The Rust Programming Language

The Rust Team

2017-07-19

目次

1

Introduction

ようこそ!この本はプログラミング言語 Rust[*1]の教材です。Rust は安全性、速度、並行性の 3 つのゴールにフォーカスしたシステムプログラミング言語です。ガーベジコレクタなしにこれらのゴールを実現していて、他の言語への埋め込み、要求された空間や時間内での動作、デバイスドライバやオペレーティングシステムのような低レベルなコードなど他の言語が苦手とする多数のユースケースを得意とします。全てのデータ競合を排除しつつも実行時オーバーヘッドのないコンパイル時の安全性検査を多数持ち、これらの領域をターゲットに置く既存の言語を改善します。Rust は高級言語のような抽象化も含めた「ゼロコスト抽象化」も目標としています。そうでありつつもなお低級言語のような精密な制御も許します。

「プログラミング言語 Rust」はいくつかの章に分かれています。このイントロダクションが一番最初の章です。この後は

- はじめる - Rust 開発へ向けた環境構築です。
- チュートリアル：数当てゲーム - 小さなプロジェクトを通して Rust について学びます。
- シンタックスとセマンティクス- Rust について一歩ずつ、小さく分割しながらやっていきます。
- Effective Rust - 良い Rust のコードを書くための高レベルな概念です。
- Nightly Rust - 安定版の Rust では使えない Rust の最前線の機能です。
- 用語集 - 本書で使われる用語の参考です。

[*1] https://www.rust-lang.org

- 関係書目 - Rust へ影響を与えたもの、Rust に関する論文です。

貢献する

本書を生成するのに使われたソースは以下から入手出来ます GitHub[*2].

訳注: 日本語の翻訳文書は以下から入手出来ます。GitHub[*3].

[*2] https://github.com/rust-lang/rust/tree/master/src/doc/book
[*3] https://github.com/rust-lang-ja/the-rust-programming-language-ja/tree/master/1.9/ja/book

2

はじめる

この最初の章では、Rust とツールについて、はじめの一歩を踏み出します。最初に Rust をインストールします。そしてお決まりの「Hello World」をやります。最後に Cargo という、Rust のビルドシステム兼パッケージマネージャについて学びます。

Rust のインストール

Rust を使い始める最初のステップはインストールです。このセクションでは、コマンドでインターネットから Rust をダウンロードしますので、インターネットへの接続が必要です。

コマンドを色々提示しますが、それらは全て `$` から始まります。`$` を入力する必要はありません。 `$` はただコマンドの先頭を示しているだけです。 これから、Web 上でも「`$` で始まるものは一般ユーザで実行し `#` で始まるものは管理者権限で実行する」というルールに従ったチュートリアルや例をよく見ることになるでしょう。

プラットフォームのサポート

Rust のコンパイラは様々なプラットフォーム上で動き、また、他のプラットフォーム向けにもコンパイルできます。しかし、全てのプラットフォームが等しくサポートされているわけではありません。Rust のサポートレベルは 3 階級に分かれていて、それぞれ違う保証をします。

プラットフォームは「ターゲットトリプル」という文字列によって識別されます。これは、どの種類のアウトプットを生成すべきかをコンパイラに伝えるためのものです。下の表は対応するコンポーネントがそのプラットフォームで動作するかを示します。

■1 級

1 級のプラットフォームは「ビルドでき、かつ動くことを保証する」ものとされています。特に以下の要求それぞれを満たします。

- 自動テストがそのプラットフォーム上で走るようセットアップされている
- `rust-lang/rust` レポジトリの master ブランチへの変更はテストが通ってからされる
- 公式のリリースがそのプラットフォーム向けに提供される
- 使用方法及びビルド方法のドキュメントがある

Target	std	rustc	cargo	notes
`i686-pc-windows-msvc`	✓	✓	✓	32-bit MSVC (Windows 7+)
`x86_64-pc-windows-msvc`	✓	✓	✓	64-bit MSVC (Windows 7+)
`i686-pc-windows-gnu`	✓	✓	✓	32-bit MinGW (Windows 7+)
`x86_64-pc-windows-gnu`	✓	✓	✓	64-bit MinGW (Windows 7+)
`i686-apple-darwin`	✓	✓	✓	32-bit OSX (10.7+, Lion+)
`x86_64-apple-darwin`	✓	✓	✓	64-bit OSX (10.7+, Lion+)
`i686-unknown-linux-gnu`	✓	✓	✓	32-bit Linux (2.6.18+)
`x86_64-unknown-linux-gnu`	✓	✓	✓	64-bit Linux (2.6.18+)

■2 級

2 級のプラットフォームは「ビルドを保証する」ものとされています。自動テストは走っておらず、ビルドできたとしてもちゃんと動く保証はありません。しかし、大抵の場合、ほぼ動きますし、パッチはいつでも歓迎しています! 特に、以下が要請されています。

- 自動ビルドはセットアップされているがテストは走っていないかもしれない

- rust-lang/rust レポジトリの master ブランチへの変更はビルドが 通ってからされる。これは、標準ライブラリしかコンパイルできないものもあれば、完全なブートストラップまでできるものもある、ということを意味しますので注意してください。
- 公式のリリースがそのプラットフォーム向けに提供される

Target	std	rustc	cargo	notes
x86_64-unknown-linux-musl	✓			64-bit Linux with MUSL
arm-linux-androideabi	✓			ARM Android
arm-unknown-linux-gnueabi	✓	✓		ARM Linux (2.6.18+)
arm-unknown-linux-gnueabihf	✓	✓		ARM Linux (2.6.18+)
aarch64-unknown-linux-gnu	✓			ARM64 Linux (2.6.18+)
mips-unknown-linux-gnu	✓			MIPS Linux (2.6.18+)
mipsel-unknown-linux-gnu	✓			MIPS (LE) Linux (2.6.18+)

■3 級

3 級のプラットフォームはサポートはされているものの、テストやビルドによる変更の管理は行なっていないものたちです。コミュニティの貢献度で信頼性が定まるので、ビルドが通るかはまちまちです。さらに、リリースやインストーラは提供されません。しかしコミュニティが、非公式な場所にリリースやインストーラを作るためのインフラを持っているかもしれません。

Target	std	rustc	cargo	notes
i686-linux-android	✓			32-bit x86 Android
aarch64-linux-android	✓			ARM64 Android
powerpc-unknown-linux-gnu	✓			PowerPC Linux (2.6.18+)
powerpc64-unknown-linux-gnu	✓			PPC64 Linux (2.6.18+)
powerpc64le-unknown-linux-gnu	✓			PPC64LE Linux (2.6.18+)
armv7-unknown-linux-gnueabihf	✓			ARMv7 Linux (2.6.18+)
i386-apple-ios	✓			32-bit x86 iOS

Target	std	rustc	cargo	notes
`x86_64-apple-ios`	✓			64-bit x86 iOS
`armv7-apple-ios`	✓			ARM iOS
`armv7s-apple-ios`	✓			ARM iOS
`aarch64-apple-ios`	✓			ARM64 iOS
`i686-unknown-freebsd`	✓	✓	✓	32-bit FreeBSD
`x86_64-unknown-freebsd`	✓	✓	✓	64-bit FreeBSD
`x86_64-unknown-openbsd`	✓	✓		64-bit OpenBSD
`x86_64-unknown-netbsd`	✓	✓		64-bit NetBSD
`x86_64-unknown-bitrig`	✓	✓		64-bit Bitrig
`x86_64-unknown-dragonfly`	✓	✓		64-bit DragonFlyBSD
`x86_64-rumprun-netbsd`	✓			64-bit NetBSD Rump Kernel
`x86_64-sun-solaris`	✓	✓		64-bit Solaris/SunOS
`i686-pc-windows-msvc` (XP)	✓			Windows XP support
`x86_64-pc-windows-msvc` (XP)	✓			Windows XP support

この表は時間と共に拡張されるかもしれないことに注意してください。これから存在する全ての 3 級のプラットフォームは網羅していないのです!

Linux または Mac でのインストール

Linux か Mac を使っているなら、ターミナルを開いて、以下のように入力するだけで済みます。

```
$ curl -sSf https://static.rust-lang.org/rustup.sh | sh
```

訳注: (Rust 1.14.0 以降)
rustup のインストール方法は変更されました。代わりに以下を入力して下さい。

```
curl https://sh.rustup.rs -sSf | sh
```

このコマンドでスクリプトをダウンロードしインストールを始めます。全てが上手くいったなら、以下のように表示されるはずです。

```
Rust is ready to roll.
```

訳注:

Rust を使う準備ができました。

(Rust 1.14.0 以降)
全てがうまくいったなら、以下のように表示されるはずです。

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

Rust はたった今インストールされました。すばらしい！

ここで「はい」の意味で y を押しましょう。そして以後の画面の指示に従ってください。

Windows でのインストール

Windows を使っているなら適切なインストーラ[*1]をダウンロードしてください。

訳注: (Rust 1.14.0 以降)
Windows にインストールするのは同じくらい簡単です。 [rustup-init.exe] をダウンロードし実行して下さい。コンソールにてインストールが始まり、成功すれば前述のメッセージが出ているでしょう。
他のインストールオプションや情報については、Rust のウェブサイトのインストール[*2]ページにアクセスして下さい。

アンインストール

Rust のアンインストールはインストールと同じくらい簡単です。Linux か Mac ならアンインストールスクリプトを使うだけです。

[*1] https://www.rust-lang.org/install.html
[*2] https://www.rust-lang.org/install.html

```
$ sudo /usr/local/lib/rustlib/uninstall.sh
```

Windows のインストーラを使ったなら `.msi` をもう一度実行すれば、アンインストールのオプションが出てきます。

> 訳注: (Rust 1.14.0 以降)
> Rust のアンインストール方法も変更されています。以下のコマンドを入力して下さい。
>
> ```
> rustup self uninstall
> ```

トラブルシューティング

既に Rust をインストールしているなら、シェルを開いて以下を打ちましょう。

```
$ rustc --version
```

バージョン番号、コミットハッシュ、そしてコミット日時が表示されるはずです。

表示されたなら Rust はちゃんとインストールされています!おめでとう!

Windows を使っていて、表示されないなら、%PATH% システム変数に Rust が入っているか確認してください。入っていなければもう一度インストーラを実行し、「Change, repair, or remove installation」ページの「Change」を選択し、「Add to PATH」が、ローカルのハードドライブにインストールされていることを確認してください。

Rust にはプログラムをリンクする機能がありませんので、別途、リンカもインストールしないといけないでしょう。インストール方法は、お使いのシステムによって異なります。詳細については、そのシステムのドキュメントを参照してください。

もし上手くいかないなら、様々な場所で助けを得られます。最も簡単なのは、irc.mozilla.org の#rust-beginners チャネル[*3]ですし、一般的な話題なら、irc.mozilla.org の#rust チャネル[*4]もあります。どちらも Mibbit[*5]からアクセスできます。リンクをクリックしたら Rustacean 達 (我々のことをふざけてこう呼ぶのです) につながりますので、チャット通して助けてもらえるでしょう。他には、ユーザフォーラ

[*3] irc://irc.mozilla.org/#rust-beginners
[*4] irc://irc.mozilla.org/#rust
[*5] http://chat.mibbit.com/?server=irc.mozilla.org&channel=%23rust-beginners,%23rust

ム[*6]や Stack Overflow[*7]などがあります。

> 訳注: Rust について、日本語で会話したり、質問したりできる場所もあります。
> - Slack の rust-jp チーム[*8]（参加登録はこちら[*9]）
> - teratail / テラテイル[*10]
> - Stack Overflow ja / スタック・オーバーフロー[*11]

インストーラはドキュメントのコピーもローカルにインストールしますので、オフラインで読めます。UNIX システムでは `/usr/local/share/doc/rust` にあります。Windows では Rust をインストールした所の `share/doc` ディレクトリにあります。

Hello, world!

Rust をインストールしたので最初の Rust のプログラムを書いていきましょう。新しい言語を学ぶ時に「Hello, world!」とスクリーンに表示する小さなプログラムを書くのが伝統で、このセクションでもそれに従います。

このように小さなプログラムから始める利点は、コンパイラがインストールされていて、正しく動くことを素早く確認できることです。情報をスクリーンに表示することも非常によくやるので、早い内に練習しておくのが良いです。

> 留意: 本書はコマンドラインをある程度使えることを仮定しています。Rust 本体はコードの編集やツール群、コードの置き場には特に要求を設けませんので、コマンドラインより IDE を好むならそうしても構いません。Rust を念頭に置いて作られた IDE、[SolidOak] を試してみるといいかもしれません。コミュニティにより多数のエクステンション（機能拡張）が開発されていますし、Rust チームも様々なエディタ[*12]向けにプラグインを用意しています。このチュートリアルではエディタや IDE の設定は扱いませんので、それぞれに合ったドキュメントを参照してください。

[*6] https://users.rust-lang.org/
[*7] http://stackoverflow.com/questions/tagged/rust
[*8] https://rust-jp.slack.com/
[*9] http://rust-jp.herokuapp.com/
[*10] https://teratail.com/tags/Rust
[*11] https://ja.stackoverflow.com/questions/tagged/rust
[*12] https://github.com/rust-lang/rust/blob/master/src/etc/CONFIGS.md

プロジェクトファイルを作る

まず、Rust のコードを書くファイルを用意します。Rust はコードがどこにあるかは気にしませんが、本書を進めるにあたってホームディレクトリ直下に *projects* ディレクトリを作って、全てのプロジェクトをそのディレクトリ下に入れることをお勧めます。ターミナルを開いて以下のコマンドを入力し、今回のプロジェクトのディレクトリを作りましょう。

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

> 留意: Windows でかつ PowerShell を使っていないのなら~ は上手く動かないかもしれません。使っているシェルのドキュメントをあたってみてください。

Rust のコードを書いて走らせる

次に、新しいソースファイルを作り、それを *main.rs* としましょう。Rust のファイルは常に *.rs* 拡張子で終わります。ファイル名に 1 つ以上の単語を使うならアンダースコアで区切りましょう。例えば、*helloworld.rs* ではなく *hello_world.rs* を使うことになります。

それでは、いま作った *main.rs* を開いて、以下のコードを打ちましょう。

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

ファイルを保存して、ターミナルのウィンドウに戻ります。Linux か OSX では以下のコマンドを入力します。

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

Windows では `main` を `main.exe` と読み替えてください。使っている OS に関わらず、 `Hello, world!`

の文字列がターミナルに印字されるのを目にするはずです。目にしたなら、おめでとうございます！ あなたは正式に Rust のプログラムを記述しました。これであなたも Rust プログラマです！ ようこそ。

Rust プログラムの構造

さて、「Hello, world!」プログラムで何が起きていたのか、詳しく見ていきましょう。パズルの最初のピースがこれです。

```
fn main() {


}
```

これらの行は Rust の **関数** を定義します。 `main` 関数は特別で、全ての Rust プログラムの開始点になります。最初の行は「引数を取らず、返り値も返さない関数 `main` を宣言します」といっています。 引数があれば、括弧 (`(` と`)`) の中に入りますし、今回はこの関数から何も値を返さないので、返り値の型を完全に省略できます。

さらに、関数の本体部が波括弧 (`{` と`}`) で括られていることに留意してください。Rust は全ての関数の本体部に波括弧を要求します。関数宣言と同じ行にスペースを 1 つ空けて開き波括弧を置くのが、良いスタイルとされます。

`main()` 関数の中では

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

この行が今回の小さなプログラムの全てを担っています。これがテキストをスクリーンに印字するのです。ここに重要な詳細がいくつもあります。1 つ目はインデントが 4 スペースであり、タブでない点です。

2 つ目の重要な部分は `println!()` の行です。これは Rust のメタプログラミング機構、**マクロ** の呼び出しです。もし関数を呼び出しているのなら、 `println()` のようになります (`!` がありません)。Rust のマクロについては、後の章で詳細に議論しますが、今のところ`!` を見たら、普通の関数ではなくマクロを呼び出していることを意味する、ということだけ知っておいてください。

次は 文字列の `"Hello, world"` です。システムプログラミング言語では文字列は驚くほど複雑なトピックで、これは**静的に確保された**文字列です。 文字列をスクリーンに印字してくれる `println!` にこれを引数として渡します。簡単ですね！

行はセミコロン (`;`) で終わります。Rust は**式指向言語**で、ほとんどのものは文ではなく式になります。`;` は式が終わり、次の式が始まることを示します。Rust のコードの大半の行は`;` で終わります。

コンパイルと実行は別の手順

「Rust のプログラムを書いて走らせる」で、新しく作ったプログラムをどうやって実行するか示しました。それぞれを分解して手順毎に見ていきましょう。

Rust のプログラムを走らせる前に、コンパイルする必要があります。Rust のコンパイラはこのように `rustc` と入力して、ソースファイルの名前を渡してあげることで使えます。

```
$ rustc main.rs
```

C または C++ のバックグラウンドを持つならこれが `gcc` や `clang` に似ていることに気付くでしょう。コンパイルが成功したら、Rust は実行可能バイナリを出力したはずです。Linux か OSX なら以下のように `ls` コマンドで確認できます。

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

Windows なら、こうなります。

```
$ dir
main.exe  main.rs
```

2 つのファイルがあるといっています。 `.rs` 拡張子を持ったソースコードと実行可能ファイル (Windows では `main.exe` 、それ以外では `main`)。 あとは `main` または `main.exe` ファイルを、このように実行するだけです。

```
$ ./main  # あるいは Windows なら main.exe
```

もし *main.rs* が「Hello, world!」プログラムなら、これで `Hello, world!` とターミナルに印字することでしょう。

もし Ruby や Python、JavaScript などの動的な言語から来たのなら、コンパイルと実行が別の手順になっていることに馴れないかもしれません。Rust は、プログラムをコンパイルして、それを別の誰かに渡したら、Rust がなくても動く**事前コンパイル** 言語です。 それと対照的に、別の誰かに `.rb` や `.py` 、`.js` を渡したら (それぞれ)Ruby、Python あるいは JavaScript の実装が必要になりますが、コンパイルにも実行にも 1 つのコマンドで事足ります。全ては言語設計上のトレードオフです。

単純なプログラムなら単に `rustc` でコンパイルすれば十分ですが、プロジェクトが大きくなるにつれて、プロジェクトの全てのオプションを管理したり、他の人やプロジェクトと容易に共有できるようにしたくなるでしょう。次は現実世界の Rust プログラムを書く手助けになる、Cargo というツールを紹介します。

Hello, Cargo!

Cargo は Rust のビルドシステムであり、パッケージマネージャであり、Rustacean は Cargo を Rust プロジェクトの管理にも使います。Cargo は 3 つのものを管理します。それは、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です。あなたのコードが必要とするライブラリを、「依存 (dependencies)」と呼びます。なぜならコードがそれに依存しているからです。

最も簡単な Rust のプログラムは依存を持たないので、ここでは Cargo の 1 つ目の機能だけを使います。もっと複雑な Rust のコードを書くにつれて、依存を追加したくなるでしょうが、Cargo を使えばそれがとても簡単にできます。

ほとんどの Rust のプロジェクトが Cargo を使うので、本書でもこれ以降は Cargo を使うことを前提とします。公式のインストーラを使ったなら、Cargo は Rust に同梱されています。他の手段で Rust をインストールしたなら、ターミナルで以下のコマンドを打てば、Cargo がインストールされているか確認できます。

```
$ cargo --version
```

バージョン番号が表示されれば大丈夫です。 もし「**コマンドが見つかりません**」などのエラーが出たなら、Rust をインストールしたシステムのドキュメントを見て、Cargo が別になっているか調べましょう。

Cargo へ変換する

それでは、Hello World プログラムを Cargo に変換しましょう。プロジェクトを Cargo 化するには 3 つのことをする必要があります。

1. ソースファイルを正しいディレクトリに置く
2. 古い実行可能ファイル (Windows なら `main.exe` 、他では `main`) を削除し、新しいものを作る
3. Cargo の設定ファイルを作る

やっていきましょう!

■新しい実行可能ファイルとソースディレクトリを作る

まずターミナルに戻って、 *hello_world* ディレクトリに移動し、次のコマンドを打ちます。

```
$ mkdir src
$ mv main.rs src/main.rs
$ rm main  # Windows なら'del main.exe' になります
```

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ているので、まずそうしましょう。README やライセンス情報、他のコードに関係ないものは、プロジェクト (このケースでは *hello_world*) 直下に残したままになります。このように、Cargo を使うことで、プロジェクトを綺麗に整頓された状態を保てます。すべてのものには場所があり、すべてが自身の場所に収まります。

では *main.rs* を *src* ディレクトリへ移動して、また `rustc` でコンパイルして作ったファイルを削除します。これまで通り、Windows なら `main` を `main.exe` に読み替えてください。

今回の例では実行可能ファイルを作るので、 `main.rs` の名前を引き続き使います。 もしライブラリを作りたいなら、`lib.rs` という名前にすることになります。この規約は Cargo でプロジェクトを正しくコンパイルするのに使われていますが、必要なら変更できます。

■設定ファイルを作る

次に *hello_world* ディレクトリ下にファイルを作ります。それを `Cargo.toml` とします。

ここで `Cargo.toml` の `C` が大文字になっていることを確認しましょう。そうしないと Cargo が設定ファイルだと認識してくれません。

このファイルは *[TOML]* (Tom's Obvious, Minimal Language ([訳注] Tom の理解しやすい、極小な言語)) フォーマットで書かれます。TOML は INI に似ていますが、いくつかの素晴しい機能が追加されており、Cargo の設定フォーマットとして使われています。

ファイル内に以下の情報を打ち込みます。

```
[package]

name = ”hello_world”
version = ”0.0.1”
authors = [ ” あなたの名前 <you@example.com>” ]
```

最初の行 `[package]` は下に続く記述がパッケージの設定であることを示します。さらなる情報をこのファイルに追加する時には、他のセクションを追加することになりますが、今のところパッケージの設定しかしていません。

残りの 3 行は Cargo がプログラムをコンパイルする時に必要な情報です。プログラムの名前、バージョン、そして著者です。

これらの情報を *Cargo.toml* ファイルに追加し終わったら、保存して設定ファイルの作成は終了です。

Cargo プロジェクトのビルドと実行

Cargo.toml をプロジェクトのルートディレクトリに置いたら、Hello World プログラムのビルドと実行の準備が整っているはずです! 以下のコマンドを入力しましょう。

```
$ cargo build
   Compiling hello_world v0.0.1 (file:///home/yourname/projects/hello_world)
$ ./target/debug/hello_world
Hello, world!
```

バババーン!全てが上手くいったら、もう一度 `Hello, world!` がターミナルに印字されるはずです。

`cargo build` でプロジェクトをビルドして`./target/debug/hello_world` でそれを実行したのですが、実は次のように `cargo run` 一発でそれらを実行できます。

```
$ cargo run
     Running `target/debug/hello_world`
Hello, world!
```

この例でプロジェクトを再度ビルドしていないことに注意してください。Cargo はファイルが変更されていないことが分かるので、バイナリの実行だけを行います。ソースコードを修正していたら、Cargo は実行する前にプロジェクトを再度ビルドし、あなたはこのようなものを目にしたことでしょう。

```
$ cargo run
   Compiling hello_world v0.0.1 (file:///home/yourname/projects/hello_world)
     Running `target/debug/hello_world`
Hello, world!
```

Cargo はプロジェクトのファイルのどれかが変更されていないか確認し、最後にビルドしてから変更さ

れたファイルがあるときにだけプロジェクトを再度ビルドします。

単純なプロジェクトでは Cargo を使っても、単に `rustc` を使うのとさほど変わないかもしれません。しかし、将来において役に立つでしょう。特にクレートを使い始めた時によく当て嵌ります。クレートは、他の言語で「ライブラリ」や「パッケージ」と呼ばれているものと同じです。複数のクレートで構成された複雑なプロジェクトでは、Cargo にビルドを任せた方がとても簡単になります。Cargo を使えば `cargo build` を実行するだけで正しく動いてくれます。

■リリースビルド

プロジェクトがリリースできる状態になったら `cargo build –release` を使うことで、最適化をかけてプロジェクトをコンパイルできます。最適化することで、Rust のコードは速くなりますが、コンパイル時間は長くなります。このような理由から、開発の時用と、ユーザへ配布する最終版プログラムを作る時用の、2 つのプロファイルが存在するのです。

■`Cargo.lock` とは？

`cargo build` を実行すると *Cargo.lock* という新しいファイルもできます。 中身はこのようになっています。

```
[root]
name = ”hello_world”
version = ”0.0.1”
```

Cargo は *Cargo.lock* でアプリケーションの依存を追跡します。これは Hello World プロジェクトの *Cargo.lock* ファイルです。このプロジェクトは依存を持たないので、ファイルの中身はほとんどありません。実際には自身でこのファイルに触ることはありません。Cargo に任せてしまいます。

できました!ここまでついて来たなら Cargo で `hello_world` をビルドする所までできたはずです。

このプロジェクトはとてもシンプルですが、これから Rust を使っていく上で実際に使うことになるツール類を色々使っています。実際、事実上全ての Rust プロジェクトで、以下のコマンドの変形を使うことになります。

```
$ git clone someurl.com/foo
$ cd foo
$ cargo build
```

新たな Cargo プロジェクトを作る簡単な方法

新たなプロジェクトを始めるのに先の手順を毎回踏む必要はありません! Cargo で即座に開発を始められる骨組だけのプロジェクトを素早く作ることができます。

Cargo で新たなプロジェクトを始めるには、 `cargo new` をコマンドラインに入力します。

```
$ cargo new hello_world --bin
```

ライブラリではなく実行可能アプリケーションを作りたいので、このコマンドは`–bin` を渡しています。 実行可能ファイルはよく **バイナリ**と呼ばれます (なので Unix システムでは `/usr/bin/` に入っています)。

Cargo は 2 つのファイルと 1 つのディレクトリ、 `Cargo.toml` と *main.rs* の入った *src* ディレクトリを生成します。上で作ったのと全く同じ、見たことのある構成ですね。

これさえあれば始められます。まず、 `Cargo.toml` を開きます。このようになっているはずです。

```
[package]

name = ”hello_world”
version = ”0.1.0”
authors = [” あなたの名前 <you@example.com>”]
```

Cargo は引数と `git` の設定を基に *Cargo.toml* に適当な値を埋めます。 Cargo が `hello_world` ディレクトリを `git` レポジトリとして初期化していることにも気付くでしょう。

`src/main.rs` の中身はこのようになっているはずです。

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

Cargo が「Hello World!」を生成したのでコードを書き始められます!

