、今や parse_config 関数の目的は Config インスタンスを生成することになったので、parse_config をただの関数から Config 構造体に紐づく new という関数に変えることができます。この変更を行うことで、コードがより慣用的になります。String などの標準ライブラリの型のインスタンスを、String::new を呼び出すことで生成できます。同様に、parse_config を Config に紐づく new 関数に変えれば、Config::new を呼び出すことで Config のインスタンスを生成できるようになります。リスト 12-7 が、行う必要のある変更を示しています。

ファイル名: src/main.rs

```
# use std::env;
# use std::fs;
#
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::new(&args);
#
#     println!("Searching for {}", config.query);
#     println!("In file {}", config.file_path);
#
#     let contents = fs::read_to_string(config.file_path)
#         .expect("Should have been able to read the file");
#
#     println!("With text:\n{contents}");

    // --snip--
}
```

```
// --snip--

# struct Config {
#     query: String,
#     file_path: String,
# }
#
impl Config {
    fn new(args: &[String]) -> Config {
        let query = args[1].clone();
        let file_path = args[2].clone();

        Config { query, file_path }
    }
}
```

リスト 12-7: parse_config を Config::new に変える

parse_config を呼び出していた main を代わりに Config::new を呼び出すように更新しました。parse_config の名前を new に変え、impl ブロックに入れ込んだので、new 関数と Config が紐づくようになりました。再度このコードをコンパイルしてみて、動作することを確かめてください。

12.3.3　エラー処理を修正する

さて、エラー処理の修正に取り掛かりましょう。ベクタが 2 個以下の要素しか含んでいないときに args ベクタの添え字 1 か 2 にアクセスしようとすると、プログラムがパニックすることを思い出してください。試しに引数なしでプログラムを実行してください。すると、こんな感じになります:

```
$ cargo run
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/minigrep`
thread 'main' panicked at src/main.rs:27:21:
index out of bounds: the len is 1 but the index is 1
(スレッド'main'はsrc/main.rs:27:21でパニックしました:
境界外アクセス: 長さは1なのに添え字も1です)
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

境界外アクセス: 長さは1なのに添え字も1です という行は、プログラマ向けのエラーメッセージです。エンドユーザが代わりにすべきことを理解する手助けにはならないでしょう。これを今修正しましょう。

12.3.3.1　エラーメッセージを改善する

リスト 12-8 で、new 関数に、添え字 1 と 2 にアクセスする前にスライスが十分長いことを実証するチェックを追加しています。スライスの長さが十分でなければ、プログラムはパニックし、より良いエラーメッセージを表示します。

ファイル名: src/main.rs

```
# use std::env;
# use std::fs;
#
# fn main() {
#     let args: Vec<String> = env::args().collect();
#
#     let config = Config::new(&args);
#
#     println!("Searching for {}", config.query);
#     println!("In file {}", config.file_path);
#
#     let contents = fs::read_to_string(config.file_path)
#         .expect("Should have been able to read the file");
#
#     println!("With text:\n{contents}");
# }
#
# struct Config {
#     query: String,
#     file_path: String,
# }
#
# impl Config {
    // --snip--
    fn new(args: &[String]) -> Config {
        if args.len() < 3 {
            //     "引数が足りません"
            panic!("not enough arguments");
        }
        // --snip--
#
#         let query = args[1].clone();
#         let file_path = args[2].clone();
#
#         Config { query, file_path }
#     }
# }
```

リスト 12-8: 引数の数のチェックを追加する

このコードは、value 引数が正常な値の範囲外だった時に panic! を呼び出していた、リスト 9-13 の中で書いた Guess::new 関数と似ています。ここでは、値の範囲を確かめる代わりに、args の長さが少なくとも 3 であることを確かめていて、関数の残りの部分は、この条件が満たされているという前提のもとで処理を行うことができます。args に 2 要素以下しかなければ、この条件は真になり、panic! マクロを呼び出して、即座にプログラムを終了させます。

では、new のこの追加の数行がある状態で、再度引数なしでプログラムを走らせ、エラーがどんな見た目か確かめましょう:

```
$ cargo run
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/minigrep`
thread 'main' panicked at src/main.rs:26:13:
not enough arguments
(スレッド'main'はsrc/main.rs:26:13でパニックしました:
引数が足りません)
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

この出力の方がマシです: これでエラーメッセージが合理的になりました。ですが、ユーザに与えたくない追加の情報も含まれてしまっています。おそらく、ここではリスト 9-13 で使用したテクニックを使用するのは最善ではありません: `panic!` の呼び出しは、第 9 章で議論したように、使用の問題よりもプログラミング上の問題により適しています。代わりに、第 9 章で学んだもう一つのテクニックを使用しましょう。成功か失敗かを示唆する `Result` を返すことです。

12.3.3.2　`panic!` を呼び出す代わりに `Result` を返す

代わりに、成功時には `Config` インスタンスを含み、エラー時には問題に言及する `Result` 値を返すことができます。また、多くのプログラマは `new` 関数は決して失敗しないことを期待するので、関数名を `new` から `build` に変更します。`Config::build` が `main` と対話する時、`Result` 型を使用して問題があったと信号を送ることができます。それから `main` を変更して、`panic!` 呼び出しが引き起こしていた `thread 'main'` と `RUST_BACKTRACE` に関する周囲のテキストがない、ユーザ向けのより実用的なエラーに `Err` 列挙子を変換することができます。

リスト 12-9 は、新しく `Config::build` と呼ばれる関数の戻り値に必要な変更と、`Result` を返すのに必要な関数の本体を示しています。`main` も更新するまで、これはコンパイルできないことに注意してください。その更新は次のリストで行います。

ファイル名: src/main.rs

```
# use std::env;
# use std::fs;
#
# fn main() {
#     let args: Vec<String> = env::args().collect();
#
#     let config = Config::new(&args);
#
#     println!("Searching for {}", config.query);
#     println!("In file {}", config.file_path);
#
#     let contents = fs::read_to_string(config.file_path)
#         .expect("Should have been able to read the file");
#
#     println!("With text:\n{contents}");
# }
```

```
#
# struct Config {
#     query: String,
#     file_path: String,
# }
#
impl Config {
    fn build(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let file_path = args[2].clone();

        Ok(Config { query, file_path })
    }
}
```

リスト 12-9: `Config::build` から `Result` を返却する

`build` 関数は、成功時には `Config` インスタンスを、エラー時には `&'static str` を伴う `Result` を返すようになりました。ここ使うエラー値は常に、`'static` ライフタイムを持つ文字列リテラルになります。

関数の本体で 2 つ変更を行いました: 十分な数の引数をユーザが渡さなかった場合に `panic!` を呼び出す代わりに、今は `Err` 値を返し、`Config` 戻り値を `Ok` に包んでいます。これらの変更により、関数が新しい型シグニチャに適合するわけです。

`Config::build` から `Err` 値を返すことにより、`main` 関数は、`build` 関数から返ってくる `Result` 値を処理し、エラー時により綺麗にプロセスから抜け出すことができます。

12.3.3.3　`Config::build` を呼び出し、エラーを処理する

エラーケースを処理し、ユーザフレンドリーなメッセージを出力するために、`main` を更新して、リスト 12-10 に示したように `Config::build` から返されている `Result` を処理する必要があります。また、`panic!` からコマンドラインツールを 0 以外のエラーコードで抜け出す責任も奪い取り、代わりに手作業でそれを実装します。0 以外の終了コードは、我々のプログラムを呼び出したプロセスにプログラムがエラー状態で終了したことを通知する慣習です。

ファイル名: src/main.rs

```
# use std::env;
# use std::fs;
use std::process;

fn main() {
    let args: Vec<String> = env::args().collect();
```

```
    let config = Config::build(&args).unwrap_or_else(|err| {
        //       "引数解析時の問題: {err}"
        println!("Problem parsing arguments: {err}");
        process::exit(1);
    });

    // --snip--
#
#     println!("Searching for {}", config.query);
#     println!("In file {}", config.file_path);
#
#     let contents = fs::read_to_string(config.file_path)
#         .expect("Should have been able to read the file");
#
#     println!("With text:\n{contents}");
# }
#
# struct Config {
#     query: String,
#     file_path: String,
# }
#
# impl Config {
#     fn build(args: &[String]) -> Result<Config, &'static str> {
#         if args.len() < 3 {
#             return Err("not enough arguments");
#         }
#
#         let query = args[1].clone();
#         let file_path = args[2].clone();
#
#         Ok(Config { query, file_path })
#     }
# }
```

リスト 12-10: `Config` 構築に失敗したら、エラーコードで終了する

このリストにおいて、まだ詳しく扱っていないメソッドを使用しました: `unwrap_or_else` です。これは標準ライブラリで `Result<T, E>` に定義されています。`unwrap_or_else` を使うことで、`panic!` ではない何らか独自のエラー処理を定義できるのです。この `Result` が `Ok` 値だったら、このメソッドの振る舞いは `unwrap` に似ています: `Ok` が包んでいる中身の値を返すのです。しかし、値が `Err` 値なら、このメソッドは、**クロージャ**内でコードを呼び出し、クロージャは私たちが定義し、引数として `unwrap_or_else` に渡す匿名関数です。クロージャについては第 13 章で詳しく講義します。とりあえず、`unwrap_or_else` は、今回リスト 12-9 で追加した `"not enough arguments"` という静的文字列の `Err` の中身を、縦棒の間に出現する `err` 引数のクロージャに渡していることだけ知っておく必要があります。クロージャのコードはそれから、実行された時に `err` 値を使用できます。

新規 `use` 行を追加して標準ライブラリから `process` をスコープ内に持ち込みました。クロージャ内

のエラー時に走るコードは、たった 2 行です: `err` の値を出力し、それから `process::exit` を呼び出します。`process::exit` 関数は、即座にプログラムを停止させ、渡された数字を終了コードとして返します。これは、リスト 12-8 で使用した `panic!` ベースの処理と似ていますが、もう余計な出力はされません。試しましょう:

```
$ cargo run
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.48s
     Running `target/debug/minigrep`
Problem parsing arguments: not enough arguments
```

素晴らしい！ この出力の方が遥かにユーザに優しいです。

12.3.4　`main` からロジックを抽出する

これで設定解析のリファクタリングが終了したので、プログラムのロジックに目を向けましょう。「バイナリプロジェクトの責任の分離」で述べたように、現在 `main` 関数に存在する設定のセットアップやエラー処理に関わらない全てのロジックを保持することになる `run` という関数を抽出します。やり終わったら、`main` は簡潔かつ視察で確かめやすくなり、他のロジック全部に対してテストを書くことができるでしょう。

リスト 12-11 は、抜き出した `run` 関数を示しています。今は少しずつ段階的に関数を抽出する改善を行っています。それでも、**src/main.rs** に関数を定義していきます。

ファイル名: src/main.rs

```
# use std::env;
# use std::fs;
# use std::process;
#
fn main() {
    // --snip--

#     let args: Vec<String> = env::args().collect();
#
#     let config = Config::build(&args).unwrap_or_else(|err| {
#         println!("Problem parsing arguments: {err}");
#         process::exit(1);
#     });
#
    println!("Searching for {}", config.query);
    println!("In file {}", config.file_path);

    run(config);
}

fn run(config: Config) {
    let contents = fs::read_to_string(config.file_path)
```

```
        .expect("Should have been able to read the file");

    println!("With text:\n{contents}");
}

// --snip--
#
# struct Config {
#     query: String,
#     file_path: String,
# }
#
# impl Config {
#     fn build(args: &[String]) -> Result<Config, &'static str> {
#         if args.len() < 3 {
#             return Err("not enough arguments");
#         }
#
#         let query = args[1].clone();
#         let file_path = args[2].clone();
#
#         Ok(Config { query, file_path })
#     }
# }
```

リスト 12-11: 残りのプログラムロジックを含む run 関数を抽出する

これで run 関数は、ファイル読み込みから始まる main 関数の残りのロジック全てを含むようになりました。この run 関数は、引数に Config インスタンスを取ります。

12.3.4.1　run 関数からエラーを返す

残りのプログラムロジックが run 関数に隔離されたので、リスト 12-9 の Config::build のように、エラー処理を改善することができます。expect を呼び出してプログラムにパニックさせる代わりに、run 関数は、何か問題が起きた時に Result<T, E> を返します。これにより、さらにエラー処理周りのロジックをユーザに優しい形で main に統合することができます。リスト 12-12 にシグニチャと run 本体に必要な変更を示しています。

ファイル名: src/main.rs

```
# use std::env;
# use std::fs;
# use std::process;
use std::error::Error;

// --snip--

#
# fn main() {
```

```
#     let args: Vec<String> = env::args().collect();
#
#     let config = Config::build(&args).unwrap_or_else(|err| {
#         println!("Problem parsing arguments: {err}");
#         process::exit(1);
#     });
#
#     println!("Searching for {}", config.query);
#     println!("In file {}", config.file_path);
#
#     run(config);
# }
#
fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.file_path)?;

    println!("With text:\n{contents}");

    Ok(())
}
#
# struct Config {
#     query: String,
#     file_path: String,
# }
#
# impl Config {
#     fn build(args: &[String]) -> Result<Config, &'static str> {
#         if args.len() < 3 {
#             return Err("not enough arguments");
#         }
#
#         let query = args[1].clone();
#         let file_path = args[2].clone();
#
#         Ok(Config { query, file_path })
#     }
# }
```

リスト 12-12: `run` 関数を変更して `Result` を返す

ここでは、3 つの大きな変更を行いました。まず、`run` 関数の戻り値を `Result<(), Box<dyn Error>>` に変えました。この関数は、以前はユニット型、`()` を返していて、それを `Ok` の場合に返される値として残しました。

エラー型については、**トレイトオブジェクト**の `Box<dyn Error>` を使用しました (同時に冒頭で `use` 文により、`std::error::Error` をスコープに導入しています)。トレイトオブジェクトについては、第 17 章で講義します。とりあえず、`Box<dyn Error>` は、関数が `Error` トレイトを実装する型を返すことを意味しますが、戻り値の型を具体的に指定しなくても良いことを知っておいてください。これにより、エラーケースによって異なる型のエラー値を返す柔軟性を得ます。`dyn` キーワード

は、”dynamic” の略です。

2 番目に、第 9 章で書いたのと同様に、? 演算子で expect の呼び出しを置き換えました。エラーでパニックするのではなく、? は呼び出し元が処理できるように、現在の関数からエラー値を返します。

3 番目に、run 関数は今、成功時に Ok 値を返すようになりました。run 関数の成功型は、シグニチャで () と定義したので、ユニット型の値を Ok 値に包む必要があります。最初は、この Ok(()) という記法は奇妙に見えるかもしれませんが、このように () を使うことは、run を副作用のためだけに呼び出していると示唆する慣習的な方法です; 必要な値は返しません。

このコードを実行すると、コンパイルは通るものの、警告が表示されるでしょう:

```
$ cargo run the poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
warning: unused `Result` that must be used
(警告: 使用されなければならない`Result`が未使用です)
  --> src/main.rs:19:5
   |
19 |     run(config);
   |     ^^^^^^^^^^^
   |
   = note: this `Result` may be an `Err` variant, which should be handled
   = note: `#[warn(unused_must_use)]` on by default
help: use `let _ = ...` to ignore the resulting value
   |
19 |     let _ = run(config);
   |     +++++++

warning: `minigrep` (bin "minigrep") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 0.71s
     Running `target/debug/minigrep the poem.txt`
Searching for the
In file poem.txt
With text:
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us - don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

コンパイラは、コードが Result 値を無視していると教えてくれて、この Result 値は、エラーが発生したと示唆しているかもしれません。しかし、エラーがあったか確認するつもりはありませんが、コンパイラは、ここにエラー処理コードを書くつもりだったんじゃないかと思い出させてくれています！ 今、その問題を改修しましょう。

12.3.4.2　main で run から返ってきたエラーを処理する

リスト 12-10 の Config::build に対して行った方法に似たテクニックを使用してエラーを確認し、扱いますが、少し違いがあります:

ファイル名: src/main.rs

```
# use std::env;
# use std::error::Error;
# use std::fs;
# use std::process;
#
fn main() {
    // --snip--

#     let args: Vec<String> = env::args().collect();
#
#     let config = Config::build(&args).unwrap_or_else(|err| {
#         println!("Problem parsing arguments: {err}");
#         process::exit(1);
#     });
#
    println!("Searching for {}", config.query);
    println!("In file {}", config.file_path);

    if let Err(e) = run(config) {
        //       "アプリケーションエラー: {e}"
        println!("Application error: {e}");
        process::exit(1);
    }
}
#
# fn run(config: Config) -> Result<(), Box<dyn Error>> {
#     let contents = fs::read_to_string(config.file_path)?;
#
#     println!("With text:\n{contents}");
#
#     Ok(())
# }
#
# struct Config {
#     query: String,
#     file_path: String,
# }
#
# impl Config {
#     fn build(args: &[String]) -> Result<Config, &'static str> {
#         if args.len() < 3 {
#             return Err("not enough arguments");
#         }
#
```

```
#         let query = args[1].clone();
#         let file_path = args[2].clone();
#
#         Ok(Config { query, file_path })
#     }
# }
```

unwrap_or_else ではなく、if let で run が Err 値を返したかどうかを確認し、そうなら process::exit(1) を呼び出しています。run 関数は、Config::build が Config インスタンスを返すのと同じように unwrap したい値を返すことはありません。run は成功時に () を返すので、エラーを検知することにのみ興味があり、ただの () でしかない包まれた値を、unwrap_or_else に返してもらう必要はないのです。

if let と unwrap_or_else 関数の中身はどちらも同じです: エラーを出力して終了します。

12.3.5　コードをライブラリクレートに分割する

ここまで minigrep は良さそうですね！ では、**src/main.rs** ファイルを分割し、一部のコードを **src/lib.rs** ファイルに置きましょう。そのようにして、コードをテストし、**src/main.rs** ファイルの責任を減らすことができます。

main 関数以外のコード全部を **src/main.rs** から **src/lib.rs** に移動しましょう:

- run 関数定義
- 関係する use 文
- Config の定義
- Config::build 関数定義

src/lib.rs の中身にはリスト 12-13 に示したようなシグニチャがあるはずです (関数の本体は簡潔性のために省略しました)。リスト 12-14 で **src/main.rs** に変更を加えるまで、このコードはコンパイルできないことに注意してください。

ファイル名: src/lib.rs

```
use std::error::Error;
use std::fs;

pub struct Config {
    pub query: String,
    pub file_path: String,
}

impl Config {
    pub fn build(args: &[String]) -> Result<Config, &'static str> {
        // --snip--
#         if args.len() < 3 {
```

```
#             return Err("not enough arguments");
#         }
#
#         let query = args[1].clone();
#         let file_path = args[2].clone();
#
#         Ok(Config { query, file_path })
    }
}

pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    // --snip--
#     let contents = fs::read_to_string(config.file_path)?;
#
#     println!("With text:\n{contents}");
#
#     Ok(())
}
```

リスト 12-13: `Config` と `run` を **src/lib.rs** に移動する

ここでは寛大に `pub` キーワードを使用しています: `Config` のフィールドと `build` メソッドと `run` 関数です。これでテスト可能な公開 API のあるライブラリクレートができました！

さて、**src/lib.rs** に移動したコードを **src/main.rs** のバイナリクレートのスコープに持っていく必要があります。リスト 12-14 に示したようにですね。

ファイル名: src/main.rs

```
use std::env;
use std::process;

use minigrep::Config;

fn main() {
    // --snip--
#     let args: Vec<String> = env::args().collect();
#
#     let config = Config::build(&args).unwrap_or_else(|err| {
#         println!("Problem parsing arguments: {err}");
#         process::exit(1);
#     });
#
#     println!("Searching for {}", config.query);
#     println!("In file {}", config.file_path);
#
    if let Err(e) = minigrep::run(config) {
        // --snip--
#         println!("Application error: {e}");
#         process::exit(1);
    }
```

```
}
```

リスト 12-14: **src/main.rs** で `minigrep` ライブラリクレートを使用する

`use minigrep::Config` の行を追加することで、`Config` 型をライブラリクレートからバイナリクレートのスコープに持ってきて、`run` 関数にクレート名を接頭辞として付けます。これで全機能が連結され、動くはずです。`cargo run` でプログラムを走らせて、すべてがうまくいっていることを確かめてください。

ふう！ 作業量が多かったですね。ですが、将来成功する準備はできています。もう、エラー処理は遥かに楽になり、コードのモジュール化もできました。ここから先の作業は、ほぼ **src/lib.rs** で完結するでしょう。

古いコードでは大変だけれども、新しいコードでは楽なことをして新発見のモジュール性を活用しましょう: テストを書くのです！

12.4　テスト駆動開発でライブラリの機能を開発する

今や、ロジックを **src/lib.rs** に抜き出し、引数集めとエラー処理を **src/main.rs** に残したので、コードの核となる機能のテストを書くのが非常に容易になりました。いろんな引数で関数を直接呼び出し、コマンドラインからバイナリを呼び出す必要なく戻り値を確認できます。

この節では、以下の手順に従ってテスト駆動開発 (TDD) プロセスを活用して、`minigrep` プログラムに検索ロジックを追加します:

1. 失敗するテストを書き、走らせて想定通りの理由で失敗することを確かめる。
2. 十分な量のコードを書くか変更して新しいテストを通過するようにする。
3. 追加または変更したばかりのコードをリファクタリングし、テストが通り続けることを確認する。
4. 手順 1 から繰り返す！

TDD はソフトウェアを書く多くの方法のうちの一つに過ぎませんが、コードデザインを駆動するために役立てることができます。テストを通過させるコードを書く前にテストを書くことで、過程を通して高いテストカバー率を保つ助けになります。

実際にクエリ文字列の検索を行う機能の実装をテスト駆動し、クエリに合致する行のリストを生成します。この機能を `search` という関数に追加しましょう。

12.4.1　失敗するテストを記述する

もう必要ないので、プログラムの振る舞いを確認していた `println!` 文を **src/lib.rs** と **src/main.rs** から削除しましょう。それから **src/lib.rs** で、テスト関数のある `tests` モジュールを追加してください。第 11 章のようにですね。このテスト関数が `search` 関数に欲しい振る舞いを指定

します: クエリと検索対象のテキストを受け取り、クエリを含む行だけをテキストから返します。リスト 12-15 にこのテストを示していますが、まだコンパイルは通りません。

ファイル名: src/lib.rs

```
# use std::error::Error;
# use std::fs;
#
# pub struct Config {
#     pub query: String,
#     pub file_path: String,
# }
#
# impl Config {
#     pub fn build(args: &[String]) -> Result<Config, &'static str> {
#         if args.len() < 3 {
#             return Err("not enough arguments");
#         }
#
#         let query = args[1].clone();
#         let file_path = args[2].clone();
#
#         Ok(Config { query, file_path })
#     }
# }
#
# pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
#     let contents = fs::read_to_string(config.file_path)?;
#
#     Ok(())
# }
#
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn one_result() {
        let query = "duct";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.";

        assert_eq!(vec!["safe, fast, productive."], search(query, contents));
    }
}
```

リスト 12-15: こうだったらいいなという search 関数の失敗するテストを作成する

このテストは、"duct" という文字列を検索します。検索対象の文字列は 3 行で、うち 1 行だけが

"duct" を含みます (開き二重引用符の後のバックスラッシュは、この文字列リテラルの内容の先頭に改行文字を置かないように、コンパイラに指示しているということに注意してください)。search 関数から返る値が想定している行だけを含むことをアサーションします。

このテストを走らせ、失敗するところを観察することは、まだできません。このテストはコンパイルもできないからです: まだ search 関数が存在していません！ TDD の原則に従って、空のベクタを常に返す search 関数の定義を追加することで、テストをコンパイルし走らせるだけのコードを追記します。リスト 12-16 に示したようにですね。そうすれば、テストはコンパイルでき、失敗するはずです。なぜなら、空のベクタは、"safe, fast, productive." という行を含むベクタとは合致しないからです。

ファイル名: src/lib.rs

```
# use std::error::Error;
# use std::fs;
#
# pub struct Config {
#     pub query: String,
#     pub file_path: String,
# }
#
# impl Config {
#     pub fn build(args: &[String]) -> Result<Config, &'static str> {
#         if args.len() < 3 {
#             return Err("not enough arguments");
#         }
#
#         let query = args[1].clone();
#         let file_path = args[2].clone();
#
#         Ok(Config { query, file_path })
#     }
# }
#
# pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
#     let contents = fs::read_to_string(config.file_path)?;
#
#     Ok(())
# }
#
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    vec![]
}
#
# #[cfg(test)]
# mod tests {
#     use super::*;
#
#     #[test]
#     fn one_result() {
```

```
#         let query = "duct";
#         let contents = "\
# Rust:
# safe, fast, productive.
# Pick three.";
#
#         assert_eq!(vec!["safe, fast, productive."], search(query, contents));
#     }
# }
```

リスト 12-16: テストがコンパイルできるのに十分なだけ search 関数を定義する

search のシグニチャ内で、明示的なライフタイム 'a を定義し、そのライフタイムを contents 引数と戻り値で使用していることに注目してください。第 10 章からライフタイム仮引数は、どの実引数のライフタイムが戻り値のライフタイムに関連づけられているかを指定することを思い出してください。この場合、返却されるベクタは、(query 引数ではなく) contents 引数のスライスを参照する文字列スライスを含むべきと示唆しています。

言い換えると、コンパイラに search 関数に返されるデータは、search 関数に contents 引数で渡されているデータと同期間生きることを教えています。これは重要なことです！ スライスに参照されるデータは、参照が有効になるために有効である必要があるのです; コンパイラが contents ではなく query の文字列スライスを生成すると想定してしまったら、安全性チェックを間違って行うことになってしまいます。

ライフタイム注釈を忘れてこの関数をコンパイルしようとすると、こんなエラーが出ます:

```
$ cargo build
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
error[E0106]: missing lifetime specifier
(エラー: ライフタイム指定子が欠けています)
  --> src/lib.rs:28:51
   |
28 | pub fn search(query: &str, contents: &str) -> Vec<&str> {
   |                      ----            ----         ^ expected named lifetime
        parameter
   |
   = help: this function's return type contains a borrowed value, but the
       signature does not say whether it is borrowed from `query` or `contents`
    (ヘルプ: この関数の戻り値は、借用された値を含んでいますが、シグニチャにはそ
        れが、`query`か`contents`から借用されたものであるかが示されていません)
help: consider introducing a named lifetime parameter
   |
28 | pub fn search<'a>(query: &'a str, contents: &'a str) -> Vec<&'a str> {
   |              ++++         ++                 ++              ++

For more information about this error, try `rustc --explain E0106`.
error: could not compile `minigrep` (lib) due to 1 previous error
```

コンパイラには、二つの引数のどちらが必要なのか知る由がないので、明示的に教えてあげる必要

があるのです。contents がテキストを全て含む引数で、合致するそのテキストの一部を返したいので、contents がライフタイム記法で戻り値に関連づくはずの引数であることをプログラマは知っています。

他のプログラミング言語では、シグニチャで引数と戻り値を関連づける必要はありませんが、この実践は時間とともに楽になっていくでしょう。この例を第 10 章の「ライフタイムで参照を検証する」節と比較してみるといいかもしれません。

さあ、テストを実行しましょう:

```
$ cargo test
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished test [unoptimized + debuginfo] target(s) in 0.97s
     Running unittests src/lib.rs (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 1 test
test tests::one_result ... FAILED

failures:

---- tests::one_result stdout ----
thread 'tests::one_result' panicked at src/lib.rs:44:9:
assertion `left == right` failed
  left: ["safe, fast, productive."]
 right: []
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::one_result

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

error: test failed, to rerun pass `--lib`
```

素晴らしい。テストは全く想定通りに失敗しています。テストが通るようにしましょう！

12.4.2　テストを通過させるコードを書く

空のベクタを常に返しているために、現状テストは失敗しています。それを修正し、search を実装するには、プログラムは以下の手順に従う必要があります:

- 中身を各行ごとに繰り返す。
- 行にクエリ文字列が含まれるか確認する。
- するなら、それを返却する値のリストに追加する。
- しないなら、何もしない。
- 一致する結果のリストを返す。

各行を繰り返す作業から、この手順に順に取り掛かりましょう。

12.4.2.1 `lines` メソッドで各行を繰り返す

Rust には、文字列を行ごとに繰り返す役立つメソッドがあり、利便性のために `lines` と名付けられ、リスト 12-17 のように動作します。まだ、これはコンパイルできないことに注意してください。

ファイル名: src/lib.rs

```
# use std::error::Error;
# use std::fs;
#
# pub struct Config {
#     pub query: String,
#     pub file_path: String,
# }
#
# impl Config {
#     pub fn build(args: &[String]) -> Result<Config, &'static str> {
#         if args.len() < 3 {
#             return Err("not enough arguments");
#         }
#
#         let query = args[1].clone();
#         let file_path = args[2].clone();
#
#         Ok(Config { query, file_path })
#     }
# }
#
# pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
#     let contents = fs::read_to_string(config.file_path)?;
#
#     Ok(())
# }
#
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    for line in contents.lines() {
        // 行に対して何かする
        // do something with line
    }
}
#
# #[cfg(test)]
# mod tests {
#     use super::*;
#
#     #[test]
#     fn one_result() {
#         let query = "duct";
#         let contents = "\
# Rust:
```

```
# safe, fast, productive.
# Pick three.";
#
#         assert_eq!(vec!["safe, fast, productive."], search(query, contents));
#     }
# }
```

リスト 12-17: `contents` の各行を繰り返す

`lines` メソッドはイテレータを返します。イテレータについて詳しくは、第 13 章で話しますが、リスト 3-5 でこのようなイテレータの使用法は見かけたことを思い出してください。そこでは、イテレータに `for` ループを使用してコレクションの各要素に対して何らかのコードを走らせていました。

12.4.2.2　クエリを求めて各行を検索する

次に現在の行がクエリ文字列を含むか確認します。幸運なことに、文字列にはこれを行ってくれる `contains` という役に立つメソッドがあります！ `search` 関数に、`contains` メソッドの呼び出しを追加してください。リスト 12-18 のようにですね。それでもまだコンパイルできないことに注意してください。

ファイル名: src/lib.rs

```
# use std::error::Error;
# use std::fs;
#
# pub struct Config {
#     pub query: String,
#     pub file_path: String,
# }
#
# impl Config {
#     pub fn build(args: &[String]) -> Result<Config, &'static str> {
#         if args.len() < 3 {
#             return Err("not enough arguments");
#         }
#
#         let query = args[1].clone();
#         let file_path = args[2].clone();
#
#         Ok(Config { query, file_path })
#     }
# }
#
# pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
#     let contents = fs::read_to_string(config.file_path)?;
#
#     Ok(())
# }
#
```

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    for line in contents.lines() {
        if line.contains(query) {
            // do something with line
        }
    }
}
#
# #[cfg(test)]
# mod tests {
#     use super::*;
#
#     #[test]
#     fn one_result() {
#         let query = "duct";
#         let contents = "\
# Rust:
# safe, fast, productive.
# Pick three.";
#
#         assert_eq!(vec!["safe, fast, productive."], search(query, contents));
#     }
# }
```

リスト 12-18: 行が query の文字列を含むか確認する機能を追加する

ここまでで機能を組み上げてきました。これをコンパイルできるようにするためには、関数のシグネチャでそうすると示したように、本体から値を返す必要があります。

12.4.2.3　合致した行を保存する

この関数を完成させるには、返そうとしている、合致した行を保存する方法が必要です。そのために、for ループの前に可変なベクタを生成し、push メソッドを呼び出して line をベクタに保存することができます。for ループの後でベクタを返却します。リスト 12-19 のようにですね。

ファイル名: src/lib.rs

```
# use std::error::Error;
# use std::fs;
#
# pub struct Config {
#     pub query: String,
#     pub file_path: String,
# }
#
# impl Config {
#     pub fn build(args: &[String]) -> Result<Config, &'static str> {
#         if args.len() < 3 {
#             return Err("not enough arguments");
#         }
```

```
#
#         let query = args[1].clone();
#         let file_path = args[2].clone();
#
#         Ok(Config { query, file_path })
#     }
# }
#
# pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
#     let contents = fs::read_to_string(config.file_path)?;
#
#     Ok(())
# }
#
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
    }

    results
}
#
# #[cfg(test)]
# mod tests {
#     use super::*;
#
#     #[test]
#     fn one_result() {
#         let query = "duct";
#         let contents = "\
# Rust:
# safe, fast, productive.
# Pick three.";
#
#         assert_eq!(vec!["safe, fast, productive."], search(query, contents));
#     }
# }
```

リスト 12-19: 合致する行を保存したので、返すことができる

これで search 関数は、query を含む行だけを返すはずであり、テストも通るはずです。テストを実行しましょう:

```
$ cargo test
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished test [unoptimized + debuginfo] target(s) in 1.22s
     Running unittests src/lib.rs (target/debug/deps/minigrep-9cd200e5fac0fc94)
```

```
running 1 test
test tests::one_result ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

     Running unittests src/main.rs (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

   Doc-tests minigrep

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s
```

テストが通り、動いていることがわかりました！

ここで、テストが通過するよう保ったまま、同じ機能を保持しながら、検索関数の実装をリファクタリングする機会を考えることもできます。検索関数のコードは悪すぎるわけではありませんが、イテレータの有用な機能の一部を活用していません。この例には第 13 章で再度触れ、そこでは、イテレータをより深く探究し、さらに改善する方法に目を向けます。

12.4.2.4　run 関数内で search 関数を使用する

search 関数が動きテストできたので、run 関数から search を呼び出す必要があります。config.query の値と、ファイルから run が読み込む contents の値を search 関数に渡す必要があります。それから run は、search から返ってきた各行を出力するでしょう:

ファイル名: src/lib.rs

```
# use std::error::Error;
# use std::fs;
#
# pub struct Config {
#     pub query: String,
#     pub file_path: String,
# }
#
# impl Config {
#     pub fn build(args: &[String]) -> Result<Config, &'static str> {
#         if args.len() < 3 {
#             return Err("not enough arguments");
#         }
#
```

```
#         let query = args[1].clone();
#         let file_path = args[2].clone();
#
#         Ok(Config { query, file_path })
#     }
# }
#
pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.file_path)?;

    for line in search(&config.query, &contents) {
        println!("{line}");
    }

    Ok(())
}
#
# pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
#     let mut results = Vec::new();
#
#     for line in contents.lines() {
#         if line.contains(query) {
#             results.push(line);
#         }
#     }
#
#     results
# }
#
# #[cfg(test)]
# mod tests {
#     use super::*;
#
#     #[test]
#     fn one_result() {
#         let query = "duct";
#         let contents = "\
# Rust:
# safe, fast, productive.
# Pick three.";
#
#         assert_eq!(vec!["safe, fast, productive."], search(query, contents));
#     }
# }
```

それでも `for` ループで `search` から各行を返し、出力しています。

さて、プログラム全体が動くはずです！ 試してみましょう。まずはエミリー・ディキンソンの詩から、ちょうど 1 行だけを返すはずの言葉から。”frog” です:

```
$ cargo run -- frog poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.38s
     Running `target/debug/minigrep frog poem.txt`
How public, like a frog
```

かっこいい！ 今度は、複数行にマッチするであろう言葉を試しましょう。”body” とかね:

```
$ cargo run -- body poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/minigrep body poem.txt`
I'm nobody! Who are you?
Are you nobody, too?
How dreary to be somebody!
```

そして最後に、詩のどこにも現れない単語を探したときに、何も出力がないことを確かめましょう。”monomorphization” などね:

```
$ cargo run -- monomorphization poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/minigrep monomorphization poem.txt`
```

最高です！ 古典的なツールの独自のミニバージョンを構築し、アプリケーションを構造化する方法を多く学びました。また、ファイル入出力、ライフタイム、テスト、コマンドライン引数の解析についても、少し学びました。

このプロジェクトをまとめ上げるために、環境変数を扱う方法と標準エラー出力に出力する方法を少しだけデモします。これらはどちらも、コマンドラインプログラムを書く際に有用です。

12.5　環境変数を取り扱う

おまけの機能を追加して `minigrep` を改善します: 環境変数でユーザがオンにできる大文字小文字無視の検索用のオプションです。この機能をコマンドラインオプションにして、適用したい度にユーザが入力しなければならないようにすることもできますが、代わりに環境変数とすることで、ユーザは 1 回環境変数をセットすれば、そのターミナルセッションの間は大文字小文字無視の検索を行うことができるようにします。

12.5.1　大文字小文字を区別しない `search` 関数用に失敗するテストを書く

まず、環境変数が値を持つ場合に呼び出される `search_case_insensitive` 関数を新しく追加します。テスト駆動開発の過程に従い続けるので、最初の手順は、今回も失敗するテストを書くことです。新しい `search_case_insensitive` 関数用の新規テストを追加し、古いテストを `one_result` から `case_sensitive` に名前変更して、二つのテストの差異を明確化します。リスト 12-20 に示したようにですね。

ファイル名: src/lib.rs

```
# use std::error::Error;
# use std::fs;
#
# pub struct Config {
#     pub query: String,
#     pub file_path: String,
# }
#
# impl Config {
#     pub fn build(args: &[String]) -> Result<Config, &'static str> {
#         if args.len() < 3 {
#             return Err("not enough arguments");
#         }
#
#         let query = args[1].clone();
#         let file_path = args[2].clone();
#
#         Ok(Config { query, file_path })
#     }
# }
#
# pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
#     let contents = fs::read_to_string(config.file_path)?;
#
#     for line in search(&config.query, &contents) {
#         println!("{line}");
#     }
#
#     Ok(())
# }
#
# pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
#     let mut results = Vec::new();
#
#     for line in contents.lines() {
#         if line.contains(query) {
#             results.push(line);
#         }
#     }
#
#     results
# }
#
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn case_sensitive() {
        let query = "duct";
        let contents = "\
```

```
Rust:
safe, fast, productive.
Pick three.
Duct tape.";

        assert_eq!(vec!["safe, fast, productive."], search(query, contents));
    }

    #[test]
    fn case_insensitive() {
        let query = "rUsT";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.
Trust me.";

        assert_eq!(
            vec!["Rust:", "Trust me."],
            search_case_insensitive(query, contents)
        );
    }
}
```

リスト 12-20: 追加しようとしている大文字小文字を区別しない関数用の失敗するテストを新しく追加する

古いテストの `contents` も変更していることに注意してください。大文字小文字を区別する検索を行う際に、`"duct"` というクエリに合致しないはずの大文字 D を使用した `"Duct tape"` (ガムテープ) という新しい行を追加しました。このように古いテストを変更することで、既に実装済みの大文字小文字を区別する検索機能を誤って壊してしまわないことを保証する助けになります。このテストはもう通り、大文字小文字を区別しない検索に取り掛かっても通り続けるはずです。

大文字小文字を区別**しない**検索の新しいテストは、クエリに `"rUsT"` を使用しています。追加直前の `search_case_insensitive` 関数では、`"rUsT"` というクエリは、両方ともクエリとは大文字小文字が異なるのに、大文字 R の `"Rust:"` を含む行と、`"Trust me."` という行にもマッチするはずです。これが失敗するテストであり、まだ `search_case_insensitive` 関数を定義していないので、コンパイルは失敗するでしょう。リスト 12-16 の `search` 関数で行ったのと同様に空のベクタを常に返すような仮実装を追加し、テストがコンパイルされるものの、失敗する様をご自由に確認してください。

12.5.2　`search_case_insensitive` 関数を実装する

`search_case_insensitive` 関数は、リスト 12-21 に示しましたが、`search` 関数とほぼ同じです。唯一の違いは、`query` と各 `line` を小文字化していることなので、入力引数の大文字小文字によらず、行がクエリを含んでいるか確認する際には、同じになるわけです。

ファイル名: src/lib.rs

```
# use std::error::Error;
# use std::fs;
#
# pub struct Config {
#     pub query: String,
#     pub file_path: String,
# }
#
# impl Config {
#     pub fn build(args: &[String]) -> Result<Config, &'static str> {
#         if args.len() < 3 {
#             return Err("not enough arguments");
#         }
#
#         let query = args[1].clone();
#         let file_path = args[2].clone();
#
#         Ok(Config { query, file_path })
#     }
# }
#
# pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
#     let contents = fs::read_to_string(config.file_path)?;
#
#     for line in search(&config.query, &contents) {
#         println!("{line}");
#     }
#
#     Ok(())
# }
#
# pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
#     let mut results = Vec::new();
#
#     for line in contents.lines() {
#         if line.contains(query) {
#             results.push(line);
#         }
#     }
#
#     results
# }
#
pub fn search_case_insensitive<'a>(
    query: &str,
    contents: &'a str,
) -> Vec<&'a str> {
    let query = query.to_lowercase();
    let mut results = Vec::new();

    for line in contents.lines() {
```

```
        if line.to_lowercase().contains(&query) {
            results.push(line);
        }
    }

    results
}
#
# #[cfg(test)]
# mod tests {
#     use super::*;
#
#     #[test]
#     fn case_sensitive() {
#         let query = "duct";
#         let contents = "\
# Rust:
# safe, fast, productive.
# Pick three.
# Duct tape.";
#
#         assert_eq!(vec!["safe, fast, productive."], search(query, contents));
#     }
#
#     #[test]
#     fn case_insensitive() {
#         let query = "rUsT";
#         let contents = "\
# Rust:
# safe, fast, productive.
# Pick three.
# Trust me.";
#
#         assert_eq!(
#             vec!["Rust:", "Trust me."],
#             search_case_insensitive(query, contents)
#         );
#     }
# }
```

リスト 12-21: 比較する前にクエリと行を小文字化するよう、search_case_insensitive 関数を定義する

まず、query 文字列を小文字化し、同じ名前の覆い隠された変数に保存します。ユーザのクエリが "rust" や "RUST" 、"Rust" 、"rUsT" などだったりしても、"rust" であり、大文字小文字を区別しないかのようにクエリを扱えるように、to_lowercase をクエリに対して呼び出すことは必須です。to_lowercase は基本的な Unicode を処理しますが、100% 正確ではありません。現実のアプリケーションを書いているとしたら、ここでもう少し処理を入れたくなるでしょうが、この節は環境変数についての節であって Unicode についての節ではないので、ここではそのままにしておきましょう。

query は最早、文字列スライスではなく String であることに注意してください。というのも、to_lowercase を呼び出すと、既存のデータを参照するというよりも、新しいデータを作成するからです。例として、クエリは "rUsT" だとしましょう: その文字列スライスは、小文字の u や t を使えるように含んでいないので、"rust" を含む新しい String のメモリを確保しなければならないのです。今、contains メソッドに引数として query を渡すと、アンド記号を追加する必要があります。contains のシグニチャは、文字列スライスを取るよう定義されているからです。

次に、各 line に対して to_lowercase の呼び出しを追加し、全文字を小文字化しています。今や line と query を小文字に変換したので、クエリが大文字であろうと小文字であろうとマッチを検索するでしょう。

この実装がテストを通過するか確認しましょう:

```
$ cargo test
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished test [unoptimized + debuginfo] target(s) in 1.33s
     Running unittests src/lib.rs (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 2 tests
test tests::case_insensitive ... ok
test tests::case_sensitive ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

     Running unittests src/main.rs (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

   Doc-tests minigrep

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s
```

素晴らしい！ どちらも通りました。では、run 関数から新しい search_case_insensitive 関数を呼び出しましょう。1 番目に大文字小文字の区別を切り替えられるよう、Config 構造体に設定オプションを追加します。まだどこでも、このフィールドの初期化をしていないので、追加するとコンパイルエラーが起きます:

ファイル名: src/lib.rs

```
# use std::error::Error;
# use std::fs;
#
```

```
pub struct Config {
    pub query: String,
    pub file_path: String,
    pub ignore_case: bool,
}
#
# impl Config {
#     pub fn build(args: &[String]) -> Result<Config, &'static str> {
#         if args.len() < 3 {
#             return Err("not enough arguments");
#         }
#
#         let query = args[1].clone();
#         let file_path = args[2].clone();
#
#         Ok(Config { query, file_path })
#     }
# }
#
# pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
#     let contents = fs::read_to_string(config.file_path)?;
#
#     let results = if config.ignore_case {
#         search_case_insensitive(&config.query, &contents)
#     } else {
#         search(&config.query, &contents)
#     };
#
#     for line in results {
#         println!("{line}");
#     }
#
#     Ok(())
# }
#
# pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
#     let mut results = Vec::new();
#
#     for line in contents.lines() {
#         if line.contains(query) {
#             results.push(line);
#         }
#     }
#
#     results
# }
#
# pub fn search_case_insensitive<'a>(
#     query: &str,
#     contents: &'a str,
# ) -> Vec<&'a str> {
#     let query = query.to_lowercase();
#     let mut results = Vec::new();
```

```
#
#     for line in contents.lines() {
#         if line.to_lowercase().contains(&query) {
#             results.push(line);
#         }
#     }
#
#     results
# }
#
# #[cfg(test)]
# mod tests {
#     use super::*;
#
#     #[test]
#     fn case_sensitive() {
#         let query = "duct";
#         let contents = "\
# Rust:
# safe, fast, productive.
# Pick three.
# Duct tape.";
#
#         assert_eq!(vec!["safe, fast, productive."], search(query, contents));
#     }
#
#     #[test]
#     fn case_insensitive() {
#         let query = "rUsT";
#         let contents = "\
# Rust:
# safe, fast, productive.
# Pick three.
# Trust me.";
#
#         assert_eq!(
#             vec!["Rust:", "Trust me."],
#             search_case_insensitive(query, contents)
#         );
#     }
# }
```

論理値を持つ `ignore_case` フィールドを追加しました。次に、`run` 関数に、`ignore_case` フィールドの値を確認し、`search` 関数か `search_case_insensitive` 関数を呼ぶかを決定するのに使ってもらう必要があります。リスト 12-22 のようにですね。それでも、これはまだコンパイルできません。

ファイル名: src/lib.rs

```
# use std::error::Error;
# use std::fs;
#
```

```
# pub struct Config {
#     pub query: String,
#     pub file_path: String,
#     pub ignore_case: bool,
# }
#
# impl Config {
#     pub fn build(args: &[String]) -> Result<Config, &'static str> {
#         if args.len() < 3 {
#             return Err("not enough arguments");
#         }
#
#         let query = args[1].clone();
#         let file_path = args[2].clone();
#
#         Ok(Config { query, file_path })
#     }
# }
#
pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.file_path)?;

    let results = if config.ignore_case {
        search_case_insensitive(&config.query, &contents)
    } else {
        search(&config.query, &contents)
    };

    for line in results {
        println!("{line}");
    }

    Ok(())
}
#
# pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
#     let mut results = Vec::new();
#
#     for line in contents.lines() {
#         if line.contains(query) {
#             results.push(line);
#         }
#     }
#
#     results
# }
#
# pub fn search_case_insensitive<'a>(
#     query: &str,
#     contents: &'a str,
# ) -> Vec<&'a str> {
#     let query = query.to_lowercase();
#     let mut results = Vec::new();
```

```
#
#     for line in contents.lines() {
#         if line.to_lowercase().contains(&query) {
#             results.push(line);
#         }
#     }
#
#     results
# }
#
# #[cfg(test)]
# mod tests {
#     use super::*;
#
#     #[test]
#     fn case_sensitive() {
#         let query = "duct";
#         let contents = "\
# Rust:
# safe, fast, productive.
# Pick three.
# Duct tape.";
#
#         assert_eq!(vec!["safe, fast, productive."], search(query, contents));
#     }
#
#     #[test]
#     fn case_insensitive() {
#         let query = "rUsT";
#         let contents = "\
# Rust:
# safe, fast, productive.
# Pick three.
# Trust me.";
#
#         assert_eq!(
#             vec!["Rust:", "Trust me."],
#             search_case_insensitive(query, contents)
#         );
#     }
# }
```

リスト 12-22: `config.ignore_case` の値に基づいて `search` か `search_case_insensitive` を呼び出す

最後に、環境変数を確認する必要があります。環境変数を扱う関数は、標準ライブラリの `env` モジュールにあるので、**src/lib.rs** の冒頭でそのモジュールをスコープ内に持ち込みます。そして、`env` モジュールから `var` 関数を使用して、`IGNORE_CASE` という環境変数に何らかの値が設定されているかチェックします。リスト 12-23 のようにですね。

ファイル名: src/lib.rs

```
use std::env;
// --snip--

# use std::error::Error;
# use std::fs;
#
# pub struct Config {
#     pub query: String,
#     pub file_path: String,
#     pub ignore_case: bool,
# }
#
impl Config {
    pub fn build(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let file_path = args[2].clone();

        let ignore_case = env::var("IGNORE_CASE").is_ok();

        Ok(Config {
            query,
            file_path,
            ignore_case,
        })
    }
}
#
# pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
#     let contents = fs::read_to_string(config.file_path)?;
#
#     let results = if config.ignore_case {
#         search_case_insensitive(&config.query, &contents)
#     } else {
#         search(&config.query, &contents)
#     };
#
#     for line in results {
#         println!("{line}");
#     }
#
#     Ok(())
# }
#
# pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
#     let mut results = Vec::new();
#
#     for line in contents.lines() {
```

```
#         if line.contains(query) {
#             results.push(line);
#         }
#     }
#
#     results
# }
#
# pub fn search_case_insensitive<'a>(
#     query: &str,
#     contents: &'a str,
# ) -> Vec<&'a str> {
#     let query = query.to_lowercase();
#     let mut results = Vec::new();
#
#     for line in contents.lines() {
#         if line.to_lowercase().contains(&query) {
#             results.push(line);
#         }
#     }
#
#     results
# }
#
# #[cfg(test)]
# mod tests {
#     use super::*;
#
#     #[test]
#     fn case_sensitive() {
#         let query = "duct";
#         let contents = "\
# Rust:
# safe, fast, productive.
# Pick three.
# Duct tape.";
#
#         assert_eq!(vec!["safe, fast, productive."], search(query, contents));
#     }
#
#     #[test]
#     fn case_insensitive() {
#         let query = "rUsT";
#         let contents = "\
# Rust:
# safe, fast, productive.
# Pick three.
# Trust me.";
#
#         assert_eq!(
#             vec!["Rust:", "Trust me."],
#             search_case_insensitive(query, contents)
#         );
```

```
#     }
# }
```

リスト 12-23: IGNORE_CASE という環境変数に何らかの値が設定されているかチェックする

ここで、ignore_case という新しい変数を生成しています。その値をセットするために、env::var 関数を呼び出し、IGNORE_CASE 環境変数の名前を渡しています。env::var 関数は、環境変数に何らかの値がセットされていたら、環境変数の値を含む Ok 列挙子の成功値になる Result を返します。環境変数がセットされていなければ、Err 列挙子を返すでしょう。

Result の is_ok メソッドを使用して、環境変数が設定されているか、つまりプログラムが大文字小文字を区別しない検索を行うべきかどうかを、チェックしています。IGNORE_CASE 環境変数が何にも設定されていなければ、is_ok は false を返し、プログラムは大文字小文字を区別する検索を実行するでしょう。環境変数の**値**はどうでもよく、セットされているかどうかだけ気にするので、unwrap や expect あるいは、他のここまで見かけた Result のメソッドを使用するのではなく、is_ok をチェックしています。

ignore_case 変数の値を Config インスタンスに渡しているので、リスト 12-22 で実装したように、run 関数はその値を読み取り、search_case_insensitive か search を呼び出すか決定できるのです。

試行してみましょう！ まず、環境変数をセットせずにクエリは to でプログラムを実行し、この時は全て小文字で”to” という言葉を含むあらゆる行が合致するはずです。

```
$ cargo run -- to poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/minigrep to poem.txt`
Are you nobody, too?
How dreary to be somebody!
```

まだ機能しているようです！ では、IGNORE_CASE を 1 にしつつ、同じクエリの to でプログラムを実行しましょう。

```
$ IGNORE_CASE=1 cargo run -- to poem.txt
```

PowerShell を使用しているなら、環境変数の設定とプログラムの実行を別々のコマンドとして実行する必要があるでしょう:

```
PS> $Env:IGNORE_CASE=1; cargo run -- to poem.txt
```

これを実行すると、IGNORE_CASE は以降のシェルセッションでも残り続けるでしょう。これは Remove-Item コマンドレットで解除することができます:

```
PS> Remove-Item Env:IGNORE_CASE
```

大文字も含む可能性のある”to” を含有する行が得られるはずです:

```
Are you nobody, too?
How dreary to be somebody!
To tell your name the livelong day
To an admiring bog!
```

素晴らしい、”To” を含む行も出てきましたね！ minigrep プログラムはこれで、環境変数によって制御できる大文字小文字を区別しない検索も行えるようになりました。もうコマンドライン引数か、環境変数を使ってオプションを管理する方法も知りましたね。

引数**と**環境変数で同じ設定を行うことができるプログラムもあります。そのような場合、プログラムはどちらが優先されるか決定します。自身の別の鍛錬として、コマンドライン引数か、環境変数で大文字小文字の区別を制御できるようにしてみてください。片方は大文字小文字を区別するようにセットされ、もう片方は無視するようにセットしてプログラムが実行された時に、コマンドライン引数と環境変数のどちらの優先度が高くなるかを決めてください。

std::env モジュールは、環境変数を扱うもっと多くの有用な機能を有しています: ドキュメンテーションを確認して、何が利用可能か確かめてください。

12.6　標準出力ではなく標準エラーにエラーメッセージを書き込む

現時点では、すべての出力を println! マクロを使用して端末に書き込んでいます。多くの端末には、2 種類の出力があります: 普通の情報用の**標準出力 (** stdout **)** とエラーメッセージ用の**標準エラー出力 (** stderr **)** です。この差異のおかげで、ユーザは、エラーメッセージを画面に表示しつつ、プログラムの成功した出力をファイルにリダイレクトすることを選択できます。

println! マクロは、標準出力に出力する能力しかないので、標準エラーに出力するには他のものを使用しなければなりません。

12.6.1　エラーが書き込まれる場所を確認する

まず、minigrep に出力される中身が、代わりに標準エラーに書き込みたいいかなるエラーメッセージも含め、どのように標準出力に書き込まれているかを観察しましょう。意図的にエラーを起こしつつ、ファイルに標準出力ストリームをリダイレクトすることでそうします。標準エラーストリームはリダイレクトしないので、標準エラーに送られる内容は、すべて画面に表示され続けます。

コマンドラインプログラムは、エラーメッセージを標準エラー出力に送信していると期待されているので、標準出力ストリームをファイルにリダイレクトしても、画面にエラーメッセージが見られます。我々のプログラムは、現状、いい振る舞いをしていません: 代わりにファイルにエラーメッセージ出力を保存するところを、目撃するところです！

この動作をデモするために、> と、標準出力ストリームをリダイレクトする先のファイルパスである **output.txt** を付けて、プログラムを走らせましょう。引数は何も渡さず、そうするとエラーが起きるはずです:

```
$ cargo run > output.txt
```

> 記法により、標準出力の中身を画面の代わりに **output.txt** に書き込むようシェルは指示されます。画面に出力されると期待していたエラーメッセージは見られないので、ファイルに入っているということでしょう。以下が **output.txt** が含んでいる内容です:

```
Problem parsing arguments: not enough arguments
```

そうです。エラーメッセージは標準出力に出力されているのです。このようなエラーメッセージは標準エラーに出力され、成功した状態のデータのみがファイルに残ると遥かに有用です。それを変更します。

12.6.2　エラーを標準エラーに出力する

リスト 12-24 のコードを使用して、エラーメッセージの出力の仕方を変更します。この章の前で行ったリファクタリングのため、エラーメッセージを出力するコードはすべて 1 関数、`main` にあります。標準ライブラリは、標準エラーストリームに出力する `eprintln!` マクロを提供しているので、`println!` を呼び出してエラーを出力していた 2 箇所を代わりに `eprintln!` を使うように変更しましょう。

ファイル名: src/main.rs

```
# use std::env;
# use std::process;
#
# use minigrep::Config;
#
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::build(&args).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {err}");
        process::exit(1);
    });

    if let Err(e) = minigrep::run(config) {
        eprintln!("Application error: {e}");
        process::exit(1);
    }
}
```

リスト 12-24: `eprintln!` を使って標準出力ではなく、標準エラーにエラーメッセージを書き込む

それでは再度同じように、引数なしで、標準出力を > でリダイレクトしつつ、プログラムを実行しましょう:

```
$ cargo run > output.txt
Problem parsing arguments: not enough arguments
```

これで、エラーは画面に見えつつ、**output.txt** は何も含まなくなり、これはコマンドラインプログラムに期待する動作です。

再度、標準出力をファイルにリダイレクトしてエラーは起こさない引数でプログラムを走らせましょう。以下のようにですね:

```
$ cargo run -- to poem.txt > output.txt
```

ターミナルには出力は見られず、**output.txt** に結果が含まれます:

ファイル名: output.txt

```
Are you nobody, too?
How dreary to be somebody!
```

これは、もう成功した出力には標準出力を、エラー出力には標準エラーを適切に使用していることをデモしています。

12.7　まとめ

この章では、ここまでに学んできた主要な概念の一部を念押しし、Rust で入出力処理を行う方法を講義しました。コマンドライン引数、ファイル、環境変数、そしてエラー出力に `eprintln!` マクロを使用することで、もう、コマンドラインアプリケーションを書く準備ができています。以前の章の概念と組み合わせて、コードはうまく体系化され、適切なデータ構造に効率的にデータを保存し、エラーをうまく扱い、よくテストされるでしょう。

次は、関数型言語に影響された Rust 機能を一部探究します: クロージャとイテレータです。

第 13 章

関数型言語の機能: イテレータとクロージャ

Rust の設計は、多くの既存の言語やテクニックにインスピレーションを得ていて、その一つの大きな影響が**関数型プログラミング**です。関数型でのプログラミングには、しばしば、引数で渡したり、関数から関数を返したり、関数を後ほど使用するために変数に代入することで関数を値として使用することが含まれます。

この章では、関数型プログラミングがどんなものであったり、なかったりするかという問題については議論しませんが、代わりに関数型とよく言及される多くの言語の機能に似た Rust の機能の一部について議論しましょう。

具体的には、以下を講義します:

- **クロージャ**、変数に保存できる関数に似た文法要素
- **イテレータ**、一連の要素を処理する方法
- クロージャとイテレータを使用して第 12 章の入出力プロジェクトを改善する方法
- クロージャとイテレータのパフォーマンス (ネタバレ: 思ったよりも速いです)

パターンマッチングや enum なども関数型スタイルに影響を受けた他の Rust の機能ですが、これらについてはすでに他の章で講義してきました。クロージャとイテレータをマスターすることは、慣用的で速い Rust コードを書くことの重要な部分なので、この章を丸ごと捧げます。

13.1 クロージャ: 環境をキャプチャする匿名関数

Rust のクロージャは、変数に保存したり、引数として他の関数に渡すことのできる匿名関数です。ある場所でクロージャを生成した後、他の場所でクロージャを呼び出して、別の文脈で評価することができます。クロージャは関数と異なり、定義されたスコープから値をキャプチャすることができま

す。これらのクロージャの特性を使用して、どのようにコードの再利用や振る舞いのカスタマイズができるか、実際に試してみましょう。

13.1.1　クロージャで環境をキャプチャする

まず、クロージャから定義された環境から後で使用するために値をキャプチャするために、どのようにクロージャを使用することができるのか、確かめてみましょう。このようなシナリオを考えます: 私たちの T シャツ会社は時折プロモーションとして、メーリングリスト上の誰かに、その人専用の限定版のシャツを配ります。メーリングリスト上の人たちは、自身のプロファイルにお気に入りの色を任意で追加することができます。シャツ無料対象に選ばれた人がお気に入りの色を設定している場合は、その色のシャツをもらえます。その人がお気に入りの色を指定していない場合は、会社が現在最も多く持っている色をもらえます。

これを実装する方法はたくさんあります。この例では、`Red` と `Blue` の列挙子を持つ、`ShirtColor` という enum を使用します (簡潔さのために入手可能な色の数は制限しています)。会社の棚卸資産は、`Vec<ShirtColor>` を含み現在在庫にあるシャツの色を表現する `shirts` というフィールドを持つ、`Inventory` 構造体で表現します。`Inventory` 上に定義された `giveaway` メソッドは、シャツ無料対象者の任意選択のシャツの色の好みを取得し、その人がもらえるシャツの色を返します。この構築をリスト 13-1 に示します:

ファイル名: src/main.rs

```
#[derive(Debug, PartialEq, Copy, Clone)]
enum ShirtColor {
    Red,
    Blue,
}

struct Inventory {
    shirts: Vec<ShirtColor>,
}

impl Inventory {
    fn giveaway(&self, user_preference: Option<ShirtColor>) -> ShirtColor {
        user_preference.unwrap_or_else(|| self.most_stocked())
    }

    fn most_stocked(&self) -> ShirtColor {
        let mut num_red = 0;
        let mut num_blue = 0;

        for color in &self.shirts {
            match color {
                ShirtColor::Red => num_red += 1,
                ShirtColor::Blue => num_blue += 1,
            }
        }
```

```
        if num_red > num_blue {
            ShirtColor::Red
        } else {
            ShirtColor::Blue
        }
    }
}

fn main() {
    let store = Inventory {
        shirts: vec![ShirtColor::Blue, ShirtColor::Red, ShirtColor::Blue],
    };

    let user_pref1 = Some(ShirtColor::Red);
    let giveaway1 = store.giveaway(user_pref1);
    println!(
        // "好み{:?}を持つユーザは{:?}を得ます"
        "The user with preference {:?} gets {:?}",
        user_pref1, giveaway1
    );

    let user_pref2 = None;
    let giveaway2 = store.giveaway(user_pref2);
    println!(
        // "好み{:?}を持つユーザは{:?}を得ます"
        "The user with preference {:?} gets {:?}",
        user_pref2, giveaway2
    );
}
```

リスト 13-1: シャツ会社の無料プレゼントのシチュエーション

`main` で定義されている `store` には、この限定版プロモーションのために残された、青いシャツが 2 着と赤いシャツが 1 着あります。赤いシャツが好みのユーザと、好みがないユーザを対象として、`giveaway` メソッドを呼び出しています。

繰り返しになりますが、このコードは多くの方法で実装することができます。ここではクロージャに着目するために、すでに学習した概念に固執しています。ただし、`giveaway` メソッドの本体はクロージャを使用していて、例外です。`giveaway` メソッドでは、ユーザの好みを `Option<ShirtColor>` 型の引数として取り、`user_preference` に対して `unwrap_or_else` メソッドを呼び出します。`Option<T>` に対する `unwrap_or_else` メソッドは標準ライブラリで定義されています。このメソッドは引数を 1 つ取ります: 値 `T` (`Option<T>` の `Some` 列挙子内に保存されているのと同じ型、この場合は `ShirtColor`) を返す、引数のないクロージャです。もし `Option<T>` が `Some` 列挙子の場合、`unwrap_or_else` は `Some` の内側の値を返します。もし `Option<T>` が `None` 列挙子の場合、`unwrap_or_else` はクロージャを呼び出し、そのクロージャによって返された値を返します。

`unwrap_or_else` への引数としてクロージャ式 `|| self.most_stocked()` を指定しています。これは引数を取らないクロージャです (クロージャが引数を持つ場合、それらは 2 本の縦線の間に現れ

ます)。クロージャの本体は `self.most_stocked()` を呼び出します。クロージャはここで定義され、`unwrap_or_else` の実装詳細がクロージャの結果を必要とするなら、このクロージャを呼び出すでしょう。

このコードを実行すると、以下が出力されます:

```
$ cargo run
   Compiling shirt-company v0.1.0 (file:///projects/shirt-company)
    Finished dev [unoptimized + debuginfo] target(s) in 0.27s
     Running `target/debug/shirt-company`
The user with preference Some(Red) gets Red
The user with preference None gets Blue
```

ここで興味深い側面は、現在の `Inventory` インスタンスに対して `self.most_stocked()` を呼び出すクロージャを渡していることです。標準ライブラリは、プログラマが定義した `Inventory` や `ShirtColor` 型や、このシナリオで使用したいロジックについては何も知りません。クロージャは `self` `Inventory` インスタンスへの不変参照をキャプチャし、この不変参照をプログラマが指定したコードとともに `unwrap_or_else` メソッドに渡します。一方で関数は、このように環境をキャプチャすることができません。

13.1.2　クロージャの型推論と注釈

関数とクロージャの違いは他にもあります。クロージャでは通常、`fn` 関数のように引数の型や戻り値の型を注釈する必要はありません。関数では、型注釈は必要です。型はユーザに露出する明示的なインターフェイスの一部だからです。このインターフェイスを堅実に定義することは、関数が使用したり、返したりする値の型についてみんなが合意していることを保証するために重要なのです。一方で、クロージャはこのような露出するインターフェイスには使用されません: 変数に保存され、名前付けしたり、ライブラリの使用者に晒されることなく、使用されます。

クロージャは典型的には短く、あらゆる任意の筋書きではなく、狭い文脈でのみ関係します。このような限定された文脈内では、コンパイラは、多くの変数の型を推論できるのと似たように、引数や戻り値の型を推論することができます (コンパイラがクロージャ型注釈を要求するレアケースもあります)。

本当に必要な以上に冗長になることと引き換えに、明示性と明瞭性を向上させたいなら、変数に型注釈を加えることもできます; クロージャに対する型注釈は、リスト 13-2 に示した定義のようになるでしょう。この例では、リスト 13-1 でやったようにクロージャを定義したその場で引数として渡すのではなく、クロージャを定義して変数に保存しています。

ファイル名: src/main.rs

```
# use std::thread;
# use std::time::Duration;
#
```

```
# fn generate_workout(intensity: u32, random_number: u32) {
    let expensive_closure = |num: u32| -> u32 {
        println!("calculating slowly...");
        thread::sleep(Duration::from_secs(2));
        num
    };
#
#     if intensity < 25 {
#         println!("Today, do {} pushups!", expensive_closure(intensity));
#         println!("Next, do {} situps!", expensive_closure(intensity));
#     } else {
#         if random_number == 3 {
#             println!("Take a break today! Remember to stay hydrated!");
#         } else {
#             println!(
#                 "Today, run for {} minutes!",
#                 expensive_closure(intensity)
#             );
#         }
#     }
# }
#
# fn main() {
#     let simulated_user_specified_value = 10;
#     let simulated_random_number = 7;
#
#     generate_workout(simulated_user_specified_value, simulated_random_number);
# }
```

リスト 13-2: クロージャの引数と戻り値の省略可能な型注釈を追加する

型注釈を付け加えると、クロージャの記法は関数の記法とより似た見た目になっていきます。以下は比較のために、引数に 1 を加える関数と、同じ振る舞いをするクロージャを定義しています。空白を追加して、関連のある部分を並べています。これにより、縦棒の使用と省略可能な記法の量を除いて、クロージャ記法が関数記法に似ているところを説明しています。

```
fn  add_one_v1   (x: u32) -> u32 { x + 1 }
let add_one_v2 = |x: u32| -> u32 { x + 1 };
let add_one_v3 = |x|             { x + 1 };
let add_one_v4 = |x|               x + 1  ;
```

1 行目が関数定義を示し、2 行目がフルに注釈したクロージャ定義を示しています。3 行目ではクロージャ定義から型注釈を取り除いています。4 行目ではかっこを取り除いてます。クロージャの本体がただ 1 つの式からなるので、このかっこは省略可能です。これらは全て、呼び出された時に同じ振る舞いになる合法な定義です。型はその使い方から推論されるので、`add_one_v3` と `add_one_v4` 行がコンパイルできるようになるためには、それを評価する必要があります。これは、コンパイラが型を推論できるようにするために、型注釈か `Vec` の中に挿入する何らかの型の値を必要とする、

`let v = Vec::new();` と似ています。

クロージャ定義では、コンパイラは引数それぞれと戻り値に対して、一つの具体的な型を推論します。例えば、リスト 13-3 に引数として受け取った値を返すだけの短いクロージャの定義を示しました。このクロージャは、この例での目的以外には有用ではありません。この定義には、何も型注釈を加えていないことに注意してください。型注釈が無いので、任意の型に対してこのクロージャを呼び出すことができ、ここでは最初は `String` に対して呼び出しています。その後整数に対して `example_closure` を呼び出そうとすると、エラーになります。

ファイル名: src/main.rs

```
# fn main() {
    let example_closure = |x| x;

    let s = example_closure(String::from("hello"));
    let n = example_closure(5);
# }
```

リスト 13-3: 2 つの異なる型で型が推論されるクロージャの呼び出しを試みる

コンパイラは、次のエラーを返します:

```
$ cargo run
   Compiling closure-example v0.1.0 (file:///projects/closure-example)
error[E0308]: mismatched types
(エラー: 型が合いません)
 --> src/main.rs:5:29
  |
5 |     let n = example_closure(5);
  |             --------------- ^- help: try using a conversion method: `.
      to_string()`
  |             |               | (ヘルプ: 変換メソッド`.to_string()`を使用して
      みてください)
  |             |               |
  |             |               expected `String`, found integer
  |             |              (`String`を予期していましたが、整数が見つかりまし
      た)
  |             arguments to this function are incorrect
  |            (この関数への引数が正しくありません)
  |
note: expected because the closure was earlier called with an argument of type `
    String`
(注釈: クロージャは以前に型`String`を持つ引数とともに呼ばれているため、これが予
    期されています)
 --> src/main.rs:4:29
  |
4 |     let s = example_closure(String::from("hello"));
  |             --------------- ^^^^^^^^^^^^^^^^^^^^^^ expected because this
      argument is of type `String`
  |             |                                      (この引数が型`String`を持つ
```

```
       ことから予期されています)
  |             |
  |             in this closure call
  |            (このクロージャ呼び出しで)
note: closure parameter defined here
 --> src/main.rs:2:28
  |
2 |     let example_closure = |x| x;
  |                           ^

For more information about this error, try `rustc --explain E0308`.
error: could not compile `closure-example` (bin "closure-example") due to 1
    previous error
```

`String` 値で `example_closure` を呼び出した最初の時点で、コンパイラは `x` とクロージャの戻り値の型を `String` と推論します。そして、その型が `example_closure` のクロージャに閉じ込められ、次に同じクロージャを異なる型で使用しようとすると、型エラーが出るのです。

13.1.3　参照をキャプチャするか、所有権を移動するか

クロージャは、3 つの方法で環境から値をキャプチャでき、この方法は関数が引数を取れる 3 つの方法に直に対応します: 不変で借用する、可変で借用する、所有権を奪うです。クロージャは、関数本体がキャプチャされた値を使って何をするかに基づいて、これらのうちどれを使用するかを決定します。

リスト 13-4 では、`list` というベクタへの不変参照をキャプチャするクロージャが定義されます。値を出力するためには不変参照しか必要としないからです:

ファイル名: src/main.rs

```
fn main() {
    let list = vec![1, 2, 3];
    //       "クロージャの定義前: {:?}"
    println!("Before defining closure: {:?}", list);

    //                              "クロージャから: {:?}"
    let only_borrows = || println!("From closure: {:?}", list);

    //       "クロージャの呼び出し前: {:?}"
    println!("Before calling closure: {:?}", list);
    only_borrows();
    //       "クロージャの呼び出し後: {:?}"
    println!("After calling closure: {:?}", list);
}
```

リスト 13-4: 不変参照をキャプチャするクロージャを定義し、呼び出す

この例は、変数をクロージャ定義に束縛することができ、その変数名が関数名であるかのように変

数名と丸かっこを使用することで、後でそのクロージャを呼び出すことができることも示しています。

`list` への不変参照は複数同時に存在することができるので、`list` には、クロージャ定義より前のコードからでも、クロージャ定義からクロージャ呼び出しまでの間のコードからでも、クロージャ呼び出し後のコードからでも、アクセスすることができます。このコードはコンパイルでき、実行でき、以下を出力します:

```
$ cargo run
   Compiling closure-example v0.1.0 (file:///projects/closure-example)
    Finished dev [unoptimized + debuginfo] target(s) in 0.43s
     Running `target/debug/closure-example`
Before defining closure: [1, 2, 3]
Before calling closure: [1, 2, 3]
From closure: [1, 2, 3]
After calling closure: [1, 2, 3]
```

次にリスト 13-5 では、`list` ベクタへ要素を追加するようにクロージャ本体を変更します。クロージャは可変参照をキャプチャするようになります:

ファイル名: src/main.rs

```
fn main() {
    let mut list = vec![1, 2, 3];
    //       "クロージャの定義前: {:?}"
    println!("Before defining closure: {:?}", list);

    let mut borrows_mutably = || list.push(7);

    borrows_mutably();
    //       "クロージャの呼び出し後: {:?}"
    println!("After calling closure: {:?}", list);
}
```

リスト 13-5: 可変参照をキャプチャするクロージャを定義し、呼び出す

このコードはコンパイルでき、実行でき、以下を出力します:

```
$ cargo run
   Compiling closure-example v0.1.0 (file:///projects/closure-example)
    Finished dev [unoptimized + debuginfo] target(s) in 0.43s
     Running `target/debug/closure-example`
Before defining closure: [1, 2, 3]
After calling closure: [1, 2, 3, 7]
```

`borrows_mutably` クロージャの定義と呼び出しの間の `println!` を消したことに注意してください: `borrows_mutably` が定義されるときに、`list` への可変参照がキャプチャされます。クロージャを呼び出した後は、クロージャを再度使用していないので、可変借用は終了します。可変借用が存在するときに他の借用は許可されないので、クロージャ定義とクロージャ呼び出しの間に、出力するための不

変借用は許可されません。println! を追加してみて、どんなエラーメッセージが得られるか確認してみてください！

クロージャの本体が厳密には所有権を必要としない場合であっても、クロージャにそれが使用する値の所有権を環境から奪うように強制したい場合は、引数リストの前で move キーワードを使用することができます。

この手法は主に、新しいスレッドにクロージャを渡すときに、データが新しいスレッドに所有されるようにムーブするために有用です。スレッドについて、そしてなぜスレッドを使用するのかについては、第 16 章で並行性について語るときに詳しく議論しますが、今のところは、move キーワードが必要なクロージャを使って新しいスレッドを生成することについて簡単に探検してみましょう。リスト 13-6 に、メインスレッド内ではなく新しいスレッド内でベクタを出力するようにリスト 13-4 を変更したものを示します:

ファイル名: src/main.rs

```
use std::thread;

fn main() {
    let list = vec![1, 2, 3];
    //       "クロージャの定義前: {:?}"
    println!("Before defining closure: {:?}", list);

    //                               "スレッドから: {:?}"
    thread::spawn(move || println!("From thread: {:?}", list))
        .join()
        .unwrap();
}
```

リスト 13-6: move を使用して、スレッドのためのクロージャに list の所有権を奪わせる

実行するクロージャを引数としてスレッドに与えて、新しいスレッドを生成します。クロージャ本体はリストを表示します。リスト 13-4 では、クロージャは不変参照を使って list をキャプチャしているだけでした。それが list を出力するために必要な最小のアクセスだからです。この例では、クロージャ本体はやはり不変参照のみを要求しますが、クロージャ定義の先頭に move キーワードを置くことで、list をクロージャ内にムーブすべきだと指定する必要があります。新しいスレッドはメインスレッドの残りの部分が終了するより前に終了するかもしれませんし、メインスレッドが先に終了するかもしれません。もしメインスレッドが list の所有権を持ち続け、新しいスレッドが終了する前に終了して list をドロップしてしまうと、スレッド内の不変参照は無効になるでしょう。そのためコンパイラは、参照が有効になるように、新しいスレッドに渡されるクロージャに list がムーブされることを要求します。move キーワードを削除したり、メインスレッド内でクロージャが定義された後に list を使用したりしてみて、どんなコンパイルエラーが得られるか確認してみてください！

13.1.4　キャプチャされた値のクロージャからのムーブと、Fn 系トレイト

クロージャが参照をキャプチャしたり、クロージャが定義されている環境から値の所有権をキャプチャする (つまりクロージャの**中に**何をムーブするかに影響を与えます) と、クロージャの本体内のコードは、後でクロージャが評価されたときにその参照または値に何が起こるかを定義します (つまりクロージャの**外に**何をムーブするかに影響を与えます)。クロージャ本体は以下のいずれかを行うことができます: キャプチャされた値をクロージャの外にムーブしたり、キャプチャされた値を変更したり、値をムーブも変更もしないでいたり、そもそも環境から何もキャプチャしないかです。

クロージャがどのように環境から値をキャプチャして使用するかは、クロージャがどのトレイトを実装するかに影響を及ぼします。トレイトは、関数や構造体がどんな種類のクロージャを使用できるかを指定するための方法として使用されます。クロージャは、その本体はどのように値を使用するかに応じて、以下の `Fn` 系トレイトのうち 1 つ、2 つ、または 3 つすべてを、付加的に自動的に実装します:

1. `FnOnce` は、一度呼び出すことができるクロージャに適用されます。すべてのクロージャは、呼び出すことができるので、最低でもこのトレイトを実装します。キャプチャされた値を本体の外にムーブするクロージャは、一度しか呼び出すことができないので、`FnOnce` のみを実装し、他の `Fn` 系トレイト群を実装しないことになるでしょう。
2. `FnMut` は、キャプチャされた値を本体の外にムーブしないが、キャプチャされた値を変更するかもしれないクロージャに適用されます。これらのクロージャは複数回呼び出すことができます。
3. `Fn` は、キャプチャされた値を本体の外にムーブせず、キャプチャされた値を変更しないクロージャと、環境から何もキャプチャしないクロージャに適用されます。これらのクロージャは環境を変更することなく複数回呼び出すことができ、クロージャを並行して複数回呼び出す場合などに重要です。

リスト 13-1 で使用した、`Option<T>` に対する `unwrap_or_else` メソッドの定義を見てみましょう:

```
impl<T> Option<T> {
    pub fn unwrap_or_else<F>(self, f: F) -> T
    where
        F: FnOnce() -> T
    {
        match self {
            Some(x) => x,
            None => f(),
        }
    }
}
```

`T` は `Option` の `Some` 列挙子内の値の型を表現するジェネリック型であることを思い出してく

ださい。型 `T` は `unwrap_or_else` 関数の戻り値型でもあります: 例えば `Option<String>` に対して `unwrap_or_else` を呼び出すコードは、`String` を得るでしょう。

次に、`unwrap_or_else` 関数は追加のジェネリック型引数 `F` を持つことに注目してください。型 `F` は引数 `f` の型で、`f` は `unwrap_or_else` を呼び出すときにこちらから提供するクロージャです。

`FnOnce() -> T` はジェネリック型 `F` に対して指定されているトレイト境界で、これは、`F` は一度呼び出すことができ、引数を取らず、`T` を返すことを意味します。トレイト境界で `FnOnce` を使用することは、`unwrap_or_else` は `f` を多くとも 1 回しか呼び出さないという制約を表現します。`unwrap_or_else` の本体の中では、`Option` が `Some` の場合は `f` は呼ばれないことがわかるでしょう。`Option` が `None` の場合は、`f` は一度呼び出されるでしょう。すべてのクロージャは `FnOnce` を実装するので、`unwrap_or_else` は最も多くの種類のクロージャを受け入れ、最も柔軟です。

> 注釈: 関数も 3 つの `Fn` 系トレイト全部を実装することができます。もし環境から値をキャプチャする必要がなければ、`Fn` 系トレイトのいずれかを実装する何かが必要になるクロージャではなく、関数の名前を使用できます。例えば、`Option<Vec<T>>` 値に対して、その値が `None` である場合には新しい空のベクタを得るためには、`unwrap_or_else(Vec::new)` と呼び出すことができます。

それでは、スライスに対して定義されている標準ライブラリメソッド `sort_by_key` を見てみて、`unwrap_or_else` とどう違うのか、`sort_by_key` はなぜトレイト境界に `FnOnce` ではなく `FnMut` を使用するのか、確認してみましょう。このクロージャは、処理対象のスライス内の現在の要素への参照という形で 1 つの引数を取り、順序付けることができる `K` 型の値を返します。この関数は、スライスを各要素の特定の属性によってソートしたいときに便利です。リスト 13-7 では、`Rectangle` インスタンスからなるリストがあり、`width` 属性によってその各要素を昇順に並び替えるために、`sort_by_key` を使用しています:

ファイル名: src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let mut list = [
        Rectangle { width: 10, height: 1 },
        Rectangle { width: 3, height: 5 },
        Rectangle { width: 7, height: 12 },
    ];

    list.sort_by_key(|r| r.width);
    println!("{:#?}", list);
}
```

リスト 13-7: `sort_by_key` を使用して、幅で長方形を順序付ける

このコードは以下を出力します:

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished dev [unoptimized + debuginfo] target(s) in 0.41s
     Running `target/debug/rectangles`
[
    Rectangle {
        width: 3,
        height: 5,
    },
    Rectangle {
        width: 7,
        height: 12,
    },
    Rectangle {
        width: 10,
        height: 1,
    },
]
```

`sort_by_key` が `FnMut` クロージャを取るように定義されている理由は、クロージャを複数回呼び出すからです: スライス内の各要素ごとに一度呼び出されます。クロージャ `|r| r.width` は、環境から何もキャプチャ、変更、ムーブしないので、トレイト境界の要求を満たしています。

対照的にリスト 13-8 では、環境から値をムーブすることから、`FnOnce` トレイトだけを実装するクロージャの例を示しています。コンパイラは、`sort_by_key` でこのクロージャを使わせてくれないでしょう:

ファイル名: src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let mut list = [
        Rectangle { width: 10, height: 1 },
        Rectangle { width: 3, height: 5 },
        Rectangle { width: 7, height: 12 },
    ];

    let mut sort_operations = vec![];
    //                        "by keyが呼ばれました"
```

```
    let value = String::from("by key called");

    list.sort_by_key(|r| {
        sort_operations.push(value);
        r.width
    });
    println!("{:#?}", list);
}
```

リスト 13-8: sort_by_key で FnOnce クロージャを使用しようとする

これは、list をソートするときに sort_by_key が呼ばれる回数を数えようとする、人為的で複雑な (しかも動かない) 方法です。このコードは、sort_operations ベクタに value —クロージャの環境からの String —をプッシュすることで、このカウントを行おうとします。このクロージャは value をキャプチャし、value の所有権を sort_operations ベクタに移転することで、value をクロージャの外にムーブします。このクロージャは一度呼び出すことができます; 2 回目の呼び出しをしようとしても、sort_operations にもう一度プッシュするための value はもう環境にないので、うまくいかないでしょう！ そのため、このクロージャは FnOnce のみを実装します。このコードをコンパイルしようとすると、クロージャが FnMut を実装していなくてはならないので、value はクロージャの外にムーブすることができない、というエラーが得られます:

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
error[E0507]: cannot move out of `value`, a captured variable in an `FnMut`
    closure
(エラー: `FnMut`クロージャ内にキャプチャされた変数`value`からムーブすることはで
    きません)
  --> src/main.rs:18:30
   |
15 |     let value = String::from("by key called");
   |         ----- captured outer variable
   |              (キャプチャされる外側の変数)
16 |
17 |     list.sort_by_key(|r| {
   |                      --- captured by this `FnMut` closure
   |                         (この`FnMut`クロージャによってキャプチャされていま
       す)
18 |         sort_operations.push(value);
   |                              ^^^^^ move occurs because `value` has type `
       String`, which does not implement the `Copy` trait
   |                                    (`value`は型`String`を持ち、`Copy`トレイ
       トを実装しないので、ムーブが発生します)

For more information about this error, try `rustc --explain E0507`.
error: could not compile `rectangles` (bin "rectangles") due to 1 previous error
```

エラーは、クロージャ本体内で、value を環境からムーブする行を指しています。これを修正するためには、値を環境からムーブしないようにクロージャ本体を変更する必要があります。sort_by_key

が呼ばれる回数を数えるためには、環境内でカウンタを保持し、クロージャ本体の中でその値をインクリメントするのが、より自然な方法です。リスト 13-9 のクロージャは、num_sort_operations カウンタへの可変参照をキャプチャするだけで、そのため複数回呼び出すことができるので、sort_by_key で使うことができます:

ファイル名: src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let mut list = [
        Rectangle { width: 10, height: 1 },
        Rectangle { width: 3, height: 5 },
        Rectangle { width: 7, height: 12 },
    ];

    let mut num_sort_operations = 0;
    list.sort_by_key(|r| {
        num_sort_operations += 1;
        r.width
    });
    //       "{:#?}、{num_sort_operations}回の操作でソートされました"
    println!("{:#?}, sorted in {num_sort_operations} operations", list);
}
```

リスト 13-9: sort_by_key で FnMut クロージャを使用することは許可される

クロージャを使用する関数や型を定義または使用するときに、Fn 系トレイトは重要です。次の節では、イテレータについて議論します。多くのイテレータメソッドはクロージャ引数を取るので、続けるにあたって、これらのクロージャの詳細のことを覚えておいてください！

13.2　一連の要素をイテレータで処理する

イテレータパターンにより、一連の要素に順番に何らかの作業を行うことができます。イテレータは、各要素を繰り返し、シーケンスが終わったことを決定するロジックの責任を負います。イテレータを使用すると、自身でそのロジックを再実装する必要がなくなるのです。

Rust において、イテレータは**怠惰**です。つまり、イテレータを使い込んで消費するメソッドを呼ぶまで何の効果もないということです。例えば、リスト 13-10 のコードは、Vec<T> に定義された iter メソッドを呼ぶことで v1 ベクタの要素に対するイテレータを生成しています。このコード単独では、何も有用なことはしません。

```
# fn main() {
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();
# }
```

リスト 13-10: イテレータを生成する

イテレータは v1_iter 変数に保存されます。一旦イテレータを生成したら、いろんな手段で使用することができます。第 3 章のリスト 3-5 では、for ループを使用して配列を走査し、各要素に対して何らかのコードを実行しました。今までそれが正確にはどう機能するのかごまかしてきましたが、裏で暗黙にイテレータを作成して消費していたのです。

リスト 13-11 の例では、イテレータの生成と for ループでイテレータを使用することを区別しています。v1_iter のイテレータで for ループが呼び出された時に、イテレータの各要素がループの繰り返しで使用され、各値が出力されます。

```
# fn main() {
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();

    for val in v1_iter {
        println!("Got: {}", val);
    }
# }
```

リスト 13-11: for ループでイテレータを使用する

標準ライブラリにより提供されるイテレータが存在しない言語では、変数を添え字 0 から始め、その変数でベクタに添え字アクセスして値を得て、ベクタの総要素数に到達するまでループでその変数の値をインクリメントすることで、この同じ機能を書く可能性が高いでしょう。

イテレータはそのロジック全てを処理してくれるので、めちゃくちゃにしてしまう可能性のあるコードの繰り返しを減らしてくれます。イテレータにより、添え字を使えるデータ構造、ベクタなどだけではなく、多くの異なるシーケンスに対して同じロジックを使う柔軟性も得られます。イテレータがそれをする方法を調査しましょう。

13.2.1　Iterator トレイトと next メソッド

全てのイテレータは、標準ライブラリで定義されている Iterator というトレイトを実装しています。このトレイトの定義は、以下のようになっています:

```
pub trait Iterator {
    type Item;
```

```
    fn next(&mut self) -> Option<Self::Item>;

    // デフォルト実装のあるメソッドは省略
    // methods with default implementations elided
}
```

この定義は新しい記法を使用していることに注目してください: `type Item` と `Self::Item` で、これらはこのトレイトとの**関連型** (associated type) を定義しています。関連型についての詳細は、第 19 章で語ります。とりあえず、知っておく必要があることは、このコードが `Iterator` トレイトを実装するには、`Item` 型も定義する必要があり、そして、この `Item` 型が `next` メソッドの戻り値の型に使われていると述べていることです。換言すれば、`Item` 型がイテレータから返ってくる型になるだろうということです。

`Iterator` トレイトは、一つのメソッドを定義することを実装者に要求することだけします: `next` メソッドで、これは 1 度に `Some` に包まれたイテレータの 1 要素を返し、繰り返しが終わったら、`None` を返します。

イテレータに対して直接 `next` メソッドを呼び出すこともできます; リスト 13-12 は、ベクタから生成されたイテレータの `next` を繰り返し呼び出した時にどんな値が返るかを模擬しています。

ファイル名: src/lib.rs

```
# #[cfg(test)]
# mod tests {
    #[test]
    fn iterator_demonstration() {
        let v1 = vec![1, 2, 3];

        let mut v1_iter = v1.iter();

        assert_eq!(v1_iter.next(), Some(&1));
        assert_eq!(v1_iter.next(), Some(&2));
        assert_eq!(v1_iter.next(), Some(&3));
        assert_eq!(v1_iter.next(), None);
    }
# }
```

リスト 13-12: イテレータに対して `next` メソッドを呼び出す

`v1_iter` を可変にする必要があったことに注目してください: イテレータの `next` メソッドを呼び出すと、今シーケンスのどこにいるかを追いかけるためにイテレータが使用している内部の状態が変わります。つまり、このコードはイテレータを**消費**、または使い込むのです。`next` の各呼び出しは、イテレータの要素を一つ、食います。`for` ループを使用した時には、`v1_iter` を可変にする必要はありませんでした。というのも、ループが `v1_iter` の所有権を奪い、陰で可変にしていたからです。

また、`next` の呼び出しで得られる値は、ベクタの値への不変な参照であることにも注目してくださ

い。iter メソッドは、不変参照へのイテレータを生成します。v1 の所有権を奪い、所有された値を返すイテレータを生成したいなら、iter ではなく into_iter を呼び出すことができます。同様に、可変参照を繰り返したいなら、iter ではなく iter_mut を呼び出せます。

13.2.2　イテレータを消費するメソッド

Iterator トレイトには、標準ライブラリが提供してくれているデフォルト実装のある多くの異なるメソッドがあります; Iterator トレイトの標準ライブラリの API ドキュメントを検索することで、これらのメソッドについて知ることができます。これらのメソッドの中には、定義内で next メソッドを呼ぶものもあり、故に Iterator トレイトを実装する際には、next メソッドを実装する必要があるのです。

next を呼び出すメソッドは、**消費アダプタ** (consuming adaptors) と呼ばれます。呼び出しがイテレータの使い込みになるからです。一例は、sum メソッドで、これはイテレータの所有権を奪い、next を繰り返し呼び出すことで要素を繰り返し、故にイテレータを消費するのです。繰り返しが進むごとに、各要素を一時的な合計に追加し、繰り返しが完了したら、その合計を返します。リスト 13-13 は、sum の使用を説明したテストです:

ファイル名: src/lib.rs

```
# #[cfg(test)]
# mod tests {
    #[test]
    fn iterator_sum() {
        let v1 = vec![1, 2, 3];

        let v1_iter = v1.iter();

        let total: i32 = v1_iter.sum();

        assert_eq!(total, 6);
    }
# }
```

リスト 13-13:　sum メソッドを呼び出してイテレータの全要素の合計を得る

sum は呼び出し対象のイテレータの所有権を奪うので、sum 呼び出し後に v1_iter を使用することはできません。

13.2.3　他のイテレータを生成するメソッド

イテレータアダプタ (iterator adaptors) は、Iterator トレイト上で定義されたメソッドのうち、イテレータを消費しないものです。これらはイテレータを消費しない代わりに、元のイテレータの一部の様相を変更することによって、別のイテレータを生成します。

リスト 13-14 は、イテレータアダプタメソッドの map の呼び出し例を示しています。map は、各要素が反復処理されるときに、その要素に対して呼び出すクロージャを取ります。map メソッドは、修正された要素を生成する新しいイテレータを返します。map 新しいイテレータを生成します。ここのクロージャは、ベクタの各要素が 1 インクリメントされる新しいイテレータを作成します:

ファイル名: src/main.rs

```
# fn main() {
    let v1: Vec<i32> = vec![1, 2, 3];

    v1.iter().map(|x| x + 1);
# }
```

リスト 13-14: イテレータアダプタの map を呼び出して新規イテレータを作成する

ところが、このコードは警告を発します:

```
$ cargo run
   Compiling iterators v0.1.0 (file:///projects/iterators)
warning: unused `Map` that must be used
(警告: 使用されねばならない`Map`が未使用です)
 --> src/main.rs:4:5
  |
4 |     v1.iter().map(|x| x + 1);
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: iterators are lazy and do nothing unless consumed
   (メモ: イテレータは怠惰で、消費されるまで何もしません)
  = note: `#[warn(unused_must_use)]` on by default
help: use `let _ = ...` to ignore the resulting value
  |
4 |     let _ = v1.iter().map(|x| x + 1);
  |     +++++++

warning: `iterators` (bin "iterators") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 0.47s
     Running `target/debug/iterators`
```

リスト 13-14 のコードは何もしません; 指定したクロージャは、決して呼ばれないのです。警告が理由を思い出させてくれています: イテレータアダプタは怠惰で、ここでイテレータを消費する必要があるのです。

この警告を修正し、イテレータを消費するには、collect メソッドを使用しますが、これは第 12 章のリスト 12-1 で env::args とともに使用しました。このメソッドはイテレータを消費し、結果の値をコレクションデータ型に集結させます。

リスト 13-15 において、map 呼び出しから返ってきたイテレータを繰り返した結果をベクタに集結させています。このベクタは、最終的に元のベクタの各要素に 1 を足したものが含まれます。

ファイル名: src/main.rs

```
# fn main() {
    let v1: Vec<i32> = vec![1, 2, 3];

    let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();

    assert_eq!(v2, vec![2, 3, 4]);
# }
```

リスト 13-15: map メソッドを呼び出して新規イテレータを作成し、それから collect メソッドを呼び出してその新規イテレータを消費し、ベクタを生成する

map はクロージャを取るので、各要素に対して行いたいどんな処理も指定することができます。これは、Iterator トレイトが提供する繰り返し動作を再利用しつつ、クロージャにより一部の動作をカスタマイズできる好例になっています。

複雑な操作を可読性の高い方法で行うために、イテレータアダプタの呼び出しを複数チェーンすることができます。ですが、すべてのイテレータは怠惰なので、イテレータアダプタの呼び出しから結果を得るには、消費アダプタメソッドのうちいずれかを呼ぶ必要があります。

13.2.4　環境をキャプチャするクロージャを使用する

多くのイテレータアダプタは引数としてクロージャを取り、また、イテレータアダプタに引数として指定するクロージャは、環境をキャプチャするクロージャであることは、よくあることでしょう。

この例では、クロージャを取る filter メソッドを使用します。クロージャは、イテレータからの要素を取り、bool を返すクロージャです。このクロージャが true を返せば、filter が生成する繰り返し処理にその値が含まれます。クロージャが false を返したら、その値は含まれません。

リスト 13-16 では、環境から shoe_size 変数をキャプチャするクロージャで filter を使って、Shoe 構造体インスタンスのコレクションを繰り返しています。指定したサイズの靴だけを返すわけです。

ファイル名: src/lib.rs

```
#[derive(PartialEq, Debug)]
struct Shoe {
    size: u32,
    style: String,
}

fn shoes_in_size(shoes: Vec<Shoe>, shoe_size: u32) -> Vec<Shoe> {
    shoes.into_iter().filter(|s| s.size == shoe_size).collect()
}

#[cfg(test)]
```

```
mod tests {
    use super::*;

    #[test]
    fn filters_by_size() {
        let shoes = vec![
            Shoe {
                size: 10,
                style: String::from("sneaker"),
            },
            Shoe {
                size: 13,
                style: String::from("sandal"),
            },
            Shoe {
                size: 10,
                style: String::from("boot"),
            },
        ];

        let in_my_size = shoes_in_size(shoes, 10);

        assert_eq!(
            in_my_size,
            vec![
                Shoe {
                    size: 10,
                    style: String::from("sneaker")
                },
                Shoe {
                    size: 10,
                    style: String::from("boot")
                },
            ]
        );
    }
}
```

リスト 13-16: `shoe_size` をキャプチャするクロージャで `filter` メソッドを使用する

`shoes_in_size` 関数は、引数として靴のベクタとサイズの所有権を奪います。指定されたサイズの靴だけを含むベクタを返します。

`shoes_in_size` の本体で、`into_iter` を呼び出してベクタの所有権を奪うイテレータを作成しています。そして、`filter` を呼び出してそのイテレータをクロージャが `true` を返した要素だけを含む新しいイテレータに適合させます。

クロージャは、環境から `shoe_size` 引数をキャプチャし、指定されたサイズの靴だけを保持しながら、その値を各靴のサイズと比較します。最後に、`collect` を呼び出すと、関数により返ってきたベクタに適合させたイテレータから返ってきた値が集まるのです。

shoes_in_size を呼び出した時に、指定した値と同じサイズの靴だけが得られることをテストは示しています。

13.3　入出力プロジェクトを改善する

このイテレータに関する新しい知識があれば、イテレータを使用してコードのいろんな場所をより明確で簡潔にすることで、第 12 章の入出力プロジェクトを改善することができます。イテレータが Config::build 関数と search 関数の実装を改善する方法に目を向けましょう。

13.3.1　イテレータを使用して clone を取り除く

リスト 12-6 において、スライスに添え字アクセスして値をクローンすることで、Config 構造体に値を所有させながら、String 値のスライスを取り、Config 構造体のインスタンスを作るコードを追記しました。リスト 13-17 では、リスト 12-23 のような Config::build の実装を再現しました:

ファイル名: src/lib.rs

```
# use std::env;
# use std::error::Error;
# use std::fs;
#
# pub struct Config {
#     pub query: String,
#     pub file_path: String,
#     pub ignore_case: bool,
# }
#
impl Config {
    pub fn build(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let file_path = args[2].clone();

        let ignore_case = env::var("IGNORE_CASE").is_ok();

        Ok(Config {
            query,
            file_path,
            ignore_case,
        })
    }
}
#
# pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
#     let contents = fs::read_to_string(config.file_path)?;
```

```
#
#     let results = if config.ignore_case {
#         search_case_insensitive(&config.query, &contents)
#     } else {
#         search(&config.query, &contents)
#     };
#
#     for line in results {
#         println!("{line}");
#     }
#
#     Ok(())
# }
#
# pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
#     let mut results = Vec::new();
#
#     for line in contents.lines() {
#         if line.contains(query) {
#             results.push(line);
#         }
#     }
#
#     results
# }
#
# pub fn search_case_insensitive<'a>(
#     query: &str,
#     contents: &'a str,
# ) -> Vec<&'a str> {
#     let query = query.to_lowercase();
#     let mut results = Vec::new();
#
#     for line in contents.lines() {
#         if line.to_lowercase().contains(&query) {
#             results.push(line);
#         }
#     }
#
#     results
# }
#
# #[cfg(test)]
# mod tests {
#     use super::*;
#
#     #[test]
#     fn case_sensitive() {
#         let query = "duct";
#         let contents = "\
# Rust:
# safe, fast, productive.
# Pick three.
```

```
# Duct tape.";
#
#         assert_eq!(vec!["safe, fast, productive."], search(query, contents));
#     }
#
#     #[test]
#     fn case_insensitive() {
#         let query = "rUsT";
#         let contents = "\
# Rust:
# safe, fast, productive.
# Pick three.
# Trust me.";
#
#         assert_eq!(
#             vec!["Rust:", "Trust me."],
#             search_case_insensitive(query, contents)
#         );
#     }
# }
```

リスト 13-17: リスト 12-23 から `Config::build` 関数の再現

その際、将来的に除去する予定なので、非効率的な `clone` 呼び出しを憂慮するなと述べました。えっと、その時は今です！

引数 `args` に `String` 要素のスライスがあるためにここで `clone` が必要だったのですが、`build` 関数は `args` を所有していません。`Config` インスタンスの所有権を返すためには、`Config` インスタンスがその値を所有できるように、`Config` の `query` と `file_path` フィールドから値をクローンしなければなりませんでした。

イテレータについての新しい知識があれば、`build` 関数をスライスを借用する代わりに、引数としてイテレータの所有権を奪うように変更することができます。スライスの長さを確認し、特定の場所に添え字アクセスするコードの代わりにイテレータの機能を使います。これにより、イテレータは値にアクセスするので、`Config::build` 関数がすることが明確化します。

ひとたび、`Config::build` がイテレータの所有権を奪い、借用する添え字アクセス処理をやめたら、`clone` を呼び出して新しくメモリ確保するのではなく、イテレータからの `String` 値を `Config` にムーブできます。

13.3.1.1　返却されるイテレータを直接使う

入出力プロジェクトの **src/main.rs** ファイルを開いてください。こんな見た目のはずです:

ファイル名: src/main.rs

```
# use std::env;
# use std::process;
#
```

```
# use minigrep::Config;
#
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::build(&args).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {err}");
        process::exit(1);
    });

    // --snip--
#
#     if let Err(e) = minigrep::run(config) {
#         eprintln!("Application error: {e}");
#         process::exit(1);
#     }
}
```

まずはリスト 12-24 のような `main` 関数の冒頭を、今回はイテレータを使用するリスト 13-18 のコードに変更します。これは、`Config::build` も更新するまでコンパイルできません。

ファイル名: src/main.rs

```
# use std::env;
# use std::process;
#
# use minigrep::Config;
#
fn main() {
    let config = Config::build(env::args()).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {err}");
        process::exit(1);
    });

    // --snip--
#
#     if let Err(e) = minigrep::run(config) {
#         eprintln!("Application error: {e}");
#         process::exit(1);
#     }
}
```

リスト 13-18:　`env::args` の戻り値を `Config::build` に渡す

`env::args` 関数は、イテレータを返します！ イテレータの値をベクタに集結させ、それからスライスを `Config::build` に渡すのではなく、今では `env::args` から返ってくるイテレータの所有権を直接 `Config::build` に渡しています。

次に、`Config::build` の定義を更新する必要があります。入出力プロジェクトの **src/lib.rs** ファイルで、`Config::build` のシグニチャをリスト 13-19 のように変えましょう。関数本体を更新する必要

があるので、それでもコンパイルはできません。

ファイル名: src/lib.rs

```
# use std::env;
# use std::error::Error;
# use std::fs;
#
# pub struct Config {
#     pub query: String,
#     pub file_path: String,
#     pub ignore_case: bool,
# }
#
impl Config {
    pub fn build(
        mut args: impl Iterator<Item = String>,
    ) -> Result<Config, &'static str> {
        // --snip--
#         if args.len() < 3 {
#             return Err("not enough arguments");
#         }
#
#         let query = args[1].clone();
#         let file_path = args[2].clone();
#
#         let ignore_case = env::var("IGNORE_CASE").is_ok();
#
#         Ok(Config {
#             query,
#             file_path,
#             ignore_case,
#         })
#     }
# }
#
# pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
#     let contents = fs::read_to_string(config.file_path)?;
#
#     let results = if config.ignore_case {
#         search_case_insensitive(&config.query, &contents)
#     } else {
#         search(&config.query, &contents)
#     };
#
#     for line in results {
#         println!("{line}");
#     }
#
#     Ok(())
# }
#
# pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
```

```
#     let mut results = Vec::new();
#
#     for line in contents.lines() {
#         if line.contains(query) {
#             results.push(line);
#         }
#     }
#
#     results
# }
#
# pub fn search_case_insensitive<'a>(
#     query: &str,
#     contents: &'a str,
# ) -> Vec<&'a str> {
#     let query = query.to_lowercase();
#     let mut results = Vec::new();
#
#     for line in contents.lines() {
#         if line.to_lowercase().contains(&query) {
#             results.push(line);
#         }
#     }
#
#     results
# }
#
# #[cfg(test)]
# mod tests {
#     use super::*;
#
#     #[test]
#     fn case_sensitive() {
#         let query = "duct";
#         let contents = "\
# Rust:
# safe, fast, productive.
# Pick three.
# Duct tape.";
#
#         assert_eq!(vec!["safe, fast, productive."], search(query, contents));
#     }
#
#     #[test]
#     fn case_insensitive() {
#         let query = "rUsT";
#         let contents = "\
# Rust:
# safe, fast, productive.
# Pick three.
# Trust me.";
#
#         assert_eq!(
```

```
#             vec!["Rust:", "Trust me."],
#             search_case_insensitive(query, contents)
#         );
#     }
# }
```

リスト 13-19: `Config::build` のシグニチャをイテレータを期待するように更新する

`env::args` 関数の標準ライブラリドキュメントは、この関数が返すイテレータの型は `std::env::Args` であること、そしてこの型は `String` 値を返す `Iterator` トレイトを実装していることを示しています。

引数 `args` の型が `&[String]` ではなく、トレイト境界 `impl Iterator<Item = String>` を持つジェネリック型を持つように、`Config::build` 関数のシグニチャを更新しています。第 10 章の「引数としてのトレイト」節で議論した `impl Trait` 構文のここでの使用は、`args` は `Iterator` 型を実装し `String` 要素を返す任意の型でよいことを意味します。

`args` の所有権を奪い、繰り返しを行うことで `args` を可変化する予定なので、`args` 引数の仕様に `mut` キーワードを追記でき、可変にします。

13.3.1.2　添え字の代わりに `Iterator` トレイトのメソッドを使用する

次に、`Config::build` の本体を修正しましょう。`args` は `Iterator` トレイトを実装しているので、それに対して `next` メソッドを呼び出せることがわかります！ リスト 13-20 は、リスト 12-23 のコードを `next` メソッドを使用するように更新したものです:

ファイル名: src/lib.rs

```
# use std::env;
# use std::error::Error;
# use std::fs;
#
# pub struct Config {
#     pub query: String,
#     pub file_path: String,
#     pub ignore_case: bool,
# }
#
impl Config {
    pub fn build(
        mut args: impl Iterator<Item = String>,
    ) -> Result<Config, &'static str> {
        args.next();

        let query = match args.next() {
            Some(arg) => arg,
            // クエリ文字列を取得できませんでした
            None => return Err("Didn't get a query string"),
        };
```

```
        let file_path = match args.next() {
            Some(arg) => arg,
            // ファイル名を取得できませんでした
            None => return Err("Didn't get a file path"),
        };

        let ignore_case = env::var("IGNORE_CASE").is_ok();

        Ok(Config {
            query,
            file_path,
            ignore_case,
        })
    }
}
#
# pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
#     let contents = fs::read_to_string(config.file_path)?;
#
#     let results = if config.ignore_case {
#         search_case_insensitive(&config.query, &contents)
#     } else {
#         search(&config.query, &contents)
#     };
#
#     for line in results {
#         println!("{line}");
#     }
#
#     Ok(())
# }
#
# pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
#     let mut results = Vec::new();
#
#     for line in contents.lines() {
#         if line.contains(query) {
#             results.push(line);
#         }
#     }
#
#     results
# }
#
# pub fn search_case_insensitive<'a>(
#     query: &str,
#     contents: &'a str,
# ) -> Vec<&'a str> {
#     let query = query.to_lowercase();
#     let mut results = Vec::new();
#
#     for line in contents.lines() {
```

```
#         if line.to_lowercase().contains(&query) {
#             results.push(line);
#         }
#     }
#
#     results
# }
#
# #[cfg(test)]
# mod tests {
#     use super::*;
#
#     #[test]
#     fn case_sensitive() {
#         let query = "duct";
#         let contents = "\
# Rust:
# safe, fast, productive.
# Pick three.
# Duct tape.";
#
#         assert_eq!(vec!["safe, fast, productive."], search(query, contents));
#     }
#
#     #[test]
#     fn case_insensitive() {
#         let query = "rUsT";
#         let contents = "\
# Rust:
# safe, fast, productive.
# Pick three.
# Trust me.";
#
#         assert_eq!(
#             vec!["Rust:", "Trust me."],
#             search_case_insensitive(query, contents)
#         );
#     }
# }
```

リスト 13-20: `Config::build` の本体をイテレータメソッドを使うように変更する

`env::args` の戻り値の 1 番目の値は、プログラム名であることを思い出してください。それは無視し、次の値を取得したいので、まず `next` を呼び出し、戻り値に対して何もしません。2 番目に、`next` を呼び出して `Config` の `query` フィールドに置きたい値を得ます。`next` が `Some` を返したら、`match` を使用してその値を抜き出します。`None` を返したら、十分な引数が与えられなかったということなので、`Err` 値で早期リターンします。`file_path` 値に対しても同じことをします。

13.3.2　イテレータアダプタでコードをより明確にする

入出力プロジェクトの search 関数でも、イテレータを活用することができます。その関数はリスト 12-19 に示していますが、以下のリスト 13-28 に再掲します。

ファイル名: src/lib.rs

```
# use std::error::Error;
# use std::fs;
#
# pub struct Config {
#     pub query: String,
#     pub file_path: String,
# }
#
# impl Config {
#     pub fn build(args: &[String]) -> Result<Config, &'static str> {
#         if args.len() < 3 {
#             return Err("not enough arguments");
#         }
#
#         let query = args[1].clone();
#         let file_path = args[2].clone();
#
#         Ok(Config { query, file_path })
#     }
# }
#
# pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
#     let contents = fs::read_to_string(config.file_path)?;
#
#     Ok(())
# }
#
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
    }

    results
}
#
# #[cfg(test)]
# mod tests {
#     use super::*;
#
```

```
#     #[test]
#     fn one_result() {
#         let query = "duct";
#         let contents = "\
# Rust:
# safe, fast, productive.
# Pick three.";
#
#         assert_eq!(vec!["safe, fast, productive."], search(query, contents));
#     }
# }
```

リスト 13-21: リスト 12-19 の search 関数の実装

イテレータアダプタメソッドを使用して、このコードをもっと簡潔に書くことができます。そうすれば、可変な中間の results ベクタをなくすこともできます。関数型プログラミングスタイルは、可変な状態の量を最小化することを好み、コードを明瞭化します。可変な状態を除去すると、検索を同時並行に行うという将来的な改善をするのが、可能になる可能性があります。なぜなら、results ベクタへの同時アクセスを管理する必要がなくなるからです。リスト 13-22 は、この変更を示しています:

ファイル名: src/lib.rs

```
# use std::env;
# use std::error::Error;
# use std::fs;
#
# pub struct Config {
#     pub query: String,
#     pub file_path: String,
#     pub ignore_case: bool,
# }
#
# impl Config {
#     pub fn build(
#         mut args: impl Iterator<Item = String>,
#     ) -> Result<Config, &'static str> {
#         args.next();
#
#         let query = match args.next() {
#             Some(arg) => arg,
#             None => return Err("Didn't get a query string"),
#         };
#
#         let file_path = match args.next() {
#             Some(arg) => arg,
#             None => return Err("Didn't get a file path"),
#         };
#
#         let ignore_case = env::var("IGNORE_CASE").is_ok();
#
```

```
#         Ok(Config {
#             query,
#             file_path,
#             ignore_case,
#         })
#     }
# }
#
# pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
#     let contents = fs::read_to_string(config.file_path)?;
#
#     let results = if config.ignore_case {
#         search_case_insensitive(&config.query, &contents)
#     } else {
#         search(&config.query, &contents)
#     };
#
#     for line in results {
#         println!("{line}");
#     }
#
#     Ok(())
# }
#
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    contents
        .lines()
        .filter(|line| line.contains(query))
        .collect()
}
#
# pub fn search_case_insensitive<'a>(
#     query: &str,
#     contents: &'a str,
# ) -> Vec<&'a str> {
#     let query = query.to_lowercase();
#     let mut results = Vec::new();
#
#     for line in contents.lines() {
#         if line.to_lowercase().contains(&query) {
#             results.push(line);
#         }
#     }
#
#     results
# }
#
# #[cfg(test)]
# mod tests {
#     use super::*;
#
#     #[test]
#     fn case_sensitive() {
```

```
#         let query = "duct";
#         let contents = "\
# Rust:
# safe, fast, productive.
# Pick three.
# Duct tape.";
#
#         assert_eq!(vec!["safe, fast, productive."], search(query, contents));
#     }
#
#     #[test]
#     fn case_insensitive() {
#         let query = "rUsT";
#         let contents = "\
# Rust:
# safe, fast, productive.
# Pick three.
# Trust me.";
#
#         assert_eq!(
#             vec!["Rust:", "Trust me."],
#             search_case_insensitive(query, contents)
#         );
#     }
# }
```

リスト 13-22: search 関数の実装でイテレータアダプタのメソッドを使用する

search 関数の目的は、query を含む contents の行全てを返すことであることを思い出してください。リスト 13-16 の filter 例に酷似して、このコードは filter アダプタを使用して line.contains(query) が true を返す行だけを残すことができます。それから、合致した行を別のベクタに collect で集結させます。ずっと単純です！ ご自由に、同じ変更を行い、search_case_insensitive 関数でもイテレータメソッドを使うようにしてください。

13.3.3　ループかイテレータかの選択

次の論理的な疑問は、自身のコードでどちらのスタイルを選ぶかと理由です: リスト 13-21 の元の実装とリスト 13-22 のイテレータを使用するバージョンです。多くの Rust プログラマは、イテレータスタイルを好みます。とっかかりが少し困難ですが、いろんなイテレータアダプタとそれがすることの感覚を一度掴めれば、イテレータの方が理解しやすいこともあります。いろんなループを少しずつもてあそんだり、新しいベクタを構築する代わりに、コードは、ループの高難度の目的に集中できるのです。これは、ありふれたコードの一部を抽象化するので、イテレータの各要素が通過しなければならないふるい条件など、このコードに独特の概念を理解しやすくなります。

ですが、本当に 2 つの実装は等価なのでしょうか？ 直観的な仮説は、より低レベルのループの方がより高速ということかもしれません。パフォーマンスに触れましょう。

13.4　パフォーマンス比較: ループ VS イテレータ

ループを使うべきかイテレータを使うべきか決定するために、どちらの実装が速いか知る必要があります: 明示的な `for` ループを使うバージョンの `search` 関数と、イテレータを使うバージョンです。

サー・アーサー・コナン・ドイル (Sir Arthur Conan Doyle) の、**シャーロックホームズの冒険** (The Adventures of Sherlock Homes) 全体を `String` に読み込み、そのコンテンツで **the** という単語を検索することでベンチマークを行いました。こちらが、`for` を使用した `search` 関数のバージョンと、イテレータを使用したバージョンに関するベンチマーク結果です。

```
test bench_search_for  ... bench:  19,620,300 ns/iter (+/- 915,700)
test bench_search_iter ... bench:  19,234,900 ns/iter (+/- 657,200)
```

イテレータバージョンの方が些 (いささ) か高速ですね！ ここでは、ベンチマークのコードは説明しません。なぜなら、要点は、2 つのバージョンが等価であることを証明することではなく、これら 2 つの実装がパフォーマンス的にどう比較されるかを大まかに把握することだからです。

より包括的なベンチマークとするためには、いろんなサイズの様々なテキストを `contents` として使用して、異なる単語、異なる長さの単語を `query` として、他のあらゆる種類のバリエーションを確認するべきです。重要なのは: イテレータは、高度な抽象化にも関わらず、低レベルのコードを自身で書いているかのように、ほぼ同じコードにコンパイルされることです。イテレータは、Rust の**ゼロコスト抽象化**の一つであり、これは、抽象化を使うことが追加の実行時オーバーヘッドを生まないことを意味しています。このことは、C++ の元の設計者であり実装者のビャーネ・ストロヴストルップ (Bjarne Stroustrup) が、**ゼロオーバーヘッド**を「C++ の基礎 (2012)」で定義したのと類似しています。

> 一般的に、C++ の実装は、ゼロオーバーヘッド原則を遵守します: 使用しないものには、支払わなくてよい。さらに: 実際に使っているものに対して、コードをそれ以上うまく渡すことはできない。

別の例として、以下のコードは、オーディオデコーダから取ってきました。デコードアルゴリズムは、線形予測数学演算を使用して、以前のサンプルの線形関数に基づいて未来の値を予測します。このコードは、イテレータ連結をしてスコープにある 3 つの変数に計算を行っています: `buffer` というデータのスライス、12 の `coefficients` (係数) の配列、`qlp_shift` でデータをシフトする量です。この例の中で変数を宣言しましたが、値は与えていません; このコードは、文脈の外では大して意味を持ちませんが、それでも Rust が高レベルな考えを低レベルなコードに翻訳する簡潔で現実的な例になっています。

```
let buffer: &mut [i32];
let coefficients: [i64; 12];
let qlp_shift: i16;
```

```
for i in 12..buffer.len() {
    let prediction = coefficients.iter()
                                 .zip(&buffer[i - 12..i])
                                 .map(|(&c, &s)| c * s as i64)
                                 .sum::<i64>() >> qlp_shift;
    let delta = buffer[i];
    buffer[i] = prediction as i32 + delta;
}
```

prediction の値を算出するために、このコードは、coefficients の 12 の値を繰り返し、zip メソッドを使用して、係数値を前の buffer の 12 の値と組にします。それから各組について、その値をかけ合わせ、結果を全て合計し、合計のビットを qlp_shift ビット分だけ右にシフトさせます。

オーディオデコーダのようなアプリケーションの計算は、しばしばパフォーマンスに最も重きを置きます。ここでは、イテレータを作成し、2 つのアダプタを使用し、それから値を消費しています。この Rust コードは、どんな機械語コードにコンパイルされるのでしょうか？ えー、執筆時点では、手作業で書いたものと同じ機械語にコンパイルされます。coefficients の値の繰り返しに対応するループは全く存在しません: コンパイラは、12 回繰り返しがあることを把握しているので、ループを「展開」します。**ループの展開**は、ループ制御コードのオーバーヘッドを除去し、代わりにループの繰り返しごとに同じコードを生成する最適化です。

係数は全てレジスタに保存されます。つまり、値に非常に高速にアクセスします。実行時に配列の境界チェックをすることもありません。コンパイラが適用可能なこれらの最適化全てにより、結果のコードは究極的に効率化されます。このことがわかったので、もうイテレータとクロージャを恐れなしに使用することができますね！ それらのおかげでコードは、高レベルだけれども、そうすることに対して実行時のパフォーマンスを犠牲にしないようになります。

13.5　まとめ

クロージャとイテレータは、関数型言語の考えに着想を得た Rust の機能です。低レベルのパフォーマンスで、高レベルの考えを明確に表現するという Rust の能力に貢献しています。クロージャとイテレータの実装は、実行時のパフォーマンスが影響されないようなものです。これは、ゼロ代償抽象化を提供するのに努力を惜しまない Rust の目標の一部です。

今や入出力プロジェクトの表現力を改善したので、プロジェクトを世界と共有するのに役に立つ cargo の機能にもっと目を向けましょう。

第 14 章

Cargo と Crates.io についてより詳しく

今まで Cargo のビルド、実行、コードのテストを行うという最も基礎的な機能のみを使ってきましたが、他にもできることはたくさんあります。この章では、そのような他のより高度な機能の一部を議論し、以下のことをする方法をお見せしましょう:

- リリースプロファイルでビルドをカスタマイズする
- crates.io でライブラリを公開する
- ワークスペースで巨大なプロジェクトを体系化する
- crates.io からバイナリをインストールする
- 独自のコマンドを使用して Cargo を拡張する

また、Cargo はこの章で講義する機能以上のこともできるので、機能の全解説を見るには、ドキュメンテーションを参照されたし。

14.1 リリースプロファイルでビルドをカスタマイズする

Rust において、**リリースプロファイル**とは、プログラマがコードのコンパイルオプションについてより制御可能にしてくれる、定義済みのカスタマイズ可能なプロファイルです。各プロファイルは、それぞれ独立して設定されます。

Cargo には 2 つの主なプロファイルが存在します: `dev` プロファイルは、`cargo build` コマンドを実行したときに使用され、`release` プロファイルは、`cargo build --release` コマンドを実行したときに使用されます。`dev` プロファイルは、開発中に役に立つデフォルト設定がなされており、`release` プロファイルは、リリース用の設定がなされています。

これらのプロファイル名は、ビルドの出力で馴染みのある可能性があります:

```
$ cargo build
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
```

```
$ cargo build --release
    Finished release [optimized] target(s) in 0.0s
```

dev と release が、コンパイラによって使用される異なるプロファイルです。

Cargo には、プロジェクトの **Cargo.toml** ファイルに [profile.*] セクションが明示的に追加されていない際に適用される、各プロファイル用のデフォルト設定が存在します。カスタマイズしたいプロファイル用の [profile.*] セクションを追加することで、デフォルト設定の一部が上書きされます。例えば、こちらが dev と release プロファイルの opt-level 設定のデフォルト値です:

ファイル名: Cargo.toml

```
[profile.dev]
opt-level = 0

[profile.release]
opt-level = 3
```

opt-level 設定は、0 から 3 の範囲でコンパイラがコードに適用する最適化の度合いを制御します。最適化を多くかけるとコンパイル時間が延びるので、開発中に頻繁にコードをコンパイルするのなら、たとえ出力結果のコードの動作速度が遅くなっても早くコンパイルできるように、より少ない最適化で済ませてほしいですよね。そのため、dev の opt-level のデフォルト設定は 0 になっています。コードのリリース準備ができたら、より長い時間をコンパイルにかけるのが最善の策です。リリースモードでコンパイルするのはたった 1 回ですが、コンパイル結果のプログラムは何度も実行するので、リリースモードでは、長いコンパイル時間と引き換えに、生成したコードが速く動作します。そのため、release の opt-level のデフォルト設定が 3 になっているのです。

デフォルト設定に対して Cargo.toml で異なる値を追加すれば、上書きすることができます。例として、開発用プロファイルで最適化レベル 1 を使用したければ、以下の 2 行をプロジェクトの **Cargo.toml** ファイルに追加できます:

ファイル名: Cargo.toml

```
[profile.dev]
opt-level = 1
```

このコードは、デフォルト設定の 0 を上書きします。こうすると、cargo build を実行したときに、dev プロファイル用のデフォルト設定に加えて、Cargo は opt-level の変更を適用します。opt-level を 1 に設定したので、Cargo はデフォルトよりは最適化を行いますが、リリースビルドほどではありません。

設定の選択肢と各プロファイルのデフォルト設定の一覧は、Cargo のドキュメンテーションを参照されたし。

14.2　Crates.io にクレートを公開する

プロジェクトの依存として crates.io のパッケージを使用しましたが、自分のパッケージを公開することで他の人とコードを共有することもできます。crates.io のクレートレジストリは、自分のパッケージのソースコードを配布するので、主にオープンソースのコードをホストします。

Rust と Cargo は、公開されたパッケージを人々が見つけやすく、使いやすくしてくれる機能を有しています。これらの機能の一部を次に語り、そして、パッケージの公開方法を説明します。

14.2.1　役に立つドキュメンテーションコメントを行う

パッケージを正確にドキュメントすることで、他のユーザがパッケージを使用する方法や、いつ使用すべきかを理解する手助けをすることになるので、ドキュメンテーションを書くことに時間を費やす価値があります。第 3 章で、2 連スラッシュ、`//` で Rust のコードにコメントをつける方法を議論しました。Rust には、ドキュメンテーション用のコメントも用意されていて、便利なことに**ドキュメンテーションコメント**として知られ、HTML ドキュメントを生成します。クレートの**実装**法とは対照的にクレートの**使用**法を知ることに興味のあるプログラマ向けの、公開 API 用のドキュメンテーションコメントの中身をこの HTML は表示します。

ドキュメンテーションコメントは、2 つではなく、3 連スラッシュ、`///` を使用し、テキストを整形する Markdown 記法もサポートしています。ドキュメント対象の要素の直前にドキュメンテーションコメントを配置してください。リスト 14-1 は、`my_crate` という名のクレートの `add_one` 関数用のドキュメンテーションコメントを示しています。

ファイル名: src/lib.rs

```
/// Adds one to the number given.
/// 与えられた数値に1を足す。
///
/// # Examples
///
/// ```
/// let arg = 5;
/// let answer = my_crate::add_one(arg);
///
/// assert_eq!(6, answer);
/// ```
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

リスト 14-1: 関数のドキュメンテーションコメント

ここで、`add_one` 関数がすることの説明を与え、`Examples` というタイトルでセクションを開始し、

add_one 関数の使用法を模擬するコードを提供しています。このドキュメンテーションコメントから cargo doc を実行することで、HTML ドキュメントを生成することができます。このコマンドはコンパイラとともに配布されている rustdoc ツールを実行し、生成された HTML ドキュメントを **target/doc** ディレクトリに配置します。

利便性のために、cargo doc --open を走らせれば、現在のクレートのドキュメント用の HTML(と、自分のクレートが依存している全てのドキュメント) を構築し、その結果を Web ブラウザで開きます。add_one 関数まで下り、図 14-1 に示したように、ドキュメンテーションコメントのテキストがどう描画されるかを確認しましょう:

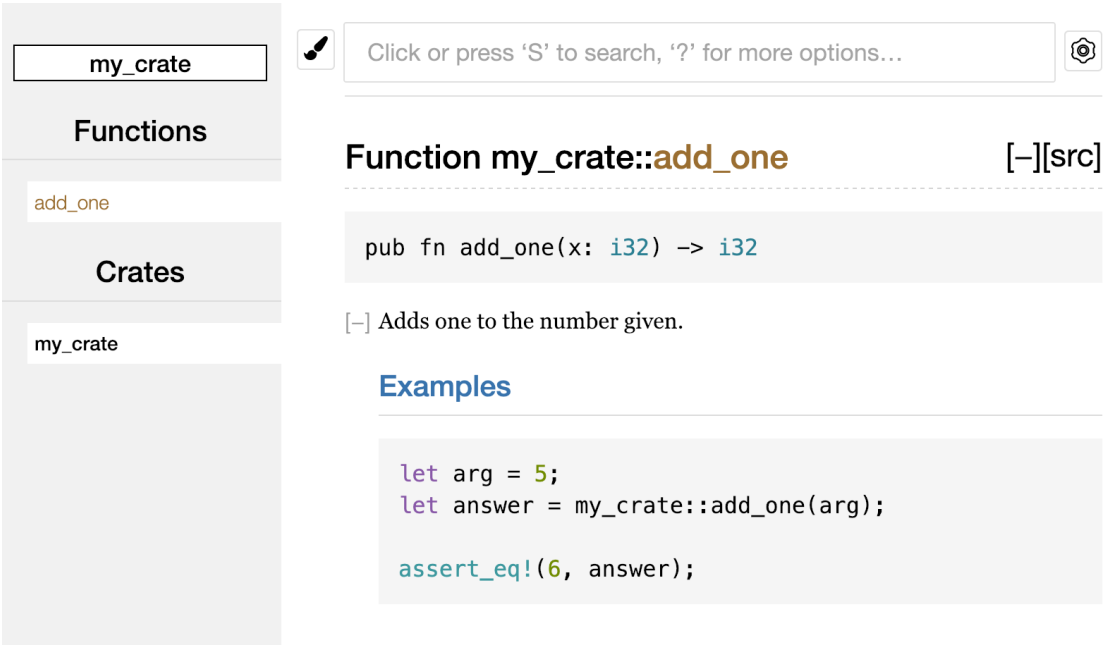


図 14-1: add_one 関数の HTML ドキュメント

14.2.1.1　よく使われるセクション

Examples マークダウンのタイトルをリスト 14-1 で使用し、「例」というタイトルのセクションを HTML に生成しました。こちらがこれ以外にドキュメントでよくクレート筆者が使用するセクションです:

- **Panics**: ドキュメント対象の関数がパニックする可能性のある筋書きです。プログラムをパニックさせたくない関数の使用者は、これらの状況で関数が呼ばれないことを確かめる必要があります。
- **Errors**: 関数が Result を返すなら、起きうるエラーの種類とどんな条件がそれらのエラーを引き起こす可能性があるのか解説すると、呼び出し側の役に立つので、エラーの種類によって

処理するコードを変えて書くことができます。

- **Safety**: 関数が呼び出すのに `unsafe` (unsafe については第 19 章で議論します) なら、関数が unsafe な理由を説明し、関数が呼び出し元に保持していると期待する不変条件を講義するセクションがあるべきです。

多くのドキュメンテーションコメントでは、これら全てのセクションが必要になることはありませんが、これはコードの利用者が知りたいと思うコードの方向性を思い出させてくれるいいチェックリストになります。

14.2.1.2　テストとしてのドキュメンテーションコメント

ドキュメンテーションコメントに例のコードブロックを追加すると、ライブラリの使用方法のデモに役立ち、おまけもついてきます: `cargo test` を走らせると、ドキュメントのコード例をテストとして実行するのです！ 例付きのドキュメントに上回るものはありません。しかし、ドキュメントが書かれてからコードが変更されたがために、動かない例がついているよりも悪いものもありません。リスト 14-1 から `add_one` 関数のドキュメンテーションとともに、`cargo test` を走らせたら、テスト結果に以下のような区域が見られます:

```
   Doc-tests my_crate

running 1 test
test src/lib.rs - add_one (line 5) ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.27s
```

さて、例の `assert_eq!` がパニックするように、関数か例を変更し、再度 `cargo test` を実行したら、doc テストが、例とコードがお互いに同期されていないことを捕捉するところを目撃するでしょう！

14.2.1.3　含まれている要素にコメントする

`//!` スタイルの doc コメントは、コメントに続く要素にではなく、コメントを含む要素にドキュメンテーションを付け加えます。典型的には、クレートのルートファイル (慣例的には、**src/lib.rs**) 内部や、モジュールの内部で使用して、クレートやモジュール全体にドキュメントをつけます。

例えば、`add_one` 関数を含む `my_crate` クレートの目的を解説するドキュメンテーションを追加するには、`//!` で始まるドキュメンテーションコメントを **src/lib.rs** ファイルの先頭につけます。リスト 14-2 に示したようにですね:

ファイル名: src/lib.rs

```
//! # My Crate
//!
//! `my_crate` is a collection of utilities to make performing certain
//! calculations more convenient.
```

```
//!
//! # 自分のクレート
//!
//! `my_crate`は、ユーティリティの集まりであり、特定の計算をより便利に行うことが
    できます。
/// Adds one to the number given.
// --snip--
# ///
# /// # Examples
# ///
# /// ```
# /// let arg = 5;
# /// let answer = my_crate::add_one(arg);
# ///
# /// assert_eq!(6, answer);
# /// ```
# pub fn add_one(x: i32) -> i32 {
#     x + 1
# }
```

リスト 14-2: 全体として `my_crate` クレートにドキュメントをつける

`//!` で始まる最後の行のあとにコードがないことに注目してください。`///` ではなく、`//!` でコメントを開始しているので、このコメントに続く要素ではなく、このコメントを含む要素にドキュメントをつけているわけです。今回の場合、そのような要素は **src/lib.rs** ファイルであり、クレートのルートです。これらのコメントは、クレート全体を解説しています。

`cargo doc --open` を実行すると、これらのコメントは、`my_crate` のドキュメントの最初のページ、クレートの公開要素のリストの上部に表示されます。図 14-2 のようにですね:

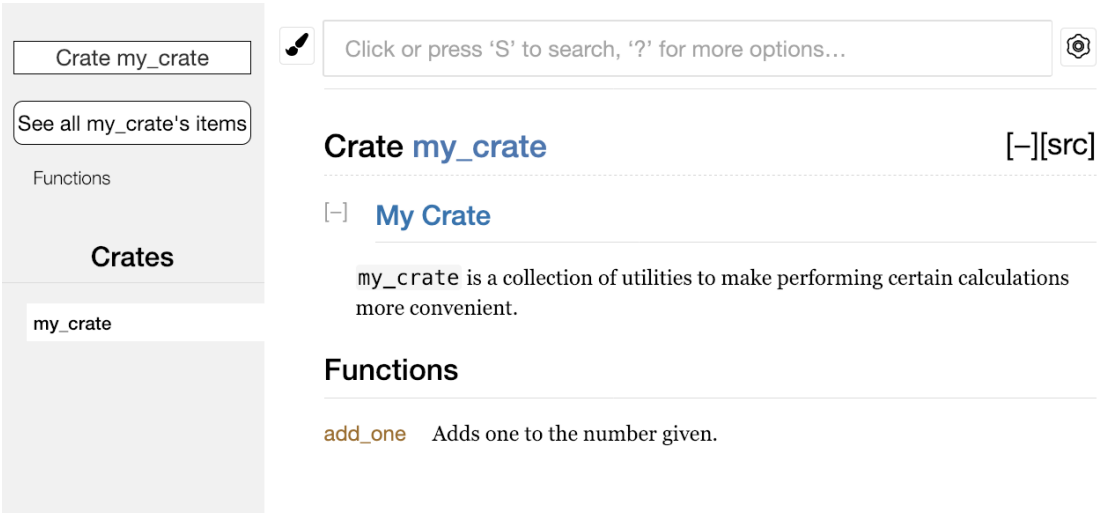


図 14-2: クレート全体を解説するコメントを含む `my_crate` の描画されたドキュメンテーション

要素内のドキュメンテーションコメントは、特にクレートやモジュールを解説するのに有用です。コンテナの全体の目的を説明し、使用者がクレートの体系を理解する手助けをするのに使用してください。

14.2.2　pub use で便利な公開 API をエクスポートする

自分の公開 API の構造は、クレートを公開する際に考慮すべき点です。自分のクレートを使用したい人は、自分よりもその構造に馴染みがないですし、クレートのモジュール階層が大きければ、使用したい部分を見つけるのが困難になる可能性があります。

第 7 章において、pub キーワードで要素を公開にする方法、use キーワードで要素をスコープに導入する方法について講義しました。しかしながら、クレートの開発中に自分にとって意味のある構造は、ユーザにはあまり便利ではない可能性があります。複数階層を含む階層で自分の構造体を体系化したくなるかもしれませんが、それから階層の深いところで定義した型を使用したい人は、型が存在することを見つけ出すのに困難を伴う可能性もあります。また、そのような人は、use my_crate::UsefulType の代わりに use my_crate::some_module::another_module::UsefulType; と入力するのを煩わしく感じる可能性もあります。

嬉しいお知らせは、構造が他人が他のライブラリから使用するのに便利では**ない**場合、内部的な体系を再構築する必要はないということです: 代わりに、要素を再エクスポートし、pub use で自分の非公開構造とは異なる公開構造にできます。再エクスポートは、ある場所の公開要素を一つ取り、別の場所で定義されているかのように別の場所で公開します。

例えば、芸術的な概念をモデル化するために art という名のライブラリを作ったとしましょう。このライブラリ内には、2 つのモジュールがあります: PrimaryColor と SecondaryColor という名前の 2 つの enum を含む、kinds モジュールと mix という関数を含む utils モジュールです。リスト 14-3 のようにですね:

ファイル名: src/lib.rs

```
//! # Art
//!
//! A library for modeling artistic concepts.
//!
//! # 芸術
//!
//! 芸術的な概念をモデル化するライブラリ。

pub mod kinds {
    /// The primary colors according to the RYB color model.
    /// RYBカラーモデルによる主色。
    pub enum PrimaryColor {
        Red,
        Yellow,
        Blue,
    }
```

```
    /// The secondary colors according to the RYB color model.
    /// RYBカラーモデルによる副色。
    pub enum SecondaryColor {
        Orange,
        Green,
        Purple,
    }
}

pub mod utils {
    use crate::kinds::*;

    /// Combines two primary colors in equal amounts to create
    /// a secondary color.
    /// 2つの主色を同じ割合で混合し、副色にする。
    pub fn mix(c1: PrimaryColor, c2: PrimaryColor) -> SecondaryColor {
        // --snip--
#         unimplemented!();
    }
}
```

リスト 14-3: `kinds` と `utils` モジュールに体系化される要素を含む `art` ライブラリ

図 14-3 は、`cargo doc` により生成されるこのクレートのドキュメンテーションの最初のページがどんな見た目になるか示しています:

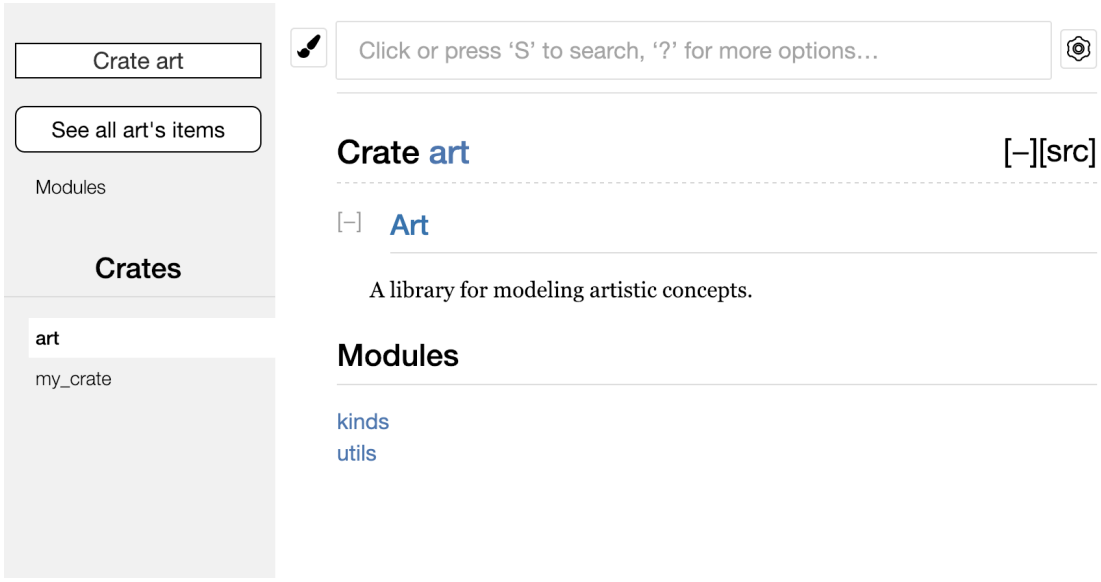


図 14-3: `kinds` と `utils` モジュールを列挙する `art` のドキュメンテーションのトップページ

`PrimaryColor` 型も `SecondaryColor` 型も、`mix` 関数もトップページには列挙されていないことに注

意してください。kinds と utils をクリックしなければ、参照することができません。

このライブラリに依存する別のクレートは、現在定義されているモジュール構造を指定して、art の要素をスコープ内に持ち込む use 文が必要になるでしょう。リスト 14-4 は、art クレートから PrimaryColor と mix 要素を使用するクレートの例を示しています:

ファイル名: src/main.rs

```
use art::kinds::PrimaryColor;
use art::utils::mix;

fn main() {
    let red = PrimaryColor::Red;
    let yellow = PrimaryColor::Yellow;
    mix(red, yellow);
}
```

リスト 14-4: 内部構造がエクスポートされて art クレートの要素を使用するクレート

リスト 14-4 は art クレートを使用していますが、このコードの作者は、PrimaryColor が kinds モジュールにあり、mix が utils モジュールにあることを理解しなければなりませんでした。art クレートのモジュール構造は、art クレートの使用者よりも、art クレートに取り組む開発者などに関係が深いです。この内部構造は art クレートの使用方法を理解しようとする人とって役に立つ情報を何も含んでおらず、むしろ、利用者はどこを見ればよいのか探し出す必要があり、モジュール名を use 文で指定しなければならないので、混乱を招くものです。

公開 API から内部体系を除去するために、リスト 14-3 の art クレートコードを変更し、pub use 文を追加して、最上位で要素を再エクスポートすることができます。リスト 14-5 みたいにですね:

ファイル名: src/lib.rs

```
//! # Art
//!
//! A library for modeling artistic concepts.

pub use self::kinds::PrimaryColor;
pub use self::kinds::SecondaryColor;
pub use self::utils::mix;

pub mod kinds {
    // --snip--
#     /// The primary colors according to the RYB color model.
#     pub enum PrimaryColor {
#         Red,
#         Yellow,
#         Blue,
#     }
#
#     /// The secondary colors according to the RYB color model.
```

```
#     pub enum SecondaryColor {
#         Orange,
#         Green,
#         Purple,
#     }
}

pub mod utils {
    // --snip--
#     use crate::kinds::*;
#
#     /// Combines two primary colors in equal amounts to create
#     /// a secondary color.
#     pub fn mix(c1: PrimaryColor, c2: PrimaryColor) -> SecondaryColor {
#         SecondaryColor::Orange
#     }
}
```

リスト 14-5: `pub use` 文を追加して要素を再エクスポートする

このクレートに対して `cargo doc` が生成する API ドキュメンテーションは、これで図 14-4 のようにトップページに再エクスポートを列挙しリンクするので、`PrimaryColor` 型と `SecondaryColor` 型と `mix` 関数を見つけやすくします。

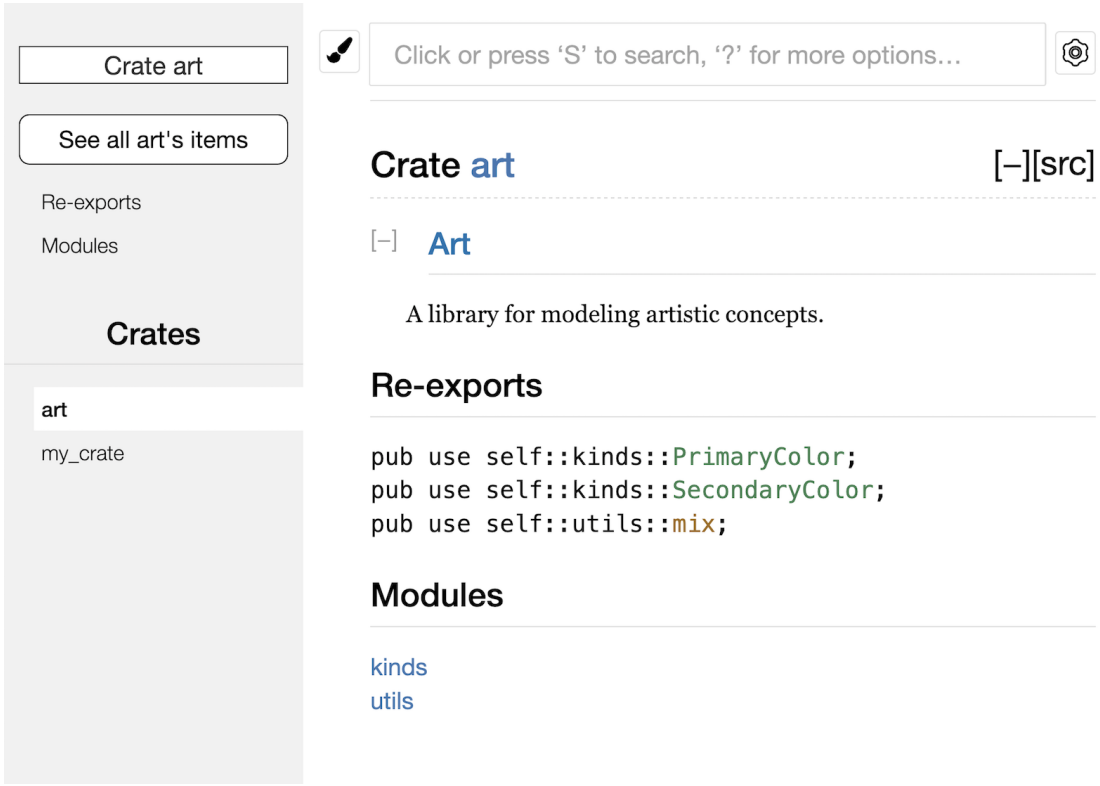


図 14-4: 再エクスポートを列挙する art のドキュメンテーションのトップページ

art クレートのユーザは、それでも、リスト 14-4 にデモされているように、リスト 14-3 の内部構造を見て使用することもできますし、リスト 14-5 のより便利な構造を使用することもできます。リスト 14-6 に示したようにですね:

ファイル名: src/main.rs

```
use art::mix;
use art::PrimaryColor;

fn main() {
    // --snip--
#     let red = PrimaryColor::Red;
#     let yellow = PrimaryColor::Yellow;
#     mix(red, yellow);
}
```

リスト 14-6: art クレートの再エクスポートされた要素を使用するプログラム

ネストされたモジュールがたくさんあるような場合、最上位階層で pub use により型を再エクスポートすることは、クレートの使用者の経験に大きな違いを生みます。別のよくある pub use の使用法は、依存クレートの定義をクレートの公開 API の一部とするために、現在のクレートの中で依存クレートの定義を再エクスポートすることです。

役に立つ API 構造を作ることは、科学というよりも芸術の領域であり、ユーザにとって何が最善の API なのか、探究するために繰り返してみることができます。pub use は、内部的なクレート構造に柔軟性をもたらし、その内部構造をユーザに提示する構造から切り離してくれます。インストールしてある他のクレートを見て、内部構造が公開 API と異なっているか確認してみてください。

14.2.3　Crates.io のアカウントをセットアップする

クレートを公開する前に、crates.io のアカウントを作成し、API トークンを取得する必要があります。そうするには、crates.io のホームページを訪れ、Github アカウントでログインしてください。(現状は、Github アカウントがなければなりませんが、いずれは他の方法でもアカウントを作成できるようになる可能性があります。) ログインしたら、https://crates.io/me/で自分のアカウントの設定に行き、API キーを取り扱ってください。そして、cargo login コマンドを API キーとともに実行してください。以下のようにですね:

```
$ cargo login abcdefghijklmnopqrstuvwxyz012345
```

このコマンドは、Cargo に API トークンを知らせ、**~/.cargo/credentials** にローカルに保存します。このトークンは、**秘密**です: 他人とは共有しないでください。なんらかの理由で他人と実際に共

有してしまったら、古いものを破棄して crates.io で新しいトークンを生成するべきです。

14.2.4　新しいクレートにメタデータを追加する

公開したいクレートがあるとしましょう。公開前に、**Cargo.toml** ファイルの [package] セクションにメタデータを追加する必要があるでしょう。

クレートには、一意な名前が必要でしょう。クレートをローカルで作成している間、クレートの名前はなんでもいい状態でした。ところが、crates.io のクレート名は、最初に来たもの勝ちの精神で付与されていますので、一旦クレート名が取られてしまったら、その名前のクレートを他の人が公開することは絶対できません。クレートを公開しようとする前に使いたい名前を検索してください。すでに使用されている場合は、他の名前を探して、**Cargo.toml** ファイルの [package] セクション下の name フィールドを編集して、公開用に新しい名前を使う必要があるでしょう。以下のように:

ファイル名: Cargo.toml

```
[package]
name = "guessing_game"
```

たとえ、一意な名前を選択していたとしても、この時点で cargo publish を実行すると、警告とエラーが出ます:

```
$ cargo publish
    Updating crates.io index
warning: manifest has no description, license, license-file, documentation,
    homepage or repository.
See https://doc.rust-lang.org/cargo/reference/manifest.html#package-metadata for
     more info.
(警告: マニフェストに説明、ライセンス、ライセンスファイル、ドキュメンテーショ
    ン、ホームページ、リポジトリがありません。
さらなる情報については https://doc.rust-lang.org/cargo/reference/manifest.html#
    package-metadata を参照してください。)
--snip--
error: failed to publish to registry at https://crates.io
(エラー: https://crates.ioにあるレジストリへの公開に失敗しました)

Caused by:
  the remote server responded with an error: missing or empty metadata fields:
      description, license. Please see https://doc.rust-lang.org/cargo/reference
      /manifest.html for how to upload metadata
(原因:
  リモートサーバがエラーを返しました: 存在しないか空のメタデータフィールド:
      description, license. メタデータをアップロードする方法については https://
      doc.rust-lang.org/cargo/reference/manifest.html を参照してください)
```

これがエラーになるのは、大事な情報を一部入れていないからです: 説明とライセンスは、他の人があなたのクレートは何をし、どんな条件の元で使っていいのかを知るために必要なのです。

Cargo.toml に 1 文か 2 文程度の説明をつけてください。これは、検索結果に表示されますからね。`license` フィールドには、**ライセンス識別子**を与える必要があります。Linux 団体の Software Package Data Exchange(SPDX) に、この値に使用できる識別子が列挙されています。例えば、自分のクレートを MIT ライセンスでライセンスするためには、`MIT` 識別子を追加してください:

ファイル名: Cargo.toml

```
[package]
name = "guessing_game"
license = "MIT"
```

SPDX に出現しないライセンスを使用したい場合、そのライセンスをファイルに配置し、プロジェクトにそのファイルを含め、それから `license` キーを使う代わりに、そのファイルの名前を指定するのに `license-file` を使う必要があります。

どのライセンスが自分のプロジェクトに相 (ふ) 応 (さわ) しいかというガイドは、この本の範疇を超えています。Rust コミュニティの多くの人間は、`MIT OR Apache-2.0` のデュアルライセンスを使用することで、Rust 自体と同じようにプロジェクトをライセンスします。この実践は、`OR` で区切られる複数のライセンス識別子を指定して、プロジェクトに複数のライセンスを持たせることもできることを模擬しています。

一意な名前、バージョン、説明、ライセンスが追加され、公開準備のできたプロジェクト用の `Cargo.toml` ファイルは以下のような見た目になっていることでしょう:

ファイル名: Cargo.toml

```
[package]
name = "guessing_game"
version = "0.1.0"
edition = "2021"
description = "A fun game where you guess what number the computer has chosen."
              (コンピュータが選択した数字を言い当てる面白いゲーム)
license = "MIT OR Apache-2.0"

[dependencies]
```

Cargo のドキュメンテーションには、指定して他人が発見し、より容易くクレートを使用できることを保証する他のメタデータが解説されています。

14.2.5　Crates.io に公開する

アカウントを作成し、API トークンを保存し、クレートの名前を決め、必要なメタデータを指定したので、公開する準備が整いました！ クレートを公開すると、特定のバージョンが、crates.io に他の人が使用できるようにアップロードされます。

公開は**永久**なので、気をつけてください。バージョンは絶対に上書きできず、コードも削除できま

せん。crates.io の一つの主な目標が、crates.io のクレートに依存している全てのプロジェクトのビルドが、動き続けるようにコードの永久アーカイブとして機能することなのです。バージョン削除を可能にしてしまうと、その目標を達成するのが不可能になってしまいます。ですが、公開できるクレートバージョンの数に制限はありません。

再度 `cargo publish` コマンドを実行してください。今度は成功するはずです:

```
$ cargo publish
    Updating crates.io index
   Packaging guessing_game v0.1.0 (file:///projects/guessing_game)
   Verifying guessing_game v0.1.0 (file:///projects/guessing_game)
   Compiling guessing_game v0.1.0
(file:///projects/guessing_game/target/package/guessing_game-0.1.0)
    Finished dev [unoptimized + debuginfo] target(s) in 0.19s
   Uploading guessing_game v0.1.0 (file:///projects/guessing_game)
```

おめでとうございます！ Rust コミュニティとコードを共有し、誰でもあなたのクレートを依存として簡単に追加できます。

14.2.6　既存のクレートの新バージョンを公開する

クレートに変更を行い、新バージョンをリリースする準備ができたら、**Cargo.toml** ファイルに指定された `version` の値を変更し、再公開します。セマンティックバージョンルールを使用して加えた変更の種類に基づいて次の適切なバージョン番号を決定してください。そして、`cargo publish` を実行し、新バージョンをアップロードします。

14.2.7　`cargo yank` で Crates.io からバージョンを非推奨化する

以前のバージョンのクレートを削除することはできないものの、将来のプロジェクトがこれに新たに依存することを防ぐことはできます。これは、なんらかの理由により、クレートバージョンが壊れている場合に有用です。そのような場面において、Cargo はクレートバージョンの**取り下げ (yank)** をサポートしています。

バージョンを取り下げると、それに依存するすべての既存のプロジェクトは引き続き依存することができますが、新規プロジェクトがそのバージョンに依存することはできなくなります。つまるところ、取り下げは、すでに **Cargo.lock** が存在するプロジェクトは壊さないが、将来的に生成された **Cargo.lock** ファイルは取り下げられたバージョンを使わない、ということを意味します。

あるバージョンのクレートを取り下げるには、以前公開したクレートのディレクトリで、取り下げたいバージョンを指定して `cargo yank` を実行してください。例えば、`guessing_game` という名前のクレートのバージョン 1.0.1 を公開したことがあり、それを取り下げたい場合は、`guessing_game` のプロジェクトディレクトリで以下を実行します:

```
$ cargo yank --vers 1.0.1
    Updating crates.io index
```

```
        Yank guessing_game@1.0.1
```

--undo をコマンドに付与することで、取り下げを取り消し、再度あるバージョンにプロジェクトを依存させ始めることもできます:

```
$ cargo yank --vers 1.0.1 --undo
    Updating crates.io index
      Unyank guessing_game@1.0.1
```

取り下げは、コードの削除は一切**しません**。例えば、誤ってアップロードされた秘密鍵を削除することはできません。もしそうなってしまったら、即座に秘密鍵をリセットしなければなりません。

14.3　Cargo のワークスペース

第 12 章で、バイナリクレートとライブラリクレートを含むパッケージを構築しました。プロジェクトの開発が進むにつれて、ライブラリクレートの肥大化が続き、その上で複数のライブラリクレートにパッケージを分割したくなることでしょう。Cargo は**ワークスペース**という協調して開発された関連のある複数のパッケージを管理するのに役立つ機能を提供しています。

14.3.1　ワークスペースを生成する

ワークスペースは、同じ **Cargo.lock** と出力ディレクトリを共有する一連のパッケージです。ワークスペースを使用したプロジェクトを作成し、ワークスペースの構造に集中できるよう、瑣末なコードを使用しましょう。ワークスペースを構築する方法は複数あるので、よく使われる方法だけを提示しましょう。バイナリ 1 つとライブラリ 2 つを含むワークスペースを作ります。バイナリは、主要な機能を提供しますが、2 つのライブラリに依存しています。一方のライブラリは、add_one 関数を提供し、2 番目のライブラリは、add_two 関数を提供します。これら 3 つのクレートが同じワークスペースの一部になります。ワークスペース用の新しいディレクトリを作ることから始めましょう:

```
$ mkdir add
$ cd add
```

次に **add** ディレクトリにワークスペース全体を設定する **Cargo.toml** ファイルを作成します。このファイルには [package] セクションはありません。代わりに、バイナリクレートを含むパッケージへのパスを指定することでワークスペースにメンバを追加させてくれる [workspace] セクションから開始します; 今回の場合、そのパスは **adder** です:

ファイル名: Cargo.toml

```
[workspace]

members = [
```

```
    "adder",
]
```

次に、**add** ディレクトリ内で `cargo new` を実行することで `adder` バイナリクレートを作成しましょう:

```
$ cargo new adder
     Created binary (application) `adder` package
```

この時点で、`cargo build` を走らせるとワークスペースを構築できます。**add** ディレクトリに存在するファイルは、以下のようになるはずです:

```
├── Cargo.lock
├── Cargo.toml
├── adder
│   ├── Cargo.toml
│   └── src
│       └── main.rs
└── target
```

ワークスペースは、コンパイルした生成物が置かれる単一の **target** ディレクトリを最上位に持ちます; `adder` パッケージには **target** ディレクトリはありません。**adder** ディレクトリ内部から `cargo build` を走らせることになっていたとしても、コンパイルされる生成物は、**add/adder/target** ではなく、**add/target** に落ち着くでしょう。ワークスペースのクレートは、お互いに依存しあうことを意味するので、Cargo はワークスペースの **target** ディレクトリをこのように構成します。各クレートが **target** ディレクトリを持っていたら、各クレートは自身の **target** ディレクトリにワークスペースの他のクレートの生成物を置くために、ワークスペースの他のクレートを再コンパイルしなくてはならなくなるでしょう。一つの **target** ディレクトリを共有することで、クレートは不必要な再ビルドを回避できるのです。

14.3.2　ワークスペース内に 2 番目のパッケージを作成する

次に、ワークスペースに別のメンバパッケージを作成し、`add_one` と呼びましょう。最上位の **Cargo.toml** を変更して `members` リストで **add_one** パスを指定するようにしてください:

ファイル名: Cargo.toml

```
[workspace]

members = [
    "adder",
    "add_one",
]
```

それから、`add_one` という名前のライブラリクレートを生成してください:

```
$ cargo new add_one --lib
     Created library `add_one` package
```

これで **add** ディレクトリには、以下のディレクトリやファイルが存在するはずです:

```
├── Cargo.lock
├── Cargo.toml
├── add_one
│   ├── Cargo.toml
│   └── src
│       └── lib.rs
├── adder
│   ├── Cargo.toml
│   └── src
│       └── main.rs
└── target
```

add_one/src/lib.rs ファイルに add_one 関数を追加しましょう:

ファイル名: add_one/src/lib.rs

```
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

これで、バイナリを持つパッケージ adder を、ライブラリを持つパッケージ add_one に依存させることができるようになりました。まず、add_one へのパス依存を **adder/Cargo.toml** に追加する必要があります。

ファイル名: adder/Cargo.toml

```
[dependencies]
add_one = { path = "../add_one" }
```

Cargo はワークスペースのクレートが、お互いに依存しているとは想定していないので、依存関係について明示する必要があります。

次に、adder クレート内で (add_one クレートの) add_one 関数を使用しましょう。**adder/src/main.rs** ファイルを開き、冒頭に use 行を追加して新しい add_one ライブラリクレートをスコープに導入してください。それから main 関数を変更し、add_one 関数を呼び出します。リスト 14-7 のようにですね。

ファイル名: adder/src/main.rs

```
use add_one;

fn main() {
```

```
    let num = 10;
    //        "こんにちは世界！ {num}+1は{}!"
    println!("Hello, world! {num} plus one is {}!", add_one::add_one(num));
}
```

リスト 14-7: adder クレートから add_one ライブラリクレートを使用する

最上位の **add** ディレクトリで cargo build を実行することでワークスペースをビルドしましょう！

```
$ cargo build
   Compiling add_one v0.1.0 (file:///projects/add/add_one)
   Compiling adder v0.1.0 (file:///projects/add/adder)
    Finished dev [unoptimized + debuginfo] target(s) in 0.68s
```

add ディレクトリからバイナリクレートを実行するには、-p 引数とパッケージ名を cargo run と共に使用して、実行したいワークスペースのパッケージを指定することができます:

```
$ cargo run -p adder
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/adder`
Hello, world! 10 plus one is 11!
```

これにより、**adder/src/main.rs** のコードが実行され、これは add_one クレートに依存しています。

14.3.2.1　ワークスペースの外部パッケージに依存する

ワークスペースには、各クレートのディレクトリそれぞれに **Cargo.lock** が存在するのではなく、最上位階層にただ一つの **Cargo.lock** が存在するだけのことに注目してください。これにより、全クレートが全依存の同じバージョンを使用していることが確認されます。rand パッケージを **adder/Cargo.toml** と **add_one/Cargo.toml** ファイルに追加すると、Cargo は両者をあるバージョンの rand に解決し、それを一つの **Cargo.lock** に記録します。ワークスペースの全クレートに同じ依存を使用させるということは、クレートが相互に互換性を常に維持するということになります。add_one クレートで rand クレートを使用できるように、**add_one/Cargo.toml** ファイルの [dependencies] セクションに rand クレートを追加しましょう:

ファイル名: add_one/Cargo.toml