留意: Cargo について詳しく知りたいなら、公式の Cargo ガイド[*13]を見ましょう。全ての機能が

[*13] http://doc.crates.io/guide.html

網羅してあります。

終わりに

この章はこれ以後の本書、そしてあなたが Rust を書いていく上で役に立つ基本を扱いました。ツールについては一歩踏み出したので、これ以降は、Rust 言語自体を扱っていきます。

2 つの選択肢があります。チュートリアル：数当てゲームでプロジェクトを体験するか、シンタックスとセマンティクスでボトムアップに進んでいくかです。経験豊富なシステムプログラマなら「チュートリアル：数当てゲーム」が好みでしょうが、動的なバックグラウンドを持つ人なら他方が馴染むでしょう。違う人同士違う学び方をするのです!自分に合ったものを選びましょう。

3

チュートリアル：数当てゲーム

Rust の学習を始めましょう。このプロジェクトでは、古典的な初心者向けのプログラミングの問題、数当てゲームを実装します。これは次のように動作します。プログラムは 1 から 100 までの数字を 1 つ、ランダムに生成します。そしてあなたに、数字を予想して入力するよう促します。予想値を入力すると、大きすぎる、あるいは、小さすぎるといったヒントを出します。当たったら、おめでとうと言ってくれます。良さそうですか？

この章を通じて、Rust に関するごく基本なことが学べるでしょう。次の章「シンタックスとセマンティクス」では、それぞれのパートについて、より深く学んでいきます。

セットアップ

新しいプロジェクトを作りましょう。プロジェクトのディレクトリへ行ってください。`hello_world` の時にどのようなディレクトリ構成で、どのように `Cargo.toml` を作る必要があったか覚えてますか？ Cargo にはそれらのことをしてくれるコマンドがあるのでした。やってみましょう。

```
$ cd ~/projects
$ cargo new guessing_game --bin
$ cd guessing_game
```

`cargo new` にプロジェクトの名前と、そしてライブラリではなくバイナリを作るので`–bin` フラグを渡し

ます。

生成された Cargo.toml を確認しましょう。

```
[package]

name = ”guessing_game”
version = ”0.1.0”
authors = [” あなたの名前 <you@example.com>”]
```

Cargo はこれらの情報を環境から取得します。もし間違っていたら、どうぞ修正してください。

最後に、Cargo は「Hello, world!」を生成します。 src/main.rs を確認しましょう。

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

Cargo が用意したものをコンパイルしてみましょう。

{bash} $ cargo build Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_-game)

素晴らしい。もう一度 src/main.rs を開きましょう。全てのコードをこの中に書いていきます。

先に進む前に、Cargo のコマンドをもう 1 つ紹介させてください。run です。cargo run は cargo build のようなものですが、生成した実行可能ファイルを走らせてくれます。試してみましょう。

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running `target/debug/guessing_game`
Hello, world!
```

いい感じです。 run コマンドはプロジェクトを細かく回す必要があるときに便利でしょう。今回のゲームがまさにそのようなプロジェクトです。すぐに試してから次の行動に移るといったことを繰り返していきます。

予想値を処理する

では作り始めましょう。数当てゲームで最初にしないといけないのは、プレイヤに予想値を入力させることです。これを `src/main.rs` に書きましょう。

```
use std::io;

fn main() {
    println!("Guess the 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);
}
```

[訳注] それぞれの文言は

- Guess the number!: 数字を当ててみて！
- Please input your guess.: 予想値を入力してください
- Failed to read line: 行の読み取りに失敗しました
- You guessed: {}: あなたの予想値: {}

の意味ですが、エディタの設定などによっては、ソースコード中に日本語を使うとコンパイルできないことがあるので、英文のままにしてあります。

いろいろと出てきましたね。順に見ていきましょう。

```
use std::io;
```

これからユーザの入力を取得して、結果を出力するわけですが、それには、標準ライブラリの中にある `io` ライブラリが必要です。Rust は全てのプログラムに、ごく限られたものをデフォルトでインポート

しますが、これを「プレリュード」[*1]と呼びます。プレリュードにないものは、直接 use する必要があります。なお、2 つ目の「プレリュード」、io プレリュード[*2]もあり、もしそれをインポートすると、 io に関連した多数の有用なものがインポートされます。

```
fn main() {
```

すでに見てきたように main() 関数がプログラムのエントリポイントになります。 fn 構文は新たな関数を宣言し、 () で引数がないことを示し、{ が関数本体の始まりです。 返り値の型は書かなかったので、() 、つまり空のタプルとして扱われます。

```
    println!("Guess the number!");

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

前に println!() が文字列をスクリーンに表示するマクロであることを学びました。

```
    let mut guess = String::new();
```

少し興味深いものが出てきました。このわずか 1 行で、様々なことが起こっています。最初に気付くのは、これが「変数束縛」を作る let 文であることです。let 文は以下の形を取ります。

```
let foo = bar;
```

これは foo という名前の束縛を作り、それを値 bar に束縛します。多くの言語ではこれを「変数」と呼びますが、Rust の変数束縛は少しばかり皮を被せてあります。

例えば、束縛はデフォルトでイミュータブル (不変) です。ですから、この例ではイミュータブルではなく、ミュータブル (可変) な束縛にするために mut を使っているのです。 let は代入の左辺に単に 1 つの名前を取るのではなく、実際にはパターンを受け取ります。パターンは後ほど使います。今のところ、すごく簡単ですね。

```
let foo = 5; // イミュータブル
let mut bar = 5; // ミュータブル
```

[*1] http://doc.rust-lang.org/std/prelude/index.html
[*2] http://doc.rust-lang.org/std/io/prelude/index.html

ああ、そして // から行末までがコメントです。Rust はコメントにある全てのものを無視します。

このように let mut guess がミュータブルな束縛 guess を導入することを知りました。 しかし = の反対側、String::new() が何であるかを見る必要があります。

String は文字列型で、標準ライブラリで提供されています。String[*3]は伸長可能で UTF-8 でエンコードされたテキスト片です。

::new() という構文ですが、これは特定の型に紐づく「関連関数」なので :: を使っています。 つまりこれは、 String のインスタンスではなく、String 自体に関連付けられているということです。これを「スタティックメソッド」と呼ぶ言語もあります。

この関数は新たな空の String を作るので、 new() と名付けられています。 new() 関数はある種の新たな値を作るのによく使われる名前なので、様々な型でこの関数を見るでしょう。

次に進みましょう。

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

いろいろ出てきました。少しずつ確認していきましょう。最初の行は 2 つの部分で構成されます。これが最初の部分です。

```
io::stdin()
```

プログラムの最初の行でどのように std::io を use したか覚えていますか？ ここでは、その関連関数を呼び出しているのです。もし use std::io としなかったなら、 std::io::stdin() と書くことになります。

この関数はターミナルの標準入力へのハンドルを返します。詳しくは std::io::Stdin[*4] を見てください。

次の部分では、ハンドルを使ってユーザからの入力を取得します。

```
.read_line(&mut guess)
```

ここで、ハンドルに対して read_line()[*5]メソッドを呼んでいます。メソッドは関連関数のようなものですが、型自体ではなくインスタンスに対してだけ使えます。read_line() に引数を 1 つ渡してます。

[*3] http://doc.rust-lang.org/std/string/struct.String.html
[*4] http://doc.rust-lang.org/std/io/struct.Stdin.html
[*5] http://doc.rust-lang.org/std/io/struct.Stdin.html#method.read_line

`&mut guess` です。

`guess` がどのように束縛されたか覚えてますか？ ミュータブルであると言いました。 しかしながら、`read_line` は `String` を引数に取りません。 `&mut String` を取るのです。Rust には参照と呼ばれる機能があり、1 つのデータに対して複数の参照を持つことができます。これにより、値をコピーする機会を減らせます。Rust の主要な売りの 1 つが、参照をいかに安全に簡単に使えるかなので、参照は複雑な機能です。しかしこのプログラムを作り終えるのに、今すぐ詳細を知る必要はありません。今のところ `let` と同じように、参照はデフォルトでイミュータブルであるということだけ覚えておいてください。なので `&guess` ではなく `&mut guess` と書く必要があるのです。

なぜ `read_line()` は文字列へのミュータブルな参照を取るのでしょうか？ `read_line()` はユーザが標準入力に打ったものを取得し、それを文字列に格納します。ですから、格納先の文字列を引数として受け取り、そこに入力文字列を追加するために、ミュータブルであることが求められるのです。

しかし、この行はまだ終わっていません。テキスト上では 1 行ですが、コードの論理行の 1 部でしかないのです。

```
        .expect("Failed to read line");
```

メソッドを `.foo()` 構文で呼び出す時、改行してスペースを入れても構いません。そうすることで長い行を分割できます。 こうすることだって**できました**

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

ですがこれだと読み辛いです。そこで 2 つのメソッド呼び出しを、2 つの行に分割したわけです。さて `read_line()` については話しましたが、 `expect()` は何でしょうか？ 実は、`read_line()` は引数として渡した`&mut String` にユーザの入力を入れるだけでなく、`io::Result`[*6] という値も返すのです。 標準ライブラリには `Result` という名の付く型がいくつもあります。まず汎用の `Result`[*7]があって、さらに個々のライブラリに特殊化されたバージョンもあり、`io::Result` もその 1 つです。

これらの `Result` 型の目的は、エラーハンドリング情報をエンコードすることです。`Result` 型の値には、他の型と同じように、メソッドが定義されています。 今回の場合 `io::Result` に `expect()` メソッド[*8]が定義されており、それは、呼び出された値が成功を表すものでなければ、与えたメッセージと共に `panic!` します。`panic!` は、メッセージを表示してプログラムをクラッシュさせます。

[*6] http://doc.rust-lang.org/std/io/type.Result.html
[*7] http://doc.rust-lang.org/std/result/enum.Result.html
[*8] http://doc.rust-lang.org/std/result/enum.Result.html#method.expect

このメソッドを呼び出さずにいると、プログラムはコンパイルできますが、警告が出ます。

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
src/main.rs:10:5: 10:39 warning: unused result which must be used,
#[warn(unused_must_use)] on by default
src/main.rs:10     io::stdin().read_line(&mut guess);
                   ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Rust は `Result` 値を使っていないことを警告します。警告は `io::Result` が持つ特別なアノテーションに由来します。Rust はエラーの可能性があるのに、処理していないことを教えてくれるのです。警告を出さないためには、実際にエラー処理を書くのが正しいやり方です。幸運にも、問題があった時にそのままクラッシュさせたいなら、`expect()` が使えます。どうにかしてエラーから回復したいなら、別のことをしないといけません。しかしそれは、将来のプロジェクトに取っておきましょう。

最初の例も残すところあと 1 行です。

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

これは入力を保持している文字列を表示します。 `{}` はプレースホルダで、引数として `guess` を渡しています。 複数の`{}` があれば、複数を引数を渡すことになります。

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

println!("x and y: {} and {}", x, y);
```

簡単ですね。

いずれにせよ、一巡り終えました。これまでのものを `cargo run` で実行できます。

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running `target/debug/guessing_game`
Guess the number!
Please input your guess.
```

```
6
You guessed: 6
```

これでよし！ 最初の部分は終わりました。キーボードからの入力を取得して、出力を返すところまでできました。

秘密の数を生成する

次に秘密の数を生成しましょう。Rust の標準ライブラリには乱数の機能がまだありません。ですが、Rust チームは `rand` クレート[*9]を提供しています。「クレート」は Rust のコードをパッケージ化したものです。今まで作ってきたのは、実行可能な「バイナリクレート」です。`rand` は「ライブラリクレート」で、他のプログラムから使われることを意図したコードが入っています。

外部のクレートを使う時にこそ、Cargo が活きてきます。 `rand` を使う前に `Cargo.toml` を修正する必要があります。`Cargo.toml` を開いて、その末尾に以下の行を追加しましょう。

```
[dependencies]

rand="0.3.0"
```

`Cargo.toml` の `[dependencies]` (依存) セクションは `[package]` セクションに似ています。後続の行は、次のセクションが始まるまでそのセクションに属します。Cargo はどの外部クレートのどのバージョンに依存するのかの情報を取得するのに、dependencies セクションを使います。今回のケースではバージョン `0.3.0` を指定していますが、Cargo は指定されたバージョンと互換性のあるバージョンだと解釈します。Cargo はバージョン記述の標準、セマンティックバージョニング[*10]を理解します。上記のように、単にバージョンを書くのは、実は `^0.3.0` の略記になっており、「0.3.0 と互換性のあるもの」という意味になります。もし正確に `0.3.0` だけを使いたいなら `rand="=0.3.0"` (等号が 2 つあることに注意してください) と書きます。さらに最新版を使いたいなら `*` を使います。また、バージョンの範囲を使うこともできます。Cargo のドキュメント[*11]に、さらなる詳細があります。

さて、コードは変更せずにプロジェクトをビルドしてみましょう。

[*9] https://crates.io/crates/rand
[*10] http://semver.org
[*11] http://doc.crates.io/crates-io.html

```
$ cargo build
    Updating registry `https://github.com/rust-lang/crates.io-index`
 Downloading rand v0.3.8
 Downloading libc v0.1.6
   Compiling libc v0.1.6
   Compiling rand v0.3.8
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
```

(もちろん、別のバージョンが表示される可能性もあります)

いろいろと新しい出力がありました。外部依存ができたので、Cargo はそれぞれの最新版についての情報を、レジストリという、Crates.io[*12]からコピーしたデータから取得します。Crates.io は、Rust のエコシステムに参加している人たちが、オープンソースの Rust プロジェクトを投稿し、共有するための場所です。

レジストリをアップデートした後に、Cargo は `[dependencies]` を確認し、まだ手元にないものがあればダウンロードします。今回のケースでは `rand` に依存するとだけ書きましたが、`libc` も取得されています。これは `rand` が動作するのに `libc` に依存するためです。ダウンロードが終わったら、それらをコンパイルし、続いてプロジェクトをコンパイルします。

もう一度 `cargo build` を走らせると、異なった出力になります。

```
$ cargo build
```

そうです、何も出力されないのです。Cargo はプロジェクトがビルドされていて、依存もビルドされていることを知っているので、それらを繰り返さないのです。何もすることがなければそのまま終了します。もし `src/main.rs` を少し変更して保存したら、次のように表示されます。

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
```

Cargo には `rand` の `0.3.x` を使うと伝えたので、執筆時点の最新版 `v0.3.8` を取得しました。ですがもし来週 `v0.3.9` が出て、重要なバグがフィクスされたらどうなるのでしょう？ バグフィクスを取り込むのは重要ですが、 `0.3.9` にコードが動かなくなるようなリグレッションがあったらどうしましょう？

[*12] https://crates.io

この問題への答えは、プロジェクトのディレクトリにある `Cargo.lock` です。プロジェクトを最初にビルドした時に、Cargo は基準を満たす全てのバージョンを探索し、`Cargo.lock` ファイルに書き出します。その後のビルドでは、Cargo はまず `Cargo.lock` ファイルがあるか確認し、再度バージョンを探索することなく、そこで指定されたバージョンを使います。これで自動的に再現性のあるビルドが手に入ります。言い換えると、明示的にアップグレードしない限り、私たちは `0.3.8` を使い続けますし、ロックファイルのおかげで、コードを共有する人たちも `0.3.8` を使い続けます。

では `v0.3.9` を **使いたい** 時はどうすればいいのでしょうか？ Cargo には別のコマンド `update` があり、次のことを意味します：「ロックを無視して、指定したバージョンを満たす全ての最新版を探しなさい。それに成功したら、ロックファイルに書きなさい」しかし、デフォルトでは Cargo は `0.3.0` より大きく、 `0.4.0` より小さいバージョンを探しにいきます。 `0.4.x` より大きなバージョンを使いたいなら直接 `Cargo.toml` を更新する必要があります。 そうしたら、次に `cargo build` をする時に、Cargo はインデックスをアップデートして、`rand` の要件を再評価します。

Cargo[*13]とそのエコシステム[*14]については、説明することがまだ色々あるのですが、今のところは、これらのことだけを知っておけば十分です。Cargo のおかげでライブラリの再利用は本当に簡単になりますし、Rustacean は他のパッケージをいくつも使った小さなライブラリをよく書きます。

`rand` を実際に **使う**ところに進みましょう。次のステップはこれです。

```
extern crate rand;

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

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

    let secret_number = rand::thread_rng().gen_range(1, 101);

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

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

[*13] http://doc.crates.io
[*14] http://doc.crates.io/crates-io.html

```
    let mut guess = String::new();

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

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

訳注:

- The secret number is: {}: 秘密の数字は: {}です

1 つ目の変更は最初の行です。 `extern crate rand` としました。`rand` を `[dependencies]` に宣言したので、`extern crate` でそれを使うことを Rust に伝えています。これはまた、 `use rand;` と同じこともしますので、 `rand` にあるものは `rand::` と前置すれば使えるようになります。

次にもう 1 行 `use` を追加しました。 `use rand::Rng` です。この後すぐ、あるメソッドを使うのですが、それが動作するには `Rng` をスコープに入れる必要があるのです。基本的な考え方は次の通りです。このメソッドは「トレイト」と呼ばれるもので定義されており、動作させるために、該当するトレイトをスコープに入れる必要があるのです。詳しくはトレイトセクションを読んでください。

中ほどにもう 2 行足してあります。

```
    let secret_number = rand::thread_rng().gen_range(1, 101);

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

`rand::thread_rng()` を使って、いま現在の実行スレッドに対してローカルな、乱数生成器のコピーを取得しています。上で `use rand::Rng` したので、生成器は `gen_range()` メソッドを使えます。このメソッドは 2 つの引数を取り、その間の数を 1 つ生成します。下限は含みますが、上限は含まないので、1 から 100 までの数を生成するには `1` と `101` を渡す必要があります。

2 行目は秘密の数字を表示します。これは開発する時には有用で、簡単に動作確認できます。もちろん最終版では削除します。最初に答えを見せたら、ゲームじゃなくなってしまいます！

更新したプログラムを、何度か実行してみましょう。

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 7
Please input your guess.
4
You guessed: 4
$ cargo run
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 83
Please input your guess.
5
You guessed: 5
```

うまくいきました。次は予想値と秘密の数を比較します。

予想値と比較する

ユーザーの入力を受け取れるようになったので、秘密の数と比較しましょう。まだコンパイルできませんが、これが次のステップです。

```
extern crate rand;

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

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

    let secret_number = rand::thread_rng().gen_range(1, 101);
```

```
    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!"),
    }
}
```

訳注:

- Too small!: 小さすぎます！
- Too big!: 大きすぎます！
- You win!: あなたの勝ちです！

いくつか新しいことがあります。まず use が増えました。std::cmp::Ordering という型をスコープに導入しています。また、それを使うためのコードを末尾に 5 行追加しました。

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

cmp() は比較可能な全てのものに対して呼べるメソッドで、引数として、比較したい相手の参照を取ります。そして、先ほど use した、Ordering 型の値を返します。match 文を使って、正確に Ordering のどれであるかを判断しています。 Ordering は enum (列挙型) で、enum は「enumeration(列挙)」の略

です。 このようなものです。

```
enum Foo {
    Bar,
    Baz,
}
```

この定義だと、 Foo 型のものは Foo::Bar あるいは Foo::Baz のいずれかです。 :: を使って enum のバリアントの名前空間を指定します。

Ordering[*15] enum は 3 つのバリアントを持ちます。 Less 、Equal 、 Greater です。 match 文ではある型の値を取って、それぞれの可能な値に対する「腕」を作れます。Ordering には 3 種類あるので、3 つの腕を作っています。

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

Less なら Too small! を、 Greater なら Too big! を、 Equal なら You win! を表示します。 match はとても便利で、Rust でよく使われます。

これはコンパイルが通らないと言いました。試してみましょう。

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
src/main.rs:28:21: 28:35 error: mismatched types:
 expected `&collections::string::String`,
    found `&_`
(expected struct `collections::string::String`,
    found integral variable) [E0308]
src/main.rs:28     match guess.cmp(&secret_number) {
                                   ^~~~~~~~~~~~~~~
error: aborting due to previous error
```

[*15] http://doc.rust-lang.org/std/cmp/enum.Ordering.html

```
Could not compile `guessing_game`.
```

うわ、大きなエラーです。核心になっているのは「型の不一致」です。Rust には強い静的な型システムがあり、また、型推論もあります。`let guess = String::new()` と書いた時、Rust は `guess` が文字列であるはずだと推論できるので、わざわざ型を書かなくてもよいのです。また、`secret_number` では、1 から 100 までの数値を表せる型として、いくつかの候補があり、例えば、32bit 数の `i32` 、符号なし 32bit 数の `u32` 、64bit 数の `i64` などが該当します。これまで、そのどれであっても良かったため、Rust はデフォルトの `i32` としてました。 しかしここで、Rust は `guess` と `secret_number` の比較のしかたが分からないのです。これらは同じ型である必要があります。ということは、私たちたちは本当は、入力として読み取った `String` を、比較のために実数の型にしたかったわけです。それは 2 行追加すればできます。 新しいプログラムです。

```
extern crate rand;

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

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

    let secret_number = rand::thread_rng().gen_range(1, 101);

    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!"),
    }
}
```