```
[dependencies]
rand = "0.8.5"
```

これで、**add_one/src/lib.rs** ファイルに use rand; を追加でき、**add** ディレクトリで cargo build を実行することでワークスペース全体をビルドすると、rand クレートを持ってきてコンパイルするでしょう。スコープ内に持ち込んだ rand を参照していないので、警告が出るでしょう:

```
$ cargo build
    Updating crates.io index
  Downloaded rand v0.8.5
   --snip--
   Compiling rand v0.8.5
   Compiling add_one v0.1.0 (file:///projects/add/add_one)
warning: unused import: `rand`
(警告: 未使用のインポート: `rand`)
 --> add_one/src/lib.rs:1:5
  |
1 | use rand;
  |     ^^^^
  |
  = note: `#[warn(unused_imports)]` on by default

warning: `add_one` (lib) generated 1 warning
   Compiling adder v0.1.0 (file:///projects/add/adder)
    Finished dev [unoptimized + debuginfo] target(s) in 10.18s
```

さて、最上位の **Cargo.lock** は、rand に対する add_one の依存の情報を含むようになりました。ですが、rand はワークスペースのどこかで使用されているにも関わらず、それぞれの **Cargo.toml** ファイルにも、rand を追加しない限り、ワークスペースの他のクレートでそれを使用することはできません。例えば、adder パッケージの **adder/src/main.rs** ファイルに use rand; を追加すると、エラーが出ます:

```
$ cargo build
  --snip--
   Compiling adder v0.1.0 (file:///projects/add/adder)
error[E0432]: unresolved import `rand`
(エラー: 未解決のインポート`rand`)
 --> adder/src/main.rs:2:5
  |
2 | use rand;
  |     ^^^^ no external crate `rand`
  |          (外部クレート`rand`は存在しません)
```

これを修正するには、adder パッケージの **Cargo.toml** ファイルを編集し、これも rand に依存していることを示してください。adder パッケージをビルドすると、rand を **Cargo.lock** の adder の依存一覧に追加しますが、rand のファイルが追加でダウンロードされることはありません。Cargo が、ワークスペース内で rand パッケージを使用するすべてのパッケージ内のすべてのクレートが、同じバージョンを使用することを保証してくれるのです。これによりスペースを節約し、ワークスペースのクレートが相互に互換性があることを保証してくれます。

14.3.2.2　ワークスペースにテストを追加する

さらなる改善として、add_one クレート内に add_one::add_one 関数のテストを追加しましょう:

ファイル名: add_one/src/lib.rs

```
pub fn add_one(x: i32) -> i32 {
    x + 1
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        assert_eq!(3, add_one(2));
    }
}
```

では、最上位の **add** ディレクトリで `cargo test` を実行してください。このような構造をしたワークスペースで `cargo test` を走らせると、ワークスペースの全クレートのテストを実行します:

```
$ cargo test
   Compiling add_one v0.1.0 (file:///projects/add/add_one)
   Compiling adder v0.1.0 (file:///projects/add/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.27s
     Running unittests src/lib.rs (target/debug/deps/add_one-f0253159197f7841)

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

     Running unittests src/main.rs (target/debug/deps/adder-49979ff40686fa8e)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

   Doc-tests add_one

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s
```

出力の最初の区域が、`add_one` クレートの `it_works` テストが通ったことを示しています。次の区域には、`adder` クレートにはテストが見つからなかったことが示され、さらに最後の区域には、`add_one` クレートにドキュメンテーションテストは見つからなかったと表示されています。

`-p` フラグを使用し、テストしたいクレートの名前を指定することで最上位ディレクトリから、ワークスペースのある特定のクレート用のテストを実行することもできます:

```
$ cargo test -p add_one
    Finished test [unoptimized + debuginfo] target(s) in 0.00s
     Running unittests src/lib.rs (target/debug/deps/add_one-b3235fea9a156f74)

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

   Doc-tests add_one

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s
```

この出力は、cargo test が add_one クレートのテストのみを実行し、adder クレートのテストは実行しなかったことを示しています。

ワークスペースのクレートを crates.io に公開したら、ワークスペースのクレートは個別に公開される必要があります。cargo test のように、-p フラグを使用して公開したいクレートの名前を指定することで、ワークスペース内の特定のクレートを公開することができます。

鍛錬を積むために、add_one クレートと同様の方法でワークスペースに add_two クレートを追加してください！

プロジェクトが肥大化してきたら、ワークスペースの使用を考えてみてください: 大きな一つのコードの塊よりも、微細で個別のコンポーネントの方が理解しやすいです。またワークスペースにクレートを保持することは、同時に変更されることが多いのなら、クレート間の協調をしやすくなることにも繋がります。

14.4　cargo install でバイナリをインストールする

cargo install コマンドにより、バイナリクレートをローカルにインストールし、使用することができます。これは、システムパッケージを置き換えることを意図したものではありません。即 (すなわ) ち、Rust の開発者が、他人が crates.io に共有したツールをインストールするのに便利な方法を意味するのです。バイナリターゲットを持つパッケージのみインストールできることに注意してください。**バイナリターゲット**とは、クレートが **src/main.rs** ファイルやバイナリとして指定された他のファイルを持つ場合に生成される実行可能なプログラムのことであり、単独では実行不可能なものの、他のプログラムに含むのには適しているライブラリターゲットとは一線を画します。通常、クレートには、**README** ファイルに、クレートがライブラリかバイナリターゲットか、両方をもつかという情報があります。

cargo install でインストールされるバイナリは全て、インストールのルートの **bin** フォルダに

保持されます。**rustup.rs** を使用し Rust をインストールし、独自の設定を何も行なっていなければ、このディレクトリは、**$HOME/.cargo/bin** になります。`cargo install` でインストールしたプログラムを実行できるようにするためには、そのディレクトリが `$PATH` に含まれていることを確かめてください。

例えば、第 12 章で、ファイルを検索する `ripgrep` という `grep` ツールの Rust 版があることに触れました。`ripgrep` をインストールするには、以下を実行することができます:

```
$ cargo install ripgrep
    Updating crates.io index
  Downloaded ripgrep v13.0.0
  Downloaded 1 crate (243.3 KB) in 0.88s
  Installing ripgrep v13.0.0
--snip--
   Compiling ripgrep v13.0.0
    Finished release [optimized + debuginfo] target(s) in 3m 10s
  Installing ~/.cargo/bin/rg
   Installed package `ripgrep v13.0.0` (executable `rg`)
```

出力の最後から 2 番目の行が、インストールされたバイナリの位置と名前を示していて、`ripgrep` の場合、`rg` です。インストールディレクトリが `$PATH` に存在する限り、前述したように、`rg --help` を走らせて、より高速で Rust らしいファイル検索ツールを使用し始めることができます！

14.5　独自のコマンドで Cargo を拡張する

Cargo は変更する必要なく、新しいサブコマンドで拡張できるように設計されています。`$PATH` にあるバイナリが `cargo-something` という名前なら、`cargo something` を実行することで、Cargo のサブコマンドであるかのように実行することができます。このような独自のコマンドは、`cargo --list` を実行すると、列挙もされます。`cargo install` を使用して拡張をインストールし、それから組み込みの Cargo ツール同様に実行できることは、Cargo の設計上の非常に便利な恩恵です！

14.6　まとめ

Cargo で crates.io とコードを共有することは、Rust のエコシステムを多くの異なる作業に有用にするものの一部です。Rust の標準ライブラリは、小さく安定的ですが、クレートは共有および使用しやすく、言語とは異なるタイムラインで進化します。積極的に crates.io で自分にとって有用なコードを共有してください; 他の誰かにとっても、役に立つものであることでしょう！

第 15 章

スマートポインタ

ポインタは、メモリのアドレスを含む変数の一般的な概念です。このアドレスは、何らかの他のデータを参照、または「指します」。Rust において最もありふれた種類のポインタは参照です。参照については第 4 章で習いましたね。参照は `&` 記号で示唆され、指している値を借用します。データを参照すること以外に特別な能力は何もなく、オーバーヘッドもありません。

一方、**スマートポインタ**は、ポインタのように振る舞いますが、追加のメタデータと能力があるデータ構造です。スマートポインタという概念は、Rust に特有のものではありません。スマートポインタは、C++ に端を発し、他の言語にも存在しています。Rust には、参照以上の機能を提供する色々なスマートポインタが標準ライブラリに定義されています。一般的な概念を探索するために、いくつかのスマートポインタの例を見ていきますが、その一つが**参照カウント**方式のスマートポインタ型です。このポインタのおかげでデータに複数の所有者を持たせることができます。所有者の数を追いかけ、所有者がいなくなったらデータの片付けをしてくれるからです。

所有権と借用の概念を持つ Rust では、参照とスマートポインタにはもう 1 つ違いがあります: 参照はデータを借用するだけである一方で、スマートポインタは、多くの場合、指しているデータを**所有**します。

そのときはそう呼ばなかったものの、私達はすでに、この本の中でいくつかのスマートポインタに遭遇してきました。例えば第 8 章の `String` や `Vec<T>` です。これらの型はどちらも、あるメモリを所有しそれを弄ることができるので、スマートポインタと見なされます。また、メタデータや追加の能力、あるいは保証もあります。例えば `String` は、その許容量をメタデータとして保持し、データが常に有効な UTF-8 であると保証する追加の能力を持ちます。

スマートポインタは普通、構造体を使用して実装されています。通常の構造体とは異なり、スマートポインタが `Deref` と `Drop` トレイトを実装します。`Deref` トレイトにより、スマートポインタ構造体のインスタンスは、参照のように振る舞うことができるので、参照あるいはスマートポインタのどちらとも動作するようにコードを書くことができます。`Drop` トレイトにより、スマートポインタのインスタンスがスコープを外れた時に走るコードをカスタマイズすることができます。この章では、どち

らのトレイトについても議論し、これらのトレイトがスマートポインタにとって重要な理由を説明します。

スマートポインタパターンが Rust においてよく使われる一般的なデザインパターンであることを考えれば、この章で既存のスマートポインタを全て取り扱うことなどできません。多くのライブラリに独自のスマートポインタがあり、自分だけのスマートポインタを書くことさえできるのです。ここでは標準ライブラリの最もありふれたスマートポインタを取り扱っていきます。

- ヒープに値を確保する `Box<T>`
- 複数の所有権を可能にする参照カウント型の `Rc<T>`
- `RefCell<T>` を通してアクセスされ、コンパイル時ではなく実行時に借用規則を強制する型の `Ref<T>` と `RefMut<T>`

さらに、**内部可変性**パターンも扱います。そこでは不変な型が、内部の値を変更するための API を公開するのです。また、**循環参照**についても議論します。つまり、循環参照によっていかにしてメモリがリークするのか、そしてどうやってそれを回避するのかを議論します。

さあ、飛び込みましょう！

15.1　ヒープのデータを指す Box<T>を使用する

最も素直なスマートポインタは**ボックス**であり、その型は `Box<T>` と記述されます。ボックスにより、スタックではなくヒープにデータを格納することができます。スタックに残るのは、ヒープデータへのポインタです。スタックとヒープの違いを再確認するには、第 4 章を参照されたし。

ボックスは、データをスタックの代わりにヒープに格納する以外は、パフォーマンスのオーバーヘッドはありません。しかし、特別な能力がたくさんあるわけでもありません。以下のような場面で最もよく使われるでしょう。

- コンパイル時にはサイズを知ることができない型があり、正確なサイズを要求する文脈でその型の値を使用する時
- 多くのデータがあり、その所有権を移したいが、その際にデータがコピーされないようにしたい時
- 値を所有する必要があり、特定の型であることではなく、特定のトレイトを実装する型であることのみ気にかけている時

「ボックスで再帰的な型を可能にする」節で 1 つ目の場合について実際に説明します。2 番目の場合、多くのデータの所有権を転送するには、データがスタック上でコピーされるので、長い時間がかかり得ます。この場面でパフォーマンスを向上させるために、多くのデータをヒープ上にボックスとして格納することができます。そして、小さなポインタのデータのみがスタック上でコピーされる一方、それが参照しているデータはヒープ上の 1 箇所に留まります。3 番目のケースは**トレイトオブジェクト**として知られています。第 17 章の「トレイトオブジェクトで異なる型の値を許容する」の

節は、すべてその話題に捧げられています。従って、ここで学ぶことは第 17 章でもまた使うことになります！

15.1.1　Box<T>を使ってヒープにデータを格納する

Box<T> のヒープストレージのユースケースを議論する前に、Box<T> の記法と、Box<T> 内に格納された値を読み書きする方法について講義しましょう。

リスト 15-1 は、ボックスを使用してヒープに i32 の値を格納する方法を示しています。

ファイル名: src/main.rs

```
fn main() {
    let b = Box::new(5);
    println!("b = {}", b);
}
```

リスト 15-1: ボックスを使用して i32 の値をヒープに格納する

変数 b を定義して Box の値を保持します。Box は値 5 を指し、値 5 はヒープに確保されています。このプログラムは、b = 5 と出力するでしょう。つまりこの場合、このデータがスタックにあるのと同じような方法でボックスのデータにアクセスできます。所有された値と全く同じでスコープを抜けるとき、実際 b は main の終わりで抜けるのですが、ボックスはメモリから解放されます。メモリの解放は（スタックに格納されている）ボックスと（ヒープに格納されている）指しているデータの両方に対して起きます。

ヒープに単独の値を置いても嬉しいことはほとんどないので、このように単独でボックスを使用することはあまりありません。単独の i32 のような値はデフォルトではスタックに置かれます。ほとんどの場合ではその方が適切です。ボックスのおかげで定義できるようになる型を見てみましょう。ボックスがなければそれらの型は定義できません。

15.1.2　ボックスで再帰的な型を可能にする

再帰的な型の値は、値の一部として同じ型の他の値を持つことができます。再帰的な型はある問題を提起します。コンパイラは、コンパイル時にある型が占有する領域の大きさを知っておかなくてはならないからです。しかし、再帰的な型の値のネストは、理論的には無限に続く可能性があるので、コンパイラはその値が必要とする領域の大きさを知ることができないのです。ボックスのサイズはわかっているので、再帰的な型の定義にボックスを挟むことで、再帰的な型を作ることができます。

再帰的な型の例として、**コンスリスト**を探究しましょう。これは関数型プログラミング言語でよく見られるデータ型です。我々が定義するコンスリストは、再帰を除けば素直です。故に、これから取り掛かる例に現れる概念は、再帰的な型が関わるもっと複雑な場面に遭遇したときには必ず役に立つでしょう。

15.1.2.1　コンスリストについてもっと詳しく

コンスリストは、Lisp プログラミング言語とその方言に由来するデータ構造で、ネストされたペアによって構成される、Lisp 版の連結リストです。その名前は、2 つの引数から新しいペアを構築する Lisp の cons 関数（”construct function” の省略形です）に由来します。cons を、値と別のペアからなるペアに対して呼び出すことで、再帰的なペアからなるコンスリストを構築することができます。

例えば、リスト 1, 2, 3 を含むコンスリストを、丸かっこを各ペアする擬似コードで表現すると、次のようになります:

```
(1, (2, (3, Nil)))
```

コンスリストの各要素は、2 つの要素を含みます。現在の要素の値と次の要素です。リストの最後の要素は、`Nil` と呼ばれる値だけを含み、次の要素を持ちません。コンスリストは、繰り返し cons 関数を呼び出すことで生成されます。繰り返しの基底ケースを示すのに標準的に使われる名前は `Nil` です。これは第 6 章の”null” や”nil” の概念とは異なることに注意してください。”null” や”nil” は、無効だったり存在しない値です。

コンスリストは Rust ではあまり使用されないデータ構造です。Rust で要素のリストがあるときはほとんど、`Vec<T>` を使用するのがよりよい選択になります。より複雑な他の再帰的なデータ型は様々な場面で役に立ち**ます**。しかしこの章ではコンスリストから始めることで、ボックスのおかげで再帰的なデータ型を定義できるわけを、あまり気を散らすことなく調べることができるのです。

リスト 15-2 には、コンスリストの enum 定義が含まれています。このコードはまだコンパイルできないことに注意してください。`List` 型のサイズが分からないからです。これについてはこの後説明します。

ファイル名: src/main.rs

```
enum List {
    Cons(i32, List),
    Nil,
}
#
# fn main() {}
```

リスト 15-2: `i32` 値のコンスリストデータ構造を表す enum を定義する最初の試行

> 注釈: この例のために `i32` 値だけを保持するコンスリストを実装します。第 10 章で議論したように、ジェネリクスを使用してどんな型の値も格納できるコンスリストを定義して実装することもできたでしょう。

この `List` 型を使用してリスト `1`，`2`，`3` を格納すると、リスト 15-3 のコードのような見た目になるでしょう。

ファイル名: src/main.rs

```
# enum List {
#     Cons(i32, List),
#     Nil,
# }
#
use crate::List::{Cons, Nil};

fn main() {
    let list = Cons(1, Cons(2, Cons(3, Nil)));
}
```

リスト 15-3: `List` enum を使用してリスト `1`, `2`, `3` を格納する

最初の `Cons` 値は、`1` と別の `List` 値を保持しています。この `List` 値は別の `Cons` 値で、`2` とまた別の `List` 値を保持しています。この `List` 値はまたまた別の `Cons` 値で、`3` と `List` 値を保持していますが、この `List` 値でついに `Nil` になります。`Nil` はリストの終端を通知する非再帰的な列挙子です。

リスト 15-3 のコードをコンパイルしようとすると、リスト 15-4 に示したエラーが出ます。

```
$ cargo run
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
error[E0072]: recursive type `List` has infinite size
(エラー: 再帰的な型`List`は無限のサイズを持ちます)
 --> src/main.rs:1:1
  |
1 | enum List {
  | ^^^^^^^^^
2 |     Cons(i32, List),
  |               ---- recursive without indirection
  |
help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to break the cycle
(ヘルプ: 何らかの間接参照(例: `Box`、`Rc`、あるいは`&`)を挿入して、循環参照を断
    ち切ってください)
  |
2 |     Cons(i32, Box<List>),
  |               ++++    +

For more information about this error, try `rustc --explain E0072`.
error: could not compile `cons-list` (bin "cons-list") due to 1 previous error
```

リスト 15-4: 再帰的な enum を定義しようとすると得られるエラー

エラーは、この型は「無限のサイズである」と表示しています。理由は、再帰的な列挙子を含む `List` を定義したからです。つまり、`List` は自身の別の値を直接保持しているのです。結果として、コンパイラは `List` 値を格納するのに必要な領域が計算できません。このエラーが出た理由を噛み砕きましょう。まず、非再帰的な型の値を格納するのに必要な領域をどうコンパイラが決定しているかを見ましょう。

15.1.2.2　非再帰的な型のサイズを計算する

第 6 章で enum 定義を議論した時にリスト 6-2 で定義した `Message` enum を思い出してください。

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
#
# fn main() {}
```

`Message` 値一つにメモリを確保するために必要な領域を決定するために、コンパイラは、各列挙子を見てどの列挙子が最も領域を必要とするかを確認します。コンパイラは、`Message::Quit` は全く領域を必要とせず、`Message::Move` は `i32` 値を 2 つ格納するのに十分な領域が必要、などと確かめます。ただ 1 つの列挙子しか使用されないので、`Message` 値一つが必要とする最大の領域は、最大の列挙子を格納するのに必要になる領域です。

これをコンパイラがリスト 15-2 の `List` enum のような再帰的な型が必要とする領域を決定しようとする時に起こることと比較してください。コンパイラは `Cons` 列挙子を見ることから始めます。この列挙子には、型 `i32` 値が一つと型 `List` の値が一つ保持されます。故に、`Cons` は 1 つの `i32` と `List` のサイズに等しい領域を必要とします。`List` が必要とするメモリ量を計算するのに、コンパイラは `Cons` 列挙子から列挙子を観察します。`Cons` 列挙子は型 `i32` を 1 つと型 `List` の値 1 つを保持し、この過程は無限に続きます。図 15-1 のようにですね。

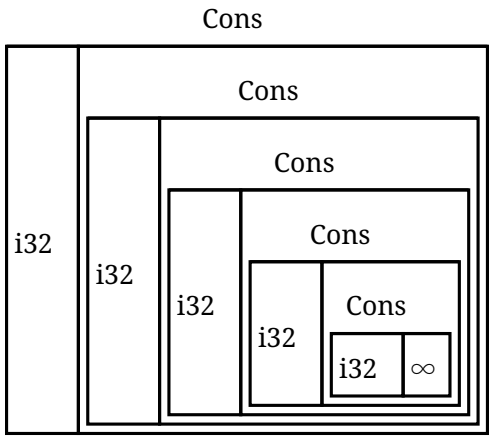


図 15-1: 無限の `Cons` 列挙子からなる無限の `List`

15.1.2.3　Box<T>で既知のサイズの再帰的な型を得る

コンパイラは、再帰的に定義された型に必要なメモリ量を計算できないので、以下の役に立つ提案を含むエラーを返します:

```
help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to break the cycle
(ヘルプ: 何らかの間接参照(例: `Box`、`Rc`、あるいは`&`)を挿入して、循環を断ち
    切ってください)
  |
2 |     Cons(i32, Box<List>),
  |               ++++    +
```

この提案において「間接参照」は、値を直接格納するのではなく、データ構造を変更して値を間接的に格納すべきであるという意味です。これは値の代わりに値へのポインタを格納することによって可能になります。

Box<T> はポインタなので、コンパイラには Box<T> が必要とする領域が必ずわかります。すなわち、ポインタのサイズは指しているデータの量に左右されません。つまり、別の List 値を直接置く代わりに、Cons 列挙子の中に Box<T> を配置することができます。Box<T> は、Cons 列挙子の中ではなく、ヒープに置かれる次の List 値を指します。概念的には、依然として我々のリストは他のリストを保持するリストによって作られたものです。しかし、今やこの実装は、要素をお互いの中に配置するというより、隣り合うように配置するような感じになります。

リスト 15-2 の List enum の定義とリスト 15-3 の List の使用をリスト 15-5 のコードに変更することができ、これはコンパイルが通ります。

ファイル名: src/main.rs

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use crate::List::{Cons, Nil};

fn main() {
    let list = Cons(1, Box::new(Cons(2, Box::new(Cons(3, Box::new(Nil))))));
}
```

リスト 15-5: 既知のサイズにするために Box<T> を使用する List の定義

Cons 列挙子は、1 つの i32 のサイズに加えてボックスのポインタデータを格納する領域を必要とします。Nil 列挙子は値を格納しないので、Cons 列挙子よりも必要な領域は小さいです。これで、どんな List 値も i32 1 つのサイズに加えてボックスのポインタデータのサイズを必要とすることがわかりました。ボックスを使うことで無限に続く再帰の連鎖を断ち切ったので、コンパイラは List 値

を格納するのに必要なサイズを計算できます。図 15-2 は、Cons 列挙子の今の見た目を示しています。

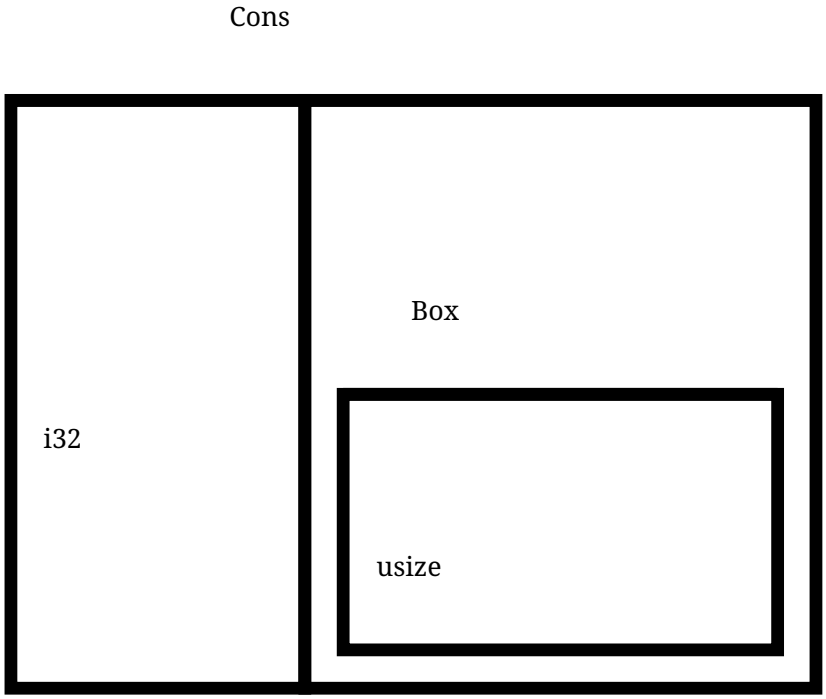


図 15-2: Cons が Box を保持しているので、無限にサイズがあるわけではない List

ボックスは、間接参照とヒープメモリ確保だけを提供します。他のスマートポインタ型に見られるような別の特別な能力は何もありません。これらの特別な能力が招くパフォーマンスのオーバーヘッドもないので、コンスリストのように間接参照だけが必要な機能である場合には便利でしょう。より多くのボックスのユースケースは第 17 章でもお見かけするでしょう。

Box<T> 型がスマートポインタなのは、Deref トレイトを実装しているからです。このトレイトにより Box<T> の値を参照のように扱うことができます。Box<T> 値がスコープを抜けると、ボックスが参照しているヒープデータも片付けられます。これは Drop トレイト実装のおかげです。これら 2 つのトレイトは、他のスマートポインタ型が提供する機能にとってさらに重要なものです。それらついて

はこの章の残りで議論します。これら 2 つのトレイトをより詳しく探究しましょう。

15.2　Deref トレイトでスマートポインタを普通の参照のように扱う

Deref トレイトを実装することで、**参照外し演算子**の * （掛け算やグロブ演算子と混同しないてください）の振る舞いをカスタマイズできます。Deref を実装してスマートポインタを普通の参照みたいに扱えるようにすれば、参照に対して処理を行うコードを書いて、そのコードをスマートポインタに対しても使うことができるのです。

まずは、参照外し演算子が普通の参照に対して動作するところを見ましょう。それから、Box<T> のように振る舞う独自の型を定義してみましょう。参照とは異なり、新しく定義した型には参照外し演算子を使えません。その理由を確認します。Deref トレイトを実装すればスマートポインタは参照と同じように機能するので、そのやり方を調べましょう。そして、Rust には**参照外し型強制**という機能があり、その機能のおかげで参照やスマートポインタをうまく使うことができるので、それに目を向けてみましょう。

> 注釈: 私たちが構築しようとしている MyBox<T> 型と本物の Box<T> との間には、1 つ大きな違いがあります: 私たちのバージョンはヒープ上にデータを保存しません。私たちは Deref の例として注目しているので、データが実際にどこに保存されるかは、ポインタのように振る舞うかどうかに比べて、重要ではありません。

15.2.1　値までポインタを追いかける

普通の参照は 1 種のポインタであり、ポインタはどこか他の場所に格納された値への矢印と見なすことができます。リスト 15-6 では、i32 値への参照を生成してから参照外し演算子を使って値まで参照を辿ります。

ファイル名: src/main.rs

```
fn main() {
    let x = 5;
    let y = &x;

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

リスト 15-6: 参照外し演算子を使用して参照を i32 値まで追いかける

変数 x は i32 値の 5 を保持しています。y は x への参照として設定します。x は 5 に等しいとアサートできます。しかしながら、y の値に関するアサートを行いたい場合、コンパイラが実際の値と比較できるように、*y を使用して参照が指している値まで追いかけなければなりません（そのため**参照外し**

です)。一旦 y の参照を外せば、y が指している整数値にアクセスできます。これは 5 と比較可能です。

代わりに `assert_eq!(5, y);` と書こうとしたら、こんなコンパイルエラーが出るでしょう。

```
$ cargo run
   Compiling deref-example v0.1.0 (file:///projects/deref-example)
error[E0277]: can't compare `{integer}` with `&{integer}`
(エラー: `{integer}`を`&{integer}`と比較することはできません)
 --> src/main.rs:6:5
  |
6 |     assert_eq!(5, y);
  |     ^^^^^^^^^^^^^^^^ no implementation for `{integer} == &{integer}`
  |                      (`{integer} == &{integer}`のための実装はありません)
  |
  = help: the trait `PartialEq<&{integer}>` is not implemented for `{integer}`
   (ヘルプ: トレイト`PartialEq<&{integer}>`は`{integer}`に対して実装されていません)
  = help: the following other types implement trait `PartialEq<Rhs>`:
   (ヘルプ: 以下の型であればトレイト`PartialEq<Rhs>`を実装しています:)
            isize
            i8
            i16
            i32
            i64
            i128
            usize
            u8
          and 6 others
  = note: this error originates in the macro `assert_eq` (in Nightly builds, run
      with -Z macro-backtrace for more info)
    (注釈: このエラーはマクロ`assert_eq`に由来します (nightlyビルドでは、詳細を
        得るには-Z macro-bactraceを実行してください))

For more information about this error, try `rustc --explain E0277`.
error: could not compile `deref-example` (bin "deref-example") due to 1 previous
     error
```

数値と数値への参照の比較は許されていません。これらは異なる型だからです。参照外し演算子を使用して、参照が指している値まで追いかけなければならないのです。

15.2.2　Box<T>を参照のように使う

リスト 15-6 のコードを、参照の代わりに `Box<T>` を使うように書き直すことができます。リスト 15-7 に `Box<T>` に対して使用されている参照外し演算子は、リスト 15-6 で参照に対して使用されている参照外し演算子と同じように機能するでしょう:

ファイル名: src/main.rs

```
fn main() {
    let x = 5;
```

```
    let y = Box::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

リスト 15-7: `Box<i32>` に対して参照外し演算子を使用する

リスト 15-7 とリスト 15-6 の主な違いは、ここでは `y` が、`x` の値を指す参照ではなく、`x` からコピーされた値を指す `Box<T>` のインスタンスとして設定されている点にあります。最後のアサートでは、参照外し演算子を使って `Box<T>` のポインタを辿ることができます。これは `y` が参照だった時と同じやり方です。参照外し演算子が使える以上 `Box<T>` には特別な何かがあるので、次はそれについて調べることにします。そのために、独自の型を定義します。

15.2.3　独自のスマートポインタを定義する

標準ライブラリが提供している `Box<T>` 型に似たスマートポインタを作りましょう。そうすれば、スマートポインタがそのままだと参照と同じ様には振る舞わないことがわかります。それから、どうすれば参照外し演算子を使えるようになるのか見てみましょう。

`Box<T>` 型は突き詰めると（訳註：データがヒープに置かれることを無視すると）1 要素のタプル構造体のような定義になります。なのでリスト 15-8 ではそのように `MyBox<T>` 型を定義しています。また、`Box<T>` に定義された `new` 関数に対応する `new` 関数も定義しています。

ファイル名: src/main.rs

```
struct MyBox<T>(T);

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
    }
}
#
# fn main() {}
```

リスト 15-8: `MyBox<T>` 型を定義する

`MyBox` という構造体を定義し、ジェネリック引数の `T` を宣言しています。この型にどんな型の値も持たせたいからです。`MyBox` 型は型 `T` の要素を 1 つ持つタプル構造体です。`MyBox::new` 関数は型 `T` の引数を 1 つ取り、渡した値を持つ `MyBox` のインスタンスを返します。

試しにリスト 15-7 の `main` 関数をリスト 15-8 に追加し、定義した `MyBox<T>` 型を `Box<T>` の代わりに使うよう変更してみてください。コンパイラは `MyBox` を参照外しする方法がわからないので、リスト 15-9 のコードはコンパイルできません。

ファイル名: src/main.rs

```
# struct MyBox<T>(T);
#
# impl<T> MyBox<T> {
#     fn new(x: T) -> MyBox<T> {
#         MyBox(x)
#     }
# }
#
fn main() {
    let x = 5;
    let y = MyBox::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

リスト 15-9: 参照と `Box<T>` を使ったのと同じように `MyBox<T>` を使おうとする

こちらが結果として出るコンパイルエラーです。

```
$ cargo run
   Compiling deref-example v0.1.0 (file:///projects/deref-example)
error[E0614]: type `MyBox<{integer}>` cannot be dereferenced
(エラー: 型`MyBox<{integer}>`は参照外しできません)
  --> src/main.rs:14:19
   |
14 |     assert_eq!(5, *y);
   |                   ^^

For more information about this error, try `rustc --explain E0614`.
error: could not compile `deref-example` (bin "deref-example") due to 1 previous
     error
```

`MyBox<T>` の参照を外すことはできません。そのための実装を与えていないからです。`*` 演算子で参照外しできるようにするには、`Deref` トレイトを実装します。

15.2.4　`Deref` トレイトを実装して型を参照のように扱う

第 10 章の「トレイトを型に実装する」節で議論したように、トレイトを実装するにはトレイトの必須メソッドに実装を与える必要があります。`Deref` トレイトは標準ライブラリで提供されており、`deref` という 1 つのメソッドの実装を要求します。`deref` は `self` を借用し、内部のデータへの参照を返すメソッドです。リスト 15-10 には、`MyBox` の定義に付け足す `Deref` の実装が含まれています。

ファイル名: src/main.rs

```
use std::ops::Deref;
```

```
impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}
#
# struct MyBox<T>(T);
#
# impl<T> MyBox<T> {
#     fn new(x: T) -> MyBox<T> {
#         MyBox(x)
#     }
# }
#
# fn main() {
#     let x = 5;
#     let y = MyBox::new(x);
#
#     assert_eq!(5, x);
#     assert_eq!(5, *y);
# }
```

リスト 15-10: `MyBox<T>` に `Deref` を実装する

`type Target = T;` という記法は、`Deref` トレイトが使用する関連型を定義しています。関連型はまた少し違ったやり方でジェネリック引数を宣言するためのものですが、今は気にする必要はありません。第 19 章でより詳しく扱います。

`deref` が `*` 演算子でアクセスしたい値への参照を返すように、`deref` メソッドの本体は `&self.0` で埋めました; 第 5 章の「異なる型を生成する名前付きフィールドのないタプル構造体を使用する」節で触れたように、`.0` はタプル構造体の最初の値にアクセスするということを思い出してください。リスト 15-9 の `MyBox<T>` に `*` を呼び出す `main` 関数はこれでコンパイルでき、アサートも通ります！

`Deref` トレイトがないと、コンパイラは `&` 参照しか参照外しできません。`deref` メソッドのおかげで、コンパイラは `Deref` を実装している型の値を取り、`deref` メソッドを呼ぶことで、参照外しが可能な `&` 参照を得られるようになります。

リスト 15-9 に `*y` を入力した時、水面下で Rust は実際にはこのようなコードを走らせていました。

```
*(y.deref())
```

Rust が `*` 演算子を `deref` メソッドの呼び出しと普通の参照外しへと置き換えてくれるので、私達は `deref` メソッドを呼び出す必要があるかどうかを考えなくて済むわけです。この Rust の機能により、普通の参照か `Deref` を実装した型であるかどうかに関わらず、等しく機能するコードを書くことができます。

`deref` メソッドが値への参照を返し、`*(y.deref())` のかっこの外にある普通の参照外しがそれでも必要になるのは、所有権システムに関係します。`deref` メソッドが値への参照ではなく値を直接返したら、値は `self` から外にムーブされてしまいます。今回もそうですが、参照外し演算子を使用するときはほとんどの場合、`MyBox<T>` の中の値の所有権を奪いたくはありません。

`*` 演算子が `deref` メソッドの呼び出しと `*` 演算子の呼び出しに置き換えられるのは、コード内で `*` を使用する毎にただ 1 回だけ、という点に注意して下さい。`*` 演算子の置き換えは無限に繰り返されないので、型 `i32` のデータに行き着きます。これはリスト 15-9 で `assert_eq!` の `5` と合致します。

15.2.5　関数やメソッドで暗黙的な参照外し型強制

参照外し型強制は、`Deref` トレイトを実装する型への参照を、別の型への参照に変換します。例えば、`String` は `&str` を返すように `Deref` トレイトを実装しているので、参照外し型強制で `&String` を `&str` に変換することができます。参照外し型強制は、コンパイラが関数やメソッドの実引数に行う便利なもので、`Deref` トレイトを実装する型に対してのみ機能します。参照外し型強制は、特定の型の値への参照を関数やメソッド定義の引数型と一致しない引数として関数やメソッドに渡すときに自動的に発生します。一連の `deref` メソッドの呼び出しが、提供した型を引数が必要とする型に変換します。

参照外し型強制は、関数やメソッド呼び出しを書くプログラマが `&` や `*` を多くの明示的な参照や参照外しとして追記する必要がないように、Rust に追加されました。また、参照外し型強制のおかげで参照あるいはスマートポインタのどちらかで動くコードをもっと書くことができます。

参照外し型強制が実際に動いていることを確認するため、リスト 15-8 で定義した `MyBox<T>` と、リスト 15-10 で追加した `Deref` の実装を使用しましょう。リスト 15-11 は、文字列スライス引数のある関数の定義を示しています:

ファイル名: src/main.rs

```
fn hello(name: &str) {
    println!("Hello, {name}!");
}
#
# fn main() {}
```

リスト 15-11: 型 `&str` の引数 `name` のある `hello` 関数

`hello` 関数は、文字列スライスを引数として呼び出すことができます。例えば、`hello("Rust")` などです。参照外し型強制により、`hello` を型 `MyBox<String>` の値への参照とともに呼び出すことができます。リスト 15-12 のようにですね:

ファイル名: src/main.rs

```
# use std::ops::Deref;
#
```

```
# impl<T> Deref for MyBox<T> {
#     type Target = T;
#
#     fn deref(&self) -> &T {
#         &self.0
#     }
# }
#
# struct MyBox<T>(T);
#
# impl<T> MyBox<T> {
#     fn new(x: T) -> MyBox<T> {
#         MyBox(x)
#     }
# }
#
# fn hello(name: &str) {
#     println!("Hello, {name}!");
# }
#
fn main() {
    let m = MyBox::new(String::from("Rust"));
    hello(&m);
}
```

リスト 15-12: `hello` を `MyBox<String>` 値とともに呼び出し、参照外し型強制のおかげで動く

ここで、`hello` 関数を引数 `&m` とともに呼び出しています。この引数は、`MyBox<String>` 値への参照です。リスト 15-10 で `MyBox<T>` に `Deref` トレイトを実装したので、コンパイラは `deref` を呼び出すことで、`&MyBox<String>` を `&String` に変換できるのです。標準ライブラリは、`String` に文字列スライスを返す `Deref` の実装を提供していて、この実装は、`Deref` の API ドキュメンテーションに載っています。コンパイラはさらに `deref` を呼び出して、`&String` を `&str` に変換し、これは `hello` 関数の定義と合致します。

Rust に参照外し型強制が実装されていなかったら、リスト 15-12 のコードの代わりにリスト 15-13 のコードを書き、型 `&MyBox<String>` の値で `hello` を呼び出さなければならなかったでしょう。

ファイル名: src/main.rs

```
# use std::ops::Deref;
#
# impl<T> Deref for MyBox<T> {
#     type Target = T;
#
#     fn deref(&self) -> &T {
#         &self.0
#     }
# }
#
```

```
# struct MyBox<T>(T);
#
# impl<T> MyBox<T> {
#     fn new(x: T) -> MyBox<T> {
#         MyBox(x)
#     }
# }
#
# fn hello(name: &str) {
#     println!("Hello, {name}!");
# }
#
fn main() {
    let m = MyBox::new(String::from("Rust"));
    hello(&(*m)[..]);
}
```

リスト 15-13: Rust に参照外し型強制がなかった場合に書かなければならないであろうコード

(*m) が MyBox<String> を String に参照外ししています。そして、& と [..] により、文字列全体と等しい String の文字列スライスを取り、hello のシグニチャと一致するわけです。参照外し型強制のないこのコードは、これらの記号が関係するので、読むのも書くのも理解するのもより難しくなります。参照外し型強制により、コンパイラはこれらの変換を自動的に扱えるのです。

Deref トレイトが関係する型に定義されていると、コンパイラは、型を分析し必要なだけ Deref::deref を使用して、参照を得、引数の型と一致させます。Deref::deref が挿入される必要のある回数は、コンパイル時に解決されるので、参照外し型強制を活用するための実行時の代償は何もありません。

15.2.6　参照外し型強制が可変性と相互作用する方法

Deref トレイトを使用して不変参照に対して * をオーバーライドするように、DerefMut トレイトを使用して可変参照の * 演算子をオーバーライドできます。

以下の 3 つの場合に型やトレイト実装を見つけた時にコンパイラは、参照外し型強制を行います:

- T: Deref<Target=U> の時、&T から &U
- T: DerefMut<Target=U> の時、&mut T から &mut U
- T: Deref<Target=U> の時、&mut T から &U

最初の 2 つは、2 番目は可変性を実装している点を除いて、一緒です。最初のケースは、&T があり、T が何らかの型 U への Deref を実装しているなら、透過的に &U を得られると述べています。2 番目のケースは、同じ参照外し型強制が可変参照についても起こることを述べています。

3 番目のケースはもっと巧妙です: Rust はさらに、可変参照を不変参照にも型強制するのです。ですが、逆はできま**せん:** 不変参照は、絶対に可変参照に型強制されないのです。借用規則により、可

変参照があるなら、その可変参照がそのデータへの唯一の参照に違いありません (でなければ、プログラムはコンパイルできません)。1 つの可変参照を 1 つの不変参照に変換することは、借用規則を絶対に破壊しません。不変参照を可変参照にするには、変換元の不変参照がそのデータへの唯一の不変参照であることが必要ですが、借用規則はそれを保証してくれません。故に、不変参照を可変参照に変換することが可能であるという前提を敷けません。

15.3　Drop トレイトで片付け時にコードを走らせる

スマートポインタパターンにとって重要な 2 番目のトレイトは、Drop であり、これのおかげで値がスコープを抜けそうになった時に起こることをカスタマイズできます。どんな型に対しても Drop トレイトの実装を提供することができ、そのコードはファイルやネットワーク接続などのリソースを解放するために使用できます。

Drop をスマートポインタの文脈で導入しています。Drop トレイトの機能は、ほぼ常にスマートポインタを実装する時に使われるからです。例えば Box<T> はドロップされるときに、そのボックスが指しているヒープの領域を解放するでしょう。

言語によっては、さらに型によっては、プログラマがその型のインスタンスを使い終わる度に、メモリやリソースを解放するコードを呼ばなければならないことがあります。そうした型の例にはファイルハンドル、ソケット、またはロックなどが含まれます。忘れてしまったら、システムは詰め込みすぎになりクラッシュする可能性があります。Rust では、値がスコープを抜ける度に特定のコードが走るよう指定でき、コンパイラはこのコードを自動的に挿入します。結果として、特定の型のインスタンスを使い終わったプログラムの箇所全部にクリーンアップコードを配置するのに配慮する必要はありません。それでもリソースをリークすることはありません。

Drop トレイトを実装することで、値がスコープを抜けた時に走るコードを指定します。Drop トレイトは、self への可変参照を取る drop という 1 つのメソッドを実装する必要があります。いつ Rust が drop を呼ぶのか確認するために、今は println! 文のある drop を実装しましょう。

リスト 15-14 では、コンパイラがいつ drop 関数を走らせるかを示すために、インスタンスがスコープを抜ける時に Dropping CustomSmartPointer! と出力するだけの独自の機能を持つ、CustomSmartPointer 構造体を示します。

ファイル名: src/main.rs

```
struct CustomSmartPointer {
    data: String,
}

impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        //       "データ`{}`を持つCustomSmartPointerをドロップ中!"
        println!("Dropping CustomSmartPointer with data `{}`!", self.data);
    }
}
```

```
fn main() {
    let c = CustomSmartPointer {
        //                  "俺のもの"
        data: String::from("my stuff"),
    };
    let d = CustomSmartPointer {
        //                  "別のもの"
        data: String::from("other stuff"),
    };
    //       "CustomSmartPointerが作成された。"
    println!("CustomSmartPointers created.");
}
```

リスト 15-14: クリーンアップコードを配置する Drop トレイトを実装する CustomSmartPointer 構造体

Drop トレイトは、初期化処理に含まれるので、スコープ内に持ち込む必要はありません。CustomSmartPointer に Drop トレイトを実装し、println! を呼び出す drop メソッドの実装を提供しています。drop 関数の本体は、自分の型のインスタンスがスコープを抜ける時に走らせたいあらゆるロジックを配置する場所です。コンパイラがいつ drop を呼ぶのかを視覚的に示すために、ここでテキストを出力しています。

main で、CustomSmartPointer のインスタンスを 2 つ作り、それから CustomSmartPointers created. と出力しています。main の最後で、CustomSmartPointer のインスタンスはスコープを抜け、コンパイラは最後のメッセージを出力しながら、drop メソッドに置いたコードを呼び出します。drop メソッドを明示的に呼び出す必要はなかったことに注意してください。

このプログラムを実行すると、以下のような出力が出ます:

```
$ cargo run
   Compiling drop-example v0.1.0 (file:///projects/drop-example)
    Finished dev [unoptimized + debuginfo] target(s) in 0.60s
     Running `target/debug/drop-example`
CustomSmartPointers created.
Dropping CustomSmartPointer with data `other stuff`!
Dropping CustomSmartPointer with data `my stuff`!
```

インスタンスがスコープを抜けた時に指定したコードを呼び出しながらコンパイラは、drop を自動的に呼び出してくれました。変数は、生成されたのと逆の順序でドロップされるので、d は c より先にドロップされました。この例の目的は drop メソッドがどう機能するのかを視覚的に案内することですが、通常は、メッセージ出力ではなく、自分の型が走らせる必要のあるクリーンアップコードを指定するでしょう。

15.3.1　std::mem::drop で早期に値をドロップする

残念ながら、自動的な drop 機能を無効化することは、単純ではありません。通常、drop を無効化する必要はありません; Drop トレイトの最重要な要点は、自動的に考慮されることです。ですが、時として、値を早期に片付けたくなる可能性があります。一例は、ロックを管理するスマートポインタを使用する時です: 同じスコープの他のコードがロックを獲得できるように、ロックを解放する drop メソッドを強制したくなる可能性があります。Rust は、Drop トレイトの drop メソッドを手動で呼ばせてくれません; スコープが終わる前に値を強制的にドロップさせたいなら、代わりに標準ライブラリが提供する std::mem::drop 関数を呼ばなければなりません。

リスト 15-14 の main 関数を変更して手動で Drop トレイトの drop メソッドを呼び出そうとしたら、コンパイルエラーになるでしょう。リスト 15-15 のようにですね:

ファイル名: src/main.rs

```
# struct CustomSmartPointer {
#     data: String,
# }
#
# impl Drop for CustomSmartPointer {
#     fn drop(&mut self) {
#         println!("Dropping CustomSmartPointer with data `{}`!", self.data);
#     }
# }
#
fn main() {
    let c = CustomSmartPointer {
        data: String::from("some data"),
    };
    println!("CustomSmartPointer created.");
    c.drop();
    //       "CustomSmartPointerがmainの終端の前でドロップされた。"
    println!("CustomSmartPointer dropped before the end of main.");
}
```

リスト 15-15: Drop トレイトから drop メソッドを手動で呼び出し、早期に片付けようとする

このコードをコンパイルしてみようとすると、こんなエラーが出ます:

```
$ cargo run
   Compiling drop-example v0.1.0 (file:///projects/drop-example)
error[E0040]: explicit use of destructor method
(エラー: デストラクタメソッドを明示的に使用しています)
  --> src/main.rs:16:7
   |
16 |     c.drop();
   |       ^^^^ explicit destructor calls not allowed
```

```
   |           (明示的なデストラクタの呼び出しは許可されません)
   |
help: consider using `drop` function
(ヘルプ: `drop`関数の使用を検討してください)
   |
16 |     drop(c);
   |     +++++ ~

For more information about this error, try `rustc --explain E0040`.
error: could not compile `drop-example` (bin "drop-example") due to 1 previous
    error
```

明示的に drop を呼び出すことは許されていないことをこのエラーメッセージは述べています。エラーメッセージは**デストラクタ**という専門用語を使っていて、これは、インスタンスを片付ける関数の一般的なプログラミング専門用語です。**デストラクタ**は、**コンストラクタ**に類似していて、これはインスタンスを生成します。Rust の drop 関数は、1 種の特定のデストラクタです。

コンパイラはそれでも、main の終端で値に対して自動的に drop を呼び出すので、drop を明示的に呼ばせてくれません。コンパイラが 2 回同じ値を片付けようとするので、これは**二重解放**エラーを発生させるでしょう。

値がスコープを抜けるときに drop が自動的に挿入されるのを無効化できず、drop メソッドを明示的に呼ぶこともできません。よって、値を早期に片付けさせる必要があるなら、std::mem::drop 関数を使用します。

std::mem::drop 関数は、Drop トレイトの drop メソッドとは異なります。強制ドロップさせたい値を引数として渡して呼び出します。この関数は prelude に含まれているので、リスト 15-15 の main を変更して drop 関数を呼び出せます。リスト 15-16 のようにですね:

ファイル名: src/main.rs

```
# struct CustomSmartPointer {
#     data: String,
# }
#
# impl Drop for CustomSmartPointer {
#     fn drop(&mut self) {
#         println!("Dropping CustomSmartPointer with data `{}`!", self.data);
#     }
# }
#
fn main() {
    let c = CustomSmartPointer {
        data: String::from("some data"),
    };
    println!("CustomSmartPointer created.");
    drop(c);
    //       "CustomSmartPointerがmainの終端の前でドロップされた。"
    println!("CustomSmartPointer dropped before the end of main.");
}
```

リスト 15-16: 値がスコープを抜ける前に明示的にドロップするために std::mem::drop を呼び出す

このコードを実行すると、以下のように出力されます:

```
$ cargo run
   Compiling drop-example v0.1.0 (file:///projects/drop-example)
    Finished dev [unoptimized + debuginfo] target(s) in 0.73s
     Running `target/debug/drop-example`
CustomSmartPointer created.
Dropping CustomSmartPointer with data `some data`!
CustomSmartPointer dropped before the end of main.
```

Dropping CustomSmartPointer with data `some data`! というテキストが、CustomSmartPointer created. と CustomSmartPointer dropped before the end of main. テキストの間に出力されるので、drop メソッドのコードがその時点で呼び出されて c をドロップしたことを示しています。

Drop トレイト実装で指定されたコードをいろんな方法で使用し、片付けを便利で安全にすることができます: 例を挙げれば、これを使用して独自のメモリアロケータを作ることもできるでしょう！ Drop トレイトと Rust の所有権システムがあれば、コンパイラが自動的に行うので、片付けを覚えておく必要はなくなります。

まだ使用中の値を間違って片付けてしまうことに起因する問題を心配する必要もなくて済みます: 参照が常に有効であると確認してくれる所有権システムが、値が最早使用されなくなった時に drop が 1 回だけ呼ばれることを保証してくれるのです。

これで Box<T> とスマートポインタの特徴の一部を調査したので、標準ライブラリに定義されている他のスマートポインタをいくつか見ましょう。

15.4　Rc<T>は、参照カウント方式のスマートポインタ

大多数の場合、所有権は明らかです: 一体どの変数が与えられた値を所有しているかわかるのです。ところが、単独の値が複数の所有者を持つ可能性のある場合もあります。例えば、グラフデータ構造では、複数の辺が同じノードを指す可能性があり、概念的にそのノードはそれを指す全ての辺に所有されるわけです。指す辺がなくならない限り、つまり所有者がなくならない限り、ノードは片付けられるべきではありません。

Rc<T> という Rust の型を使用することで、明示的に複数の所有権を可能にしなくてはなりません。これは、**reference counting**(参照カウント) の省略形です。Rc<T> 型は、値がまだ使用中かどうか決定するために、値への参照の数を追跡します。値への参照が 0 なら、どの参照も無効にすることなく、値は片付けられます。

Rc<T> を家族部屋のテレビと想像してください。1 人がテレビを見に部屋に入ったら、テレビをつけます。他の人も部屋に入ってテレビを観ることができます。最後の人が部屋を離れる時、もう使用

されていないので、テレビを消します。他の人がまだ観ているのに誰かがテレビを消したら、残りのテレビ視聴者が騒ぐでしょう！

ヒープにプログラムの複数箇所で読む何らかのデータを確保したいけれど、コンパイル時にはどの部分が最後にデータを使用し終わるか決定できない時に `Rc<T>` 型を使用します。どの部分が最後に終わるかわかっているなら、単にその部分をデータの所有者にして、コンパイル時に強制される普通の所有権ルールが効果を発揮するでしょう。

`Rc<T>` は、シングルスレッドの筋書きで使用するためだけのものであることに注意してください。第 16 章で並行性について議論する時に、マルチスレッドプログラムで参照カウントをする方法を講義します。

15.4.1　Rc<T>でデータを共有する

リスト 15-5 のコンスリストの例に回帰しましょう。`Box<T>` を使って定義したことを思い出してください。今回は、両方とも 3 番目のリストの所有権を共有する 2 つのリストを作成します。これは概念的には図 15-3 のような見た目になります:

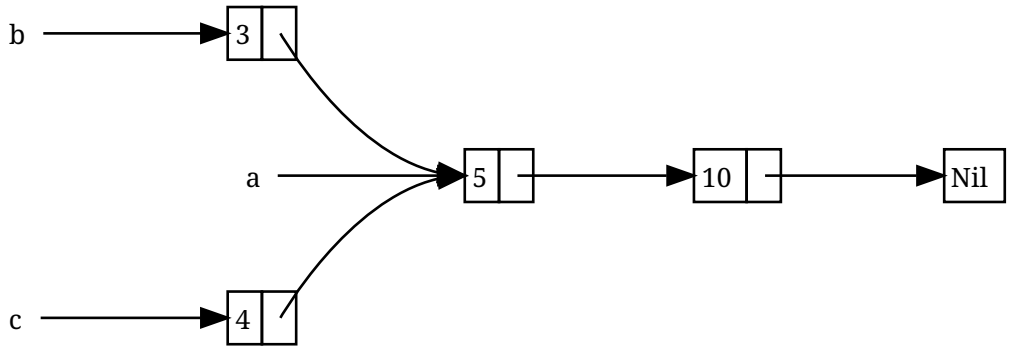


図 15-3: 3 番目のリスト、a の所有権を共有する 2 つのリスト、b と c

5 と 10 を含むリスト `a` を作ります。さらにもう 2 つリストを作ります: 3 で始まる `b` と 4 で始まる `c` です。`b` と `c` のどちらもそれから 5 と 10 を含む最初の `a` リストに続きます。換言すれば、どちらのリストも 5 と 10 を含む最初のリストを共有しています。

`List` の定義を使用して `Box<T>` とともにこの筋書きを実装しようとしても、うまくいきません。リスト 15-17 のようにですね:

ファイル名: src/main.rs

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}
```

```
use crate::List::{Cons, Nil};

fn main() {
    let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
    let b = Cons(3, Box::new(a));
    let c = Cons(4, Box::new(a));
}
```

リスト 15-17: 3 番目のリストの所有権を共有しようとする Box<T> を使った 2 つのリストを存在させることはできないとデモする

このコードをコンパイルすると、こんなエラーが出ます:

```
$ cargo run
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
error[E0382]: use of moved value: `a`
  --> src/main.rs:11:30
   |
9  |     let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
   |         - move occurs because `a` has type `List`, which does not implement
        the `Copy` trait
10 |     let b = Cons(3, Box::new(a));
   |                              - value moved here
11 |     let c = Cons(4, Box::new(a));
   |                              ^ value used here after move

For more information about this error, try `rustc --explain E0382`.
error: could not compile `cons-list` (bin "cons-list") due to 1 previous error
```

Cons 列挙子は、保持しているデータを所有するので、b リストを作成する時に、a が b にムーブされ、b が a を所有します。それから c を作る際に再度 a を使用しようとすると、a はムーブ済みなので、できないわけです。

Cons の定義を代わりに参照を保持するように変更することもできますが、そうしたら、ライフタイム引数を指定しなければなりません。ライフタイム引数を指定することで、リストの各要素が最低でもリスト全体と同じ期間だけ生きることを指定することになります。これはリスト 15-17 での要素とリストに対しては当てはまりますが、すべての用途で当てはまるものではありません。

代わりに、List の定義をリスト 15-18 のように、Box<T> の箇所に Rc<T> を使うように変更します。これで各 Cons 列挙子は、値と List を指す Rc<T> を保持するようになりました。b を作る際、a の所有権を奪うのではなく、a が保持している Rc<List> をクローンします。それによって、参照の数が 1 から 2 に増え、a と b にその Rc<List> にあるデータの所有権を共有させます。また、c を生成する際にも a をクローンするので、参照の数は 2 から 3 になります。Rc::clone を呼ぶ度に、Rc<List> 内のデータの参照カウントが増え、参照が 0 にならない限りデータは片付けられません。

ファイル名: src/main.rs

```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
}
```

リスト 15-18: `Rc<T>` を使用する `List` の定義

初期化処理に含まれていないので、`use` 文を追加して `Rc<T>` をスコープに導入する必要があります。`main` で 5 と 10 を保持するリストを作成し、`a` の新しい `Rc<List>` に格納しています。それから、`b` と `c` を作成する際に、`Rc::clone` 関数を呼び出し、引数として `a` の `Rc<List>` への参照を渡しています。

`Rc::clone(&a)` ではなく、`a.clone()` を呼ぶこともできますが、Rust のしきたりは、この場合 `Rc::clone` を使うことです。`Rc::clone` の実装は、多くの型の `clone` 実装のように、全てのデータのディープコピーをすることではありません。`Rc::clone` の呼び出しは、参照カウントをインクリメントするだけであり、時間はかかりません。データのディープコピーは時間がかかることもあります。参照カウントに `Rc::clone` を使うことで、視覚的にディープコピーをする類のクローンと参照カウントを増やす種類のクローンを区別することができます。コード内でパフォーマンスの問題を探す際、ディープコピーのクローンだけを考慮し、`Rc::clone` の呼び出しを無視できるのです。

15.4.2　Rc<T>をクローンすると、参照カウントが増える

`a` の `Rc<List>` への参照を作ったりドロップする毎に参照カウントが変化するのが確かめられるように、リスト 15-18 の動く例を変更しましょう。

リスト 15-19 で、リスト `c` を囲む内側のスコープができるよう `main` を変更します; そうすれば、`c` がスコープを抜けるときに参照カウントがどう変化するか確認できます。

ファイル名: src/main.rs

```
# enum List {
#     Cons(i32, Rc<List>),
#     Nil,
# }
#
# use crate::List::{Cons, Nil};
# use std::rc::Rc;
#
```

```
fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    //       "a作成後のカウント = {}"
    println!("count after creating a = {}", Rc::strong_count(&a));
    let b = Cons(3, Rc::clone(&a));
    //       "b作成後のカウント = {}"
    println!("count after creating b = {}", Rc::strong_count(&a));
    {
        let c = Cons(4, Rc::clone(&a));
        //       "c作成後のカウント = {}"
        println!("count after creating c = {}", Rc::strong_count(&a));
    }
    //       "cがスコープを抜けた後のカウント = {}"
    println!("count after c goes out of scope = {}", Rc::strong_count(&a));
}
```

リスト 15-19: 参照カウントを出力する

プログラム内で参照カウントが変更される度に、参照カウントを出力します。参照カウントは、`Rc::strong_count` 関数を呼び出すことで得ています。`Rc<T>` 型には `weak_count` もあるので、この関数は `count` ではなく `strong_count` と命名されています; `weak_count` の使用目的は、「循環参照を回避する: `Rc<T>` を `Weak<T>` に変換する」節で確かめます。

このコードは、以下の出力をします:

```
$ cargo run
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
    Finished dev [unoptimized + debuginfo] target(s) in 0.45s
     Running `target/debug/cons-list`
count after creating a = 1
count after creating b = 2
count after creating c = 3
count after c goes out of scope = 2
```

`a` の `Rc<List>` は最初 1 という参照カウントであることがわかります; そして、`clone` を呼び出す度に、カウントは 1 ずつ上がります。`c` がスコープを抜けると、カウントは 1 下がります。参照カウントを増やすのに、`Rc::clone` を呼ばなければいけなかったみたいに参照カウントを減らすのに関数を呼び出す必要はありません: `Rc<T>` 値がスコープを抜けるときに `Drop` トレイトの実装が自動的に参照カウントを減らします。

この例でわからないことは、`b` そして `a` が、`main` の終端でスコープを抜ける時に、カウントが 0 になり、`Rc<List>` が完全に片付けられることです。`Rc<T>` を使用すると、単独の値に複数の所有者を持たせることができ、所有者のいずれかが存在している限り、値が有効であり続けることをカウントは保証します。

不変参照経由で、`Rc<T>` は読み取り専用にプログラムの複数箇所間でデータを共有させてくれます。`Rc<T>` が複数の可変参照を存在させることも許可してくれたら、第 4 章で議論した借用ルールの 1 つを侵害する虞 (おそれ) があります: 同じ場所への複数の可変借用は、データ競合や矛盾を引き起こす

ことがあるのです。しかし、データを可変化する能力はとても有用です！ 次の節では、内部可変性パターンと、`Rc<T>` と絡めて使用してこの不変性制限を手がけられる `RefCell<T>` 型について議論します。

15.5　RefCell<T>と内部可変性パターン

内部可変性は、そのデータへの不変参照がある時でさえもデータを可変化できる Rust でのデザインパターンです: 普通、この行動は借用規則により許可されません。データを可変化するために、このパターンは、データ構造内で `unsafe` コードを使用して、可変性と借用を支配する Rust の通常の規則を捻じ曲げています。unsafe コードは、コンパイラが規則をチェックしてくれることに頼らず、自分でチェックしていることをコンパイラに表明します; unsafe コードについては第 19 章で詳しく議論します。

たとえ、コンパイラが保証できなくても、借用規則に実行時に従うことが保証できる時に限り、内部可変性パターンを使用した型を使用できます。関係する `unsafe` コードはそうしたら、安全な API にラップされ、外側の型は、それでも不変です。

内部可変性パターンに従う `RefCell<T>` 型を眺めてこの概念を探究しましょう。

15.5.1　RefCell<T>で実行時に借用規則を強制する

`Rc<T>` と異なり、`RefCell<T>` 型は、保持するデータに対して単独の所有権を表します。では、どうして `RefCell<T>` が `Box<T>` のような型と異なるのでしょうか？ 第 4 章で学んだ借用規則を思い出してください:

- いかなる時点においても、1 個の可変参照または任意個の不変参照の**どちらか** (両方ではない) が存在できる。
- 参照は常に有効でなければならない。

参照と `Box<T>` では、借用規則の不変条件は、コンパイル時に強制されています。`RefCell<T>` では、これらの不変条件は、**実行時に**強制されます。参照でこれらの規則を破ったら、コンパイルエラーになりました。`RefCell<T>` でこれらの規則を破ったら、プログラムはパニックし、終了します。

コンパイル時に借用規則を精査することの利点は、エラーが開発過程の早い段階で捕捉されることと、あらかじめ全ての分析が終わるので、実行パフォーマンスへの影響がないことです。それらの理由により、多くの場合でコンパイル時に借用規則を精査することが最善の選択肢であり、これが Rust の既定になっているのです。

借用規則を実行時に代わりに精査する利点は、コンパイル時の精査では許容されなかった特定のメモリ安全な筋書きが許容されることです。Rust コンパイラのような静的解析は、本質的に保守的です。コードの特性には、コードを解析するだけでは検知できないものもあります: 最も有名な例は停止性問題であり、この本の範疇を超えていますが、調べると面白い話題です。

不可能な分析もあるので、Rust のコンパイラが、コードが所有権規則に応じていると確証を得られない場合、正しいプログラムを拒否する可能性があります; このように、保守的なのです。コンパイラが不正なプログラムを受け入れたら、ユーザは、コンパイラが行う保証を信じることはできなくなるでしょう。しかしながら、コンパイラが正当なプログラムを拒否するのなら、プログラマは不便に思うでしょうが、悲劇的なことは何も起こり得ません。コードが借用規則に従っているとプログラマは確証を得ているが、コンパイラがそれを理解し保証することができない時に `RefCell<T>` 型は有用です。

`Rc<T>` と類似して、`RefCell<T>` もシングルスレッドの筋書きで使用するためのものであり、試しにマルチスレッドの文脈で使ってみようとすると、コンパイルエラーを出します。`RefCell<T>` の機能をマルチスレッドのプログラムで得る方法については、第 16 章で語ります。

こちらに `Box<T>`，`Rc<T>`，`RefCell<T>` を選択する理由を要約しておきます:

- `Rc<T>` は、同じデータに複数の所有者を持たせてくれる; `Box<T>` と `RefCell<T>` は単独の所有者。
- `Box<T>` では、不変借用も可変借用もコンパイル時に精査できる; `Rc<T>` では不変借用のみがコンパイル時に精査できる; `RefCell<T>` では、不変借用も可変借用も実行時に精査される。
- `RefCell<T>` は実行時に精査される可変借用を許可するので、`RefCell<T>` が不変でも、`RefCell<T>` 内の値を可変化できる。

不変な値の中の値を可変化することは、**内部可変性**パターンです。内部可変性が有用になる場面を見て、それが可能になる方法を調査しましょう。

15.5.2　内部可変性: 不変値への可変借用

借用規則の結果は、不変値がある時、可変で借用することはできないということです。例えば、このコードはコンパイルできません:

```
fn main() {
    let x = 5;
    let y = &mut x;
}
```

このコードをコンパイルしようとしたら、以下のようなエラーが出るでしょう:

```
$ cargo run
   Compiling borrowing v0.1.0 (file:///projects/borrowing)
error[E0596]: cannot borrow `x` as mutable, as it is not declared as mutable
(エラー: `x`は可変として宣言されていないので、可変として借用することはできません
    )
 --> src/main.rs:3:13
  |
3 |     let y = &mut x;
  |             ^^^^^^ cannot borrow as mutable
```

```
  |                    (可変として借用できません)
  |
help: consider changing this to be mutable
(ヘルプ: これを可変に変更することを考慮してください)
  |
2 |     let mut x = 5;
  |         +++

For more information about this error, try `rustc --explain E0596`.
error: could not compile `borrowing` (bin "borrowing") due to 1 previous error
```

ですが、メソッド内で値が自身を可変化するけれども、他のコードにとっては、不変に見えることが有用な場面もあります。その値のメソッドの外のコードは、その値を可変化することはできないでしょう。`RefCell<T>` を使うことは、内部可変性を取得する能力を得る 1 つの方法です。しかし、`RefCell<T>` は借用規則を完全に回避するものではありません: コンパイラの借用チェッカーは、内部可変性を許可し、借用規則は代わりに実行時に精査されます。この規則を侵害したら、コンパイルエラーではなく `panic!` になるでしょう。

`RefCell<T>` を使用して不変値を可変化する実践的な例に取り組み、それが役に立つ理由を確認しましょう。

15.5.2.1　内部可変性のユースケース: モックオブジェクト

プログラマはテスト中に、特定の振る舞いを観測して、それが適切に実装されていることを確認するために、ある型の代わりに他の型を使用することがあるでしょう。このプレースホルダ型は**テストダブル**と呼ばれます。映画撮影において特定のトリッキーなシーンを演じるために、役者の代わりとして参加する人を指す、「スタントダブル」と同じような意味で考えてください。テストダブルはテストの実行時に、他の型の代役を務めます。**モックオブジェクト**は、テスト中に起きることを記録するテストダブルの特定の型なので、正しい動作が起きたことをアサートできます。

Rust には、他の言語でいうオブジェクトは存在せず、また、他の言語のように標準ライブラリにモックオブジェクトの機能が組み込まれてもいません。ですが、同じ目的をモックオブジェクトとして提供する構造体を作成することは確実にできます。

以下が、テストを行う筋書きです: 値を最大値に対して追跡し、現在値がどれくらい最大値に近いかに基づいてメッセージを送信するライブラリを作成します。このライブラリは、ユーザが行うことのできる API コールの数の割り当てを追跡するのに使用することができるでしょう。

作成するライブラリは、値がどれくらい最大に近いかと、いつどんなメッセージになるべきかを追いかける機能を提供するだけです。このライブラリを使用するアプリケーションは、メッセージを送信する機構を提供すると期待されるでしょう: アプリケーションは、アプリケーションにメッセージを置いたり、メールを送ったり、テキストメッセージを送るなどできるでしょう。ライブラリはその詳細を知る必要はありません。必要なのは、提供する `Messenger` と呼ばれるトレイトを実装している何かなのです。リスト 15-20 は、ライブラリのコードを示しています:

ファイル名: src/lib.rs

```
pub trait Messenger {
    fn send(&self, msg: &str);
}

pub struct LimitTracker<'a, T: Messenger> {
    messenger: &'a T,
    value: usize,
    max: usize,
}

impl<'a, T> LimitTracker<'a, T>
where
    T: Messenger,
{
    pub fn new(messenger: &'a T, max: usize) -> LimitTracker<'a, T> {
        LimitTracker {
            messenger,
            value: 0,
            max,
        }
    }

    pub fn set_value(&mut self, value: usize) {
        self.value = value;

        let percentage_of_max = self.value as f64 / self.max as f64;

        if percentage_of_max >= 1.0 {
            //                  "エラー: 割り当てを超えています!"
            self.messenger.send("Error: You are over your quota!");
        } else if percentage_of_max >= 0.9 {
            //         "緊急警告: 割り当ての90%以上を使用してしまいました!"
            self.messenger
                .send("Urgent warning: You've used up over 90% of your quota!");
        } else if percentage_of_max >= 0.75 {
            //         "警告: 割り当ての75%以上を使用してしまいました!"
            self.messenger
                .send("Warning: You've used up over 75% of your quota!");
        }
    }
}
```

リスト 15-20: 値が最大値にどれくらい近いかを追跡し、特定のレベルの時に警告するライブラリ

このコードの重要な部分の 1 つは、`Messenger` トレイトには、`self` への不変参照とメッセージのテキストを取る `send` というメソッドが 1 つあることです。このトレイトが、モックが実際のオブジェクトと同じように使用できるために、モックオブジェクトが実装する必要のあるインターフェイスなのです。もう 1 つの重要な部分は、`LimitTracker` の `set_value` メソッドの振る舞いをテストしたい

ということです。value 引数に渡すものを変えることができますが、set_value はアサートを行えるものは何も返してくれません。LimitTracker を Messenger トレイトを実装する何かと、max の特定の値で生成したら、value に異なる数値を渡した時にメッセンジャーは適切なメッセージを送ると指示されると言えるようになりたいです。

send を呼び出す時にメールやテキストメッセージを送る代わりに送ると指示されたメッセージを追跡するだけのモックオブジェクトが必要です。モックオブジェクトの新規インスタンスを生成し、モックオブジェクトを使用する LimitTracker を生成し、LimitTracker の set_value を呼び出し、それからモックオブジェクトに期待しているメッセージがあることを確認できます。リスト 15-21 は、それだけをするモックオブジェクトを実装しようとするところを示しますが、借用チェッカーが許可してくれません:

ファイル名: src/lib.rs

```
# pub trait Messenger {
#     fn send(&self, msg: &str);
# }
#
# pub struct LimitTracker<'a, T: Messenger> {
#     messenger: &'a T,
#     value: usize,
#     max: usize,
# }
#
# impl<'a, T> LimitTracker<'a, T>
# where
#     T: Messenger,
# {
#     pub fn new(messenger: &'a T, max: usize) -> LimitTracker<'a, T> {
#         LimitTracker {
#             messenger,
#             value: 0,
#             max,
#         }
#     }
#
#     pub fn set_value(&mut self, value: usize) {
#         self.value = value;
#
#         let percentage_of_max = self.value as f64 / self.max as f64;
#
#         if percentage_of_max >= 1.0 {
#             self.messenger.send("Error: You are over your quota!");
#         } else if percentage_of_max >= 0.9 {
#             self.messenger
#                 .send("Urgent warning: You've used up over 90% of your quota!"
    );
#         } else if percentage_of_max >= 0.75 {
#             self.messenger
#                 .send("Warning: You've used up over 75% of your quota!");
```

```
#         }
#     }
# }
#
#[cfg(test)]
mod tests {
    use super::*;

    struct MockMessenger {
        sent_messages: Vec<String>,
    }

    impl MockMessenger {
        fn new() -> MockMessenger {
            MockMessenger {
                sent_messages: vec![],
            }
        }
    }

    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
            self.sent_messages.push(String::from(message));
        }
    }

    #[test]
    fn it_sends_an_over_75_percent_warning_message() {
        let mock_messenger = MockMessenger::new();
        let mut limit_tracker = LimitTracker::new(&mock_messenger, 100);

        limit_tracker.set_value(80);

        assert_eq!(mock_messenger.sent_messages.len(), 1);
    }
}
```

リスト 15-21: 借用チェッカーが許可してくれない MockMessenger を実装しようとする

このテストコードは String の Vec で送信すると指示されたメッセージを追跡する sent_messages フィールドのある MockMessenger 構造体を定義しています。また、空のメッセージリストから始まる新しい MockMessenger 値を作るのを便利にしてくれる関連関数の new も定義しています。それから MockMessenger に Messenger トレイトを実装しているので、LimitTracker に MockMessenger を与えられます。send メソッドの定義で引数として渡されたメッセージを取り、sent_messages の MockMessenger リストに格納しています。

テストでは、max 値の 75% 以上になる何かに value をセットしろと LimitTracker が指示される時に起きることをテストしています。まず、新しい MockMessenger を生成し、空のメッセージリストから始まります。そして、新しい LimitTracker を生成し、新しい MockMessenger の参照と 100 という

max 値を与えます。LimitTracker の set_value メソッドは 80 という値で呼び出し、これは 100 の 75% を上回っています。そして、MockMessenger が追いかけているメッセージのリストが、今は 1 つのメッセージを含んでいるはずとアサートします。

ところが、以下のようにこのテストには 1 つ問題があります:

```
$ cargo test
   Compiling limit-tracker v0.1.0 (file:///projects/limit-tracker)
error[E0596]: cannot borrow `self.sent_messages` as mutable, as it is behind a
    `&` reference
(エラー: `self.sent_messages`は`&`参照の後ろにあるので、可変として借用できません
    )
  --> src/lib.rs:58:13
   |
58 |             self.sent_messages.push(String::from(message));
   |             ^^^^^^^^^^^^^^^^^^ `self` is a `&` reference, so the data it
       refers to cannot be borrowed as mutable
   |                                (`self`は`&`参照なので、それが参照するデータ
       は可変として借用できません)
   |
help: consider changing this to be a mutable reference
(ヘルプ: 可変参照に変更することを検討してください)
   |
2  |     fn send(&mut self, msg: &str);
   |             ~~~~~~~~~

For more information about this error, try `rustc --explain E0596`.
error: could not compile `limit-tracker` (lib test) due to 1 previous error
```

send メソッドは self への不変参照を取るので、MockMessenger を変更してメッセージを追跡できないのです。代わりに &mut self を使用するというエラーテキストからの提言を選ぶこともできないのです。そうしたら、send のシグニチャが、Messenger トレイト定義のシグニチャと一致しなくなるからです (気軽に試してエラーメッセージを確認してください)。

これは、内部可変性が役に立つ場面なのです！ sent_messages を RefCell<T> 内部に格納し、そうしたら send メソッドは、sent_messages を変更して見かけたメッセージを格納できるようになるでしょう。リスト 15-22 は、それがどんな感じかを示しています:

ファイル名: src/lib.rs

```
# pub trait Messenger {
#     fn send(&self, msg: &str);
# }
#
# pub struct LimitTracker<'a, T: Messenger> {
#     messenger: &'a T,
#     value: usize,
#     max: usize,
# }
#
```

```
# impl<'a, T> LimitTracker<'a, T>
# where
#     T: Messenger,
# {
#     pub fn new(messenger: &'a T, max: usize) -> LimitTracker<'a, T> {
#         LimitTracker {
#             messenger,
#             value: 0,
#             max,
#         }
#     }
#
#     pub fn set_value(&mut self, value: usize) {
#         self.value = value;
#
#         let percentage_of_max = self.value as f64 / self.max as f64;
#
#         if percentage_of_max >= 1.0 {
#             self.messenger.send("Error: You are over your quota!");
#         } else if percentage_of_max >= 0.9 {
#             self.messenger
#                 .send("Urgent warning: You've used up over 90% of your quota!"
    );
#         } else if percentage_of_max >= 0.75 {
#             self.messenger
#                 .send("Warning: You've used up over 75% of your quota!");
#         }
#     }
# }
#
#[cfg(test)]
mod tests {
    use super::*;
    use std::cell::RefCell;

    struct MockMessenger {
        sent_messages: RefCell<Vec<String>>,
    }

    impl MockMessenger {
        fn new() -> MockMessenger {
            MockMessenger {
                sent_messages: RefCell::new(vec![]),
            }
        }
    }

    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
            self.sent_messages.borrow_mut().push(String::from(message));
        }
    }
```

```
    #[test]
    fn it_sends_an_over_75_percent_warning_message() {
        // --snip--
#         let mock_messenger = MockMessenger::new();
#         let mut limit_tracker = LimitTracker::new(&mock_messenger, 100);
#
#         limit_tracker.set_value(80);

        assert_eq!(mock_messenger.sent_messages.borrow().len(), 1);
    }
}
```

リスト 15-22: 外側の値は不変と考えられる一方で RefCell<T> で内部の値を可変化する

さて、sent_messages フィールドは、Vec<String> ではなく、型 RefCell<Vec<String>> になりました。new 関数で、空のベクタの周りに RefCell<Vec<String>> を新しく作成しています。

send メソッドの実装については、最初の引数はそれでも self への不変借用で、トレイト定義と合致しています。RefCell<Vec<String>> の borrow_mut を self.sent_messages に呼び出し、RefCell<Vec<String>> の中の値への可変参照を得て、これはベクタになります。それからベクタへの可変参照に push を呼び出して、テスト中に送られるメッセージを追跡しています。

行わなければならない最後の変更は、アサート内部にあります: 内部のベクタにある要素の数を確認するため、RefCell<Vec<String>> に borrow を呼び出し、ベクタへの不変参照を得ています。

RefCell<T> の使用法を見かけたので、動作の仕方を深掘りしましょう！

15.5.2.2　RefCell<T>で実行時に借用を追いかける

不変および可変参照を作成する時、それぞれ & と &mut 記法を使用します。RefCell<T> では、borrow と borrow_mut メソッドを使用し、これらは RefCell<T> に所属する安全な API の一部です。borrow メソッドは、スマートポインタ型の Ref<T> を返し、borrow_mut はスマートポインタ型の RefMut<T> を返します。どちらの型も Deref を実装しているので、普通の参照のように扱うことができます。

RefCell<T> は、現在活動中の Ref<T> と RefMut<T> スマートポインタの数を追いかけます。borrow を呼び出す度に、RefCell<T> は活動中の不変参照の数を増やします。Ref<T> の値がスコープを抜けたら、不変参照の数は 1 下がります。コンパイル時の借用規則と全く同じように、RefCell<T> はいかなる時も、複数の不変借用または 1 つの可変借用を持たせてくれるのです。

これらの規則を侵害しようとすれば、参照のようにコンパイルエラーになるのではなく、RefCell<T> の実装は実行時にパニックするでしょう。リスト 15-23 は、リスト 15-22 の send 実装に対する変更を示しています。同じスコープで 2 つの可変借用が活動するようわざと生成し、RefCell<T> が実行時にこれをすることを阻止してくれるところを説明しています。

ファイル名: src/lib.rs

```
# pub trait Messenger {
#     fn send(&self, msg: &str);
```

```
# }
#
# pub struct LimitTracker<'a, T: Messenger> {
#     messenger: &'a T,
#     value: usize,
#     max: usize,
# }
#
# impl<'a, T> LimitTracker<'a, T>
# where
#     T: Messenger,
# {
#     pub fn new(messenger: &'a T, max: usize) -> LimitTracker<'a, T> {
#         LimitTracker {
#             messenger,
#             value: 0,
#             max,
#         }
#     }
#
#     pub fn set_value(&mut self, value: usize) {
#         self.value = value;
#
#         let percentage_of_max = self.value as f64 / self.max as f64;
#
#         if percentage_of_max >= 1.0 {
#             self.messenger.send("Error: You are over your quota!");
#         } else if percentage_of_max >= 0.9 {
#             self.messenger
#                 .send("Urgent warning: You've used up over 90% of your quota!"
    );
#         } else if percentage_of_max >= 0.75 {
#             self.messenger
#                 .send("Warning: You've used up over 75% of your quota!");
#         }
#     }
# }
#
# #[cfg(test)]
# mod tests {
#     use super::*;
#     use std::cell::RefCell;
#
#     struct MockMessenger {
#         sent_messages: RefCell<Vec<String>>,
#     }
#
#     impl MockMessenger {
#         fn new() -> MockMessenger {
#             MockMessenger {
#                 sent_messages: RefCell::new(vec![]),
#             }
#         }
```

```
#     }
#
    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
            let mut one_borrow = self.sent_messages.borrow_mut();
            let mut two_borrow = self.sent_messages.borrow_mut();

            one_borrow.push(String::from(message));
            two_borrow.push(String::from(message));
        }
    }
#
#     #[test]
#     fn it_sends_an_over_75_percent_warning_message() {
#         let mock_messenger = MockMessenger::new();
#         let mut limit_tracker = LimitTracker::new(&mock_messenger, 100);
#
#         limit_tracker.set_value(80);
#
#         assert_eq!(mock_messenger.sent_messages.borrow().len(), 1);
#     }
# }
```

リスト 15-23: 同じスコープで 2 つの可変参照を生成して `RefCell<T>` がパニックすることを確かめる

`borrow_mut` から返ってきた `RefMut<T>` スマートポインタに対して変数 `one_borrow` を生成しています。そして、同様にして変数 `two_borrow` にも別の可変借用を生成しています。これにより同じスコープで 2 つの可変参照ができ、これは許可されないことです。このテストを自分のライブラリ用に走らせると、リスト 15-23 のコードはエラーなくコンパイルできますが、テストは失敗するでしょう:

```
$ cargo test
   Compiling limit-tracker v0.1.0 (file:///projects/limit-tracker)
    Finished test [unoptimized + debuginfo] target(s) in 0.91s
     Running unittests src/lib.rs (target/debug/deps/limit_tracker-
         e599811fa246dbde)

running 1 test
test tests::it_sends_an_over_75_percent_warning_message ... FAILED

failures:

---- tests::it_sends_an_over_75_percent_warning_message stdout ----
thread 'tests::it_sends_an_over_75_percent_warning_message' panicked at src/lib.
    rs:60:53:
already borrowed: BorrowMutError
(スレッド'tests::it_sends_an_over_75_percent_warning_message'は、src/lib.rs
    :60:53でパニックしました:
すでに借用されています: BorrowMutError)
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

```
failures:
    tests::it_sends_an_over_75_percent_warning_message

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.00s

error: test failed, to rerun pass `--lib`
```

コードは、already borrowed: BorrowMutError というメッセージとともにパニックしたことに注目してください。このようにして RefCell<T> は実行時に借用規則の侵害を扱うのです。

ここで行ったように、コンパイル時ではなく実行時に借用エラーをキャッチすることを選択するということは、開発過程のより遅い段階でコードのミスを発見することになる可能性があることを意味します: コードをプロダクションにデプロイする時までに発見できないかもしれません。また、コンパイル時ではなく、実行時に借用を追いかける結果として、少し実行時にパフォーマンスを犠牲にするでしょう。しかしながら、RefCell<T> を使うことで、不変値のみが許可される文脈で使用しつつ、自身を変更して見かけたメッセージを追跡するモックオブジェクトを書くことが可能になります。代償はありますが、RefCell<T> を使用すれば、普通の参照よりも多くの機能を得ることができるわけです。

15.5.3　Rc<T>と RefCell<T>を組み合わせることで可変なデータに複数の所有者を持たせる

RefCell<T> の一般的な使用法は、Rc<T> と組み合わせることにあります。Rc<T> は何らかのデータに複数の所有者を持たせてくれるけれども、そのデータに不変のアクセスしかさせてくれないことを思い出してください。RefCell<T> を抱える Rc<T> があれば、複数の所有者を持ち**そして**、可変化できる値を得ることができるのです。

例を挙げれば、Rc<T> を使用して複数のリストに別のリストの所有権を共有させたリスト 15-18 のコンスリストの例を思い出してください。Rc<T> は不変値だけを抱えるので、一旦生成したら、リストの値はどれも変更できません。RefCell<T> を含めて、リストの値を変更する能力を得ましょう。RefCell<T> を Cons 定義で使用することで、リスト全てに格納されている値を変更できることをリスト 15-24 は示しています:

ファイル名: src/main.rs

```
#[derive(Debug)]
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
```

```
use std::cell::RefCell;
use std::rc::Rc;

fn main() {
    let value = Rc::new(RefCell::new(5));

    let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));

    let b = Cons(Rc::new(RefCell::new(3)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(4)), Rc::clone(&a));

    *value.borrow_mut() += 10;

    println!("a after = {:?}", a);
    println!("b after = {:?}", b);
    println!("c after = {:?}", c);
}
```

リスト 15-24: `Rc<RefCell<i32>>` で可変化できる `List` を生成する

`Rc<RefCell<i32>>` のインスタンスの値を生成し、`value` という名前の変数に格納しているので、直接後ほどアクセスすることができます。そして、`a` に `value` を持つ `Cons` 列挙子で `List` を生成しています。`value` から `a` に所有権を移したり、`a` が `value` から借用するのではなく、`a` と `value` どちらにも中の `5` の値の所有権を持たせるよう、`value` をクローンする必要があります。

リスト `a` を `Rc<T>` に包んでいるので、リスト `b` と `c` を生成する時に、どちらも `a` を参照できます。リスト 15-18 ではそうしていました。

`a`、`b`、`c` のリストを作成した後、`value` の値に 10 を足したいとします。これを `value` の `borrow_mut` を呼び出すことで行い、これは、第 5 章で議論した自動参照外し機能 (「`->` 演算子はどこに行ったの？」節をご覧ください) を使用して、`Rc<T>` を内部の `RefCell<T>` 値に参照外ししています。`borrow_mut` メソッドは、`RefMut<T>` スマートポインタを返し、それに対して参照外し演算子を使用し、中の値を変更します。

`a`、`b`、`c` を出力すると、全て 5 ではなく、変更された 15 という値になっていることがわかります。

```
$ cargo run
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
    Finished dev [unoptimized + debuginfo] target(s) in 0.63s
     Running `target/debug/cons-list`
a after = Cons(RefCell { value: 15 }, Nil)
b after = Cons(RefCell { value: 3 }, Cons(RefCell { value: 15 }, Nil))
c after = Cons(RefCell { value: 4 }, Cons(RefCell { value: 15 }, Nil))
```

このテクニックは非常に綺麗です！ `RefCell<T>` を使用することで表面上は不変な `List` 値を持てます。しかし、内部可変性へのアクセスを提供する `RefCell<T>` のメソッドを使用できるので、必要な時にはデータを変更できます。借用規則を実行時に精査することでデータ競合を防ぎ、時としてデータ構造でちょっとのスピードを犠牲にこの柔軟性を得るのは価値があります。`RefCell<T>` は、マルチ

スレッドで実行されるコードに対しては機能しないことに注意してください！ `RefCell<T>` のスレッドセーフなバージョンは `Mutex<T>` で、`Mutex<T>` については第 16 章で議論しましょう。

15.6　循環参照は、メモリをリークすることもある

Rust のメモリ安全保証により誤って絶対に片付けられることのないメモリ (**メモリリーク**として知られています) を生成してしまいにくくなりますが、不可能にはなりません。メモリリークを完全に回避することは、Rust の保証の一つではなく、メモリリークは Rust においてはメモリ安全であることを意味します。Rust では、`Rc<T>` と `RefCell<T>` を使用してメモリリークを許可するとわかります: 要素がお互いに循環して参照する参照を生成することも可能ということです。循環の各要素の参照カウントが絶対に 0 にならないので、これはメモリリークを起こし、値は絶対にドロップされません。

15.6.1　循環参照させる

リスト 15-25 の `List` enum の定義と `tail` メソッドから始めて、どう循環参照が起こる可能性があるのかとその回避策を見ましょう:

ファイル名: src/main.rs

```
use crate::List::{Cons, Nil};
use std::cell::RefCell;
use std::rc::Rc;

#[derive(Debug)]
enum List {
    Cons(i32, RefCell<Rc<List>>),
    Nil,
}

impl List {
    fn tail(&self) -> Option<&RefCell<Rc<List>>> {
        match self {
            Cons(_, item) => Some(item),
            Nil => None,
        }
    }
}

fn main() {}
```

リスト 15-25: `Cons` 列挙子が参照しているものを変更できるように `RefCell<T>` を抱えているコンスリストの定義

リスト 15-5 の `List` 定義の別バリエーションを使用しています。`Cons` 列挙子の 2 番目の要素はこれで `RefCell<Rc<List>>` になり、リスト 15-24 のように `i32` 値を変更する能力があるのではなく、

Cons 列挙子が指している先の List 値を変えたいということです。また、tail メソッドを追加して Cons 列挙子があるときに 2 番目の要素にアクセスするのが便利になるようにしています。

リスト 15-26 でリスト 15-25 の定義を使用する main 関数を追加しています。このコードは、a にリストを、b に a のリストを指すリストを作成します。それから a のリストを変更して b を指し、循環参照させます。その流れの中に過程のいろんな場所での参照カウントを示す println! 文が存在しています。

ファイル名: src/main.rs

```
# use crate::List::{Cons, Nil};
# use std::cell::RefCell;
# use std::rc::Rc;
#
# #[derive(Debug)]
# enum List {
#     Cons(i32, RefCell<Rc<List>>),
#     Nil,
# }
#
# impl List {
#     fn tail(&self) -> Option<&RefCell<Rc<List>>> {
#         match self {
#             Cons(_, item) => Some(item),
#             Nil => None,
#         }
#     }
# }
#
fn main() {
    let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));

    //       "aの最初の参照カウント = {}"
    println!("a initial rc count = {}", Rc::strong_count(&a));
    //       "aの次の要素は = {:?}"
    println!("a next item = {:?}", a.tail());

    let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a))));

    //       "b作成後のaの参照カウント = {}"
    println!("a rc count after b creation = {}", Rc::strong_count(&a));
    //       "bの最初の参照カウント = {}"
    println!("b initial rc count = {}", Rc::strong_count(&b));
    //       "bの次の要素 = {:?}"
    println!("b next item = {:?}", b.tail());

    if let Some(link) = a.tail() {
        *link.borrow_mut() = Rc::clone(&b);
    }

    //       "aを変更後のbの参照カウント = {}"
```

```
    println!("b rc count after changing a = {}", Rc::strong_count(&b));
    //       "aを変更後のaの参照カウント = {}"
    println!("a rc count after changing a = {}", Rc::strong_count(&a));

    // Uncomment the next line to see that we have a cycle;
    // it will overflow the stack
    // 次の行のコメントを外して循環していると確認してください;
    // スタックオーバーフローします
    // //       "aの次の要素 = {:?}"
    // println!("a next item = {:?}", a.tail());
}
```

リスト 15-26: 2 つの `List` 値がお互いを指して循環参照する

最初のリストが `5`, `Nil` の `List` 値を保持する `Rc<List>` インスタンスを変数 `a` に生成します。そして、値 10 と `a` のリストを指す別の `List` 値を保持する `Rc<List>` インスタンスを変数 `b` に生成します。

`a` が `Nil` ではなく `b` を指すように変更して、循環させます。`tail` メソッドを使用して、`a` の `RefCell<Rc<List>>` への参照を得ることで循環させて、この参照は変数 `link` に配置します。それから `RefCell<Rc<List>>` の `borrow_mut` メソッドを使用して中の値を `Nil` 値を持つ `Rc<List>` から、`b` の `Rc<List>` に変更します。

最後の `println!` を今だけコメントアウトしたまま、このコードを実行すると、こんな出力が得られます:

```
$ cargo run
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
    Finished dev [unoptimized + debuginfo] target(s) in 0.53s
     Running `target/debug/cons-list`
a initial rc count = 1
a next item = Some(RefCell { value: Nil })
a rc count after b creation = 2
b initial rc count = 1
b next item = Some(RefCell { value: Cons(5, RefCell { value: Nil }) })
b rc count after changing a = 2
a rc count after changing a = 2
```

`a` のリストを `b` を指すように変更した後の `a` と `b` の `Rc<List>` インスタンスの参照カウントは 2 です。`main` の終端で、コンパイラは変数 `b` をドロップし、`b` `Rc<List>` インスタンスの参照カウントを 2 から 1 に減らします。`Rc<List>` の参照カウントが 0 ではなく 1 なので、`Rc<List>` がヒープに確保していたメモリは、この時点ではドロップされません。次に、コンパイラは `a` をドロップし、`a` `Rc<List>` インスタンスの参照カウントも同様に 2 から 1 に減らします。他の `Rc<List>` インスタンスがまだ参照しているので、このインスタンスのメモリもドロップされません。リストに割り当てられたメモリは、永遠に回収されないままになるでしょう。この循環参照を可視化するために、図 15-4 に図式を作成しました。

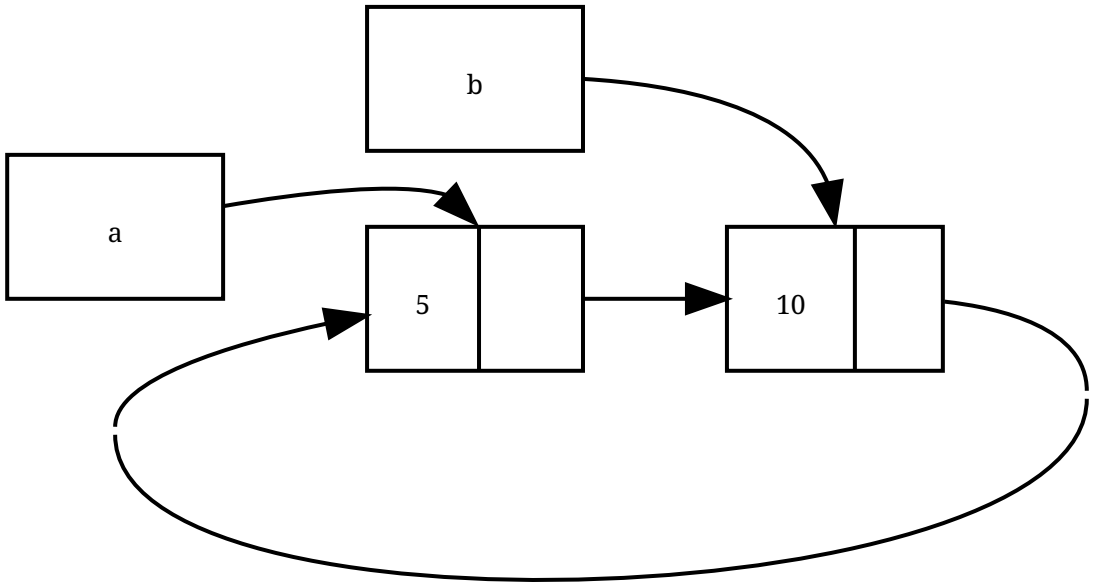


図 15-4: お互いを指すリスト a と b の循環参照

最後の `println!` のコメントを外してプログラムを実行したら、a が b を指して、b が a を指してと、スタックがオーバーフローするまでコンパイラはこの循環を出力しようとするでしょう。

現実世界のプログラムと比較すれば、この例で循環参照を作成してしまうという結果は、それほど悲壮なものではありません: 循環参照を作った直後にプログラムが終了するからです。しかしながら、より複雑なプログラムが多くのメモリを循環で確保し長い間その状態を保ったら、プログラムは必要以上のメモリを使用し、使用可能なメモリを枯渇させてシステムを参らせてしまう可能性があります。

循環参照は簡単にできることではありませんが、不可能というわけでもありません。`Rc<T>` 値を含む `RefCell<T>` 値があるなどの内部可変性と参照カウントのある型がネストして組み合わさっていたら、循環していないことを保証しなければなりません; コンパイラがそれを捕捉することを信頼できないのです。循環参照をするのは、自動テストやコードレビューなどの他のソフトウェア開発手段を使用して最小化すべきプログラム上のロジックバグでしょう。

循環参照を回避する別の解決策は、ある参照は所有権を表現して他の参照はしないというようにデータ構造を再構成することです。結果として、所有権のある関係と所有権のない関係からなる循環ができ、所有権のある関係だけが、値がドロップされうるかどうかに影響します。リスト 15-25 では、常に `Cons` 列挙子にリストを所有してほしいので、データ構造を再構成することはできません。親ノードと子ノードからなるグラフを使った例に目を向けて、どんな時に所有権のない関係が循環参照を回避するのに適切な方法になるか確認しましょう。

15.6.2　循環参照を回避する: Rc<T>を Weak<T>に変換する

ここまで、Rc::clone を呼び出すと Rc<T> インスタンスの strong_count が増えることと、strong_count が 0 になった時に Rc<T> インスタンスは片付けられることをデモしてきました。Rc::downgrade を呼び出し、Rc<T> への参照を渡すことで、Rc<T> インスタンス内部の値への**弱い参照** (weak reference) を作ることもできます。強い参照は、Rc<T> インスタンスの所有権を共有する方法です。弱い参照は所有関係を表現せず、そのカウントは Rc<T> インスタンスが片付けられるときに影響を与えません。ひとたび、関係する値の強い参照カウントが 0 になれば、弱い参照が関わる循環はなんでも破壊されるので、循環参照にはなりません。

Rc::downgrade を呼び出すと、型 Weak<T> のスマートポインタが得られます。Rc<T> インスタンスの strong_count を 1 増やす代わりに、Rc::downgrade を呼び出すと、weak_count が 1 増えます。strong_count 同様、Rc<T> 型は weak_count を使用して、幾つの Weak<T> 参照が存在しているかを追跡します。違いは、Rc<T> が片付けられるのに、weak_count が 0 である必要はないということです。

Weak<T> が参照する値はドロップされてしまっている可能性があるので、Weak<T> が指す値に何かをするには、値がまだ存在することを確認しなければなりません。Weak<T> の upgrade メソッドを呼び出すことでこれをしてください。このメソッドは Option<Rc<T>> を返します。Rc<T> 値がまだドロップされていなければ、Some の結果が、Rc<T> 値がドロップ済みなら、None の結果が得られます。upgrade が Option<Rc<T>> を返すので、コンパイラは、Some ケースと None ケースが扱われていることを確かめてくれ、無効なポインタは存在しません。

例として、要素が次の要素を知っているだけのリストを使うのではなく、要素が子要素**と**親要素を知っている木を作りましょう。

15.6.2.1　木データ構造を作る: 子ノードのある Node

手始めに子ノードを知っているノードのある木を構成します。独自の i32 値と子供の Node 値への参照を抱える Node という構造体を作ります:

ファイル名: src/main.rs

```
use std::cell::RefCell;
use std::rc::Rc;

#[derive(Debug)]
struct Node {
    value: i32,
    children: RefCell<Vec<Rc<Node>>>,
}
#
# fn main() {
#     let leaf = Rc::new(Node {
#         value: 3,
#         children: RefCell::new(vec![]),
```

```
#     });
#
#     let branch = Rc::new(Node {
#         value: 5,
#         children: RefCell::new(vec![Rc::clone(&leaf)]),
#     });
# }
```

Node に子供を所有してほしく、木の各 Node に直接アクセスできるよう、その所有権を変数と共有したいです。こうするために、Vec<T> 要素を型 Rc<Node> の値になるよう定義しています。どのノードが他のノードの子供になるかも変更したいので、Vec<Rc<Node>> の周りの children を RefCell<T> にしています。

次にこの構造体定義を使って値 3 と子供なしの leaf という 1 つの Node インスタンスと、値 5 と leaf を子要素の一つとして持つ branch という別のインスタンスを作成します。リスト 15-27 のようにですね:

ファイル名: src/main.rs

```
# use std::cell::RefCell;
# use std::rc::Rc;
#
# #[derive(Debug)]
# struct Node {
#     value: i32,
#     children: RefCell<Vec<Rc<Node>>>,
# }
#
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        children: RefCell::new(vec![]),
    });

    let branch = Rc::new(Node {
        value: 5,
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });
}
```

リスト 15-27: 子供なしの leaf ノードと leaf を子要素に持つ branch ノードを作る

leaf の Rc<Node> をクローンし、branch に格納しているので、leaf の Node は leaf と branch という 2 つの所有者を持つことになります。branch.children を通して branch から leaf へ辿ることはできるものの、leaf から branch へ辿る方法はありません。理由は、leaf には branch への参照がなく、関係していることを知らないからです。leaf に branch が親であることを知ってほしいです。次はそれを行います。

15.6.2.2　子供から親に参照を追加する

子供に親の存在を気付かせるために、Node 構造体定義に parent フィールドを追加する必要があります。parent の型を決める際に困ったことになります。Rc<T> を含むことができないのはわかります。そうしたら、leaf.parent が branch を指し、branch.children が leaf を指して循環参照になり、strong_count 値が絶対に 0 にならなくなってしまうからです。

この関係を別の方法で捉えると、親ノードは子供を所有すべきです: 親ノードがドロップされたら、子ノードもドロップされるべきなのです。ですが、子供は親を所有するべきではありません: 子ノードをドロップしても、親はまだ存在するべきです。弱い参照を使う場面ですね！

従って、Rc<T> の代わりに parent の型を Weak<T> を使ったもの、具体的には RefCell<Weak<Node>> にします。さあ、Node 構造体定義はこんな見た目になりました:

ファイル名: src/main.rs

```
use std::cell::RefCell;
use std::rc::{Rc, Weak};

#[derive(Debug)]
struct Node {
    value: i32,
    parent: RefCell<Weak<Node>>,
    children: RefCell<Vec<Rc<Node>>>,
}
#
# fn main() {
#     let leaf = Rc::new(Node {
#         value: 3,
#         parent: RefCell::new(Weak::new()),
#         children: RefCell::new(vec![]),
#     });
#
#     //       "leafの親 = {:?}"
#     println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
#
#     let branch = Rc::new(Node {
#         value: 5,
#         parent: RefCell::new(Weak::new()),
#         children: RefCell::new(vec![Rc::clone(&leaf)]),
#     });
#
#     *leaf.parent.borrow_mut() = Rc::downgrade(&branch);
#
#     println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
# }
```

ノードは親ノードを参照できるものの、所有はしないでしょう。リスト 15-28 で、leaf ノードが親の branch を参照できるよう、この新しい定義を使用するように main を更新します:

ファイル名: src/main.rs

```
# use std::cell::RefCell;
# use std::rc::{Rc, Weak};
#
# #[derive(Debug)]
# struct Node {
#     value: i32,
#     parent: RefCell<Weak<Node>>,
#     children: RefCell<Vec<Rc<Node>>>,
# }
#
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    //       "leafの親 = {:?}"
    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());

    let branch = Rc::new(Node {
        value: 5,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });

    *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
}
```

リスト 15-28: 親ノードの branch への弱い参照がある leaf ノード

leaf ノードを作成することは、parent フィールドの例外を除いてリスト 15-27 に似ています: leaf は親なしで始まるので、新しく空の Weak<Node> 参照インスタンスを作ります。

この時点で upgrade メソッドを使用して leaf の親への参照を得ようとすると、None 値になります。このことは、最初の println! 文の出力でわかります:

```
leaf parent = None
```

branch ノードを作る際、branch には親ノードがないので、こちらも parent フィールドには新しい Weak<Node> 参照が入ります。それでも、leaf は branch の子供になっています。一旦 branch に Node インスタンスができたら、leaf を変更して親への Weak<Node> 参照を与えることができます。leaf の parent フィールドには、RefCell<Weak<Node>> の borrow_mut メソッドを使用して、それから Rc::downgrade 関数を使用して、branch の Rc<Node> から branch への Weak<Node> 参照を作ります。

再度 leaf の親を出力すると、今度は branch を保持する Some 列挙子が得られます: これで leaf が

親にアクセスできるようになったのです！ `leaf` を出力すると、リスト 15-26 で起こっていたような最終的にスタックオーバーフローに行き着く循環を避けることもできます; `Weak<Node>` 参照は、`(Weak)` と出力されます:

```
leaf parent = Some(Node { value: 5, parent: RefCell { value: (Weak) },
children: RefCell { value: [Node { value: 3, parent: RefCell { value: (Weak) },
children: RefCell { value: [] } }] } })
```

無限の出力が欠けているということは、このコードは循環参照しないことを示唆します。このことは、`Rc::strong_count` と `Rc::weak_count` を呼び出すことで得られる値を見てもわかります。

15.6.2.3　strong_count と weak_count への変更を可視化する

新しい内部スコープを作り、`branch` の作成をそのスコープに移動することで、`Rc<Node>` インスタンスの `strong_count` と `weak_count` 値がどう変化するかを眺めましょう。そうすることで、`branch` が作成され、それからスコープを抜けてドロップされる時に起こることが確認できます。変更は、リスト 15-29 に示してあります:

ファイル名: src/main.rs

```
# use std::cell::RefCell;
# use std::rc::{Rc, Weak};
#
# #[derive(Debug)]
# struct Node {
#     value: i32,
#     parent: RefCell<Weak<Node>>,
#     children: RefCell<Vec<Rc<Node>>>,
# }
#
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    println!(
        "leaf strong = {}, weak = {}",
        Rc::strong_count(&leaf),
        Rc::weak_count(&leaf),
    );

    {
        let branch = Rc::new(Node {
            value: 5,
            parent: RefCell::new(Weak::new()),
            children: RefCell::new(vec![Rc::clone(&leaf)]),
        });
```

```
        *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

        println!(
            "branch strong = {}, weak = {}",
            Rc::strong_count(&branch),
            Rc::weak_count(&branch),
        );

        println!(
            "leaf strong = {}, weak = {}",
            Rc::strong_count(&leaf),
            Rc::weak_count(&leaf),
        );
    }

    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
    println!(
        "leaf strong = {}, weak = {}",
        Rc::strong_count(&leaf),
        Rc::weak_count(&leaf),
    );
}
```

リスト 15-29: 内側のスコープで branch を作成し、強弱参照カウントを調査する

leaf 作成後、その Rc<Node> の強カウントは 1、弱カウントは 0 になります。内側のスコープで branch を作成し、leaf に紐付け、この時点でカウントを出力すると、branch の Rc<Node> の強カウントは 1、弱カウントも 1 になります (leaf.parent が Weak<Node> で branch を指しているため)。leaf のカウントを出力すると、強カウントが 2 になっていることがわかります。branch が今は、branch.children に格納された leaf の Rc<Node> のクローンを持っているからですが、それでも弱カウントは 0 でしょう。

内側のスコープが終わると、branch はスコープを抜け、Rc<Node> の強カウントは 0 に減るので、この Node はドロップされます。leaf.parent からの弱カウント 1 は、Node がドロップされるか否かには関係ないので、メモリリークはしないのです！

このスコープの終端以後に leaf の親にアクセスしようとしたら、再び None が得られます。プログラムの終端で leaf の Rc<Node> の強カウントは 1、弱カウントは 0 です。変数 leaf が今では Rc<Node> への唯一の参照に再度なったからです。

カウントや値のドロップを管理するロジックは全て、Rc<T> や Weak<T> とその Drop トレイトの実装に組み込まれています。Node の定義で子供から親への関係は Weak<T> 参照になるべきと指定することで、循環参照やメモリリークを引き起こさずに親ノードに子ノードを参照させたり、その逆を行うことができます。

15.7　まとめ

この章は、スマートポインタを使用して Rust が既定で普通の参照に対して行うのと異なる保証や代償を行う方法を講義しました。`Box<T>` 型は、既知のサイズで、ヒープに確保されたデータを指します。`Rc<T>` 型は、ヒープのデータへの参照の数を追跡するので、データは複数の所有者を保有できます。内部可変性のある `RefCell<T>` 型は、不変型が必要だけれども、その型の中の値を変更する必要がある時に使用できる型を与えてくれます; また、コンパイル時ではなく実行時に借用規則を強制します。

`Deref` と `Drop` トレイトについても議論しましたね。これらは、スマートポインタの多くの機能を可能にしてくれます。メモリリークを引き起こす循環参照と `Weak<T>` でそれを回避する方法も探究しました。

この章で興味をそそられ、独自のスマートポインタを実装したくなったら、もっと役に立つ情報を求めて、“The Rustonomicon” をチェックしてください。

> 訳注: 日本語版の The Rustonomicon はこちらです。

次は、Rust での並行性について語ります。もういくつか新しいスマートポインタについてさえも学ぶでしょう。

第16章

恐れるな！ 並行性

並行性を安全かつ効率的に扱うことは、Rust の別の主な目標です。**並行プログラミング**は、プログラムの異なる部分が独立して実行することであり、**並列プログラミング**はプログラムの異なる部分が同時に実行することですが、多くのコンピュータが複数のプロセッサの利点を生かすようになるにつれ、重要度を増しています。歴史的に、これらの文脈で行うプログラミングは困難で、エラーが起きやすいものでした: Rust はこれを変えると願っています。

当初、Rust チームは、メモリ安全性を保証することと、並行性問題を回避することは、異なる方法で解決すべき別々の課題だと考えていました。時間とともに、チームは、所有権と型システムは、メモリ安全性**と**並行性問題を管理する役に立つ一連の強力な道具であることを発見しました。所有権と型チェックを活用することで、多くの並行性エラーは、実行時エラーではなくコンパイル時エラーになります。故に、実行時に並行性のバグが起きた状況と全く同じ状況を再現しようと時間を浪費させるよりも、不正なコードはコンパイルを拒み、問題を説明するエラーを提示するでしょう。結果として、プロダクトになった後でなく、作業中にコードを修正できます。Rust のこの方向性を**恐れるな！並行性**とニックネーム付けしました。これにより、潜在的なバグがなく、かつ、新しいバグを導入することなく簡単にリファクタリングできるコードを書くことができます。

> 注釈: 簡潔性のため、並行または並列と述べることで正確を期するのではなく、多くの問題を**並行**と割り切ってしまいます。この本がもし**並行性あるいは並列性**に関した本ならば、詳述していたでしょう。この章に対しては、**並行**を使ったら、脳内で**並行または並列**と置き換えてください。

多くの言語は、自分が提供する並行性問題を扱う解決策について独断的です。例えば、Erlang には、メッセージ受け渡しの並行性に関する素晴らしい機能がありますが、スレッド間で状態を共有することに関しては、曖昧な方法しかありません。可能な解決策の一部のみをサポートすることは、高級言語にとっては合理的な施策です。なぜなら、高級言語は一部の制御を失う代わりに抽象化することから恩恵を受けるからです。ところが、低級言語は、どんな場面でも最高のパフォーマンスで解決

策を提供すると想定され、ハードウェアに関してほとんど抽象化はしません。そのため、Rust は、自分の状況と必要性に適した方法が何であれ、問題をモデル化するためのいろんな道具を備えています。

こちらが、この章で講義する話題です:

- スレッドを生成して、複数のコードを同時に走らせる方法
- チャンネルがスレッド間でメッセージを送る**メッセージ受け渡し**並行性
- 複数のスレッドが何らかのデータにアクセスする**状態共有**並行性
- 標準ライブラリが提供する型だけでなく、ユーザが定義した型に対しても Rust の並行性の安全保証を拡張する `Sync` と `Send` トレイト

16.1　スレッドを使用してコードを同時に走らせる

多くの現代の OS では、実行中のプログラムのコードは**プロセス**で走り、OS は同時に複数のプロセスを管理するでしょう。プログラム内では、独立した部分を同時に実行することもできます。これらの独立した部分を走らせる機能を**スレッド**と呼びます。例えば web サーバは、同時に複数のリクエストに応答できるように、複数のスレッドを使用することができます。

同時に複数のタスクを実行するために、プログラム内の計算を複数のスレッドに分けることで、パフォーマンスを改善することができますが、複雑度も増します。スレッドは同時に走らせることができるので、異なるスレッドのコードが走る順番に関して、本来的に保証はありません。これは例えば以下のような問題を招きます:

- スレッドがデータやリソースに矛盾した順番でアクセスする競合状態
- 2 つのスレッドがお互いを待ち、両者が継続するのを防ぐデッドロック
- 特定の状況でのみ起き、確実な再現や修正が困難なバグ

Rust は、スレッドを使用する際の悪影響を軽減しようとしていますが、それでも、マルチスレッドの文脈でのプログラミングでは、注意深い思考と、シングルスレッドで走るプログラムでのそれとは異なるコード構造が必要です。

プログラミング言語によってスレッドはいくつかの方法で実装されており、多くの OS は、言語が呼び出すことができる、新規スレッドを生成するための API を提供しています。Rust 標準ライブラリは **1:1** モデルのスレッド実装を使用しており、1 つの言語スレッドに対して 1 つの OS スレッドを使用します。1:1 モデルとは異なるトレードオフを選択して、他のモデルのスレッドを実装するクレートもあります。

16.1.1　`spawn` で新規スレッドを生成する

新規スレッドを生成するには、`thread::spawn` 関数を呼び出し、新規スレッドで走らせたいコードを含むクロージャ (クロージャについては第 13 章で語りました) を渡します。リスト 16-1 の例は、

メインスレッドと新規スレッドからテキストを出力します:

ファイル名: src/main.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            //       "やあ！ 立ち上げたスレッドから数字{}だよ！ "
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        //       "やあ！ メインスレッドから数字{}だよ！ "
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

リスト 16-1: メインスレッドが別のものを出力する間に新規スレッドを生成して何かを出力する

Rust プログラムのメインスレッドが完了するときには、立ち上げられたすべてのスレッドは、その実行が完了したかどうかにかかわらず、停止されることに注意してください。このプログラムからの出力は毎回少々異なる可能性がありますが、だいたい以下のような感じでしょう:

```
hi number 1 from the main thread!
hi number 1 from the spawned thread!
hi number 2 from the main thread!
hi number 2 from the spawned thread!
hi number 3 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the main thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
```

`thread::sleep` を呼び出すと、少々の間、スレッドの実行を止め、違うスレッドを走らせることができます。スレッドはおそらく切り替わるでしょうが、保証はありません: OS がスレッドのスケジュールを行う方法によります。この実行では、コード上では立ち上げられたスレッドの print 文が先に現れているのに、メインスレッドが先に出力しています。また、立ち上げたスレッドには `i` が 9 になるまで出力するよう指示しているのに、メインスレッドが終了する前の 5 までしか到達していません。

このコードを実行してメインスレッドの出力しか目の当たりにできなかったり、オーバーラップがなければ、範囲の値を増やして OS がスレッド切り替えを行う機会を増やしてみてください。

16.1.2　join ハンドルで全スレッドの終了を待つ

リスト 16-1 のコードはほとんどの場合、メインスレッドが終了することで、立ち上げたスレッドが完了する前に停止されるだけでなく、スレッドの実行順に保証がないことから、立ち上げたスレッドがそもそも実行されるかどうかも保証できません。

thread::spawn の戻り値を変数に保存することで、立ち上げたスレッドが実行されなかったり、完了する前に終了してしまったりする問題を修正することができます。thread::spawn の戻り値の型は JoinHandle です。JoinHandle は、その join メソッドを呼び出したときにスレッドの終了を待つ所有された値です。リスト 16-2 は、リスト 16-1 で生成したスレッドの JoinHandle を使用し、join を呼び出して、main が終了する前に、立ち上げたスレッドが確実に完了する方法を示しています:

ファイル名: src/main.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap();
}
```

リスト 16-2: thread::spawn の JoinHandle を保存してスレッドが完了するのを保証する

ハンドルに対して join を呼び出すと、ハンドルが表すスレッドが終了するまで現在実行中のスレッドをブロックします。スレッドを**ブロック**するとは、そのスレッドが動いたり、終了したりすることを防ぐことです。join の呼び出しをメインスレッドの for ループの後に配置したので、リスト 16-2 を実行すると、以下のように出力されるはずです:

```
hi number 1 from the main thread!
hi number 2 from the main thread!
hi number 1 from the spawned thread!
hi number 3 from the main thread!
hi number 2 from the spawned thread!
hi number 4 from the main thread!
```

```
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!
```

2 つのスレッドが代わる代わる実行されていますが、`handle.join()` 呼び出しのためにメインスレッドは待機し、立ち上げたスレッドが終了するまで終わりません。

ですが、代わりに `handle.join()` を `for` ループの前に移動したらどうなるのか確認しましょう。こんな感じに:

ファイル名: src/main.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    handle.join().unwrap();

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

メインスレッドは、立ち上げたスレッドが終了するまで待ち、それから `for` ループを実行するので、以下のように出力はもう混ざらないでしょう:

```
hi number 1 from the spawned thread!
hi number 2 from the spawned thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!
hi number 1 from the main thread!
hi number 2 from the main thread!
hi number 3 from the main thread!
hi number 4 from the main thread!
```

どこで join を呼ぶかといったほんの些細なことが、スレッドが同時に走るかどうかに影響することもあります。

16.1.3　スレッドで move クロージャを使用する

thread::spawn に渡されるクロージャでは、move キーワードを多用することになるでしょう。そうすることで、クロージャは環境から使用する値の所有権を奪い、あるスレッドから別のスレッドに値の所有権を移すからです。第 13 章の「参照をキャプチャするか、所有権を移動するか」節では、クロージャの文脈で move について議論しました。それでは、move と thread::spawn の相互作用により集中していきましょう。

リスト 16-1 において、thread::spawn に渡したクロージャには引数がなかったことに注目してください: 立ち上げたスレッドのコードでメインスレッドからのデータは何も使用していないのです。立ち上げたスレッドでメインスレッドのデータを使用するには、立ち上げるスレッドのクロージャは、必要な値をキャプチャしなければなりません。リスト 16-3 は、メインスレッドでベクタを生成し、立ち上げたスレッドで使用する試みを示しています。しかしながら、すぐにわかるように、これはまだ動きません。

ファイル名: src/main.rs

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        //       "こちらがベクタ: {:?}"
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

リスト 16-3: 別のスレッドでメインスレッドが生成したベクタを使用しようとする

クロージャは v を使用しているので、v をキャプチャし、クロージャの環境の一部にしています。thread::spawn はこのクロージャを新しいスレッドで走らせるので、その新しいスレッド内で v にアクセスできるはずです。しかし、このコードをコンパイルすると、以下のようなエラーが出ます:

```
$ cargo run
   Compiling threads v0.1.0 (file:///projects/threads)
error[E0373]: closure may outlive the current function, but it borrows `v`,
    which is owned by the current function
(エラー: クロージャは現在の関数よりも長生きするかもしれませんが、現在の関数が所
```

```
    有している`v`を借用しています)
 --> src/main.rs:6:32
  |
6 |     let handle = thread::spawn(|| {
  |                                ^^ may outlive borrowed value `v`
  |                                   (借用されている値`v`より長生きするかもしれ
      ません)
7 |         println!("Here's a vector: {:?}", v);
  |                                            - `v` is borrowed here
  |                                             (`v`はここで借用されています)
  |
note: function requires argument type to outlive `'static`
(注釈: 関数は引数型が`'static`より長生きすることを要求しています)
 --> src/main.rs:6:18
  |
6 |       let handle = thread::spawn(|| {
  |  __________________^
7 | |         println!("Here's a vector: {:?}", v);
8 | |     });
  | |______^
help: to force the closure to take ownership of `v` (and any other referenced
    variables), use the `move` keyword
(ヘルプ: `v`(や他の参照されている変数)の所有権をクロージャに奪わせるには、`move`
    キーワードを使用してください)
  |
6 |     let handle = thread::spawn(move || {
  |                                ++++

For more information about this error, try `rustc --explain E0373`.
error: could not compile `threads` (bin "threads") due to 1 previous error
```

Rust は v のキャプチャ方法を**推論**し、println! は v への参照のみを必要とするので、クロージャは、v を借用しようとします。ですが、問題があります: コンパイラには、立ち上げたスレッドがどのくらいの期間走るのかわからないので、v への参照が常に有効であるか把握できないのです。

リスト 16-4 は、v への参照がより有効でなさそうな筋書きです:

ファイル名: src/main.rs

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    });

    drop(v); // あちゃー！

    handle.join().unwrap();
}
```

リスト 16-4: v をドロップするメインスレッドから v への参照をキャプチャしようとするクロージャを伴うスレッド

このコードを実行できてしまうなら、立ち上げたスレッドはまったく実行されることなく即座にバックグラウンドに置かれる可能性があります。立ち上げたスレッドは内部に v への参照を保持していますが、メインスレッドは、第 15 章で議論した drop 関数を使用して、即座に v をドロップしています。そして、立ち上げたスレッドが実行を開始する時には、v はもう有効ではなく、参照も不正になるのです。あちゃー！

リスト 16-3 のコンパイルエラーを修正するには、エラーメッセージのアドバイスを活用できます:

```
help: to force the closure to take ownership of `v` (and any other referenced
    variables), use the `move` keyword
(ヘルプ: `v`(や他の参照されている変数)の所有権をクロージャに奪わせるには、`move`
    キーワードを使用してください)
  |
6 |     let handle = thread::spawn(move || {
  |                                ++++
```

クロージャの前に move キーワードを付することで、コンパイラに値を借用すべきと推論させるのではなく、クロージャに使用している値の所有権を強制的に奪わせます。リスト 16-5 に示したリスト 16-3 に対する変更は、コンパイルでき、意図通りに動きます:

ファイル名: src/main.rs

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(move || {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

リスト 16-5: move キーワードを使用してクロージャに使用している値の所有権を強制的に奪わせる

リスト 16-4 の、メインスレッドが drop を呼び出すコードを修正するためにも、move クロージャを使用して同じことを試したくなるかもしれません。しかしながら、リスト 16-4 が試みていることは別の理由によりできないので、この修正はうまくいきません。クロージャに move を付与したら、v をクロージャの環境にムーブするので、最早メインスレッドで drop を呼び出すことは叶わなくなるで

しょう。代わりにこのようなコンパイルエラーが出るでしょう:

```
$ cargo run
   Compiling threads v0.1.0 (file:///projects/threads)
error[E0382]: use of moved value: `v`
(エラー: ムーブされた値の使用: `v`)
  --> src/main.rs:10:10
   |
4  |     let v = vec![1, 2, 3];
   |         - move occurs because `v` has type `Vec<i32>`, which does not
       implement the `Copy` trait
   |           (`v`は`Copy`トレイトを実装しない`Vec<i32>`型を持つので、ムーブが発
       生します)
5  |
6  |     let handle = thread::spawn(move || {
   |                                ------- value moved into closure here
   |                                        (値はここでクロージャ内にムーブされま
       す)
7  |         println!("Here's a vector: {:?}", v);
   |                                            - variable moved due to use in
       closure
   |                                             (変数はクロージャ内で使用されて
       いるためムーブされます)
...
10 |     drop(v); // oh no!
   |          ^ value used here after move
   |            (値はここでムーブ後に使用されています)

For more information about this error, try `rustc --explain E0382`.
error: could not compile `threads` (bin "threads") due to 1 previous error
```

再三 Rust の所有権規則が救ってくれました！ リスト 16-3 のコードはエラーになりました。コンパイラが一時的に保守的になり、スレッドに対して v を借用しただけだったからで、これは、メインスレッドは理論上、立ち上げたスレッドの参照を不正化する可能性があることを意味します。v の所有権を立ち上げたスレッドに移動するとコンパイラに指示することで、メインスレッドはもう v を使用しないとコンパイラに保証しているのです。リスト 16-4 も同様に変更したら、メインスレッドで v を使用しようとする際に所有権の規則に違反することになります。move キーワードにより、Rust の保守的な借用のデフォルトが上書きされるのです; 所有権の規則を侵害させてくれないのです。

スレッドとスレッド API の基礎知識を得たので、スレッドで**できる**ことを見ていきましょう。

16.2　メッセージ受け渡しを使ってスレッド間でデータを転送する

人気度を増してきている安全な並行性を保証する一つのアプローチが**メッセージ受け渡し**で、スレッドやアクターがデータを含むメッセージを相互に送り合うことでやり取りします。こちらが、Go 言語のドキュメンテーションのスローガンにある考えです:「メモリを共有することでやり取りするな; 代わりにやり取りすることでメモリを共有しろ」

メッセージ送信による並行性を達成するために、Rust の標準ライブラリは**チャンネル**の実装を提供しています。チャンネルは、あるスレッドから別のスレッドへデータを送信する手段である、普遍的なプログラミング概念です。

プログラミングのチャンネルは、川などの、向きのある水路のように考えることができます（訳注: channel は日常語としては水路を意味します）。アヒルのおもちゃなどを川に置いたら、水路の終端まで下流に流れていきます。

チャンネルは、2 分割できます: 転送機と受信機です。転送機はアヒルのおもちゃを川に置く上流になり、受信機は、アヒルのおもちゃが行き着く下流になります。コードのある箇所が送信したいデータとともに転送機のメソッドを呼び出し、別の部分がメッセージが到着していないか受信側を調べます。転送機と受信機のどちらかがドロップされると、チャンネルは**閉じられた**と言います。

ここで、1 つのスレッドが値を生成し、それをチャンネルに送信し、別のスレッドがその値を受け取り、出力するプログラムに取り掛かります。チャンネルを使用してスレッド間に単純な値を送り、機能の説明を行います。一旦そのテクニックに慣れてしまえば、チャットシステムや、多くのスレッドが計算の一部を担い、結果をまとめる 1 つのスレッドにその部分を送るようなシステムなど、任意のスレッドが互いに通信する必要があるシステムに対して、チャンネルを使用できるでしょう。

まず、リスト 16-6 において、チャンネルを生成するものの、何もしません。チャンネル越しにどんな型の値を送りたいのかコンパイラがわからないため、これはまだコンパイルできないことに注意してください。

ファイル名: src/main.rs

```
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();
}
```

リスト 16-6: チャンネルを生成し、2 つの部品を `tx` と `rx` に代入する

`mpsc::channel` 関数で新しいチャンネルを生成しています; `mpsc` は **multiple producer, single consumer** を表しています。簡潔に言えば、Rust の標準ライブラリがチャンネルを実装している方法は、1 つのチャンネルが値を生成する複数の**送信**側と、その値を消費するたった 1 つの**受信**側を持つことができるということを意味します。複数の小川が互いに合わさって 1 つの大きな川になるところを想像してください: どの小川を通っても、送られたものは最終的に 1 つの川に行き着きます。今は、1 つの生成器から始めますが、この例が動作するようになったら、複数の生成器を追加します。

`mpsc::channel` 関数はタプルを返し、1 つ目の要素は、送信側（転送機）、2 つ目の要素は受信側（受信機）になります。`tx` と `rx` という略称は、多くの分野で伝統的に**転送機**と**受信機**にそれぞれ使用されているので、変数をそのように名付けて、各終端を示します。タプルを分配するパターンを伴う `let` 文を使用しています; `let` 文でパターンを使用することと分配については、第 18 章で議論しま

しょう。今のところは、このように `let` 文を使うと、`mpsc::channel` で返ってくるタプルの部品を抽出するのが便利になるということだけ知っておいてください。

立ち上げたスレッドがメインスレッドとやり取りするように、転送機を立ち上げたスレッドに移動し、1 文字列を送らせましょう。リスト 16-7 のようにですね。川の上流にアヒルのおもちゃを置いたり、チャットのメッセージをあるスレッドから別のスレッドに送るみたいですね。

ファイル名: src/main.rs

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });
}
```

リスト 16-7: `tx` を立ち上げたスレッドに移動し、「やあ」を送る

今回も、`thread::spawn` を使用して新しいスレッドを生成し、それから `move` を使用して、立ち上げたスレッドが `tx` を所有するようにクロージャに `tx` をムーブしています。立ち上げたスレッドは、メッセージをチャンネルを通して送信できるように、転送機を所有する必要があります。転送機には、送信したい値を取る `send` メソッドがあります。`send` メソッドは `Result<T, E>` 型を返すので、既に受信機がドロップされ、値を送信する場所がなければ、送信処理はエラーを返します。この例では、エラーの場合には、パニックするように `unwrap` を呼び出しています。ですが、実際のアプリケーションでは、ちゃんと扱うでしょう: 第 9 章に戻ってちゃんとしたエラー処理の方法を再確認してください。

リスト 16-8 において、メインスレッドのチャンネルの受信機から値を得ます。アヒルのおもちゃを川の終端で水から回収したり、チャットメッセージを受信するみたいですね。

ファイル名: src/main.rs

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });

    let received = rx.recv().unwrap();
```

```
    //        "取得しました: {}"
    println!("Got: {}", received);
}
```

リスト 16-8:「やあ」の値をメインスレッドで受け取り、出力する

受信機には有用なメソッドが 2 つあります: `recv` と `try_recv` です。**receive** の省略形である `recv` を使っています。これは、メインスレッドの実行をブロックし、値がチャンネルを流れてくるまで待機します。一旦値が送信されたら、`recv` はそれを `Result<T, E>` に含んで返します。転送機が閉じたら、`recv` はエラーを返し、もう値は来ないと通知します。

`try_recv` メソッドはブロックせず、代わりに即座に `Result<T, E>` を返します: メッセージがあったら、それを含む `Ok` 値、今回は何もメッセージがなければ、`Err` 値です。メッセージを待つ間にこのスレッドにすることが他にあれば、`try_recv` は有用です: `try_recv` を頻繁に呼び出し、メッセージがあったら処理し、それ以外の場合は、再度チェックするまでちょっとの間、他の作業をするループを書くことができるでしょう。

この例では、簡潔性のために `recv` を使用しました; メッセージを待つこと以外にメインスレッドがすべき作業はないので、メインスレッドをブロックするのは適切です。

リスト 16-8 のコードを実行したら、メインスレッドから値が出力されるところを目撃するでしょう:

```
Got: hi
```

完璧です！

16.2.1　チャンネルと所有権の転送

安全な並行コードを書く手助けをしてくれるので、所有権規則は、メッセージ送信で重要な役割を担っています。並行プログラミングでエラーを回避することは、Rust プログラム全体で所有権について考える利点です。実験をしてチャンネルと所有権がともに動いて、どう問題を回避するかをお見せしましょう: `val` 値を立ち上げたスレッドで、チャンネルに送った**後**に使用を試みます。リスト 16-9 のコードのコンパイルを試みて、このコードが許容されない理由を確認してください:

ファイル名: src/main.rs

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
```

```
        //       "valは{}です"
        println!("val is {}", val);
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

リスト 16-9: チャンネルに送信後に val の使用を試みる

ここで、tx.send 経由でチャンネルに送信後に val を出力しようとしています。これを許可するのは、悪い考えです: 一旦、値が他のスレッドに送信されたら、再度値を使用しようとする前にそのスレッドが変更したりドロップできてしまいます。可能性として、その別のスレッドの変更により、矛盾していたり存在しないデータのせいでエラーが発生したり、予期しない結果になるでしょう。ですが、リスト 16-9 のコードのコンパイルを試みると、Rust はエラーを返します:

```
$ cargo run
   Compiling message-passing v0.1.0 (file:///projects/message-passing)
error[E0382]: borrow of moved value: `val`
(エラー: ムーブされた値の借用: `val`)
  --> src/main.rs:10:31
   |
8  |         let val = String::from("hi");
   |             --- move occurs because `val` has type `String`, which does not
        implement the `Copy` trait
   |                (`val`は`Copy`トレイトを実装しない`String`型を持つので、ムー
       ブが発生します)
9  |         tx.send(val).unwrap();
   |                 --- value moved here
   |                    (値はここでムーブされます)
10 |         println!("val is {}", val);
   |                                ^^^ value borrowed here after move
   |                                   (ここでムーブ後に借用されます)
   |
   = note: this error originates in the macro `$crate::format_args_nl` which
       comes from the expansion of the macro `println` (in Nightly builds, run
       with -Z macro-backtrace for more info)
help: consider cloning the value if the performance cost is acceptable
(ヘルプ: パフォーマンスコストが許容できる場合は、クローンすることを検討してくだ
    さい)
   |
9  |         tx.send(val.clone()).unwrap();
   |                    ++++++++

For more information about this error, try `rustc --explain E0382`.
error: could not compile `message-passing` (bin "message-passing") due to 1
    previous error
```

並行性のミスがコンパイルエラーを招きました。send 関数は引数の所有権を奪い、値がムーブされ

ると、受信側が所有権を得るのです。これにより、送信後に誤って再度値を使用するのを防いでくれます; 所有権システムが、万事問題ないことを確認してくれます。

16.2.2　複数の値を送信し、受信側が待機するのを確かめる

リスト 16-8 のコードはコンパイルでき、動きましたが、2 つの個別のスレッドがお互いにチャンネル越しに会話していることは、明瞭に示されませんでした。リスト 16-10 において、リスト 16-8 のコードが並行に動いていることを証明する変更を行いました: 立ち上げたスレッドは、複数のメッセージを送信し、各メッセージ間で、1 秒待機します。

ファイル名: src/main.rs

```
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        // スレッドからやあ (hi from the thread)
        let vals = vec![
            String::from("hi"),
            String::from("from"),
            String::from("the"),
            String::from("thread"),
        ];

        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    for received in rx {
        println!("Got: {}", received);
    }
}
```

リスト 16-10: 複数のメッセージを送信し、メッセージ間で停止する

今回は、メインスレッドに送信したい文字列のベクタを立ち上げたスレッドが持っています。それらを繰り返し、各々個別に送信し、`Duration` の値 1 秒とともに `thread::sleep` 関数を呼び出すことで、メッセージ間で停止します。

メインスレッドにおいて、最早 `recv` 関数を明示的に呼んではいません: 代わりに、`rx` をイテレータとして扱っています。受信した値それぞれを出力します。チャンネルが閉じられると、繰り返しも終

わります。

リスト 16-10 のコードを走らせると、各行の間に 1 秒の待機をしつつ、以下のような出力を目の当たりにするはずです:

```
Got: hi
Got: from
Got: the
Got: thread
```

メインスレッドの `for` ループには停止したり、遅れせたりするコードは何もないので、メインスレッドが立ち上げたスレッドから値を受け取るのを待機していることがわかります。

16.2.3　転送機をクローンして複数の生成器を作成する

`mpsc` は、**mutiple producer, single consumer** の頭字語であると前述しました。`mpsc` を使い、リスト 16-10 のコードを拡張して、全ての値を同じ受信機に送信する複数のスレッドを生成しましょう。転送機をクローンすることでそうすることができます。リスト 16-11 のようにですね:

ファイル名: src/main.rs

```
# use std::sync::mpsc;
# use std::thread;
# use std::time::Duration;
#
# fn main() {
    // --snip--

    let (tx, rx) = mpsc::channel();

    let tx1 = tx.clone();
    thread::spawn(move || {
        let vals = vec![
            String::from("hi"),
            String::from("from"),
            String::from("the"),
            String::from("thread"),
        ];

        for val in vals {
            tx1.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    thread::spawn(move || {
        // 君のためにもっとメッセージを (more messages for you)
        let vals = vec![
            String::from("more"),
            String::from("messages"),
```

```
            String::from("for"),
            String::from("you"),
        ];

        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    for received in rx {
        println!("Got: {}", received);
    }

    // --snip--
# }
```

リスト 16-11: 複数の生成器から複数のメッセージを送信する

今回、最初のスレッドを立ち上げる前に、転送機に対して `clone` を呼び出しています。これにより、最初に立ち上げたスレッドに渡せる新しい転送機が得られます。元の転送機は、2 番目に立ち上げたスレッドに渡します。これにより 2 つスレッドが得られ、1 つの受信機にそれぞれ異なるメッセージを送信します。

コードを実行すると、出力は以下のようなものになるはずです:

```
Got: hi
Got: more
Got: from
Got: messages
Got: for
Got: the
Got: thread
Got: you
```

システム次第で、別の順番で値が出る可能性もあります。並行性が面白いと同時に難しい部分でもあります。異なるスレッドで色々な値を与えて `thread::sleep` で実験をしたら、走らせるたびにより非決定的になり、毎回異なる出力をするでしょう。

チャンネルの動作方法を見たので、他の並行性に目を向けましょう。

16.3　状態共有並行性

メッセージ受け渡しは、並行性を扱う素晴らしい方法ですが、唯一の方法ではありません。他の方法としては、複数のスレッドが同一の共有されたデータにアクセスする方法があるでしょう。Go 言語ドキュメンテーションのスローガンのこの部分を再び考えてください:「メモリを共有することでやり取りするな。」

メモリを共有することでやり取りするとはどんな感じなのでしょうか？ さらに、なぜメッセージ受け渡しに熱狂的な人は、メモリ共有を使うなと警告するのでしょうか？

ある意味では、どんなプログラミング言語のチャンネルも単独の所有権に類似しています。一旦チャンネルに値を転送したら、その値は最早使用することがないからです。メモリ共有並行性は、複数の所有権に似ています: 複数のスレッドが同時に同じメモリ位置にアクセスできるのです。第 15 章でスマートポインタが複数の所有権を可能にするのを目の当たりにしたように、異なる所有者を管理する必要があるので、複数の所有権は複雑度を増させます。Rust の型システムと所有権規則は、この管理を正しく行う大きな助けになります。例として、メモリ共有を行うより一般的な並行性の基本型の一つであるミューテックスを見てみましょう。

16.3.1 ミューテックスを使用して一度に 1 つのスレッドからデータにアクセスすることを許可する

ミューテックスは、どんな時も 1 つのスレッドにしかなんらかのデータへのアクセスを許可しないというように、**mutual exclusion**(相互排他) の省略形です。ミューテックスにあるデータにアクセスするには、ミューテックスのロックを所望することでアクセスしたいことをまず、スレッドは通知しなければなりません。ロックとは、現在誰がデータへの排他的アクセスを行なっているかを追跡するミューテックスの一部をなすデータ構造です。故に、ミューテックスはロックシステム経由で保持しているデータを**死守する** (guarding) と解説されます。

ミューテックスは、2 つの規則を覚えておく必要があるため、難しいという評判があります:

- データを使用する前にロックの獲得を試みなければならない。
- ミューテックスが死守しているデータの使用が終わったら、他のスレッドがロックを獲得できるように、データをアンロックしなければならない。

ミューテックスを現実世界の物で例えるなら、マイクが 1 つしかない会議のパネルディスカッションを思い浮かべてください。パネリストが発言できる前に、マイクを使用したいと申し出たり、通知しなければなりません。マイクを受け取ったら、話したいだけ話し、それから次に発言を申し出たパネリストにマイクを手渡します。パネリストが発言し終わった時に、マイクを手渡すのを忘れていたら、誰も他の人は発言できません。共有されているマイクの管理がうまくいかなければ、パネルは予定通りに機能しないでしょう！

ミューテックスの管理は、正しく行うのに著しく技巧を要することがあるので、多くの人がチャンネルに熱狂的になるわけです。しかしながら、Rust の型システムと所有権規則のおかげで、ロックとアンロックをおかしくすることはありません。

16.3.1.1 `Mutex<T>`の API

ミューテックスの使用方法の例として、ミューテックスをシングルスレッドの文脈で使うことから始めましょう。リスト 16-12 のようにですね:

ファイル名: src/main.rs

```
use std::sync::Mutex;

fn main() {
    let m = Mutex::new(5);

    {
        let mut num = m.lock().unwrap();
        *num = 6;
    }

    println!("m = {:?}", m);
}
```

リスト 16-12: 簡潔性のために Mutex<T> の API をシングルスレッドの文脈で探究する

多くの型同様、new という関連関数を使用して Mutex<T> を生成します。ミューテックス内部のデータにアクセスするには、lock メソッドを使用してロックを獲得します。この呼び出しは、現在のスレッドをブロックするので、ロックを得られる順番が来るまで何も作業はできません。

ロックを保持している他のスレッドがパニックしたら、lock の呼び出しは失敗するでしょう。その場合、誰もロックを取得することは叶わないので、unwrap すると決定し、そのような状況になったら、このスレッドをパニックさせます。

ロックを獲得した後、今回の場合、num と名付けられていますが、戻り値を中に入っているデータへの可変参照として扱うことができます。型システムにより、m の値を使用する前にロックを獲得していることが確認されます。m の型は Mutex<i32> であって i32 ではないので、i32 を使用できるようにするには、lock を呼び出さ**なければならない**のです。忘れることはあり得ません; 型システムにより、それ以外の場合に内部の i32 にアクセスすることは許されません。

お察しかもしれませんが、Mutex<T> はスマートポインタです。より正確を期すなら、lock を呼び出すと LockResult に包まれた形で MutexGuard というスマートポインタを**返却**し、これを unwrap の呼び出しによって処理しました。MutexGuard スマートポインタが、内部のデータを指す Deref を実装しています; このスマートポインタはさらに MutexGuard がスコープを外れた時に、自動的にロックを解除する Drop 実装もしていて、これが内部スコープの終わりで発生します。結果として、ロックの解除が自動的に行われるので、ロックの解除を忘れ、ミューテックスが他のスレッドで使用されるのを阻害するリスクを負いません。

ロックをドロップした後、ミューテックスの値を出力し、内部の i32 の値を 6 に変更できたことが確かめられるのです。

16.3.1.2　複数のスレッド間で Mutex<T>を共有する

さて、Mutex<T> を使って複数のスレッド間で値を共有してみましょう。10 個のスレッドを立ち上げ、各々カウンタの値を 1 ずつインクリメントさせるので、カウンタは 0 から 10 まで上がります。

次のリスト 16-13 の例はコンパイルエラーになりますが、そのエラーを使用して、`Mutex<T>` の使用法と、コンパイラがそれを正しく活用する手助けをしてくれる方法について学びます。

ファイル名: src/main.rs

```
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Mutex::new(0);
    let mut handles = vec![];

    for _ in 0..10 {
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

リスト 16-13: `Mutex<T>` により死守されているカウンタを 10 個のスレッドがそれぞれインクリメントする

リスト 16-12 のように、`counter` 変数を生成して `Mutex<T>` の内部に `i32` を保持しています。次に、数値の範囲を走査して 10 個のスレッドを生成しています。`thread::spawn` を使用して、全スレッドに同じクロージャを与えています。このクロージャは、スレッド内にカウンタをムーブし、`lock` メソッドを呼ぶことで `Mutex<T>` のロックを獲得し、それからミューテックスの値に 1 を足します。スレッドがクロージャを実行し終わったら、`num` はスコープ外に出てロックを解除するので、他のスレッドが獲得できるわけです。

メインスレッドで全ての join ハンドルを収集します。それからリスト 16-2 のように、各ハンドルに対して `join` を呼び出し、全スレッドが終了するのを確かめています。その時点で、メインスレッドはロックを獲得し、このプログラムの結果を出力します。

この例はコンパイルできないでしょうと仄めかしました。では、理由を探りましょう！

```
$ cargo run
   Compiling shared-state v0.1.0 (file:///projects/shared-state)
error[E0382]: borrow of moved value: `counter`
(エラー: ムーブされた値の借用: `counter`)
```

```
  --> src/main.rs:21:29
   |
5  |     let counter = Mutex::new(0);
   |         ------- move occurs because `counter` has type `Mutex<i32>`, which
       does not implement the `Copy` trait
   |                 (`counter`は`Copy`トレイトを実装しない`Mutex<i32>`型を持つの
       で、ムーブが発生します)
...
9  |         let handle = thread::spawn(move || {
   |                                    ------- value moved into closure here,
       in previous iteration of loop
   |                                            (値は、ループの前回の反復時に、こ
       こでクロージャ内にムーブされます)
...
21 |     println!("Result: {}", *counter.lock().unwrap());
   |                             ^^^^^^^ value borrowed here after move
   |                                    (値はここでムーブ後に借用されています)

For more information about this error, try `rustc --explain E0382`.
error: could not compile `shared-state` (bin "shared-state") due to 1 previous
    error
```

エラーメッセージには、counter 値はループの前回の反復時にムーブされたと書いてあります。コンパイラは、ロック counter の所有権を複数のスレッドに移動することはできないと教えてくれているのです。第 15 章で議論した、複数所有権の手法を使って、コンパイラエラーを修正しましょう。

16.3.1.3　複数のスレッドで複数の所有権

第 15 章で、スマートポインタの Rc<T> を使用して参照カウントの値を作ることで、1 つの値に複数の所有者を与えました。同じことをここでもして、どうなるか見ましょう。リスト 16-14 で Rc<T> に Mutex<T> を包含し、所有権をスレッドに移す前に Rc<T> をクローンします。

ファイル名: src/main.rs

```
use std::rc::Rc;
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Rc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Rc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
```

```
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

リスト 16-14: Rc<T> を使用して複数のスレッドに Mutex<T> を所有させようとする

再三、コンパイルし……別のエラーが出ました！ コンパイラはいろんなことを教えてくれています。

```
$ cargo run
   Compiling shared-state v0.1.0 (file:///projects/shared-state)
error[E0277]: `Rc<Mutex<i32>>` cannot be sent between threads safely
(エラー: `Rc<Mutex<i32>>`はスレッド間で安全に送信できません)
  --> src/main.rs:11:36
   |
11 |           let handle = thread::spawn(move || {
   |                        ------------- ^------
   |                        |             |
   |  _____________________|_____________within this `{closure@src/main.rs
       :11:36: 11:43}`
   | |                      |            (この`{[closure@src/main.rs:11:36:
       11:43}`の中で)
   | |                      |
   | |                      required by a bound introduced by this call
   | |                     (この呼び出しによって導入される境界によって必要とされ
       ます)
12 | |             let mut num = counter.lock().unwrap();
13 | |
14 | |             *num += 1;
15 | |         });
   | |_________^ `Rc<Mutex<i32>>` cannot be sent between threads safely
   |            (`Rc<Mutex<i32>>`はスレッド間で安全に送信できません)
   |
   = help: within `{closure@src/main.rs:11:36: 11:43}`, the trait `Send` is not
       implemented for `Rc<Mutex<i32>>`
   =(ヘルプ: `{closure@src/main.rs:11:36: 11:43}`の中で、トレイト`Send`は`Rc<
       Mutex<i32>>`に対して実装されていません
note: required because it's used within this closure
(注釈: このクロージャの中で使用されているので、要求されます)
  --> src/main.rs:11:36
   |
11 |         let handle = thread::spawn(move || {
   |                                    ^^^^^^^
note: required by a bound in `spawn`
(注釈: `spawn`の境界によって要求されます)
  --> /rustc/07dca489ac2d933c78d3c5158e3f43beefeb02ce/library/std/src/thread/mod
```

```
      .rs:678:1

For more information about this error, try `rustc --explain E0277`.
error: could not compile `shared-state` (bin "shared-state") due to 1 previous
    error
```

おお、このエラーメッセージはとても長ったらしいですね！ 注目すべき重要な部分はここです: `Rc<Mutex<i32>>` cannot be sent between threads safely。コンパイラは、その理由も伝えてくれています: the trait `Send` is not implemented for `Rc<Mutex<i32>>`。Send については、次の節で語ります: スレッドとともに使用している型が並行な場面で使われることを意図したものであることを保証するトレイトの 1 つです。

残念ながら、Rc<T> はスレッド間で共有するには安全ではないのです。Rc<T> が参照カウントを管理する際、clone が呼び出されるたびにカウントを追加し、クローンがドロップされるたびにカウントを差し引きます。しかし、並行基本型を使用してカウントの変更が別のスレッドに妨害されないことを確認していないのです。これは間違ったカウントにつながる可能性があり、今度はメモリリークや、使用し終わる前に値がドロップされることにつながる可能性のある潜在的なバグです。必要なのは、いかにも Rc<T> のようだけれども、参照カウントへの変更をスレッドセーフに行うものです。

16.3.1.4　Arc<T>で原子的な参照カウント

幸いなことに、Arc<T> は Rc<T> のような並行な状況で安全に使用できる型**です。a** は **atomic** を表し、原子的に参照カウントする型を意味します。アトミックは、ここでは詳しく講義しない並行性の別の基本型です: 詳細は、std::sync::atomic の標準ライブラリドキュメンテーションを参照されたし。現時点では、アトミックは、基本型のように動くけれども、スレッド間で共有しても安全なことだけ知っていれば良いです。

そうしたらあなたは、なぜ全ての基本型がアトミックでなく、標準ライブラリの型も標準で Arc<T> を使って実装されていないのか疑問に思う可能性があります。その理由は、スレッド安全性が、本当に必要な時だけ支払いたいパフォーマンスの犠牲とともに得られるものだからです。シングルスレッドで値に処理を施すだけなら、アトミックが提供する保証を強制する必要がない方がコードはより速く走るのです。

例に回帰しましょう: Arc<T> と Rc<T> の API は同じなので、use 行と new の呼び出しと clone の呼び出しを変更して、プログラムを修正します。リスト 16-15 は、ようやくコンパイルでき、動作します:

ファイル名: src/main.rs

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];
```

```
    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

リスト 16-15: `Arc<T>` を使用して `Mutex<T>` をラップし、所有権を複数のスレッド間で共有できるようにする

このコードは、以下のように出力します:

```
Result: 10
```

やりました！ 0 から 10 まで数え上げました。これは、あまり印象的ではないように思えるかもしれませんが、本当に `Mutex<T>` とスレッド安全性についていろんなことを教えてくれました。このプログラムの構造を使用して、カウンタをインクリメントする以上の複雑な処理を行うこともできるでしょう。この手法を使えば、計算を独立した部分に小分けにし、その部分をスレッドに分割し、それから `Mutex<T>` を使用して、各スレッドに最終結果を更新させることができます。

単純な数値演算を行おうとしているなら、`Mutex<T>` 型よりも単純な、標準ライブラリの `std::sync::atomic` モジュールによって提供される型があることに注意してください。これらの型はプリミティブ型に対する安全かつ並行的なアトミックアクセスを提供します。この例では `Mutex<T>` がどのように機能するかに集中できるように、プリミティブ型に対して `Mutex<T>` を使用することをあえて選択しました。

16.3.2 `RefCell<T>`/`Rc<T>`と `Mutex<T>`/`Arc<T>`の類似性

`counter` は不変なのに、その内部にある値への可変参照を得ることができたことに気付いたでしょうか; つまり、`Mutex<T>` は、`Cell` 系のように内部可変性を提供するわけです。第 15 章で `RefCell<T>` を使用して `Rc<T>` の内容を可変化できるようにしたのと同様に、`Mutex<T>` を使用して `Arc<T>` の内容を可変化しているのです。

気付いておくべき別の詳細は、`Mutex<T>` を使用する際にあらゆる種類のロジックエラーからは、コンパイラは保護してくれないということです。第 15 章で `Rc<T>` は、循環参照を生成してしまうリス

クを伴い、そうすると、2 つの Rc<T> の値がお互いを参照し合い、メモリリークを引き起こしてしまうことを思い出してください。同様に、Mutex<T> は**デッドロック**を生成するリスクを伴っています。これは、処理が 2 つのリソースをロックする必要があり、2 つのスレッドがそれぞれにロックを 1 つ獲得して永久にお互いを待ちあってしまうときに起こります。デッドロックに興味があるのなら、デッドロックのある Rust プログラムを組んでみてください; それからどんな言語でもいいので、ミューテックスに対してデッドロックを緩和する方法を調べて、Rust で是非、それを実装してみてください。Mutex<T> と MutexGuard に関する標準ライブラリの API ドキュメンテーションは、役に立つ情報を提供してくれます。

Send と Sync トレイトと、それらを独自の型で使用する方法について語って、この章を締めくくります。

16.4　Sync と Send トレイトで拡張可能な並行性

面白いことに、Rust 言語には、**寡**少な並行性機能があります。この章でここまでに語った並行性機能のほとんどは、標準ライブラリの一部であり、言語ではありません。並行性を扱う選択肢は、言語や標準ライブラリに制限されません; 独自の並行性機能を書いたり、他人が書いたものを利用したりできるのです。

ですが、2 つの並行性概念が言語に埋め込まれています: std::marker トレイトの Sync と Send です。

16.4.1　Send でスレッド間の所有権の転送を許可する

Send マーカートレイトは、Send を実装した型を持つ値の所有権をスレッド間で転送できることを示唆します。Rust のほとんどの型は Send ですが、Rc<T> を含めて一部例外があります: この型は、Rc<T> の値をクローンし、クローンしたものの所有権を別のスレッドに転送しようとしたら、両方のスレッドが同時に参照カウントを更新できてしまうので、Send になり得ません。このため、Rc<T> はスレッド安全性のためのパフォーマンスの犠牲を支払わなくても済む、シングルスレッド環境で使用するために実装されているわけです。

故に、Rust の型システムとトレイト境界により、Rc<T> の値を不安全にスレッド間で誤って送信することが絶対ないよう保証してくれるのです。リスト 16-14 でこれを試みた時には、the trait Send is not implemented for Rc<Mutex<i32>> というエラーが出ました。Send の Arc<T> に切り替えたら、コードはコンパイルできたわけです。

完全に Send の型からなる型も全て自動的に Send と印付けされます。生ポインタを除くほとんどの基本型も Send で、生ポインタについては第 19 章で議論します。

16.4.2　Sync で複数のスレッドからのアクセスを許可する

Sync マーカートレイトは、Sync を実装した型は、複数のスレッドから参照されても安全であることを示唆します。言い換えると、&T (T への不変参照) が Send なら、型 T は Sync であり、参照が他のスレッドに安全に送信できることを意味します。Send 同様、基本型は Sync であり、Sync の型からのみ構成される型もまた Sync です。

Send ではなかったのと同じ理由で、スマートポインタの Rc<T> もまた Sync ではありません。RefCell<T> 型 (これについては第 15 章で話しました) と関連する Cell<T> 系についても Sync ではありません。RefCell<T> が実行時に行う借用チェックの実装は、スレッド安全ではないのです。スマートポインタの Mutex<T> は Sync で、「複数のスレッド間で Mutex<T> を共有する」節で見たように、複数のスレッドでアクセスを共有するのに使用することができます。

16.4.3　Send と Sync を手動で実装するのは非安全である

Send と Sync トレイトから構成される型は自動的に Send と Sync にもなるので、それらのトレイトを手動で実装する必要はありません。マーカートレイトとして、実装すべきメソッドさえも何もありません。並行性に関連する不変条件を強制することに役立つだけなのです。

これらのトレイトを手動で実装するには、unsafe な Rust コードを実装することが関わってきます。unsafe な Rust コードを使用することについては第 19 章で語ります; とりあえず、重要な情報は、Send と Sync ではない部品からなる新しい並行な型を構成するには、安全性保証を保持するために、注意深い思考が必要になるということです。“The Rustonomicon” には、これらの保証とそれを保持する方法についての情報がより多くあります。

> 訳注: 日本語版の The Rustonomicon はこちらです。

16.5　まとめ

この本において並行性を見かけるのは、これで最後ではありません: 第 20 章のプロジェクトでは、この章の概念をここで議論した微小な例よりもより現実的な場面で使用するでしょう。

前述のように、Rust による並行性の取扱いのごく一部のみが言語仕様なので、多くの並行性の解決策はクレートとして実装されています。これらは標準ライブラリよりも迅速に進化するので、マルチスレッド環境で使用すべき現在の最先端のクレートを必ずネットで検索してください。

Rust の標準ライブラリは、メッセージ受け渡しにチャンネルを、並行の文脈で安全に使用できる、Mutex<T> や Arc<T> などのスマートポインタ型を提供しています。型システムと借用チェッカーにより、これらの解決策を使用するコードがデータ競合や無効な参照に行き着かないことを保証してくれます。一旦コードをコンパイルすることができたら、他の言語ではありふれている追跡困難な類のバグなしに、複数のスレッドでも喜んで動くので安心できます。並行プログラミングは、もはや恐れる

べき概念ではありません: 恐れることなく前進し、プログラムを並行にしてください！

次は、Rust プログラムが肥大化するにつれて問題をモデル化し、解決策を構造化する慣例的な方法について話します。さらに、Rust のイディオムがオブジェクト指向プログラミングで馴染み深いかもしれないイディオムにどのように関連しているかについても議論します。

第 17 章

Rust のオブジェクト指向プログラミング機能

オブジェクト指向プログラミング (OOP) は、プログラムをモデル化する手段です。プログラム上の概念としてのオブジェクトは、1960 年代のプログラミング言語 Simula で導入されました。このオブジェクトは、お互いにメッセージを渡し合うというアラン・ケイ (Alan Kay) のプログラミングアーキテクチャに影響を及ぼしました。彼は、このアーキテクチャを記述するために、**オブジェクト指向プログラミング**という用語を造語しました。多くの競合する定義が OOP が何かを記述しており、こうした定義の一部によれば、Rust はオブジェクト指向であり他の定義によれば、Rust はオブジェクト指向ではありません。この章では、広くオブジェクト指向と捉えられる特定の特徴と、それらの特徴がこなれた Rust でどう表現されるかを探究します。それからオブジェクト指向のデザインパターンを Rust で実装する方法を示し、そうすることと Rust の強みを活用して代わりの解決策を実装する方法の代償を議論します。

17.1 オブジェクト指向言語の特徴

言語がオブジェクト指向と考えられるのになければならない機能について、プログラミングコミュニティ内での総意はありません。Rust は OOP を含めた多くのプログラミングパラダイムに影響を受けています; 例えば、第 13 章で関数型プログラミングに由来する機能を探究しました。議論はあるかもしれませんが、OOP 言語は特定の一般的な特徴を共有しています。具体的には、オブジェクトやカプセル化、継承などです。それらの個々の特徴が意味するものと Rust がサポートしているかを見ましょう。

17.1.1　オブジェクトは、データと振る舞いを含む

エーリヒ・ガンマ (Erich Gamma)、リチャード・ヘルム (Richard Helm)、ラルフ・ジョンソン (Ralph Johnson)、ジョン・ブリシディース (John Vlissides) による本 **Design Patterns: Elements of Reusable Object-Oriented Software** (Addison-Wesley Professional, 1994) は、俗に **4 人のギャング**の本と呼ばれる、オブジェクト指向デザインパターンのカタログです。そこでは、OOP は以下のように定義されています:

> オブジェクト指向プログラムは、オブジェクトで構成される。オブジェクトは、データとそのデータを処理するプロシージャを梱包している。このプロシージャは、典型的に**メソッド**または**オペレーション**と呼ばれる。

この定義を使用すれば、Rust はオブジェクト指向です: 構造体と enum にはデータがありますし、`impl` ブロックが構造体と enum にメソッドを提供します。メソッドのある構造体と enum は、オブジェクトとは呼ばれないものの、GoF のオブジェクト定義によると、同じ機能を提供します。