新しい 2 行はこれです。

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

ちょっと待ってください、既に guess を定義してありますよね？ たしかにそうですが、Rust では以前の guess の定義を新しいもので「覆い隠す」ことができるのです (訳注: このように隠すことをシャドーイングといいます)。 まさにこのように、最初 String であった guess を u32 に変換したい、というような状況でよく使われます。シャドーイングのおかげで guess_str と guess のように別々の名前を考える必要はなくなり、 guess の名前を再利用できます。

guess を先に書いたような値に束縛します。

```
guess.trim().parse()
```

ここでの guess は、古い guess を指しており、入力を保持する String です。 String の trim() メソッドは、文字列の最初と最後にある空白を取り除きます。read_line() を満たすには「リターン」キーを押す必要があるので、これは重要です。つまり 5 と入力してリターンを押したら、 guess は 5\n のようになっています。\n は「改行」、つまり、エンターキーを表しています。 trim() することで、5 だけを残してこれを取り除けます。文字列の parse() メソッド[*16]は、文字列を何かの数値へとパースします。様々な数値をパースできるので、Rust に正確にどの型の数値が欲しいのかを伝える必要があります。なので let guess: u32 と書いたのです。 guess の後のコロン (:) は型注釈を付けようとしていることを Rust に伝えます。u32 は符号なし 32bit 整数です。 Rust には様々なビルトインの数値型 がありますが、今回は u32 を選びました。小さな正整数にはちょうどいいデフォルトとなる選択肢です。

[*16] http://doc.rust-lang.org/std/primitive.str.html#method.parse

`read_line()` と同じように、 `parse()` の呼び出しでもエラーが起こり得ます。 文字列に `A%` が含まれていたらどうなるでしょう？ それは数値には変換できません。ですから `read_line()` と同じように `expect()` を使って、エラーがあったらクラッシュするようにします。

プログラムを試してみましょう。

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running `target/guessing_game`
Guess the number!
The secret number is: 58
Please input your guess.
  76
You guessed: 76
Too big!
```

ばっちりです。予想値の前にスペースも入れてみましたが、それでも私が 76 と予想したんだと、ちゃんと理解してくれました。何度か動かしてみて、当たりが動くこと、小さい数字も動くことを確認してみてください。

ゲームが完成に近づいてきましたが、まだ、1 回しか予想できません。ループを使って書き換えましょう。

ループ

`loop` キーワードで無限ループが得られます。追加しましょう。

```
extern crate rand;

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

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

    let secret_number = rand::thread_rng().gen_range(1, 101);
```

```
    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);

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

試してみましょう。え？ でも待ってください、無限ループを追加しましたよね。そうです。でも `parse()` に関する議論を覚えてますか？ 数字でない答えを入力すると `panic!` して終了するのでした。見ててください。

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running `target/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!'
```

はいこの通り、たしかに `quit` で終了しました。他の数字でないものを入れても同じです。でもこれは、お世辞にも良いやり方とは言えませんね。まず、ゲームに勝ったら本当に終了するようにしましょう。

```
extern crate rand;

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

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

    let secret_number = rand::thread_rng().gen_range(1, 101);

    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);

        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 を加えることで、ゲームに勝った時にループを抜けます。ループを抜けることは同時に、それが main() の最後の要素なので、プログラムが終了することも意味します。もう 1 つ修正します。数値でない入力をした時に終了するのではなく、無視させましょう。それはこのようにできます。

```
extern crate rand;

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

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

    let secret_number = rand::thread_rng().gen_range(1, 101);
```

```
    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 = 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;
            }
        }
    }
}
```

変更はこれです。

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

このように、「エラーならクラッシュ」から「実際に戻り値のエラーをハンドルすること」へ移行する一般的な方法は、expect() を match 文に変更することです。 parse() は Result を返します。 これは Ordering と同じような enum ですが、今回はそれぞれのバリアントにデータが関連付いています。Ok は成功で、 Err は失敗です。それぞれには追加の情報もあります。パースに成功した整数、あるいはエラーの種類です。このケースでは Ok(num) に対して match していて、これは Ok をアンラップして得られた値 (整数値) を num という名前に設定します。続く右側では、その値をそのまま返しています。Err の場合、エラーの種類は気にしにしないので、名前ではなく、任意の値にマッチする_ を使いました。 こうすれば Ok 以外の全てをキャッチすることができ、continue によって、loop の次の繰り返しに進みます。こうして全てのエラーを無視し、プログラムの実行を続けることが可能になるのです。

これでいいはずです。試してみましょう。

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running `target/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！ 最後にほんの少し修正して、数当てゲームの制作を終えましょう。なんだか分かりますか？ そうです、秘密の数字は表示したくありません。テストには便利でしたが、ゲームを台無しにしてしまいます。これが最終的なソースコードです。

```
extern crate rand;

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

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

    let secret_number = rand::thread_rng().gen_range(1, 101);

    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;
            }
        }
```

```
    }
}
```

終わりに

数当てゲームが遂に完成しました！　お疲れ様でした！

このプロジェクトでは、様々なものをお見せしました。 `let` 、`match` 、メソッド、関連関数、外部クレートの使い方、などなど。次の章では、それぞれについて、さらに深く学んでいきましょう。

4

シンタックスとセマンティクス

この章では、Rust を構成する様々な概念について、それぞれ独立したセクションに分けて説明します。

Rust をボトムアップで学びたいなら、この章を順番に読んでいくのが近道です。

これらのセクションは、それぞれの概念に対するリファレンスにもなっていますので、他のチュートリアルを読んでいてよく分からないものが出てきたら、ここで説明されているはずです。

変数束縛

事実上全ての「Hello World」でない Rust のプログラムは **変数束縛**を使っています。変数束縛は何らかの値を名前へと束縛するので、後でその値を使えます。このように、 `let` が束縛を導入するのに使われています。

> 訳注: 普通、束縛というときは名前 を 値 へと束縛しますが、このドキュメントでは逆になっています。Rust では他の言語と違って 1 つの値に対して 1 つの名前が対応するのであえてこう書いてるのかもしれません。

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

例で毎回 `fn main() {` と書くのは長ったらしいのでこれ以後は省略します。もし試しながら読んでいるのならそのまま書くのではなくちゃんと `main()` 関数の中身を編集するようにしてください。そうしないとエラーになります。

パターン

多くの言語では変数束縛は **変数**と呼ばれるでしょうが、Rust の変数束縛は多少皮を被せてあります。例えば、`let` 文の左側は「パターン」であって、ただの変数名ではありません。これはこのようなことができるということです。

```
let (x, y) = (1, 2);
```

この文が評価されたあと、 `x` は 1 になり、 `y` は 2 になります。パターンは本当に強力で、本書にはパターンのセクションもあります。今のところこの機能は必要ないので頭の片隅に留めておいてだけいてください。

型アノテーション

Rust は静的な型付言語であり、前もって型を与えておいて、それがコンパイル時に検査されます。じゃあなぜ最初の例はコンパイルが通るのでしょう?ええと、Rust には「型推論」と呼ばれるものがあります。型推論が型が何であるか判断できるなら、型を書く必要はなくなります。

書きたいなら型を書くこともできます。型はコロン (`:`) のあとに書きます。

```
let x: i32 = 5;
```

これをクラスのみんなに聞こえるように声に出して読むなら、「 `x` は型 `i32` を持つ束縛で、値は 五 である。」となります。

この場合 `x` を 32bit 符号付き整数として表現することを選びました。Rust には多くのプリミティブな整数型があります。プリミティブな整数型は符号付き型は `i` 、符号無し型は `u` から始まります。整数型として可能なサイズは 8、16、32、64 ビットです。

以後の例では型はコメントで注釈することにします。先の例はこのようになります。

```
fn main() {
    let x = 5; // x: i32
}
```

この注釈と `let` の時に使う記法の類似性に留意してください。このようなコメントを書くのは Rust 的ではありませんが、時折理解の手助けのために Rust が推論する型をコメントで注釈します。

可変性

デフォルトで、 束縛は **イミュータブル**です。このコードのコンパイルは通りません。

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

次のようなエラーが出ます。

```
error: re-assignment of immutable variable `x`
     x = 10;
     ^~~~~~
```

訳注:

エラー：イミュータブルな変数 `x` に再代入しています

束縛をミュータブルにしたいなら、`mut` が使えます。

```
let mut x = 5; // mut x: i32
x = 10;
```

束縛がデフォルトでイミュータブルであるのは複合的な理由によるものですが、Rust の主要な焦点、安全性の一環だと考えることができます。もし `mut` を忘れたらコンパイラが捕捉して、変更するつもりでなかったものを変更した旨を教えてくれます。束縛がデフォルトでミュータブルだったらコンパイラはこれを捕捉できません。もし **本当に** 変更を意図していたのなら話は簡単です。 `mut` をつけ加えればいいのです。

可能な時にはミュータブルを避けた方が良い理由は他にもあるのですがそれはこのガイドの範囲を越え

ています。一般に、明示的な変更は避けられることが多いので Rust でもそうした方が良いのです。しかし変更が本当に必要なこともあるという意味で、厳禁という訳ではないのです。

束縛を初期化する

Rust の束縛はもう 1 つ他の言語と異る点があります。束縛を使う前に値で初期化されている必要があるのです。

試してみましょう。 `src/main.rs` をいじってこのようにしてみてください。

```
fn main() {
    let x: i32;

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

コマンドラインで `cargo build` を使ってビルドできます。警告が出ますが、それでもまだ「Hello, world!」は表示されます。

```
   Compiling hello_world v0.0.1 (file:///home/you/projects/hello_world)
src/main.rs:2:9: 2:10 warning: unused variable: `x`, #[warn(unused_variable)]
   on by default
src/main.rs:2     let x: i32;
                      ^
```

Rust は一度も使われない変数について警告を出しますが、一度も使われないので人畜無害です。ところがこの `x` を使おうとすると事は一変します。やってみましょう。プログラムをこのように変更してください。

```
fn main() {
    let x: i32;

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

そしてビルドしてみてください。このようなエラーが出るはずです。

```
$ cargo build
   Compiling hello_world v0.0.1 (file:///home/you/projects/hello_world)
src/main.rs:4:39: 4:40 error: use of possibly uninitialized variable: `x`
src/main.rs:4     println!("The value of x is: {}", x);
                                                   ^
note: in expansion of format_args!
<std macros>:2:23: 2:77 note: expansion site
<std macros>:1:1: 3:2 note: in expansion of println!
src/main.rs:4:5: 4:42 note: expansion site
error: aborting due to previous error
Could not compile `hello_world`.
```

Rust では未初期化の値を使うことは許されていません。 次に、`println!` に追加したものについて話しましょう。

表示する文字列に 2 つの波括弧 (`{}`、口髭という人もいます…(訳注: 海外の顔文字は横になっているので首を傾けて `{` を眺めてみてください。また、日本語だと「中括弧」と呼ぶ人もいますね)) を入れました。Rust はこれを、何かの値で補間 (interpolate) してほしいのだと解釈します。**文字列補間** (string interpolation) はコンピュータサイエンスの用語で、「文字列の間に差し込む」という意味です。その後に続けてカンマ、そして `x` を置いて、 `x` が補間に使う値だと指示しています。カンマは 2 つ以上の引数を関数やマクロに渡す時に使われます。

波括弧を使うと、Rust は補間に使う値の型を調べて意味のある方法で表示しようとします。フォーマットをさらに詳しく指定したいなら数多くのオプションが利用できます[*1]。とりあえずのところ、デフォルトに従いましょう。整数の表示はそれほど複雑ではありません。

スコープとシャドーイング

束縛に話を戻しましょう。変数束縛にはスコープがあります。変数束縛は定義されたブロック内でしか有効でありません。ブロックは `{` と `}` に囲まれた文の集まりです。関数定義もブロックです! 以下の例では異なるブロックで有効な 2 つの変数束縛、 `x` と `y` を定義しています。 `x` は `fn main() {}` ブロックの中でアクセス可能ですが、 `y` は内側のブロックからのみアクセスできます。

[*1] http://doc.rust-lang.org/std/fmt/index.html

```
fn main() {
    let x: i32 = 17;
    {
        let y: i32 = 3;
        println!("The value of x is {} and value of y is {}", x, y);
    }
   k
    println!("The value of x is {} and value of y is {}", x, y); // これは動きません
}
```

最初の `println!` は「The value of x is 17 and the value of y is 3」(訳注: 「x の値は 17 で y の値は 3」) と表示するはずですが、 2 つめの `println!` は `y` がもうスコープにいないため `y` にアクセスできないのでこの例はコンパイルできません。代わりに以下のようなエラーが出ます。

```
$ cargo build
   Compiling hello v0.1.0 (file:///home/you/projects/hello_world)
main.rs:7:62: 7:63 error: unresolved name `y`. Did you mean `x`? [E0425]
main.rs:7     println!("The value of x is {} and value of y is {}", x, y); // これは動きません
                                                                        ^
note: in expansion of format_args!
<std macros>:2:25: 2:56 note: expansion site
<std macros>:1:1: 2:62 note: in expansion of print!
<std macros>:3:1: 3:54 note: expansion site
<std macros>:1:1: 3:58 note: in expansion of println!
main.rs:7:5: 7:65 note: expansion site
main.rs:7:62: 7:63 help: run `rustc --explain E0425` to see a detailed explanation
error: aborting due to previous error
Could not compile `hello`.

To learn more, run the command again with --verbose.
```

さらに加えて、変数束縛は覆い隠すことができます (訳注: このことをシャドーイングと言います)。つまり後に出てくる同じ名前の変数束縛があるとそれがスコープに入り、以前の束縛を上書きするのです。

```
let x: i32 = 8;
{
    println!("{}", x); // "8" を表示する
    let x = 12;
    println!("{}", x); // "12" を表示する
}
println!("{}", x); // "8" を表示する
let x =  42;
println!("{}", x); // "42" を表示する
```

シャドーイングとミュータブルな束縛はコインの表と裏のように見えるかもしれませんが、それぞれ独立な概念であり互いに代用ができないケースがあります。その 1 つにシャドーイングは同じ名前に違う型の値を再束縛することができます。

```
let mut x: i32 = 1;
x = 7;
let x = x; // x はイミュータブルになって 7 に束縛されました

let y = 4;
let y = "I can also be bound to text!"; // y は違う型になりました
```

関数

Rust における全てのプログラムには、少なくとも 1 つの関数、 `main` 関数があります。

```
fn main() {
}
```

これは評価可能な関数定義の最も単純なものです。 前に言ったように、`fn` は「これは関数です」ということを示します。この関数には引数がないので、名前と丸括弧が続きます。そして、その本文を表す波括弧が続きます。これが `foo` という名前の関数です。

```
fn foo() {
}
```

それでは、引数を取る場合はどうでしょうか。これが数値を表示する関数です。

```
fn print_number(x: i32) {
    println!("x is: {}", x);
}
```

これが `print_number` を使う完全なプログラムです。

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

fn print_number(x: i32) {
    println!("x is: {}", x);
}
```

見てのとおり、関数の引数は `let` 宣言と非常によく似た動きをします。引数の名前にコロンに続けて型を追加します。

これが 2 つの数値を足して結果を表示する完全なプログラムです。

```
fn main() {
    print_sum(5, 6);
}

fn print_sum(x: i32, y: i32) {
    println!("sum is: {}", x + y);
}
```

関数を呼び出すときも、それを宣言したときと同様に、引数をコンマで区切ります。

`let` と異なり、あなたは関数の引数の型を宣言**しなければなりません** 。 これは動きません。

```
fn print_sum(x, y) {
    println!("sum is: {}", x + y);
}
```

このエラーが発生します。

```
expected one of `!`, `:`, or `@`, found `)`
fn print_sum(x, y) {
```

これはよく考えられた設計上の決断です。プログラムのすべての箇所で型推論をするという設計も可能ですが、一方で、そのように型推論を行なう Haskell のような言語でも、ドキュメント目的で型を明示するのはよい習慣だと言われています。私たちの意見は、関数の型を明示することは強制しつつ、関数本体では型を推論するようにすることが、すべての箇所で型推論をするのとまったく型推論をしないことの間のすばらしいスイートスポットである、というところで一致しています。

戻り値についてはどうでしょうか。 これが整数に 1 を加える関数です。

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

Rust の関数は値を 1 つだけ返します。そして、ダッシュ（ - ）の後ろに大なりの記号（ > ）を続けた「矢印」の後にその型を宣言します。関数の最後の行が何を返すのかを決定します。ここにセミコロンがないことに気が付くでしょう。もしそれを追加すると、こうなります。

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

エラーが発生するでしょう。

```
error: not all control paths return a value
fn add_one(x: i32) -> i32 {
     x + 1;
}
```

```
help: consider removing this semicolon:
     x + 1;
          ^
```

これは Rust について 2 つの興味深いことを明らかにします。それが式ベースの言語であること、そしてセミコロンが他の「波括弧とセミコロン」ベースの言語でのセミコロンとは違っているということです。これら 2 つのことは関連します。

式と文

Rust は主として式ベースの言語です。文には 2 種類しかなく、その他の全ては式です。

ではその違いは何でしょうか。 式は値を返しますが、文は返しません。それが「not all control paths return a value」で終わった理由です。文 `x + 1;` は値を返さないからです。Rust には 2 種類の文があります。「宣言文」と「式文」です。その他の全ては式です。 まずは宣言文について話しましょう。

いくつかの言語では、変数束縛を文としてだけではなく、式としても書けます。Ruby ではこうなります。

```
x = y = 5
```

しかし、Rust では束縛を導入するための `let` の使用は式では**ありません** 。 次の例はコンパイルエラーを起こします。

```
# // let x = (let y = 5); // expected identifier, found keyword `let`
let x = (let y = 5); // 識別子を期待していましたが、キーワード `let` が見つかりました
```

ここでコンパイラが言っているのは、式の先頭が来ることを期待していましたが、`let` は式ではなく、文の先頭にしかなれませんよ、ということです。

次のことに注意しましょう。 既に束縛されている変数への代入（例えば `y = 5` ）は、その値が特に役に立つものではありませんが、やはり式です。他の言語の中には、代入式を評価すると、代入された値（例えば、前の例では `5` ）が返されるものもあります。しかし、Rust の代入式はそれと異なり、評価すると空のタプル `()` が返されます。 なぜなら、代入された値には単一の所有者しかおらず、他のどんな値を返したとしても予想外の出来事になってしまうからです。

```
let mut y = 5;

let x = (y = 6);  // x は値 `()` を持っており、 `6` ではありません
```

Rust での 2 種類目の文は **式文** です。これの目的は式を文に変換することです。実際には Rust の文法は文の後には他の文が続くことが期待されています。これはそれぞれの式を区切るためにセミコロンを使うということを意味します。これは Rust が全ての行末にセミコロンを使うことを要求する他の言語のほとんどとよく似ていること、そして見られる Rust のコードのほとんど全ての行末で、セミコロンが見られるということを意味します。

「ほとんど」と言ったところの例外は何でしょうか。この例で既に見ています。

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

この関数は `i32` を返そうとしていますが、セミコロンを付ければ、それは代わりに `()` を返します。Rust はこの挙動がおそらく求めているものではないということを理解するので、前に見たエラーの中で、セミコロンを削除することを提案するのです。

早期リターン

しかし、早期リターンについてはどうでしょうか。Rust はそのためのキーワード `return` を持っています。

```
fn foo(x: i32) -> i32 {
    return x;

    // このコードは走りません!
    x + 1
}
```

`return` を関数の最後の行で使っても動きますが、それはよろしくないスタイルだと考えられています。

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

あなたがこれまで式ベースの言語を使ったことがなければ、 `return` のない前の定義の方がちょっと変に見えるかもしれません。しかし、それは時間とともに直観的に感じられるようになります。

発散する関数

Rust には「発散する関数」、すなわち値を返さない関数のための特別な構文がいくつかあります。

```
fn diverges() -> ! {
    panic!(”This function never returns!”);
}
```

panic! は既に見てきた println! と同様にマクロです。println! とは違って、 panic! は実行中の現在のスレッドを与えられたメッセージとともにクラッシュさせます。この関数はクラッシュを引き起こすので、決して値を返しません。そのため、この関数は「! 」型を持つのです。「 ! 」は「発散する（diverges）」と読みます。

もし diverges() を呼び出すメイン関数を追加してそれを実行するならば、次のようなものが出力されるでしょう。

```
thread ‘<main>’ panicked at ‘This function never returns!’, hello.rs:2
```

もしもっと情報を得たいと思うのであれば、RUST_BACKTRACE 環境変数をセットすることでバックトレースが得られます。

```
$ RUST_BACKTRACE=1 ./diverges
thread '<main>' panicked at 'This function never returns!', hello.rs:2
stack backtrace:
   1:     0x7f402773a829 - sys::backtrace::write::h0942de78b6c02817K8r
   2:     0x7f402773d7fc - panicking::on_panic::h3f23f9d0b5f4c91bu9w
   3:     0x7f402773960e - rt::unwind::begin_unwind_inner::h2844b8c5e81e79558Bw
   4:     0x7f4027738893 - rt::unwind::begin_unwind::h4375279447423903650
   5:     0x7f4027738809 - diverges::h2266b4c4b850236beaa
   6:     0x7f40277389e5 - main::h19bb1149c2f00ecfBaa
   7:     0x7f402773f514 - rt::unwind::try::try_fn::h13186883479104382231
   8:     0x7f402773d1d8 - __rust_try
   9:     0x7f402773f201 - rt::lang_start::ha172a3ce74bb453aK5w
  10:     0x7f4027738a19 - main
  11:     0x7f402694ab44 - __libc_start_main
```

```
  12:     0x7f40277386c8 - <unknown>
  13:                0x0 - <unknown>
```

もしすでに RUST_BACKTRACE 変数がセットされており、それをアンセットできないなどの理由により値を上書きするのなら、0 にセットすればバックトレースが取得されなくなります。それ以外の全ての値では（環境変数はあるが値がない場合も含め）バックトレースがオンになります。

```
$ export RUST_BACKTRACE=1
...
$ RUST_BACKTRACE=0 ./diverges
thread '<main>' panicked at 'This function never returns!', hello.rs:2
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

RUST_BACKTRACE は Cargo の run コマンドでも使えます。

```
$ RUST_BACKTRACE=1 cargo run
     Running `target/debug/diverges`
thread '<main>' panicked at 'This function never returns!', hello.rs:2
stack backtrace:
   1:     0x7f402773a829 - sys::backtrace::write::h0942de78b6c02817K8r
   2:     0x7f402773d7fc - panicking::on_panic::h3f23f9d0b5f4c91bu9w
   3:     0x7f402773960e - rt::unwind::begin_unwind_inner::h2844b8c5e81e79558Bw
   4:     0x7f4027738893 - rt::unwind::begin_unwind::h4375279447423903650
   5:     0x7f4027738809 - diverges::h2266b4c4b850236beaa
   6:     0x7f40277389e5 - main::h19bb1149c2f00ecfBaa
   7:     0x7f402773f514 - rt::unwind::try::try_fn::h13186883479104382231
   8:     0x7f402773d1d8 - __rust_try
   9:     0x7f402773f201 - rt::lang_start::ha172a3ce74bb453aK5w
  10:     0x7f4027738a19 - main
  11:     0x7f402694ab44 - __libc_start_main
  12:     0x7f40277386c8 - <unknown>
  13:                0x0 - <unknown>
```

発散する関数は任意の型としても使えます。

```
# fn diverges() -> ! {
```

```
#    panic!("This function never returns!");
# }
let x: i32 = diverges();
let x: String = diverges();
```

関数ポインタ

関数を指す（ポイントする）変数束縛も作れます。

```
let f: fn(i32) -> i32;
```

f という変数束縛は、 i32 を引数として受け取り、i32 を返す関数へのポインタになります。 例えばこうです。

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

// 型推論なし
let f: fn(i32) -> i32 = plus_one;

// 型推論あり
let f = plus_one;
```

そして f を使って関数を呼び出せます。

```
let six = f(5);
```

プリミティブ型

Rust 言語は「プリミティブ」とみなされる多数の型を持ちます。これはそれらが言語に組み込まれていることを意味します。Rust の型は構造化されており、標準ライブラリでもプリミティブ型を用いて構築した数多くの便利な型を提供しています。しかし、プリミティブ型が最もプリミティブ（基本的で素朴）な型です。

ブーリアン型

Rust には `bool` と名付けられた組込みのブーリアン型があります。それは `true` と `false` という 2 つの値を持ちます。

```
let x = true;

let y: bool = false;
```

ブーリアンの一般的な使い方は、 `if` 条件 で用いるものです。

`bool` の詳しいドキュメントは標準ライブラリのドキュメント[*2]にあります。

char

`char` 型は 1 つのユニコードのスカラ値を表現します。 `char` はシングルクオート（ ' ）で作られます。

```
let x = 'x';
let two_hearts = '';
```

`char` が 1 バイトである他の言語と異なり、これは Rust の `char` が 1 バイトではなく 4 バイトであるということを意味します。

`char` の詳しいドキュメントは標準ライブラリのドキュメント[*3]にあります。

数値型

Rust にはいくつかのカテゴリにたくさんの種類の数値型があります。そのカテゴリは符号ありと符号なし、固定長と可変長、浮動小数点数と整数です。

それらの型はカテゴリとサイズという 2 つの部分から成ります。 例えば、`u16` はサイズ 16 ビットで符号なしの型です。ビット数を大きくすれば、より大きな数値を扱うことができます。

[*2] http://doc.rust-lang.org/std/primitive.bool.html
[*3] http://doc.rust-lang.org/std/primitive.char.html

もし数値リテラルがその型を推論させるものを何も持たないのであれば、以下のとおりデフォルトになります。

```
let x = 42; // x は i32 型を持つ
let y = 1.0; // y は f64 型を持つ
```

これはいろいろな数値型のリストにそれらの標準ライブラリのドキュメントへのリンクを付けたものです。

- i8[*4]
- i16[*5]
- i32[*6]
- i64[*7]
- u8[*8]
- u16[*9]
- u32[*10]
- u64[*11]
- isize[*12]
- usize[*13]
- f32[*14]
- f64[*15]

それらをカテゴリ別に調べましょう。

[*4] http://doc.rust-lang.org/std/primitive.i8.html
[*5] http://doc.rust-lang.org/std/primitive.i16.html
[*6] http://doc.rust-lang.org/std/primitive.i32.html
[*7] http://doc.rust-lang.org/std/primitive.i64.html
[*8] http://doc.rust-lang.org/std/primitive.u8.html
[*9] http://doc.rust-lang.org/std/primitive.u16.html
[*10] http://doc.rust-lang.org/std/primitive.u32.html
[*11] http://doc.rust-lang.org/std/primitive.u64.html
[*12] http://doc.rust-lang.org/std/primitive.isize.html
[*13] http://doc.rust-lang.org/std/primitive.usize.html
[*14] http://doc.rust-lang.org/std/primitive.f32.html
[*15] http://doc.rust-lang.org/std/primitive.f64.html

符号ありと符号なし

整数型には符号ありと符号なしという 2 つの種類があります。違いを理解するために、サイズ 4 ビットの数値を考えましょう。符号あり 4 ビット整数は `-8` から `+7` までの数値を保存できます。 符号ありの数値は「2 の補数表現」を使います。符号なし 4 ビット整数は、マイナスを保存する必要がないため、 `0` から `+15` までの値を保存できます。

符号なし（訳注：unsigned）型はそれらのカテゴリに `u` を使い、符号あり型は `i` を使います。 `i` は「整数（訳注：integer）」の頭文字です。 そのため、 `u8` は 8 ビット符号なし数値、 `i8` は 8 ビット符号あり数値です。

固定長型

固定長型はそれらの表現に特定のビット数を持ちます。指定することのできるビット長は `8` 、 `16` 、 `32` 、 `64` です。 そのため、 `u32` は符号なし 32 ビット整数、`i64` は符号あり 64 ビット整数です。

可変長型

Rust が提供する型には、そのサイズが実行しているマシンのポインタのサイズに依存するものもあります。それらの型はカテゴリとして「size」を使い、符号ありと符号なしの種類があります。これが `isize` と `usize` という 2 つの型を作ります。

浮動小数点型

Rust は `f32` と `f64` という 2 つの浮動小数点型を持ちます。それらは IEEE-754 単精度及び倍精度小数点数に対応します。

配列

多くのプログラミング言語のように、Rust には何かのシーケンスを表現するためのリスト型があります。最も基本的なものは **配列** 、固定長の同じ型の要素のリストです。デフォルトでは、配列はイミュータブルです。

```
let a = [1, 2, 3]; // a: [i32; 3]
let mut m = [1, 2, 3]; // m: [i32; 3]
```

配列は `[T; N]` という型を持ちます。 この `T` 記法についてはジェネリクスのセクション で話します。`N` は配列の長さのためのコンパイル時の定数です。

配列の各要素を同じ値で初期化するための省略表現があります。 この例では、`a` の各要素は `0` で初期化されます。

```
let a = [0; 20]; // a: [i32; 20]
```

配列 `a` の要素の個数は `a.len()` で得られます。

```
let a = [1, 2, 3];

println!("a has {} elements", a.len());
```

配列の特定の要素には **添字記法** でアクセスできます。

```
let names = ["Graydon", "Brian", "Niko"]; // names: [&str; 3]

println!("The second name is: {}", names[1]);
```

添字はほとんどのプログラミング言語と同じように 0 から始まります。そのため、最初の名前は `names[0]` で 2 つ目の名前は `names[1]` です。前の例は `The second name is: Brian` と表示します。もし配列に含まれない添字を使おうとすると、エラーが出ます。配列アクセスは実行時に境界チェックを受けます。他のシステムプログラミング言語では、そのような誤ったアクセスは多くのバグの源となります。

`array` の詳しいドキュメントは標準ライブラリのドキュメント[*16]にあります。

スライス

「スライス」は他のデータ構造への参照（又は「ビュー」）です。それらはコピーすることなく配列の要素への安全で効率的なアクセスを許すために便利です。例えば、メモリに読み込んだファイルの 1 行だけを参照したいことがあるかもしれません。性質上、スライスは直接作られるのではなく、既存の変数束縛から作られます。スライスは定義された長さを持ち、ミュータブルにもイミュータブルにもできます。

スライスは内部的には、データの先頭へのポインタとデータ長の組み合わせで表現されます。

[*16] http://doc.rust-lang.org/std/primitive.array.html

スライシング構文

& と [] を組合せて使うと、様々なものからスライスが作れます。 & はスライスが、参照と似たものであることを示します（参照については、後ほど詳細をカバーします）。 [] はレンジを持ち、スライスの長さを定義します。

```
let a = [0, 1, 2, 3, 4];
let complete = &a[..]; // a に含まれる全ての要素を持つスライス
let middle = &a[1..4]; // 1、2、3 のみを要素に持つ a のスライス
```

スライスは型 &[T] を持ちます。ジェネリクス をカバーするときにその T について話すでしょう。

slice の詳しいドキュメントは標準ライブラリのドキュメント[*17]にあります。

str

Rust の str 型は最もプリミティブな文字列型です。サイズ不定型のように、それ単体ではあまり便利ではありませんが、&str のように参照の後ろに置かれたときに便利になります。文字列 と参照についてカバーする際に、より正確に学びましょう。

str の詳しいドキュメントは標準ライブラリのドキュメント[*18]にあります。

タプル

タプルは固定サイズの順序ありリストです。 このようなものです。

```
let x = (1, "hello");
```

丸括弧とコンマがこの長さ 2 のタプルを形成します。これは同じコードですが、型注釈が付いています。

```
let x: (i32, &str) = (1, "hello");
```

[*17] http://doc.rust-lang.org/std/primitive.slice.html
[*18] http://doc.rust-lang.org/std/primitive.str.html

見てのとおり、タプルの型はタプルと同じように見えます。しかし、各位置には値ではなく型名が付いています。注意深い読者は、タプルが異なる型の値を含んでいることにも気が付くでしょう。このタプルには i32 と &str が入っています。システムプログラミング言語では、文字列は他の言語よりも少し複雑です。今のところ、 &str を **文字列スライス**と読みましょう。それ以上のことは後で学ぶでしょう。

もしそれらの持っている型と アリティが同じであれば、あるタプルを他のタプルに割り当てられます。タプルの長さが同じであれば、それらのタプルのアリティは同じです。

```
let mut x = (1, 2); // x: (i32, i32)
let y = (2, 3); // y: (i32, i32)

x = y;
```

タプルのフィールドには **分配束縛** *let* を通じてアクセスできます。これが例です。

```
let (x, y, z) = (1, 2, 3);

println!("x is {}", x);
```

前に let 文の左辺は、単なる束縛の割り当てよりも強力だと言ったときのことを覚えていますか。ここで説明します。 let の左辺にはパターンを書くことができ、もしそれが右辺とマッチしたならば、複数の束縛を一度に割り当てられます。この場合、 let が「分配束縛」、つまりタプルを「分解して」、要素を 3 つの束縛に割り当てます。

このパターンは非常に強力で、後で繰り返し見るでしょう。

コンマを付けることで要素 1 のタプルを丸括弧の値と混同しないように明示できます。

```
(0,); // 1 要素のタプル
(0); // 丸括弧に囲まれたゼロ
```

タプルのインデックス

タプルのフィールドにはインデックス構文でアクセスすることもできます。

```
let tuple = (1, 2, 3);

let x = tuple.0;
let y = tuple.1;
let z = tuple.2;

println!("x is {}", x);
```

配列のインデックスと同じように、それは 0 から始まります。しかし、配列のインデックスと異なり、それは `[]` ではなく `.` を使います。

タプルの詳しいドキュメントは標準ライブラリのドキュメント[*19]にあります。

関数

関数も型を持ちます! それらはこのようになります。

```
fn foo(x: i32) -> i32 { x }

let x: fn(i32) -> i32 = foo;
```

この場合、 `x` は `i32` を受け取り `i32` を戻す関数への「関数ポインタ」です。

コメント

いくつかの関数ができたので、コメントについて学ぶことはよい考えです。コメントはコードについての何かを説明する助けになるように、他のプログラマに残すメモです。コンパイラはそれらをほとんど無視します。

Rust には気にすべき 2 種類のコメント、 **行コメント** と**ドキュメンテーションコメント** があります。

```
// 行コメントは「//」以降の全ての文字であり、行末まで続く
```

[*19] http://doc.rust-lang.org/std/primitive.tuple.html

```
let x = 5; // this is also a line comment.

// もし何かのために長い説明を書くのであれば、行コメントを複数行に渡って書くこと
// ができる。//とコメントとの間にスペースを置くことで、より読みやすくなる
```

その他の種類のコメントはドキュメンテーションコメントです。ドキュメンテーションコメントは `//` の代わりに `///` を使い、その中で Markdown 記法をサポートします。

```
/// 与えられた数値に 1 を加える
///
/// # Examples
///
/// ```
/// let five = 5;
///
/// assert_eq!(6, add_one(5));
/// # fn add_one(x: i32) -> i32 {
/// #     x + 1
/// # }
/// ```
fn add_one(x: i32) -> i32 {
    x + 1
}
```

もう 1 つのスタイルのドキュメンテーションコメントに `//!` があります。これは、その後に続く要素ではなく、それを含んでいる要素（例えばクレート、モジュール、関数）にコメントを付けます。一般的にはクレートルート（lib.rs）やモジュールルート（mod.rs）の中で使われます。

```
//! # Rust 標準ライブラリ
//!
//! Rust 標準ライブラリはポータブルな Rust ソフトウェアをビルドするために不可欠な
//! ランタイム関数を提供する。
```

ドキュメンテーションコメントを書いているとき、いくつかの使い方の例を提供することは非常に非常に有用です。ここでは新しいマクロ、 `assert_eq!` を使っていることに気付くでしょう。これは 2 つの

値を比較し、もしそれらが互いに等しくなければ `panic!` します。 これはドキュメントの中で非常に便利です。 もう 1 つのマクロ、`assert!` は、それに渡された値が `false` であれば `panic!` します。

それらのドキュメンテーションコメントから HTML ドキュメントを生成するため、そしてコード例をテストとして実行するためにも `rustdoc` ツールを使うことができます!

if

Rust における `if` の扱いはさほど複雑ではありませんが、伝統的なシステムプログラミング言語のそれと比べて、動的型付け言語の `if` により近いものになっています。そのニュアンスをしっかり理解できるように、説明しましょう。

`if` はより一般的なコンセプトの一つである、「分岐 (branch)」の具体的な形です。この名称は、木の枝 (branch) に由来します: 決定点はひとつの選択に依存し、複数のパスを取ることができます。

`if` の場合は、二つのパスを導く ひとつの選択があります。

```
let x = 5;

if x == 5 {
    println!("x は 5 です!");
}
```

もし、`x` を別の値に変更すると、この行は出力されません。よりわかりやすく説明すると、`if` のあとにくる式が `true` に評価された場合、そのブロックが実行されます。また、`false` の場合は、それは実行されません。

`false` の場合にも何かをしたい時は、 `else` を使います:

```
let x = 5;

if x == 5 {
    println!("x は 5 です!");
} else {
    println!("x は 5 ではありません :(");
}
```

複数の条件がある時は、 `else if` を使います:

```
let x = 5;

if x == 5 {
    println!("x は 5 です!");
} else if x == 6 {
    println!("x は 6 です!");
} else {
    println!("x は 5 でも 6 でもありません :(");
}
```

これは当然なことですが、次のように書くこともできます:

```
let x = 5;

let y = if x == 5 {
    10
} else {
    15
}; // y: i32
```

また、次のように書くのがほとんどの場合良いでしょう:

```
let x = 5;

let y = if x == 5 { 10 } else { 15 }; // y: i32
```

これが出来るのは `if` が式だからです。その式の値は、選択された条件の最後の式の値です。`else` のない `if` では、その値は常に `()` になります。

ループ

現在、Rust は、なんらかの繰り返しを伴う処理に対して、3 種類の手法: `loop`, `while`, `for` を提供しています。各アプローチにはそれぞれの使い方があります。

loop

Rust で使えるループのなかで最もシンプルな形式が、無限 `loop` です。Rust のキーワード `loop` によって、何らかの終了状態に到達するまで延々とループし続ける手段を提供します。Rust の無限 `loop` は次の通りです:

```
loop {
    println!("Loop forever!");
}
```

while

Rust には `while` ループもあります。次の通りです:

```
let mut x = 5; // mut x: i32
let mut done = false; // mut done: bool

while !done {
    x += x - 3;

    println!("{}", x);

    if x % 5 == 0 {
        done = true;
    }
}
```

何回ループする必要があるか明らかではない状況では、`while` ループは正しい選択です。

無限ループの必要があるとき、次のように書きたくなるかもしれません:

```
while true {
```

しかし、 `loop` は、 こういった場合に はるかに適しています。

```
loop {
```

Rust の制御フロー解析では、必ずループすると知っていることから、これを `while true` とは異なる構造として扱います。一般に、コンパイラへ与える情報量が多いほど、安全性が高くより良いコード生成につながるため、無限にループするつもりであれば、常に `loop` を使うべきです。

for

特定の回数だけループするときには `for` ループを使います。しかし、Rust の `for` ループは他のシステムプログラミング言語のそれとは少し異なる働きをします。Rust の `for` ループは、次のような「C スタイル」 `for` ループとは似ていません:

```
for (x = 0; x < 10; x++) {
    printf( "%d\n", x );
}
```

代わりに、このように書きます:

```
for x in 0..10 {
    println!("{}", x); // x: i32
}
```

もう少し抽象的な用語を使うと、

```
for var in expression {
    code
}
```

式 (expression) は [`IntoIterator`] を用いてイテレータへと変換することができるアイテムです。イテレータは一連の要素を返します。それぞれの要素がループの 1 回の反復になります。その値は、ループ本体に有効な名前, `var` に束縛されています。いったんループ本体を抜けると、次の値がイテレータから取り出され、次のループ処理を行います。それ以上の値が存在しない時は、`for` ループは終了します。

この例では、`0..10` が開始と終了位置をとる式であり、同範囲の値を返すイテレータを与えます。上限はその値自身を含まないため、このループは `0` から `9` までを表示します。 `10` ではありません。

Rust では意図的に「C スタイル」 `for` ループを持ちません。経験豊富な C 言語の開発者でさえ、ループの各要素を手動で制御することは複雑であり、また間違いを犯しやすいのです。

■列挙

ループの中で何回目の繰り返しかを把握する必要がある時、`.enumerate()` 関数が使えます。

レンジを対象に:

```
for (i,j) in (5..10).enumerate() {
    println!("i = {} and j = {}", i, j);
}
```

出力:

```
i = 0 and j = 5
i = 1 and j = 6
i = 2 and j = 7
i = 3 and j = 8
i = 4 and j = 9
```

レンジを括弧で囲うのを忘れないで下さい。

イテレータを対象に:

```
let lines = "hello\nworld".lines();

for (linenumber, line) in lines.enumerate() {
    println!("{}: {}", linenumber, line);
}
```

出力:

```
0: hello
1: world
```

反復の早期終了

さきほどの `while` ループを見てみましょう:

```
let mut x = 5;
let mut done = false;

while !done {
    x += x - 3;

    println!("{}", x);

    if x % 5 == 0 {
        done = true;
    }
}
```

ループを終了する時を知るために、、専用の `mut` である boolean 変数束縛, `done` を使わなければなりませんでした。Rust には反復の変更を手伝けする 2 つのキーワード: `break` と `continue` があります。

この例では、 `break` を使ってループを記述した方が良いでしょう:

```
let mut x = 5;

loop {
    x += x - 3;

    println!("{}", x);

    if x % 5 == 0 { break; }
}
```

ここでは `loop` による永久ループと 早期にループを抜けるため `break` を使っています。明示的な `return` 文の発行でもループを早期に終了します。

`continue` も似ていますが、ループを終了させるのではなく、次の反復へと進めます。これは奇数だけを

表示するでしょう:

```
for x in 0..10 {
    if x % 2 == 0 { continue; }

    println!("{}", x);
}
```

ループラベル

入れ子のループがあり、break や continue 文がどのループに対応するか指定する必要がある、そのような状況に出会うかもしれません。大抵の他言語と同様に、デフォルトで break や continue は最内ループに適用されます。 外側のループに break や continue を使いたいという状況では、 break や continue 文の適用先を指定するラベルを使えます。これは x と y 両方がともに奇数のときだけ表示を行います:

```
'outer: for x in 0..10 {
    'inner: for y in 0..10 {
        if x % 2 == 0 { continue 'outer; } // x のループを継続
        if y % 2 == 0 { continue 'inner; } // y のループを継続
        println!("x: {}, y: {}", x, y);
    }
}
```

ベクタ

「ベクタ」は動的な、または「拡張可能な」配列です、標準ライブラリ上で Vec<T>[*20] として提供されています。 T はどんなタイプのベクタをも作成することが可能なことを意味しています。(詳細はジェネリクスを御覧ください) ベクタはデータを常にヒープ上にアロケーションします。ベクタは以下のように vec! マクロを用いて作成できます:

[*20] http://doc.rust-lang.org/std/vec/index.html

```
let v = vec![1, 2, 3, 4, 5]; // v: Vec<i32>
```

(以前使った println! マクロと異なり、vec! マクロで角括弧 [] を利用しました。) Rust ではどちらの括弧もどちらのシチュエーションでも利用可能であり、解りやすさのためです。

vec! には初期値の繰り返しを表現するための形式があります:

```
let v = vec![0; 10]; // 0 が 10 個
```

要素へのアクセス

ベクタ中の特定のインデックスの値にアクセスするには [] を利用します:

```
let v = vec![1, 2, 3, 4, 5];

println!("The third element of v is {}", v[2]);
```

インデックスは 0 から始まります、なので三番目の要素は v[2] となります。

また、インデックスは usize 型でなければならない点に注意しましょう:

```
let v = vec![1, 2, 3, 4, 5];

let i: usize = 0;
let j: i32 = 0;

# // // works
// これは動作します
v[i];

# // // doesn't
// 一方、こちらは動作しません
v[j];
```

usize 型でないインデックスを用いた場合、以下の様なエラーが発生します:

```
error: the trait `core::ops::Index<i32>` is not implemented for the type
`collections::vec::Vec<_>` [E0277]
v[j];
^~~~
note: the type `collections::vec::Vec<_>` cannot be indexed by `i32`
error: aborting due to previous error
```

エラーメッセージ中には多くの点が含まれていますが、一番大切な部分は `i32` をインデックスとして用いることはできないという点です。

イテレーティング

ベクタである値に対して `for` を用いて以下の様な 3 つの方法でイテレートすることができます:

```
let mut v = vec![1, 2, 3, 4, 5];

for i in &v {
    println!("A reference to {}", i);
}

for i in &mut v {
    println!("A mutable reference to {}", i);
}

for i in v {
    println!("Take ownership of the vector and its element {}", i);
}
```

ベクタにはもっと多くの便利なメソッドが定義されています。それらのメソッドについては API ドキュメント[*21]で確認することができます。

[*21] http://doc.rust-lang.org/std/vec/index.html

所有権

このガイドは Rust の所有権システムの 3 つの解説の 1 つ目です。これは Rust の最も独特で注目されている機能です。そして、Rust 開発者はそれについて高度に精通しておくべきです。所有権こそは Rust がその最大の目標、メモリ安全性を得るための方法です。そこにはいくつかの別個の概念があり、各概念が独自の章を持ちます。

- 今読んでいる、所有権
- 借用、そしてそれらに関連する機能、「参照」
- 借用のもう一歩進んだ概念、ライフタイム

それらの 3 つの章は関連していて、それらは順番に並んでいます。所有権システムを完全に理解するためには、3 つ全てを必要とするでしょう。

概論

詳細に入る前に、所有権システムについての 2 つの重要な注意があります。

Rust は安全性とスピードに焦点を合わせます。Rust はそれらの目標を、様々な「ゼロコスト抽象化」を通じて成し遂げます。それは、Rust では抽象化を機能させるためのコストをできる限り小さくすることを意味します。所有権システムはゼロコスト抽象化の主な例です。このガイドの中で話すであろう解析の全ては **コンパイル時に行われます。** それらのどの機能に対しても実行時のコストは全く掛かりません。

しかし、このシステムはあるコストを持ちます。それは学習曲線です。多くの Rust 入門者は、私たちが「借用チェッカとの戦い」と呼ぶものを経験します。そこでは Rust コンパイラが、開発者が正しいと考えるプログラムをコンパイルすることを拒絶します。所有権がどのように機能するのかについてのプログラマのメンタルモデルが Rust の実装する実際のルールにマッチしないため、これはしばしば起きます。しかし、よいニュースがあります。より経験豊富な Rust の開発者は次のことを報告します。それは、所有権システムのルールと共にしばらく仕事をすれば、借用チェッカと戦うことは次第に少なくなっていく、というものです。

それを念頭に置いて、所有権について学びましょう。

所有権

Rust では [変数束縛][ownership.md–bindings] はある特性を持ちます。それは、束縛されているものの「所有権を持つ」ということです。これは束縛がスコープから外れるとき、Rust は束縛されているリソースを解放するだろうということを意味します。例えばこうです。