17.1.2　カプセル化は、実装詳細を隠蔽する

OOP とよく紐づけられる別の側面は、カプセル化の思想です。これは、オブジェクトの実装詳細は、そのオブジェクトを使用するコードにはアクセスできないことを意味します。故に、オブジェクトと相互作用する唯一の手段は、その公開 API を通してです; オブジェクトを使用するコードは、オブジェクトの内部に到達して、データや振る舞いを直接変更できるべきではありません。このために、プログラマはオブジェクトの内部をオブジェクトを使用するコードを変更する必要なく、変更しリファクタリングできます。

カプセル化を制御する方法は、第 7 章で議論しました: `pub` キーワードを使用して、自分のコードのどのモジュールや型、関数、メソッドを公開するか決められ、既定ではそれ以外のものは全て非公開になります。例えば、`i32` 値のベクタを含むフィールドのある `AveragedCollection` という構造体を定義できます。この構造体はさらに、ベクタの値の平均を含むフィールドを持てます。つまり、平均は誰かが必要とする度に、オンデマンドで計算する必要はないということです。言い換えれば、`AveragedCollection` は、計算した平均をキャッシュしてくれるわけです。リスト 17-1 には、`AveragedCollection` 構造体の定義があります:

ファイル名: src/lib.rs

```
pub struct AveragedCollection {
    list: Vec<i32>,
    average: f64,
}
```

リスト 17-1: 整数のリストとコレクションの要素の平均を管理する `AveragedCollection` 構造体

構造体は、他のコードが使用できるように pub で印づけされていますが、構造体のフィールドは非公開のままです。値が追加されたりリストから削除される度に、平均も更新されることを保証したいので、今回の場合重要です。add や remove 、average メソッドを構造体に実装することでこれをします。リスト 17-2 のようにですね:

ファイル名: src/lib.rs

```
# pub struct AveragedCollection {
#     list: Vec<i32>,
#     average: f64,
# }
#
impl AveragedCollection {
    pub fn add(&mut self, value: i32) {
        self.list.push(value);
        self.update_average();
    }

    pub fn remove(&mut self) -> Option<i32> {
        let result = self.list.pop();
        match result {
            Some(value) => {
                self.update_average();
                Some(value)
            }
            None => None,
        }
    }

    pub fn average(&self) -> f64 {
        self.average
    }

    fn update_average(&mut self) {
        let total: i32 = self.list.iter().sum();
        self.average = total as f64 / self.list.len() as f64;
    }
}
```

リスト 17-2: AveragedCollection の add 、remove 、average 公開メソッドの実装

add 、remove 、average の公開メソッドが、AveragedCollection のインスタンス内のデータにアクセスまたは変更するための、唯一の方法になります。要素が add メソッドを使用して list に追加されたり、remove メソッドを使用して削除されたりすると、各メソッドの実装が average フィールドの更新を扱う非公開の update_average メソッドも呼び出します。

list と average フィールドを非公開のままにしているので、外部コードが要素を list フィールドに直接追加したり削除したりする方法はありません; そうでなければ、average フィールドは、list が

変更された時に同期されなくなる可能性があります。average メソッドは average フィールドの値を返し、外部コードに average を読ませるものの、変更は許可しません。

構造体 AveragedCollection の実装詳細をカプセル化したので、データ構造などの側面を将来容易に変更することができます。例を挙げれば、list フィールドに Vec<i32> ではなく HashSet<i32> を使うこともできます。add 、remove 、average といった公開メソッドのシグニチャが同じである限り、AveragedCollection を使用するコードは変更する必要がないでしょう。代わりに list を公開にしたら、必ずしもこうはならないでしょう: HashSet<i32> と Vec<i32> は、要素の追加と削除に異なるメソッドを持っているので、外部コードが直接 list を変更しているなら、外部コードも変更しなければならない可能性が高いでしょう。

カプセル化が、言語がオブジェクト指向と考えられるのに必要な側面ならば、Rust はその条件を満たしています。コードの異なる部分で pub を使用するかしないかという選択肢のおかげで、実装詳細をカプセル化することが可能になります。

17.1.3　型システム、およびコード共有としての継承

継承は、それによってオブジェクトが他のオブジェクトの定義から要素を受け継ぐことができる機構であり、それ故に、再定義する必要なく、親オブジェクトのデータと振る舞いを得ます。

言語がオブジェクト指向言語であるために継承がなければならないのならば、Rust は違います。親構造体のフィールドとメソッドの実装を受け継ぐ構造体をマクロを使わずに定義する方法はありません。

しかしながら、継承がプログラミング道具箱にあることに慣れていれば、そもそも継承に手を伸ばす理由によって、Rust で他の解決策を使用することができます。

継承を選択する理由は主に 2 つあります。1 つ目は、コードの再利用です: ある型に特定の振る舞いを実装し、継承により、その実装を他の型にも再利用できるわけです。Rust コードでは、デフォルトのトレイトメソッド実装を使用することで、限定された形でこれを行うことができ、これは、リスト 10-14 で Summary トレイトに summarize メソッドのデフォルト実装を追加した時に見かけました。Summary トレイトを実装する型は全て、追加のコードなく summarize メソッドが使用できます。これは、親クラスにメソッドの実装があり、継承した子クラスにもそのメソッドの実装があることと似ています。また、Summary トレイトを実装する時に、summarize メソッドのデフォルト実装を上書きすることもでき、これは、親クラスから継承したメソッドの実装を子クラスが上書きすることに似ています。

継承を使用するもう 1 つの理由は、型システムに関連しています: 親の型と同じ箇所で子供の型を使用できるようにです。これは、**多相性** (polymorphism) とも呼ばれ、複数のオブジェクトが特定の特徴を共有しているなら、実行時にお互いに代用できることを意味します。

17.1.4　多相性

多くの人にとって、多相性は、継承の同義語です。ですが、実際には複数の型のデータを取り扱えるコードを指すより一般的な概念です。継承について言えば、それらの型は一般的にはサブクラスです。

Rust は代わりにジェネリクスを使用して様々な可能性のある型を抽象化し、トレイト境界を使用してそれらの型が提供するものに制約を課します。これは時に、**パラメータ境界多相性** (bounded parametric polymorphism) と呼ばれます。

継承は、近年、多くのプログラミング言語において、プログラムの設計解決策としては軽んじられています。というのも、しばしば必要以上にコードを共有してしまう危険性があるからです。サブクラスは、必ずしも親クラスの特徴を全て共有するべきではないのに、継承ではそうなってしまうのです。これにより、プログラムの設計の柔軟性を失わせることもあります。また、道理に合わなかったり、メソッドがサブクラスには適用されないために、エラーを発生させるようなサブクラスのメソッドの呼び出しを引き起こす可能性が出てくるのです。さらに、単一継承のみを許可する (サブクラスは 1 つのクラスからのみ継承できる) 言語もあり、さらにプログラムの設計の柔軟性が制限されます。

これらの理由により Rust は、継承ではなくトレイトオブジェクトを使用するという異なるアプローチを取っています。Rust において、トレイトオブジェクトがどう多相性を可能にするかを見ましょう。

17.2　トレイトオブジェクトで異なる型の値を許容する

第 8 章で、ベクタの 1 つの制限は、たった 1 つの型の要素を保持することしかできないことだと述べました。リスト 8-9 で整数、浮動小数点数、テキストを保持する列挙子のある `SpreadsheetCell` enum を定義して、これを回避しました。つまり、各セルに異なる型のデータを格納しつつ、1 行のセルを表すベクタを保持するということです。コンパイル時にわかるある固定されたセットの型にしか取り替え可能な要素がならない場合には、完璧な解決策です。

ところが、時として、ライブラリの使用者が特定の場面で合法になる型のセットを拡張できるようにしたくなることがあります。これをどう実現する可能性があるか示すために、各アイテムに `draw` メソッドを呼び出してスクリーンに描画するという、GUI ツールで一般的なテクニックをしてあるリストの要素を走査する例の GUI ツールを作ります。GUI ライブラリの構造を含む `gui` と呼ばれるライブラリクレートを作成します。このクレートには、他人が使用できる `Button` や `TextField` などの型が包含されるかもしれません。さらに、`gui` の使用者は、描画可能な独自の型を作成したくなるでしょう: 例えば、ある人は `Image` を追加し、別の人は `SelectBox` を追加するかもしれません。

この例のために本格的な GUI ライブラリは実装するつもりはありませんが、部品がどう組み合わさるかは示します。ライブラリの記述時点では、他のプログラマが作成したくなる可能性のある型全てを知る由もなければ、定義することもできません。しかし、`gui` は異なる型の多くの値を追いかけ、

この異なる型の値に対して draw メソッドを呼び出す必要があることは、確かにわかっています。draw メソッドを呼び出した時に正確に何が起きるかを知っている必要はありません。値にそのメソッドが呼び出せるようあることだけわかっていればいいのです。

継承のある言語でこれを行うには、draw という名前のメソッドがある Component というクラスを定義するかもしれません。Button 、Image 、SelectBox などの他のクラスは、Component を継承し、故に draw メソッドを継承します。個々に draw メソッドをオーバーライドして、独自の振る舞いを定義するものの、フレームワークは、Component インスタンスであるかのようにその型全部を扱い、この型に対して draw を呼び出します。ですが、Rust に継承は存在しないので、使用者に新しい型で拡張してもらうために gui ライブラリを構成する他の方法が必要です。

17.2.1　一般的な振る舞いにトレイトを定義する

gui に欲しい振る舞いを実装するには、draw という 1 つのメソッドを持つ Draw というトレイトを定義します。それから**トレイトオブジェクト**を取るベクタを定義できます。トレイトオブジェクトは、指定されたトレイトを実装するある型のインスタンスと、実行時にその型に対するトレイトメソッドを探すのに使用されるテーブルの、両方を指します。& 参照や Box<T> スマートポインタなどの、何らかのポインタを指定し、それから dyn キーワード、そして対象のトレイトを指定する (トレイトオブジェクトがポインタを使用しなければならない理由については、第 19 章の「動的サイズ決定型と Sized トレイト」節で語ります) ことでトレイトオブジェクトを作成します。ジェネリックまたは具体的な型があるところにトレイトオブジェクトは使用できます。どこでトレイトオブジェクトを使用しようと、Rust の型システムは、コンパイル時にその文脈で使用されているあらゆる値がそのトレイトオブジェクトのトレイトを実装していることを保証します。結果としてコンパイル時に可能性のある型を全て知る必要はなくなるのです。

Rust では、構造体と enum を他の言語のオブジェクトと区別するために「オブジェクト」と呼ぶことを避けていることに触れましたね。構造体や enum において、構造体のフィールドのデータや impl ブロックの振る舞いは区分けされているものの、他の言語では 1 つの概念に押し込められるデータと振る舞いは、しばしばオブジェクトと分類されます。しかしながら、トレイトオブジェクトは、データと振る舞いをごちゃ混ぜにするという観点で他の言語のオブジェクトに近い**です**。しかし、トレイトオブジェクトは、データを追加できないという点で伝統的なオブジェクトと異なっています。トレイトオブジェクトは、他の言語のオブジェクトほど一般的に有用ではありません: その特定の目的は、共通の振る舞いに対して抽象化を行うことです。

リスト 17-3 は、draw という 1 つのメソッドを持つ Draw というトレイトを定義する方法を示しています:

ファイル名: src/lib.rs

```
pub trait Draw {
    fn draw(&self);
}
```

リスト 17-3: `Draw` トレイトの定義

この記法は、第 10 章のトレイトの定義方法に関する議論で馴染み深いはずです。その次は、新しい記法です: リスト 17-4 では、`components` というベクタを保持する `Screen` という名前の構造体を定義しています。このベクタの型は `Box<dyn Draw>` で、これはトレイトオブジェクトです; `Draw` トレイトを実装する `Box` 内部の任意の型に対する代役です。

ファイル名: src/lib.rs

```
# pub trait Draw {
#     fn draw(&self);
# }
#
pub struct Screen {
    pub components: Vec<Box<dyn Draw>>,
}
```

リスト 17-4: `Draw` トレイトを実装するトレイトオブジェクトのベクタを保持する `components` フィールドがある `Screen` 構造体の定義

`Screen` 構造体に、`components` の各要素に対して `draw` メソッドを呼び出す `run` というメソッドを定義します。リスト 17-5 のようにですね:

ファイル名: src/lib.rs

```
# pub trait Draw {
#     fn draw(&self);
# }
#
# pub struct Screen {
#     pub components: Vec<Box<dyn Draw>>,
# }
#
impl Screen {
    pub fn run(&self) {
        for component in self.components.iter() {
            component.draw();
        }
    }
}
```

リスト 17-5: 各コンポーネントに対して `draw` メソッドを呼び出す `Screen` の `run` メソッド

これは、トレイト境界を伴うジェネリックな型引数を使用する構造体を定義するのとは異なる動作をします。ジェネリックな型引数は、一度に 1 つの具体型にしか置き換えられないのに対して、トレ

イトオブジェクトは、実行時にトレイトオブジェクトに対して複数の具体型で埋めることができます。例として、ジェネリックな型とトレイト境界を使用してリスト 17-6 のように Screen 構造体を定義することもできました:

ファイル名: src/lib.rs

```
# pub trait Draw {
#     fn draw(&self);
# }
#
pub struct Screen<T: Draw> {
    pub components: Vec<T>,
}

impl<T> Screen<T>
where
    T: Draw,
{
    pub fn run(&self) {
        for component in self.components.iter() {
            component.draw();
        }
    }
}
```

リスト 17-6: ジェネリクスとトレイト境界を使用した Screen 構造体と run メソッドの対立的な実装

こうすると、全てのコンポーネントの型が Button だったり、TextField だったりする Screen のインスタンスに制限されてしまいます。絶対に同種のコレクションしか持つ予定がないのなら、ジェネリクスとトレイト境界は、定義がコンパイル時に具体的な型を使用するように単相化されるので、望ましいです。

一方で、メソッドがトレイトオブジェクトを使用すると、1 つの Screen インスタンスが、Box<Button> と Box<TextField> を含む Vec<T> を保持できます。この動作方法を見、それから実行時性能の裏の意味について語りましょう。

17.2.2　トレイトを実装する

さて、Draw トレイトを実装する型を追加しましょう。Button 型を提供します。ここも、実際に GUI ライブラリを実装することは、この本の範疇を超えているので、draw メソッドの本体は、何も有用な実装はしません。実装がどんな感じになるか想像するために、Button 構造体は、width 、height 、label フィールドを持っている可能性があります。リスト 17-7 に示したようにですね:

ファイル名: src/lib.rs

```
# pub trait Draw {
#     fn draw(&self);
# }
#
# pub struct Screen {
#     pub components: Vec<Box<dyn Draw>>,
# }
#
# impl Screen {
#     pub fn run(&self) {
#         for component in self.components.iter() {
#             component.draw();
#         }
#     }
# }
#
pub struct Button {
    pub width: u32,
    pub height: u32,
    pub label: String,
}

impl Draw for Button {
    fn draw(&self) {
        // code to actually draw a button
        // 実際にボタンを描画するコード
    }
}
```

リスト 17-7: `Draw` トレイトを実装するある `Button` 構造体

`Button` の `width` 、`height` 、`label` フィールドは、他のコンポーネントのフィールドとは異なるでしょう; 例えば `TextField` 型は、それらと同じフィールドに加えて `placeholder` フィールドを持つかもしれません。スクリーンに描画したい型のコンポーネントはそれぞれ `Draw` トレイトを実装しますが、`Button` がここでしているように、`draw` メソッドでは異なるコードを使用してその特定の型を描画する方法を定義しています (先にも書いた通り、実際の GUI コードはありませんが)。例えば、`Button` には、ユーザがボタンをクリックした時に起こることに関連するメソッドを含む、追加の `impl` ブロックがある可能性があります。この種のメソッドは、`TextField` のような型には適用されません。

ライブラリの使用者が、`width` 、`height` 、`options` フィールドのある `SelectBox` 構造体を実装しようと決めたら、`SelectBox` 型にも `Draw` トレイトを実装します。リスト 17-8 のようにですね:

ファイル名: src/main.rs

```
use gui::Draw;

struct SelectBox {
    width: u32,
```

```
    height: u32,
    options: Vec<String>,
}

impl Draw for SelectBox {
    fn draw(&self) {
        // code to actually draw a select box
        //セレクトボックスを実際に描画するコード
    }
}
#
# fn main() {}
```

リスト 17-8: gui を使用し、SelectBox 構造体に Draw トレイトを実装する別のクレート

ライブラリの使用者はもう、main 関数を書き、Screen インスタンスを生成できます。Screen インスタンスには、それぞれを Box<T> に放り込んでトレイトオブジェクト化して SelectBox と Button を追加できます。それから Screen インスタンスに対して run メソッドを呼び出すことができ、そうすると各コンポーネントの draw が呼び出されます。リスト 17-9 は、この実装を示しています:

ファイル名: src/main.rs

```
# use gui::Draw;
#
# struct SelectBox {
#     width: u32,
#     height: u32,
#     options: Vec<String>,
# }
#
# impl Draw for SelectBox {
#     fn draw(&self) {
#         // code to actually draw a select box
#     }
# }
#
use gui::{Button, Screen};

fn main() {
    let screen = Screen {
        components: vec![
            Box::new(SelectBox {
                width: 75,
                height: 10,
                options: vec![
                    // はい
                    String::from("Yes"),
                    // 多分
                    String::from("Maybe"),
                    // いいえ
```

```
                    String::from("No"),
                ],
            }),
            Box::new(Button {
                width: 50,
                height: 10,
                label: String::from("OK"),
            }),
        ],
    };

    screen.run();
}
```

リスト 17-9: トレイトオブジェクトを使って同じトレイトを実装する異なる型の値を格納する

ライブラリを記述した時点では、誰かが `SelectBox` 型を追加する可能性があるなんて知りませんでしたが、`Screen` の実装は、新しい型を処理し、描画することができました。何故なら、`SelectBox` は `Draw` トレイト、つまり、`draw` メソッドを実装しているからです。

この値の具体的な型ではなく、値が応答したメッセージにのみ関係するという概念は、動的型付け言語の**ダックタイピング**に似た概念です: アヒルのように歩き、鳴くならば、アヒルに違いないのです！ リスト 17-5 の `Screen` の `run` の実装では、`run` は、各コンポーネントの実際の型がなんであるか知る必要はありません。コンポーネントが、`Button` や `SelectBox` のインスタンスであるかを確認することはなく、コンポーネントの `draw` メソッドを呼び出すだけです。`components` ベクタで `Box<dyn Draw>` を値の型として指定することで、`Screen` を、`draw` メソッドを呼び出せる値を必要とするように定義できたのです。

17.2.2.1　注釈: ダックタイピングについて

ご存知かもしれませんが、ダックタイピングについて補足です。ダックタイピングとは、動的型付け言語や C++ のテンプレートで使用される、特定のフィールドやメソッドがあることを想定してコンパイルを行い、実行時に実際にあることを確かめるというプログラミング手法です。ダック・テストという思考法に由来するそうです。

ダックタイピングの利点は、XML や JSON など、厳密なスキーマがないことが多い形式を扱いやすくなること、欠点は、実行してみるまで動くかどうかわからないことでしょう。

トレイトオブジェクトと Rust の型システムを使用してダックタイピングを活用したコードに似たコードを書くことの利点は、実行時に値が特定のメソッドを実装しているか確認したり、値がメソッドを実装していない時にエラーになることを心配したりする必要は絶対になく、とにかく呼び出せることです。コンパイラは、値が、トレイトオブジェクトが必要としているトレイトを実装していなければ、コンパイルを通さないのです。

例えば、リスト 17-10 は、コンポーネントに `String` のある `Screen` を作成しようとした時に起こ

ることを示しています:

ファイル名: src/main.rs

```
use gui::Screen;

fn main() {
    let screen = Screen {
        components: vec![Box::new(String::from("Hi"))],
    };

    screen.run();
}
```

リスト 17-10: トレイトオブジェクトのトレイトを実装しない型の使用を試みる

`String` は `Draw` トレイトを実装していないので、このようなエラーが出ます:

```
$ cargo run
   Compiling gui v0.1.0 (file:///projects/gui)
error[E0277]: the trait bound `String: Draw` is not satisfied
 --> src/main.rs:5:26
  |
5 |         components: vec![Box::new(String::from("Hi"))],
  |                          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^ the trait `Draw` is
      not implemented for `String`
  |
  = help: the trait `Draw` is implemented for `Button`
  = note: required for the cast from `Box<String>` to `Box<dyn Draw>`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `gui` (bin "gui") due to 1 previous error
```

このエラーは、渡すことを意図していないものを `Screen` に渡しているので、異なる型を渡すべきか、`Screen` が `draw` を呼び出せるように `String` に `Draw` を実装するべきのどちらかであることを知らせてくれています。

17.2.3　トレイトオブジェクトは、ダイナミックディスパッチを行う

第 10 章の「ジェネリクスを使用したコードのパフォーマンス」節でジェネリクスに対してトレイト境界を使用した時に、コンパイラが行う単相化過程の議論を思い出してください: コンパイラは、関数やメソッドのジェネリックでない実装を、ジェネリックな型引数の箇所に使用している具体的な型に対して生成するのでした。単相化の結果吐かれるコードは、**スタティックディスパッチ**を行い、これは、コンパイル時にコンパイラがどのメソッドを呼び出しているかわかる時のことです。これは、**ダイナミックディスパッチ**とは対照的で、この時、コンパイラは、コンパイル時にどのメソッドを呼び出しているのかわかりません。ダイナミックディスパッチの場合、コンパイラは、どのメソッドを

呼び出すか実行時に弾き出すコードを生成します。

トレイトオブジェクトを使用すると、コンパイラはダイナミックディスパッチを使用しなければなりません。コンパイラは、トレイトオブジェクトを使用しているコードで使用される可能性のある型全てを把握しないので、どの型に実装されたどのメソッドを呼び出すかわからないのです。代わりに実行時に、トレイトオブジェクト内でポインタを使用して、コンパイラは、どのメソッドを呼ぶか知ります。この検索は、スタティックディスパッチでは発生しない実行時コストを招きます。また、ダイナミックディスパッチは、コンパイラがメソッドのコードをインライン化することも妨げ、そのため、ある種の最適化が不可能になります。ですが、リスト 17-5 で記述し、リスト 17-9 ではサポートできたコードで追加の柔軟性を確かに得られたので、考慮すべき代償です。

17.3　オブジェクト指向デザインパターンを実装する

ステートパターンは、オブジェクト指向デザインパターンの 1 つです。このパターンの肝は、値が内部的に持つことができる状態の集合を定義するということです。状態は**ステートオブジェクト**の集合として表現され、その状態に基づいて値の振る舞いが変化します。ここでは、自身の状態を保持するフィールドを持つ、ブログ記事構造体の例に取り組んでいきます。この構造体のフィールドは「草稿」、「査読」、「公開済み」からなる集合のうちのいずれかの状態オブジェクトになるでしょう。

ステートオブジェクトは機能を共有します: Rust では、もちろん、オブジェクトと継承ではなく、構造体とトレイトを使用します。各ステートオブジェクトは、自身の振る舞いと別の状態に変化すべき時を司ることに責任を持ちます。ステートオブジェクトを保持する値は、状態ごとの異なる振る舞いや、いつ状態が移行するかについては何も知りません。

ステートパターンを使用することの利点は、プログラムの業務要件が変わる時、状態を保持する値のコードや、値を使用するコードを変更する必要がない点です。ステートオブジェクトの 1 つのコードを更新して、規則を変更したり、あるいはおそらくステートオブジェクトを追加する必要しかないのです。

まずは、より伝統的なオブジェクト指向の手法でステートパターンを実装し、その後、Rust としてもう少し自然なアプローチを使用します。それでは、ステートパターンを利用したブログ記事のワークフローの漸進的な実装に取り組んでいきましょう。

最終的な機能は以下のような感じになるでしょう:

1. ブログ記事は、空の草稿から始まる。
2. 草稿ができたら、査読が要求される。
3. 記事が承認されたら、公開される。
4. 公開されたブログ記事だけが表示する内容を返すので、未承認の記事は、誤って公開されない。

それ以外の記事に対する変更は、効果を持つべきではありません。例えば、査読を要求する前にブログ記事の草稿を承認しようとしたら、記事は、非公開の草稿のままになるべきです。

リスト 17-11 は、このワークフローをコードの形で示しています: これは、`blog` というライブラリ

クレートに実装する API の使用例です。blog クレートを実装していないので、コンパイルはできません。

ファイル名: src/main.rs

```
use blog::Post;

fn main() {
    let mut post = Post::new();

    //            "今日はお昼にサラダを食べた"
    post.add_text("I ate a salad for lunch today");
    assert_eq!("", post.content());

    post.request_review();
    assert_eq!("", post.content());

    post.approve();
    assert_eq!("I ate a salad for lunch today", post.content());
}
```

リスト 17-11: blog クレートに欲しい振る舞いをデモするコード

ユーザが Post::new で新しいブログ記事の草稿を作成できるようにしたいです。ブログ記事にテキストを追加できるようにしたいです。承認前に記事の内容を即座に得ようとしたら、記事はまだ草稿なので、何のテキストも得られるべきではありません。デモ目的でコードに assert_eq! を追加しました。これに対する素晴らしい単体テストは、ブログ記事の草稿が content メソッドから空の文字列を返すことをアサートすることでしょうが、この例に対してテストを書くつもりはありません。

次に、記事の査読を要求できるようにしたく、また査読を待機している間は content に空の文字列を返してほしいです。記事が承認を受けたら、公開されるべきです。つまり、content を呼んだ時に記事のテキストが返されるということです。

クレートから相互作用している唯一の型は、Post だけであることに注意してください。この型はステートパターンを使用し、記事がなり得る種々の状態を表す 3 つのステートオブジェクトのうちの 1 つになる値を保持します。草稿、査読待ち、公開中です。1 つの状態から別の状態への変更は、Post 型内部で管理されます。Post インスタンスのライブラリ使用者が呼び出すメソッドに呼応して状態は変化しますが、状態の変化を直接管理する必要はありません。また、ユーザは、査読前に記事を公開するなど状態を誤ることはありません。

17.3.1　Post を定義し、草稿状態で新しいインスタンスを生成する

ライブラリの実装に取り掛かりましょう！ なんらかの内容を保持する公開の Post 構造体が必要なことはわかるので、構造体の定義と、関連する公開の Post インスタンスを生成する new 関数から始めましょう。リスト 17-12 のようにですね。また、Post のすべての状態オブジェクトが持たなくては

ならない振る舞いを定義する、非公開の State トレイトも作成します。

それから Post は、状態オブジェクトを保持するための state という非公開のフィールドに、Option<T> で Box<dyn State> のトレイトオブジェクトを保持します。Option<T> が必要な理由はすぐわかります。

ファイル名: src/lib.rs

```
pub struct Post {
    state: Option<Box<dyn State>>,
    content: String,
}

impl Post {
    pub fn new() -> Post {
        Post {
            state: Some(Box::new(Draft {})),
            content: String::new(),
        }
    }
}

trait State {}

struct Draft {}

impl State for Draft {}
```

リスト 17-12: Post 構造体、新規 Post インスタンスを生成する new 関数、State トレイト、Draft 構造体の定義

State トレイトは、異なる記事の状態で共有される振る舞いを定義します。状態オブジェクトは Draft 、PendingReview 、Published で、これらはすべて State トレイトを実装します。今は、トレイトにメソッドは何もなく、Draft が記事の初期状態にしたい状態なので、その状態だけを定義することから始めます。

新しい Post を作る時、state フィールドは、Box を保持する Some 値にセットします。この Box が Draft 構造体の新しいインスタンスを指します。これにより、新しい Post を作る度に、草稿から始まることが保証されます。Post の state フィールドは非公開なので、Post を他の状態で作成する方法はないのです！ Post::new 関数では、content フィールドを新しい空の String にセットしています。

17.3.2　記事の内容のテキストを格納する

リスト 17-11 で、add_text というメソッドを呼び出し、ブログ記事のテキスト内容として追加される &str を渡せるようになりたいことを確認しました。後ほど content フィールドデータの読まれ方を制御するメソッドを実装できるように、content フィールドを pub にして晒すのではなく、これ

をメソッドとして実装しています。add_text メソッドは非常に素直なので、リスト 17-13 の実装を impl Post ブロックに追加しましょう:

ファイル名: src/lib.rs

```
# pub struct Post {
#     state: Option<Box<dyn State>>,
#     content: String,
# }
#
impl Post {
    // --snip--
#     pub fn new() -> Post {
#         Post {
#             state: Some(Box::new(Draft {})),
#             content: String::new(),
#         }
#     }
#
    pub fn add_text(&mut self, text: &str) {
        self.content.push_str(text);
    }
}
#
# trait State {}
#
# struct Draft {}
#
# impl State for Draft {}
```

リスト 17-13: 記事の content にテキストを追加する add_text メソッドを実装する

add_text メソッドは、self への可変参照を取ります。というのも、add_text を呼び出した Post インスタンスを変更しているからです。それから content の String に対して push_str を呼び出し、text 引数を渡して保存された content に追加しています。この振る舞いは、記事の状態によらないので、ステートパターンの一部ではありません。add_text メソッドは、state フィールドと全く相互作用しませんが、サポートしたい振る舞いの一部ではあります。

17.3.3　草稿の記事の内容は空であることを保証する

add_text を呼び出して記事に内容を追加した後でさえ、記事はまだ草稿状態なので、それでも content メソッドには空の文字列スライスを返してほしいです。リスト 17-11 の 7 行目で示したようにですね。とりあえず、この要求を実現する最も単純な方法で content メソッドを実装しましょう: 常に空の文字列スライスを返すことです。一旦、記事の状態を変更する能力を実装したら、公開できるように、これを後ほど変更します。ここまで、記事は草稿状態にしかなり得ないので、記事の内容は常に空のはずです。リスト 17-14 は、この仮の実装を表示しています:

ファイル名: src/lib.rs

```
# pub struct Post {
#     state: Option<Box<dyn State>>,
#     content: String,
# }
#
impl Post {
    // --snip--
#     pub fn new() -> Post {
#         Post {
#             state: Some(Box::new(Draft {})),
#             content: String::new(),
#         }
#     }
#
#     pub fn add_text(&mut self, text: &str) {
#         self.content.push_str(text);
#     }
#
    pub fn content(&self) -> &str {
        ""
    }
}
#
# trait State {}
#
# struct Draft {}
#
# impl State for Draft {}
```

リスト 17-14: `Post` に常に空の文字列スライスを返す `content` の仮の実装を追加する

この追加された `content` メソッドとともに、リスト 17-11 の 7 行目までのコードは、想定通り動きます。

17.3.4　記事の査読を要求すると、状態が変化する

次に、記事の査読を要求する機能を追加する必要があり、これをすると、状態が `Draft` から `PendingReview` に変わるはずです。リスト 17-15 はこのコードを示しています:

ファイル名: src/lib.rs

```
# pub struct Post {
#     state: Option<Box<dyn State>>,
#     content: String,
# }
#
impl Post {
```

```
    // --snip--
#     pub fn new() -> Post {
#         Post {
#             state: Some(Box::new(Draft {})),
#             content: String::new(),
#         }
#     }
#
#     pub fn add_text(&mut self, text: &str) {
#         self.content.push_str(text);
#     }
#
#     pub fn content(&self) -> &str {
#         ""
#     }
#
    pub fn request_review(&mut self) {
        if let Some(s) = self.state.take() {
            self.state = Some(s.request_review())
        }
    }
}

trait State {
    fn request_review(self: Box<Self>) -> Box<dyn State>;
}

struct Draft {}

impl State for Draft {
    fn request_review(self: Box<Self>) -> Box<dyn State> {
        Box::new(PendingReview {})
    }
}

struct PendingReview {}

impl State for PendingReview {
    fn request_review(self: Box<Self>) -> Box<dyn State> {
        self
    }
}
```

リスト 17-15: `Post` と `State` トレイトに `request_review` メソッドを実装する

`Post` に `self` への可変参照を取る `request_review` という公開メソッドを与えます。それから、`Post` の現在の状態に対して内部の `request_review` メソッドを呼び出し、この 2 番目の `request_review` が現在の状態を消費し、新しい状態を返します。

`State` トレイトに `request_review` メソッドを追加します; このトレイトを実装する型は全て、これで `request_review` メソッドを実装する必要があります。メソッドの第 1 引数に `self` 、`&self` 、

`&mut self` ではなく、`self: Box<Self>` としていることに注意してください。この記法は、型を保持する `Box` に対して呼ばれた時のみ、このメソッドが合法になることを意味しています。この記法は、`Box<Self>` の所有権を奪い、古い状態を無効化するので、`Post` の状態値は、新しい状態に変形できます。

古い状態を消費するために、`request_review` メソッドは、状態値の所有権を奪う必要があります。ここで `Post` の `state` フィールドの `Option` が問題になるのです: `take` メソッドを呼び出して、`state` フィールドから `Some` 値を取り出し、その箇所に `None` を残します。なぜなら、Rust は、構造体に未代入のフィールドを持たせてくれないからです。これにより、借用するのではなく、`Post` の `state` 値をムーブすることができます。それから、記事の `state` 値をこの処理の結果にセットするのです。

`self.state = self.state.request_review();` のようなコードで直接 `state` 値の所有権を得るよう設定するのではなく、一時的に `None` に `state` をセットする必要があります。これにより、新しい状態に変形した後に、`Post` が古い `state` 値を使えないことが保証されるのです。

`Draft` の `request_review` メソッドは、新しい `PendingReview` 構造体の新しいボックスのインスタンスを返し、これが、記事が査読待ちの時の状態を表します。`PendingReview` 構造体も `request_review` メソッドを実装しますが、何も変形はしません。むしろ、自身を返します。というのも、既に `PendingReview` 状態にある記事の査読を要求したら、`PendingReview` 状態に留まるべきだからです。

ようやくステートパターンの利点が見えてき始めました: `state` 値が何であれ、`Post` の `request_review` メソッドは同じです。各状態は、独自の規則にのみ責任を持ちます。

`Post` の `content` メソッドを空の文字列スライスを返してそのままにします。これで `Post` は `PendingReview` と `Draft` 状態になり得ますが、`PendingReview` 状態でも、同じ振る舞いが欲しいです。もうリスト 17-11 は 10 行目まで動くようになりました！

17.3.5　content の振る舞いを変化させる approve を追加する

`approve` メソッドは、`request_review` メソッドと類似するでしょう: 状態が承認された時に、現在の状態があるべきと言う値に `state` をセットします。リスト 17-16 のようにですね:

ファイル名: src/lib.rs

```
# pub struct Post {
#     state: Option<Box<dyn State>>,
#     content: String,
# }
#
impl Post {
    // --snip--
#     pub fn new() -> Post {
#         Post {
#             state: Some(Box::new(Draft {})),
#             content: String::new(),
#         }
#     }
```

```
#
#     pub fn add_text(&mut self, text: &str) {
#         self.content.push_str(text);
#     }
#
#     pub fn content(&self) -> &str {
#         ""
#     }
#
#     pub fn request_review(&mut self) {
#         if let Some(s) = self.state.take() {
#             self.state = Some(s.request_review())
#         }
#     }
#
    pub fn approve(&mut self) {
        if let Some(s) = self.state.take() {
            self.state = Some(s.approve())
        }
    }
}

trait State {
    fn request_review(self: Box<Self>) -> Box<dyn State>;
    fn approve(self: Box<Self>) -> Box<dyn State>;
}

struct Draft {}

impl State for Draft {
    // --snip--
#     fn request_review(self: Box<Self>) -> Box<dyn State> {
#         Box::new(PendingReview {})
#     }
#
    fn approve(self: Box<Self>) -> Box<dyn State> {
        self
    }
}

struct PendingReview {}

impl State for PendingReview {
    // --snip--
#     fn request_review(self: Box<Self>) -> Box<dyn State> {
#         self
#     }
#
    fn approve(self: Box<Self>) -> Box<dyn State> {
        Box::new(Published {})
    }
}
```

```
struct Published {}

impl State for Published {
    fn request_review(self: Box<Self>) -> Box<dyn State> {
        self
    }

    fn approve(self: Box<Self>) -> Box<dyn State> {
        self
    }
}
```

リスト 17-16: Post と State トレイトに approve メソッドを実装する

State トレイトに approve メソッドを追加し、Published 状態という State を実装する新しい構造体を追加します。

PendingReview に対して request_review が行っているのと同様に、Draft に対して approve メソッドを呼び出しても、approve は self を返すので、何も効果はありません。PendingReview に対して approve を呼び出すと、Published 構造体の新しいボックス化されたインスタンスを返します。Published 構造体は State トレイトを実装し、request_review メソッドと approve メソッド両方に対して、自身を返します。そのような場合に記事は、Published 状態に留まるべきだからです。

さて、Post の content メソッドを更新する必要が出てきました。content から返される値を Post の現在の状態に依存するようにしたいので、Post は、その state に定義された content メソッドに委譲するようにします。リスト 17-17 のようにですね:

ファイル名: src/lib.rs

```
# pub struct Post {
#     state: Option<Box<dyn State>>,
#     content: String,
# }
#
impl Post {
    // --snip--
#     pub fn new() -> Post {
#         Post {
#             state: Some(Box::new(Draft {})),
#             content: String::new(),
#         }
#     }
#
#     pub fn add_text(&mut self, text: &str) {
#         self.content.push_str(text);
#     }
#
    pub fn content(&self) -> &str {
        self.state.as_ref().unwrap().content(self)
```

```
    }
    // --snip--
#
#     pub fn request_review(&mut self) {
#         if let Some(s) = self.state.take() {
#             self.state = Some(s.request_review())
#         }
#     }
#
#     pub fn approve(&mut self) {
#         if let Some(s) = self.state.take() {
#             self.state = Some(s.approve())
#         }
#     }
# }
#
# trait State {
#     fn request_review(self: Box<Self>) -> Box<dyn State>;
#     fn approve(self: Box<Self>) -> Box<dyn State>;
# }
#
# struct Draft {}
#
# impl State for Draft {
#     fn request_review(self: Box<Self>) -> Box<dyn State> {
#         Box::new(PendingReview {})
#     }
#
#     fn approve(self: Box<Self>) -> Box<dyn State> {
#         self
#     }
# }
#
# struct PendingReview {}
#
# impl State for PendingReview {
#     fn request_review(self: Box<Self>) -> Box<dyn State> {
#         self
#     }
#
#     fn approve(self: Box<Self>) -> Box<dyn State> {
#         Box::new(Published {})
#     }
# }
#
# struct Published {}
#
# impl State for Published {
#     fn request_review(self: Box<Self>) -> Box<dyn State> {
#         self
#     }
#
#     fn approve(self: Box<Self>) -> Box<dyn State> {
```

```
#         self
#     }
# }
```

リスト 17-17: Post の content メソッドを更新して State の content メソッドに委譲する

目的は、これらの規則全てを State を実装する構造体の内部に押し留めることなので、state の値に対して content メソッドを呼び出し、記事のインスタンス (要するに、self) を引数として渡します。そして、state 値の content メソッドを使用したことから返ってきた値を返します。

Option に対して as_ref メソッドを呼び出します。値の所有権ではなく、Option 内部の値への参照が欲しいからです。state は Option<Box<dyn State>> なので、as_ref を呼び出すと、Option<&Box<dyn State>> が返ってきます。as_ref を呼ばなければ、state を関数引数の借用した &self からムーブできないので、エラーになるでしょう。

さらに unwrap メソッドを呼び出し、これは絶対にパニックしないことがわかっています。何故なら、Post のメソッドが、それらのメソッドが完了した際に state は常に Some 値を含んでいることを保証するからです。これは、コンパイラには理解不能であるものの、None 値が絶対にあり得ないとわかる第 9 章の「コンパイラよりもプログラマがより情報を持っている場合」節で語った一例です。

この時点で、&Box<dyn State> に対して content を呼び出すと、参照外し型強制が & と Box に働くので、究極的に content メソッドが State トレイトを実装する型に対して呼び出されることになります。つまり、content を State トレイト定義に追加する必要があり、そこが現在の状態に応じてどの内容を返すべきかというロジックを配置する場所です。リスト 17-18 のようにですね:

ファイル名: src/lib.rs

```
# pub struct Post {
#     state: Option<Box<dyn State>>,
#     content: String,
# }
#
# impl Post {
#     pub fn new() -> Post {
#         Post {
#             state: Some(Box::new(Draft {})),
#             content: String::new(),
#         }
#     }
#
#     pub fn add_text(&mut self, text: &str) {
#         self.content.push_str(text);
#     }
#
#     pub fn content(&self) -> &str {
#         self.state.as_ref().unwrap().content(self)
#     }
#
```

```
#     pub fn request_review(&mut self) {
#         if let Some(s) = self.state.take() {
#             self.state = Some(s.request_review())
#         }
#     }
#
#     pub fn approve(&mut self) {
#         if let Some(s) = self.state.take() {
#             self.state = Some(s.approve())
#         }
#     }
# }
#
trait State {
    // --snip--
#     fn request_review(self: Box<Self>) -> Box<dyn State>;
#     fn approve(self: Box<Self>) -> Box<dyn State>;
#
    fn content<'a>(&self, post: &'a Post) -> &'a str {
        ""
    }
}

// --snip--
#
# struct Draft {}
#
# impl State for Draft {
#     fn request_review(self: Box<Self>) -> Box<dyn State> {
#         Box::new(PendingReview {})
#     }
#
#     fn approve(self: Box<Self>) -> Box<dyn State> {
#         self
#     }
# }
#
# struct PendingReview {}
#
# impl State for PendingReview {
#     fn request_review(self: Box<Self>) -> Box<dyn State> {
#         self
#     }
#
#     fn approve(self: Box<Self>) -> Box<dyn State> {
#         Box::new(Published {})
#     }
# }
#
struct Published {}

impl State for Published {
    // --snip--
```

```
#     fn request_review(self: Box<Self>) -> Box<dyn State> {
#         self
#     }
#
#     fn approve(self: Box<Self>) -> Box<dyn State> {
#         self
#     }
#
    fn content<'a>(&self, post: &'a Post) -> &'a str {
        &post.content
    }
}
```

リスト 17-18: State トレイトに content メソッドを追加する

空の文字列スライスを返すデフォルト実装を content メソッドに追加しています。これにより、Draft と PendingReview 構造体に content を実装する必要はありません。Published 構造体は、content メソッドをオーバーライドし、post.content の値を返します。

第 10 章で議論したように、このメソッドにはライフタイム注釈が必要なことに注意してください。post への参照を引数として取り、その post の一部への参照を返しているので、返却される参照のライフタイムは、post 引数のライフタイムに関連します。

出来上がりました。要するに、リスト 17-11 はもう動くようになったのです！ ブログ記事ワークフローの規則でステートパターンを実装しました。その規則に関連するロジックは、Post 中に散乱するのではなく、ステートオブジェクトに息づいています。

17.3.5.1　enum を使えばいいのでは？

可能な記事の状態を列挙子とする enum を使わなかったのはなぜか、疑問に感じているかもしれません。それももちろんあり得る解決策です。試してみて、どちらが好みか、最終結果を比較してみてください！ enum を使うことの不利な点のひとつは、enum の値をチェックするすべての場所で、match 式か、それに類似する、すべての可能な列挙子を処理するための方法が必要になるだろうということです。これは、トレイトオブジェクトを使用する解決策と比較して、より煩わしくなることがあります。

17.3.6　ステートパターンの代償

オブジェクト指向のステートパターンを実装して各状態の時に記事がなり得る異なる種類の振る舞いをカプセル化する能力が、Rust にあることを示してきました。Post のメソッドは、種々の振る舞いについては何も知りません。コードを体系化する仕方によれば、公開された記事が振る舞うことのある様々な方法を知るには、1 箇所のみを調べればいいのです: Published 構造体の State トレイトの実装です。

ステートパターンを使用しない対立的な実装を作ることになったら、代わりに Post のメソッドか、あるいは記事の状態を確認し、それらの箇所 (編注： Post のメソッドのことか) の振る舞いを変更する main コードでさえ、match 式を使用したかもしれません。そうなると、複数個所を調べて記事が公開状態にあることの裏の意味全てを理解しなければならなくなります！ これは、追加した状態が増えれば、さらに上がるだけでしょう: 各 match 式には、別のアームが必要になるのです。

ステートパターンでは、Post のメソッドと Post を使用する箇所で、match 式が必要になることはなく、新しい状態を追加するのにも、新しい構造体を追加し、その 1 つの構造体にトレイトメソッドを実装するだけでいいわけです。

ステートパターンを使用した実装は、拡張して機能を増やすことが容易です。ステートパターンを使用するコードの管理の単純さを確認するために、以下の提言を試してみてください:

- 記事の状態を PendingReview から Draft に戻す reject メソッドを追加する。
- 状態が Published に変化させられる前に approve を 2 回呼び出す必要があるようにする。
- 記事が Draft 状態の時のみテキスト内容をユーザが追加できるようにする。ヒント: ステートオブジェクトに内容について変わる可能性のあるものの責任を持たせつつも、Post を変更することには責任を持たせない。

ステートパターンの欠点の 1 つは、状態が状態間の遷移を実装しているので、状態の一部が密に結合した状態になってしまうことです。PendingReview と Published の間に、Scheduled のような別の状態を追加したら、代わりに PendingReview のコードを Scheduled に遷移するように変更しなければならないでしょう。状態が追加されても PendingReview を変更する必要がなければ、作業が減りますが、そうすれば別のデザインパターンに切り替えることになるでしょう。

別の欠点は、ロジックの一部を重複させてしまうことです。重複を除くためには、State トレイトの request_review と approve メソッドに self を返すデフォルト実装を試みる可能性があります; ですが、これはオブジェクト安全性を侵害するでしょう。というのも、具体的な self が一体なんなのかトレイトには知りようがないからです。State をトレイトオブジェクトとして使用できるようにしたいので、メソッドにはオブジェクト安全になってもらう必要があるのです。

他の重複には、Post の request_review と approve メソッドの実装が似ていることが含まれます。両メソッドは Option の state の値に対する同じメソッドの実装に委譲していて、state フィールドの新しい値を結果にセットします。このパターンに従う Post のメソッドが多くあれば、マクロを定義して繰り返しを排除することも考慮する可能性があります (第 19 章の「マクロ」節を参照)。

オブジェクト指向言語で定義されている通り忠実にステートパターンを実装することで、Rust の強みをできるだけ発揮していません。blog クレートに対して行える無効な状態と遷移をコンパイルエラーにできる変更に目を向けましょう。

17.3.6.1　状態と振る舞いを型としてコード化する

ステートパターンを再考して別の代償を得る方法をお見せします。状態と遷移を完全にカプセル化して、外部のコードに知らせないようにするよりも、状態を異なる型にコード化します。結果的に、

Rust の型検査システムが、公開記事のみが許可される箇所で草稿記事の使用を試みることをコンパイルエラーを発して阻止します。

リスト 17-11 の `main` の最初の部分を考えましょう:

ファイル名: src/main.rs

```
# use blog::Post;
#
fn main() {
    let mut post = Post::new();

    //            "今日はお昼にサラダを食べた"
    post.add_text("I ate a salad for lunch today");
    assert_eq!("", post.content());
#
#     post.request_review();
#     assert_eq!("", post.content());
#
#     post.approve();
#     assert_eq!("I ate a salad for lunch today", post.content());
}
```

それでも、`Post::new` で草稿状態の新しい記事を生成することと記事の内容にテキストを追加する能力は可能にします。しかし、空の文字列を返す草稿記事の `content` メソッドを保持する代わりに、草稿記事は、`content` メソッドを全く持たないようにします。そうすると、草稿記事の内容を得ようとしたら、メソッドが存在しないというコンパイルエラーになるでしょう。その結果、誤ってプロダクションコードで草稿記事の内容を表示することが不可能になります。そのようなコードは、コンパイルさえできないからです。リスト 17-19 は `Post` 構造体、`DraftPost` 構造体、さらにメソッドの定義を示しています:

ファイル名: src/lib.rs

```
pub struct Post {
    content: String,
}

pub struct DraftPost {
    content: String,
}

impl Post {
    pub fn new() -> DraftPost {
        DraftPost {
            content: String::new(),
        }
    }

    pub fn content(&self) -> &str {
```

```
        &self.content
    }
}

impl DraftPost {
    pub fn add_text(&mut self, text: &str) {
        self.content.push_str(text);
    }
}
```

リスト 17-19: content メソッドのある Post と content メソッドのない DraftPost

Post と DraftPost 構造体どちらにもブログ記事のテキストを格納する非公開の content フィールドがあります。状態のコード化を構造体の型に移動したので、この構造体は最早 state フィールドを持ちません。Post は公開された記事を表し、content を返す content メソッドがあります。

それでも Post::new 関数はありますが、Post のインスタンスを返すのではなく、DraftPost のインスタンスを返します。content は非公開であり、Post を返す関数も存在しないので、現状 Post のインスタンスを生成することは不可能です。

DraftPost 構造体には、以前のようにテキストを content に追加できるよう add_text メソッドがありますが、DraftPost には content メソッドが定義されていないことに注目してください！ 従って、これでプログラムは、全ての記事が草稿記事から始まり、草稿記事は表示できる内容がないことを保証します。この制限をかいくぐる試みは、全てコンパイルエラーに落ち着くでしょう。

17.3.6.2　遷移を異なる型への変形として実装する

では、どうやって公開された記事を得るのでしょうか？ 公開される前に草稿記事は査読され、承認されなければならないという規則を強制したいです。査読待ち状態の記事は、それでも内容を表示するべきではありません。別の構造体 PendingReviewPost を追加し、DraftPost に PendingReviewPost を返す request_review メソッドを定義し、PendingReviewPost に Post を返す approve メソッドを定義してこれらの制限を実装しましょう。リスト 17-20 のようにですね:

ファイル名: src/lib.rs

```
# pub struct Post {
#     content: String,
# }
#
# pub struct DraftPost {
#     content: String,
# }
#
# impl Post {
#     pub fn new() -> DraftPost {
#         DraftPost {
#             content: String::new(),
```

```
#         }
#     }
#
#     pub fn content(&self) -> &str {
#         &self.content
#     }
# }
#
impl DraftPost {
    // --snip--
#     pub fn add_text(&mut self, text: &str) {
#         self.content.push_str(text);
#     }
#
    pub fn request_review(self) -> PendingReviewPost {
        PendingReviewPost {
            content: self.content,
        }
    }
}

pub struct PendingReviewPost {
    content: String,
}

impl PendingReviewPost {
    pub fn approve(self) -> Post {
        Post {
            content: self.content,
        }
    }
}
```

リスト 17-20: DraftPost の request_review を呼び出すことで生成される PendingReviewPost と、PendingReviewPost を公開された Post に変換する approve メソッド

request_review と approve メソッドは self の所有権を奪い、故に DraftPost と PendingReviewPost インスタンスを消費し、それぞれ PendingReviewPost と公開された Post に変形します。このように、DraftPost インスタンスに request_review を呼んだ後には、DraftPost インスタンスは生きながらえず、以下同様です。PendingReviewPost 構造体には、content メソッドが定義されていないので、DraftPost 同様に、その内容を読もうとするとコンパイルエラーに落ち着きます。content メソッドが確かに定義された公開された Post インスタンスを得る唯一の方法が、PendingReviewPost に対して approve を呼び出すことであり、PendingReviewPost を得る唯一の方法が、DraftPost に request_review を呼び出すことなので、これでブログ記事のワークフローを型システムにコード化しました。

ですが、さらに main にも多少小さな変更を行わなければなりません。request_review と approve メソッドは、呼ばれた構造体を変更するのではなく、新しいインスタンスを返すので、let post = と

いうシャドーイング代入をもっと追加し、返却されたインスタンスを保存する必要があります。また、草稿と査読待ち記事の内容を空の文字列でアサートすることも、する必要もありません: 最早、その状態にある記事の内容を使用しようとするコードはコンパイル不可能だからです。main の更新されたコードは、リスト 17-21 に示されています:

ファイル名: src/main.rs

```
use blog::Post;

fn main() {
    let mut post = Post::new();

    post.add_text("I ate a salad for lunch today");

    let post = post.request_review();

    let post = post.approve();

    assert_eq!("I ate a salad for lunch today", post.content());
}
```

リスト 17-21: ブログ記事ワークフローの新しい実装を使う main の変更

post を再代入するために main に行う必要のあった変更は、この実装がもう、全くオブジェクト指向のステートパターンに沿っていないことを意味します: 状態間の変形は最早、Post 実装内に完全にカプセル化されていません。ですが、型システムとコンパイル時に起きる型チェックのおかげでもう無効な状態があり得なくなりました。これにより、未公開の記事の内容が表示されるなどの特定のバグが、プロダクションコードに移る前に発見されることが保証されます。

blog クレートに関してこの節の冒頭で提言される作業をそのままリスト 17-21 の後に試してみて、このバージョンのコードについてどう思うか確かめてください。この設計では、既に作業の一部が達成されている可能性があることに注意してください。

Rust は、オブジェクト指向のデザインパターンを実装する能力があるものの、状態を型システムにコード化するなどの他のパターンも、Rust では利用可能なことを確かめました。これらのパターンには、異なる代償があります。あなたが、オブジェクト指向のパターンには非常に馴染み深い可能性があるものの、問題を再考して Rust の機能の強みを活かすと、コンパイル時に一部のバグを回避できるなどの利益が得られることもあります。オブジェクト指向のパターンは、オブジェクト指向言語にはない所有権などの特定の機能により Rust では、必ずしも最善の解決策ではないでしょう。

17.4　まとめ

この章読了後に、あなたが Rust はオブジェクト指向言語であると考えるかどうかに関わらず、もうトレイトオブジェクトを使用して Rust でオブジェクト指向の機能の一部を得ることができると

知っています。ダイナミックディスパッチは、多少の実行時性能と引き換えにコードに柔軟性を齎 (もたら) してくれます。この柔軟性を利用してコードのメンテナンス性に寄与するオブジェクト指向パターンを実装することができます。Rust にはまた、オブジェクト指向言語にはない所有権などの他の機能もあります。オブジェクト指向パターンは、必ずしも Rust の強みを活かす最善の方法にはなりませんが、利用可能な選択肢の 1 つではあります。

次は、パターンを見ます。パターンも多くの柔軟性を可能にする Rust の別の機能です。本全体を通して僅かに見かけましたが、まだその全能力は目の当たりにしていません。さあ、行きましょう！

第 18 章

パターンとマッチング

パターンは、複雑であれ、単純であれ、Rust で型の構造に一致する特別な記法です。match 式や他の構文と組み合わせてパターンを使用すると、プログラムの制御フローをよりコントロールできます。パターンは、以下を組み合わせることで構成されます:

- リテラル
- 分配された配列、enum、構造体、タプル
- 変数
- ワイルドカード
- プレースホルダー

これらの要素が取り組んでいるデータの形を説明し、それから値に対してマッチを行い、プログラムに正しい値があって特定のコードを実行し続けられるかどうかを決定します。

パターンを使用するには、なんらかの値と比較します。パターンが値に合致したら、コードで値の部分を使用します。コイン並び替えマシンの例のような第 6 章でパターンを使用した match 式を思い出してください。値がパターンの形に当てはまったら、名前のある部品を使用できます。当てはまらなければ、パターンに紐づいたコードは実行されません。

この章は、パターンに関連するあらゆるものの参考文献です。パターンを使用するのが合法な箇所、論駁 (ろんばく) 可能と論駁不可能なパターンの違い、目撃する可能性のある色々な種類のパターン記法を講義します。章の終わりまでに、パターンを使用して多くの概念をはっきり表現する方法を知るでしょう。

18.1 パターンが使用されることのある箇所全部

Rust において、パターンはいろんな箇所に出現し、そうと気づかないうちにたくさん使用してきました！ この節は、パターンが合法な箇所全部を議論します。

18.1.1　match アーム

第 6 章で議論したように、パターンを match 式のアームで使います。正式には、match 式はキーワード match 、マッチ対象の値、パターンとそのアームのパターンに値が合致したら実行される式からなる 1 つ以上のマッチアームとして定義されます。以下のように:

```
match VALUE {
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
}
```

match 式の必須事項の 1 つは、match 式の値の可能性全てが考慮されなければならないという意味で**網羅的**である必要があることです。全可能性をカバーしていると保証する 1 つの手段は、最後のアームに包括的なパターンを入れることです: 例えば、どんな値にも合致する変数名は失敗することがあり得ないので、故に残りの全ケースをカバーできます。

_ という特定のパターンは何にでもマッチしますが、変数には束縛されないので、よく最後のマッチアームに使用されます。例えば、_ パターンは、指定されていないあらゆる値を無視したい時に有用です。_ パターンについて詳しくは、この章の後ほど、「パターンで値を無視する」節で講義します。

18.1.2　条件分岐 if let 式

第 6 章で主に if let 式を 1 つの場合にしか合致しない match と同様のものを書く省略法として使用する方法を議論しました。オプションとして、if let には if let のパターンが合致しない時に走るコードを含む対応する else も用意できます。

リスト 18-1 は、if let 、else if 、else if let 式を混ぜてマッチさせることもできることを示しています。そうすると、パターンと 1 つの値しか比較することを表現できない match 式よりも柔軟性が高くなります。また、一連の if let 、else if 、else if let アームの条件は、お互いに関連している必要はありません。

リスト 18-1 のコードは、背景色が何になるべきかを決定するいくつかの条件を連なって確認するところを示しています。この例では、実際のプログラムではユーザ入力を受け付ける可能性のある変数をハードコードされた値で生成しています。

ファイル名: src/main.rs

```
fn main() {
    let favorite_color: Option<&str> = None;
    let is_tuesday = false;
    let age: Result<u8, _> = "34".parse();

    if let Some(color) = favorite_color {
        // あなたのお気に入りの色、{}を背景色に使用します
```

```
        println!("Using your favorite color, {}, as the background", color);
    } else if is_tuesday {
        // 火曜日は緑の日！
        println!("Tuesday is green day!");
    } else if let Ok(age) = age {
        if age > 30 {
            // 紫を背景色に使用します
            println!("Using purple as the background color");
        } else {
            // オレンジを背景色に使用します
            println!("Using orange as the background color");
        }
    } else {
        // 青を背景色に使用します
        println!("Using blue as the background color");
    }
}
```

リスト 18-1: `if let`、`else if`、`else if let`、`else` を混ぜる

ユーザがお気に入りの色を指定したら、その色が背景色になります。今日が火曜日なら、背景色は緑です。ユーザが年齢を文字列で指定し、数値として解析することができたら、背景色は、その数値の値によって紫かオレンジになります。どの条件も適用できなければ、背景色は青になります:

この条件分岐構造により、複雑な要件をサポートさせてくれます。ここにあるハードコードされた値では、この例は `Using purple as the background color` と出力するでしょう。

`match` アームのように `if let` もシャドーイングされた変数を導入できることがわかります: `if let Ok(age) = age` の行は、`Ok` 列挙子の中の値を含むシャドーイングされた新しい `age` 変数を導入します。つまり、`if age > 30` という条件は、そのブロック内に配置する必要があります: これら 2 つの条件を組み合わせて、`if let Ok(age) = age && age > 30` とすることはできません。30 と比較したいシャドーイングされた `age` は、波括弧で新しいスコープが始まるまで有効にならないのです。

`if let` 式を使うことの欠点は、コンパイラが網羅性を確認してくれないことです。一方で `match` 式ではしてくれます。最後の `else` ブロックを省略して故に、扱い忘れたケースがあっても、コンパイラは、ロジックバグの可能性を指摘してくれないでしょう。

18.1.3　`while let` 条件分岐ループ

`if let` と構成が似て、`while let` 条件分岐ループは、パターンが合致し続ける限り、`while` ループを走らせます。リスト 18-2 の例は、ベクタをスタックとして使用する `while let` ループを示し、ベクタの値をプッシュしたのとは逆順に出力します:

```
let mut stack = Vec::new();

stack.push(1);
stack.push(2);
```

```
stack.push(3);

while let Some(top) = stack.pop() {
    println!("{}", top);
}
```

リスト 18-2: `while let` ループを使って `stack.pop()` が `Some` を返す限り値を出力する

この例は、3, 2, そして 1 と出力します。pop メソッドはベクタの最後の要素を取り出して `Some(value)` を返します。ベクタが空なら、pop は `None` を返します。`while` ループは pop が `Some` を返す限り、ブロックのコードを実行し続けます。pop が `None` を返すと、ループは停止します。`while let` を使用してスタックから全ての要素を取り出せるのです。

18.1.4　for ループ

第 3 章で、Rust コードにおいては、`for` ループが最もありふれたループ構造だと述べましたが、`for` が取るパターンについてはまだ議論していませんでした。`for` ループにおいて、直接キーワード `for` に続く値がパターンなので、`for x in y` では、`x` がパターンになります。

リスト 18-3 は `for` ループでパターンを使用して `for` ループの一部としてタプルを分配あるいは、分解する方法をデモしています。

```
let v = vec!['a', 'b', 'c'];

for (index, value) in v.iter().enumerate() {
    println!("{} is at index {}", value, index);
}
```

リスト 18-3: `for` ループでパターンを使用してタプルを分配する

リスト 18-3 のコードは、以下のように出力するでしょう:

```
a is at index 0
b is at index 1
c is at index 2
```

`enumerate` メソッドを使用してイテレータを改造し、値とその値のイテレータでの添え字をタプルに配置して生成しています。`enumerate` の最初の呼び出しは、タプル `(0, 'a')` を生成します。この値がパターン `(index, value)` とマッチさせられると、`index` は `0`、`value` は `'a'` になり、出力の最初の行を出力するのです。

18.1.5　let 文

この章に先駆けて、match と if let でパターンを使用することだけ明示的に議論してきましたが、実は let 文を含む他の箇所でもパターンを使用してきたのです。例として、この let での率直な変数代入を考えてください:

```
let x = 5;
```

この本を通してこのような let を何百回も使用してきて、お気付きではなかったかもしれませんが、パターンを使用していたのです！ より正式には、let 文はこんな見た目をしています:

```
let PATTERN = EXPRESSION;
```

let x = 5; のような変数名が PATTERN スロットにある文で、変数名は、ただ特に単純な形態のパターンなのです。Rust は式をパターンと比較し、見つかったあらゆる名前を代入します。故に、let x = 5; の例では、x は「ここでマッチしたものを変数 x に束縛する」ことを意味するパターンです。名前 x がパターンの全容なので、このパターンは実質的に「値が何であれ、全てを変数 x に束縛しろ」を意味します。

let のパターンマッチングの観点をよりはっきり確認するためにリスト 18-4 を考えてください。これは let でパターンを使用し、タプルを分配します。

```
let (x, y, z) = (1, 2, 3);
```

リスト 18-4: パターンを使用してタプルを分配し、3 つの変数を一度に生成する

ここでタプルに対してパターンをマッチさせています。Rust は値 (1, 2, 3) をパターン (x, y, z) と比較し、値がパターンに合致すると確認するので、1 を x に、2 を y に、3 を z に束縛します。このタプルパターンを個別の 3 つの変数パターンが内部にネストされていると考えることもできます。

パターンの要素数がタプルの要素数と一致しない場合、全体の型が一致せず、コンパイルエラーになるでしょう。例えば、リスト 18-5 は、3 要素のタプルを 2 つの変数に分配しようとしているところを表示していて、動きません。

```
let (x, y) = (1, 2, 3);
```

リスト 18-5: 変数がタプルの要素数と一致しないパターンを間違って構成する

このコードのコンパイルを試みると、このような型エラーに落ち着きます:

```
error[E0308]: mismatched types
 --> src/main.rs:2:9
  |
```

```
2 |     let (x, y) = (1, 2, 3);
  |         ^^^^^^ expected a tuple with 3 elements, found one with 2 elements
  |                (3要素のタプルを予期したのに、2要素のタプルが見つかりました)
  |
  = note: expected type `({integer}, {integer}, {integer})`
             found type `(_, _)`
```

タプルの値のうち 1 つ以上を無視したかったら、「パターンで値を無視する」節で見かけるように、_ か .. を使用できるでしょう。パターンに変数が多すぎるというのが問題なら、変数の数がタプルの要素数と一致するように変数を減らすことで、型を一致させることが解決策です。

18.1.6　関数の引数

関数の引数もパターンにできます。リスト 18-6 のコードは、型 i32 の x という引数 1 つを取る foo という関数を宣言していますが、これまでに馴染み深くなっているはずです。

```
fn foo(x: i32) {
    // コードがここに来る
    // code goes here
}
```

リスト 18-6: 関数シグニチャが引数にパターンを使用している

x の部分がパターンです！ let のように、関数の引数でパターンにタプルを合致させられるでしょう。リスト 18-7 では、タプルを関数に渡したのでその中の値を分離しています。

ファイル名: src/main.rs

```
fn print_coordinates(&(x, y): &(i32, i32)) {
    // 現在の位置: ({}, {})
    println!("Current location: ({}, {})", x, y);
}

fn main() {
    let point = (3, 5);
    print_coordinates(&point);
}
```

リスト 18-7: タプルを分配する引数を伴う関数

このコードは Current location: (3, 5) と出力します。値 &(3, 5) はパターン &(x, y) と合致するので、x は値 3 、y は値 5 になります。

また、クロージャの引数リストでも、関数の引数リストのようにパターンを使用することができます。第 13 章で議論したように、クロージャは関数に似ているからです。

この時点で、パターンを使用する方法をいくつか見てきましたが、パターンを使用できる箇所全部

で同じ動作をするわけではありません。パターンが論駁不可能でなければならない箇所もあります。他の状況では、論駁可能にもなり得ます。この 2 つの概念を次に議論します。

18.2　論駁可能性: パターンが合致しないかどうか

パターンには 2 つの形態があります: 論駁可能なものと論駁不可能なものです。渡される可能性のあるあらゆる値に合致するパターンは、**論駁不可能**なものです。文 `let x = 5;` の `x` は一例でしょう。`x` は何にでも合致し、故に合致に失敗することがあり得ないからです。なんらかの可能性のある値に対して合致しないことがあるパターンは、**論駁可能**なものです。一例は、式 `if let Some(x) = a_value` の `Some(x)` になるでしょう; `a_value` 変数の値が `Some` ではなく、`None` なら、`Some(x)` パターンは合致しないでしょうから。

関数の引数、`let` 文、`for` ループは、値が合致しなかったら何も意味のあることをプログラムが実行できないので、論駁不可能なパターンしか受け付けられません。`if let` と `while let` 式は、定義により失敗する可能性を処理することを意図したものなので、論駁可能なパターンのみを受け付けます: 条件式の機能は、成功か失敗によって異なる振る舞いをする能力にあるのです。

一般的に、論駁可能と論駁不可能なパターンの差異について心配しなくてもいいはずです; しかしながら、エラーメッセージで見かけた際に対応できるように、論駁可能性の概念に確かに慣れておく必要があります。そのような場合には、コードの意図した振る舞いに応じて、パターンかパターンを使用している構文を変える必要があるでしょう。

コンパイラが論駁不可能なパターンを必要とする箇所で論駁可能なパターンを使用しようとしたら、何が起きるかとその逆の例を見ましょう。リスト 18-8 は `let` 文を示していますが、パターンには `Some(x)` と指定し、論駁可能なパターンです。ご想像通りかもしれませんが、このコードはコンパイルできません。

```
let Some(x) = some_option_value;
```

リスト 18-8: `let` で論駁可能なパターンを使用しようとする

`some_option_value` が `None` 値だったなら、パターン `Some(x)` に合致しないことになり、パターンが論駁可能であることを意味します。ですが、`let` 文は論駁不可能なパターンしか受け付けられません。`None` 値に対してコードができる合法なことは何もないからです。コンパイル時にコンパイラは、論駁不可能なパターンが必要な箇所に論駁可能なパターンを使用しようとしたと文句を言うでしょう:

```
error[E0005]: refutable pattern in local binding: `None` not covered
(エラー: ローカル束縛に論駁可能なパターン: `None`がカバーされていません)
 -->
  |
3 | let Some(x) = some_option_value;
  |     ^^^^^^^ pattern `None` not covered
```

パターン `Some(x)` で全ての合法な値をカバーしなかった (できませんでした！) ので、コンパイラは当然、コンパイルエラーを生成します。

論駁不可能なパターンが必要な箇所に論駁可能なパターンがある問題を修正するには、パターンを使用するコードを変えればいいのです: `let` の代わりに `if let` を使用できます。そして、パターンが合致しなかったら、コードは合法に継続する手段を残して、波括弧内のコードを飛ばすだけでしょう。リスト 18-9 は、リスト 18-8 のコードの修正方法を示しています。

```
# let some_option_value: Option<i32> = None;
if let Some(x) = some_option_value {
    println!("{}", x);
}
```

リスト 18-9: `let` ではなく、`if let` と論駁可能なパターンを含むブロックを使用する

コードに逃げ道を与えました！ このコードは完全に合法ですが、エラーを受け取らないで論駁不可能なパターンを使用することはできないことを意味します。リスト 18-10 のように、`x` のような常にマッチするパターンを `if let` に与えたら、コンパイルできないでしょう。

```
if let x = 5 {
    println!("{}", x);
};
```

リスト 18-10: `if let` で論駁不可能なパターンを使用してみる

コンパイラは、論駁不可能なパターンと `if let` を使用するなんて道理が通らないと文句を言います:

```
error[E0162]: irrefutable if-let pattern
(エラー: 論駁不可能なif-letパターン)
 --> <anon>:2:8
  |
2 | if let x = 5 {
  |        ^ irrefutable pattern
```

このため、マッチアームは、論駁不可能なパターンで残りのあらゆる値に合致すべき最後のアームを除いて、論駁可能なパターンを使用しなければなりません。コンパイラは、たった 1 つしかアームのない `match` で論駁不可能なパターンを使用させてくれますが、この記法は特別有用なわけではなく、より単純な `let` 文に置き換えることもできるでしょう。

今やパターンを使用すべき箇所と論駁可能と論駁不可能なパターンの違いを知ったので、パターンを生成するために使用できる全ての記法を講義しましょう。

18.3　パターン記法

本全体で、多くの種類のパターンの例を見かけてきました。この節では、パターンで合法な記法全てを集め、それぞれを使用したくなる可能性がある理由について議論します。

18.3.1　リテラルにマッチする

第 6 章で目撃したように、パターンを直接リテラルに合致させられます。以下のコードが例を挙げています:

```
let x = 1;

match x {
    1 => println!("one"),       // 1
    2 => println!("two"),       // 2
    3 => println!("three"),     // 3
    _ => println!("anything"),  // なんでも
}
```

このコードは、x の値が 1 なので、one を出力します。この記法は、コードが特定の具体的な値を得た時に行動を起こしてほしい時に有用です。

18.3.2　名前付き変数にマッチする

名前付き変数はどんな値にも合致する論駁不可能なパターンであり、この本の中で何度も使用してきました。ですが、名前付き変数を match 式で使うと、厄介な問題があります。match は新しいスコープを開始するので、match 式内のパターンの一部として宣言された変数は、あらゆる変数同様に match 構文外部の同じ名前の変数を覆い隠します。リスト 18-11 で、値 Some(5) の x という変数と値 10 の変数 y を宣言しています。それから値 x に対して match 式を生成します。マッチアームのパターンと最後の println! を見て、このコードを実行したり、先まで読み進める前にこのコードが何を出力するか推測してみてください。

ファイル名: src/main.rs

```
fn main() {
    let x = Some(5);
    let y = 10;

    match x {
        // 50だったよ
        Some(50) => println!("Got 50"),
        // マッチしたよ
        Some(y) => println!("Matched, y = {:?}", y),
```

```
        // 既定のケース
        _ => println!("Default case, x = {:?}", x),
    }

    // 最後にはx = {}, y = {}
    println!("at the end: x = {:?}, y = {:?}", x, y);
}
```

リスト 18-11: シャドーイングされた変数 y を導入するアームのある match 式

match 式を実行した時に起こることを見ていきましょう。最初のマッチアームのパターンは、x の定義された値に合致しないので、コードは継続します。

2 番目のマッチアームのパターンは、Some 値内部のあらゆる値に合致する新しい y という変数を導入します。match 式内の新しいスコープ内にいるので、これは新しい y 変数であり、最初に値 10 で宣言した y ではありません。この新しい y 束縛は、Some 内のあらゆる値に合致し、x にあるものはこれです。故に、この新しい y は、x の中身の値に束縛されます。その値は 5 なので、そのアームの式が実行され、Matched, y = 5 と出力されます。

x が Some(5) ではなく None 値だったなら、最初の 2 つのアームのパターンはマッチしなかったので、値はアンダースコアに合致したでしょう。アンダースコアのアームのパターンでは x 変数を導入しなかったので、その式の x は、まだシャドーイングされない外側の x のままです。この架空の場合、match は Default case, x = None と出力するでしょう。

match 式が完了すると、スコープが終わるので、中の y のスコープも終わります。最後の println! は at the end: x = Some(5), y = 10 を生成します。

シャドーイングされた変数を導入するのではなく、外側の x と y の値を比較する match 式を生成するには、代わりにマッチガード条件式を使用する必要があるでしょう。マッチガードについては、後ほど、「マッチガードで追加の条件式」節で語ります。

18.3.3　複数のパターン

match 式で | 記法で複数のパターンに合致させることができ、これは **or** を意味します。例えば、以下のコードは x の値をマッチアームに合致させ、最初のマッチアームには **or** 選択肢があり、x の値がそのアームのどちらかの値に合致したら、そのアームのコードが走ることを意味します:

```
let x = 1;

match x {
    // 1か2
    1 | 2 => println!("one or two"),
    // 3
    3 => println!("three"),
    // なんでも
    _ => println!("anything"),
}
```

このコードは、one or two を出力します。

18.3.4　..=で値の範囲に合致させる

..= 記法により、限度値を含む値の範囲にマッチさせることができます。以下のコードでは、パターンが範囲内のどれかの値に合致すると、そのアームが実行されます:

```
let x = 5;

match x {
    // 1から5まで
    1..=5 => println!("one through five"),
    // それ以外
    _ => println!("something else"),
}
```

x が 1、2、3、4 か 5 なら、最初のアームが合致します。この記法は、| 演算子を使用して同じ考えを表現するより便利です; 1..=5 ではなく、| を使用したら、1 | 2 | 3 | 4 | 5 と指定しなければならないでしょう。範囲を指定する方が遥かに短いのです。特に 1 から 1000 までの値と合致させたいとかなら！

範囲は、数値か char 値でのみ許可されます。コンパイラがコンパイル時に範囲が空でないことを確認しているからです。範囲が空かそうでないかコンパイラにわかる唯一の型が char か数値なのです。

こちらは、char 値の範囲を使用する例です:

```
let x = 'c';

match x {
    // ASCII文字前半
    'a'..='j' => println!("early ASCII letter"),
    // ASCII文字後半
    'k'..='z' => println!("late ASCII letter"),
    // それ以外
    _ => println!("something else"),
}
```

コンパイラには 'c' が最初のパターンの範囲にあることがわかり、early ASCII letter と出力されます。

18.3.5　分配して値を分解する

また、パターンを使用して構造体、enum、タプル、参照を分配し、これらの値の異なる部分を使用することもできます。各値を見ていきましょう。

18.3.5.1　構造体を分配する

リスト 18-12 は、let 文でパターンを使用して分解できる 2 つのフィールド x と y のある Point 構造体を示しています。

ファイル名: src/main.rs

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p = Point { x: 0, y: 7 };

    let Point { x: a, y: b } = p;
    assert_eq!(0, a);
    assert_eq!(7, b);
}
```

リスト 18-12: 構造体のフィールドを個別の変数に分配する

このコードは、p 変数の x と y フィールドの値に合致する変数 a と b を生成します。この例は、パターンの変数の名前は、構造体のフィールド名と合致する必要はないことを示しています。しかし、変数名をフィールド名と一致させてどの変数がどのフィールド由来のものなのか覚えやすくしたくなることは一般的なことです。

変数名をフィールドに一致させることは一般的であり、let Point{ x: x, y: y } = p; と書くことは多くの重複を含むので、構造体のフィールドと一致するパターンには省略法があります: 構造体のフィールドの名前を列挙するだけで、パターンから生成される変数は同じ名前になるのです。リスト 18-13 は、リスト 18-12 と同じ振る舞いをするコードを表示していますが、let パターンで生成される変数は a と b ではなく、x と y です。

ファイル名: src/main.rs

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p = Point { x: 0, y: 7 };

    let Point { x, y } = p;
    assert_eq!(0, x);
    assert_eq!(7, y);
}
```

リスト 18-13: 構造体フィールド省略法で構造体のフィールドを分配する

このコードは、p 変数の x と y フィールドに一致する変数 x と y を生成します。結果は、変数 x と y が p 構造体の値を含むというものです。

また、全フィールドに対して変数を生成するのではなく、リテラル値を構造体パターンの一部にして分配することもできます。そうすることで他のフィールドは分配して変数を生成しつつ、一部のフィールドは特定の値と一致するか確認できます。

リスト 18-14 は、Point 値を 3 つの場合に区別する match 式を表示しています: x 軸上の点 (y = 0 ならそうなる)、y 軸上の点 (x = 0)、あるいはどちらでもありません。

ファイル名: src/main.rs

```
# struct Point {
#     x: i32,
#     y: i32,
# }
#
fn main() {
    let p = Point { x: 0, y: 7 };

    match p {
        // x軸上の{}
        Point { x, y: 0 } => println!("On the x axis at {}", x),
        // y軸上の{}
        Point { x: 0, y } => println!("On the y axis at {}", y),
        // どちらの軸上でもない: ({}, {})
        Point { x, y } => println!("On neither axis: ({}, {})", x, y),
    }
}
```

リスト 18-14: 分配とリテラル値との一致を 1 つのパターンで

最初のアームは、y フィールドの値がリテラル 0 と一致するならマッチすると指定することで、x 軸上にあるどんな点とも一致します。このパターンはそれでも、このアームのコードで使用できる x 変数を生成します。

同様に、2 番目のアームは、x フィールドが 0 ならマッチすると指定することで y 軸上のどんな点とも一致し、y フィールドの値には変数 y を生成します。3 番目のアームは何もリテラルを指定しないので、それ以外のあらゆる Point に合致し、x と y フィールド両方に変数を生成します。

この例で、値 p は 0 を含む x の力で 2 番目のアームに一致するので、このコードは On the y axis at 7 と出力します。

18.3.5.2　enum を分配する

例えば、第 6 章のリスト 6-5 で `Option<i32>` を分配するなどこの本の前半で enum を分配しました。明示的に触れなかった詳細の 1 つは、enum を分配するパターンは、enum 内に格納されているデータが定義されている手段に対応すべきということです。例として、リスト 18-15 では、リスト 6-2 から `Message` enum を使用し、内部の値それぞれを分配するパターンを伴う `match` を書いています。