```
fn foo() {
    let v = vec![1, 2, 3];
}
```

v がスコープに入るとき、新しい [ベクタ][ownership.md–vectors] が [スタック][ownership.md–stack] 上に作られ、要素を格納するために [ヒープ][ownership.md–heap] に空間を割り当てます。`foo()` の最後で v がスコープから外れるとき、Rust はベクタに関連するもの全てを取り除くでしょう。それがヒープ割り当てのメモリであってもです。これはスコープの最後で決定的に起こります。

[ベクタ][ownership.md–vectors] については、前のセクションで説明済みですが、簡単に復習しましょう。ここではベクタを、実行時にヒープに空間を割り当てる型の例として用いています。ベクタは 配列のように振る舞いますが、追加の要素を `push()` するとサイズが変わるところは違います。

ベクタは [ジェネリクス型][ownership.md–generics] `Vec<T>` を持ちますので、この例における v は `Vec<i32>` 型になるでしょう。ジェネリクスについては、この章の後の方で詳しく説明します。

[ownership.md–vectors]: #sec–vectors [ownership.md–heap]: #ヒープ [ownership.md–stack]: #スタック [ownership.md–bindings]: #sec–variable-bindings [ownership.md–generics]: #sec–generics

ムーブセマンティクス

しかし、ここではもっと些細に見えることがあります。それは、Rust は与えられたリソースに対する束縛が *1 つだけ* あることを保証するというものです。例えば、もしベクタがあれば、それを別の束縛に割り当てることはできます。

```
let v = vec![1, 2, 3];

let v2 = v;
```

しかし、もし後で v を使おうとすると、エラーが出ます。

```
let v = vec![1, 2, 3];

let v2 = v;

println!("v[0] is: {}", v[0]);
```

こんな感じのエラーです。

```
error: use of moved value: `v`
println!("v[0] is: {}", v[0]);
                        ^
```

もし所有権を受け取る関数を定義して、引数として何かを渡した後でそれを使おうとするならば、同じようなことが起きます。

```
fn take(v: Vec<i32>) {
    // ここで何が起きるかは重要ではない
}

let v = vec![1, 2, 3];

take(v);

println!("v[0] is: {}", v[0]);
```

「use of moved value」という同じエラーです。所有権を何か別のものに転送するとき、参照するものを「ムーブした」と言います。これは特別な種類の注釈なしに行われます。つまり Rust のデフォルトの動作です。

詳細

束縛をムーブした後で、それを使うことができないと言いました。その理由は、ごく詳細かもしれませんが、とても重要です。

このようなコードを書いた時、

```
let x = 10;
```

Rust は スタック 上に整数 [i32] のためのメモリを割り当て、そこに、10 という値を表すビットパターンをコピーします。そして後から参照できるよう、変数名 x をこのメモリ領域に束縛します。

今度は、こんなコード片について考えてみましょう。

```
let v = vec![1, 2, 3];

let mut v2 = v;
```

最初の行では、先ほどの x と同様に、ベクタオブジェクト v のために、スタック上にメモリを割り当てます。しかし、これに加えて、実際のデータ（ [1, 2, 3] ）のために、ヒープ上にもメモリを割り当てます。スタック上のベクタオブジェクトの中にはポインタがあり、Rust はいま割り当てたヒープのアドレスをそこへコピーします。

すでに分かりきっているかもしれませんが、念のためここで確認しておきたいのは、ベクタオブジェクトとそのデータは、それぞれが別のメモリ領域に格納されていることです。決してそれらは、1 つの連続したメモリ領域に置かれているわけではありません（その理由についての詳細は、いまは省きます）。そして、ベクタにおけるこれら 2 つの部分（スタック上のものと、ヒープ上のもの）は、要素数やキャパシティ（容量）などについて、常にお互いの間で一貫性が保たれている必要があります。

v を v2 にムーブするとき Rust が実際に行うのは、ビット単位のコピーを使って、ベクタオブジェクト v が示すスタック領域の情報を、 v2 が示すスタック領域へコピーすることです。この浅いコピーでは、実際のデータを格納しているヒープ領域はコピーしません。これは、ベクタの内容として、同一のヒープメモリ領域を指すポインタが 2 つあることを意味します。もし誰かが v と v2 に同時にアクセスできるとしたら？ これはデータ競合を持ち込むことになり、Rust の安全性保証に違反するでしょう。

例えば v2 を通して、ベクタを 2 要素分、切り詰めたとしましょう。

```
v2.truncate(2);
```

もしまだ v1 にアクセスできたとしたら、v1 はヒープデータが切り詰められたことを知らないので、不正なベクタを提供することになってしまいます。ここでスタック上の v1 は、ヒープ上で対応する相手と一貫性が取れていません。 v1 はベクタにまだ 3 つの要素があると思っているので、もし私たちが存在しない要素 v1[2] にアクセスしようとしたら、喜んでそうさせるでしょう。しかし、すでにお気づきの通り、特に次のような理由から大惨事に繋がるかもしれません。これはセグメンテーション違反を起こ

すかもしれませんし、最悪の場合、権限を持たないユーザーが、本来アクセスできないはずのメモリを読めてしまうかもしれないのです。

このような理由から、Rust はムーブを終えた後の v の使用を禁止するのです。

また知っておいてほしいのは、状況によっては最適化により、スタック上のバイトを実際にコピーする処理が省かれる可能性があることです。そのため、ムーブは最初に思ったほど非効率ではないかもしれません。

Copy 型

所有権が他の束縛に転送されるとき、元の束縛を使うことができないということを証明しました。しかし、この挙動を変更する トレイトがあります。それは Copy と呼ばれます。トレイトについてはまだ議論していませんが、とりあえずそれらを、ある型に対してある挙動を追加するための、注釈のようなものとして考えて構いません。例えばこうです。

```
let v = 1;

let v2 = v;

println!("v is: {}", v);
```

この場合、 v は i32 で、それは Copy トレイトを実装します。 これはちょうどムーブと同じように、 v を v2 に代入するとき、データのコピーが作られることを意味します。しかし、ムーブと違って、後でまだ v を使うことができます。これは i32 がどこか別の場所へのポインタを持たず、コピーが完全コピーだからです。

全てのプリミティブ型は Copy トレイトを実装しているので、推測どおりそれらの所有権は「所有権ルール」に従ってはムーブしません。例として、次の 2 つのコードスニペットはコンパイルが通ります。なぜなら、i32 型と bool 型は Copy トレイトを実装するからです。

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

    let _y = double(a);
    println!("{}", a);
}
```

```
fn double(x: i32) -> i32 {
    x * 2
}
```

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

    let _y = change_truth(a);
    println!("{}", a);
}

fn change_truth(x: bool) -> bool {
    !x
}
```

もし Copy トレイトを実装していない型を使っていたならば、ムーブした値を使おうとしたため、コンパイルエラーが出ていたでしょう。

```
error: use of moved value: `a`
println!("{}", a);
               ^
```

独自の Copy 型を作る方法はトレイト セクションで議論するでしょう。

所有権を越えて

もちろん、もし書いた全ての関数で所有権を返さなければならないのであれば、こうなります。

```
fn foo(v: Vec<i32>) -> Vec<i32> {
    // v についての作業を行う

    // 所有権を返す
    v
}
```

これは非常に退屈になるでしょう。もっとたくさんのものの所有権を受け取ろうとすると、状況はさらに悪化します。

```
fn foo(v1: Vec<i32>, v2: Vec<i32>) -> (Vec<i32>, Vec<i32>, i32) {
    // v1 と v2 についての作業を行う

    // 所有権と関数の結果を返す
    (v1, v2, 42)
}

let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];

let (v1, v2, answer) = foo(v1, v2);
```

うわあ！　戻り値の型、リターン行、関数呼出しがもっと複雑になります。

幸運なことに、Rust は借用という機能を提供します。それはこの問題を解決するために手助けしてくれます。それが次のセクションの話題です。

参照と借用

このガイドは Rust の所有権システムの 3 つの解説の 2 つ目です。これは Rust の最も独特で注目されている機能です。そして、Rust 開発者はそれについて高度に精通しておくべきです。所有権こそは Rust がその最大の目標、メモリ安全性を得るための方法です。そこにはいくつかの別個の概念があり、各概念が独自の章を持ちます。

- キーとなる概念、 所有権
- 今読んでいる、借用
- 借用のもう一歩進んだ概念、ライフタイム

それらの 3 つの章は関連していて、それらは順番に並んでいます。所有権システムを完全に理解するためには、3 つ全てを必要とするでしょう。

概論

詳細に入る前に、所有権システムについての 2 つの重要な注意があります。

Rust は安全性とスピードに焦点を合わせます。Rust はそれらの目標を、様々な「ゼロコスト抽象化」を通じて成し遂げます。それは、Rust では抽象化を機能させるためのコストをできる限り小さくすることを意味します。所有権システムはゼロコスト抽象化の主な例です。このガイドの中で話すであろう解析の全ては **コンパイル時に行われます。** それらのどの機能に対しても実行時のコストは全く掛かりません。

しかし、このシステムはあるコストを持ちます。それは学習曲線です。多くの Rust 入門者は、私たちが「借用チェッカとの戦い」と呼ぶものを経験します。そこでは Rust コンパイラが、開発者が正しいと考えるプログラムをコンパイルすることを拒絶します。所有権がどのように機能するのかについてのプログラマのメンタルモデルが Rust の実装する実際のルールにマッチしないため、これはしばしば起きます。しかし、よいニュースがあります。より経験豊富な Rust の開発者は次のことを報告します。それは、所有権システムのルールと共にしばらく仕事をすれば、借用チェッカと戦うことは次第に少なくなっていく、というものです。

それを念頭に置いて、借用について学びましょう。

借用

所有権セクションの最後に、このような感じの厄介な関数に出会いました。

```
fn foo(v1: Vec<i32>, v2: Vec<i32>) -> (Vec<i32>, Vec<i32>, i32) {
    // v1 と v2 についての作業を行う

    // 所有権と関数の結果を返す
    (v1, v2, 42)
}

let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];

let (v1, v2, answer) = foo(v1, v2);
```

しかし、これは Rust 的なコードではありません。なぜなら、それは借用の利点を生かしていないからです。これが最初のステップです。

```
fn foo(v1: &Vec<i32>, v2: &Vec<i32>) -> i32 {
    // v1 と v2 についての作業を行う

    // 答えを返す
    42
}

let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];

let answer = foo(&v1, &v2);

// ここでは v1 と v2 が使える!
```

引数として `Vec<i32>` を使う代わりに、参照、つまり `&Vec<i32>` を使います。 そして、 `v1` と `v2` を直接渡す代わりに、`&v1` と `&v2` を渡します。 `&T` 型は「参照」と呼ばれ、それは、リソースを所有するのではなく、所有権を借用します。何かを借用した束縛はそれがスコープから外れるときにリソースを割当解除しません。これは `foo()` の呼出しの後に元の束縛を再び使うことができることを意味します。

参照は束縛と同じようにイミュータブルです。 これは `foo()` の中ではベクタは全く変更できないことを意味します。

```
fn foo(v: &Vec<i32>) {
     v.push(5);
}

let v = vec![];

foo(&v);
```

次のようなエラーが出ます。

```
error: cannot borrow immutable borrowed content `*v` as mutable
```

```
v.push(5);
^
```

値の挿入はベクタを変更するものであり、そうすることは許されていません。

&mut 参照

参照には 2 つ目の種類、 `&mut T` があります。「ミュータブルな参照」によって借用しているリソースを変更できるようになります。例は次のとおりです。

```
let mut x = 5;
{
    let y = &mut x;
    *y += 1;
}
println!("{}", x);
```

これは `6` を表示するでしょう。 `y` を `x` へのミュータブルな参照にして、それから `y` の指示先に 1 を足します。 `x` も `mut` とマークしなければならないことに気付くでしょう。そうしないと、イミュータブルな値へのミュータブルな借用ということになってしまい、使うことができなくなってしまいます。

アスタリスク（ `*` ）を `y` の前に追加して、それを`*y` にしたことにも気付くでしょう。これは、 `y` が`&mut` 参照だからです。参照の内容にアクセスするためにもそれらを使う必要があるでしょう。

それ以外は、 `&mut` 参照は普通の参照と同じです。しかし、2 つの間には、そしてそれらがどのように相互作用するかには大きな違いがあります 。前の例で何かが怪しいと思ったかもしれません。なぜなら、`{` と`}` を使って追加のスコープを必要とするからです。もしそれらを削除すれば、次のようなエラーが出ます。

```
error: cannot borrow `x` as immutable because it is also borrowed as mutable
    println!("{}", x);
                   ^
note: previous borrow of `x` occurs here; the mutable borrow prevents
subsequent moves, borrows, or modification of `x` until the borrow ends
        let y = &mut x;
                     ^
```

```
note: previous borrow ends here
fn main() {

}
^
```

結論から言うと、ルールがあります。

ルール

これが Rust での借用についてのルールです。

最初に、借用は全て所有者のスコープより長く存続してはなりません。次に、次の 2 種類の借用のどちらか 1 つを持つことはありますが、両方を同時に持つことはありません。

- リソースに対する 1 つ以上の参照（ `&T` ）
- ただ 1 つのミュータブルな参照（ `&mut T` ）

これがデータ競合の定義と非常に似ていることに気付くかもしれません。全く同じではありませんが。

> 「データ競合」は 2 つ以上のポインタがメモリの同じ場所に同時にアクセスするとき、少なくともそれらの 1 つが書込みを行っていて、作業が同期されていないところで「データ競合」は起きます。

書込みを行わないのであれば、参照は好きな数だけ使うことができます。`&mut` は同時に 1 つしか持つことができないので、データ競合は起き得ません。これが Rust がデータ競合をコンパイル時に回避する方法です。もしルールを破れば、そのときはエラーが出るでしょう。

これを念頭に置いて、もう一度例を考えましょう。

スコープの考え方

このコードについて考えていきます。

```
let mut x = 5;
let y = &mut x;
```

```
*y += 1;

println!("{}", x);
```

このコードは次のようなエラーを出します。

```
error: cannot borrow `x` as immutable because it is also borrowed as mutable
    println!("{}", x);
                   ^
```

なぜなら、これはルールに違反しているからです。つまり、 x を指示する &mut T を持つので、 &T を作ることは許されないのです。 どちらか 1 つです。 note の部分はこの問題についての考え方のヒントを示します。

```
note: previous borrow ends here
fn main() {

}
^
```

言い換えると、ミュータブルな借用は、先ほどの例の残りの間、ずっと保持されるということです。ここで私たちが求めているのは、y によるミュータブルな借用が終わり、リソースがその所有者である x に返却されることです。 そうすれば x は println! にイミュータブルな借用を提供できるわけです。Rust では借用はその有効なスコープと結び付けられます。そしてスコープはこのように見えます。

```
let mut x = 5;

let y = &mut x;    // -+ x の&mut 借用がここから始まる
                   //  |
*y += 1;           //  |
                   //  |
println!("{}", x); // -+ - ここで x を借用しようとする
                   // -+ x の&mut 借用がここで終わる
```

スコープは衝突します。 y がスコープにある間は、 &x を作ることができません。

そして、波括弧を追加するときはこうなります。

```
let mut x = 5;

{
    let y = &mut x; // -+ &mut 借用がここから始まる
    *y += 1;        //  |
}                   // -+ ... そしてここで終わる

println!("{}", x);  // <- ここで x を借用しようとする
```

これなら問題ありません。ミュータブルな借用はイミュータブルな借用を作る前にスコープから外れます。しかしスコープは、借用がどれくらい存続するのか理解するための鍵となります。

借用が回避する問題

なぜこのような厳格なルールがあるのでしょうか。そう、前述したように、それらのルールはデータ競合を回避します。データ競合はどのような種類の問題を起こすのでしょうか。ここに一部を示します。

■イテレータの無効

一例は「イテレータの無効」です。それは繰返しを行っているコレクションを変更しようとするときに起こります。Rust の借用チェッカはこれの発生を回避します。

```
let mut v = vec![1, 2, 3];

for i in &v {
    println!("{}", i);
}
```

これは 1 から 3 までを表示します。ベクタに対して繰り返すとき、要素への参照だけを受け取ります。そして、v はそれ自体イミュータブルとして借用され、それは繰返しを行っている間はそれを変更できないことを意味します。

```
let mut v = vec![1, 2, 3];

for i in &v {
    println!("{}", i);
    v.push(34);
}
```

これがエラーです。

```
error: cannot borrow `v` as mutable because it is also borrowed as immutable
    v.push(34);
    ^
note: previous borrow of `v` occurs here; the immutable borrow prevents
subsequent moves or mutable borrows of `v` until the borrow ends
for i in &v {
          ^
note: previous borrow ends here
for i in &v {
    println!( “{}” , i);
    v.push(34);
}
^
```

v はループによって借用されるので、それを変更することはできません。

■解放後の使用

参照はそれらの指示するリソースよりも長く生存することはできません。Rust はこれが真であることを保証するために、参照のスコープをチェックするでしょう。

もし Rust がこの性質をチェックしなければ、無効な参照をうっかり使ってしまうかもしれません。例えばこうです。

```
let y: &i32;
{
    let x = 5;
```

```
    y = &x;
}

println!("{}", y);
```

次のようなエラーが出ます。

```
error: `x` does not live long enough
    y = &x;
         ^
note: reference must be valid for the block suffix following statement 0 at
2:16...
let y: &i32;
{
    let x = 5;
    y = &x;
}

note: ...but borrowed value is only valid for the block suffix following
statement 0 at 4:18
    let x = 5;
    y = &x;
}
```

言い換えると、 `y` は `x` が存在するスコープの中でだけ有効だということです。 `x` がなくなるとすぐに、それを指示することは不正になります。そのように、エラーは借用が「十分長く生存していない」ことを示します。なぜなら、それが正しい期間有効ではないからです。

参照がそれの参照する変数より **前に**宣言されたとき、同じ問題が起こります。これは同じスコープにあるリソースはそれらの宣言された順番と逆に解放されるからです。

```
let y: &i32;
let x = 5;
y = &x;
```

```
println!("{}", y);
```

次のようなエラーが出ます。

```
error: `x` does not live long enough
y = &x;
     ^
note: reference must be valid for the block suffix following statement 0 at
2:16...
    let y: &i32;
    let x = 5;
    y = &x;

    println!("{}", y);
}

note: ...but borrowed value is only valid for the block suffix following
statement 1 at 3:14
    let x = 5;
    y = &x;

    println!("{}", y);
}
```

前の例では、 y は x より前に宣言されています。それは、y が x より長く生存することを意味し、それは許されません。

ライフタイム

このガイドは Rust の所有権システムの 3 つの解説の 3 つ目です。これは Rust の最も独特で注目されている機能です。そして、Rust 開発者はそれについて高度に精通しておくべきです。所有権こそは Rust がその最大の目標、メモリ安全性を得るための方法です。そこにはいくつかの別個の概念があり、各概念が独自の章を持ちます。

- キーとなる概念、 所有権
- 借用、そしてそれらに関連する機能、「参照」
- 今読んでいる、ライフタイム

それらの 3 つの章は関連していて、それらは順番に並んでいます。所有権システムを完全に理解するためには、3 つ全てを必要とするでしょう。

概論

詳細に入る前に、所有権システムについての 2 つの重要な注意があります。

Rust は安全性とスピードに焦点を合わせます。Rust はそれらの目標を、様々な「ゼロコスト抽象化」を通じて成し遂げます。それは、Rust では抽象化を機能させるためのコストをできる限り小さくすることを意味します。所有権システムはゼロコスト抽象化の主な例です。このガイドの中で話すであろう解析の全ては **コンパイル時に行われます。** それらのどの機能に対しても実行時のコストは全く掛かりません。

しかし、このシステムはあるコストを持ちます。それは学習曲線です。多くの Rust 入門者は、私たちが「借用チェッカとの戦い」と呼ぶものを経験します。そこでは Rust コンパイラが、開発者が正しいと考えるプログラムをコンパイルすることを拒絶します。所有権がどのように機能するのかについてのプログラマのメンタルモデルが Rust の実装する実際のルールにマッチしないため、これはしばしば起きます。しかし、よいニュースがあります。より経験豊富な Rust の開発者は次のことを報告します。それは、所有権システムのルールと共にしばらく仕事をすれば、借用チェッカと戦うことは次第に少なくなっていく、というものです。

それを念頭に置いて、ライフタイムについて学びましょう。

ライフタイム

他の誰かの所有するリソースへの参照の貸付けは複雑になることがあります。例えば、次のような一連の作業を想像しましょう。

1. 私はある種のリソースへのハンドルを取得する
2. 私はあなたにリソースへの参照を貸し付ける
3. 私はリソースを使い終わり、それを解放することを決めるが、あなたはそれに対する参照をまだ持っている
4. あなたはリソースを使うことを決める

あー！　あなたの参照は無効なリソースを指しています。リソースがメモリであるとき、これはダングリングポインタまたは「解放後の使用」と呼ばれます。

これを修正するためには、ステップ 3 の後にステップ 4 が絶対に起こらないようにしなければなりません。Rust での所有権システムはこれをライフタイムと呼ばれる概念を通じて行います。それは参照の有効なスコープを記述するものです。

引数として参照を受け取る関数について、参照のライフタイムを黙示または明示できます。

```
// 黙示的に
fn foo(x: &i32) {
}

// 明示的に
fn bar<'a>(x: &'a i32) {
}
```

'a は「ライフタイム a」と読みます。技術的には参照は全てそれに関連するライフタイムを持ちますが、一般的な場合にはコンパイラがそれらを省略してもよいように計らってくれます（つまり、「省略」できるということです。「ライフタイムの省略」 以下を見ましょう）。しかし、それに入る前に、明示の例を分解しましょう。

```
fn bar<'a>(...)
```

関数の構文については前に少し話しました。しかし、関数名の後の<> については議論しませんでした。関数は<> の間に「ジェネリックパラメータ」を持つことができ、ライフタイムはその一種です。他の種類のジェネリクスについては本書の後の方で議論しますが、とりあえず、ライフタイムの面に焦点を合わせましょう。

<> はライフタイムを宣言するために使われます。 これは bar が 1 つのライフタイム 'a を持つことを意味します。もし 2 つの参照引数があれば、それは次のような感じになるでしょう。

```
fn bar<'a, 'b>(...)
```

そして引数リストでは、名付けたライフタイムを使います。

```
...(x: &'a i32)
```

もし &mut 参照が欲しいのならば、次のようにします。

```
...(x: &'a mut i32)
```

もし &mut i32 を &'a mut i32 と比較するならば、それらは同じです。それはライフタイム 'a が& と mut i32 の間にこっそり入っているだけです。&mut i32 は「 i32 へのミュータブルな参照」のように読み、 &'a mut i32 は「ライフタイム 'a を持つ i32 へのミュータブルな参照」のように読みます。

struct の中

参照を含む struct を使うときにも、明示的なライフタイムを必要とするでしょう。

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let y = &5; // これは `let _y = 5; let y = &_y;`と同じ
    let f = Foo { x: y };

    println!("{}", f.x);
}
```

見てのとおり、 struct もライフタイムを持てます。これは関数と同じ方法です。

```
struct Foo<'a> {
```

このようにライフタイムを宣言します。

```
x: &'a i32,
```

そしてそれを使います。それではなぜここでライフタイムを必要とするのでしょうか。 Foo への全ての参照がそれの含む i32 への参照より長い間有効にはならないことを保証する必要があるからです。

impl ブロック

Foo に次のようなメソッドを実装しましょう。

```
struct Foo<'a> {
    x: &'a i32,
}

impl<'a> Foo<'a> {
    fn x(&self) -> &'a i32 { self.x }
}

fn main() {
    let y = &5; // これは `let _y = 5; let y = &_y;`と同じ
    let f = Foo { x: y };

    println!("x is: {}", f.x());
}
```

見てのとおり、 Foo のライフタイムは impl 行で宣言する必要があります。 関数のときのように 'a は 2 回繰り返されます。つまり、 impl<'a> はライフタイム 'a を定義し、Foo<'a> はそれを使うのです。