ファイル名: src/main.rs

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}

fn main() {
    let msg = Message::ChangeColor(0, 160, 255);

    match msg {
        Message::Quit => {
            // Quit列挙子には分配すべきデータがない
            println!("The Quit variant has no data to destructure.")
        },
        Message::Move { x, y } => {
            println!(
                // x方向に{}、y方向に{}だけ動く
                "Move in the x direction {} and in the y direction {}",
                x,
                y
            );
        }
        // テキストメッセージ: {}
        Message::Write(text) => println!("Text message: {}", text),
        Message::ChangeColor(r, g, b) => {
            println!(
                // 色を赤{}, 緑{}, 青{}に変更
                "Change the color to red {}, green {}, and blue {}",
                r,
                g,
                b
            )
        }
    }
}
```

リスト 18-15: 異なる種類の値を保持する enum の列挙子を分配する

このコードは、Change the color to red 0, green 160, blue 255 と出力します。試しに msg の値を変更して、他のアームのコードが走るところを確認してください。

Message::Quit のようなデータのない enum 列挙子については、それ以上値を分配することができません。リテラル Message::Quit 値にマッチするだけで、変数はそのパターンに存在しません。

Message::Move のような構造体に似た enum の列挙子については、構造体と一致させるために指定するパターンと似たパターンを使用できます。列挙子の名前の後に波括弧を配置し、それから変数とともにフィールドを列挙するので、部品を分解してこのアームのコードで使用します。ここでは、リスト 18-13 のように省略形態を使用しています。

1 要素タプルを保持する Message::Write や、3 要素タプルを保持する Message::ChangeColor のようなタプルに似た enum の列挙子について、パターンは、タプルと一致させるために指定するパターンと類似しています。パターンの変数の数は、マッチ対象の列挙子の要素数と一致しなければなりません。

18.3.5.3　参照を分配する

パターンとマッチさせている値に参照が含まれる場合、値から参照を分配する必要があり、パターンに & を指定することでそうすることができます。そうすることで参照を保持する変数を得るのではなく、参照が指している値を保持する変数が得られます。このテクニックは、参照を走査するイテレータがあるクロージャで特に役に立ちますが、そのクロージャで参照ではなく、値を使用したいです。

リスト 18-16 の例は、ベクタの Point インスタンスへの参照を走査し、x と y 値に簡単に計算を行えるように、参照と構造体を分配します。

```
# struct Point {
#     x: i32,
#     y: i32,
# }
#
let points = vec![
    Point { x: 0, y: 0 },
    Point { x: 1, y: 5 },
    Point { x: 10, y: -3 },
];

let sum_of_squares: i32 = points
    .iter()
    .map(|&Point { x, y }| x * x + y * y)
    .sum();
```

リスト 18-16: 構造体への参照を構造体のフィールド値に分配する

このコードは、値 135 を保持する変数 sum_of_squares を返してきて、これは、x 値と y 値を 2 乗し、足し合わせ、points ベクタの Point それぞれの結果を足して 1 つの数値にした結果です。

&Point { x, y } に & が含まれていなかったら、型不一致エラーが発生していたでしょう。iter は

そうすると、実際の値ではなく、ベクタの要素への参照を走査するからです。そのエラーはこんな見た目でしょう:

```
error[E0308]: mismatched types
  -->
   |
14 |         .map(|Point { x, y }| x * x + y * y)
   |               ^^^^^^^^^^^^ expected &Point, found struct `Point`
   |
   = note: expected type `&Point`
              found type `Point`
```

このエラーは、コンパイラがクロージャに `&Point` と一致することを期待しているのに、`Point` への参照ではなく、`Point` 値に直接一致させようとしたことを示唆しています。

18.3.5.4　構造体とタプルを分配する

分配パターンをさらに複雑な方法で混ぜてマッチさせ、ネストすることができます。以下の例は、構造体とタプルをタプルにネストし、全ての基本的な値を取り出している複雑な分配を表示しています:

```
# struct Point {
#     x: i32,
#     y: i32,
# }
#
let ((feet, inches), Point {x, y}) = ((3, 10), Point { x: 3, y: -10 });
```

このコードは、複雑な型を構成する部品に分配させてくれるので、興味のある値を個別に使用できます。

パターンで分配することは、構造体の各フィールドからの値のように、複数の値をお互いに区別して使用する便利な方法です。

18.3.6　パターンの値を無視する

`match` の最後のアームのように、パターンの値を無視して実際には何もしないけれども、残りの全ての値の可能性を考慮する包括的なものを得ることは、時として有用であると認識しましたね。値全体やパターンの一部の値を無視する方法はいくつかあります: `_` パターンを使用すること (もう見かけました)、他のパターン内で `_` パターンを使用すること、アンダースコアで始まる名前を使用すること、`..` を使用して値の残りの部分を無視することです。これらのパターンそれぞれを使用する方法と理由を探究しましょう。

18.3.6.1　_で値全体を無視する

どんな値にも一致するけれども、値を束縛しないワイルドカードパターンとしてアンダースコア、_ を使用してきました。アンダースコア、_ パターンは特に `match` 式の最後のアームとして役に立ちま

すが、関数の引数も含めてあらゆるパターンで使えます。リスト 18-17 に示したようにですね。

ファイル名: src/main.rs

```
fn foo(_: i32, y: i32) {
    // このコードは、y引数を使うだけです: {}
    println!("This code only uses the y parameter: {}", y);
}

fn main() {
    foo(3, 4);
}
```

リスト 18-17: 関数シグニチャで _ を使用する

このコードは、最初の引数として渡された値 `3` を完全に無視し、`This code only uses the y parameter: 4` と出力します。

特定の関数の引数が最早必要ないほとんどの場合、未使用の引数が含まれないようにシグニチャを変更するでしょう。関数の引数を無視することが特に有用なケースもあり、例えば、トレイトを実装する際、特定の型シグニチャが必要だけれども、自分の実装の関数本体では引数の 1 つが必要ない時などです。そうすれば、代わりに名前を使った場合のようには、未使用関数引数についてコンパイラが警告することはないでしょう。

18.3.6.2　ネストされた_で値の一部を無視する

また、他のパターンの内部で _ を使用して、値の一部だけを無視することもでき、例えば、値の一部だけを確認したいけれども、走らせたい対応するコードでは他の部分を使用することがない時などです。リスト 18-18 は、設定の値を管理する責任を負ったコードを示しています。業務要件は、ユーザが既存の設定の変更を上書きすることはできないべきだけれども、設定を解除し、現在設定がされていなければ設定に値を与えられるというものです。

```
let mut setting_value = Some(5);
let new_setting_value = Some(10);

match (setting_value, new_setting_value) {
    (Some(_), Some(_)) => {
        // 既存の値の変更を上書きできません
        println!("Can't overwrite an existing customized value");
    }
    _ => {
        setting_value = new_setting_value;
    }
}

// 設定は{:?}です
println!("setting is {:?}", setting_value);
```

リスト 18-18: Some 内の値を使用する必要がない時に Some 列挙子と合致するパターンでアンダースコアを使用する

このコードは、Can't overwrite an existing customized value 、そして setting is Some(5) と出力するでしょう。最初のマッチアームで、どちらの Some 列挙子内部の値にも合致させたり、使用する必要はありませんが、setting_value と new_setting_value が Some 列挙子の場合を確かに確認する必要があります。その場合、何故 setting_value を変更しないかを出力し、変更しません。

2 番目のアームの _ パターンで表現される他のあらゆる場合 (setting_value と new_setting_value どちらかが None なら) には、new_setting_value に setting_value になってほしいです。

また、1 つのパターンの複数箇所でアンダースコアを使用して特定の値を無視することもできます。リスト 18-19 は、5 要素のタプルで 2 番目と 4 番目の値を無視する例です。

```
let numbers = (2, 4, 8, 16, 32);

match numbers {
    (first, _, third, _, fifth) => {
        // 何らかの数値: {}, {}, {}
        println!("Some numbers: {}, {}, {}", first, third, fifth)
    },
}
```

リスト 18-19: タプルの複数の部分を無視する

このコードは、Some numbers: 2, 8, 32 と出力し、値 4 と 16 は無視されます。

18.3.6.3　名前を_で始めて未使用の変数を無視する

変数を作っているのにどこでも使用していなければ、バグかもしれないのでコンパイラは通常、警告を発します。しかし時として、まだ使用しない変数を作るのが有用なこともあります。プロトタイプを開発していたり、プロジェクトを始めた直後だったりなどです。このような場面では、変数名をアンダースコアで始めることで、コンパイラに未使用変数について警告しないよう指示することができます。リスト 18-20 で 2 つの未使用変数を生成していますが、このコードを実行すると、そのうちの 1 つにしか警告が出ないはずです。

ファイル名: src/main.rs

```
fn main() {
    let _x = 5;
    let y = 10;
}
```

リスト 18-20: アンダースコアで変数名を始めて未使用変数警告が出るのを回避する

ここで、変数 y を使用していないことに対して警告が出ていますが、アンダースコアが接頭辞になっている変数には、使用していないという警告が出ていません。

_ だけを使うのとアンダースコアで始まる名前を使うことには微妙な違いがあることに注意してください。_x 記法はそれでも、値を変数に束縛する一方で、_ は全く束縛しません。この差異が問題になる場合を示すために、リスト 18-21 はエラーを提示するでしょう。

```
// こんにちは!
let s = Some(String::from("Hello!"));

if let Some(_s) = s {
    // 文字列が見つかりました
    println!("found a string");
}

println!("{:?}", s);
```

リスト 18-21: それでも、アンダースコアで始まる未使用の変数は値を束縛し、値の所有権を奪う可能性がある

それでも s 値は _s にムーブされ、再度 s を使用できなくするので、エラーを受け取るでしょう。ですが、アンダースコアを単独で使用すれば、値を束縛することは全くありません。s が _ にムーブされないので、リスト 18-22 はエラーなくコンパイルできます。

```
let s = Some(String::from("Hello!"));

if let Some(_) = s {
    println!("found a string");
}

println!("{:?}", s);
```

リスト 18-22: アンダースコアを使用すると、値を束縛しない

このコードは、s を何にも束縛しないので、ただ単に上手く動きます。つまり、ムーブされないのです。

18.3.6.4　.. で値の残りの部分を無視する

多くの部分がある値では、.. 記法を使用していくつかの部分だけを使用して残りを無視し、無視する値それぞれにアンダースコアを列挙する必要性を回避できます。.. パターンは、パターンの残りで明示的にマッチさせていない値のどんな部分も無視します。リスト 18-23 では、3 次元空間で座標を保持する `Point` 構造体があります。`match` 式で x 座標のみ処理し、y と z フィールドの値は無視したいです。

```
struct Point {
```

```
    x: i32,
    y: i32,
    z: i32,
}

let origin = Point { x: 0, y: 0, z: 0 };

match origin {
    Point { x, .. } => println!("x is {}", x),
}
```

リスト 18-23: `..` で `x` 以外の `Point` のフィールド全てを無視する

`x` 値を列挙し、それから `..` パターンを含んでいるだけです。これは、`y: _` や `z: _` と列挙しなければいけないのに比べて、手っ取り早いです。特に 1 つや 2 つのフィールドのみが関連する場面で多くのフィールドがある構造体に取り掛かっている時には。

`..` 記法は、必要な数だけ値に展開されます。リスト 18-24 は、タプルで `..` を使用する方法を表示しています。

ファイル名: src/main.rs

```
fn main() {
    let numbers = (2, 4, 8, 16, 32);

    match numbers {
        (first, .., last) => {
            println!("Some numbers: {}, {}", first, last);
        },
    }
}
```

リスト 18-24: タプルの最初と最後の値にだけ合致し、他の値を無視する

このコードにおいて、最初と最後の値は `first` と `last` に合致します。`..` は、途中のもの全部に合致し、無視します。

しかしながら、`..` を使うのは明確でなければなりません。どの値がマッチしてどの値が無視されるべきかが不明瞭なら、コンパイラはエラーを出します。リスト 18-25 は、`..` を曖昧に使用する例なので、コンパイルできません。

ファイル名: src/main.rs

```
fn main() {
    let numbers = (2, 4, 8, 16, 32);

    match numbers {
        (.., second, ..) => {
```

```
            println!("Some numbers: {}", second)
        },
    }
}
```

リスト 18-25: `..` を曖昧に使用しようとする試み

この例をコンパイルすると、こんなエラーが出ます:

```
error: `..` can only be used once per tuple or tuple struct pattern
(エラー: `..`は、タプルやタプル構造体パターン1つにつき、1回しか使用できません)
 --> src/main.rs:5:22
  |
5 |         (.., second, ..) => {
  |                      ^^
```

コンパイラが、second の値に合致する前にタプルの幾つの値を無視し、それからそれによってさらに幾つの値を無視するかを決めることは不可能です。このコードは、2 を無視し、second に 4 を束縛し、それから 8 、16 、32 を無視したり、2 と 4 を無視して second に 8 を束縛し、それから 16 と 32 を無視するなどを意味することもあるでしょう。変数名の second は、コンパイラにとってなんの特別な意味もなく、このように 2 箇所で `..` を使うのは曖昧なので、コンパイルエラーになります。

18.3.7　ref と ref mut でパターンに参照を生成する

ref を使用して値の所有権がパターンの変数にムーブされないように、参照を生成することに目を向けましょう。通常、パターンにマッチさせると、パターンで導入された変数は値に束縛されます。Rust の所有権規則は、その値が match などパターンを使用しているあらゆる場所にムーブされることを意味します。リスト 18-26 は、変数があるパターンとそれから match の後に値全体を println! 文で後ほど使用する match の例を示しています。このコードはコンパイルに失敗します。robot_name 値の一部の所有権が、最初の match アームのパターンの name 変数に移るからです。

```
let robot_name = Some(String::from("Bors"));

match robot_name {
    // 名前が見つかりました: {}
    Some(name) => println!("Found a name: {}", name),
    None => (),
}

// robot_nameは: {:?}
println!("robot_name is: {:?}", robot_name);
```

リスト 18-26: match アームパターンで変数を生成すると、値の所有権が奪われる

robot_name の一部の所有権が name にムーブされたので、robot_name に最早所有権がないために、

match の後に println! で最早 robot_name を使用することは叶いません。

このコードを修正するために、Some(name) パターンに所有権を奪わせるのではなく、robot_name のその部分を借用させたいです。パターンの外なら、値を借用する手段は、& で参照を生成することだと既にご認識でしょうから、解決策は Some(name) を Some(&name) に変えることだとお考えかもしれませんね。

しかしながら、「分配して値を分解する」節で見かけたように、パターンにおける & 記法は参照を**生成**せず、値の既存の参照に**マッチ**します。パターンにおいて & には既にその意味があるので、& を使用してパターンで参照を生成することはできません。

その代わりに、パターンで参照を生成するには、リスト 18-27 のように、新しい変数の前に ref キーワードを使用します。

```
let robot_name = Some(String::from("Bors"));

match robot_name {
    Some(ref name) => println!("Found a name: {}", name),
    None => (),
}

println!("robot_name is: {:?}", robot_name);
```

リスト 18-27: パターンの変数が値の所有権を奪わないように参照を生成する

robot_name の Some 列挙子の値が match にムーブされないので、この例はコンパイルできます; match はムーブするのではなく、robot_name のデータへの参照を取っただけなのです。

パターンで合致した値を可変化できるように可変参照を生成するには、&mut の代わりに ref mut を使用します。理由は今度も、パターンにおいて、前者は既存の可変参照にマッチするためにあり、新しい参照を生成しないからです。リスト 18-28 は、可変参照を生成するパターンの例です。

```
let mut robot_name = Some(String::from("Bors"));

match robot_name {
    // 別の名前
    Some(ref mut name) => *name = String::from("Another name"),
    None => (),
}

println!("robot_name is: {:?}", robot_name);
```

リスト 18-28: ref mut を使用して、パターンの一部として値への可変参照を生成する

この例はコンパイルが通り、robot_name is: Some("Another name") と出力するでしょう。name は可変参照なので、値を可変化するためにマッチアーム内で * 演算子を使用して参照外しする必要があります。

18.3.8　マッチガードで追加の条件式

マッチガードは、match アームのパターンの後に指定されるパターンマッチングとともに、そのアームが選択されるのにマッチしなければならない追加の if 条件です。マッチガードは、1 つのパターン単独でできるよりも複雑な考えを表現するのに役に立ちます。

この条件は、パターンで生成された変数を使用できます。リスト 18-29 は、最初のアームにパターン Some(x) と if x < 5 というマッチガードもある match を示しています。

```
let num = Some(4);

match num {
    // 5未満です: {}
    Some(x) if x < 5 => println!("less than five: {}", x),
    Some(x) => println!("{}", x),
    None => (),
}
```

リスト 18-29: パターンにマッチガードを追記する

この例は、less than five: 4 と出力します。num が最初のアームのパターンと比較されると、Some(4) は Some(x) に一致するので、マッチします。そして、マッチガードが x の値が 5 未満か確認し、そうなっているので、最初のアームが選択されます。

代わりに num が Some(10) だったなら、最初のアームのマッチガードは偽になったでしょう。10 は 5 未満ではないからです。Rust はそうしたら 2 番目のアームに移動し、マッチするでしょう。2 番目のアームにはマッチガードがなく、それ故にあらゆる Some 列挙子に一致するからです。

パターン内で if x < 5 という条件を表現する方法はありませんので、マッチガードにより、この論理を表現する能力が得られるのです。

リスト 18-11 において、マッチガードを使用すれば、パターンがシャドーイングする問題を解決できると述べました。match の外側の変数を使用するのではなく、match 式のパターン内部では新しい変数が作られることを思い出してください。その新しい変数は、外側の変数の値と比較することができないことを意味しました。リスト 18-30 は、マッチガードを使ってこの問題を修正する方法を表示しています。

ファイル名: src/main.rs

```
fn main() {
    let x = Some(5);
    let y = 10;

    match x {
        Some(50) => println!("Got 50"),
        Some(n) if n == y => println!("Matched, n = {:?}", n),
```

```
        _ => println!("Default case, x = {:?}", x),
    }

    println!("at the end: x = {:?}, y = {:?}", x, y);
}
```

リスト 18-30: マッチガードを使用して外側の変数と等しいか確認する

このコードは今度は、`Default case, x = Some(5)` と出力するでしょう。2 番目のマッチアームのパターンは、外側の `y` を覆い隠してしまう新しい変数 `y` を導入せず、マッチガード内で外側の `y` を使用できることを意味します。外側の `y` を覆い隠してしまう `Some(y)` としてパターンを指定するのではなく、`Some(n)` を指定しています。これにより、何も覆い隠さない新しい変数 `n` が生成されます。`match` の外側には `n` 変数は存在しないからです。

マッチガードの `if n == y` はパターンではなく、故に新しい変数を導入しません。この `y` は、新しいシャドーイングされた `y` ではなく、外側の `y` **であり**、`n` と `y` を比較することで、外側の `y` と同じ値を探すことができます。

また、マッチガードで **or** 演算子の `|` を使用して複数のパターンを指定することもできます; マッチガードの条件は全てのパターンに適用されます。リスト 18-31 は、`|` を使用するパターンとマッチガードを組み合わせる優先度を示しています。この例で重要な部分は、`if y` は `6` にしか適用されないように見えるのに、`if y` マッチガードが `4`、`5`、**そして** `6` に適用されることです。

```
let x = 4;
let y = false;

match x {
    // はい
    4 | 5 | 6 if y => println!("yes"),
    // いいえ
    _ => println!("no"),
}
```

リスト 18-31: 複数のパターンとマッチガードを組み合わせる

マッチの条件は、`x` の値が `4`、`5`、`6` に等しく**かつ** `y` が `true` の場合だけにアームがマッチすると宣言しています。このコードが走ると、最初のアームのパターンは `x` が `4` なので、合致しますが、マッチガード `if y` は偽なので、最初のアームは選ばれません。コードは 2 番目のアームに移動して、これがマッチし、このプログラムは `no` と出力します。理由は、`if` 条件が最後の値の `6` だけでなく、パターン全体 `4 | 5 | 6` に適用されるからです。言い換えると、パターンと関わるマッチガードの優先度は、以下のように振る舞います:

```
(4 | 5 | 6) if y => ...
```

以下のようにではありません:

```
4 | 5 | (6 if y) => ...
```

コードを実行後には、優先度の動作は明らかになります: マッチガードが | 演算子で指定される値のリストの最後の値にしか適用されないなら、アームはマッチし、プログラムは yes と出力したでしょう。

18.3.9　@束縛

at 演算子 (`@`) により、値を保持する変数を生成するのと同時にその値がパターンに一致するかを調べることができます。リスト 18-32 は、`Message::Hello` の `id` フィールドが範囲 `3..=7` にあるかを確かめたいという例です。しかし、アームに紐づいたコードで使用できるように変数 `id_variable` に値を束縛もしたいです。この変数をフィールドと同じ、`id` と名付けることもできますが、この例では異なる名前にします。

```
enum Message {
    Hello { id: i32 },
}

let msg = Message::Hello { id: 5 };

match msg {
    Message::Hello { id: id_variable @ 3..=7 } => {
        // 範囲内のidが見つかりました: {}
        println!("Found an id in range: {}", id_variable)
    },
    Message::Hello { id: 10..=12 } => {
        // 別の範囲内のidが見つかりました
        println!("Found an id in another range")
    },
    Message::Hello { id } => {
        // それ以外のidが見つかりました
        println!("Found some other id: {}", id)
    },
}
```

`@` を使用してテストしつつ、パターンの値に束縛する

この例は、`Found an id in range: 5` と出力します。範囲 `3..=7` の前に `id_variable @` と指定することで、値が範囲パターンに一致することを確認しつつ、範囲にマッチしたどんな値も捕捉しています。

パターンで範囲しか指定していない 2 番目のアームでは、アームに紐づいたコードに `id` フィールドの実際の値を含む変数はありません。`id` フィールドの値は 10、11、12 だった可能性があるでしょうが、そのパターンに来るコードは、どれなのかわかりません。パターンのコードは `id` フィールドの値を使用することは叶いません。`id` の値を変数に保存していないからです。

範囲なしに変数を指定している最後のアームでは、確かにアームのコードで使用可能な値が `id` という変数にあります。理由は、構造体フィールド省略記法を使ったからです。しかし、このアームで `id` フィールドの値に対して、最初の 2 つのアームのようには、確認を行っていません: どんな値でも、このパターンに一致するでしょう。

`@` を使用することで、値を検査しつつ、1 つのパターン内で変数に保存させてくれるのです。

18.4　まとめ

Rust のパターンは、異なる種類のデータを区別するのに役立つという点でとても有用です。`match` 式で使用されると、コンパイラはパターンが全ての可能性を網羅しているか保証し、そうでなければプログラムはコンパイルできません。`let` 文や関数の引数のパターンは、その構文をより有用にし、値を分配して小さな部品にすると同時に変数に代入できるようにしてくれます。単純だったり複雑だったりするパターンを生成してニーズに合わせることができます。

次の本書の末尾から 2 番目の章では、Rust の多彩な機能の高度な視点に目を向けます。

第 19 章

高度な機能

今までに、Rust プログラミング言語の最もよく使われる部分を学んできました。第 20 章でもう 1 つ別のプロジェクトを行う前に、時折遭遇する言語の側面をいくつか見ましょう。この章は、Rust を使用する際に知らないことに遭遇した時に参考にすることができます。この章で使用することを学ぶ機能は、かなり限定的な場面でしか役に立ちません。あまり頻繁には手を伸ばすことがない可能性はありますが、Rust が提供しなければならない機能全ての概要を確かに把握してもらいたいのです。

この章で講義するのは:

- Unsafe Rust: Rust の保証の一部を抜けてその保証を手動で保持する責任を負う方法
- 高度なトレイト: 関連型、デフォルト型引数、フルパス記法、スーパートレイト、トレイトに関連するニュータイプパターン
- 高度な型: ニュータイプパターンについてもっと、型エイリアス、never 型、動的サイズ決定型
- 高度な関数とクロージャ: 関数ポインタとクロージャの返却
- マクロ: コンパイル時に、より多くのコードを定義するコードを定義する方法

皆さんのための何かがある Rust の機能の盛大な儀式です！ さあ、飛び込みましょう！

19.1 Unsafe Rust

ここまでに議論してきたコードは全て、Rust のメモリ安全保証がコンパイル時に強制されていました。しかしながら、Rust には、これらのメモリ安全保証を強制しない第 2 の言語が中に隠されています: それは **unsafe Rust** と呼ばれ、普通の Rust のように動きますが、おまけの強大な力を与えてくれます。

静的解析は原理的に保守的なので、unsafe Rust が存在します。コードが保証を保持しているかコンパイラが決定しようとする際、なんらかの不正なプログラムを受け入れるよりも合法なプログラムを拒否したほうがいいのです。コードは大丈夫かもしれないけれど、コンパイラにわかる範囲ではダ

メなのです！ このような場合、unsafe コードを使用してコンパイラに「信じて！ 何をしているかわかってるよ」と教えられます。欠点は、自らのリスクで使用することです: unsafe コードを誤って使用したら、null ポインタ参照外しなどのメモリ非安全に起因する問題が起こることもあるのです。

Rust に unsafe な分身がある別の理由は、根本にあるコンピュータのハードウェアが本質的に unsafe だからです。Rust が unsafe な処理を行わせてくれなかったら、特定の仕事を行えないでしょう。Rust は、低レベルなシステムプログラミングを許可する必要があります。直接 OS と相互作用したり、独自の OS を書くことさえもそうです。低レベルなシステムプログラミングに取り組むことは、言語の目標の 1 つなのです。unsafe Rust でできることとその方法を探究しましょう。

19.1.1　unsafe の強大な力 (superpower)

unsafe Rust に切り替えるには、`unsafe` キーワードを使用し、それから unsafe コードを保持する新しいブロックを開始してください。safe Rust では行えない 4 つの行動を unsafe Rust では行え、これは **unsafe superpowers** と呼ばれます。その superpower には、以下の能力が含まれています:

- 生ポインタを参照外しすること
- unsafe な関数やメソッドを呼ぶこと
- 可変で静的な変数にアクセスしたり変更すること
- unsafe なトレイトを実装すること

`unsafe` は、借用チェッカーや他の Rust の安全性チェックを無効にしないことを理解するのは重要なことです: unsafe コードで参照を使用しても、チェックはされます。`unsafe` キーワードにより、これら 4 つの機能にアクセスできるようになり、その場合、コンパイラによってこれらのメモリ安全性は確認されないのです。unsafe ブロック内でも、ある程度の安全性は得られます。

また、unsafe は、そのブロックが必ずしも危険だったり、絶対メモリ安全上の問題を抱えていることを意味するものではありません: その意図は、プログラマとして `unsafe` ブロック内のコードがメモリに合法的にアクセスすることを保証することです。

人間は失敗をするもので、間違いも起きますが、これら 4 つの unsafe な処理を `unsafe` で注釈されたブロックに入れる必要があることで、メモリ安全性に関するどんなエラーも `unsafe` ブロック内にあるに違いないと知ります。`unsafe` ブロックは小さくしてください; メモリのバグを調査するときに感謝することになるでしょう。

unsafe なコードをできるだけ分離するために、unsafe なコードを安全な抽象の中に閉じ込め、安全な API を提供するのが最善です。これについては、後ほど unsafe な関数とメソッドを調査する際に議論します。標準ライブラリの一部は、検査された unsafe コードの安全な抽象として実装されています。安全な抽象に unsafe なコードを包むことで、`unsafe` が、あなたやあなたのユーザが `unsafe` コードで実装された機能を使いたがる可能性のある箇所全部に漏れ出ることを防ぎます。安全な抽象を使用することは、安全だからです。

4 つの unsafe な superpower を順に見ていきましょう。unsafe なコードへの安全なインターフェイスを提供する一部の抽象化にも目を向けます。

19.1.2　生ポインタを参照外しする

第 4 章の「ダングリング参照」節で、コンパイラは、参照が常に有効であることを保証することに触れました。unsafe Rust には参照に類似した**生ポインタ**と呼ばれる 2 つの新しい型があります。参照同様、生ポインタも不変や可変になり得て、それぞれ `*const T` と `*mut T` と表記されます。このアスタリスクは、参照外し演算子ではありません; 型名の一部です。生ポインタの文脈では、**不変**は、参照外し後に直接ポインタに代入できないことを意味します。

参照やスマートポインタと異なり、生ポインタは:

- 同じ場所への不変と可変なポインタや複数の可変なポインタが存在することで借用規則を無視できる
- 有効なメモリを指しているとは保証されない
- null の可能性がある
- 自動的な片付けは実装されていない

これらの保証をコンパイラに強制させることから抜けることで、保証された安全性を諦めてパフォーマンスを向上させたり、Rust の保証が適用されない他の言語やハードウェアとのインターフェイスの能力を得ることができます。

リスト 19-1 は、参照から不変と可変な生ポインタを生成する方法を示しています。

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;
```

リスト 19-1: 参照から生ポインタを生成する

このコードには `unsafe` キーワードを含めていないことに注意してください。safe コードで生ポインタを生成できます; もうすぐわかるように、unsafe ブロックの外では、生ポインタを参照外しできないだけなのです。

`as` を使って不変と可変な参照を対応する生ポインタの型にキャストして生ポインタを生成しました。有効であることが保証される参照から直接生ポインタを生成したので、これらの特定の生ポインタは有効であることがわかりますが、その前提をあらゆる生ポインタに敷くことはできません。

次に、有効であることが確信できない生ポインタを生成します。リスト 19-2 は、メモリの任意の箇所を指す生ポインタの生成法を示しています。任意のメモリを使用しようとすることは未定義です: そのアドレスにデータがある可能性もあるし、ない可能性もあり、コンパイラがコードを最適化してメモリアクセスがなくなる可能性もあるし、プログラムがセグメンテーションフォールトでエラーに

なる可能性もあります。通常、このようなコードを書くいい理由はありませんが、可能ではあります。

```
let address = 0x012345usize;
let r = address as *const i32;
```

リスト 19-2: 任意のメモリアドレスへの生ポインタを生成する

safe コードで生ポインタを生成できるけれども、生ポインタを**参照外し**して指しているデータを読むことはできないことを思い出してください。リスト 19-3 では、`unsafe` ブロックが必要になる参照外し演算子の `*` を生ポインタに使っています。

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;

unsafe {
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", *r2);
}
```

リスト 19-3: `unsafe` ブロック内で生ポインタを参照外しする

ポインタの生成は害を及ぼしません; 問題が起こり得るのはポインタが指している値にアクセスしようとするときのみで、この際に無効な値を扱うことになる可能性があります。

また、リスト 19-1 とリスト 19-3 では、`num` が格納されている同じメモリ上の場所を両方とも指す `*const i32` と `*mut i32` の生ポインタを生成したことに注目してください。代わりに `num` への不変と可変な参照を生成しようとしたら、コードはコンパイルできなかったでしょう。Rust の所有権規則により、不変参照と可変参照を同時に存在させられないからです。生ポインタなら、同じ場所への可変なポインタと不変なポインタを生成でき、可変なポインタを通してデータを変更し、データ競合を引き起こす可能性があります。気を付けてください！

これらの危険がありながら、一体何故生ポインタを使うのでしょうか？ 主なユースケースの 1 つは、次の節「unsafe な関数やメソッドを呼ぶ」で見るように、C コードとのインターフェイスです。別のユースケースは、借用チェッカーには理解できない安全な抽象を構成する時です。unsafe な関数を導入し、それから unsafe コードを使用する安全な抽象の例に目を向けます。

19.1.3　unsafe な関数やメソッドを呼ぶ

unsafe ブロックが必要になる 2 番目の処理は、unsafe 関数の呼び出しです。unsafe な関数やメソッドも見た目は、普通の関数やメソッドと全く同じですが、残りの定義の前に追加の `unsafe` があります。この文脈での `unsafe` キーワードは、この関数を呼ぶ際に保持しておく必要のある要求が関数にあることを示唆します。コンパイラには、この要求を満たしているか保証できないからです。

unsafe ブロックで unsafe な関数を呼び出すことで、この関数のドキュメンテーションを読み、関数の契約を守っているという責任を取ると宣言します。

こちらは、本体で何もしない dangerous という unsafe な関数です:

```
unsafe fn dangerous() {}

unsafe {
    dangerous();
}
```

個別の unsafe ブロックで dangerous 関数を呼ばなければなりません。unsafe ブロックなしで dangerous を呼ぼうとすれば、エラーになるでしょう:

```
error[E0133]: call to unsafe function requires unsafe function or block
(エラー: unsafe関数の呼び出しには、unsafeな関数かブロックが必要です)
 -->
  |
4 |     dangerous();
  |     ^^^^^^^^^^ call to unsafe function
```

dangerous への呼び出しの周りに unsafe ブロックを挿入することで、コンパイラに関数のドキュメンテーションを読み、適切に使用する方法を理解したことをアサートし、関数の契約を満たしていると実証しました。

unsafe 関数の本体は、実効的に unsafe ブロックになるので、unsafe 関数内で unsafe な別の処理を行うのに、別の unsafe ブロックは必要ないのです。

19.1.3.1　unsafe コードに安全な抽象を行う

関数が unsafe なコードを含んでいるだけで関数全体を unsafe でマークする必要があることにはなりません。事実、安全な関数で unsafe なコードをラップすることは一般的な抽象化です。例として、なんらかの unsafe コードが必要になる標準ライブラリの関数 split_at_mut を学び、その実装方法を探究しましょう。この安全なメソッドは、可変なスライスに定義されています: スライスを 1 つ取り、引数で与えられた添え字でスライスを分割して 2 つにします。リスト 19-4 は、split_at_mut の使用法を示しています。

```
let mut v = vec![1, 2, 3, 4, 5, 6];

let r = &mut v[..];

let (a, b) = r.split_at_mut(3);

assert_eq!(a, &mut [1, 2, 3]);
assert_eq!(b, &mut [4, 5, 6]);
```

リスト 19-4: 安全な split_at_mut 関数を使用する

この関数を safe Rust だけを使用して実装することはできません。試みは、リスト 19-5 のようになる可能性がありますが、コンパイルできません。簡単のため、split_at_mut をメソッドではなく関数として実装し、ジェネリックな型 T ではなく、i32 のスライス用に実装します。

```
fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
    let len = slice.len();

    assert!(mid <= len);

    (&mut slice[..mid],
     &mut slice[mid..])
}
```

リスト 19-5: safe Rust だけを使用した split_at_mut の未遂の実装

この関数はまず、スライスの全体の長さを得ます。それから引数で与えられた添え字が長さ以下であるかを確認してスライス内にあることをアサートします。このアサートは、スライスを分割する添え字よりも大きい添え字を渡したら、その添え字を使用しようとする前に関数がパニックすることを意味します。

そして、2 つの可変なスライスをタプルで返します: 1 つは元のスライスの最初から mid 添え字まで、もう一方は、mid からスライスの終わりまでです。

リスト 19-5 のコードのコンパイルを試みると、エラーになるでしょう。

```
error[E0499]: cannot borrow `*slice` as mutable more than once at a time
(エラー: 一度に2回以上、`*slice`を可変で借用できません)
 -->
  |
6 |     (&mut slice[..mid],
  |           ----- first mutable borrow occurs here
7 |      &mut slice[mid..])
  |           ^^^^^ second mutable borrow occurs here
8 | }
  | - first borrow ends here
```

Rust の借用チェッカーには、スライスの異なる部分を借用していることが理解できないのです; 同じスライスから 2 回借用していることだけ知っています。2 つのスライスが被らないので、スライスの異なる部分を借用することは、根本的に大丈夫なのですが、コンパイラはこれを知れるほど賢くありません。プログラマにはコードが大丈夫とわかるのに、コンパイラにはわからないのなら、unsafe コードに手を伸ばすタイミングです。

リスト 19-6 は unsafe ブロック、生ポインタ、unsafe 関数への呼び出しをして split_at_mut の実装が動くようにする方法を示しています。

```
use std::slice;

fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
```

```
    let len = slice.len();
    let ptr = slice.as_mut_ptr();

    assert!(mid <= len);

    unsafe {
        (slice::from_raw_parts_mut(ptr, mid),
         slice::from_raw_parts_mut(ptr.offset(mid as isize), len - mid))
    }
}
```

リスト 19-6: `split_at_mut` 関数の実装で unsafe コードを使用する

第 4 章の「スライス型」節から、スライスはなんらかのデータへのポインタとスライスの長さであることを思い出してください。`len` メソッドを使用してスライスの長さを得て、`as_mut_ptr` メソッドを使用してスライスの生ポインタにアクセスしています。この場合、`i32` 値の可変スライスがあるので、`as_mut_ptr` は型 `*mut i32` の生ポインタを返し、これを変数 `ptr` に格納しました。

`mid` 添え字がスライス内にあるかというアサートを残しています。そして、unsafe コードに到達します: `slice::from_raw_parts_mut` 関数は、生ポインタと長さを取り、スライスを生成します。この関数を使って、`ptr` から始まり、`mid` の長さのスライスを生成しています。それから `ptr` に `mid` を引数として `offset` メソッドを呼び出し、`mid` で始まる生ポインタを得て、そのポインタと `mid` の後の残りの要素数を長さとして使用してスライスを生成しています。

関数 `slice::from_raw_parts_mut` は、unsafe です。何故なら、生ポインタを取り、このポインタが有効であることを信用しなければならないからです。生ポインタの `offset` メソッドも unsafe です。オフセット位置もまた有効なポインタであることを信用しなければならないからです。故に、`slice::from_raw_parts_mut` と `offset` を呼べるように、その呼び出しの周りに `unsafe` ブロックを置かなければならなかったのです。コードを眺めて `mid` が `len` 以下でなければならないとするアサートを追加することで、`unsafe` ブロック内で使用されている生ポインタが全てスライス内のデータへの有効なポインタであることがわかります。これは、受け入れられ、適切な `unsafe` の使用法です。

できあがった `split_at_mut` 関数を `unsafe` でマークする必要はなく、この関数を safe Rust から呼び出せることに注意してください。`unsafe` コードを安全に使用する関数の実装で、unsafe コードへの安全な抽象化を行いました。この関数がアクセスするデータからの有効なポインタだけを生成するからです。

対照的に、リスト 19-7 の `slice::from_raw_parts_mut` の使用は、スライスが使用されるとクラッシュする可能性が高いでしょう。このコードは任意のメモリアドレスを取り、10,000 要素の長さのスライスを生成します:

```
use std::slice;

let address = 0x012345usize;
let r = address as *mut i32;
```

```
let slice = unsafe {
    slice::from_raw_parts_mut(r, 10000)
};
```

リスト 19-7: 任意のメモリアドレスからスライスを生成する

この任意の場所のメモリは所有していなく、このコードが生成するスライスに有効な `i32` 値が含まれる保証もありません。`slice` を有効なスライスであるかのように使用しようとすると、未定義動作に陥ります。

19.1.3.2　`extern` 関数を使用して、外部のコードを呼び出す

時として、自分の Rust コードが他の言語で書かれたコードと相互作用する必要が出てくる可能性があります。このために、Rust には `extern` というキーワードがあり、これは、**FFI**(Foreign Function Interface: 外部関数インターフェイス) の生成と使用を容易にします。FFI は、あるプログラミング言語に関数を定義させ、異なる (外部の) プログラミング言語にそれらの関数を呼び出すことを可能にする方法です

リスト 19-8 は、C の標準ライブラリから `abs` 関数を統合するセットアップ方法をデモしています。`extern` ブロック内で宣言された関数は、常に Rust コードから呼ぶには unsafe になります。理由は、他の言語では、Rust の規則や保証が強制されず、コンパイラもチェックできないので、安全性を保証する責任はプログラマに降りかかるのです。

ファイル名: src/main.rs

```
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    unsafe {
        // -3の絶対値は、Cによると{}
        println!("Absolute value of -3 according to C: {}", abs(-3));
    }
}
```

リスト 19-8: 他の言語で定義された `extern` 関数を宣言し、呼び出す

`extern "C"` ブロック内で他の言語から呼び出した関数の名前とシグニチャを列挙します。`"C"` の部分は、外部関数がどの **ABI**(application binary interface: アプリケーション・バイナリ・インターフェイス) を使用しているか定義します: ABI は関数の呼び出し方法をアセンブリレベルで定義します。`"C"` ABI は最も一般的で C プログラミング言語の ABI に従っています。

19.1.3.3　他の言語から Rust の関数を呼び出す

また、extern を使用して他の言語に Rust の関数を呼ばせるインターフェイスを生成することもできます。extern ブロックの代わりに、extern キーワードを追加し、fn キーワードの直前に使用する ABI を指定します。さらに、#[no_mangle] 注釈を追加して Rust コンパイラに関数名をマングルしないように指示する必要もあります。**マングル**とは、コンパイラが関数に与えた名前を他のコンパイル過程の情報をより多く含むけれども、人間に読みにくい異なる名前にすることです。全ての言語のコンパイラは、少々異なる方法でマングルを行うので、Rust の関数が他の言語で名前付けできるように、Rust コンパイラの名前マングルをオフにしなければならないのです。

以下の例では、共有ライブラリにコンパイルし、C からリンクした後に call_from_c 関数を C コードからアクセスできるようにしています:

```
#[no_mangle]
pub extern "C" fn call_from_c() {
    // CからRust関数を呼び出したばかり！
    println!("Just called a Rust function from C!");
}
```

この extern の使用法では、unsafe は必要ありません。

19.1.4　可変で静的な変数にアクセスしたり、変更する

今までずっと、**グローバル変数**について語りませんでした。グローバル変数を Rust は確かにサポートしていますが、Rust の所有権規則で問題になることもあります。2 つのスレッドが同じ可変なグローバル変数にアクセスしていたら、データ競合を起こすこともあります。

Rust では、グローバル変数は、**static**(静的) 変数と呼ばれます。リスト 19-9 は、値として文字列スライスのある静的変数の宣言例と使用を示しています。

ファイル名: src/main.rs

```
static HELLO_WORLD: &str = "Hello, world!";

fn main() {
    // 名前は: {}
    println!("name is: {}", HELLO_WORLD);
}
```

リスト 19-9: 不変で静的な変数を定義し、使用する

静的変数は、定数に似ています。定数については、第 3 章の「変数と定数の違い」節で議論しました。静的変数の名前は慣習で SCREAMING_SNAKE_CASE (直訳：叫ぶスネークケース) になり、変数の型

を注釈し**なければなりません**。この例では `&'static str` です。静的変数は、`'static` ライフタイムの参照のみ格納でき、これは、Rust コンパイラがライフタイムを推量できることを意味します; 明示的に注釈する必要はありません。不変で静的な変数にアクセスすることは安全です。

定数と不変で静的な変数は、類似して見える可能性がありますが、微妙な差異は、静的変数の値は固定されたメモリアドレスになることです。値を使用すると、常に同じデータにアクセスします。一方、定数は使用される度にデータを複製させることができます。

定数と静的変数の別の違いは、静的変数は可変にもなることです。可変で静的な変数にアクセスし変更することは、unsafe です。リスト 19-10 は、`COUNTER` という可変で静的な変数を宣言し、アクセスし、変更する方法を表示しています。

ファイル名: src/main.rs

```
static mut COUNTER: u32 = 0;

fn add_to_count(inc: u32) {
    unsafe {
        COUNTER += inc;
    }
}

fn main() {
    add_to_count(3);

    unsafe {
        println!("COUNTER: {}", COUNTER);
    }
}
```

リスト 19-10: 可変で静的な変数を読んだり、書き込むのは unsafe である

普通の変数同様、`mut` キーワードを使用して可変性を指定します。`COUNTER` を読み書きするコードはどれも、`unsafe` ブロックになければなりません。シングルスレッドなので、このコードは想定通り、コンパイルでき、`COUNTER: 3` と出力します。複数のスレッドに `COUNTER` にアクセスさせると、データ競合になる可能性が高いでしょう。

グローバルにアクセス可能な可変なデータがあると、データ競合がないことを保証するのは難しくなり、そのため、Rust は可変で静的な変数を unsafe と考えるのです。可能なら、コンパイラが、データが異なるスレッドからアクセスされることが安全に行われているかを確認するように、第 16 章で議論した並行性テクニックとスレッド安全なスマートポインタを使用するのが望ましいです。

19.1.5　unsafe なトレイトを実装する

`unsafe` でのみ動く最後の行動は、unsafe なトレイトを実装することです。少なくとも、1 つのメソッドにコンパイラが確かめられないなんらかの不変条件があると、トレイトは unsafe になります。

trait の前に unsafe キーワードを追加し、トレイトの実装も unsafe でマークすることで、トレイトが unsafe であると宣言できます。リスト 19-11 のようにですね。

```
unsafe trait Foo {
    // methods go here
    // メソッドがここに来る
}

unsafe impl Foo for i32 {
    // method implementations go here
    // メソッドの実装がここに来る
}
```

リスト 19-11: unsafe なトレイトを定義して実装する

unsafe impl を使用することで、コンパイラが確かめられない不変条件を守ることを約束しています。

例として、第 16 章の「Sync と Send トレイトで拡張可能な並行性」節で議論した Sync と Send マーカートレイトを思い出してください: 型が完全に Send と Sync 型だけで構成されていたら、コンパイラはこれらのトレイトを自動的に実装します。生ポインタなどの Send や Sync でない型を含む型を実装し、その型を Send や Sync でマークしたいなら、unsafe を使用しなければなりません。コンパイラは、型がスレッド間を安全に送信できたり、複数のスレッドから安全にアクセスできるという保証を保持しているか確かめられません; 故に、そのチェックを手動で行い、unsafe でそのように示唆する必要があります。

19.1.6　いつ unsafe コードを使用するべきか

unsafe を使って議論したばかりの 4 つの行動 (強大な力) のうちの 1 つを行うのは間違っていたり、認められさえもしないものではありません。ですが、unsafe コードを正しくするのは、より巧妙なことでしょう。コンパイラがメモリ安全性を保持する手助けをできないからです。unsafe コードを使用する理由があるなら、そうすることができ、明示的に unsafe 注釈をすることで問題が起きたら、その原因を追求するのが容易になります。

19.2　高度なトレイト

最初にトレイトについて講義したのは、第 10 章の「トレイト: 共通の振る舞いを定義する」節でしたが、ライフタイム同様、より高度な詳細は議論しませんでした。今や、Rust に詳しくなったので、核心に迫れるでしょう。

19.2.1　関連型でトレイト定義においてプレースホルダーの型を指定する

関連型は、トレイトのメソッド定義がシグニチャでプレースホルダーの型を使用できるように、トレイトと型のプレースホルダーを結び付けます。トレイトを実装するものがこの特定の実装で型の位置に使用される具体的な型を指定します。そうすることで、なんらかの型を使用するトレイトをトレイトを実装するまでその型が一体なんであるかを知る必要なく定義できます。

この章のほとんどの高度な機能は、稀にしか必要にならないと解説しました。関連型はその中間にあります: 本の他の部分で説明される機能よりは使用されるのが稀ですが、この章で議論される他の多くの機能よりは頻繁に使用されます。

関連型があるトレイトの一例は、標準ライブラリが提供する `Iterator` トレイトです。その関連型は `Item` と名付けられ、`Iterator` トレイトを実装している型が走査している値の型の代役を務めます。第 13 章の「`Iterator` トレイトと `next` メソッド」節で、`Iterator` トレイトの定義は、リスト 19-20 に示したようなものであることに触れました。

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
}
```

リスト 19-20: 関連型 `Item` がある `Iterator` トレイトの定義

型 `Item` はプレースホルダー型で `next` メソッドの定義は、型 `Option<Self::Item>` の値を返すことを示しています。`Iterator` トレイトを実装するものは、`Item` の具体的な型を指定し、`next` メソッドは、その具体的な型の値を含む `Option` を返します。

関連型は、ジェネリクスにより扱う型を指定せずに関数を定義できるという点でジェネリクスに似た概念のように思える可能性があります。では、何故関連型を使用するのでしょうか？

2 つの概念の違いを第 13 章から `Counter` 構造体に `Iterator` トレイトを実装する例で調査しましょう。リスト 13-21 で、`Item` 型は `u32` だと指定しました:

ファイル名: src/lib.rs

```
impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        // --snip--
```

この記法は、ジェネリクスと比較可能に思えます。では、何故単純にリスト 19-21 のように、`Iterator` トレイトをジェネリクスで定義しないのでしょうか？

```
pub trait Iterator<T> {
    fn next(&mut self) -> Option<T>;
}
```

リスト 19-21: ジェネリクスを使用した架空の `Iterator` トレイトの定義

差異は、リスト 19-21 のようにジェネリクスを使用すると、各実装で型を注釈しなければならないことです; `Iterator<String> for Counter` や他のどんな型にも実装することができるので、`Counter` の `Iterator` の実装が複数できるでしょう。換言すれば、トレイトにジェネリックな引数があると、毎回ジェネリックな型引数の具体的な型を変更してある型に対して複数回実装できるということです。`Counter` に対して `next` メソッドを使用する際に、どの `Iterator` の実装を使用したいか型注釈をつけなければならないでしょう。

関連型なら、同じ型に対してトレイトを複数回実装できないので、型を注釈する必要はありません。関連型を使用する定義があるリスト 19-20 では、`Item` の型は 1 回しか選択できませんでした。1 つしか `impl Iterator for Counter` がないからです。`Counter` に `next` を呼び出す度に、`u32` 値のイテレータが欲しいと指定しなくてもよいわけです。

19.2.2　デフォルトのジェネリック型引数と演算子オーバーロード

ジェネリックな型引数を使用する際、ジェネリックな型に対して既定の具体的な型を指定できます。これにより、既定の型が動くのなら、トレイトを実装する側が具体的な型を指定する必要を排除します。ジェネリックな型に既定の型を指定する記法は、ジェネリックな型を宣言する際に `<PlaceholderType=ConcreteType>` です。

このテクニックが有用になる場面の好例が、演算子オーバーロードです。**演算子オーバーロード**とは、特定の状況で演算子 (`+` など) の振る舞いをカスタマイズすることです。

Rust では、独自の演算子を作ったり、任意の演算子をオーバーロードすることはできません。しかし、演算子に紐づいたトレイトを実装することで `std::ops` に列挙された処理と対応するトレイトをオーバーロードできます。例えば、リスト 19-22 で `+` 演算子をオーバーロードして 2 つの `Point` インスタンスを足し合わせています。`Point` 構造体に `Add` トレイトを実装することでこれを行なっています。

ファイル名: src/main.rs

```
use std::ops::Add;

#[derive(Debug, PartialEq)]
struct Point {
    x: i32,
    y: i32,
}

impl Add for Point {
```

```
    type Output = Point;

    fn add(self, other: Point) -> Point {
        Point {
            x: self.x + other.x,
            y: self.y + other.y,
        }
    }
}

fn main() {
    assert_eq!(Point { x: 1, y: 0 } + Point { x: 2, y: 3 },
               Point { x: 3, y: 3 });
}
```

リスト 19-22: Add トレイトを実装して Point インスタンス用に + 演算子をオーバーロードする

add メソッドは 2 つの Point インスタンスの x 値と 2 つの Point インスタンスの y 値を足します。Add トレイトには、add メソッドから返却される型を決定する Output という関連型があります。

このコードの既定のジェネリック型は、Add トレイト内にあります。こちらがその定義です:

```
trait Add<RHS=Self> {
    type Output;

    fn add(self, rhs: RHS) -> Self::Output;
}
```

このコードは一般的に馴染みがあるはずです: 1 つのメソッドと関連型が 1 つあるトレイトです。新しい部分は、RHS=Self です: この記法は、**デフォルト型引数**と呼ばれます。RHS というジェネリックな型引数 (”right hand side”: 右辺の省略形) が、add メソッドの rhs 引数の型を定義しています。Add トレイトを実装する際に RHS の具体的な型を指定しなければ、RHS の型は標準で Self になり、これは Add を実装している型になります。

Point に Add を実装する際、2 つの Point インスタンスを足したかったので、RHS の規定を使用しました。既定を使用するのではなく、RHS の型をカスタマイズしたくなる Add トレイトの実装例に目を向けましょう。

異なる単位で値を保持する構造体、Millimeters と Meters (それぞれ ミリメートル と メートル) が 2 つあります。ミリメートルの値をメートルの値に足し、Add の実装に変換を正しくしてほしいです。Add を RHS に Meters のある Millimeters に実装することができます。リスト 19-23 のように:

ファイル名: src/lib.rs

```
use std::ops::Add;

struct Millimeters(u32);
struct Meters(u32);
```

```
impl Add<Meters> for Millimeters {
    type Output = Millimeters;

    fn add(self, other: Meters) -> Millimeters {
        Millimeters(self.0 + (other.0 * 1000))
    }
}
```

リスト 19-23: `Millimeters` に `Add` トレイトを実装して、`Meters` に `Millimeters` を足す

`Millimeters` を `Meters` に足すため、`Self` という既定を使う代わりに `impl Add<Meters>` を指定して、`RHS` 型引数の値をセットしています。

主に 2 通りの方法でデフォルト型引数を使用します:

- 既存のコードを破壊せずに型を拡張する
- ほとんどのユーザは必要としない特定の場合でカスタマイズを可能にする

標準ライブラリの `Add` トレイトは、2 番目の目的の例です: 通常、2 つの似た型を足しますが、`Add` トレイトはそれ以上にカスタマイズする能力を提供します。`Add` トレイト定義でデフォルト型引数を使用することは、ほとんどの場合、追加の引数を指定しなくてもよいことを意味します。つまり、トレイトを使いやすくして、ちょっとだけ実装の定型コードが必要なくなるのです。

最初の目的は 2 番目に似ていますが、逆です: 既存のトレイトに型引数を追加したいなら、既定を与えて、既存の実装コードを破壊せずにトレイトの機能を拡張できるのです。

19.2.3　明確化のためのフルパス記法: 同じ名前のメソッドを呼ぶ

Rust において、別のトレイトのメソッドと同じ名前のメソッドがトレイトにあったり、両方のトレイトを 1 つの型に実装することを妨げるものは何もありません。トレイトのメソッドと同じ名前のメソッドを直接型に実装することも可能です。

同じ名前のメソッドを呼ぶ際、コンパイラにどれを使用したいのか教える必要があるでしょう。両方とも `fly` というメソッドがある 2 つのトレイト、`Pilot` と `Wizard` (訳注 : パイロットと魔法使い) を定義したリスト 19-24 のコードを考えてください。それから両方のトレイトを既に `fly` というメソッドが実装されている型 `Human` (訳注 : 人間) に実装します。各 `fly` メソッドは異なることをします。

ファイル名: src/main.rs

```
trait Pilot {
    fn fly(&self);
}

trait Wizard {
    fn fly(&self);
}
```

```
struct Human;

impl Pilot for Human {
    fn fly(&self) {
        // キャプテンのお言葉
        println!("This is your captain speaking.");
    }
}

impl Wizard for Human {
    fn fly(&self) {
        // 上がれ！
        println!("Up!");
    }
}

impl Human {
    fn fly(&self) {
        // *激しく腕を振る*
        println!("*waving arms furiously*");
    }
}
```

リスト 19-24: 2 つのトレイトに `fly` があるように定義され、`Human` に実装されつつ、`fly` メソッドは `Human` に直接にも実装されている

`Human` のインスタンスに対して `fly` を呼び出すと、コンパイラは型に直接実装されたメソッドを標準で呼び出します。リスト 19-25 のようにですね:

ファイル名: src/main.rs

```
# trait Pilot {
#     fn fly(&self);
# }
#
# trait Wizard {
#     fn fly(&self);
# }
#
# struct Human;
#
# impl Pilot for Human {
#     fn fly(&self) {
#         println!("This is your captain speaking.");
#     }
# }
#
# impl Wizard for Human {
#     fn fly(&self) {
#         println!("Up!");
```

```
#     }
# }
#
# impl Human {
#     fn fly(&self) {
#         println!("*waving arms furiously*");
#     }
# }
#
fn main() {
    let person = Human;
    person.fly();
}
```

リスト 19-25: `Human` のインスタンスに対して `fly` を呼び出す

このコードを実行すると、*waving arms furiously* と出力され、コンパイラが `Human` に直接実装された `fly` メソッドを呼んでいることを示しています。

`Pilot` トレイトか、`Wizard` トレイトの `fly` メソッドを呼ぶためには、より明示的な記法を使用して、どの `fly` メソッドを意図しているか指定する必要があります。リスト 19-26 は、この記法をデモしています。

ファイル名: src/main.rs

```
# trait Pilot {
#     fn fly(&self);
# }
#
# trait Wizard {
#     fn fly(&self);
# }
#
# struct Human;
#
# impl Pilot for Human {
#     fn fly(&self) {
#         println!("This is your captain speaking.");
#     }
# }
#
# impl Wizard for Human {
#     fn fly(&self) {
#         println!("Up!");
#     }
# }
#
# impl Human {
#     fn fly(&self) {
#         println!("*waving arms furiously*");
#     }
```

```
# }
#
fn main() {
    let person = Human;
    Pilot::fly(&person);
    Wizard::fly(&person);
    person.fly();
}
```

リスト 19-26: どのトレイトの fly メソッドを呼び出したいか指定する

メソッド名の前にトレイト名を指定すると、コンパイラにどの fly の実装を呼び出したいか明確化できます。また、Human::fly(&person) と書くこともでき、リスト 19-26 で使用した person.fly() と等価ですが、こちらの方は明確化する必要がないなら、ちょっと記述量が増えます。

このコードを実行すると、こんな出力がされます:

```
This is your captain speaking.
Up!
*waving arms furiously*
```

fly メソッドは self 引数を取るので、1 つの**トレイト**を両方実装する**型**が 2 つあれば、コンパイラには、self の型に基づいてどのトレイトの実装を使うべきかわかるでしょう。

しかしながら、トレイトの一部になる関連関数には self 引数がありません。同じスコープの 2 つの型がそのトレイトを実装する場合、**フルパス記法** (fully qualified syntax) を使用しない限り、どの型を意図しているかコンパイラは推論できません。例えば、リスト 19-27 の Animal トレイトには、関連関数 baby_name 、構造体 Dog の Animal の実装、Dog に直接定義された関連関数 baby_name があります。

ファイル名: src/main.rs

```
trait Animal {
    fn baby_name() -> String;
}

struct Dog;

impl Dog {
    fn baby_name() -> String {
        // スポット(Wikipediaによると、飼い主の事故死後もその人の帰りを待つ忠犬
    の名前の模様)
        String::from("Spot")
    }
}

impl Animal for Dog {
    fn baby_name() -> String {
        // 子犬
```

```
        String::from("puppy")
    }
}

fn main() {
    // 赤ちゃん犬は{}と呼ばれる
    println!("A baby dog is called a {}", Dog::baby_name());
}
```

リスト 19-27: 関連関数のあるトレイトとそのトレイトも実装し、同じ名前の関連関数がある型

このコードは、全ての子犬をスポットと名付けたいアニマル・シェルター (訳注：身寄りのないペットを保護する保健所みたいなところ) 用で、Dog に定義された baby_name 関連関数で実装されています。Dog 型は、トレイト Animal も実装し、このトレイトは全ての動物が持つ特徴を記述します。赤ちゃん犬は子犬と呼ばれ、それが Dog の Animal トレイトの実装の Animal トレイトと紐づいた base_name 関数で表現されています。

main で、Dog::baby_name 関数を呼び出し、直接 Dog に定義された関連関数を呼び出しています。このコードは以下のような出力をします:

```
A baby dog is called a Spot
```

この出力は、欲しかったものではありません。Dog に実装した Animal トレイトの一部の baby_name 関数を呼び出したいので、コードは A baby dog is called a puppy と出力します。リスト 19-26 で使用したトレイト名を指定するテクニックは、ここでは役に立ちません; main をリスト 19-28 のようなコードに変更したら、コンパイルエラーになるでしょう。

ファイル名: src/main.rs

```
fn main() {
    println!("A baby dog is called a {}", Animal::baby_name());
}
```

リスト 19-28: Animal トレイトの baby_name 関数を呼び出そうとするも、コンパイラにはどの実装を使うべきかわからない

Animal::baby_name はメソッドではなく関連関数であり、故に self 引数がないので、どの Animal::baby_name が欲しいのか、コンパイラには推論できません。こんなコンパイルエラーが出るでしょう:

```
error[E0283]: type annotations required: cannot resolve `_: Animal`
(エラー: 型注釈が必要です: `_: Animal`を解決できません)
  --> src/main.rs:20:43
   |
20 |     println!("A baby dog is called a {}", Animal::baby_name());
```

```
    |                                                   ^^^^^^^^^^^^^^^^^
    |
    = note: required by `Animal::baby_name`
    (注釈: `Animal::baby_name`に必要です)
```

Dog に対して Animal 実装を使用したいと明確化し、コンパイラに指示するには、フルパス記法を使う必要があります。リスト 19-29 は、フルパス記法を使用する方法をデモしています。

ファイル名: src/main.rs

```
# trait Animal {
#     fn baby_name() -> String;
# }
#
# struct Dog;
#
# impl Dog {
#     fn baby_name() -> String {
#         String::from("Spot")
#     }
# }
#
# impl Animal for Dog {
#     fn baby_name() -> String {
#         String::from("puppy")
#     }
# }
#
fn main() {
    println!("A baby dog is called a {}", <Dog as Animal>::baby_name());
}
```

リスト 19-29: フルパス記法を使って Dog に実装されているように、Animal トレイトからの baby_name 関数を呼び出したいと指定する

コンパイラに山カッコ内で型注釈を提供し、これは、この関数呼び出しでは Dog 型を Animal として扱いたいと宣言することで、Dog に実装されたように、Animal トレイトの baby_name メソッドを呼び出したいと示唆しています。もうこのコードは、望み通りの出力をします:

```
A baby dog is called a puppy
```

一般的に、フルパス記法は、以下のように定義されています:

```
<Type as Trait>::function(receiver_if_method, next_arg, ...);
```

関連関数では、receiver がないでしょう: 他の引数のリストがあるだけでしょう。関数やメソッドを呼び出す箇所全部で、フルパス記法を使用することもできるでしょうが、プログラムの他の情報か

らコンパイラが推論できるこの記法のどの部分も省略することが許容されています。同じ名前を使用する実装が複数あり、どの実装を呼び出したいかコンパイラが特定するのに助けが必要な場合だけにこのより冗長な記法を使用する必要があるのです。

19.2.4　スーパートレイトを使用して別のトレイト内で、あるトレイトの機能を必要とする

時として、あるトレイトに別のトレイトの機能を使用させる必要がある可能性があります。この場合、依存するトレイトも実装されることを信用する必要があります。信用するトレイトは、実装しているトレイトの**スーパートレイト**です。

例えば、アスタリスクをフレームにする値を出力する `outline_print` メソッドがある `OutlinePrint` トレイトを作りたくなったとしましょう。つまり、`Display` を実装し、`(x, y)` という結果になる `Point` 構造体が与えられて、`x` が `1`、`y` が `3` の `Point` インスタンスに対して `outline_print` を呼び出すと、以下のような出力をするはずです:

```
**********
*        *
* (1, 3) *
*        *
**********
```

`outline_print` の実装では、`Display` トレイトの機能を使用したいです。故に、`Display` も実装する型に対してだけ `OutlinePrint` が動くと指定し、`OutlinePrint` が必要とする機能を提供する必要があるわけです。トレイト定義で `OutlinePrint: Display` と指定することで、そうすることができます。このテクニックは、トレイトにトレイト境界を追加することに似ています。リスト 19-30 は、`OutlinePrint` トレイトの実装を示しています。

ファイル名: src/main.rs

```
use std::fmt;

trait OutlinePrint: fmt::Display {
    fn outline_print(&self) {
        let output = self.to_string();
        let len = output.len();
        println!("{}", "*".repeat(len + 4));
        println!("*{}*", " ".repeat(len + 2));
        println!("* {} *", output);
        println!("*{}*", " ".repeat(len + 2));
        println!("{}", "*".repeat(len + 4));
    }
}
```

リスト 19-30:　`Display` からの機能を必要とする `OutlinePrint` トレイトを実装する

OutlinePrint は Display トレイトを必要とすると指定したので、Display を実装するどんな型にも自動的に実装される to_string 関数を使えます。トレイト名の後にコロンと Display トレイトを追加せずに to_string を使おうとしたら、現在のスコープで型 &Self に to_string というメソッドは存在しないというエラーが出るでしょう。

Display を実装しない型、Point 構造体などに OutlinePrint を実装しようとしたら、何が起きるか確認しましょう:

ファイル名: src/main.rs

```
# trait OutlinePrint {}
struct Point {
    x: i32,
    y: i32,
}

impl OutlinePrint for Point {}
```

Display が必要だけれども、実装されていないというエラーが出ます:

```
error[E0277]: the trait bound `Point: std::fmt::Display` is not satisfied
  --> src/main.rs:20:6
   |
20 | impl OutlinePrint for Point {}
   |      ^^^^^^^^^^^^ `Point` cannot be formatted with the default formatter;
try using `:?` instead if you are using a format string
   |
   = help: the trait `std::fmt::Display` is not implemented for `Point`
```

これを修正するために、Point に Display を実装し、OutlinePrint が必要とする制限を満たします。こんな感じで:

ファイル名: src/main.rs

```
# struct Point {
#     x: i32,
#     y: i32,
# }
#
use std::fmt;

impl fmt::Display for Point {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "({}, {})", self.x, self.y)
    }
}
```

そうすれば、Point に OutlinePrint トレイトを実装してもコンパイルは成功し、Point インスタン

スに対して `outline_print` を呼び出し、アスタリスクのふちの中に表示することができます。

19.2.5　ニュータイプパターンを使用して外部の型に外部のトレイトを実装する

第 10 章の「型にトレイトを実装する」節で、トレイトか型がクレートにローカルな限り、型にトレイトを実装できると述べるオーファンルールについて触れました。**ニュータイプパターン**を使用してこの制限を回避することができ、タプル構造体に新しい型を作成することになります。(タプル構造体については、第 5 章の「異なる型を生成する名前付きフィールドのないタプル構造体を使用する」節で講義しました。) タプル構造体は 1 つのフィールドを持ち、トレイトを実装したい型の薄いラッパになるでしょう。そして、ラッパの型はクレートにローカルなので、トレイトをラッパに実装できます。**ニュータイプ**という用語は、Haskell プログラミング言語に端を発しています。このパターンを使用するのに実行時のパフォーマンスを犠牲にすることはなく、ラッパ型はコンパイル時に省かれます。

例として、`Vec<T>` に `Display` を実装したいとしましょう。`Display` トレイトも `Vec<T>` 型もクレートの外で定義されているので、直接それを行うことはオーファンルールにより妨げられます。`Vec<T>` のインスタンスを保持する `Wrapper` 構造体を作成できます; そして、`Wrapper` に `Display` を実装し、`Vec<T>` 値を使用できます。リスト 19-31 のように。

ファイル名: src/main.rs

```
use std::fmt;

struct Wrapper(Vec<String>);

impl fmt::Display for Wrapper {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "[{}]", self.0.join(", "))
    }
}

fn main() {
    let w = Wrapper(vec![String::from("hello"), String::from("world")]);
    println!("w = {}", w);
}
```

リスト 19-31: `Vec<String>` の周りに `Wrapper` を作成して `Display` を実装する

`Display` の実装は、`self.0` で中身の `Vec<T>` にアクセスしています。`Wrapper` はタプル構造体で、`Vec<T>` がタプルの添え字 0 の要素だからです。それから、`Wrapper` に対して `Display` 型の機能を使用できます。

このテクニックを使用する欠点は、`Wrapper` が新しい型なので、保持している値のメソッドがないことです。`self.0` に委譲して、`Wrapper` を `Vec<T>` と全く同様に扱えるように、`Wrapper` に直接 `Vec<T>` の全てのメソッドを実装しなければならないでしょう。内部の型が持つ全てのメソッドを新しい型に

持たせたいなら、Deref トレイト (第 15 章の「Deref トレイトでスマートポインタを普通の参照のように扱う」節で議論しました) を Wrapper に実装して、内部の型を返すことは解決策の 1 つでしょう。内部の型のメソッド全部を Wrapper 型に持たせたくない (例えば、Wrapper 型の機能を制限するなど) なら、本当に欲しいメソッドだけを手動で実装しなければならないでしょう。

もう、トレイトに関してニュータイプパターンが使用される方法を知りました; トレイトが関連しなくても、有用なパターンでもあります。焦点を変更して、Rust の型システムと相互作用する一部の高度な方法を見ましょう。

19.3　高度な型

Rust の型システムには、この本で触れたけれども、まだ議論していない機能があります。ニュータイプが何故型として有用なのかを調査するため、一般化してニュータイプを議論することから始めます。そして、型エイリアスに移ります。ニュータイプに類似しているけれども、多少異なる意味を持つ機能です。また、! 型と動的サイズ決定型も議論します。

> 注釈: 次の節は、前節「外部の型に外部のトレイトを実装するニュータイプパターン」を読了済みであることを前提にしています。

19.3.1　型安全性と抽象化を求めてニュータイプパターンを使用する

ここまでに議論した以上の作業についてもニュータイプパターンは有用で、静的に絶対に値を混同しないことを強制したり、値の単位を示すことを含みます。ニュータイプを使用して単位を示す例をリスト 19-23 で見かけました: Millimeters と Meters 構造体は、u32 値をニュータイプにラップしていたことを思い出してください。型 Millimeters を引数にする関数を書いたら、誤ってその関数を型 Meters や普通の u32 で呼び出そうとするプログラムはコンパイルできないでしょう。

型の実装の詳細を抽象化する際にニュータイプパターンを使用するでしょう: 例えば、新しい型を直接使用して、利用可能な機能を制限したら、非公開の内部の型の API とは異なる公開 API を新しい型は露出できます。

ニュータイプはまた、内部の実装を隠匿 (いんとく) することもできます。例を挙げれば、People 型を提供して、人の ID と名前を紐づけて格納する HashMap<i32, String> をラップすることができるでしょう。People を使用するコードは、名前の文字列を People コレクションに追加するメソッドなど、提供している公開 API とだけ相互作用するでしょう; そのコードは、内部で i32 ID を名前に代入していることを知る必要はないでしょう。ニュータイプパターンは、カプセル化を実現して実装の詳細を隠匿する軽い方法であり、実装の詳細を隠匿することは、第 17 章の「カプセル化は実装詳細を隠蔽する」節で議論しましたね。

19.3.2　型エイリアスで型同義語を生成する

ニュータイプパターンに付随して、Rust では、既存の型に別の名前を与える**型エイリアス** (type alias: 型別名) を宣言する能力が提供されています。このために、`type` キーワードを使用します。例えば、以下のように `i32` に対して `Kilometers` というエイリアスを作れます。

```
type Kilometers = i32;
```

これで、別名の `Kilometers` は `i32` と**同義語**になりました; リスト 19-23 で生成した `Millimeters` と `Meters` とは異なり、`Kilometers` は個別の新しい型ではありません。型 `Kilometers` の値は、型 `i32` の値と同等に扱われます。

```
type Kilometers = i32;

let x: i32 = 5;
let y: Kilometers = 5;

println!("x + y = {}", x + y);
```

`Kilometers` と `i32` が同じ型なので、両方の型の値を足し合わせたり、`Kilometers` の値を `i32` 引数を取る関数に渡せたりします。ですが、この方策を使用すると、先ほど議論したニュータイプパターンで得られる型チェックの利便性は得られません。

型同義語の主なユースケースは、繰り返しを減らすことです。例えば、こんな感じの長い型があるかもしれません:

```
Box<Fn() + Send + 'static>
```

この長ったらしい型を関数シグニチャや型注釈としてコードのあちこちで記述するのは、面倒で間違いも起きやすいです。リスト 19-32 のそのようなコードで溢れかえったプロジェクトがあることを想像してください。

```
let f: Box<Fn() + Send + 'static> = Box::new(|| println!("hi"));

fn takes_long_type(f: Box<Fn() + Send + 'static>) {
    // --snip--
}

fn returns_long_type() -> Box<Fn() + Send + 'static> {
    // --snip--
#     Box::new(|| ())
}
```

リスト 19-32: 長い型を多くの場所で使用する

型エイリアスは、繰り返しを減らすことでこのコードをより管理しやすくしてくれます。リスト 19-33 で、冗長な型に `Thunk` (注釈：塊) を導入し、その型の使用全部をより短い別名の `Thunk` で置き換えることができます。

```
type Thunk = Box<Fn() + Send + 'static>;

let f: Thunk = Box::new(|| println!("hi"));

fn takes_long_type(f: Thunk) {
    // --snip--
}

fn returns_long_type() -> Thunk {
    // --snip--
#     Box::new(|| ())
}
```

リスト 19-33: 型エイリアスの `Thunk` を導入して繰り返しを減らす

このコードの方が遥かに読み書きしやすいです！ 型エイリアスに意味のある名前を選択すると、意図を伝えるのにも役に立つことがあります (**thunk** は後ほど評価されるコードのための単語なので、格納されるクロージャーには適切な名前です)。

型エイリアスは、繰り返しを減らすために `Result<T, E>` 型ともよく使用されます。標準ライブラリの `std::io` モジュールを考えてください。I/O 処理はしばしば、`Result<T, E>` を返して処理がうまく動かなかった時を扱います。このライブラリには、全ての可能性のある I/O エラーを表す `std::io::Error` 構造体があります。`std::io` の関数の多くは、`Write` トレイトの以下の関数のように `E` が `std::io::Error` の `Result<T, E>` を返すでしょう:

```
use std::io::Error;
use std::fmt;

pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize, Error>;
    fn flush(&mut self) -> Result<(), Error>;

    fn write_all(&mut self, buf: &[u8]) -> Result<(), Error>;
    fn write_fmt(&mut self, fmt: fmt::Arguments) -> Result<(), Error>;
}
```

`Result<..., Error>` が何度も繰り返されてます。そんな状態なので、`std::io` にはこんな類のエイリアス宣言があります:

```
type Result<T> = Result<T, std::io::Error>;
```

この宣言は `std::io` モジュール内にあるので、フルパスエイリアスの `std::io::Result<T>` を使用

できます。つまり、E が std::io::Error で埋められた Result<T, E> です。その結果、Write トレイトの関数シグニチャは、以下のような見た目になります:

```
pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    fn write_all(&mut self, buf: &[u8]) -> Result<()>;
    fn write_fmt(&mut self, fmt: Arguments) -> Result<()>;
}
```

型エイリアスは、2 通りの方法で役に立っています: コードを書きやすくすることと std::io を通して首尾一貫したインターフェイスを与えてくれることです。別名なので、ただの Result<T, E> であり、要するに Result<T, E> に対して動くメソッドはなんでも使えるし、? 演算子のような特殊な記法も使えます。

19.3.3　never 型は絶対に返らない

Rust には、! という名前の特別な型があります。それは型理論の専門用語では **Empty 型**と呼ばれ値なしを表します。私たちは、関数が値を返すことが決して (never) ない時に戻り値の型を記す場所に使われるので、**never type**(訳注：日本語にはできないので、never 型と呼ぶしかないか) と呼ぶのが好きです。こちらが例です:

```
fn bar() -> ! {
    // --snip--
}
```

このコードは、「関数 bar は never を返す」と解読します。never を返す関数は、**発散する関数** (diverging function) と呼ばれます。型 ! の値は生成できないので、bar からリターンする（呼び出し元に制御を戻す）ことは決してできません。

ですが、値を絶対に生成できない型をどう使用するのでしょうか？ リスト 2-5 のコードを思い出してください; リスト 19-34 に一部を再掲します。

```
# let guess = "3";
# loop {
let guess: u32 = match guess.trim().parse() {
    Ok(num) => num,
    Err(_) => continue,
};
# break;
# }
```

リスト 19-34: continue になるアームがある match

この時点では、このコードの詳細の一部を飛ばしました。第 6 章の「match 制御フロー演算子」節で、match アームは全て同じ型を返さなければならないと議論しました。従って、例えば以下のコードは動きません:

```
let guess = match guess.trim().parse() {
    Ok(_) => 5,
    Err(_) => "hello",
}
```

このコードの guess は整数**かつ**文字列にならなければならないでしょうが、Rust では、guess は 1 つの型にしかならないことを要求されます。では、continue は何を返すのでしょうか？ どうやってリスト 19-34 で 1 つのアームからは u32 を返し、別のアームでは、continue で終わっていたのでしょうか？

もうお気付きかもしれませんが、continue は ! 値です。つまり、コンパイラが guess の型を計算する時、両方の match アームを見て、前者は u32 の値、後者は ! 値となります。! は絶対に値を持ち得ないので、コンパイラは、guess の型は u32 と決定するのです。

この振る舞いを解説する公式の方法は、型 ! の式は、他のどんな型にも型強制され得るということです。この match アームを continue で終えることができます。何故なら、continue は値を返さないからです; その代わりに制御をループの冒頭に戻すので、Err の場合、guess には絶対に値を代入しないのです。

never 型は、panic! マクロとも有用です。Option<T> 値に対して呼び出して、値かパニックを生成した unwrap 関数を覚えていますか？ こちらがその定義です:

```
impl<T> Option<T> {
    pub fn unwrap(self) -> T {
        match self {
            Some(val) => val,
            None => panic!("called `Option::unwrap()` on a `None` value"),
        }
    }
}
```

このコードにおいて、リスト 19-34 の match と同じことが起きています: コンパイラは、val の型は T で、panic! の型は ! なので、match 式全体の結果は T と確認します。panic! は値を生成しないので、このコードは動きます。つまり、プログラムを終了するのです。None の場合、unwrap から値は返さないので、このコードは合法なのです。

型が ! の最後の式は、loop です:

```
// 永遠に
print!("forever ");

loop {
    // さらに永遠に
```

```
    print!("and ever ");
}
```

ここで、ループは終わりませんので、! が式の値です。ところが、break を含んでいたら、これは真実にはならないでしょう。break に到達した際にループが終了してしまうからです。

19.3.4　動的サイズ決定型と Sized トレイト

コンパイラが特定の型の値 1 つにどれくらいのスペースのメモリを確保するのかなどの特定の詳細を知る必要があるために、Rust の型システムには混乱を招きやすい細かな仕様があります: **動的サイズ決定型**の概念です。時として **DST** や**サイズなし型**とも称され、これらの型により、実行時にしかサイズを知ることのできない値を使用するコードを書かせてくれます。

str と呼ばれる動的サイズ決定型の詳細を深掘りしましょう。本を通して使用してきましたね。そうです。&str ではなく、str は単独で DST なのです。実行時までは文字列の長さを知ることができず、これは、型 str の変数を生成したり、型 str を引数に取ることはできないことを意味します。動かない以下のコードを考えてください:

```
// こんにちは
let s1: str = "Hello there!";
// 調子はどう?
let s2: str = "How's it going?";
```

コンパイラは、特定の型のどんな値に対しても確保するメモリ量を知る必要があり、ある型の値は全て同じ量のメモリを使用しなければなりません。Rust でこのコードを書くことが許容されたら、これら 2 つの str 値は、同じ量のスペースを消費する必要があったでしょう。ですが、長さが異なります: s1 は、12 バイトのストレージが必要で、s2 は 15 バイトです。このため、動的サイズ決定型を保持する変数を生成することはできないのです。

では、どうすればいいのでしょうか？ この場合、もう答えはご存知です: s1 と s2 の型を str ではなく、&str にすればいいのです。第 4 章の「文字列スライス」節でスライスデータ構造は、開始地点とスライスの長さを格納していると述べたことを思い出してください。