複数のライフタイム

もし複数の参照があるなら、同じライフタイムを何度でも使えます。

```
fn x_or_y<'a>(x: &'a str, y: &'a str) -> &'a str {
```

これは x と y が両方とも同じスコープで有効であり、戻り値もそのスコープで有効であることを示します。もし x と y に違うライフタイムを持たせたいのであれば、複数のライフタイムパラメータを使えます。

```
fn x_or_y<'a, 'b>(x: &'a str, y: &'b str) -> &'a str {
```

この例では x と y が異なる有効なスコープを持ちますが、戻り値は x と同じライフタイムを持ちます。

スコープの考え方

ライフタイムについて考えるには、参照の有効なスコープを可視化することです。例えばこうです。

```
fn main() {
    let y = &5;     // -+ y がスコープに入る
                    //  |
    // stuff        //  |
                    //  |
}                   // -+ y がスコープから出る
```

Foo を追加するとこうなります。

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let y = &5;           // -+ y がスコープに入る
    let f = Foo { x: y }; // -+ f がスコープに入る
    // stuff              //  |
                          //  |
}                         // -+ f と y がスコープから出る
```

f は y のスコープの中で有効なので、全て動きます。もしそれがそうではなかったらどうでしょうか。このコードは動かないでしょう。

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let x;                    // -+ x がスコープに入る
                              //  |
```

```
    {                           //  |
        let y = &5;             // ---+ y がスコープに入る
        let f = Foo { x: y }; // ---+ f がスコープに入る
        x = &f.x;               //  | | ここでエラーが起きる
    }                           // ---+ f と y がスコープから出る
                                //  |
    println!("{}", x);          //  |
}                               // -+ x がスコープから出る
```

ふう！ 見てのとおり、ここでは f と y のスコープは x のスコープよりも小さいです。 しかし x = &f.x を実行するとき、 x をまさにスコープから外れた何かの参照にしてしまいます。

名前の付いたライフタイムはそれらのスコープに名前を与える方法です。何かに名前を与えることはそれについて話をできるようになるための最初のステップです。

'static

「static」と名付けられたライフタイムは特別なライフタイムです。それは何かがプログラム全体に渡るライフタイムを持つことを示します。ほとんどの Rust のプログラマが最初に 'static に出会うのは、文字列を扱うときです。

```
let x: &'static str = "Hello, world.";
```

文字列リテラルは &'static str 型を持ちます。なぜなら、参照が常に有効だからです。それらは最終的なバイナリのデータセグメントに焼き付けられます。もう 1 つの例はグローバルです。

```
static FOO: i32 = 5;
let x: &'static i32 = &FOO;
```

これはバイナリのデータセグメントに i32 を追加します。そして、x はそれへの参照です。

ライフタイムの省略

Rust は関数本体については強力なローカル型推論をサポートしますが、要素のシグネチャについては別です。そこで型推論が許されていないのは、要素のシグネチャだけで型がわかるようにするためです。とはいえ、エルゴノミック（人間にとっての扱いやすさ）上の理由により、ライフタイムを決定する際

には、「ライフタイムの省略」と呼ばれる、非常に制限された第二の推論アルゴリズムが適用されます。ライフタイムの推論は、ライフタイムパラメータの推論だけに関係しており、たった 3 つの覚えやすく明確なルールに従います。ライフタイムの省略は要素のシグネチャを短く書けることを意味しますが、ローカル型推論が適用されるときのように実際の型を隠すことはできません。

ライフタイムの省略について話すときには、 **入力ライフタイム** と**出力ライフタイム** という用語を使います。 **入力ライフタイム**は関数の引数に関連するライフタイムで、 **出力ライフタイム**は関数の戻り値に関連するライフタイムです。例えば、次の関数は入力ライフタイムを持ちます。

```
fn foo<'a>(bar: &'a str)
```

この関数は出力ライフタイムを持ちます。

```
fn foo<'a>() -> &'a str
```

この関数は両方の位置のライフタイムを持ちます。

```
fn foo<'a>(bar: &'a str) -> &'a str
```

3 つのルールを以下に示します。

- 関数の引数の中の省略された各ライフタイムは、互いに異なるライフタイムパラメータになる
- もし入力ライフタイムが 1 つだけならば、省略されたかどうかにかかわらず、そのライフタイムはその関数の戻り値の中の省略されたライフタイム全てに割り当てられる
- もし入力ライフタイムが複数あるが、その 1 つが `&self` または`&mut self` であれば、 `self` のライフタイムは省略された出力ライフタイム全てに割り当てられる

そうでないときは、出力ライフタイムの省略はエラーです。

例

ここにライフタイムの省略された関数の例を示します。省略されたライフタイムの各例をその展開した形式と組み合わせています。

```rust
fn print(s: &str); // 省略された形
fn print<'a>(s: &'a str); // 展開した形

fn debug(lvl: u32, s: &str); // 省略された形
fn debug<'a>(lvl: u32, s: &'a str); // 展開された形

// 前述の例では `lvl` はライフタイムを必要としません。なぜなら、それは参照（`&`）
// ではないからです。（参照を含む `struct` のような）参照に関係するものだけがライ
// フタイムを必要とします。

fn substr(s: &str, until: u32) -> &str; // 省略された形
fn substr<'a>(s: &'a str, until: u32) -> &'a str; // 展開された形

fn get_str() -> &str; // 不正。入力がない

   ambiguous
fn frob(s: &str, t: &str) -> &str; // 不正。入力が 2 つある
fn frob<'a, 'b>(s: &'a str, t: &'b str) -> &str; // 展開された形。出力ライフタイムが決まらない

fn get_mut(&mut self) -> &mut T; // 省略された形
fn get_mut<'a>(&'a mut self) -> &'a mut T; // 展開された形

   panded
fn args<T: ToCStr>(&mut self, args: &[T]) -> &mut Command; // 省略された形
fn args<'a, 'b, T: ToCStr>(&'a mut self, args: &'b [T]) -> &'a mut Command; // 展開された形

fn new(buf: &mut [u8]) -> BufWriter; // 省略された形
fn new<'a>(buf: &'a mut [u8]) -> BufWriter<'a>; // 展開された形
```

ミュータビリティ

Rust におけるミュータビリティ、何かを変更する能力は、他のプログラミング言語とはすこし異なっています。ミュータビリティの一つ目の特徴は、それがデフォルトでは無いという点です:

```
let x = 5;
x = 6; // エラー!
```

mut キーワードによりミュータビリティを導入できます:

```
let mut x = 5;

x = 6; // 問題なし!
```

これはミュータブルな 変数束縛です。束縛がミュータブルであるとき、その束縛が何を指すかを変更して良いことを意味します。つまり上記の例では、x の値を変更したのではなく、ある i32 から別の値へと束縛が変わったのです。

束縛が指す先を変更する場合は、ミュータブル参照を使う必要があるでしょう:

```
let mut x = 5;
let y = &mut x;
```

y はミュータブル参照へのイミュータブルな束縛であり、 y を他の束縛に変える (y = &mut z) ことはできません。しかし、y に束縛されているものを変化させること (*y = 5) は可能です。微妙な区別です。

もちろん、両方が必要ならば:

```
let mut x = 5;
let mut y = &mut x;
```

今度は y が他の値を束縛することもできますし、参照している値を変更することもできます。

mut は パターンの一部を成すことに十分注意してください。つまり、次のようなことが可能です:

```
let (mut x, y) = (5, 6);

fn foo(mut x: i32) {
```

内側 vs. 外側のミュータビリティ

一方で、Rust で「イミュータブル (immutable)」について言及するとき、変更不可能であることを意味しない: 「外側のミュータビリティ (exterior mutability)」を表します。例として、`Arc<T>`[*22] を考えます:

```
use std::sync::Arc;

let x = Arc::new(5);
let y = x.clone();
```

`clone()` を呼び出すとき、`Arc<T>` は参照カウントを更新する必要があります。しかし、 ここでは `mut` を一切使っていません。つまり `x` はイミュータブルな束縛であり、`&mut 5` のような引数もとりません。一体どうなっているの?

これを理解するには、Rust 言語の設計哲学の中心をなすメモリ安全性と、Rust がそれを保証するメカニズムである所有権 システム、 特に借用 に立ち返る必要があります。

> 次の 2 種類の借用のどちらか 1 つを持つことはありますが、両方を同時に持つことはありません。
> - リソースに対する 1 つ以上の参照（`&T`）
> - ただ 1 つのミュータブルな参照（`&mut T`）

つまり、「イミュータビリティ」の真の定義はこうです: これは 2 箇所から指されても安全ですか? `Arc<T>` の例では、イエス: 変更は完全にそれ自身の構造の内側で行われます。ユーザからは見えません。このような理由により、 `clone()` を用いて `&T` を配るのです。仮に `&mut T` を配ってしまうと、問題になるでしょう。 （訳注: `Arc<T>`を用いて複数スレッドにイミュータブル参照を配布し、スレッド間でオブジェクトを共有できます。）

`std::cell`[*23] モジュールにあるような別の型では、反対の性質: 内側のミュータビリティ (interior mutability) を持ちます。 例えば:

[*22] http://doc.rust-lang.org/std/sync/struct.Arc.html
[*23] http://doc.rust-lang.org/std/cell/index.html

```
use std::cell::RefCell;

let x = RefCell::new(42);

let y = x.borrow_mut();
```

RefCell では `borrow_mut()` メソッドによって、その内側にある値への `&mut` 参照を配ります。それって危ないのでは? もし次のようにすると:

```
use std::cell::RefCell;

let x = RefCell::new(42);

let y = x.borrow_mut();
let z = x.borrow_mut();
```

実際に、このコードは実行時にパニックするでしょう。これが `RefCell` が行うことです: Rust の借用ルールを実行時に強制し、違反したときには `panic!` を呼び出します。これにより Rust のミュータビリティ・ルールのもう一つの特徴を回避できるようになります。最初に見ていきましょう。

フィールド・レベルのミュータビリティ

ミュータビリティとは、借用 (`&mut`) や束縛 (`let mut`) に関する属性です。これが意味するのは、例えば、一部がミュータブルで一部がイミュータブルなフィールドを持つ `struct` は作れないということです。

```
struct Point {
    x: i32,
    mut y: i32, // ダメ
}
```

構造体のミュータビリティは、それへの束縛の一部です。

```
struct Point {
    x: i32,
    y: i32,
```

```
}

let mut a = Point { x: 5, y: 6 };

a.x = 10;

let b = Point { x: 5, y: 6};

b.x = 10; // エラー: イミュータブルなフィールド `b.x` へ代入できない
```

しかし、Cell<T>[*24] を使えば、フィールド・レベルのミュータビリティをエミュレートできます。

```
use std::cell::Cell;

struct Point {
    x: i32,
    y: Cell<i32>,
}

let point = Point { x: 5, y: Cell::new(6) };

point.y.set(7);

println!("y: {:?}", point.y);
```

このコードは y: Cell { value: 7 } と表示するでしょう。ちゃんと y を更新できました。

構造体

struct はより複雑なデータ型を作る方法の 1 つです。例えば、もし私たちが 2 次元空間の座標に関する計算を行っているとして、x と y 、両方の値が必要になるでしょう。

[*24] http://doc.rust-lang.org/std/cell/struct.Cell.html

```
let origin_x = 0;
let origin_y = 0;
```

struct でこれら 2 つを 1 つのデータ型にまとめることができます。

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let origin = Point { x: 0, y: 0 }; // origin: Point

    println!("The origin is at ({}, {})", origin.x, origin.y);
}
```

ここで多くの情報が出てきましたから、順番に見ていきましょう。まず、struct キーワードを使って構造体とその名前を宣言しています。慣習により、構造体は初めが大文字のキャメルケースで記述しています。PointInSpace であり、 Point_In_Space ではありません。

いつものように、 let で struct のインスタンスを作ることができますが、ここでは key: value スタイルの構文でそれぞれのフィールドに値をセットしています。順序は元の宣言と同じである必要はありません。

最後に、作成された構造体のフィールドは名前を持つため、 origin.x というようにドット表記でアクセスできます。

Rust の他の束縛のように、 struct が持つ値はイミュータブルがデフォルトです。 mut を使うと値をミュータブルにできます。

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let mut point = Point { x: 0, y: 0 };
```

```
    point.x = 5;

    println!("The point is at ({}, {})", point.x, point.y);
}
```

これは `The point is at (5, 0)` と出力されます。

Rust は言語レベルでフィールドのミュータビリティに対応していないため、以下の様に書くことはできません。

```
struct Point {
    mut x: i32,
    y: i32,
}
```

ミュータビリティは束縛に付与できる属性であり、構造体自体に付与できる属性ではありません。もしあなたがフィールドレベルのミュータビリティを使うのであれば、初めこそ奇妙に見えるものの、非常に簡単に実現できる方法があります。以下の方法で少しの間だけミュータブルな構造体を作ることができます。

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let mut point = Point { x: 0, y: 0 };

    point.x = 5;

    let point = point; // この新しい束縛でここから変更できなくなります

    point.y = 6; // これはエラーになります
}
```

アップデート構文

struct の初期化時には、値の一部を他の構造体からコピーしたいことを示す.. を含めることができます。例えば、

```
struct Point3d {
    x: i32,
    y: i32,
    z: i32,
}

let mut point = Point3d { x: 0, y: 0, z: 0 };
point = Point3d { y: 1, .. point };
```

ここでは point に新しい y を与えていますが、x と z は元の値のままです。コピー先は元の構造体と同じである必要はなく、この構文で新しい構造体を作ることもできます。その場合、指定しなかったフィールドは元の構造体からコピーされます。

```
let origin = Point3d { x: 0, y: 0, z: 0 };
let point = Point3d { z: 1, x: 2, .. origin };
```

タプル構造体

Rust には「タプル構造体」と呼ばれる、タプルと struct のハイブリットのようなデータ型があります。タプル構造体自体には名前がありますが、そのフィールドには名前がありません。

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);
```

これら 2 つは同じ値を持つ同士であったとしても等しくありません。

```
let black = Color(0, 0, 0);
let origin = Point(0, 0, 0);
```

ほとんどの場合タプル構造体よりも `struct` を使ったほうが良いです。 `Color` や `Point` はこのようにも書けます。

```
struct Color {
    red: i32,
    blue: i32,
    green: i32,
}

struct Point {
    x: i32,
    y: i32,
    z: i32,
}
```

今、私たちはフィールドの位置ではなく実際のフィールドの名前を持っています。良い名前は重要で、`struct` を使うということは、実際に名前を持っているということです。

> 訳注: 原文を元に噛み砕くと、「タプルはフィールドの並びによって区別され、構造体はフィールドの名前によって区別されます。これはタプルと構造体の最たる違いであり、構造体を持つことは名前を付けられたデータの集まりを持つことに等しいため、構造体における名前付けは重要です。」といった所でしょうか。

ただし、タプル構造体が非常に便利な場合も**あります**。要素が 1 つだけの場合です。要素の値と区別でき、独自の意味を表現できるような新しい型を作成できることから、私たちはこれを「newtype」パターンと呼んでいます。

```
struct Inches(i32);

let length = Inches(10);

let Inches(integer_length) = length;
println!("length is {} inches", integer_length);
```

上記の通り、標準のタプルと同じように `let` を使って分解することで内部の整数型を取り出すことがで

きます。このケースでは `let Inches(integer_length)` が `integer_length` に `10` を代入します。

Unit-like 構造体

全くメンバを持たない `struct` を定義することもできます。

```
struct Electron;

let x = Electron;
```

このような構造体は「unit-like」であると言われます。空のタプルであり「unit」とも呼ばれる `()` とよく似ているからです。タプル構造体と同様に、 unit-like 構造体も新しい型を定義します。

これは単体では滅多に役に立ちません（マーカ型として使える場合もあります）が、他の機能と組み合わせると便利な場合があります。例えば、ライブラリがイベントを処理する特定のトレイトを実装する構造体の作成を要求するかもしれません。もしその構造体の中に保存すべき値が何もなければ、単に unit-like な `struct` を作るだけで良いのです。

列挙型

Rust の `enum` は、いくつかのヴァリアントのうちからどれか一つをとるデータを表す型です。`enum` の各ヴァリアントは、それぞれ自身に関連するデータを持つこともできます。

```
enum Message {
    Quit,
    ChangeColor(i32, i32, i32),
    Move { x: i32, y: i32 },
    Write(String),
}
```

ヴァリアントの定義のための構文は、構造体を定義するのに使われる構文と似ており、（unit-like 構造体のような）データを持たないヴァリアント、名前付きデータを持つヴァリアント、（タプル構造体のような）名前なしデータを持つヴァリアントがありえます。しかし、別々に構造体を定義する場合とは異なり、 `enum` は一つの型です。 列挙型の値はどのヴァリアントにもマッチしうるのです。このことから、列

挙型は「直和型」(sum type) と呼ばれることもあります。列挙型としてとりうる値の集合は、各ヴァリアントとしてとりうる値の集合の和であるためです。

各ヴァリアントの名前を使うためには、 `::` 構文を使います。すなわち、ヴァリアントの名前は `enum` 自体の名前によってスコープ化されています。これにより、以下のどちらもうまく動きます。

```
let x: Message = Message::Move { x: 3, y: 4 };

enum BoardGameTurn {
    Move { squares: i32 },
    Pass,
}

let y: BoardGameTurn = BoardGameTurn::Move { squares: 1 };
```

どちらのヴァリアントも `Move` という名前ですが、列挙型の名前でスコープ化されているため、衝突することなく使うことができます。

列挙型の値は、ヴァリアントに関連するデータに加え、その値自身がどのヴァリアントであるかという情報を持っています。これを「タグ付き共用体」(tagged union) ということもあります。データが、それ自身がどの型なのかを示す「タグ」をもっているためです。コンパイラはこの情報を用いて、列挙型内のデータへ安全にアクセスすることを強制します。例えば、値をどれか一つのヴァリアントであるかのようにみなして、その中身を取り出すということはできません。

```
fn process_color_change(msg: Message) {
    let Message::ChangeColor(r, g, b) = msg; // コンパイル時エラー
}
```

こういった操作が許されないことで制限されているように感じられるかもしれませんが、この制限は克服できます。それには二つの方法があります。一つは等値性を自分で実装する方法、もう一つは次のセクションで学ぶ `match` 式でヴァリアントのパターンマッチを行う方法です。等値性を実装する方法についてはまだ説明していませんが、**トレイト**のセクションに書いてあります。

関数としてのコンストラクタ

列挙型のコンストラクタも、関数のように使うことができます。例えばこうです。

```
let m = Message::Write("Hello, world".to_string());
```

これは、以下と同じです。

```
fn foo(x: String) -> Message {
    Message::Write(x)
}

let x = foo("Hello, world".to_string());
```

このことは今すぐ役立つことではないのですが、**クロージャ**のセクションでは関数を他の関数へ引数として渡す話をします。例えば、これを **イテレータ**とあわせることで、`String` のベクタから `Message::Write` のベクタへ変換することができます。

```
let v = vec!["Hello".to_string(), "World".to_string()];

let v1: Vec<Message> = v.into_iter().map(Message::Write).collect();
```

マッチ

しばしば、２つ以上の可能な処理が存在するためや、分岐条件が非常に複雑になるために単純な `if`/`else` では充分でない場合があります。 Rust にはキーワード `match` が存在し、複雑な `if`/`else` のグループをさらに強力なもので置き換えられます。以下の例を見てみましょう:

```
let x = 5;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    4 => println!("four"),
    5 => println!("five"),
    _ => println!("something else"),
```

```
}
```

match は一つの式とその式の値に基づく複数のブランチを取ります。一つ一つの「腕」は val => expression という形式を取ります。値がマッチした時に、対応する腕の式が評価されます。 このような式が match と呼ばれるのは「パターンマッチ」という用語に由来します。パターンのセクションではこの部分に書けるすべてのパターンを説明しています。

数ある match の利点のうちの一つに「網羅性検査」を行なうということが上げられます。例えば最後の _ の腕を消すと、コンパイラはエラーを出します。

```
error: non-exhaustive patterns: `_` not covered
```

Rust は何かしらの値を忘れていると教えてくれています。 コンパイラは x が任意の 32bit の値、例えば-2,147,483,648 から 2,147,483,647 を取り得ると推論します。_ が「がらくた入れ」として振舞います、match の腕で指定され **なかった** 可能な値全てを捕捉します。先の例で見た通り、 match の腕は 1〜5 の値を書いたので、x が 6、あるいは他の値だった時は _ に捕捉されます。

match は式でもあります、これはつまり let 束縛の右側や式が使われているところで利用することができるということを意味しています。

```
let x = 5;

let number = match x {
    1 => ”one”,
    2 => ”two”,
    3 => ”three”,
    4 => ”four”,
    5 => ”five”,
    _ => ”something else”,
};
```

このようにして、ある型から他の型への変換がうまく書ける場合があります。この例では整数が String に変換されています。

列挙型に対するマッチ

match の他の重要な利用方法としては列挙型のバリアントを処理することがあります:

```
enum Message {
    Quit,
    ChangeColor(i32, i32, i32),
    Move { x: i32, y: i32 },
    Write(String),
}

fn quit() { /* ... */ }
fn change_color(r: i32, g: i32, b: i32) { /* ... */ }
fn move_cursor(x: i32, y: i32) { /* ... */ }

fn process_message(msg: Message) {
    match msg {
        Message::Quit => quit(),
        Message::ChangeColor(r, g, b) => change_color(r, g, b),
        Message::Move { x: x, y: y } => move_cursor(x, y),
        Message::Write(s) => println!("{}", s),
    };
}
```

繰り返しになりますが、Rust コンパイラは網羅性のチェックを行い、列挙型のすべてのバリアントに対して、マッチする腕が存在することを要求します。もし、一つでもマッチする腕のないバリアントを残している場合、 _ を用いるか可能な腕を全て書くかしなければコンパイルエラーが発生します。

先ほど説明した値に対する match の利用とは異なり、列挙型のバリアントに基いた分岐に if を用いることはできません。 列挙型のバリアントに基いた分岐には if let 文を用いることが可能です。 if let は match の短縮形と捉えることができます。

パターン

パターンは Rust において極めて一般的です。 パターンは変数束縛、マッチ文 などで使われています。さあ、めくるめくパターンの旅を始めましょう！

簡単な復習：リテラルに対しては直接マッチ出来ます。 また、 _ は「任意の」ケースとして振る舞います。

```
let x = 1;

match x {
    1 => println!(”one”),
    2 => println!(”two”),
    3 => println!(”three”),
    _ => println!(”anything”),
}
```

これは one を表示します。

パターンには一つ落とし穴があります。新しい束縛を導入する他の構文と同様、パターンはシャドーイングをします。例えば：

```
let x = 1;
let c = 'c';

match c {
    x => println!(”x: {} c: {}”, x, c),
}

println!(”x: {}”, x)
```

これは以下のように出力します。

```
x: c c: c
x: 1
```

別の言い方をすると、 x => は値をパターンにマッチさせ、 x という名前の束縛を導入します。この束縛はマッチの腕内で有効で、値は c を取ります。このマッチのスコープ外の x はスコープ内の x の値に何の関係もないことに注意して下さい。 既に x という束縛が存在していたので、新たに導入した x は、その古い x をシャドーイングします。

複式パターン

| を使うと、複数のパターンにマッチさせることができます：

```
let x = 1;

match x {
    1 | 2 => println!("one or two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

これは、 one or two を出力します。

分配束縛

struct のような複合データ型が存在するとき、パターン内でその値を分解することができます。

```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
    Point { x, y } => println!("({},{})", x, y),
}
```

値に別の名前を付けたいときは、 : を使うことができます。

```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
    Point { x: x1, y: y1 } => println!("({},{})", x1, y1),
}
```

値の一部にだけ興味がある場合は、値のすべてに名前を付ける必要はありません。