従って、&T は、T がどこにあるかのメモリアドレスを格納する単独の値だけれども、&str は **2 つ**の値なのです: str のアドレスとその長さです。そのため、コンパイル時に &str のサイズを知ることができます: usize の長さの 2 倍です。要するに、参照している文字列の長さによらず、常に &str のサイズがわかります。通常、このようにして Rust では動的サイズ決定型が使用されます: 動的情報のサイズを格納する追加のちょっとしたメタデータがあるのです。動的サイズ決定型の黄金規則は、常に動的サイズ決定型の値をなんらかの種類のポインタの背後に配置しなければならないということです。

str を全ての種類のポインタと組み合わせられます: 例を挙げれば、Box<str> や Rc<str> などです。実際、これまでに見かけましたが、異なる動的サイズ決定型でした: トレイトです。全てのトレ

イトは、トレイト名を使用して参照できる動的サイズ決定型です。第 17 章の「トレイトオブジェクトで異なる型の値を許容する」節で、トレイトをトレイトオブジェクトとして使用するには、`&Trait` や `Box<Trait>` (`Rc<Trait>` も動くでしょう) など、ポインタの背後に配置しなければならないことに触れました。

DST を扱うために、Rust には `Sized` トレイトと呼ばれる特定のトレイトがあり、型のサイズがコンパイル時にわかるかどうかを決定します。このトレイトは、コンパイル時にサイズの判明する全てのものに自動的に実装されます。加えて、コンパイラは暗黙的に全てのジェネリックな関数に `Sized` の境界を追加します。つまり、こんな感じのジェネリック関数定義は:

```
fn generic<T>(t: T) {
    // --snip--
}
```

実際にはこう書いたかのように扱われます:

```
fn generic<T: Sized>(t: T) {
    // --snip--
}
```

既定では、ジェネリック関数はコンパイル時に判明するサイズがある型に対してのみ動きます。ですが、以下の特別な記法を用いてこの制限を緩めることができます:

```
fn generic<T: ?Sized>(t: &T) {
    // --snip--
}
```

`?Sized` のトレイト境界は、`Sized` のトレイト境界の逆になります: これを「`T` は `Sized` かもしれないし、違うかもしれない」と解読するでしょう。この記法は、`Sized` にのみ利用可能で、他のトレイトにはありません。

また、`t` 引数の型を `T` から `&T` に切り替えたことにも注目してください。型は `Sized` でない可能性があるので、なんらかのポインタの背後に使用する必要があるのです。今回は、参照を選択しました。

次は、関数とクロージャについて語ります！

19.4　高度な関数とクロージャ

最後に関数とクロージャに関連する高度な機能の一部を探究し、これには関数ポインタとクロージャの返却が含まれます。

19.4.1　関数ポインタ

クロージャを関数に渡す方法について語りました; 普通の関数を関数に渡すこともできるのです！新しいクロージャを定義するのではなく、既に定義した関数を渡したい時にこのテクニックは有用です。これを関数ポインタで行うと、関数を引数として他の関数に渡して使用できます。関数は、型 `fn` (小文字の f です) に型強制されます。`Fn` クロージャトレイトと混同すべきではありません。`fn` 型は、**関数ポインタ**と呼ばれます。引数が関数ポインタであると指定する記法は、クロージャのものと似ています。リスト 19-35 のように。

ファイル名: src/main.rs

```
fn add_one(x: i32) -> i32 {
    x + 1
}

fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}

fn main() {
    let answer = do_twice(add_one, 5);

    // 答えは{}
    println!("The answer is: {}", answer);
}
```

リスト 19-35:　`fn` 型を使用して引数として関数ポインタを受け入れる

このコードは、`The answer is: 12` と出力します。`do_twice` の引数 `f` は、型 `i32` の 1 つの引数を取り、`i32` を返す `fn` と指定しています。それから、`do_twice` の本体で `f` を呼び出すことができます。`main` では、関数名の `add_one` を最初の引数として `do_twice` に渡せます。

クロージャと異なり、`fn` はトレイトではなく型なので、トレイト境界として `Fn` トレイトの 1 つでジェネリックな型引数を宣言するのではなく、直接 `fn` を引数の型として指定します。

関数ポインタは、クロージャトレイト 3 つ全て (`Fn` 、`FnMut` 、`FnOnce`) を実装するので、常に関数ポインタを引数として、クロージャを期待する関数に渡すことができます。関数が関数とクロージャどちらも受け入れられるように、ジェネリックな型とクロージャトレイトの 1 つを使用して関数を書くのが最善です。

クロージャではなく `fn` だけを受け入れたくなる箇所の一例は、クロージャのない外部コードとのインターフェイスです: C 関数は引数として関数を受け入れられますが、C にはクロージャがありません。

インラインでクロージャが定義されるか、名前付きの関数を使用できるであろう箇所の例として、`map` の使用に目を向けましょう。`map` 関数を使用して数字のベクタを文字列のベクタに変換するには、

このようにクロージャを使用できるでしょう:

```
let list_of_numbers = vec![1, 2, 3];
let list_of_strings: Vec<String> = list_of_numbers
    .iter()
    .map(|i| i.to_string())
    .collect();
```

あるいは、このようにクロージャの代わりに map に引数として関数を名指しできるでしょう:

```
let list_of_numbers = vec![1, 2, 3];
let list_of_strings: Vec<String> = list_of_numbers
    .iter()
    .map(ToString::to_string)
    .collect();
```

先ほど「高度なトレイト」節で語ったフルパス記法を使わなければならないことに注意してください。というのも、to_string という利用可能な関数は複数あるからです。ここでは、ToString トレイトで定義された to_string 関数を使用していて、このトレイトは標準ライブラリが、Display を実装するあらゆる型に実装しています。

このスタイルを好む方もいますし、クロージャを使うのを好む方もいます。どちらも結果的に同じコードにコンパイルされるので、どちらでも、自分にとって明確な方を使用してください。

19.4.2　クロージャを返却する

クロージャはトレイトによって表現されます。つまり、クロージャを直接は返却できないのです。トレイトを返却したい可能性のあるほとんどの場合、代わりにトレイトを実装する具体的な型を関数の戻り値として使用できます。ですが、クロージャではそれはできません。返却可能な具体的な型がないからです; 例えば、関数ポインタの fn を戻り値の型として使うことは許容されていません。

以下のコードは、クロージャを直接返そうとしていますが、コンパイルできません:

```
fn returns_closure() -> Fn(i32) -> i32 {
    |x| x + 1
}
```

コンパイルエラーは以下の通りです:

```
error[E0277]: the trait bound `std::ops::Fn(i32) -> i32 + 'static:
std::marker::Sized` is not satisfied
 -->
  |
1 | fn returns_closure() -> Fn(i32) -> i32 {
  |                         ^^^^^^^^^^^^^^ `std::ops::Fn(i32) -> i32 + 'static`
  does not have a constant size known at compile-time
  |
```

```
  = help: the trait `std::marker::Sized` is not implemented for
  `std::ops::Fn(i32) -> i32 + 'static`
  = note: the return type of a function must have a statically known size
```

エラーは、再度 `Sized` トレイトを参照しています！ コンパイラには、クロージャを格納するのに必要なスペースがどれくらいかわからないのです。この問題の解決策は先ほど見かけました。トレイトオブジェクトを使えます:

```
fn returns_closure() -> Box<Fn(i32) -> i32> {
    Box::new(|x| x + 1)
}
```

このコードは、問題なくコンパイルできます。トレイトオブジェクトについて詳しくは、第 17 章の「トレイトオブジェクトで異なる型の値を許容する」節を参照してください。

次は、マクロを見てみましょう！

19.5　マクロ

本全体を通じて `println!` のようなマクロを使用してきましたが、マクロがなんなのかや、どう動いているのかということは完全には探究していませんでした。Rust において、**マクロ**という用語はある機能の集合のことを指します：`macro_rules!` を使った**宣言的 (declarative)** マクロと、3 種類の**手続き的 (procedural)** マクロ：

- 構造体と enum に `derive` 属性を使ったときに追加されるコードを指定する、カスタムの `#[derive]` マクロ
- 任意の要素に使えるカスタムの属性を定義する、属性風のマクロ
- 関数のように見えるが、引数として指定されたトークンに対して作用する関数風のマクロ

です。

それぞれについて一つずつ話していきますが、その前にまず、どうして関数がすでにあるのにマクロなんてものが必要なのか見てみましょう。

19.5.1　マクロと関数の違い

基本的に、マクロは、他のコードを記述するコードを書く術であり、これは**メタプログラミング**として知られています。付録 C で、`derive` 属性を議論し、これは、色々なトレイトの実装を生成してくれるのでした。また、本を通して `println!` や `vec!` マクロを使用してきました。これらのマクロは全て、**展開**され、手で書いたよりも多くのコードを生成します。

メタプログラミングは、書いて管理しなければならないコード量を減らすのに有用で、これは、関数の役目の一つでもあります。ですが、マクロには関数にはない追加の力があります。

関数シグニチャは、関数の引数の数と型を宣言しなければなりません。一方、マクロは可変長の引

数を取れます: `println!("hello")` のように 1 引数で呼んだり、`println!("hello {}", name)` のように 2 引数で呼んだりできるのです。また、マクロは、コンパイラがコードの意味を解釈する前に展開されるので、例えば、与えられた型にトレイトを実装できます。関数ではできません。何故なら、関数は実行時に呼ばれ、トレイトはコンパイル時に実装される必要があるからです。

関数ではなくマクロを実装する欠点は、Rust コードを記述する Rust コードを書いているので、関数定義よりもマクロ定義は複雑になることです。この間接性のために、マクロ定義は一般的に、関数定義よりも、読みにくく、わかりにくく、管理しづらいです。

マクロと関数にはもう一つ、重要な違いがあります: ファイル内で呼び出す**前**にマクロは定義したりスコープに導入しなければなりませんが、一方で関数はどこにでも定義でき、どこでも呼び出せます。

19.5.2　一般的なメタプログラミングのために `macro_rules!` で宣言的なマクロ

Rust において、最もよく使用される形態のマクロは、**宣言的マクロ**です。これらは時として、**例によるマクロ**、**`macro_rules!` マクロ**、あるいはただ単に**マクロ**とも称されます。核となるのは、宣言的マクロは、Rust の `match` 式に似た何かを書けるということです。第 6 章で議論したように、`match` 式は、式を取り、式の結果の値をパターンと比較し、それからマッチしたパターンに紐づいたコードを実行する制御構造です。マクロも、あるコードと紐付けられたパターンと値を比較します。ここで、値とはマクロに渡されたリテラルの Rust のソースコードそのもののこと。パターンがそのソースコードの構造と比較されます。各パターンに紐づいたコードは、それがマッチしたときに、マクロに渡されたコードを置き換えます。これは全て、コンパイル時に起きます。

マクロを定義するには、`macro_rules!` 構文を使用します。`vec!` マクロが定義されている方法を見て、`macro_rules!` を使用する方法を探究しましょう。`vec!` マクロを使用して特定の値で新しいベクタを生成する方法は、第 8 章で講義しました。例えば、以下のマクロは、3 つの整数を持つ新しいベクタを生成します:

```
let v: Vec<u32> = vec![1, 2, 3];
```

また、`vec!` マクロを使用して 2 整数のベクタや、5 つの文字列スライスのベクタなども生成できます。同じことを関数を使って行うことはできません。予め、値の数や型がわかっていないからです。

リスト 19-28 で些 (いささ) か簡略化された `vec!` マクロの定義を見かけましょう。

ファイル名: src/lib.rs

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
```

```
            )*
            temp_vec
        }
    };
}
```

リスト 19-28: `vec!` マクロ定義の簡略化されたバージョン

> 標準ライブラリの `vec!` マクロの実際の定義は、予め正確なメモリ量を確保するコードを含みます。その最適化コードは、ここでは簡略化のために含みません。

`#[macro_export]` 注釈は、マクロを定義しているクレートがスコープに持ち込まれたなら、無条件でこのマクロが利用可能になるべきということを示しています。この注釈がなければ、このマクロはスコープに導入されることができません。

それから、`macro_rules!` でマクロ定義と定義しているマクロの名前をビックリマーク**なしで**始めています。名前はこの場合 `vec` であり、マクロ定義の本体を意味する波括弧が続いています。

`vec!` 本体の構造は、`match` 式の構造に類似しています。ここではパターン `( $( $x:expr ),* )` の 1 つのアーム、`=>` とこのパターンに紐づくコードのブロックが続きます。パターンが合致すれば、紐づいたコードのブロックが発されます。これがこのマクロの唯一のパターンであることを踏まえると、合致する合法的な方法は一つしかありません; それ以外は、全部エラーになるでしょう。より複雑なマクロには、2 つ以上のアームがあるでしょう。

マクロ定義で合法なパターン記法は、第 18 章で講義したパターン記法とは異なります。というのも、マクロのパターンは値ではなく、Rust コードの構造に対してマッチされるからです。リスト 19-28 のパターンの部品がどんな意味か見ていきましょう; マクロパターン記法全ては参考文献をご覧ください。

まず、1 組のカッコがパターン全体を囲んでいます。次にドル記号 (`$`)、そして 1 組のカッコが続き、このかっこは、置き換えるコードで使用するためにかっこ内でパターンにマッチする値をキャプチャします。`$()` の内部には、`$x:expr` があり、これは任意の Rust 式にマッチし、その式に `$x` という名前を与えます。

`$()` に続くカンマは、`$()` にキャプチャされるコードにマッチするコードの後に、区別を意味するリテラルのカンマ文字が現れるという選択肢もあることを示唆しています。`*` は、パターンが `*` の前にあるもの 0 個以上にマッチすることを指定しています。

このマクロを `vec![1, 2, 3];` と呼び出すと、`$x` パターンは、3 つの式 `1`、`2`、`3` で 3 回マッチします。

さて、このアームに紐づくコードの本体のパターンに目を向けましょう: `$()*` 部分内部の `temp_vec.push()` コードは、パターンがマッチした回数に応じて 0 回以上パターン内で `$()` にマッチする箇所ごとに生成されます。`$x` はマッチした式それぞれに置き換えられます。このマクロを `vec![1, 2, 3];` と呼び出すと、このマクロ呼び出しを置き換え、生成されるコードは以下のようになるでしょう:

```
{
    let mut temp_vec = Vec::new();
    temp_vec.push(1);
    temp_vec.push(2);
    temp_vec.push(3);
    temp_vec
}
```

任意の型のあらゆる数の引数を取り、指定した要素を含むベクタを生成するコードを生成できるマクロを定義しました。

`macro_rules!` には、いくつかの奇妙なコーナーケースがあります。将来、Rust には別種の宣言的マクロが登場する予定です。これは、同じように働くけれども、それらのコーナーケースのうちいくらかを修正します。そのアップデート以降、`macro_rules!` は事実上非推奨 (deprecated) となる予定です。この事実と、ほとんどの Rust プログラマーはマクロを**書く**よりも**使う**ことが多いということを考えて、`macro_rules!` についてはこれ以上語らないことにします。もしマクロの書き方についてもっと知りたければ、オンラインのドキュメントや、“The Little Book of Rust Macros”のようなその他のリソースを参照してください。

19.5.3　属性からコードを生成する手続き的マクロ

2 つ目のマクロの形は、**手続き的マクロ**と呼ばれ、より関数のように働きます（そして一種の手続きです）。宣言的マクロがパターンマッチングを行い、マッチしたコードを他のコードで置き換えていたのとは違い、手続き的マクロは、コードを入力として受け取り、そのコードに対して作用し、出力としてコードを生成します。

3 種の手続き的マクロ (カスタムの derive マクロ, 属性風マクロ、関数風マクロ) はみな同じような挙動をします。

手続き的マクロを作る際は、その定義はそれ専用の特殊なクレート内に置かれる必要があります。これは複雑な技術的理由によるものであり、将来的には解消したいです。手続き的マクロを使うと Listing 19-29 のコードのようになります。`some_attribute` がそのマクロを使うためのプレースホルダーです。

ファイル名: src/lib.rs

```
use proc_macro;

#[some_attribute]
pub fn some_name(input: TokenStream) -> TokenStream {
}
```

Listing 19-29: 手続き的マクロの使用例

手続き的マクロを定義する関数は TokenStream を入力として受け取り、TokenStream を出力として生成します。TokenStream 型は Rust に内蔵されている proc_macro クレートで定義されており、トークンの列を表します。ここがマクロの一番重要なところなのですが、マクロが作用するソースコードは、入力の TokenStream へと変換され、マクロが生成するコードが出力の TokenStream なのです。この関数には属性もつけられていますが、これはどの種類の手続き的マクロを作っているのかを指定します。同じクレート内に複数の種類の手続き的マクロを持つことも可能です。

様々な種類の手続き的マクロを見てみましょう。カスタムの derive マクロから始めて、そのあと他の種類との小さな相違点を説明します。

19.5.4　カスタムの derive マクロの書き方

hello_macro という名前のクレートを作成してみましょう。このクレートは、hello_macro という関連関数が 1 つある HelloMacro というトレイトを定義します。クレートの使用者に使用者の型に HelloMacro トレイトを実装することを強制するのではなく、使用者が型を #[derive(HelloMacro)] で注釈して hello_macro 関数の既定の実装を得られるように、手続き的マクロを提供します。既定の実装は、Hello, Macro! My name is TypeName! (訳注：こんにちは、マクロ！ 僕の名前は TypeName だよ！) と出力し、ここで TypeName はこのトレイトが定義されている型の名前です。言い換えると、他のプログラマに我々のクレートを使用して、リスト 19-30 のようなコードを書けるようにするクレートを記述します。

ファイル名: src/main.rs

```
use hello_macro::HelloMacro;
use hello_macro_derive::HelloMacro;

#[derive(HelloMacro)]
struct Pancakes;

fn main() {
    Pancakes::hello_macro();
}
```

リスト 19-30: 我々の手続き的マクロを使用した時にクレートの使用者が書けるようになるコード

このコードは完成したら、Hello, Macro! My name is Pancakes! (Pancakes：ホットケーキ) と出力します。最初の手順は、新しいライブラリクレートを作成することです。このように:

```
$ cargo new hello_macro --lib
```

次に HelloMacro トレイトと関連関数を定義します:

ファイル名: src/lib.rs

```
pub trait HelloMacro {
    fn hello_macro();
}
```

トレイトと関数があります。この時点でクレートの使用者は、以下のように、このトレイトを実装して所望の機能を達成できるでしょう。

```
use hello_macro::HelloMacro;

struct Pancakes;

impl HelloMacro for Pancakes {
    fn hello_macro() {
        println!("Hello, Macro! My name is Pancakes!");
    }
}

fn main() {
    Pancakes::hello_macro();
}
```

しかしながら、使用者は、hello_macro を使用したい型それぞれに実装ブロックを記述する必要があります; この作業をしなくても済むようにしたいです。

さらに、まだ hello_macro 関数にトレイトが実装されている型の名前を出力する既定の実装を提供することはできません: Rust にはリフレクションの能力がないので、型の名前を実行時に検索することができないのです。コンパイル時にコード生成するマクロが必要です。

> 注釈: リフレクションとは、実行時に型名や関数の中身などを取得する機能のことです。言語によって提供されていたりいなかったりしますが、実行時にメタデータがないと取得できないので、Rust や C++ のようなアセンブリコードに翻訳され、パフォーマンスを要求される高級言語では、提供されないのが一般的と思われます。

次の手順は、手続き的マクロを定義することです。これを執筆している時点では、手続き的マクロは、独自のクレートに存在する必要があります。最終的には、この制限は持ち上げられる可能性があります。クレートとマクロクレートを構成する慣習は以下の通りです: foo というクレートに対して、カスタムの derive 手続き的マクロクレートは foo_derive と呼ばれます。hello_macro プロジェクト内に、hello_macro_derive と呼ばれる新しいクレートを開始しましょう:

```
$ cargo new hello_macro_derive --lib
```

2 つのクレートは緊密に関係しているので、hello_macro クレートのディレクトリ内に手続き的マクロクレートを作成しています。hello_macro のトレイト定義を変更したら、hello_macro_derive の手続き的マクロの実装も変更しなければならないでしょう。2 つのクレートは個別に公開される必要

があり、これらのクレートを使用するプログラマは、両方を依存に追加し、スコープに導入する必要があるでしょう。hello_macro クレートに依存として、hello_macro_derive を使用させ、手続き的マクロのコードを再エクスポートすることもできるかもしれませんが、このようなプロジェクトの構造にすることで、プログラマが derive 機能を使用したくなくても、hello_macro を使用することが可能になります。

hello_macro_derive クレートを手続き的マクロクレートとして宣言する必要があります。また、すぐにわかるように、syn と quote クレートの機能も必要になるので、依存として追加する必要があります。以下を hello_macro_derive の **Cargo.toml** ファイルに追加してください:

ファイル名: hello_macro_derive/Cargo.toml

```
[lib]
proc-macro = true

[dependencies]
syn = "1.0"
quote = "1.0"
```

手続き的マクロの定義を開始するために、hello_macro_derive クレートの **src/lib.rs** ファイルにリスト 19-31 のコードを配置してください。impl_hello_macro 関数の定義を追加するまでこのコードはコンパイルできないことに注意してください。

ファイル名: hello_macro_derive/src/lib.rs

```
extern crate proc_macro;

use proc_macro::TokenStream;
use quote::quote;
use syn;

#[proc_macro_derive(HelloMacro)]
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {
    // 操作可能な構文木としてのRustコードの表現を構築する
    // Construct a representation of Rust code as a syntax tree
    // that we can manipulate
    let ast = syn::parse(input).unwrap();

    // トレイトの実装内容を構築
    // Build the trait implementation
    impl_hello_macro(&ast)
}
```

リスト 19-31: Rust コードを処理するためにほとんどの手続き的マクロクレートに必要になるコード

TokenStream をパースする役割を持つ hello_macro_derive 関数と、構文木を変換する役割を持つ

impl_hello_macro 関数にコードを分割したことに注目してください：これにより手続き的マクロを書くのがより簡単になります。外側の関数（今回だと hello_macro_derive ）のコードは、あなたが見かけたり作ったりするであろうほとんどすべての手続き的マクロのクレートで同じです。内側の関数（今回だと impl_hello_macro ）の内部に書き込まれるコードは、手続き的マクロの目的によって異なってくるでしょう。

3 つの新しいクレートを導入しました: proc_macro 、syn 、quote です。proc_macro クレートは、Rust に付随してくるので、**Cargo.toml** の依存に追加する必要はありませんでした。proc_macro クレートはコンパイラの API で、私達のコードから Rust のコードを読んだり操作したりすることを可能にします。

syn クレートは、文字列から Rust コードを構文解析し、処理を行えるデータ構造にします。quote クレートは、syn データ構造を取り、Rust コードに変換し直します。これらのクレートにより、扱いたい可能性のあるあらゆる種類の Rust コードを構文解析するのがはるかに単純になります: Rust コードの完全なパーサを書くのは、単純な作業ではないのです。

hello_macro_derive 関数は、ライブラリの使用者が型に #[derive(HelloMacro)] を指定した時に呼び出されます。それが可能な理由は、ここで hello_macro_derive 関数を proc_macro_derive で注釈し、トレイト名に一致する HelloMacro を指定したからです; これは、ほとんどの手続き的マクロが倣う慣習です。

この関数はまず、TokenStream からの input をデータ構造に変換し、解釈したり操作したりできるようにします。ここで syn が登場します。syn の parse 関数は TokenStream を受け取り、パースされた Rust のコードを表現する DeriveInput 構造体を返します。Listing 19-32 は struct Pancakes; という文字列をパースすることで得られる DeriveInput 構造体の関係ある部分を表しています。

```
DeriveInput {
    // --snip--

    ident: Ident {
        ident: "Pancakes",
        span: #0 bytes(95..103)
    },
    data: Struct(
        DataStruct {
            struct_token: Struct,
            fields: Unit,
            semi_token: Some(
                Semi
            )
        }
    )
}
```

Listing 19-32: このマクロを使った属性を持つ Listing 19-30 のコードをパースしたときに得られる DeriveInput インスタンス

この構造体のフィールドは、構文解析した Rust コードが Pancakes という ident (識別子、つまり名前) のユニット構造体であることを示しています。この構造体には Rust コードのあらゆる部分を記述するフィールドがもっと多くあります; DeriveInput の syn ドキュメンテーションで詳細を確認してください。

まもなく impl_hello_macro 関数を定義し、そこにインクルードしたい新しい Rust コードを構築します。でもその前に、私達の derive マクロのための出力もまた TokenStream であることに注目してください。返された TokenStream をクレートの使用者が書いたコードに追加しているので、クレートをコンパイルすると、我々が修正した TokenStream で提供している追加の機能を得られます。

ここで、unwrap を呼び出すことで、syn::parse 関数が失敗したときに hello_macro_derive 関数をパニックさせていることにお気付きかもしれません。エラー時にパニックするのは、手続き的マクロコードでは必要なことです。何故なら、proc_macro_derive 関数は、手続き的マクロの API に従うために、Result ではなく TokenStream を返さなければならないからです。この例については、unwrap を使用して簡略化することを選択しました; プロダクションコードでは、panic! か expect を使用して何が間違っていたのかより具体的なエラーメッセージを提供すべきです。

今や、TokenStream からの注釈された Rust コードを DeriveInput インスタンスに変換するコードができたので、Listing 19-33 のように、注釈された型に HelloMacro トレイトを実装するコードを生成しましょう:

ファイル名: hello_macro_derive/src/lib.rs

```
# extern crate proc_macro;
#
# use proc_macro::TokenStream;
# use quote::quote;
# use syn;
#
# #[proc_macro_derive(HelloMacro)]
# pub fn hello_macro_derive(input: TokenStream) -> TokenStream {
#     // Construct a representation of Rust code as a syntax tree
#     // that we can manipulate
#     let ast = syn::parse(input).unwrap();
#
#     // Build the trait implementation
#     impl_hello_macro(&ast)
# }
#
fn impl_hello_macro(ast: &syn::DeriveInput) -> TokenStream {
    let name = &ast.ident;
    let gen = quote! {
        impl HelloMacro for #name {
            fn hello_macro() {
                println!("Hello, Macro! My name is {}!", stringify!(#name));
            }
        }
    };
```

```
    gen.into()
}
```

Listing 19-33: パースされた Rust コードを用いて `HelloMacro` トレイトを実装する

`ast.ident` を使って、注釈された型の名前 (識別子) を含む `Ident` 構造体インスタンスを得ています。Listing 19-32 の構造体を見ると、`impl_hello_macro` 関数を Listing 19-30 のコードに実行したときに私達の得る `ident` は、フィールド `ident` の値として `"Pancakes"` を持つだろうとわかります。従って、Listing 19-33 における変数 `name` は構造体 `Ident` のインスタンスをもちます。このインスタンスは、print された時は文字列 `"Pancakes"` 、即ち Listing 19-30 の構造体の名前となります。

`quote!` マクロを使うことで、私達が返したい Rust コードを定義することができます。ただ、コンパイラが期待しているものは `quote!` マクロの実行結果とはちょっと違うものです。なので、`TokenStream` に変換してやる必要があります。マクロの出力する直接表現を受け取り、必要とされている `TokenStream` 型の値を返す `into` メソッドを呼ぶことでこれを行います。

このマクロはまた、非常にかっこいいテンプレート機構も提供してくれます; `#name` と書くと、`quote!` はそれを `name` という変数の値と置き換えます。普通のマクロが動作するのと似た繰り返しさえ行えます。本格的に入門したいなら、`quote` クレートの doc をご確認ください。

手続き的マクロには使用者が注釈した型に対して `HelloMacro` トレイトの実装を生成してほしいですが、これは `#name` を使用することで得られます。トレイトの実装には 1 つの関数 `hello_macro` があり、この本体に提供したい機能が含まれています: `Hello, Macro! My name is` 、そして、注釈した型の名前を出力する機能です。

ここで使用した `stringify!` マクロは、言語に組み込まれています。`1 + 2` などのような Rust の式を取り、コンパイル時に `"1 + 2"` のような文字列リテラルにその式を変換します。これは、`format!` や `println!` のような、式を評価し、そしてその結果を `String` に変換するマクロとは異なります。`#name` 入力が文字通り出力されるべき式という可能性もあるので、`stringify!` を使用しています。`stringify!` を使用すると、コンパイル時に `#name` を文字列リテラルに変換することで、メモリ確保しなくても済みます。

この時点で、`cargo build` は `hello_macro` と `hello_macro_derive` の両方で成功するはずです。これらのクレートをリスト 19-30 のコードにフックして、手続き的マクロが動くところを確認しましょう！ `cargo new pancakes` であなたの**プロジェクトの**ディレクトリ（訳注：これまでに作った 2 つのクレート内ではないということ）に新しいバイナリプロジェクトを作成してください。`hello_macro` と `hello_macro_derive` を依存として `pancakes` クレートの **Cargo.toml** に追加する必要があります。自分のバージョンの `hello_macro` と `hello_macro_derive` を crates.io に公開しているなら、普通の依存になるでしょう; そうでなければ、以下のように `path` 依存として指定すればよいです:

```
[dependencies]
hello_macro = { path = "../hello_macro" }
hello_macro_derive = { path = "../hello_macro/hello_macro_derive" }
```

リスト 19-30 のコードを **src/main.rs** に配置し、`cargo run` を実行してください: `Hello, Macro! My name is Pancakes` と出力するはずです。手続き的マクロの `HelloMacro` トレイトの実装は、`pancakes` クレートが実装する必要なく、包含されました; `#[derive(HelloMacro)]` がトレイトの実装を追加したのです。

続いて、他の種類の手続き的マクロがカスタムの derive マクロとどのように異なっているか見てみましょう。

19.5.5　属性風マクロ

属性風マクロはカスタムの derive マクロと似ていますが、`derive` 属性のためのコードを生成するのではなく、新しい属性を作ることができます。また、属性風マクロはよりフレキシブルでもあります：`derive` は構造体と enum にしか使えませんでしたが、属性は関数のような他の要素に対しても使えるのです。属性風マクロを使った例を以下に示しています：web アプリケーションフレームワークを使っているときに、`route` という関数につける属性名があるとします：

```
#[route(GET, "/")]
fn index() {
```

この `#[route]` 属性はそのフレームワークによって手続き的マクロとして定義されたものなのでしょう。マクロを定義する関数のシグネチャは以下のようになっているでしょう：

```
#[proc_macro_attribute]
pub fn route(attr: TokenStream, item: TokenStream) -> TokenStream {
```

ここで、2 つ `TokenStream` 型の引数がありますね。1 つ目は属性の中身：`GET, "/"` に対応しており、2 つ目は属性が付けられた要素の中身に対応しています。今回だと `fn index() {}` と関数の本体の残りですね。

それ以外において、属性風マクロはカスタムの derive マクロと同じ動きをします：クレートタイプとして `proc-macro` を使ってクレートを作り、あなたのほしいコードを生成してくれる関数を実装すればよいです！

19.5.6　関数風マクロ

関数風マクロは、関数呼び出しのように見えるマクロを定義します。これらは、`macro_rules!` マクロのように、関数よりフレキシブルです。たとえば、これらは任意の数の引数を取ることができます。しかし、一般的なメタプログラミングのために `macro_rules!` で宣言的なマクロで話したように、`macro_rules!` マクロは match 風の構文を使ってのみ定義できたのでした。関数風マクロは引数として `TokenStream` をとり、その `TokenStream` を定義に従って操作します。操作には、他の 2 つの手続き的マクロと同じように、Rust コードが使われます。例えば、`sql!` マクロという関数風マクロで、以下のように呼び出されるものを考えてみましょう：

```
let sql = sql!(SELECT * FROM posts WHERE id=1);
```

このマクロは、中に入れられた SQL 文をパースし、それが構文的に正しいことを確かめます。これは `macro_rules!` マクロが可能とするよりも遥かに複雑な処理です。`sql!` マクロは以下のように定義することができるでしょう：

```
#[proc_macro]
pub fn sql(input: TokenStream) -> TokenStream {
```

この定義はカスタムの derive マクロのシグネチャと似ています：カッコの中のトークンを受け取り、生成したいコードを返すのです。

19.6　まとめ

ふう！ あなたがいま手にした Rust の機能はあまり頻繁に使うものではありませんが、非常に特殊な状況ではその存在を思い出すことになるでしょう。たくさんの難しいトピックを紹介しましたが、これは、もしあなたがエラー時の推奨メッセージや他の人のコードでそれらに遭遇した時、その概念と文法を理解できるようになっていてほしいからです。この章を、解決策にたどり着くためのリファレンスとして活用してください。

次は、この本で話してきたすべてのことを実際に使って、もう一つプロジェクトをやってみましょう！

第 20 章

最後のプロジェクト: マルチスレッドの Web サーバを構築する

長い旅でしたが、本の末端に到達しました。この章では、共にもう一つプロジェクトを構築して最後の方の章で講義した概念の一部をデモしつつ、それより前の方で学習した内容を思い出してもらいます。

最後のプロジェクトでは、`hello` と話す Web サーバを作り、Web ブラウザでは、図 20-1 のような見た目になります。

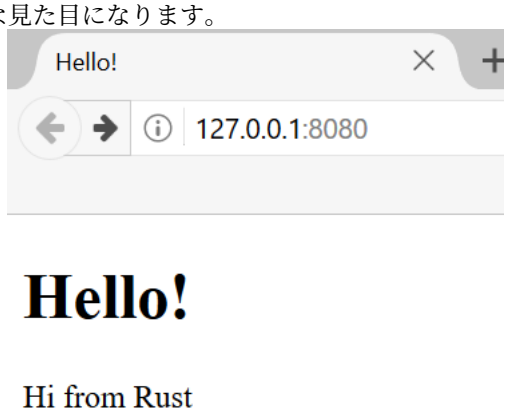


図 20-1: 最後の共有されたプロジェクト

こちらが Web サーバを構築するプランです:

1. TCP と HTTP について少し学ぶ。
2. ソケットで TCP 接続をリッスンする。
3. 少量の HTTP リクエストを構文解析する。
4. 適切な HTTP レスポンスを生成する。

5. スレッドプールでサーバのスループットを強化する。

ですが、取り掛かる前に、ある小さな事実に触れなければなりません: わたしたちがこれから行うやり方は、Rust で Web サーバを構築する最善の方法ではないだろうということです。これから構築するよりもより完全な Web サーバとスレッドプールの実装を提供する製品利用可能な多くのクレートが、**https://crates.io/** で利用可能なのです。

しかしながら、この章での意図は、学習を手助けすることであり、簡単なやり方を選ぶことではありません。Rust はシステムプログラミング言語なので、取りかかる抽象度を選ぶことができ、他の言語で可能だったり実践的だったりするよりも低レベルまで行くことができます。一般的な考えと将来使う可能性のあるクレートの背後にある技術を学べるように、手動で基本的な HTTP サーバとスレッドプールを書きます。

20.1　シングルスレッドの Web サーバを構築する

シングルスレッドの Web サーバを動かすところから始めます。始める前に、Web サーバ構築に関係するプロトコルをさっと一覧しましょう。これらのプロトコルの詳細は、この本の範疇を超えていますが、さっと眺めることで必要な情報が得られるでしょう。

主に 2 つのプロトコルが Web サーバに関係し、**Hypertext Transfer Protocol (HTTP)**(注釈 : ハイパーテキスト転送プロトコル) と、**Transmission Control Protocol (TCP)**(注釈 : 伝送制御プロトコル) です。両者のプロトコルは、**リクエスト・レスポンス**プロトコルであり、つまり、**クライアント**がリクエスト (要求) を初期化し、**サーバ**はリクエストをリッスンし、クライアントにレスポンス (応答) を提供するということです。それらのリクエストとレスポンスの中身は、プロトコルで規定されています。

TCP は、情報がとあるサーバから別のサーバへどう到達するかの詳細を記述するものの、その情報がなんなのかは指定しない、より低レベルのプロトコルです。HTTP はリクエストとレスポンスの中身を定義することで TCP の上に成り立っています。技術的には HTTP を他のプロトコルとともに使用することができますが、大抵の場合、HTTP は TCP の上にデータを送信します。TCP と HTTP のリクエストとレスポンスの生のバイトを取り扱います。

20.1.1　TCP 接続をリッスンする

Web サーバは TCP 接続をリッスンするので、そこが最初に取り掛かる部分になります。標準ライブラリは、`std::net` というこれを行うモジュールを用意しています。通常通り、新しいプロジェクトを作りましょう:

```
$ cargo new hello --bin
     Created binary (application) `hello` project
$ cd hello
```

さて、リスト 20-1 のコードを **src/main.rs** に入力して始めてください。このコードは、TCP ストリームを受信するため `127.0.0.1:7878` というアドレスをリッスンします。入力ストリームを得ると、`Connection established!` と出力します。

ファイル名: src/main.rs

```
use std::net::TcpListener;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        // 接続が確立しました
        println!("Connection established!");
    }
}
```

リスト 20-1: 入力ストリームをリッスンし、ストリームを受け付けた時にメッセージを出力する

`TcpListener` により、アドレス `127.0.0.1:7878` で TCP 接続をリッスンできます。アドレス内で、コロンの前の区域は、自分のコンピュータを表す IP アドレスで (これはどんなコンピュータでも同じで、特に著者のコンピュータを表すわけではありません)、`7878` はポートです。このポートを選択した理由は 2 つあります: HTTP は通常このポートで受け付けられることと、7878 は電話で “rust” と入力されるからです。

この筋書きでの `bind` 関数は、新しい `TcpListener` インスタンスを返すという点で `new` 関数のような働きをします。この関数が `bind` と呼ばれている理由は、ネットワークにおいて、リッスンすべきポートに接続することは、「ポートに束縛する」(binding to a port) こととして知られているからです。

`bind` 関数は `Result<T, E>` を返し、束縛が失敗することもあることを示しています。例えば、ポート 80 に接続するには管理者権限が必要なので (管理者以外はポート 1024 以上しかリッスンできません) 管理者にならずにポート 80 に接続を試みたら、束縛はうまくいかないでしょう。また、別の例として自分のプログラムを 2 つ同時に立ち上げて 2 つのプログラムが同じポートをリッスンしたら、束縛は機能しないでしょう。学習目的のためだけに基本的なサーバを記述しているので、この種のエラーを扱う心配はしません; その代わり、`unwrap` を使用してエラーが発生したら、プログラムを停止します。

`TcpListener` の `incoming` メソッドは、一連のストリームを与えるイテレータを返します (具体的には、型 `TcpStream` のストリーム)。単独の**ストリーム**がクライアント・サーバ間の開かれた接続を表します。**接続** (connection) は、クライアントがサーバに接続し、サーバがレスポンスを生成し、サーバが接続を閉じるというリクエストとレスポンス全体の過程の名前です。そのため、`TcpStream` は自身を読み取って、クライアントが送信したことを確認し、それからレスポンスをストリームに記

述させてくれます。総括すると、この for ループは各接続を順番に処理し、我々が扱えるように一連のストリームを生成します。

とりあえず、ストリームの扱いは、unwrap を呼び出してストリームにエラーがあった場合にプログラムを停止することから構成されています; エラーがなければ、プログラムはメッセージを出力します。次のリストで成功した時にさらに多くの機能を追加します。クライアントがサーバに接続する際に incoming メソッドからエラーを受け取る可能性がある理由は、実際には接続を走査していないからです。代わりに**接続の試行**を走査しています。接続は多くの理由で失敗する可能性があり、そのうちの多くは、OS 特有です。例を挙げれば、多くの OS には、サポートできる同時に開いた接続数に上限があります; 開かれた接続の一部が閉じられるまでその数字を超えた接続の試行はエラーになります。

このコードを試しに実行してみましょう！ 端末で cargo run を呼び出し、それから Web ブラウザで **127.0.0.1:7878** をロードしてください。ブラウザは、「接続がリセットされました」などのエラーメッセージを表示するはずです。サーバが現状、何もデータを返してこないからです。ですが、端末に目を向ければ、ブラウザがサーバに接続した際にいくつかメッセージが出力されるのを目の当たりにするはずです。

```
     Running `target/debug/hello`
Connection established!
Connection established!
Connection established!
```

時々、1 回のブラウザリクエストで複数のメッセージが出力されるのを目の当たりにするでしょう; その理由は、ブラウザがページだけでなく、ブラウザのタブに出現する **favicon.ico** アイコンなどの他のリソースにもリクエストを行なっているということかもしれません。

サーバが何もデータを送り返してこないので、ブラウザがサーバに何度も接続を試みているということである可能性もあるでしょう。stream がスコープを抜け、ループの最後でドロップされると、接続は drop 実装の一部として閉じられます。ブラウザは、再試行することで閉じられた接続を扱うことがあります。問題が一時的なものである可能性があるからです。重要な要素は、TCP 接続へのハンドルを得ることに成功したということです！

特定のバージョンのコードを走らせ終わった時に ctrl-c を押して、プログラムを止めることを忘れないでください。そして、一連のコード変更を行った後に cargo run を再起動し、最新のコードを実行していることを確かめてください。

20.1.2　リクエストを読み取る

ブラウザからリクエストを読み取る機能を実装しましょう！ まず接続を得、それから接続に対して何らかの行動を行う責任を分離するために、接続を処理する新しい関数を開始します。この新しい handle_connection 関数において、TCP ストリームからデータを読み取り、ブラウザからデータが送られていることを確認できるように端末に出力します。コードをリスト 20-2 のように変更してくだ

さい。

ファイル名: src/main.rs

```
use std::io::prelude::*;
use std::net::TcpStream;
use std::net::TcpListener;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        handle_connection(stream);
    }
}

fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 1024];

    stream.read(&mut buffer).unwrap();

    println!("Request: {}", String::from_utf8_lossy(&buffer[..]));
}
```

リスト 20-2: `TcpStream` から読み取り、データを出力する

`std::io::prelude` をスコープに導入して、ストリームから読み書きさせてくれる特定のトレイトにアクセスできるようにしています。`main` 関数内の `for` ループで、接続を確立したというメッセージを出力する代わりに、今では、新しい `handle_connection` 関数を呼び出し、`stream` を渡しています。

`handle_connection` 関数において、`stream` 引数を可変にしました。理由は、`TcpStream` インスタンスが内部で返すデータを追いかけているからです。要求した以上のデータを読み取り、次回データを要求した時のためにそのデータを保存する可能性があります。故に、内部の状態が変化する可能性があるので、`mut` にする必要があるのです; 普通、「読み取り」に可変化は必要ないと考えてしまいますが、この場合、`mut` キーワードが必要です。

次に、実際にストリームから読み取る必要があります。これを 2 つの手順で行います: まず、スタックに読み取ったデータを保持する `buffer` を宣言します。バッファーのサイズは 1024 バイトにしました。これは、基本的なリクエストには十分な大きさでこの章の目的には必要十分です。任意のサイズのリクエストを扱いたければ、バッファーの管理はもっと複雑にする必要があります; 今は、単純に保っておきます。このバッファーを `stream.read` に渡し、これが `TcpStream` からバイトを読み取ってバッファーに置きます。

2 番目にバッファーのバイトを文字列に変換し、その文字列を出力します。`String::from_utf8_lossy` 関数は、`&[u8]` を取り、`String` を生成します。名前の “lossy” の箇所は、無効な

UTF-8 シーケンスを目の当たりにした際のこの関数の振る舞いを示唆しています: 無効なシーケンスを ? 、U+FFFD REPLACEMENT CHARACTER で置き換えます。リクエストデータによって埋められなかったバッファーの部分 (訳注 バッファーとして 1024 バイトの領域を用意しているが、リクエストデータは 1024 バイト存在しないことがほとんどなので変数 buffer の後ろ部分が埋められないまま放置されることを意図していると思われる) は、置換文字が表示される場合があります。

このコードを試しましょう！ プログラムを開始して Web ブラウザで再度リクエストを送ってください。ブラウザではそれでも、エラーページが得られるでしょうが、端末のプログラムの出力はこんな感じになっていることに注目してください:

```
$ cargo run
   Compiling hello v0.1.0 (file:///projects/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.42 secs
     Running `target/debug/hello`
Request: GET / HTTP/1.1
Host: 127.0.0.1:7878
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:52.0) Gecko/20100101
Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
????????????????????????????????????
```

ブラウザによって、少し異なる出力になる可能性があります。今やリクエストデータを出力しているので、Request: GET の後のパスを見ることで 1 回のブラウザリクエストから複数の接続が得られる理由が確認できます。繰り返される接続が全て/ を要求しているなら、ブラウザは、我々のプログラムからレスポンスが得られないので、繰り返し/ をフェッチしようとしていることがわかります。

このリクエストデータを噛み砕いて、ブラウザが我々のプログラムに何を要求しているかを理解しましょう。

20.1.3　HTTP リクエストを詳しく見る

HTTP はテキストベースのプロトコルで、1 つの要求はこのようなフォーマットに則っています:

```
Method Request-URI HTTP-Version CRLF
headers CRLF
message-body
```

1 行目は、クライアントが要求しているものがなんなのかについての情報を保持するリクエスト行です。リクエスト行の最初の部分は使用されている GET や POST などの**メソッド**を示し、これは、どのようにクライアントがこの要求を行なっているかを記述します。クライアントは GET リクエストを使用しました。

リクエスト行の次の部分は/ で、これはクライアントが要求している **Uniform Resource Identi-**

fier (URI)(注釈：統一資源識別子) を示します: URI はほぼ **Uniform Resource Locator (URL)**(注釈：統一資源位置指定子) と同じですが、完全に同じではありません。URI と URL の違いは、この章の目的には重要ではありませんが、HTTP の規格は URI という用語を使用しているので、ここでは脳内で URI を URL と読み替えられます。

最後の部分は、クライアントが使用している HTTP のバージョンで、それからリクエスト行は **CRLF** で終了します。(CRLF は **carriage return** と **line feed**(無理に日本語でいえば、キャリッジ (紙を固定するシリンダー) が戻ることと行を (コンピュータに) 与えること) を表していて、これはタイプライター時代からの用語です！)CRLF は `\r\n` とも表記され、`\r` がキャリッジ・リターンで `\n` がライン・フィードです。CRLF により、リクエスト行がリクエストデータの残りと区別されています。CRLF を出力すると、`\r\n` ではなく、新しい行が開始されることに注意してください。

ここまでプログラムを実行して受け取ったリクエスト行のデータをみると、`GET` がメソッド、`/` が要求 URI、`HTTP/1.1` がバージョンであることが確認できます。

リクエスト行の後に、`Host:` 以下から始まる残りの行は、ヘッダです。`GET` リクエストには、本体（ 訳注：`message-body` のこと）がありません。

試しに他のブラウザからリクエストを送ったり、**127.0.0.1:7878/test** などの異なるアドレスを要求してみて、どうリクエストデータが変わるか確認してください。

さて、ブラウザが要求しているものがわかったので、何かデータを返しましょう！

20.1.4　レスポンスを記述する

さて、クライアントのリクエストに対する返答としてデータの送信を実装します。レスポンスは、以下のようなフォーマットです:

```
HTTP-Version Status-Code Reason-Phrase CRLF
headers CRLF
message-body
```

最初の行は、レスポンスで使用される HTTP バージョン、リクエストの結果を要約する数値ステータス・コード、そしてステータス・コードのテキスト記述を提供する理由句を含む**ステータス行**です。CRLF シーケンスの後には、任意のヘッダ、別の CRLF シーケンス、そしてレスポンスの本体が続きます。

こちらが HTTP バージョン 1.1 を使用し、ステータスコードが 200 で、OK フレーズ、ヘッダと本体なしの例のレスポンスです:

```
HTTP/1.1 200 OK\r\n\r\n
```

ステータスコード 200 は、一般的な成功のレスポンスです。テキストは、矮小 (わいしょう) な成功の HTTP レスポンスです。これを成功したリクエストへの返答としてストリームに書き込みましょう！ `handle_connection` 関数から、リクエストデータを出力していた `println!` を除去し、リスト 20-3 のコードと置き換えてください。

ファイル名: src/main.rs

```
# use std::io::prelude::*;
# use std::net::TcpStream;
fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 1024];

    stream.read(&mut buffer).unwrap();

    let response = "HTTP/1.1 200 OK\r\n\r\n";

    stream.write(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}
```

リスト 20-3: ストリームに矮小な成功の HTTP レスポンスを書き込む

新しい最初の行に成功したメッセージのデータを保持する `response` 変数を定義しています。そして、`response` に対して `as_bytes` を呼び出し、文字列データをバイトに変換します。`stream` の `write` メソッドは、`&[u8]` を取り、接続に直接そのバイトを送信します。

`write` 処理は失敗することもあるので、以前のようにエラーの結果には `unwrap` を使用します。今回も、実際のアプリでは、エラー処理をここに追加するでしょう。最後に `flush` は待機し、バイトが全て接続に書き込まれるまでプログラムが継続するのを防ぎます; `TcpStream` は内部にバッファーを保持して、元となる OS への呼び出しを最小化します。

これらの変更とともに、コードを実行し、リクエストをしましょう。最早、端末にどんなデータも出力していないので、Cargo からの出力以外には何も出力はありません。Web ブラウザで **127.0.0.1:7878** をロードすると、エラーではなく空のページが得られるはずです。HTTP リクエストとレスポンスを手で実装したばかりなのです！

20.1.5　本物の HTML を返す

空のページ以上のものを返す機能を実装しましょう。新しいファイル **hello.html** を **src** ディレクトリではなく、プロジェクトのルートディレクトリに作成してください。お好きなように HTML を書いてください; リスト 20-4 は、一つの可能性を示しています。

ファイル名: hello.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello!</title>
  </head>
  <body>
<!--
```

```
やあ！
-->
    <h1>Hello!</h1>
<!--
Rustからやあ
-->
    <p>Hi from Rust</p>
  </body>
</html>
```

リスト 20-4: レスポンスで返すサンプルの HTML ファイル

これは、ヘッドとテキストのある最低限の HTML5 ドキュメントです。リクエストを受け付けた際にこれをサーバから返すには、リスト 20-5 のように `handle_connection` を変更して HTML ファイルを読み込み、本体としてレスポンスに追加して送ります。

ファイル名: src/main.rs

```
# use std::io::prelude::*;
# use std::net::TcpStream;
use std::fs::File;
// --snip--

fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 1024];
    stream.read(&mut buffer).unwrap();

    let mut file = File::open("hello.html").unwrap();

    let mut contents = String::new();
    file.read_to_string(&mut contents).unwrap();

    let response = format!("HTTP/1.1 200 OK\r\n\r\n{}", contents);

    stream.write(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}
```

リスト 20-5: レスポンスの本体として **hello.html** の中身を送る

先頭に行を追加して標準ライブラリの `File` をスコープに導入しました。ファイルを開き、中身を読み込むコードは、馴染みがあるはずです; リスト 12-4 で I/O プロジェクト用にファイルの中身を読み込んだ時に第 12 章で使用しましたね。

次に `format!` でファイルの中身を成功したレスポンスの本体として追記しています。

このコードを `cargo run` で走らせ、**127.0.0.1:7878** をブラウザでロードしてください; HTML が描画されるのが確認できるはずです！

現時点では、`buffer` 内のリクエストデータは無視し、無条件で HTML ファイルの中身を送り返し

ているだけです。これはつまり、ブラウザで **127.0.0.1:7878/something-else** をリクエストしても、この同じ HTML レスポンスが得られるということです。我々のサーバはかなり限定的で、多くの Web サーバとは異なっています。リクエストに基づいてレスポンスをカスタマイズし、/ への合法なリクエストに対してのみ HTML ファイルを送り返したいです。

20.1.6　リクエストにバリデーションをかけ、選択的にレスポンスを返す

現状、この Web サーバはクライアントが何を要求しても、このファイルの HTML を返します。HTML ファイルを返却する前にブラウザが/ をリクエストしているか確認し、ブラウザが他のものを要求していたらエラーを返す機能を追加しましょう。このために、`handle_connection` をリスト 20-6 のように変更する必要があります。この新しいコードは、/ への要求がどんな見た目になるのか我々が知っていることに対して受け取ったリクエストの中身を検査し、`if` と `else` ブロックを追加して、リクエストを異なる形で扱います。

ファイル名: src/main.rs

```
# use std::io::prelude::*;
# use std::net::TcpStream;
# use std::fs::File;
// --snip--

fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 1024];
    stream.read(&mut buffer).unwrap();

    let get = b"GET / HTTP/1.1\r\n";

    if buffer.starts_with(get) {
        let mut file = File::open("hello.html").unwrap();

        let mut contents = String::new();
        file.read_to_string(&mut contents).unwrap();

        let response = format!("HTTP/1.1 200 OK\r\n\r\n{}", contents);

        stream.write(response.as_bytes()).unwrap();
        stream.flush().unwrap();
    } else {
        // 何か他の要求
        // some other request
    }
}
```

リスト 20-6: リクエストをマッチさせ、/ へのリクエストを他のリクエストとは異なる形で扱う

まず、/ リクエストに対応するデータを `get` 変数にハードコードしています。生のバイトをバッ

ファーに読み込んでいるので、b"" バイト文字列記法を中身のデータの先頭に追記することで、get をバイト文字列に変換しています。そして、buffer が get のバイトから始まっているか確認します。もしそうなら、/ への合法なリクエストを受け取ったことを意味し、これが、HTML ファイルの中身を返す if ブロックで扱う成功した場合になります。

buffer が get のバイトで始まら**ない**のなら、何か他のリクエストを受け取ったことになります。この後すぐ、else ブロックに他のリクエストに対応するコードを追加します。

さあ、このコードを走らせて **127.0.0.1:7878** を要求してください; **hello.html** の HTML が得られるはずです。**127.0.0.1:7878/something-else** などの他のリクエストを行うと、リスト 20-1 や 20-2 のコードを走らせた時に見かけた接続エラーになるでしょう。

では、else ブロックにリスト 20-7 のコードを追記して、ステータスコード 404 のレスポンスを返しましょう。これは、リクエストの中身が見つからなかったことを通知します。エンドユーザへのレスポンスを示し、ページをブラウザに描画するよう、何か HTML も返します。

ファイル名: src/main.rs

```
# use std::io::prelude::*;
# use std::net::TcpStream;
# use std::fs::File;
# fn handle_connection(mut stream: TcpStream) {
# if true {
// --snip--

} else {
    let status_line = "HTTP/1.1 404 NOT FOUND\r\n\r\n";
    let mut file = File::open("404.html").unwrap();
    let mut contents = String::new();

    file.read_to_string(&mut contents).unwrap();

    let response = format!("{}{}", status_line, contents);

    stream.write(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}
# }
```

リスト 20-7: / 以外の何かが要求されたら、ステータスコード 404 とエラーページで応答する

ここでは、レスポンスにはステータスコード 404 と理由フレーズ NOT FOUND のステータス行があります。それでもヘッダは返さず、レスポンスの本体は、ファイル **404.html** の HTML になります。エラーページのために、**hello.html** の隣に **404.html** ファイルを作成する必要があります; 今回も、ご自由にお好きな HTML にしたり、リスト 20-8 の例の HTML を使用したりしてください。

ファイル名: 404.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello!</title>
  </head>
  <body>
<!--
あ あ !
-->
    <h1>Oops!</h1>
<!--
す い ま せ ん 。 要 求 し て い る も の が 理 解 で き ま せ ん
-->
    <p>Sorry, I don't know what you're asking for.</p>
  </body>
</html>
```

リスト 20-8: あらゆる 404 レスポンスでページが送り返す中身のサンプル

これらの変更とともに、もう一度サーバを実行してください。**127.0.0.1:7878** を要求すると、**hello.html** の中身が返り、**127.0.0.1:7878/foo** などの他のリクエストには **404.html** からのエラー HTML が返るはずです。

20.1.7　リファクタリングの触り

現在、`if` と `else` ブロックには多くの繰り返しがあります: どちらもファイルを読み、ファイルの中身をストリームに書き込んでいます。唯一の違いは、ステータス行とファイル名だけです。それらの差異を、ステータス行とファイル名の値を変数に代入する個別の `if` と `else` 行に引っ張り出して、コードをより簡潔にしましょう; そうしたら、それらの変数を無条件にコードで使用し、ファイルを読んでレスポンスを書き込めます。リスト 20-9 は、大きな `if` と `else` ブロックを置き換えた後の結果のコードを示しています。

ファイル名: src/main.rs

```
# use std::io::prelude::*;
# use std::net::TcpStream;
# use std::fs::File;
// --snip--

fn handle_connection(mut stream: TcpStream) {
#     let mut buffer = [0; 1024];
#     stream.read(&mut buffer).unwrap();
#
#     let get = b"GET / HTTP/1.1\r\n";
    // --snip--
```

```
    let (status_line, filename) = if buffer.starts_with(get) {
        ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
    } else {
        ("HTTP/1.1 404 NOT FOUND\r\n\r\n", "404.html")
    };

    let mut file = File::open(filename).unwrap();
    let mut contents = String::new();

    file.read_to_string(&mut contents).unwrap();

    let response = format!("{}{}", status_line, contents);

    stream.write(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}
```

リスト 20-9: 2 つの場合で異なるコードだけを含むように、`if` と `else` ブロックをリファクタリングする

これで、`if` と `else` ブロックは、タプルにステータス行とファイル名の適切な値を返すだけになりました; それから、分配を使用してこれら 2 つの値を第 18 章で議論したように、`let` 文のパターンで `status_line` と `filename` に代入しています。

前は重複していたコードは、今では `if` と `else` ブロックの外に出て、`status_line` と `filename` 変数を使用しています。これにより、2 つの場合の違いがわかりやすくなり、ファイル読み取りとレスポンス記述の動作法を変更したくなった際に、1 箇所だけコードを更新すればいいようになったことを意味します。リスト 20-9 のコードの振る舞いは、リスト 20-8 と同じです。

素晴らしい！ もう、およそ 40 行の Rust コードで、あるリクエストには中身のあるページで応答し、他のあらゆるリクエストには 404 レスポンスで応答する単純な Web サーバができました。

現状、このサーバは、シングルスレッドで実行されます。つまり、1 回に 1 つのリクエストしか捌けないということです。何か遅いリクエストをシミュレーションすることで、それが問題になる可能性を調査しましょう。それから 1 度にサーバが複数のリクエストを扱えるように修正します。

20.2　シングルスレッドサーバをマルチスレッド化する

現状、サーバはリクエストを順番に処理します。つまり、最初の接続が処理し終わるまで、2 番目の接続は処理しないということです。サーバが受け付けるリクエストの量が増えるほど、この連続的な実行は、最適ではなくなるでしょう。サーバが処理するのに長い時間がかかるリクエストを受け付けたら、新しいリクエストは迅速に処理できても、続くリクエストは長いリクエストが完了するまで待たなければならなくなるでしょう。これを修正する必要がありますが、まずは、実際に問題が起こっているところを見ます。

20.2.1　現在のサーバの実装で遅いリクエストをシミュレーションする

処理が遅いリクエストが現在のサーバ実装に対して行われる他のリクエストにどう影響するかに目を向けます。リスト 20-10 は、応答する前に 5 秒サーバをスリープさせる遅いレスポンスをシミュレーションした**/sleep** へのリクエストを扱う実装です。

ファイル名: src/main.rs

```
use std::thread;
use std::time::Duration;
# use std::io::prelude::*;
# use std::net::TcpStream;
# use std::fs::File;
// --snip--

fn handle_connection(mut stream: TcpStream) {
#     let mut buffer = [0; 1024];
#     stream.read(&mut buffer).unwrap();
    // --snip--

    let get = b"GET / HTTP/1.1\r\n";
    let sleep = b"GET /sleep HTTP/1.1\r\n";

    let (status_line, filename) = if buffer.starts_with(get) {
        ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
    } else if buffer.starts_with(sleep) {
        thread::sleep(Duration::from_secs(5));
        ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
    } else {
        ("HTTP/1.1 404 NOT FOUND\r\n\r\n", "404.html")
    };

    // --snip--
}
```

リスト 20-10: **/sleep** を認識して 5 秒間スリープすることで遅いリクエストをシミュレーションする

このコードはちょっと汚いですが、シミュレーション目的には十分です。2 番目のリクエスト `sleep` を作成し、そのデータをサーバは認識します。`if` ブロックの後に `else if` を追加し、**/sleep** へのリクエストを確認しています。そのリクエストが受け付けられると、サーバは成功の HTML ページを描画する前に 5 秒間スリープします。

我々のサーバがどれだけ基礎的か見て取れます: 本物のライブラリは、もっと冗長でない方法で複数のリクエストの認識を扱うでしょう！

`cargo run` でサーバを開始してください。それから 2 つブラウザのウインドウを開いてください:

1 つは、**http://localhost:7878/** 用、そしてもう 1 つは **http://localhost:7878/sleep** 用です。以前のように/ URI を数回入力したら、素早く応答するでしょう。しかし、**/sleep** を入力し、それから/ をロードしたら、sleep がロードする前にきっかり 5 秒スリープし終わるまで、/ は待機するのを目撃するでしょう。

より多くのリクエストが遅いリクエストの背後に回ってしまうのを回避するよう Web サーバが動く方法を変える方法は複数あります; これから実装するのは、スレッドプールです。

20.2.2　スレッドプールでスループットを向上させる

スレッドプールは、待機し、タスクを処理する準備のできた一塊りの大量に生成されたスレッドです。プログラムが新しいタスクを受け取ったら、プールのスレッドのどれかをタスクにあてがい、そのスレッドがそのタスクを処理します。プールの残りのスレッドは、最初のスレッドが処理中にやってくる他のあらゆるタスクを扱うために利用可能です。最初のスレッドがタスクの処理を完了したら、アイドル状態のスレッドプールに戻り、新しいタスクを処理する準備ができます。スレッドプールにより、並行で接続を処理でき、サーバのスループットを向上させます。

プール内のスレッド数は、小さい数字に制限し、DoS(Denial of Service; サービスの拒否) 攻撃から保護します; リクエストが来た度に新しいスレッドをプログラムに生成させたら、1000 万リクエストをサーバに行う誰かが、サーバのリソースを使い尽くし、リクエストの処理を停止に追い込むことで、大混乱を招くことができてしまうでしょう。

無制限にスレッドを大量生産するのではなく、プールに固定された数のスレッドを待機させます。リクエストが来る度に、処理するためにプールに送られます。プールは、やって来るリクエストのキューを管理します。プールの各スレッドがこのキューからリクエストを取り出し、リクエストを処理し、そして、別のリクエストをキューに要求します。この設計により、N リクエストを並行して処理でき、ここで N はスレッド数です。各スレッドが実行に時間のかかるリクエストに応答していたら、続くリクエストはそれでも、キュー内で待機させられてしまうこともありますが、その地点に到達する前に扱える時間のかかるリクエスト数を増加させました。

このテクニックは、Web サーバのスループットを向上させる多くの方法の 1 つに過ぎません。探究する可能性のある他の選択肢は、fork/join モデルと、シングルスレッドの非同期 I/O モデルです。この話題にご興味があれば、他の解決策についてもっと読み、Rust で実装を試みることができます; Rust のような低レベル言語であれば、これらの選択肢全部が可能なのです。

スレッドプールを実装し始める前に、プールを使うのはどんな感じになるはずなのかについて語りましょう。コードの設計を試みる際、クライアントのインターフェイスをまず書くことは、設計を導く手助けになることがあります。呼び出したいように構成されるよう、コードの API を記述してください; そして、機能を実装してから公開 API の設計をするのではなく、その構造内で機能を実装してください。

第 12 章のプロジェクトで TDD を使用したように、ここでは Compiler Driven Development(コンパイラ駆動開発) を使用します。欲しい関数を呼び出すコードを書き、それからコンパイラの出す

エラーを見てコードが動くように次に何を変更すべきかを決定します。

20.2.2.1　各リクエストに対してスレッドを立ち上げられる場合のコードの構造

まず、全接続に対して新しいスレッドを確かに生成した場合にコードがどんな見た目になるかを探究しましょう。先ほど述べたように、無制限にスレッドを大量生産する可能性があるという問題のため、これは最終的な計画ではありませんが、開始点です。リスト 20-11 は、新しいスレッドを立ち上げて `for` ループ内で各ストリームを扱うために `main` に行う変更を示しています。

ファイル名: src/main.rs

```
# use std::thread;
# use std::io::prelude::*;
# use std::net::TcpListener;
# use std::net::TcpStream;
#
fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        thread::spawn(|| {
            handle_connection(stream);
        });
    }
}
# fn handle_connection(mut stream: TcpStream) {}
```

リスト 20-11: 各ストリームに対して新しいスレッドを立ち上げる

第 16 章で学んだように、`thread::spawn` は新しいスレッドを生成し、それからクロージャ内のコードを新しいスレッドで実行します。このコードを実行してブラウザで**/sleep** をロードし、それからもう 2 つのブラウザのタブで/ をロードしたら、確かに/ へのリクエストは、**/sleep** が完了するのを待機しなくても済むことがわかるでしょう。ですが、前述したように、無制限にスレッドを生成することになるので、これは最終的にシステムを参らせてしまうでしょう。

20.2.2.2　有限数のスレッド用に似たインターフェイスを作成する

スレッドからスレッドプールへの変更に API を使用するコードへの大きな変更が必要ないように、スレッドプールには似た、馴染み深い方法で動作してほしいです。リスト 20-12 は、`thread::spawn` の代わりに使用したい `ThreadPool` 構造体の架空のインターフェイスを表示しています。

ファイル名: src/main.rs

```
# use std::thread;
```

```
# use std::io::prelude::*;
# use std::net::TcpListener;
# use std::net::TcpStream;
# struct ThreadPool;
# impl ThreadPool {
#    fn new(size: u32) -> ThreadPool { ThreadPool }
#    fn execute<F>(&self, f: F)
#        where F: FnOnce() + Send + 'static {}
# }
#
fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
    let pool = ThreadPool::new(4);

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
    }
}
# fn handle_connection(mut stream: TcpStream) {}
```

リスト 20-12: `ThreadPool` の理想的なインターフェイス

`ThreadPool::new` を使用して設定可能なスレッド数で新しいスレッドプールを作成し、今回の場合は 4 です。それから `for` ループ内で、`pool.execute` は、プールが各ストリームに対して実行すべきクロージャを受け取るという点で、`thread::spawn` と似たインターフェイスです。`pool.execute` を実装する必要があるので、これはクロージャを取り、実行するためにプール内のスレッドに与えます。このコードはまだコンパイルできませんが、コンパイラがどう修正したらいいかガイドできるように試してみます。

20.2.2.3　コンパイラ駆動開発で `ThreadPool` 構造体を構築する

リスト 20-12 の変更を **src/main.rs** に行い、それから開発を駆動するために `cargo check` からのコンパイラエラーを活用しましょう。こちらが得られる最初のエラーです:

```
$ cargo check
   Compiling hello v0.1.0 (file:///projects/hello)
error[E0433]: failed to resolve. Use of undeclared type or module `ThreadPool`
(エラー: 解決に失敗しました。未定義の型またはモジュール`ThreadPool`を使用しています)
  --> src\main.rs:10:16
   |
10 |     let pool = ThreadPool::new(4);
   |                ^^^^^^^^^^^^^^^ Use of undeclared type or module
   `ThreadPool`
```

```
error: aborting due to previous error
```

よろしい!このエラーは ThreadPool 型かモジュールが必要なことを教えてくれているので、今構築します。ThreadPool の実装は、Web サーバが行う仕事の種類とは独立しています。従って、hello クレートをバイナリクレートからライブラリクレートに切り替え、ThreadPool の実装を保持させましょう。ライブラリクレートに変更後、個別のスレッドプールライブラリを Web リクエストを提供するためだけではなく、スレッドプールでしたいあらゆる作業にも使用できます。

以下を含む **src/lib.rs** を生成してください。これは、現状存在できる最も単純な ThreadPool の定義です:

ファイル名: src/lib.rs

```
pub struct ThreadPool;
```

それから新しいディレクトリ、**src/bin** を作成し、**src/main.rs** に根付くバイナリクレートを **src/bin/main.rs** に移動してください。そうすると、ライブラリクレートが **hello** ディレクトリ内で主要クレートになります; それでも、cargo run で **src/bin/main.rs** のバイナリを実行することはできます。**main.rs** ファイルを移動後、編集してライブラリクレートを持ち込み、以下のコードを **src/bin/main.rs** の先頭に追記して ThreadPool をスコープに導入してください:

ファイル名: src/bin/main.rs

```
extern crate hello;
use hello::ThreadPool;
```

このコードはまだ動きませんが、再度それを確認して扱う必要のある次のエラーを手に入れましょう:

```
$ cargo check
   Compiling hello v0.1.0 (file:///projects/hello)
error[E0599]: no function or associated item named `new` found for type
`hello::ThreadPool` in the current scope
(エラー: 現在のスコープで型`hello::ThreadPool`の関数または関連アイテムに`new`と
    いうものが見つかりません)
 --> src/bin/main.rs:13:16
   |
13 |     let pool = ThreadPool::new(4);
   |                ^^^^^^^^^^^^^^^ function or associated item not found in
   `hello::ThreadPool`
```

このエラーは、次に、ThreadPool に対して new という関連関数を作成する必要があることを示唆しています。また、new には 4 を引数として受け入れる引数 1 つがあり、ThreadPool インスタンスを返すべきということも知っています。それらの特徴を持つ最も単純な new 関数を実装しましょう:

ファイル名: src/lib.rs

```
pub struct ThreadPool;

impl ThreadPool {
    pub fn new(size: usize) -> ThreadPool {
        ThreadPool
    }
}
```

size 引数の型として、usize を選択しました。何故なら、マイナスのスレッド数は、何も筋が通らないことを知っているからです。また、この 4 をスレッドのコレクションの要素数として使用し、第 3 章の「整数型」節で議論したように、これは usize のあるべき姿であることも知っています。

コードを再度確認しましょう:

```
$ cargo check
   Compiling hello v0.1.0 (file:///projects/hello)
warning: unused variable: `size`
(警告: 未使用の変数: `size`)
 --> src/lib.rs:4:16
  |
4 |     pub fn new(size: usize) -> ThreadPool {
  |                ^^^^
  |
  = note: #[warn(unused_variables)] on by default
  = note: to avoid this warning, consider using `_size` instead

error[E0599]: no method named `execute` found for type `hello::ThreadPool` in
    the current scope
  --> src/bin/main.rs:18:14
   |
18 |         pool.execute(|| {
   |              ^^^^^^^
```

今度は、警告とエラーが出ました。一時的に警告は無視して、ThreadPool に execute メソッドがないためにエラーが発生しました。「有限数のスレッド用に似たインターフェイスを作成する」節で我々のスレッドプールは、thread::spawn と似たインターフェイスにするべきと決定したことを思い出してください。さらに、execute 関数を実装するので、与えられたクロージャを取り、実行するようにプールの待機中のスレッドに渡します。

ThreadPool に execute メソッドをクロージャを引数として受け取るように定義します。第 13 章の「ジェネリック引数と Fn トレイトを使用してクロージャを保存する」節から、3 つの異なるトレイトでクロージャを引数として取ることができることを思い出してください: Fn 、FnMut 、FnOnce です。ここでは、どの種類のクロージャを使用するか決定する必要があります。最終的には、標準ライブラリの thread::spawn 実装に似たことをすることがわかっているので、thread::spawn のシグニチャで引数にどんな境界があるか見ることができます。ドキュメンテーションは、以下のものを示してい

ます:

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
    where
        F: FnOnce() -> T + Send + 'static,
        T: Send + 'static
```

F 型引数がここで関心のあるものです; T 型引数は戻り値と関係があり、関心はありません。spawn は、F のトレイト境界として FnOnce を使用していることが確認できます。これはおそらく、我々が欲しているものでもあるでしょう。というのも、最終的には execute で得た引数を spawn に渡すからです。さらに FnOnce は使用したいトレイトであると自信を持つことができます。リクエストを実行するスレッドは、そのリクエストのクロージャを 1 回だけ実行し、これは FnOnce の Once に合致するからです。

F 型引数にはまた、トレイト境界の Send とライフタイム境界の 'static もあり、この状況では有用です: あるスレッドから別のスレッドにクロージャを移動するのに Send が必要で、スレッドの実行にどれくらいかかるかわからないので、'static も必要です。ThreadPool にこれらの境界のジェネリックな型 F の引数を取る execute メソッドを生成しましょう:

ファイル名: src/lib.rs

```
# pub struct ThreadPool;
impl ThreadPool {
    // --snip--

    pub fn execute<F>(&self, f: F)
        where
            F: FnOnce() + Send + 'static
    {

    }
}
```

それでも、FnOnce の後に () を使用しています。この FnOnce は引数を取らず、値も返さないクロージャを表すからです。関数定義同様に、戻り値の型はシグニチャから省略できますが、引数がなくても、カッコは必要です。

またもや、これが execute メソッドの最も単純な実装です: 何もしませんが、コードがコンパイルできるようにしようとしているだけです。再確認しましょう:

```
$ cargo check
   Compiling hello v0.1.0 (file:///projects/hello)
warning: unused variable: `size`
 --> src/lib.rs:4:16
  |
4 |     pub fn new(size: usize) -> ThreadPool {
  |                ^^^^
```

```
  |
  = note: #[warn(unused_variables)] on by default
  = note: to avoid this warning, consider using `_size` instead

warning: unused variable: `f`
 --> src/lib.rs:8:30
  |
8 |     pub fn execute<F>(&self, f: F)
  |                              ^
  |
  = note: to avoid this warning, consider using `_f` instead
```

これで警告を受け取るだけになり、コンパイルできるようになりました！ しかし、cargo run を試して、ブラウザでリクエストを行うと、章の冒頭で見かけたエラーがブラウザに現れることに注意してください。ライブラリは、まだ実際に execute に渡されたクロージャを呼び出していないのです！

> 注釈: Haskell や Rust などの厳密なコンパイラがある言語についての格言として「コードがコンパイルできたら、動作する」というものをお聴きになったことがある可能性があります。ですが、この格言は普遍的に当てはまるものではありません。このプロジェクトはコンパイルできますが、全く何もしません！ 本物の完璧なプロジェクトを構築しようとしているのなら、ここが単体テストを書き始めて、コードがコンパイルでき、**かつ**欲しい振る舞いを保持していることを確認するのに良い機会でしょう。

20.2.2.4　new でスレッド数を検査する

new と execute の引数で何もしていないので、警告が出続けます。欲しい振る舞いでこれらの関数の本体を実装しましょう。まずはじめに、new を考えましょう。先刻、size 引数に非負整数型を選択しました。負のスレッド数のプールは、全く道理が通らないからです。しかしながら、0 スレッドのプールも全く意味がわかりませんが、0 も完全に合法な usize です。ThreadPool インスタンスを返す前に size が 0 よりも大きいことを確認するコードを追加し、リスト 20-13 に示したように、assert! マクロを使用することで 0 を受け取った時にプログラムをパニックさせます。

ファイル名: src/lib.rs

```
# pub struct ThreadPool;
impl ThreadPool {
    /// 新しいThreadPoolを生成する。
    ///
    /// sizeがプールのスレッド数です。
    ///
    /// # パニック
    ///
    /// sizeが0なら、`new`関数はパニックします。
    ///
    /// Create a new ThreadPool.
    ///
```

```
    /// The size is the number of threads in the pool.
    ///
    /// # Panics
    ///
    /// The `new` function will panic if the size is zero.
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        ThreadPool
    }

    // --snip--
}
```

リスト 20-13: `ThreadPool::new` を実装して `size` が 0 ならパニックする

doc comment で `ThreadPool` にドキュメンテーションを追加しました。第 14 章で議論したように、関数がパニックすることもある場面を声高に叫ぶセクションを追加することで、いいドキュメンテーションの実践に倣 (なら) っていることに注意してください。試しに `cargo doc --open` を実行し、`ThreadPool` 構造体をクリックして、`new` の生成されるドキュメンテーションがどんな見た目か確かめてください！

ここでしたように `assert!` マクロを追加する代わりに、リスト 12-9 の I/O プロジェクトの `Config::new` のように、`new` に `Result` を返させることもできるでしょう。しかし、今回の場合、スレッドなしでスレッドプールを作成しようとするのは、回復不能なエラーであるべきと決定しました。野心を感じるのなら、以下のシグニチャの `new` も書いてみて、両者を比較してみてください:

```
pub fn new(size: usize) -> Result<ThreadPool, PoolCreationError> {
```

20.2.2.5　スレッドを格納するスペースを生成する

今や、プールに格納する合法なスレッド数を知る方法ができたので、`ThreadPool` 構造体を返す前にスレッドを作成して格納できます。ですが、どのようにスレッドを「格納」するのでしょうか？ もう一度、`thread::spawn` シグニチャを眺めてみましょう:

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
    where
        F: FnOnce() -> T + Send + 'static,
        T: Send + 'static
```

`spawn` 関数は、`JoinHandle<T>` を返し、ここで `T` は、クロージャが返す型です。試しに同じように `JoinHandle` を使ってみて、どうなるか見てみましょう。我々の場合、スレッドプールに渡すクロージャは接続を扱い、何も返さないので、`T` はユニット型 `()` になるでしょう。

リスト 20-14 のコードはコンパイルできますが、まだスレッドは何も生成しません。`ThreadPool` の

定義を変更して、thread::JoinHandle<()> インスタンスのベクタを保持し、size キャパシティのベクタを初期化し、スレッドを生成する何らかのコードを実行する for ループを設定し、それらを含む ThreadPool インスタンスを返します。

ファイル名: src/lib.rs

```
use std::thread;

pub struct ThreadPool {
    threads: Vec<thread::JoinHandle<()>>,
}

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let mut threads = Vec::with_capacity(size);

        for _ in 0..size {
            // スレッドを生成してベクタに格納する
            // create some threads and store them in the vector
        }

        ThreadPool {
            threads
        }
    }

    // --snip--
}
```

リスト 20-14: ThreadPool にスレッドを保持するベクタを生成する

ライブラリクレート内で std::thread をスコープに導入しました。ThreadPool のベクタの要素の型として、thread::JoinHandle を使用しているからです。

一旦、合法なサイズを受け取ったら、ThreadPool は size 個の要素を保持できる新しいベクタを生成します。この本ではまだ、with_capacity 関数を使用したことがありませんが、これは Vec::new と同じ作業をしつつ、重要な違いがあります: ベクタに予めスペースを確保しておくのです。ベクタに size 個の要素を格納する必要があることはわかっているので、このメモリ確保を前もってしておくと、Vec::new よりも少しだけ効率的になります。Vec::new は、要素が挿入されるにつれて、自身のサイズを変更します。

再び cargo check を実行すると、もういくつか警告が出るものの、成功するはずです。

20.2.2.6　ThreadPool からスレッドにコードを送信する責任を負う Worker 構造体

リスト 20-14 の for ループにスレッドの生成に関するコメントを残しました。ここでは、実際にスレッドを生成する方法に目を向けます。標準ライブラリはスレッドを生成する手段として thread::spawn を提供し、thread::spawn は、生成されるとすぐにスレッドが実行すべき何らかのコードを得ることを予期します。ところが、我々の場合、スレッドを生成して、後ほど送信するコードを**待機**してほしいです。標準ライブラリのスレッドの実装は、それをするいかなる方法も含んでいません; それを手動で実装しなければなりません。

この新しい振る舞いを管理するスレッドと ThreadPool 間に新しいデータ構造を導入することでこの振る舞いを実装します。このデータ構造を Worker と呼び、プール実装では一般的な用語です。レストランのキッチンで働く人々を思い浮かべてください: 労働者は、お客さんからオーダーが来るまで待機し、それからそれらのオーダーを取り、満たすことに責任を負います。

スレッドプールに JoinHandle<()> インスタンスのベクタを格納する代わりに、Worker 構造体のインスタンスを格納します。各 Worker が単独の JoinHandle<()> インスタンスを格納します。そして、Worker に実行するコードのクロージャを取り、既に走っているスレッドに実行してもらうために送信するメソッドを実装します。ログを取ったり、デバッグする際にプールの異なるワーカーを区別できるように、各ワーカーに id も付与します。

ThreadPool を生成する際に発生することに以下の変更を加えましょう。このように Worker をセットアップした後に、スレッドにクロージャを送信するコードを実装します:

1. id と JoinHandle<()> を保持する Worker 構造体を定義する。
2. ThreadPool を変更し、Worker インスタンスのベクタを保持する。
3. id 番号を取り、id と空のクロージャで大量生産されるスレッドを保持する Worker インスタンスを返す Worker::new 関数を定義する。
4. ThreadPool::new で for ループカウンタを使用して id を生成し、その id で新しい Worker を生成し、ベクタにワーカーを格納する。

挑戦に積極的ならば、リスト 20-15 のコードを見る前にご自身でこれらの変更を実装してみてください。

いいですか？ こちらが先ほどの変更を行う 1 つの方法を行ったリスト 20-15 です。

ファイル名: src/lib.rs