```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
    Point { x, .. } => println!("x is {}", x),
}
```

これは `x is 0` を出力します。

最初のメンバだけでなく、どのメンバに対してもこの種のマッチを行うことができます。

```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };
```

```
match origin {
    Point { y, .. } => println!("y is {}", y),
}
```

これは `y is 0` を出力します。

この「分配束縛」 (destructuring) と呼ばれる振る舞いは、タプル や 列挙型のような、任意の複合データ型で使用できます。

束縛の無視

パターン内の型や値を無視するために _ を使うことができます。例として、 `Result<T, E>` に対して `match` をしてみましょう：

```
match some_value {
    Ok(value) => println!("got a value: {}", value),
    Err(_) => println!("an error occurred"),
}
```

最初の部分では `Ok` ヴァリアント内の値に `value` を束縛しています。 しかし `Err` 部分では、ヴァリアント内のエラー情報を無視して一般的なエラーメッセージを表示するために_ を使っています。

_ は束縛を導入するどのようなパターンにおいても有効です。これは大きな構造の一部を無視する際に有用です。

```
fn coordinate() -> (i32, i32, i32) {
    // 3 要素のタプルを生成して返す
}

let (x, _, z) = coordinate();
```

ここでは、タプルの最初と最後の要素に `x` と `z` を束縛します。

_ はそもそも値に束縛されない、つまり値をムーブしないということは特筆に値します。

```
let tuple: (u32, String) = (5, String::from("five"));

// この場合 tuple はムーブされます。何故なら第 2 要素の文字列がムーブされているからです:
let (x, _s) = tuple;

// 次の行は「error: use of partially moved value: `tuple`」になります。
// println!("Tuple is: {:?}", tuple);

// しかしながら、

let tuple = (5, String::from("five"));

// この場合は tuple はムーブ _されません_ 。何故なら第 2 要素の文字列はムーブされず、第 1 要素の u32 は Copy だからです:
let (x, _) = tuple;

// つまりこれは動きます:
println!("Tuple is: {:?}", tuple);
```

またこれは、（訳注: 値に束縛されない）一時変数は文の終わりでドロップされるということでもあります。

```
// 生成された String は変数を束縛しないので即座にドロップされる
let _ = String::from("  hello  ").trim();
```

複数の値を無視するのには `..` パターンが使えます。

```
enum OptionalTuple {
    Value(i32, i32, i32),
    Missing,
}

let x = OptionalTuple::Value(5, -2, 3);

match x {
```

```
    OptionalTuple::Value(..) => println!("Got a tuple!"),
    OptionalTuple::Missing => println!("No such luck."),
}
```

これは `Got a tuple!` を出力します。

ref と ref mut

参照を取得したいときは `ref` キーワードを使いましょう。

```
let x = 5;

match x {
    ref r => println!("Got a reference to {}", r),
}
```

これは `Got a reference to 5` を出力します。

ここで `match` 内の `r` は `&i32` 型を持っています。 言い換えると、 `ref` キーワードはパターン内で使う参照を **作り出します** 。ミュータブルな参照が必要な場合は、同様に `ref mut` を使います。

```
let mut x = 5;

match x {
    ref mut mr => println!("Got a mutable reference to {}", mr),
}
```

範囲

`...` で値の範囲にマッチさせることができます：

```
let x = 1;

match x {
```

```
    1 ... 5 => println!(”one through five”),
    _ => println!(”anything”),
}
```

これは `one through five` を出力します。

範囲は多くの場合、整数か `char` 型で使われます：

```
let x = '';

match x {
    'a' ... 'j' => println!(”early letter”),
    'k' ... 'z' => println!(”late letter”),
    _ => println!(”something else”),
}
```

これは `something else` を出力します。

束縛

`@` で値に名前を束縛することができます。

```
let x = 1;

match x {
    e @ 1 ... 5 => println!(”got a range element {}”, e),
    _ => println!(”anything”),
}
```

これは `got a range element 1` を出力します。データ構造の一部に対して複雑なマッチングをしたいときに有用です：

```
#[derive(Debug)]
struct Person {
    name: Option<String>,
```

```
}

let name = "Steve".to_string();
let x: Option<Person> = Some(Person { name: Some(name) });
match x {
    Some(Person { name: ref a @ Some(_), .. }) => println!("{:?}", a),
    _ => {}
}
```

これは `Some("Steve")` を出力します。内側の `name` の値への参照に `a` を束縛します。

`@` を `|` と組み合わせて使う場合は、それぞれのパターンで同じ名前が束縛されるようにする必要があります：

```
let x = 5;

match x {
    e @ 1 ... 5 | e @ 8 ... 10 => println!("got a range element {}", e),
    _ => println!("anything"),
}
```

ガード

`if` を使うことで「マッチガード」を導入することができます：

```
enum OptionalInt {
    Value(i32),
    Missing,
}

let x = OptionalInt::Value(5);

match x {
    OptionalInt::Value(i) if i > 5 => println!("Got an int bigger than five!"),
```

```
    OptionalInt::Value(..) => println!("Got an int!"),
    OptionalInt::Missing => println!("No such luck."),
}
```

これは `Got an int!` を出力します。

複式パターンで `if` を使うと、 `if` は `|` の両側に適用されます：

```
let x = 4;
let y = false;

match x {
    4 | 5 if y => println!("yes"),
    _ => println!("no"),
}
```

これは `no` を出力します。なぜなら `if` は `4 | 5` 全体に適用されるのであって、 `5` 単独に対して適用されるのではないからです。つまり `if` 節は以下のように振舞います：

```
(4 | 5) if y => ...
```

次のようには解釈されません：

```
4 | (5 if y) => ...
```

混ぜてマッチ

ふう、マッチには様々な方法があるのですね。やりたいことに応じて、それらを混ぜてマッチさせることもできます：

```
match x {
    Foo { x: Some(ref name), y: None } => ...
}
```

パターンはとても強力です。上手に使いましょう。

メソッド構文

関数は素晴らしいのですが、幾つかのデータに対し複数の関数をまとめて呼び出したい時、困ったことになります。以下のコードについて考えてみます。

```
baz(bar(foo));
```

私たちはこれを左から右へ、「baz bar foo」と読むことになりますが、関数が呼び出される順番は異なり、内側から外へ「foo bar baz」となります。 もし代わりにこうできたらいいとは思いませんか?

```
foo.bar().baz();
```

最初の質問でもう分かっているかもしれませんが、幸いにもこれは可能です！ Rust は `impl` キーワードによってこの「メソッド呼び出し構文」の機能を提供しています。

メソッド呼び出し

どんな風に動作するかが以下になります。

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

fn main() {
    let c = Circle { x: 0.0, y: 0.0, radius: 2.0 };
```

```
    println!("{}", c.area());
}
```

これは `12.566371` と出力します。

私たちは円を表す `struct` を作りました。 その際 `impl` ブロックを書き、その中に `area` というメソッドを定義しています。

メソッドに渡す特別な第 1 引数として、 `self` 、 `&self` 、`&mut self` という 3 つの変形があります。 第一引数は `foo.bar()` に於ける `foo` だと考えて下さい。 3 つの変形は `foo` が成り得る 3 種類の状態に対応しており、それぞれ `self` がスタック上の値である場合、 `&self` が参照である場合、`&mut self` がミュータブルな参照である場合となっています。`area` では `&self` を受け取っているため、他の引数と同じように扱えます。 引数が `Circle` であるのは分かっていますから、他の `struct` でするように `radius` へアクセスできます。

所有権を渡すよりも借用を好んで使うべきなのは勿論のこと、ミュータブルな参照よりもイミュータブルな参照を渡すべきですから、`&self` を常用すべきです。以下が 3 種類全ての例です。

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn reference(&self) {
        println!("taking self by reference!");
    }

    fn mutable_reference(&mut self) {
        println!("taking self by mutable reference!");
    }

    fn takes_ownership(self) {
        println!("taking ownership of self!");
    }
```

```
}
```

好きな数だけ `impl` ブロックを使用することができます。前述の例は以下のように書くこともできるでしょう。

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn reference(&self) {
        println!("taking self by reference!");
    }
}

impl Circle {
    fn mutable_reference(&mut self) {
        println!("taking self by mutable reference!");
    }
}

impl Circle {
    fn takes_ownership(self) {
        println!("taking ownership of self!");
    }
}
```

メソッドチェーン

ここまでで、`foo.bar()` というようなメソッドの呼び出し方が分かりましたね。 ですが元の例の `foo.bar().baz()` はどうなっているのでしょう？ これは「メソッドチェーン」と呼ばれています。 以下の例を見て下さい。

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }

    fn grow(&self, increment: f64) -> Circle {
        Circle { x: self.x, y: self.y, radius: self.radius + increment }
    }
}

fn main() {
    let c = Circle { x: 0.0, y: 0.0, radius: 2.0 };
    println!("{}", c.area());

    let d = c.grow(2.0).area();
    println!("{}", d);
}
```

以下の返す型を確認して下さい。

```
fn grow(&self, increment: f64) -> Circle {
```

単に `Circle` を返しているだけです。このメソッドにより、私たちは新しい `Circle` を任意の大きさに拡大することができます。

関連関数

あなたは `self` を引数に取らない関連関数を定義することもできます。以下のパターンは Rust のコードにおいて非常にありふれた物です。

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn new(x: f64, y: f64, radius: f64) -> Circle {
        Circle {
            x: x,
            y: y,
            radius: radius,
        }
    }
}

fn main() {
    let c = Circle::new(0.0, 0.0, 2.0);
}
```

この「関連関数」(associated function) は新たに `Circle` を構築します。 この関数は `ref.method()` ではなく、`Struct::function()` という構文で呼び出されることに注意して下さい。幾つかの言語では、関連関数を「静的メソッド」(static methods) と呼んでいます。

Builder パターン

ユーザが `Circle` を作成できるようにしつつも、書き換えたいプロパティだけを設定すれば良いようにしたいとしましょう。もし指定が無ければ `x` と `y` が `0.0` 、`radius` が `1.0` であるものとします。Rust は

メソッドのオーバーロードや名前付き引数、可変個引数といった機能がない代わりに Builder パターンを採用しており、それは以下のようになります。

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

struct CircleBuilder {
    x: f64,
    y: f64,
    radius: f64,
}

impl CircleBuilder {
    fn new() -> CircleBuilder {
        CircleBuilder { x: 0.0, y: 0.0, radius: 1.0, }
    }

    fn x(&mut self, coordinate: f64) -> &mut CircleBuilder {
        self.x = coordinate;
        self
    }

    fn y(&mut self, coordinate: f64) -> &mut CircleBuilder {
        self.y = coordinate;
        self
    }
```

```
    fn radius(&mut self, radius: f64) -> &mut CircleBuilder {
        self.radius = radius;
        self
    }

    fn finalize(&self) -> Circle {
        Circle { x: self.x, y: self.y, radius: self.radius }
    }
}

fn main() {
    let c = CircleBuilder::new()
                .x(1.0)
                .y(2.0)
                .radius(2.0)
                .finalize();

    println!("area: {}", c.area());
    println!("x: {}", c.x);
    println!("y: {}", c.y);
}
```

ここではもう 1 つの `struct` である `CircleBuilder` を作成しています。 その中に Builder メソッドを定義しました。 また `Circle` に `area()` メソッドを定義しました。 そして `CircleBuilder` にもう 1 つ `finalize()` というメソッドを作りました。 このメソッドは Builder から最終的な `Circle` を作成します。さて、先程の要求を実施するために型システムを使いました。`CircleBuilder` のメソッドを好きなように組み合わせ、作る `Circle` への制約を与えることができます。

文字列

文字列は、プログラマがマスタすべき重要な概念です。Rust の文字列の扱いは、Rust 言語がシステムプログラミングにフォーカスしているため、少し他の言語と異なります。動的なサイズを持つデータ構造があるといつも、物事は複雑性を孕みます。そして文字列もまたサイズを変更することができるデー

タ構造です。これはつまり、Rust の文字列もまた、C のような他のシステム言語とは少し異なる振る舞いをするということです。

詳しく見ていきましょう。「文字列」は、UTF-8 のバイトストリームとしてエンコードされたユニコードのスカラ値のシーケンスです。すべての文字列は、妥当な UTF-8 のシーケンスであることが保証されています。また、他のシステム言語とは異なり、文字列は null 終端でなく、null バイトを保持することもできます。

Rust には主要な文字列型が二種類あります。`&str` と `String` です。 まず `&str` について説明しましょう。`&str` は「文字列スライス」と呼ばれます。文字列スライスは固定サイズで変更不可能です。文字列スライスは UTF-8 のバイトシーケンスへの参照です。

```
let greeting = "Hello there."; // greeting: &'static str
```

`"Hello there."` は文字列リテラルで、 `&'static str` 型を持ちます。文字列リテラルは、静的にアロケートされた文字列スライスです。これはつまりコンパイルされたプログラム内に保存されていて、プログラムの実行中全てにわたって存在しているということです。`greeting` の束縛はこのように静的にアロケートされた文字列を参照しています。文字列スライスを引数として期待している関数はすべて文字列リテラルを引数に取ることができます。

文字列リテラルは複数行にわたることができます。複数行文字列リテラルには 2 つの形式があります。一つ目の形式は、改行と行頭の空白を含む形式です:

```
let s = "foo
    bar";

assert_eq!("foo\n        bar", s);
```

もう一つは `\` を使って空白と改行を削る形式です:

```
let s = "foo\
    bar";

assert_eq!("foobar", s);
```

通常、`str` には直接アクセスできず、 `&str` 経由でのみアクセス出来ることに注意して下さい。 これは `str` がサイズ不定型であり追加の実行時情報がないと使用できないからです。詳しくはサイズ不定型の

章を読んで下さい。

Rust には &str だけでなく、 String というヒープアロケートされる文字列もあります。この文字列は伸張可能であり、また UTF-8 であることも保証されています。String は一般的に文字列スライスを to_string メソッドで変換することで作成されます。

```
let mut s = "Hello".to_string(); // mut s: String
println!("{}", s);

s.push_str(", world.");
println!("{}", s);
```

String は & によって &str に型強制されます。

```
fn takes_slice(slice: &str) {
    println!("Got: {}", slice);
}

fn main() {
    let s = "Hello".to_string();
    takes_slice(&s);
}
```

このような変換は &str ではなく &str の実装するトレイトを引数として取る関数に対しては自動的には行われません。たとえば、TcpStream::connect[*25] は引数として型 ToSocketAddrs を要求しています。このような関数には &str は渡せますが、 String は&* を用いて明示的に変換しなければなりません。

```
use std::net::TcpStream;

TcpStream::connect("192.168.0.1:3000"); // 引数として &str を渡す

let addr_string = "192.168.0.1:3000".to_string();
TcpStream::connect(&*addr_string); // addr_string を &str に変換して渡す
```

[*25] http://doc.rust-lang.org/std/net/struct.TcpStream.html#method.connect

String を &str として見るコストは低いのですが、&str を String に変換するとメモリアロケーションが発生します。必要がなければ、やるべきではないでしょう！

インデクシング

文字列は妥当な UTF-8 であるため、文字列はインデクシングをサポートしていません:

```
let s = "hello";

println!("The first letter of s is {}", s[0]); // エラー!!!
```

普通、ベクタへの [] を用いたアクセスはとても高速です。しかし、UTF-8 でエンコードされた文字列内の文字は複数のバイト対応することがあるため、文字列の n 番目の文字を探すには文字列上を走査していく必要があります。そのような処理はベクタのアクセスに比べると非常に高コストな演算であり、誤解を招きたくなかったのです。さらに言えば、上の「文字 (letter)」というのは Unicode での定義と厳密には一致しません。文字列をバイト列として見るかコードポイント列として見るか選ぶことができます。

```
let hachiko = "忠犬ハチ公";

for b in hachiko.as_bytes() {
    print!("{}, ", b);
}

println!("");

for c in hachiko.chars() {
    print!("{}, ", c);
}

println!("");
```

これは、以下の様な出力をします:

```
229, 191, 160, 231, 138, 172, 227, 131, 143, 227, 131, 129, 229, 133, 172,
忠, 犬, ハ, チ, 公,
```

ご覧のように、 char の数よりも多くのバイトが含まれています。

インデクシングするのと近い結果を以下の様にして得ることができます:

```
let dog = hachiko.chars().nth(1); // hachiko[1] のような感じで
```

このコードは、chars のリストの上を先頭から走査しなければならないことを強調しています。

スライシング

文字列スライスは以下のようにスライス構文を使って取得することができます:

```
let dog = "hachiko";
let hachi = &dog[0..5];
```

しかし、注意しなくてはならない点はこれらのオフセットは **バイト**であって **文字** のオフセットではないという点です。そのため、以下のコードは実行時に失敗します:

```
let dog = "忠犬ハチ公";
let hachi = &dog[0..2];
```

そして、次のようなエラーが発生します:

```
thread '<main>' panicked at 'index 0 and/or 2 in `忠犬ハチ公` do not lie on
character boundary'
```

連結

String が存在するとき、 &str を末尾に連結することができます:

```
let hello = "Hello ".to_string();
let world = "world!";

let hello_world = hello + world;
```

しかし、2 つの String を連結するには、 & が必要になります:

```
let hello = "Hello ".to_string();
let world = "world!".to_string();

let hello_world = hello + &world;
```

これは、 `&String` が自動的に `&str` に型強制されるためです。 このフィーチャは 「`Deref` による型強制」と呼ばれています。

ジェネリクス

時々、関数やデータ型を書いていると、引数が複数の型に対応したものが欲しくなることもあります。Rust では、ジェネリクスを用いてこれを実現しています。ジェネリクスは型理論において「パラメトリック多相」(parametric polymorphism) と呼ばれ、与えられたパラメータにより (「parametric」) 型もしくは関数が多くの相 (「poly」は「多くの」、「morph」は「相 (かたち)」を意味します) (訳注: ここで「相」は型を指します) を持つことを意味しています。

さて、型理論はもう十分です。続いてジェネリックなコードをいくつか見ていきましょう。Rust が標準ライブラリで提供している型 `Option<T>` はジェネリックです。

```
enum Option<T> {
    Some(T),
    None,
}
```

`<T>` の部分は、前に少し見たことがあると思いますが、これがジェネリックなデータ型であることを示しています。`enum` の宣言内であれば、どこでも `T` を使うことができ、宣言内に登場する同じ型をジェネリック内で `T` 型に置き換えています。型注釈を用いた `Option<T>`の使用例が以下になります。

```
let x: Option<i32> = Some(5);
```

この型宣言では `Option<i32>` と書かれています。 `Option<T>` の違いに注目して下さい。 そう、上記の `Option` では `T` の値は `i32` です。 この束縛の右辺の `Some(T)` では、`T` は `5` となります。 それが `i32` なので、両辺の型が一致するため、Rust は満足します。型が不一致であれば、以下のようなエラーが発生します。

```
let x: Option<f64> = Some(5);
// error: mismatched types: expected `core::option::Option<f64>`,
// found `core::option::Option<_>` (expected f64 but found integral variable)
```

これは f64 を保持する Option<T> が作れないという意味ではありませんからね！ リテラルと宣言の型を合わせなければなりません。

```
let x: Option<i32> = Some(5);
let y: Option<f64> = Some(5.0f64);
```

これだけで結構です。 1 つの定義で、多くの用途に対応できます。

ジェネリクスにおいてジェネリックな型は 1 つまで、といった制限はありません。Rust の標準ライブラリに入っている類似の型 Result<T, E> について考えてみます。

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

この型では T と E の *2 つ* がジェネリックです。ちなみに、大文字の部分はあなたの好きな文字で構いません。もしあなたが望むなら Result<T, E> を、

```
enum Result<A, Z> {
    Ok(A),
    Err(Z),
}
```

のように定義できます。慣習としては、「Type」から第 1 ジェネリックパラメータは T であるべきですし、「Error」から E を用いるのですが、Rust は気にしません。

Result<T, E> 型は計算の結果を返すために使われることが想定されており、正常に動作しなかった場合にエラーの値を返す機能を持っています。

ジェネリック関数

似た構文でジェネリックな型を取る関数を記述できます。

```
fn takes_anything<T>(x: T) {
  // x で何か行う
}
```

構文は 2 つのパーツから成ります。 `<T>` は「この関数は 1 つの型、 `T` に対してジェネリックである」ということであり、 `x: T` は「x は T 型である」という意味です。

複数の引数が同じジェネリックな型を持つこともできます。

```
fn takes_two_of_the_same_things<T>(x: T, y: T) {
    // ...
}
```

複数の型を取るバージョンを記述することも可能です。

```
fn takes_two_things<T, U>(x: T, y: U) {
    // ...
}
```

ジェネリック構造体

また、 `struct` 内にジェネリックな型の値を保存することもできます。

```
struct Point<T> {
    x: T,
    y: T,
}

let int_origin = Point { x: 0, y: 0 };
let float_origin = Point { x: 0.0, y: 0.0 };
```

関数と同様に、 `<T>` がジェネリックパラメータを宣言する場所であり、型宣言において `x: T` を使うのも同じです。

ジェネリックな `struct` に実装を追加したい場合、 `impl` の後に型パラメータを宣言します。

```
impl<T> Point<T> {
    fn swap(&mut self) {
        std::mem::swap(&mut self.x, &mut self.y);
    }
}
```

ここまででありとあらゆる型をとることのできるジェネリクスについて見てきました。多くの場合これらは有用です。 `Option<T>` は既に見た通りですし、のちに `Vec<T>` のような普遍的なコンテナ型を知ることになるでしょう。一方で、その柔軟性と引き換えに表現力を増加させたくなることもあります。それは何故か、そしてその方法を知るためにはトレイト境界 を読んで下さい。

トレイト

トレイトはある型が提供しなければならない機能を Rust のコンパイラに伝える言語機能です。

メソッド構文で関数を呼び出すのに用いていた、`impl` キーワードを思い出して下さい。

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}
```

始めにトレイトをメソッドのシグネチャと共に定義し、続いてある型のためにトレイトを実装するという流れを除けばトレイトはメソッド構文に似ています。この例では、 `Circle` に `HasArea` トレイトを実装しています。

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

trait HasArea {
    fn area(&self) -> f64;
}

impl HasArea for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}
```

このように、 trait ブロックは impl ブロックにとても似ているように見えますが、関数本体を定義せず、型シグネチャだけを定義しています。トレイトを impl するときは、 impl Item とだけ書くのではなく、impl Trait for Item と書きます。

ジェネリック関数におけるトレイト境界

トレイトはある型の振る舞いを確約できるため有用です。ジェネリック関数は制約、あるいは境界が許容する型のみを受け取るためにトレイトを利用できます。以下の関数を考えて下さい、これはコンパイルできません。

```
fn print_area<T>(shape: T) {
    println!("This shape has an area of {}", shape.area());
}
```

Rust は以下のエラーを吐きます。

```
error: no method named `area` found for type `T` in the current scope
```

T はあらゆる型になれるため、 area メソッドが実装されているか確認できません。ですがジェネリック

な T にはトレイト境界を追加でき、境界が実装を保証してくれます。

```
fn print_area<T: HasArea>(shape: T) {
    println!("This shape has an area of {}", shape.area());
}
```

<T: HasArea> 構文は「HasArea トレイトを実装するあらゆる型」という意味です。トレイトは関数の型シグネチャを定義しているため、HasArea を実装するあらゆる型が .area() メソッドを持っていることを確認できます。

トレイトの動作を確認するために拡張した例が以下になります。