```
use std::thread;

pub struct ThreadPool {
    workers: Vec<Worker>,
}

impl ThreadPool {
    // --snip--
```

```
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id));
        }

        ThreadPool {
            workers
        }
    }
    // --snip--
}

struct Worker {
    id: usize,
    thread: thread::JoinHandle<()>,
}

impl Worker {
    fn new(id: usize) -> Worker {
        let thread = thread::spawn(|| {});

        Worker {
            id,
            thread,
        }
    }
}
```

リスト 20-15: `ThreadPool` を変更してスレッドを直接保持するのではなく、`Worker` インスタンスを保持する

`ThreadPool` のフィールド名を `threads` から `workers` に変更しました。`JoinHandle<()>` インスタンスではなく、`Worker` インスタンスを保持するようになったからです。`for` ループのカウンタを `Worker::new` への引数として使用し、それぞれの新しい `Worker` を `workers` というベクタに格納します。

外部のコード **(src/bin/main.rs** のサーバなど**)** は、`ThreadPool` 内で `Worker` 構造体を使用していることに関する実装の詳細を知る必要はないので、`Worker` 構造体とその `new` 関数は非公開にしています。`Worker::new` 関数は与えた `id` を使用し、空のクロージャを使って新しいスレッドを立ち上げることで生成される `JoinHandle<()>` インスタンスを格納します。

このコードはコンパイルでき、`ThreadPool::new` への引数として指定した数の `Worker` インスタンスを格納します。ですが**それでも**、`execute` で得るクロージャを処理してはいません。次は、それをする方法に目を向けましょう。

20.2.2.7　チャンネル経由でスレッドにリクエストを送信する

さて、thread::spawn に与えられたクロージャが全く何もしないという問題に取り組みましょう。現在、execute メソッドで実行したいクロージャを得ています。ですが、ThreadPool の生成中、Worker それぞれを生成する際に、実行するクロージャを thread::spawn に与える必要があります。

作ったばかりの Worker 構造体に ThreadPool が保持するキューから実行するコードをフェッチして、そのコードをスレッドが実行できるように送信してほしいです。

第 16 章でこのユースケースにぴったりであろう**チャンネル** (2 スレッド間コミュニケーションをとる単純な方法) について学びました。チャンネルをキューの仕事として機能させ、execute は ThreadPool から Worker インスタンスに仕事を送り、これが仕事をスレッドに送信します。こちらが計画です:

1. ThreadPool はチャンネルを生成し、チャンネルの送信側に就く。
2. Worker それぞれは、チャンネルの受信側に就く。
3. チャンネルに送信したいクロージャを保持する新しい Job 構造体を生成する。
4. execute メソッドは、実行したい仕事をチャンネルの送信側に送信する。
5. スレッド内で、Worker はチャンネルの受信側をループし、受け取ったあらゆる仕事のクロージャを実行する。

ThreadPool::new 内でチャンネルを生成し、ThreadPool インスタンスに送信側を保持することから始めましょう。リスト 20-16 のようにですね。今の所、Job 構造体は何も保持しませんが、チャンネルに送信する種類の要素になります。

ファイル名: src/lib.rs

```
# use std::thread;
// --snip--
use std::sync::mpsc;

pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: mpsc::Sender<Job>,
}

struct Job;

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let mut workers = Vec::with_capacity(size);
```

```
        for id in 0..size {
            workers.push(Worker::new(id));
        }

        ThreadPool {
            workers,
            sender,
        }
    }
    // --snip--
}
#
# struct Worker {
#     id: usize,
#     thread: thread::JoinHandle<()>,
# }
#
# impl Worker {
#     fn new(id: usize) -> Worker {
#         let thread = thread::spawn(|| {});
#
#         Worker {
#             id,
#             thread,
#         }
#     }
# }
```

リスト 20-18: `ThreadPool` を変更して `Job` インスタンスを送信するチャンネルの送信側を格納する

`ThreadPool::new` 内で新しいチャンネルを生成し、プールに送信側を保持させています。これはコンパイルに成功しますが、まだ警告があります。

スレッドプールがワーカーを生成する際に各ワーカーにチャンネルの受信側を試しに渡してみましょう。受信側はワーカーが大量生産するスレッド内で使用したいことがわかっているので、クロージャ内で `receiver` 引数を参照します。リスト 20-17 のコードはまだ完璧にはコンパイルできません。

ファイル名: src/lib.rs

```
impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
```

```
            workers.push(Worker::new(id, receiver));
        }

        ThreadPool {
            workers,
            sender,
        }
    }
    // --snip--
}

// --snip--

impl Worker {
    fn new(id: usize, receiver: mpsc::Receiver<Job>) -> Worker {
        let thread = thread::spawn(|| {
            receiver;
        });

        Worker {
            id,
            thread,
        }
    }
}
```

リスト 20-17: チャンネルの受信側をワーカーに渡す

多少些細で単純な変更を行いました: チャンネルの受信側を `Worker::new` に渡し、それからクロージャの内側で使用しています。

このコードのチェックを試みると、このようなエラーが出ます:

```
$ cargo check
   Compiling hello v0.1.0 (file:///projects/hello)
error[E0382]: use of moved value: `receiver`
  --> src/lib.rs:27:42
   |
27 |             workers.push(Worker::new(id, receiver));
   |                                          ^^^^^^^^ value moved here in
   previous iteration of loop
   |
   = note: move occurs because `receiver` has type
   `std::sync::mpsc::Receiver<Job>`, which does not implement the `Copy` trait
```

このコードは、`receiver` を複数の `Worker` インスタンスに渡そうとしています。第 16 章を思い出すように、これは動作しません: Rust が提供するチャンネル実装は、複数の**生成者**、単独の**消費者**です。要するに、チャンネルの消費側をクローンするだけでこのコードを修正することはできません。たとえできたとしても、使用したいテクニックではありません; 代わりに、全ワーカー間で単独の `receiver` を共有することで、スレッド間に仕事を分配したいです。

さらに、チャンネルキューから仕事を取り出すことは、receiver を可変化することに関連するので、スレッドには、receiver を共有して変更する安全な方法が必要です; さもなくば、競合状態に陥る可能性があります (第 16 章で講義しました)。

第 16 章で議論したスレッド安全なスマートポインタを思い出してください: 複数のスレッドで所有権を共有しつつ、スレッドに値を可変化させるためには、Arc<Mutex<T>> を使用する必要があります。Arc 型は、複数のワーカーに受信者を所有させ、Mutex により、1 度に受信者から 1 つの仕事をたった 1 つのワーカーが受け取ることを保証します。リスト 20-18 は、行う必要のある変更を示しています。

ファイル名: src/lib.rs

```
# use std::thread;
# use std::sync::mpsc;
use std::sync::Arc;
use std::sync::Mutex;
// --snip--

# pub struct ThreadPool {
#     workers: Vec<Worker>,
#     sender: mpsc::Sender<Job>,
# }
# struct Job;
#
impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let receiver = Arc::new(Mutex::new(receiver));

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id, Arc::clone(&receiver)));
        }

        ThreadPool {
            workers,
            sender,
        }
    }

    // --snip--
}

# struct Worker {
#     id: usize,
```

```
#     thread: thread::JoinHandle<()>,
# }
#
impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        // --snip--
#         let thread = thread::spawn(|| {
#            receiver;
#         });
#
#         Worker {
#             id,
#             thread,
#         }
    }
}
```

リスト 20-18: `Arc` と `Mutex` を使用してワーカー間でチャンネルの受信側を共有する

`ThreadPool::new` で、チャンネルの受信側を `Arc` と `Mutex` に置いています。新しいワーカーそれぞれに対して、`Arc` をクローンして参照カウントを跳ね上げているので、ワーカーは受信側の所有権を共有することができます。

これらの変更でコードはコンパイルできます！ ゴールはもうすぐそこです！

20.2.2.8　execute メソッドを実装する

最後に `ThreadPool` に `execute` メソッドを実装しましょう。`Job` も構造体から `execute` が受け取るクロージャの型を保持するトレイトオブジェクトの型エイリアスに変更します。第 19 章の「型エイリアスで型同義語を生成する」節で議論したように、型エイリアスにより長い型を短くできます。リスト 20-19 をご覧ください。

ファイル名: src/lib.rs

```
// --snip--
# pub struct ThreadPool {
#     workers: Vec<Worker>,
#     sender: mpsc::Sender<Job>,
# }
# use std::sync::mpsc;
# struct Worker {}

type Job = Box<FnOnce() + Send + 'static>;

impl ThreadPool {
    // --snip--

    pub fn execute<F>(&self, f: F)
        where
```

```
            F: FnOnce() + Send + 'static
    {
        let job = Box::new(f);

        self.sender.send(job).unwrap();
    }
}

// --snip--
```

リスト 20-19: 各クロージャを保持する `Box` に対して `Job` 型エイリアスを生成し、それからチャンネルに仕事を送信する

`execute` で得たクロージャを使用して新しい `Job` インスタンスを生成した後、その仕事をチャンネルの送信側に送信しています。送信が失敗した時のために `send` に対して `unwrap` を呼び出しています。これは例えば、全スレッドの実行を停止させるなど、受信側が新しいメッセージを受け取るのをやめてしまったときなどに起こる可能性があります。現時点では、スレッドの実行を止めることはできません: スレッドは、プールが存在する限り実行し続けます。`unwrap` を使用している理由は、失敗する場合が起こらないとわかっているからですが、コンパイラにはわかりません。

ですが、まだやり終えたわけではありませんよ！ ワーカー内で `thread::spawn` に渡されているクロージャは、それでもチャンネルの受信側を**参照**しているだけです。その代わりに、クロージャには永遠にループし、チャンネルの受信側に仕事を要求し、仕事を得たらその仕事を実行してもらう必要があります。リスト 20-20 に示した変更を `Worker::new` に行いましょう。

ファイル名: src/lib.rs

```
// --snip--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        let thread = thread::spawn(move || {
            loop {
                let job = receiver.lock().unwrap().recv().unwrap();

                // ワーカー{}は仕事を得ました; 実行します
                println!("Worker {} got a job; executing.", id);

                (*job)();
            }
        });

        Worker {
            id,
            thread,
        }
    }
}
```

リスト 20-20: ワーカーのスレッドで仕事を受け取り、実行する

ここで、まず receiver に対して lock を呼び出してミューテックスを獲得し、それから unwrap を呼び出して、エラーの際にはパニックします。ロックの獲得は、ミューテックスが**毒された**状態なら失敗する可能性があり、これは、他のどれかのスレッドがロックを保持している間に、解放するのではなく、パニックした場合に起き得ます。この場面では、unwrap を呼び出してこのスレッドをパニックさせるのは、取るべき正当な行動です。この unwrap をあなたにとって意味のあるエラーメッセージを伴う expect に変更することは、ご自由に行なってください。

ミューテックスのロックを獲得できたら、recv を呼び出してチャンネルから Job を受け取ります。最後の unwrap もここであらゆるエラーを超えていき、これはチャンネルの送信側を保持するスレッドが閉じた場合に発生する可能性があり、受信側が閉じた場合に send メソッドが Err を返すのと似ています。

recv の呼び出しはブロックするので、まだ仕事がなければ、現在のスレッドは、仕事が利用可能になるまで待機します。Mutex<T> により、ただ 1 つの Worker スレッドのみが一度に仕事の要求を試みることを保証します。

理論的には、このコードはコンパイルできるはずです。残念ながら、Rust コンパイラはまだ完全ではなく、このようなエラーが出ます:

```
error[E0161]: cannot move a value of type std::ops::FnOnce() +
std::marker::Send: the size of std::ops::FnOnce() + std::marker::Send cannot be
statically determined
(エラー: std::ops::FnOnce() + std::marker::Sendの値をムーブできません:
std::ops::FnOnce() + std::marker::Sendのサイズを静的に決定できません)
  --> src/lib.rs:63:17
   |
63 |                 (*job)();
   |                 ^^^^^^
```

問題が非常に謎めいているので、エラーも非常に謎めいています。Box<T> に格納された FnOnce クロージャを呼び出すためには (Job 型エイリアスがそう)、呼び出す際にクロージャが self の所有権を奪うので、クロージャは自身を Box<T> **から**ムーブする必要があります。一般的に、Rust は Box<T> から値をムーブすることを許可しません。コンパイラには、Box<T> の内側の値がどれほどの大きさなのか見当がつかないからです: 第 15 章で Box<T> に格納して既知のサイズの値を得たい未知のサイズの何かがあるために Box<T> を正確に使用したことを思い出してください。

リスト 17-15 で見かけたように、記法 self: Box<Self> を使用するメソッドを書くことができ、これにより、メソッドは Box<T> に格納された Self 値の所有権を奪うことができます。それがまさしくここで行いたいことですが、残念ながらコンパイラはさせてくれません: クロージャが呼び出された際に振る舞いを実装する Rust の一部は、self: Box<Self> を使用して実装されていないのです。故に、コンパイラはまだこの場面において self: Box<Self> を使用してクロージャの所有権を奪い、ク

ロージャを Box<T> からムーブできることを理解していないのです。

Rust はまだコンパイラの改善途上にあり、リスト 20-20 のコードは、将来的にうまく動くようになるべきです。まさしくあなたのような方がこれや他の問題を修正しています！ この本を完了したら、是非ともあなたにも参加していただきたいです。

ですがとりあえず、手頃なトリックを使ってこの問題を回避しましょう。この場合、self: Box<Self> で、Box<T> の内部の値の所有権を奪うことができることをコンパイラに明示的に教えてあげます; そして、一旦クロージャの所有権を得たら、呼び出せます。これには、シグニチャに self: Box<Self> を使用する call_box というメソッドのある新しいトレイト FnBox を定義すること、FnOnce() を実装する任意の型に対して FnBox を定義すること、型エイリアスを新しいトレイトを使用するように変更すること、Worker を call_box メソッドを使用するように変更することが関連します。これらの変更は、リスト 20-21 に表示されています。

ファイル名: src/lib.rs

```
trait FnBox {
    fn call_box(self: Box<Self>);
}

impl<F: FnOnce()> FnBox for F {
    fn call_box(self: Box<F>) {
        (*self)()
    }
}

type Job = Box<FnBox + Send + 'static>;

// --snip--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        let thread = thread::spawn(move || {
            loop {
                let job = receiver.lock().unwrap().recv().unwrap();

                println!("Worker {} got a job; executing.", id);

                job.call_box();
            }
        });

        Worker {
            id,
            thread,
        }
    }
}
```

リスト 20-21: 新しいトレイト FnBox を追加して Box<FnOnce()> の現在の制限を回避する

まず、FnBox という新しいトレイトを作成します。このトレイトには call_box という 1 つのメソッドがあり、これは、self: Box<Self> を取って self の所有権を奪い、Box<T> から値をムーブする点を除いて、他の $\text{Fn}_\star$ トレイトの call メソッドと類似しています。

次に、FnOnce() トレイトを実装する任意の型 F に対して FnBox トレイトを実装します。実質的にこれは、あらゆる FnOnce() クロージャが call_box メソッドを使用できることを意味します。call_box の実装は、$(\star\text{self})()$ を使用して Box<T> からクロージャをムーブし、クロージャを呼び出します。

これで Job 型エイリアスには、新しいトレイトの FnBox を実装する何かの Box である必要が出てきました。これにより、クロージャを直接呼び出す代わりに Job 値を得た時に Worker の call_box を使えます。任意の FnOnce() クロージャに対して FnBox トレイトを実装することは、チャンネルに送信する実際の値は何も変えなくてもいいことを意味します。もうコンパイラは、我々が行おうとしていることが平気なことであると認識できます。

このトリックは非常にこそこそしていて複雑です。完璧に筋が通らなくても心配しないでください; いつの日か、完全に不要になるでしょう。

このトリックの実装で、スレッドプールは動く状態になります！ cargo run を実行し、リクエストを行なってください:

```
$ cargo run
   Compiling hello v0.1.0 (file:///projects/hello)
warning: field is never used: `workers`
 --> src/lib.rs:7:5
  |
7 |     workers: Vec<Worker>,
  |     ^^^^^^^^^^^^^^^^^^^^
  |
  = note: #[warn(dead_code)] on by default

warning: field is never used: `id`
  --> src/lib.rs:61:5
   |
61 |     id: usize,
   |     ^^^^^^^^^
   |
   = note: #[warn(dead_code)] on by default

warning: field is never used: `thread`
  --> src/lib.rs:62:5
   |
62 |     thread: thread::JoinHandle<()>,
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: #[warn(dead_code)] on by default

    Finished dev [unoptimized + debuginfo] target(s) in 0.99 secs
     Running `target/debug/hello`
```

```
Worker 0 got a job; executing.
Worker 2 got a job; executing.
Worker 1 got a job; executing.
Worker 3 got a job; executing.
Worker 0 got a job; executing.
Worker 2 got a job; executing.
Worker 1 got a job; executing.
Worker 3 got a job; executing.
Worker 0 got a job; executing.
Worker 2 got a job; executing.
```

成功！ もう非同期に接続を実行するスレッドプールができました。絶対に 4 つ以上のスレッドが生成されないので、サーバが多くのリクエストを受け取っても、システムは過負荷にならないでしょう。**/sleep** にリクエストを行なっても、サーバは他のスレッドに実行させることで他のリクエストを提供できるでしょう。

第 18 章で `while let` ループを学んだ後で、なぜリスト 20-22 に示したようにワーカースレッドのコードを記述しなかったのか、不思議に思っている可能性があります。

ファイル名: src/lib.rs

```
// --snip--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        let thread = thread::spawn(move || {
            while let Ok(job) = receiver.lock().unwrap().recv() {
                println!("Worker {} got a job; executing.", id);

                job.call_box();
            }
        });

        Worker {
            id,
            thread,
        }
    }
}
```

リスト 20-22: `while let` を使用したもう 1 つの `Worker::new` の実装

このコードはコンパイルでき、動きますが、望み通りのスレッドの振る舞いにはなりません: 遅いリクエストがそれでも、他のリクエストが処理されるのを待機させてしまうのです。理由はどこか捉えがたいものです: `Mutex` 構造体には公開の `unlock` メソッドがありません。ロックの所有権が、`lock` メソッドが返す `LockResult<MutexGuard<T>>` 内の `MutexGuard<T>` のライフタイムに基づくからです。コンパイル時には、ロックを保持していない限り、借用チェッカーはそうしたら、`Mutex` に保護されるリソースにはアクセスできないという規則を強制できます。しかし、この実装は、`MutexGuard<T>`

のライフタイムについて熟考しなければ、意図したよりもロックが長い間保持される結果になり得ます。`while` 式の値がブロックの間中スコープに残り続けるので、ロックは `job.call_box` の呼び出し中保持されたままになり、つまり、他のワーカーが仕事を受け取れなくなるのです。

代わりに `loop` を使用し、ロックと仕事をブロックの外ではなく、内側で獲得することで、`lock` メソッドが返す `MutexGuard` は `let job` 文が終わると同時にドロップされます。これにより、複数のリクエストを並行で提供し、ロックは `recv` の呼び出しの間は保持されるけれども、`job.call_box` の呼び出しの前には解放されることを保証します。

20.3　正常なシャットダウンと片付け

リスト 20-21 のコードは、意図した通り、スレッドプールの使用を通してリクエストに非同期に応答できます。直接使用していない `workers`、`id`、`thread` フィールドについて警告が出ます。この警告は、現在のコードは何も片付けていないことを思い出させてくれます。優美さに欠ける ctrl-c を使用してメインスレッドを停止させる方法を使用すると、リクエストの処理中であっても、他のスレッドも停止します。

では、閉じる前に取り掛かっているリクエストを完了できるように、プールの各スレッドに対して `join` を呼び出す `Drop` トレイトを実装します。そして、スレッドに新しいリクエストの受付を停止し、終了するように教える方法を実装します。このコードが動いているのを確かめるために、サーバを変更して正常にスレッドプールを終了する前に 2 つしかリクエストを受け付けないようにします。

20.3.1　`ThreadPool` に `Drop` トレイトを実装する

スレッドプールに `Drop` を実装するところから始めましょう。プールがドロップされると、スレッドは全て join して、作業を完了するのを確かめるべきです。リスト 20-23 は、`Drop` 実装の最初の試みを表示しています; このコードはまだ完全には動きません。

ファイル名: src/lib.rs

```
impl Drop for ThreadPool {
    fn drop(&mut self) {
        for worker in &mut self.workers {
            // ワーカー{}を終了します
            println!("Shutting down worker {}", worker.id);

            worker.thread.join().unwrap();
        }
    }
}
```

リスト 20-23: スレッドプールがスコープを抜けた時にスレッドを join させる

まず、スレッドプール `workers` それぞれを走査します。`self` は可変参照であり、`worker` を可変化

できる必要もあるので、これには &mut を使用しています。ワーカーそれぞれに対して、特定のワーカーを終了する旨のメッセージを出力し、それから join をワーカースレッドに対して呼び出しています。join の呼び出しが失敗したら、unwrap を使用して Rust をパニックさせ、正常でないシャットダウンに移行します。

こちらが、このコードをコンパイルする際に出るエラーです:

```
error[E0507]: cannot move out of borrowed content
  --> src/lib.rs:65:13
   |
65 |             worker.thread.join().unwrap();
   |             ^^^^^^ cannot move out of borrowed content
```

各 worker の可変参照しかなく、join は引数の所有権を奪うためにこのエラーは join を呼び出せないと教えてくれています。この問題を解決するには、join がスレッドを消費できるように、thread を所有する Worker インスタンスからスレッドをムーブする必要があります。これをリスト 17-15 では行いました: Worker が代わりに Option<thread::JoinHandle<()>> を保持していれば、Option に対して take メソッドを呼び出し、Some 列挙子から値をムーブし、その場所に None 列挙子を残すことができます。言い換えれば、実行中の Worker には thread に Some 列挙子があり、Worker を片付けたい時には、ワーカーが実行するスレッドがないように Some を None で置き換えるのです。

従って、Worker の定義を以下のように更新したいことがわかります:

ファイル名: src/lib.rs

```
# use std::thread;
struct Worker {
    id: usize,
    thread: Option<thread::JoinHandle<()>>,
}
```

さて、コンパイラを頼りにして他に変更する必要がある箇所を探しましょう。このコードをチェックすると、2 つのエラーが出ます:

```
error[E0599]: no method named `join` found for type
`std::option::Option<std::thread::JoinHandle<()>>` in the current scope
  --> src/lib.rs:65:27
   |
65 |             worker.thread.join().unwrap();
   |                           ^^^^

error[E0308]: mismatched types
  --> src/lib.rs:89:13
   |
89 |             thread,
   |             ^^^^^^
   |             |
   |             expected enum `std::option::Option`, found struct
```

```
    `std::thread::JoinHandle`
    |             help: try using a variant of the expected type: `Some(thread)`
    |
    = note: expected type `std::option::Option<std::thread::JoinHandle<()>>`
               found type `std::thread::JoinHandle<_>`
```

2 番目のエラーを扱いましょう。これは、Worker::new の最後のコードを指しています; 新しい Worker を作成する際に、Some に thread の値を包む必要があります。このエラーを修正するために以下の変更を行なってください:

ファイル名: src/lib.rs

```
impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        // --snip--

        Worker {
            id,
            thread: Some(thread),
        }
    }
}
```

最初のエラーは Drop 実装内にあります。先ほど、Option 値に対して take を呼び出し、thread を worker からムーブする意図があることに触れました。以下の変更がそれを行います:

ファイル名: src/lib.rs

```
impl Drop for ThreadPool {
    fn drop(&mut self) {
        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            if let Some(thread) = worker.thread.take() {
                thread.join().unwrap();
            }
        }
    }
}
```

第 17 章で議論したように、Option の take メソッドは、Some 列挙子を取り出し、その箇所に None を残します。if let を使用して Some を分配し、スレッドを得ています; そして、スレッドに対して join を呼び出します。ワーカーのスレッドが既に None なら、ワーカーはスレッドを既に片付け済みであることがわかるので、その場合には何も起きません。

20.3.2　スレッドに仕事をリッスンするのを止めるよう通知する

これらの変更によって、コードは警告なしでコンパイルできます。ですが悪い知らせは、このコードが期待したようにはまだ機能しないことです。鍵は、`Worker` インスタンスのスレッドで実行されるクロージャのロジックです: 現時点で `join` を呼び出していますが、仕事を求めて永遠に `loop` するので、スレッドを終了しません。現在の `drop` の実装で `ThreadPool` をドロップしようとしたら、最初のスレッドが完了するのを待機してメインスレッドは永遠にブロックされるでしょう。

この問題を修正するには、スレッドが、実行すべき `Job` か、リッスンをやめて無限ループを抜ける通知をリッスンするように、変更します。`Job` インスタンスの代わりに、チャンネルはこれら 2 つの enum 列挙子の一方を送信します。

ファイル名: src/lib.rs

```
# struct Job;
enum Message {
    NewJob(Job),
    Terminate,
}
```

この `Message` enum はスレッドが実行すべき `Job` を保持する `NewJob` 列挙子か、スレッドをループから抜けさせ、停止させる `Terminate` 列挙子のどちらかになります。

チャンネルを調整し、型 `Job` ではなく、型 `Message` を使用するようにする必要があります。リスト 20-24 のようにですね。

ファイル名: src/lib.rs

```
pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: mpsc::Sender<Message>,
}

// --snip--

impl ThreadPool {
    // --snip--

    pub fn execute<F>(&self, f: F)
        where
            F: FnOnce() + Send + 'static
    {
        let job = Box::new(f);

        self.sender.send(Message::NewJob(job)).unwrap();
    }
}
```

```
// --snip--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Message>>>) ->
        Worker {

        let thread = thread::spawn(move ||{
            loop {
                let message = receiver.lock().unwrap().recv().unwrap();

                match message {
                    Message::NewJob(job) => {
                        println!("Worker {} got a job; executing.", id);

                        job.call_box();
                    },
                    Message::Terminate => {
                        // ワーカー{}は停止するよう指示された
                        println!("Worker {} was told to terminate.", id);

                        break;
                    },
                }
            }
        });

        Worker {
            id,
            thread: Some(thread),
        }
    }
}
```

リスト 20-24: `Message` 値を送受信し、`Worker` が `Message::Terminate` を受け取ったら、ループを抜ける

`Message` enum を具体化するために、2 箇所で `Job` を `Message` に変更する必要があります: `ThreadPool` の定義と `Worker::new` のシグニチャです。`ThreadPool` の `execute` メソッドは、仕事を `Message::NewJob` 列挙子に包んで送信する必要があります。それから、`Message` がチャンネルから受け取られる `Worker::new` で、`NewJob` 列挙子が受け取られたら、仕事が処理され、`Terminate` 列挙子が受け取られたら、スレッドはループを抜けます。

これらの変更と共に、コードはコンパイルでき、リスト 20-21 の後と同じように機能し続けます。ですが、`Terminate` のメッセージを何も生成していないので、警告が出るでしょう。`Drop` 実装をリスト 20-25 のような見た目に変更してこの警告を修正しましょう。

ファイル名: src/lib.rs

```
impl Drop for ThreadPool {
    fn drop(&mut self) {
        println!("Sending terminate message to all workers.");

        for _ in &mut self.workers {
            self.sender.send(Message::Terminate).unwrap();
        }

        // 全ワーカーを閉じます
        println!("Shutting down all workers.");

        for worker in &mut self.workers {
            // ワーカー{}を閉じます
            println!("Shutting down worker {}", worker.id);

            if let Some(thread) = worker.thread.take() {
                thread.join().unwrap();
            }
        }
    }
}
```

リスト 20-25: 各ワーカースレッドに対して `join` を呼び出す前にワーカーに `Message::Terminate` を送信する

今では、ワーカーを 2 回走査しています: 各ワーカーに `Terminate` メッセージを送信するために 1 回と、各ワーカースレッドに `join` を呼び出すために 1 回です。メッセージ送信と `join` を同じループで即座に行おうとすると、現在の繰り返しのワーカーがチャンネルからメッセージを受け取っているものであるか保証できなくなってしまいます。

2 つの個別のループが必要な理由をよりよく理解するために、2 つのワーカーがある筋書きを想像してください。単独のループで各ワーカーを走査すると、最初の繰り返しでチャンネルに停止メッセージが送信され、`join` が最初のワーカースレッドで呼び出されます。その最初のワーカーが現在、リクエストの処理で忙しければ、2 番目のワーカーがチャンネルから停止メッセージを受け取り、閉じます。最初のワーカーの終了待ちをしたままですが、2 番目のスレッドが停止メッセージを拾ってしまったので、終了することは絶対にありません。デッドロックです！

この筋書きを回避するために、1 つのループでまず、チャンネルに対して全ての `Terminate` メッセージを送信します; そして、別のループで全スレッドの join を待ちます。一旦停止メッセージを受け取ったら、各ワーカーはチャンネルからのリクエストの受付をやめます。故に、存在するワーカーと同じ数だけ停止メッセージを送れば、`join` がスレッドに対して呼び出される前に、停止メッセージを各ワーカーが受け取ると確信できるわけです。

このコードが動いているところを確認するために、`main` を変更してサーバを正常に閉じる前に 2 つしかリクエストを受け付けないようにしましょう。リスト 20-26 のようにですね。

ファイル名: src/bin/main.rs

```
fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
    let pool = ThreadPool::new(4);

    for stream in listener.incoming().take(2) {
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
    }

    println!("Shutting down.");
}
```

リスト 20-26: ループを抜けることで、2 つのリクエストを処理した後にサーバを閉じる

現実世界の Web サーバには、たった 2 つリクエストを受け付けた後にシャットダウンしてほしくはないでしょう。このコードは、単に正常なシャットダウンとクリーンアップが正しく機能することを示すだけです。

take メソッドは、Iterator トレイトで定義されていて、最大でも繰り返しを最初の 2 つの要素だけに制限します。ThreadPool は main の末端でスコープを抜け、drop 実装が実行されます。

cargo run でサーバを開始し、3 つリクエストを行なってください。3 番目のリクエストはエラーになるはずで、端末にはこのような出力が目撃できるはずです:

```
$ cargo run
   Compiling hello v0.1.0 (file:///projects/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 1.0 secs
     Running `target/debug/hello`
Worker 0 got a job; executing.
Worker 3 got a job; executing.
Shutting down.
Sending terminate message to all workers.
Shutting down all workers.
Shutting down worker 0
Worker 1 was told to terminate.
Worker 2 was told to terminate.
Worker 0 was told to terminate.
Worker 3 was told to terminate.
Shutting down worker 1
Shutting down worker 2
Shutting down worker 3
```

ワーカーとメッセージの順番は異なる可能性があります。どうやってこのコードが動くのかメッセージからわかります: ワーカー 0 と 3 が最初の 2 つのリクエストを受け付け、そして 3 番目のリクエストではサーバは接続の受け入れをやめます。main の最後で ThreadPool がスコープを抜ける際、Drop 実装が割り込み、プールが全ワーカーに停止するよう指示します。ワーカーはそれぞれ、停止

メッセージを確認した時にメッセージを出力し、それからスレッドプールは各ワーカースレッドを閉じる join を呼び出します。

この特定の実行のある面白い側面に注目してください: ThreadPool はチャンネルに停止メッセージを送信しますが、どのワーカーがそのメッセージを受け取るよりも前に、ワーカー 0 の join を試みています。ワーカー 0 はまだ停止メッセージを受け取っていなかったので、メインスレッドはワーカー 0 が完了するまで待機してブロックされます。その間に、各ワーカーは停止メッセージを受け取ります。ワーカー 0 が完了したら、メインスレッドは残りのワーカーが完了するのを待機します。その時点で全ワーカーは停止メッセージを受け取った後で、閉じることができたのです。

おめでとうございます！ プロジェクトを完成させました; スレッドプールを使用して非同期に応答する基本的な Web サーバができました。サーバの正常なシャットダウンを行うことができ、プールの全スレッドを片付けます。

参考までに、こちらが全コードです:

ファイル名: src/bin/main.rs

```
extern crate hello;
use hello::ThreadPool;

use std::io::prelude::*;
use std::net::TcpListener;
use std::net::TcpStream;
use std::fs::File;
use std::thread;
use std::time::Duration;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
    let pool = ThreadPool::new(4);

    for stream in listener.incoming().take(2) {
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
    }

    // 閉じます
    println!("Shutting down.");
}

fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 1024];
    stream.read(&mut buffer).unwrap();

    let get = b"GET / HTTP/1.1\r\n";
    let sleep = b"GET /sleep HTTP/1.1\r\n";
```

```
    let (status_line, filename) = if buffer.starts_with(get) {
        ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
    } else if buffer.starts_with(sleep) {
        thread::sleep(Duration::from_secs(5));
        ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
    } else {
        ("HTTP/1.1 404 NOT FOUND\r\n\r\n", "404.html")
    };

     let mut file = File::open(filename).unwrap();
     let mut contents = String::new();

     file.read_to_string(&mut contents).unwrap();

     let response = format!("{}{}", status_line, contents);

     stream.write(response.as_bytes()).unwrap();
     stream.flush().unwrap();
}
```

ファイル名: src/lib.rs

```
use std::thread;
use std::sync::mpsc;
use std::sync::Arc;
use std::sync::Mutex;

enum Message {
    NewJob(Job),
    Terminate,
}

pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: mpsc::Sender<Message>,
}

trait FnBox {
    fn call_box(self: Box<Self>);
}

impl<F: FnOnce()> FnBox for F {
    fn call_box(self: Box<F>) {
        (*self)()
    }
}

type Job = Box<FnBox + Send + 'static>;

impl ThreadPool {
    /// Create a new ThreadPool.
```

```
    ///
    /// The size is the number of threads in the pool.
    ///
    /// # Panics
    ///
    /// The `new` function will panic if the size is zero.
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let receiver = Arc::new(Mutex::new(receiver));

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id, Arc::clone(&receiver)));
        }

        ThreadPool {
            workers,
            sender,
        }
    }

    pub fn execute<F>(&self, f: F)
        where
            F: FnOnce() + Send + 'static
    {
        let job = Box::new(f);

        self.sender.send(Message::NewJob(job)).unwrap();
    }
}

impl Drop for ThreadPool {
    fn drop(&mut self) {
        println!("Sending terminate message to all workers.");

        for _ in &mut self.workers {
            self.sender.send(Message::Terminate).unwrap();
        }

        println!("Shutting down all workers.");

        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            if let Some(thread) = worker.thread.take() {
                thread.join().unwrap();
            }
        }
    }
```

```
}

struct Worker {
    id: usize,
    thread: Option<thread::JoinHandle<()>>,
}

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Message>>>) ->
        Worker {

        let thread = thread::spawn(move ||{
            loop {
                let message = receiver.lock().unwrap().recv().unwrap();

                match message {
                    Message::NewJob(job) => {
                        println!("Worker {} got a job; executing.", id);

                        job.call_box();
                    },
                    Message::Terminate => {
                        println!("Worker {} was told to terminate.", id);

                        break;
                    },
                }
            }
        });

        Worker {
            id,
            thread: Some(thread),
        }
    }
}
```

ここでできることはまだあるでしょう！ よりこのプロジェクトを改善したいのなら、こちらがアイディアの一部です:

- `ThreadPool` とその公開メソッドにもっとドキュメンテーションを追加する。
- ライブラリの機能のテストを追加する。
- `unwrap` の呼び出しをもっと頑健なエラー処理に変更する。
- `ThreadPool` を使用して Web リクエスト以外のなんらかの作業を行う。
- **https://crates.io** でスレッドプールのクレートを探して、そのクレートを代わりに使用して似た Web サーバを実装する。そして、API と頑健性を我々が実装したものと比較する。

20.4　まとめ

よくやりました！ 本の最後に到達しました！ Rust のツアーに参加していただき、感謝の辞を述べたいです。もう、ご自身の Rust プロジェクトや他の方のプロジェクトのお手伝いをする準備ができています。あなたがこれからの Rust の旅で遭遇する、あらゆる困難の手助けを是非とも行いたい Rustacean たちの温かいコミュニティがあることを心に留めておいてくださいね。

付録

以下の節は、Rust の旅で役に立つと思えるかもしれない参考資料を含んでいます。

付録 A: キーワード

以下のリストは、現在、あるいは将来 Rust 言語により使用されるために予約されているキーワードです。そのため、識別子として使用することはできません。識別子には、関数名、変数名、引数名、構造体のフィールド名、モジュール名、クレート名、定数名、マクロ名、静的な値の名前、属性名、型名、トレイト名、ライフタイム名などがあります。ただし、生識別子のところで議論する生識別子は例外です。

現在使用されているキーワード

以下のキーワードは、解説された通りの機能が現状あります。

- `as` - 基礎的なキャストの実行、要素を含む特定のトレイトの明確化、`use` や `extern crate` 文の要素名を変更する
- `async` - 現在のスレッドをブロックする代わりに `Future` を返す
- `await` - `Future` の結果が準備できるまで実行を停止する
- `break` - 即座にループを抜ける
- `const` - 定数要素か定数の生ポインタを定義する
- `continue` - 次のループの繰り返しに継続する
- `crate` - 外部のクレートかマクロが定義されているクレートを表すマクロ変数をリンクする
- `else` - `if` と `if let` 制御フロー構文の規定
- `enum` - 列挙型を定義する
- `extern` - 外部のクレート、関数、変数をリンクする
- `false` - bool 型の false リテラル
- `fn` - 関数か関数ポインタ型を定義する
- `for` - イテレータの要素を繰り返す、トレイトの実装、高階ライフタイムの指定
- `if` - 条件式の結果によって条件分岐

- `impl` - 固有の機能やトレイトの機能を実装する
- `in` - `for` ループ記法の一部
- `let` - 変数を束縛する
- `loop` - 無条件にループする
- `match` - 値をパターンとマッチさせる
- `mod` - モジュールを定義する
- `move` - クロージャにキャプチャした変数全ての所有権を奪わせる
- `mut` - 参照、生ポインタ、パターン束縛で可変性に言及する
- `pub` - 構造体フィールド、`impl` ブロック、モジュールで公開性について言及する
- `ref` - 参照で束縛する
- `return` - 関数から帰る
- `Self` - 定義しようとしている・実装 (implement) しようとしている型の型エイリアス
- `self` - メソッドの主題、または現在のモジュール
- `static` - グローバル変数、またはプログラム全体に渡るライフタイム
- `struct` - 構造体を定義する
- `super` - 現在のモジュールの親モジュール
- `trait` - トレイトを定義する
- `true` - bool 型の true リテラル
- `type` - 型エイリアスか関連型を定義する
- `unsafe` - unsafe なコード、関数、トレイト、実装に言及する
- `use` - スコープにシンボルを持ち込む
- `where` - 型を制限する節に言及する
- `while` - 式の結果に基づいて条件的にループする

将来的な使用のために予約されているキーワード

以下のキーワードには機能が何もないものの、将来的に使用される可能性があるので、Rust により予約されています。

- `abstract`
- `become`
- `box`
- `do`
- `final`
- `macro`
- `override`
- `priv`

- try
- typeof
- unsized
- virtual
- yield

生識別子

生識別子とは、普段は使うことが許されないキーワードを使わせてくれる構文です。生識別子はキーワードの前に r# を置いて使うことができます。

たとえば、match はキーワードです。次の、名前が match である関数をコンパイルしようとすると：

ファイル名: src/main.rs

```
fn match(needle: &str, haystack: &str) -> bool {
    haystack.contains(needle)
}
// 訳注: 引数名は、"a needle in a haystack" すなわち「干し草の中の針」という、
// 「見つかりそうにない捜し物」を意味する成句からもじった命名。
// 検索をする関数でよく使われる。
```

次のエラーを得ます：

```
error: expected identifier, found keyword `match`
 --> src/main.rs:4:4
  |
4 | fn match(needle: &str, haystack: &str) -> bool {
  |    ^^^^^ expected identifier, found keyword
```

このエラーは match というキーワードを関数の識別子としては使えないと示しています。match を関数名として使うには、次のように、生識別子構文を使う必要があります。

ファイル名: src/main.rs

```
fn r#match(needle: &str, haystack: &str) -> bool {
    haystack.contains(needle)
}

fn main() {
    assert!(r#match("foo", "foobar"));
}
```

このコードはなんのエラーもなくコンパイルできます。r# は、定義のときも、main 内で呼ばれたときにも、関数名の前につけられていることに注意してください。

生識別子を使えば、仮にそれが予約されたキーワードであろうとも、任意の単語を識別子として使えるようになります。更に、あなたのクレートが使っている Rust の edition とは異なる edition で書かれたライブラリを呼び出すこともできるようになります。たとえば、`try` は 2015 edition ではキーワードではありませんでしたが、2018 edition ではキーワードです。もし、2015 edition で書かれており、`try` 関数を持っているライブラリに依存している場合、あなたの 2018 edition のコードからその関数を呼び出すためには、生識別子構文を使う必要がでてくるでしょう。今回なら `r#try` ですね。edition に関して、より詳しくは付録 E を見てください。

付録 B: 演算子と記号

この付録は、演算子や、単独で現れたり、パス、ジェネリクス、トレイト境界、マクロ、属性、コメント、タプル、かっこの文脈で現れる他の記号を含む Rust の記法の用語集を含んでいます。

演算子

表 B-1 は、Rust の演算子、演算子が文脈で現れる例、短い説明、その演算子がオーバーロード可能かどうかを含んでいます。演算子がオーバーロード可能ならば、オーバーロードするのに使用する関係のあるトレイトも列挙されています。

表 B-1: 演算子

演算子	例	説明	オーバーロードできる？
`!`	`ident!(...)`, `ident!{...}`, `ident![...]`	マクロ展開	
`!`	`!expr`	ビット反転、または論理反転	`Not`
`!=`	`var != expr`	非等価比較	`PartialEq`
`%`	`expr % expr`	余り演算	`Rem`
`%=`	`var %= expr`	余り演算後に代入	`RemAssign`
`&`	`&expr`, `&mut expr`	借用	
`&`	`&type`, `&mut type`, `&'a type`, `&'a mut type`	借用されたポインタ型	
`&`	`expr & expr`	ビット AND	`BitAnd`
`&=`	`var &= expr`	ビット AND 後に代入	`BitAndAssign`
`&&`	`expr && expr`	論理 AND	
`*`	`expr * expr`	掛け算	`Mul`
`*`	`*expr`	参照外し	
`*`	`*const type`, `*mut type`	生ポインタ	

演算子	例	説明	オーバーロードできる？
*=	`var *= expr`	掛け算後に代入	`MulAssign`
+	`trait + trait, 'a + trait`	型制限の複合化	
+	`expr + expr`	足し算	`Add`
+=	`var += expr`	足し算後に代入	`AddAssign`
,	`expr, expr`	引数と要素の区別	
-	`- expr`	算術否定	`Neg`
-	`expr - expr`	引き算	`Sub`
-=	`var -= expr`	引き算後に代入	`SubAssign`
->	`fn(...) -> type`, `|...| -> type`	関数とクロージャの戻り値型	
.	`expr.ident`	メンバーアクセス	
..	`..`, `expr..`, `..expr`, `expr..expr`	未満範囲リテラル	
..	`..expr`	構造体リテラル更新記法	
..	`variant(x, ..)`, `struct_type { x, .. }`	「残り全部」パターン束縛	
...	`expr...expr`	パターンで: 以下範囲パターン	
/	`expr / expr`	割り算	`Div`
/=	`var /= expr`	割り算後に代入	`DivAssign`
:	`pat: type`, `ident: type`	型制約	
:	`ident: expr`	構造体フィールド初期化子	
:	`'a: loop {...}`	ループラベル	
;	`expr;`	文、要素終端子	
;	`[...; len]`	固定長配列記法の一部	
<<	`expr << expr`	左シフト	`Shl`
<<=	`var <<= expr`	左シフト後に代入	`ShlAssign`
<	`expr < expr`	未満比較	`PartialOrd`
<=	`expr <= expr`	以下比較	`PartialOrd`
=	`var = expr`, `ident = type`	代入/等価	
==	`expr == expr`	等価比較	`PartialEq`
=>	`pat => expr`	match アーム記法の一部	
>	`expr > expr`	より大きい比較	`PartialOrd`
>=	`expr >= expr`	以上比較	`PartialOrd`
>>	`expr >> expr`	右シフト	`Shr`

演算子	例	説明	オーバーロードできる？
`>>=`	`var >>= expr`	右シフト後に代入	`ShrAssign`
`@`	`ident @ pat`	パターン束縛	
`^`	`expr ^ expr`	ビット XOR	`BitXor`
`^=`	`var ^= expr`	ビット XOR 後に代入	`BitXorAssign`
`|`	`pat | pat`	パターン OR	
`|`	`|…| expr`	クロージャ	
`|`	`expr | expr`	ビット OR	`BitOr`
`|=`	`var |= expr`	ビット OR 後に代入	`BitOrAssign`
`||`	`expr || expr`	論理 OR	
`?`	`expr?`	エラー委譲	

演算子以外のシンボル

以下のリストは、演算子として機能しない記号全部を含んでいます; つまり、関数やメソッド呼び出しのようには、振る舞わないということです。

表 B-2 は、単独で出現し、いろんな箇所で合法になる記号を示しています。

表 B-2: スタンドアローン記法

シンボル	説明
`'ident`	名前付きのライフタイム、あるいはループラベル
`...u8`, `...i32`, `...f64`, `...usize` など	特定の型の数値リテラル
`"..."`	文字列リテラル
`r"..."`, `r#"..."#`, `r##"..."##` など	生文字列リテラル、エスケープ文字は処理されません
`b"..."`	バイト文字列リテラル、文字列の代わりに `[u8]` を構築します
`br"..."`, `br#"..."#`, `br##"..."##` など	生バイト文字列リテラル、生文字列とバイト文字列の組み合わせ
`'...'`	文字リテラル
`b'...'`	ASCII バイトリテラル
`|...| expr`	クロージャ
`!`	常に発散関数の空のボトム型

シンボル	説明
`_`	「無視」パターン束縛: 整数リテラルを見やすくするのにも使われる

表 B-3 は、要素へのモジュール階層を通したパスの文脈で出現する記号を示しています。

表 B-3: パス関連記法

シンボル	説明
`ident::ident`	名前空間パス
`::path`	クレートルートに相対的なパス (すなわち、明示的な絶対パス)
`self::path`	現在のモジュールに相対的なパス (すなわち、明示的な相対パス)
`super::path`	現在のモジュールの親モジュールに相対的なパス
`type::ident`, `<type as trait>::ident`	関連定数、関数、型
`<type>::...`	直接名前付けできない型の関連要素 (例, `<&T>::...`, `<[T]>::...` など)
`trait::method(...)`	定義したトレイトを名指ししてメソッド呼び出しを明確化する
`type::method(...)`	定義されている型を名指ししてメソッド呼び出しを明確化する
`<type as trait>::method(...)`	トレイト**と**型を名指ししてメソッド呼び出しを明確化する

表 B-4 は、ジェネリックな型引数の文脈で出現する記号を示しています。

表 B-4: ジェネリクス

シンボル	説明
`path<...>`	型の内部のジェネリック型への引数を指定する (例、`Vec<u8>`)
`path::<...>`, `method::<...>`	式中のジェネリックな型、関数、メソッドへの引数を指定する。しばしばターボ・フィッシュ (turbofish) と称される。(例、`"42".parse::<i32>()`)
`fn ident<...> ...`	ジェネリックな関数を定義する
`struct ident<...> ...`	ジェネリックな構造体を定義する
`enum ident<...> ...`	ジェネリックな列挙型を定義する
`impl<...> ...`	ジェネリックな実装を定義する
`for<...> type`	高階ライフタイム境界

シンボル	説明
`type<ident=type>`	1 つ以上の関連型に代入されたジェネリックな型 (例、`Iterator<Item=T>`)

表 B-5 は、ジェネリック型引数をトレイト境界で制約する文脈で出現する記号を示しています。

表 B-5: トレイト境界制約

シンボル	説明
`T: U`	`U` を実装する型に制約されるジェネリック引数 `T`
`T: 'a`	ライフタイム `'a` よりも長生きしなければならないジェネリック型 `T` (型がライフタイムより長生きするとは、`'a` よりも短いライフタイムの参照を何も遷移的に含められないことを意味する)
`T : 'static`	ジェネリック型 `T` が `'static` なもの以外の借用された参照を何も含まない
`'b: 'a`	ジェネリックなライフタイム `'b` がライフタイム `'a` より長生きしなければならない
`T: ?Sized`	ジェネリック型引数が動的サイズ決定型であることを許容する
`'a + trait, trait + trait`	複合型制約

表 B-6 は、マクロの呼び出しや定義、要素に属性を指定する文脈で出現する記号を示しています。

表 B-6: マクロと属性

シンボル	説明
`#[meta]`	外部属性
`#![meta]`	内部属性
`$ident`	マクロ代用
`$ident:kind`	マクロキャプチャ
`$(…)…`	マクロの繰り返し

表 B-7 は、コメントを生成する記号を示しています。

表 B-7: コメント

シンボル	説明
`//`	行コメント
`//!`	内部行 doc コメント
`///`	外部行 doc コメント
`/*...*/`	ブロックコメント
`/*!...*/`	内部ブロック doc コメント
`/**...*/`	外部ブロック doc コメント

タプル

表 B-8 は、タプルの文脈で出現する記号を示しています。

表 B-8: タプル

シンボル	説明
`()`	空のタプル (ユニットとしても知られる)、リテラル、型両方
`(expr)`	括弧付きの式
`(expr,)`	1 要素タプル式
`(type,)`	1 要素タプル型
`(expr, ...)`	タプル式
`(type, ...)`	タプル型
`expr(expr, ...)`	関数呼び出し式; タプル `struct` やタプル `enum` 列挙子を初期化するのにも使用される
`ident!(...)`, `ident!{...}`, `ident![...]`	マクロ呼び出し
`expr.0`, `expr.1`, など	タプル添え字アクセス

表 B-9 は、波括弧が使用される文脈を表示しています。

表 B-9: 波括弧

文脈	説明
`{...}`	ブロック式
`Type {...}`	`struct` リテラル

表 B-10 は、角括弧が使用される文脈を表示しています。

表 B-10: 角括弧

文脈	説明
`[...]`	配列リテラル
`[expr; len]`	`len` 個 `expr` を含む配列リテラル
`[type; len]`	`len` 個の `type` のインスタンスを含む配列型
`expr[expr]`	コレクション添え字アクセス。オーバーロード可能 (`Index` , `IndexMut`)
`expr[..]` , `expr[a..]` , `expr[..b]` , `expr[a..b]`	`Range` 、`RangeFrom` 、`RangeTo` 、`RangeFull` を「添え字」として使用してコレクション・スライシングの振りをするコレクション添え字アクセス

付録 C: 導出可能なトレイト

本のいろんな箇所で `derive` 属性について議論しました。これは構造体や、enum 定義に適用できます。`derive` 属性は、`derive` 記法で注釈した型に対して独自の既定の実装でトレイトを実装するコードを生成します。

この付録では、標準ライブラリの `derive` と共に使用できる全トレイトの参照を提供します。各節は以下を講義します:

- このトレイトを導出する演算子やメソッドで可能になること
- `derive` が提供するトレイトの実装がすること
- トレイトを実装することが型についてどれほど重要か
- そのトレイトを実装できたりできなかったりする条件
- そのトレイトが必要になる処理の例

`derive` 属性が提供する以外の異なる振る舞いが欲しいなら、それらを手動で実装する方法の詳細について、各トレイトの標準ライブラリのドキュメンテーションを調べてください。

標準ライブラリで定義されている残りのトレイトは、`derive` で自分の型に実装することはできません。これらのトレイトには知覚できるほどの既定の振る舞いはないので、自分が達成しようしていることに対して、道理が通る方法でそれらを実装するのはあなた次第です。

導出できないトレイトの例は `Display` で、これはエンドユーザ向けのフォーマットを扱います。常に、エンドユーザ向けに型を表示する適切な方法について、考慮すべきです。型のどの部分をエンドユーザは見ることができるべきでしょうか？ どの部分を関係があると考えるでしょうか？ どんな形式のデータがエンドユーザにとって最も関係があるでしょうか？ Rust コンパイラには、この見識がないため、適切な既定動作を提供してくれないのです。

この付録で提供される導出可能なトレイトのリストは、包括的ではありません: ライブラリは、自身のトレイトに `derive` を実装でき、`derive` と共に使用できるトレイトのリストが実に限りのないも

のになってしまうのです。derive の実装には、プロシージャルなマクロが関連します。マクロについては、付録 D で講義します。

プログラマ用の出力の Debug

Debug トレイトにより、フォーマット文字列でのデバッグ整形が可能になり、{} プレースホルダー内に :? を追記することで表します。

Debug トレイトにより、デバッグ目的で型のインスタンスを出力できるようになるので、あなたや型を使用する他のプログラマが、プログラムの実行の特定の箇所でインスタンスを調べられます。

Debug トレイトは、例えば、assert_eq! マクロを使用する際などに必要になります。このマクロは、プログラマがどうして 2 つのインスタンスが等価でなかったのか確認できるように、等価アサートが失敗したら、引数として与えられたインスタンスの値を出力します。

等価比較のための PartialEq と Eq

PartialEq トレイトにより、型のインスタンスを比較して、等価性をチェックでき、== と != 演算子の使用を可能にします。

PartialEq を導出すると、eq メソッドを実装します。構造体に PartialEq を導出すると、**全**フィールドが等しい時のみ 2 つのインスタンスは等価になり、いずれかのフィールドが等価でなければ、インスタンスは等価ではなくなります。enum に導出すると、各列挙子は、自身には等価ですが、他の列挙子には等価ではありません。

PartialEq トレイトは例えば、assert_eq! マクロを使用する際に必要になります。これは、等価性のためにとある型の 2 つのインスタンスを比較できる必要があります。

Eq トレイトにはメソッドはありません。その目的は、注釈された型の全値に対して、値が自身と等しいことを通知することです。Eq トレイトは、PartialEq を実装する全ての型が Eq を実装できるわけではないものの、PartialEq も実装する型に対してのみ適用できます。これの一例は、浮動小数点数型です: 浮動小数点数の実装により、非数字 (NaN) 値の 2 つのインスタンスはお互いに等価ではないことが宣言されます。

Eq が必要になる一例が、HashMap<K, V> のキーで、HashMap<K, V> が、2 つのキーが同じであると判定できます。

順序付き比較のための PartialOrd と Ord

PartialOrd トレイトにより、ソートする目的で型のインスタンスを比較できます。PartialOrd を実装する型は、< 、> 、<= 、>= 演算子を使用することができます。PartialEq も実装する型に対してのみ、PartialOrd トレイトを適用できます。

PartialOrd を導出すると、partial_cmp メソッドを実装し、これは、与えられた値が順序付けられない時に None になる Option<Ordering> を返します。その型のほとんどの値は比較できるものの、順

序付けできない値の例として、非数字 (NaN) 浮動小数点値が挙げられます。partial_cmp をあらゆる浮動小数点数と NaN 浮動小数点数で呼び出すと、None が返るでしょう。

構造体に導出すると、フィールドが構造体定義で現れる順番で各フィールドの値を比較することで 2 つのインスタンスを比較します。enum に導出すると、enum 定義で先に定義された列挙子が、後に列挙された列挙子よりも小さいと考えられます。

PartialOrd トレイトが必要になる例には、低い値と高い値で指定される範囲の乱数を生成する rand クレートの gen_range メソッドが挙げられます。

Ord トレイトにより、注釈した型のあらゆる 2 つの値に対して、合法な順序付けが行えることがわかります。Ord トレイトは cmp メソッドを実装し、これは、常に合法な順序付けが可能なので、Option<Ordering> ではなく、Ordering を返します。PartialOrd と Eq (Eq は PartialEq も必要とします) も実装している型にしか、Ord トレイトを適用することはできません。構造体と enum で導出したら、PartialOrd で、partial_cmp の導出した実装と同じように cmp は振る舞います。

Ord が必要になる例は、BTreeSet<T> に値を格納する時です。これは、値のソート順に基づいてデータを格納するデータ構造です。

値を複製する Clone と Copy

Clone トレイトにより値のディープコピーを明示的に行うことができ、複製のプロセスは、任意のコードを実行し、ヒープデータをコピーすることに関係がある可能性があります。Clone について詳しくは、第 4 章の「変数とデータの相互作用法: Clone」節を参照されたし。

Clone を導出すると、clone メソッドを実装し、これは型全体に対して実装されると、型の各部品に対して clone を呼び出します。要するに、Clone を導出するには、型のフィールドと値全部も Clone を実装していなければならないということです。

Clone が必要になる例は、スライスに対して to_vec メソッドを呼び出すことです。スライスは、含んでいる型のインスタンスの所有権を持たないが、to_vec で返されるベクタはそのインスタンスを所有する必要があるので、to_vec は各要素に対して clone を呼び出します。故に、スライスに格納される型は、Clone を実装しなければならないのです。

Copy トレイトにより、スタックに格納されたビットをコピーするだけで値を複製できます; 任意のコードは必要ありません。Copy について詳しくは、第 4 章の「スタックのみのデータ: Copy」を参照されたし。

Copy トレイトは、プログラマがメソッドをオーバーロードし、任意のコードが実行されないという前提を侵害することを妨げるメソッドは何も定義しません。そのため、全プログラマは、値のコピーは非常に高速であることを前提にすることができます。

部品すべてが Copy を実装する任意の型に対して Copy を導出することができます。Clone も実装する型に対してのみ、Copy トレイトを適用することができます。何故なら、Copy を実装する型には、Copy と同じ作業を行う Clone の瑣末 (さまつ) な実装があるからです。

Copy トレイトは稀にしか必要になりません; Copy を実装する型では最適化が利用可能になります。

つまり、clone を呼び出す必要がなくなり、コードがより簡潔になるということです。

Copy で可能なこと全てが Clone でも達成可能ですが、コードがより遅い可能性や、clone を使用しなければならない箇所があったりします。

値を固定サイズの値にマップする Hash

Hash トレイトにより、任意のサイズの型のインスタンスを取り、そのインスタンスをハッシュ関数で固定サイズの値にマップできます。Hash を導出すると、hash メソッドを実装します。hash の導出された実装は、型の各部品に対して呼び出した hash の結果を組み合わせます。つまり、Hash を導出するには、全フィールドと値も Hash を実装しなければならないということです。

Hash が必要になる例は、HashMap<K, V> にキーを格納し、データを効率的に格納することです。

既定値のための Default

Default トレイトにより、型に対して既定値を生成できます。Default を導出すると、default 関数を実装します。default 関数の導出された実装は、型の各部品に対して default 関数を呼び出します。つまり、Default を導出するには、型の全フィールドと値も Default を実装しなければならないということです。

Default::default 関数は、第 5 章の「構造体更新記法で他のインスタンスからインスタンスを生成する」節で議論した構造体更新記法と組み合わせてよく使用されます。構造体のいくつかのフィールドをカスタマイズし、それから ..Default::default() を使用して、残りのフィールドに対して既定値をセットし使用することができます。

例えば、Default トレイトは、Option<T> インスタンスに対してメソッド unwrap_or_default を使用する時に必要になります。Option<T> が None ならば、メソッド unwrap_or_default は、Option<T> に格納された型 T に対して Default::default の結果を返します。

付録 D - 便利な開発ツール

この付録では、Rust プロジェクトの提供する便利な開発ツールについていくつかお話します。自動フォーマット、警告に対する修正をすばやく適用する方法、lint ツール、そして IDE との統合について見ていきます。

rustfmt を使った自動フォーマット

rustfmt というツールは、コミュニティのコードスタイルに合わせてあなたのコードをフォーマットしてくれます。Rust を書くときにどのスタイルを使うかで揉めないように、多くの共同で行われるプロジェクトが rustfmt を使っています：全員がこのツールでコードをフォーマットするのです。

rustfmt をインストールするには、以下を入力してください：

```
$ rustup component add rustfmt
```

これで rustfmt と cargo-fmt が使えるようになります。これは rustc と cargo の両方のコマンドがあるのと似たようなものです。どんな Cargo のプロジェクトも、次を入力するとフォーマットできます：

```
$ cargo fmt
```

このコマンドを実行すると、現在のクレートのあらゆる Rust コードをフォーマットし直します。これを行うと、コードのスタイルのみが変わり、コードの意味は変わりません。rustfmt についてより詳しく知るにはドキュメントを読んでください。

rustfix でコードを修正する

rustfix というツールは Rust をインストールすると同梱されており、コンパイラの警告 (warning) を自動で直してくれます。Rust でコードを書いたことがある人なら、コンパイラの警告を見たことがあるでしょう。たとえば、下のコードを考えます：

Filename: src/main.rs

```
fn do_something() {}

fn main() {
    for i in 0..100 {
        do_something();
    }
}
```

ここで、do_something 関数を 100 回呼んでいますが、for ループの内部で変数 i を一度も使っていません。Rust はこれについて警告します：

```
$ cargo build
   Compiling myprogram v0.1.0 (file:///projects/myprogram)
warning: unused variable: `i`
 --> src/main.rs:4:9
  |
4 |     for i in 1..100 {
  |         ^ help: consider using `_i` instead
  |
  = note: #[warn(unused_variables)] on by default

    Finished dev [unoptimized + debuginfo] target(s) in 0.50s
```

警告は、変数名に _i を使ってはどうかと提案しています：アンダーバーはその変数を使わないという意図を示すのです。cargo fix というコマンドを実行することで、この提案を rustfix ツールで

自動で適用できます。

```
$ cargo fix
    Checking myprogram v0.1.0 (file:///projects/myprogram)
      Fixing src/main.rs (1 fix)
    Finished dev [unoptimized + debuginfo] target(s) in 0.59s
```

src/main.rs をもう一度見てみると、`cargo fix` によってコードが変更されていることがわかります。

Filename: src/main.rs

```
fn do_something() {}

fn main() {
    for _i in 0..100 {
        do_something();
    }
}
```

`for` ループの変数は `_i` という名前になったので、警告はもう現れません。

`cargo fix` コマンドを使うと、異なる Rust edition の間でコードを変換することもできます。edition については付録 E に書いています。

Clippy でもっと lint を

Clippy というツールは、コードを分析することで、よくある間違いを見つけ、Rust のコードを改善させてくれる lint を集めたものです（訳注：いわゆる静的解析ツール）。

Clippy をインストールするには、次を入力してください：

```
$ rustup component add clippy
```

Clippy の lint は、次のコマンドでどんな Cargo プロジェクトに対しても実行できます：

```
$ cargo clippy
```

たとえば、下のように、円周率などの数学定数の近似を使ったプログラムを書いているとします。

ファイル名: src/main.rs

```
fn main() {
    let x = 3.1415;
    let r = 8.0;
    println!("the area of the circle is {}", x * r * r);
}
```

`cargo clippy` をこのプロジェクトに実行すると次のエラーを得ます：

```
error: approximate value of `f{32, 64}::consts::PI` found. Consider using it
    directly
 --> src/main.rs:2:13
  |
2 |     let x = 3.1415;
  |             ^^^^^^
  |
  = note: #[deny(clippy::approx_constant)] on by default
  = help: for further information visit https://rust-lang-nursery.github.io/rust
      -clippy/master/index.html#approx_constant
```

あなたは、このエラーのおかげで、Rust にはより正確に定義された定数がすでにあり、これを代わりに使うとプログラムがより正しくなるかもしれないと気づくことができます。なので、あなたはコードを定数 `PI` を使うように変更するでしょう。以下のコードはもう Clippy からエラーや警告は受けません。

ファイル名: src/main.rs

```
fn main() {
    let x = std::f64::consts::PI;
    let r = 8.0;
    println!("the area of the circle is {}", x * r * r);
}
```

Clippy についてより詳しく知るには、ドキュメントを読んでください。

Rust Language Server を使って IDE と統合する

IDE での開発の助けになるよう、Rust プロジェクトは **Rust Language Server** (`rls`) を配布しています。このツールは、Language Server Protocol という、IDE とプログラミング言語が対話するための仕様に対応しています。Visual Studio Code の Rust プラグインをはじめ、様々なクライアントが `rls` を使うことができます。

`rls` をインストールするには、以下を入力してください：

```
$ rustup component add rls
```

つづけて、あなたの IDE 向けの language server サポートをインストールしてください。すると、自動補完、定義へのジャンプ、インラインのエラー表示などの機能が得られるはずです。

`rls` についてより詳しく知るにはドキュメントを読んでください。

付録 E：エディション

第 1 章で、`cargo new` がエディションに関するちょっとしたメタデータを **Cargo.toml** に追加しているのを見ましたね。この付録ではその意味についてお話します！

Rust 言語とコンパイラは 6 週間のリリースサイクルを採用しています。つまり、ユーザにはコンスタントに新しい機能が流れてくるというわけです。他のプログラミング言語は、より少ない回数で、より大きなリリースを行いますが、Rust は小さなアップデートを頻繁に行います。しばらくすると、これらの小さな変更が溜まっていきます。しかし、これらを振り返って、「Rust 1.10 と Rust 1.31 を比較すると、すごく変わったねえ！」などとリリースごとに言うのは難しいです。

2、3 年ごとに、Rust チームは Rust の新しい**エディション**を作ります。それぞれのエディションには、それまでに Rust にやってきた新しい機能が、完全に更新されたドキュメントとツール群とともに、一つのパッケージとなってまとめられています。新しいエディションは通常の 6 週間ごとのリリースの一部として配布されます。

それぞれの人々にとってエディションは異なる意味を持ちます。

- アクティブな Rust ユーザにとっては、新しいエディションは、少しずつ増えてきた変更点を理解しやすいパッケージにしてまとめるものです。
- Rust ユーザでない人にとっては、新しいエディションは、何かしら大きな達成がなされたことを示します。Rust には今一度目を向ける価値があると感じさせるかもしれません。
- Rust を開発している人にとっては、新しいエディションは、プロジェクト全体の目標地点となります。

この文書を書いている時点（訳注：原文のコミットは 2021 年 12 月 23 日）では、3 つの Rust のエディションが利用できます。Rust 2015、Rust 2018、Rust 2021 です。この本は Rust 2021 エディションの慣例に従って書かれています。

Cargo.toml における `edition` キーは、コードに対してコンパイラがどのエディションを適用すべきかを示しています。もしキーが存在しなければ、Rust は後方互換性のため `2015` をエディションの値として使います。

標準の 2015 エディション以外のエディションを使うという選択はそれぞれのプロジェクトですることができます。エディションには、コード内の識別子と衝突してしまう新しいキーワードの導入など、互換性のない変更が含まれる可能性があります。しかし、それらの変更を選択しない限り、Rust のコンパイラのバージョンを更新しても、コードは変わらずコンパイルできます。

Rust コンパイラは全バージョンにおいて、そのコンパイラのリリースまでに存在したすべてのエディションをサポートしており、またサポートされているエディションのクレートはすべてリンクできます。エディションの変更はコンパイラが最初にコードを構文解析するときにのみ影響します。なので、あなたが Rust 2015 を使っていて、依存先に Rust 2018 を使うものがあったとしても、あなたのプロジェクトはコンパイルでき、その依存先を使うことができます。逆に、あなたのプロジェク

トが Rust 2018 を、依存先が Rust 2015 を使っていても、同じく問題はありません。

まあ実のところ、ほとんどの機能はすべてのエディションで利用可能でしょう。どの Rust エディションを使っている開発者も、新しい安定リリースが出ると改善したなと感じるのは変わらないでしょう。しかし、場合によって（多くは新しいキーワードが追加されたとき）は、新機能が新しいエディションでしか利用できないことがあるかもしれません。そのような機能を利用したいなら、エディションを切り替える必要があるでしょう。

より詳しく知りたいなら、**エディションガイド**という、エディションに関するすべてを説明している本があります。エディション同士の違いや、`cargo fix` を使って自動的にコードを新しいエディションにアップグレードする方法が書かれています。

> 訳注：日本語版のエディションガイドはこちらにあります。

付録 F: 本の翻訳

英語以外の言語の資料についてです。ほとんどはまだ翻訳中です。手助けいただける際や、新しい翻訳について教えていただける際は、Translations ラベルを確認してください！

- Português (BR)
- Português (PT)
- 简体中文
- Українська
- Español, alternate
- Italiano
- Русский
- 한국어
- 日本語
- Français
- Polski
- עברית
- Cebuano
- Tagalog
- Esperanto
- ελληνική
- Svenska
- Farsi
- Deutsch

付録 G: Rust の作られ方と “Nightly Rust”

この付録は、Rust のでき方と、それが Rust 開発者としてあなたにどう影響するかについてです。この本の出力は安定版 Rust 1.21.0 で生成されていますが、コンパイルできるいかなる例も、それより新しい Rust のどんな安定版でもコンパイルでき続けられるはずということに触れました。この節は、これが本当のことであると保証する方法を説明します！

停滞なしの安定性

言語として、Rust はコードの安定性について**大い**に注意しています。Rust には、その上に建築できる岩のように硬い基礎であってほしく、物事が定期的に変わっていたら、それは実現できません。同時に新しい機能で実験できなければ、もはや何も変更できないリリースの時まで、重大な瑕疵 (かし) を発見できなくなるかもしれません。

この問題に対する我々の解決策は「停滞なしの安定性」と呼ばれるもので、ガイドの原則は以下の通りです: 安定版 Rust の新しいバージョンにアップグレードするのを恐れる必要は何もないはずです。各アップグレードは痛みのないもののはずですが、新しい機能、より少ないバグ、高速なコンパイル時間も齎すべきです。

シュポシュポ！ リリースチャンネルと列車に乗ること

Rust 開発は、**電車のダイヤ**に合わせて処理されます。つまり、全開発は Rust リポジトリの `master` ブランチで行われます。リリースはソフトウェアのリリーストレインモデル (software release train model) に従い、これは Cisco IOS や他のソフトウェアプロジェクトで活用されています。Rust には**リリースチャンネル**が 3 つあります:

> 注釈: software release train model とは、あるバージョンのソフトウェアリリースの順番を列車に見立て、列車のダイヤのように、決まった間隔でリリースに持って行く手法のことの模様。一つの列車は、Rust の場合、ナイトリー、ベータ、安定版の順に「駅」に停車していくものと思われる。

- ナイトリー
- ベータ
- 安定版

多くの Rust 開発者は主に安定版チャンネルを使用しますが、新しい実験的な機能を試したい方は、ナイトリーやベータを使用するかもしれません。

こちらが、開発とリリースプロセスの動き方の例です: Rust チームが Rust1.5 のリリースに取り掛かっていると想定しましょう。そのリリースは、2015 年の 11 月に発生しましたが、現実的なバー

ジョンナンバーを与えてくれるでしょう。新しい機能が Rust に追加されます: 新しいコミットが master ブランチに着地します。毎晩、新しいナイトリ版の Rust が生成されます。毎日がリリース日で、これらのリリースは、リリースインフラにより自動で作成されます。故に、時間が経てばリリースは、毎晩 1 回、以下のような見た目になります:

```
nightly: * - - * - - *
```

6 週間ごとに、新しいリリースを準備するタイミングになります！ Rust リポジトリの beta ブランチが、ナイトリで使用される master ブランチから枝分かれします。さて、リリースが二つになりました:

```
nightly: * - - * - - *
                    |
beta:               *
```

ほとんどの Rust ユーザはベータリリースを積極的には使用しませんが、自身の CI システム内でベータに対してテストを行い、Rust が不具合の可能性を発見するのを手伝います。その間も、やはりナイトリリリースは毎晩あります:

> 注釈: CI は Continuous Integration(継続統合といったところか) のことと思われる。開発者のコードを 1 日に何度も、メインのブランチに統合することらしい。

```
nightly: * - - * - - * - - * - - *
                    |
beta:               *
```

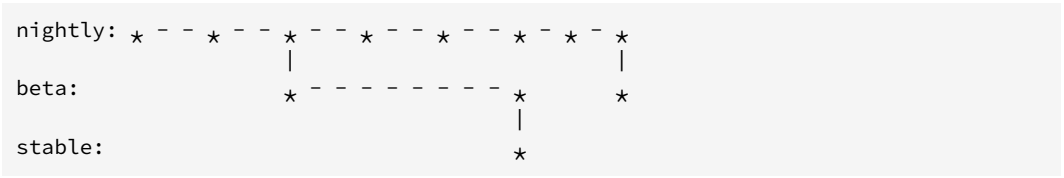
不具合が見つかったとしましょう。よいことに、不具合が安定版のリリースにこっそり持ち込まれる前にベータリリースをテストする時間がありました！ 修正が master に適用されるので、ナイトリは修正され、それから修正が beta ブランチにバックポートされ、ベータの新しいリリースが生成されます:

```
nightly: * - - * - - * - - * - - * - - *
                    |
beta:               * - - - - - - - - *
```

最初のベータが作成されてから 6 週間後、安定版のリリースの時間です！ stable ブランチが beta ブランチから生成されます:

```
nightly: * - - * - - * - - * - - * - - * - * - *
                    |
beta:               * - - - - - - - - *
                                      |
stable:                               *
```

やりました！ Rust1.5 が完了しました！ ですが、1 つ忘れていることがあります: 6 週間が経過したので、**次**のバージョンの Rust(1.6) の新しいベータも必要です。従って、`stable` が `beta` から枝分かれした後に、次のバージョンの `beta` が `nightly` から再度枝分かれします:



これが「トレイン・モデル」と呼ばれます。6 週間ごとにリリースが「駅を出発する」からですが、安定版リリースとして到着する前にベータチャンネルの旅をそれでもしなければなりません。

Rust は 6 週間ごとに時計仕掛けのようにリリースされます。ある Rust リリースの日付を知っていれば、次のリリースの日付もわかります: 6 週間後です。6 週間ごとにリリースを組むことのいい側面は、次の列車がすぐにやってくることです。ある機能が偶然、特定のリリースを逃しても、心配する必要はありません: 別のリリースがすぐに起きます！ これにより、リリースの締め切りが近い洗練されていない可能性のある機能をこっそり持ち込むプレッシャーが減る助けになるのです。

このプロセスのおかげで、Rust の次のビルドを常に確認し、アップグレードするのが容易であると自身に対して確かめることができます: ベータリリースが予想した通りに動かなければ、チームに報告して、次の安定版のリリースが起きる前に直してもらうことができるのです！ ベータリリースでの破損はどちらかといえば稀ですが、`rustc` もソフトウェアの一種であり、バグは確実に存在します。

安定しない機能

このリリースモデルにはもう一つ掴み所があります: 安定しない機能です。Rust は「機能フラグ」と呼ばれるテクニックを使用して、あるリリースで有効にする機能を決定します。新しい機能が活発に開発中なら、`master` に着地し、故にナイトリーでは**機能フラグ**の背後に存在します。ユーザとして、絶賛作業中の機能を試したいとお望みならば、可能ですが、ナイトリリリースの Rust を使用し、ソースコードに適切なフラグを注釈して同意しなければなりません。

ベータか安定リリースの Rust を使用しているなら、機能フラグは使用できません。これが、永遠に安定であると宣言する前に、新しい機能を実用に供することができる鍵になっています。最先端を選択するのをお望みの方はそうすることができ、岩のように硬い経験をお望みの方は、安定版に執着し自分のコードが壊れることはないとわかります。停滞なしの安定性です。

この本は安定な機能についての情報のみ含んでいます。現在進行形の機能は、変化中であり、確実にこの本が執筆された時と安定版ビルドで有効化された時で異なるからです。ナイトリ限定の機能についてのドキュメンテーションは、オンラインで発見できます。

Rustup と Rust ナイトリの役目

rustup は、グローバルかプロジェクトごとに Rust のリリースチャンネルを変更しやすくしてくれます。標準では、安定版の Rust がインストールされます。例えば、ナイトリをインストールするには:

```
$ rustup install nightly
```

rustup でインストールした全ツールチェーン (Rust のリリースと関連するコンポーネント) も確認できます。こちらは、著者の一人の Windows コンピュータの例です:

```
> rustup toolchain list
stable-x86_64-pc-windows-msvc (default)
beta-x86_64-pc-windows-msvc
nightly-x86_64-pc-windows-msvc
```

おわかりのように、安定版のツールチェーンが標準です。ほとんどの Rust ユーザは、ほとんどの場合、安定版を使用します。あなたもほとんどの場合安定版を使用したい可能性がありますが、最前線の機能が気になるので、特定のプロジェクトではナイトリを使用したいかもしれません。そうするためには、そのプロジェクトのディレクトリで rustup override を使用して、そのディレクトリにいる時に、rustup が使用するべきツールチェーンとしてナイトリ版のものをセットします。

```
$ cd ~/projects/needs-nightly
$ rustup override set nightly
```

これで**/projects/needs-nightly** 内で rustc や cargo を呼び出す度に、rustup は既定の安定版の Rust ではなく、ナイトリ Rust を使用していることを確かめます。Rust プロジェクトが大量にある時には、重宝します。

RFC プロセスとチーム

では、これらの新しい機能をどう習うのでしょうか？ Rust の開発モデルは、**Request For Comments (RFC; コメントの要求) プロセス**に従っています。Rust に改善を行いたければ、RFC と呼ばれる提案を書き上げます。

誰もが RFC を書いて Rust を改善でき、提案は Rust チームにより査読され議論され、このチームは多くの話題のサブチームから構成されています。Rust の Web サイトにはチームの完全なリストがあり、プロジェクトの各分野のチームも含みます: 言語設計、コンパイラ実装、インフラ、ドキュメンテーションなどです。適切なチームが提案とコメントを読み、自身のコメントを書き、最終的にその機能を受け入れるか拒否するかの同意があります。

機能が受け入れられれば、Rust リポジトリで issue が開かれ、誰かがそれを実装します。うまく実装できる人は、そもそもその機能を提案した人ではないかもしれません！ 実装の準備ができたら、「安定しない機能」節で議論したように、機能ゲートの背後の master に着地します。

時間経過後、一旦ナイトリリリースを使用する Rust 開発者が新しい機能を試すことができたら、チームのメンバーがその機能と、ナイトリでどう機能しているかについて議論し、安定版の Rust に導入すべきかどうか決定します。決定が進行させることだったら、機能ゲートは取り除かれ、その機能はもう安定と考えられます！ Rust の新しい安定版リリースまで、列車に乗っているのです。