```
trait HasArea {
    fn area(&self) -> f64;
}

struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl HasArea for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

struct Square {
    x: f64,
    y: f64,
    side: f64,
}

impl HasArea for Square {
```

```
    fn area(&self) -> f64 {
        self.side * self.side
    }
}

fn print_area<T: HasArea>(shape: T) {
    println!("This shape has an area of {}", shape.area());
}

fn main() {
    let c = Circle {
        x: 0.0f64,
        y: 0.0f64,
        radius: 1.0f64,
    };

    let s = Square {
        x: 0.0f64,
        y: 0.0f64,
        side: 1.0f64,
    };

    print_area(c);
    print_area(s);
}
```

このプログラムの出力は、

```
This shape has an area of 3.141593
This shape has an area of 1
```

見ての通り、上記の `print_area` はジェネリックですが、適切な型が渡されることを保証しています。もし不適切な型を渡すと、

```
print_area(5);
```

コンパイル時エラーが発生します。

```
error: the trait bound `_ : HasArea` is not satisfied [E0277]
```

ジェネリック構造体におけるトレイト境界

ジェネリック構造体もトレイト境界による恩恵を受けることができます。型パラメータを宣言する際に境界を追加するだけで良いのです。以下が新しい型 `Rectangle<T>` とそのメソッド `is_square()` です。

```
struct Rectangle<T> {
    x: T,
    y: T,
    width: T,
    height: T,
}

impl<T: PartialEq> Rectangle<T> {
    fn is_square(&self) -> bool {
        self.width == self.height
    }
}

fn main() {
    let mut r = Rectangle {
        x: 0,
        y: 0,
        width: 47,
        height: 47,
    };

    assert!(r.is_square());
```

```
    r.height = 42;
    assert!(!r.is_square());
}
```

is_square() は両辺が等しいかチェックする必要があるため、両辺の型は core::cmp::PartialEq[*26] トレイトを実装しなければなりません。

```
impl<T: PartialEq> Rectangle<T> { ... }
```

これで、長方形を等値性の比較できる任意の型として定義できました。

上記の例では任意の精度の数値を受け入れる Rectangle 構造体を新たに定義しました-実は、等値性を比較できるほぼ全ての型に対して利用可能なオブジェクトです。同じことを Square や Circle のような HasArea を実装する構造体に対してできるでしょうか?可能では有りますが乗算が必要になるため、それをするにはオペレータトレイトについてより詳しく知らなければなりません。

トレイト実装のルール

ここまでで、構造体へトレイトの実装を追加することだけを説明してきましたが、あらゆる型についてトレイトを実装することもできます。技術的には、i32 に HasArea を実装することも **できなくはない** です。

```
trait HasArea {
    fn area(&self) -> f64;
}

impl HasArea for i32 {
    fn area(&self) -> f64 {
        println!("this is silly");

        *self as f64
    }
}
```

[*26] http://doc.rust-lang.org/core/cmp/trait.PartialEq.html

```
5.area();
```

しかし例え可能であったとしても、そのようなプリミティブ型のメソッドを実装するのは適切でない手法だと考えられています。

ここまでくると世紀末感が漂いますが、手に負えなくなることを防ぐためにトレイトの実装周りには 2 つの制限が設けられています。第 1 に、あなたのスコープ内で定義されていないトレイトは適用されません。例えば、標準ライブラリは `File` に I/O 機能を追加するための `Write` トレイトを提供しています。デフォルトでは、 `File` は `Writes` で定義されるメソッド群を持っていません。

```
let mut f = std::fs::File::open("foo.txt").expect("Couldn't open foo.txt");
let buf = b"whatever"; // buf: &[u8; 8] はバイト文字列リテラルです。
let result = f.write(buf);
```

エラーは以下のようになります。

```
error: type `std::fs::File` does not implement any method in scope named `write`
let result = f.write(buf);
               ^~~~~~~~~
```

始めに `Write` トレイトを `use` する必要があります。

```
use std::io::Write;

let mut f = std::fs::File::open("foo.txt").expect("Couldn't open foo.txt");
let buf = b"whatever";
let result = f.write(buf);
```

これはエラー無しでコンパイルされます。

これは、例え誰かが `i32` へメソッドを追加するような望ましくない何かを行ったとしても、あなたがトレイトを `use` しない限り、影響はないことを意味します。

トレイトの実装における制限はもう 1 つあります。トレイトまたはあなたがそれを実装している型はあなた自身によって定義されなければなりません。より正確に言えば、それらの内の 1 つはあなたが書く `impl` と同一のクレートに定義されなければなりません。Rust のモジュールとパッケージシステムにつ

いての詳細は、クレートとモジュールの章を見てください。

以上により `i32` について `HasArea` 型が実装できるはずです、コードには `HasArea` を定義しましたからね。しかし `i32` に Rust によって提供されている `ToString` を実装しようとすると失敗するはずです、トレイトと型の両方が私達のクレートで定義されていませんからね。

トレイトに関して最後に 1 つ。トレイト境界が設定されたジェネリック関数は「単相化」(monomorphization)（mono: 単一の、morph: 相）されるため、静的ディスパッチが行われます。一体どういう意味でしょうか？ 詳細については、トレイトオブジェクトの章をチェックしてください。

複数のトレイト境界

トレイトによってジェネリックな型パラメータに境界が与えられることを見てきました。

```
fn foo<T: Clone>(x: T) {
    x.clone();
}
```

1 つ以上の境界を与えたい場合、 `+` を使えます。

```
use std::fmt::Debug;

fn foo<T: Clone + Debug>(x: T) {
    x.clone();
    println!("{:?}", x);
}
```

この `T` 型は `Clone` と `Debug` 両方が必要です。

Where 節

ジェネリック型もトレイト境界の数も少ない関数を書いているうちは悪く無いのですが、数が増えるとこの構文ではいよいよ不便になってきます。

```
use std::fmt::Debug;

fn foo<T: Clone, K: Clone + Debug>(x: T, y: K) {
    x.clone();
    y.clone();
    println!("{:?}", y);
}
```

関数名は左端にあり、引数リストは右端にあります。境界を記述する部分が邪魔になっているのです。

Rust は「 where 節」と呼ばれる解決策を持っています。

```
use std::fmt::Debug;

fn foo<T: Clone, K: Clone + Debug>(x: T, y: K) {
    x.clone();
    y.clone();
    println!("{:?}", y);
}

fn bar<T, K>(x: T, y: K) where T: Clone, K: Clone + Debug {
    x.clone();
    y.clone();
    println!("{:?}", y);
}

fn main() {
    foo("Hello", "world");
    bar("Hello", "world");
}
```

foo() は先程見せたままの構文で、 bar() は where 節を用いています。型パラメータを定義する際に境界の設定をせず、引数リストの後ろに where を追加するだけで良いのです。長いリストであれば、空白を加えることもできます。

```
use std::fmt::Debug;

fn bar<T, K>(x: T, y: K)
    where T: Clone,
          K: Clone + Debug {

    x.clone();
    y.clone();
    println!("{:?}", y);
}
```

この柔軟性により複雑な状況であっても可読性を改善できます。

また、where は基本の構文よりも強力です。例えば、

```
trait ConvertTo<Output> {
    fn convert(&self) -> Output;
}

impl ConvertTo<i64> for i32 {
    fn convert(&self) -> i64 { *self as i64 }
}

// T == i32 の時に呼び出せる
fn normal<T: ConvertTo<i64>>(x: &T) -> i64 {
    x.convert()
}

// T == i64 の時に呼び出せる
fn inverse<T>() -> T
        // これは「ConvertTo<i64>」であるかのように ConvertTo を用いている
        where i32: ConvertTo<T> {
    42.convert()
}
```

ここでは where 節の追加機能を披露しています。この節は左辺に型パラメータ T だけでなく具体的な型 (このケースでは i32) を指定できます。この例だと、 i32 は ConvertTo<T> を実装していなければなりません。(それは明らかですから) ここの where 節は i32 が何であるか定義しているというよりも、T に対して制約を設定しているといえるでしょう。

デフォルトメソッド

典型的な実装者がどうメソッドを定義するか既に分かっているならば、トレイトの定義にデフォルトメソッドを加えることができます。例えば、以下の is_invalid() は is_valid() の反対として定義されます。

```
trait Foo {
    fn is_valid(&self) -> bool;

    fn is_invalid(&self) -> bool { !self.is_valid() }
}
```

Foo トレイトの実装者は is_valid() を実装する必要がありますが、デフォルトの動作が加えられている is_invalid() には必要ありません。

```
struct UseDefault;

impl Foo for UseDefault {
    fn is_valid(&self) -> bool {
        println!("Called UseDefault.is_valid.");
        true
    }
}

struct OverrideDefault;

impl Foo for OverrideDefault {
    fn is_valid(&self) -> bool {
        println!("Called OverrideDefault.is_valid.");
        true
```

```
    }

    fn is_invalid(&self) -> bool {
        println!("Called OverrideDefault.is_invalid!");
        true // 予期される is_invalid() の値をオーバーライドする
    }
}

let default = UseDefault;
assert!(!default.is_invalid()); //  「Called UseDefault.is_valid.」を表示

let over = OverrideDefault;
assert!(over.is_invalid()); //  「Called OverrideDefault.is_invalid!」を表示
```

継承

時々、1 つのトレイトの実装に他のトレイトの実装が必要になります。

```
trait Foo {
    fn foo(&self);
}

trait FooBar : Foo {
    fn foobar(&self);
}
```

FooBar の実装者は Foo も実装しなければなりません。以下のようになります。

```
struct Baz;

impl Foo for Baz {
    fn foo(&self) { println!("foo"); }
}
```

```
impl FooBar for Baz {
    fn foobar(&self) { println!("foobar"); }
}
```

Foo の実装を忘れると、Rust は以下のように伝えるでしょう。

```
error: the trait bound `main::Baz : main::Foo` is not satisfied [E0277]
```

Derive

繰り返し Debug や Default のようなトレイトを実装するのは非常にうんざりさせられます。そのような理由から、Rust は自動的にトレイトを実装するためのアトリビュート を提供しています。

```
#[derive(Debug)]
struct Foo;

fn main() {
    println!("{:?}", Foo);
}
```

ただし、derive は以下の特定のトレイトに制限されています。

- Clone[*27]
- Copy[*28]
- Debug[*29]
- Default[*30]
- Eq[*31]
- Hash[*32]

[*27] http://doc.rust-lang.org/core/clone/trait.Clone.html
[*28] http://doc.rust-lang.org/core/marker/trait.Copy.html
[*29] http://doc.rust-lang.org/core/fmt/trait.Debug.html
[*30] http://doc.rust-lang.org/core/default/trait.Default.html
[*31] http://doc.rust-lang.org/core/cmp/trait.Eq.html
[*32] http://doc.rust-lang.org/core/hash/trait.Hash.html

- `Ord`[*33]
- `PartialEq`[*34]
- `PartialOrd`[*35]

Drop

トレイトについて学びましたので、Rust の標準ライブラリによって提供されている具体的なトレイト `Drop`[*36] について説明しましょう。 `Drop` トレイトは値がスコープ外になった時にコードを実行する方法を提供します。

```
struct HasDrop;

impl Drop for HasDrop {
    fn drop(&mut self) {
        println!("Dropping!");
    }
}

fn main() {
    let x = HasDrop;

    // いくつかの処理

} // x はここでスコープ外になります
```

`x` が `main()` の終わりでスコープ外になった時、`Drop` のコードが実行されます。 `Drop` にはメソッドは 1 つだけあり、それもまた `drop()` と呼ばれます。ミュータブルな `self` への参照を引数に取ります。

これだけです！ `Drop` のメカニズムは非常にシンプルですが、少しばかり注意があります。たとえば、値がドロップされる順序は、それらが定義された順序と反対の順序になります。

[*33] http://doc.rust-lang.org/core/cmp/trait.Ord.html
[*34] http://doc.rust-lang.org/core/cmp/trait.PartialEq.html
[*35] http://doc.rust-lang.org/core/cmp/trait.PartialOrd.html
[*36] http://doc.rust-lang.org/std/ops/trait.Drop.html

```
struct Firework {
    strength: i32,
}

impl Drop for Firework {
    fn drop(&mut self) {
        println!("BOOM times {}!!!", self.strength);
    }
}

fn main() {
    let firecracker = Firework { strength: 1 };
    let tnt = Firework { strength: 100 };
}
```

このコードは以下のように出力します。

```
BOOM times 100!!!
BOOM times 1!!!
```

tnt が firecracker （爆竹）が鳴る前に爆発してしまいました。これは TNT が定義されたのは爆竹よりも後だったためです。ラストイン・ファーストアウトです。

Drop は何をするのに向いているのでしょうか？ 一般的に Drop は struct に関連付けられているリソースのクリーンアップに使われます。 たとえば Arc<T> 型[*37] は参照カウントを行う型です。 Drop が呼ばれると、参照カウントがデクリメントされ、もし参照の合計数が 0 になっていたら、内包している値がクリーンアップされます。

if let

if let によって if と let を一体化して用いることが可能となり、ある種のパターンマッチに伴うオーバーヘッドを削減することができます。

[*37] http://doc.rust-lang.org/std/sync/struct.Arc.html

例えば今、 Option<T> 型の値が有るとして、 この値が Some<T> であるならば何らかの関数を呼び出し、None ならば何もしたくないとしましょうそのような処理は例えば以下のように書けるでしょう:

```
match option {
    Some(x) => { foo(x) },
    None => {},
}
```

このような場合 match を用いなくても良く、 if を使って以下のように書けます:

```
if option.is_some() {
    let x = option.unwrap();
    foo(x);
}
```

上述のコードのどちらもまだ理想的ではありません。 if let を用いてより良い方法で記述できます:

```
if let Some(x) = option {
    foo(x);
}
```

もし パターンマッチが成功した場合、パターンに含まれる変数に適切に値が割り当てられ、式が評価されます。もしパターンマッチが失敗した場合には何も起こりません。

もしパターンマッチに失敗した場合に実行したいコードが有る場合は else を使うことができます:

```
if let Some(x) = option {
    foo(x);
} else {
    bar();
}
```

while let

同じように、 while let を使うことで、値がパターンにマッチし続ける限り繰り返し実行することができます。例えば以下の様なコードが有るときに:

```
let mut v = vec![1, 3, 5, 7, 11];
loop {
    match v.pop() {
        Some(x) =>  println!("{}", x),
        None => break,
    }
}
```

while let を用いることで、以下のように書くことができます:

```
let mut v = vec![1, 3, 5, 7, 11];
while let Some(x) = v.pop() {
    println!("{}", x);
}
```

トレイトオブジェクト

コードがポリモーフィズムを伴う場合、実際に実行するバージョンを決定するメカニズムが必要です。これは「ディスパッチ」(dispatch) と呼ばれます。ディスパッチには主に静的ディスパッチと動的ディスパッチという 2 つの形態があります。Rust は静的ディスパッチを支持している一方で、「トレイトオブジェクト」(trait objects) と呼ばれるメカニズムにより動的ディスパッチもサポートしています。

背景

本章の後のために、トレイトとその実装が幾つか必要です。 単純に Foo としましょう。 これは String 型の値を返す関数を 1 つ持っています。

```
trait Foo {
    fn method(&self) -> String;
}
```

また、このトレイトを u8 と String に実装します。

```
impl Foo for u8 {
    fn method(&self) -> String { format!("u8: {}", *self) }
}

impl Foo for String {
    fn method(&self) -> String { format!("string: {}", *self) }
}
```

静的ディスパッチ

トレイト境界を使ってこのトレイトで静的ディスパッチが出来ます。

```
fn do_something<T: Foo>(x: T) {
    x.method();
}

fn main() {
    let x = 5u8;
    let y = "Hello".to_string();

    do_something(x);
    do_something(y);
}
```

これは Rust が `u8` と `String` それぞれ専用の `do_something()` を作成し、それら特殊化された関数を宛てがうように呼び出しの部分を書き換えるという意味です。(訳注: 作成された専用の `do_something()` は「特殊化された関数」(specialized function) と呼ばれます)

```
fn do_something_u8(x: u8) {
    x.method();
}

fn do_something_string(x: String) {
    x.method();
```

```
}

fn main() {
    let x = 5u8;
    let y = "Hello".to_string();

    do_something_u8(x);
    do_something_string(y);
}
```

これは素晴らしい利点です。呼び出される関数はコンパイル時に分かっているため、静的ディスパッチは関数呼び出しをインライン化できます。インライン化は優れた最適化の鍵です。静的ディスパッチは高速ですが、バイナリ的には既にあるはずの同じ関数をそれぞれの型毎に幾つもコピーするため、トレードオフとして「コードの膨張」(code bloat) が発生してしまいます。

その上、コンパイラは完璧ではなく、「最適化」したコードが遅くなってしまうこともあります。例えば、あまりにも熱心にインライン化された関数は命令キャッシュを膨張させてしまいます（地獄の沙汰もキャッシュ次第）。それが `#[inline]` や `#[inline(always)]` を慎重に使うべきである理由の 1 つであり、時として動的ディスパッチが静的ディスパッチよりも効率的である 1 つの理由なのです。

しかしながら、一般的なケースでは静的ディスパッチを使用する方が効率的であり、また、動的ディスパッチを行う薄い静的ディスパッチラッパ関数を実装することは常に可能ですが、その逆はできません。これは静的な呼び出しの方が柔軟性に富むことを示唆しています。標準ライブラリはこの理由から可能な限り静的ディスパッチで実装するよう心がけています。

> 訳注: 「動的ディスパッチを行う薄い静的ディスパッチラッパ関数を実装することは常に可能だがその逆はできない」について
>
> 静的ディスパッチはコンパイル時に定まるのに対し、動的ディスパッチは実行時に結果が分かります。従って、動的ディスパッチが伴う処理を静的ディスパッチ関数でラッピングし、半静的なディスパッチとすることは常に可能（原文で「thin」と形容しているのはこのため）ですが、動的ディスパッチで遷移した値を元に静的ディスパッチを行うことはできないと言うわけです。

動的ディスパッチ

Rust は「トレイトオブジェクト」と呼ばれる機能によって動的ディスパッチを提供しています。トレイトオブジェクトは `&Foo` か `Box<Foo>` の様に記述され、指定されたトレイトを実装する **あらゆる**型の値を保持する通常の値です。ただし、その正確な型は実行時になって初めて判明します。

トレイトオブジェクトはトレイトを実装した具体的な型を指すポインタから**キャスト** する (e.g. `&x as &Foo`) か、 **型強制**する（e.g. `&Foo` を取る関数の引数として `&x` を用いる）ことで得られます。

これらトレイトオブジェクトの型強制とキャストは `&mut T` を`&mut Foo` へ、 `Box<T>` を `Box<Foo>` へ、というようにどちらもポインタに対する操作ですが、今の所はこれだけです。型強制とキャストは同一です。

この操作がまるでポインタのある型に関するコンパイラの記憶を「消去している」(erasing) ように見えることから、トレイトオブジェクトは時に「型消去」(type erasure) とも呼ばれます。

上記の例に立ち帰ると、キャストによるトレイトオブジェクトを用いた動的ディスパッチの実現にも同じトレイトが使用できます。

```
fn do_something(x: &Foo) {
    x.method();
}

fn main() {
    let x = 5u8;
    do_something(&x as &Foo);
}
```

または型強制によって、

```
fn do_something(x: &Foo) {
    x.method();
}

fn main() {
    let x = "Hello".to_string();
```

```
    do_something(&x);
}
```

トレイトオブジェクトを受け取った関数が `Foo` を実装した型ごとに特殊化されることはありません。関数は 1 つだけ生成され、多くの場合（とはいえ常にではありませんが）コードの膨張は少なく済みます。しかしながら、これは低速な仮想関数の呼び出しが必要となるため、実質的にインライン化とそれに関連する最適化の機会を阻害してしまいます。

■何故ポインタなのか？

Rust はガーベジコレクタによって管理される多くの言語とは異なり、デフォルトではポインタの参照先に値を配置するようなことはしませんから、型によってサイズが違います。関数へ引数として渡されるような値を、スタック領域へムーブしたり保存のためヒープ領域上にメモリをアロケート（デアロケートも同様）するには、コンパイル時に値のサイズを知っていることが重要となります。

`Foo` のためには、 `String` (24 bytes) か `u8` (1 byte) もしくは `Foo` （とにかくどんなサイズでも）を実装する依存クレート内の型のうちから少なくとも 1 つの値を格納する必要があります。ポインタ無しで値を保存した場合、その直後の動作が正しいかどうかを保証する方法がありません。型によって値のサイズが異なるからです。

ポインタの参照先に値を配置することはトレイトオブジェクトを渡す場合に値自体のサイズが無関係になり、ポインタのサイズのみになることを意味しています。

■トレイトオブジェクトの内部表現

トレイトのメソッドはトレイトオブジェクト内にある伝統的に「vtable」（これはコンパイラによって作成、管理されます）と呼ばれる特別な関数ポインタのレコードを介して呼び出されます。

トレイトオブジェクトは単純ですが難解でもあります。核となる表現と設計は非常に率直ですが、複雑なエラーメッセージを吐いたり、予期せぬ振る舞いが見つかったりします。

単純な例として、トレイトオブジェクトの実行時の表現から見て行きましょう。`std::raw` モジュールは複雑なビルドインの型と同じレイアウトの構造体を格納しており、トレイトオブジェクトも含まれています[*38]。

[*38] https://doc.rust-lang.org/std/raw/

```
pub struct TraitObject {
    pub data: *mut (),
    pub vtable: *mut (),
}
```

つまり、 `&Foo` のようなトレイトオブジェクトは「data」ポインタと「vtable」ポインタから成るわけです。

data ポインタはトレイトオブジェクトが保存している（何らかの不明な型 `T` の）データを指しており、vtable ポインタは `T` への `Foo` の実装に対応する vtable（「virtual method table」）を指しています。

vtable は本質的には関数ポインタの構造体で、実装内における各メソッドの具体的な機械語の命令列を指しています。`trait_object.method()` のようなメソッド呼び出しを行うと vtable の中から適切なポインタを取り出し動的に呼び出しを行います。例えば、

```
struct FooVtable {
    destructor: fn(*mut ()),
    size: usize,
    align: usize,
    method: fn(*const ()) -> String,
}

// u8:

fn call_method_on_u8(x: *const ()) -> String {
    // コンパイラは `x` が u8 を指しているときにのみこの関数が呼ばれることを保障します
    let byte: &u8 = unsafe { &*(x as *const u8) };

    byte.method()
}

static Foo_for_u8_vtable: FooVtable = FooVtable {
    destructor: /* コンパイラマジック */,
    size: 1,
    align: 1,
```

```
    // 関数ポインタへキャスト
    method: call_method_on_u8 as fn(*const ()) -> String,
};


// String:

fn call_method_on_String(x: *const ()) -> String {
    // コンパイラは `x` が String を指しているときにのみこの関数が呼ばれることを保障します
    let string: &String = unsafe { &*(x as *const String) };

    string.method()
}

static Foo_for_String_vtable: FooVtable = FooVtable {
    destructor: /* コンパイラマジック */,
    // この値は 64bit コンピュータ向けのものです、32bit コンピュータではこの半分にします
    size: 24,
    align: 8,

    method: call_method_on_String as fn(*const ()) -> String,
};
```

各 vtable の `destructor` フィールドは vtable が対応する型のリソースを片付ける関数を指しています。`u8` の vtable は単純な型なので何もしませんが、 `String` の vtable はメモリを解放します。 このフィールドは `Box<Foo>` のような自作トレイトオブジェクトのために必要であり、 `Box` によるアロケートは勿論のことスコープ外に出た際に内部の型のリソースを片付けるのにも必要です。`size` 及び `align` フィールドは消去された型のサイズとアライメント要件を保存しています。これらの情報はデストラクタにも組み込まれているため現時点では基本的に使われていませんが、将来、トレイトオブジェクトがより柔軟になることで使われるようになるでしょう。

例えば `Foo` を実装する値を幾つか得たとします。 `Foo` トレイトオブジェクトを作る、あるいは使う時のコードを明示的に書いたものは少しだけ似ているでしょう。(型の違いを無視すればですが、どのみちただのポインタになります)

```
let a: String = "foo".to_string();
let x: u8 = 1;

// let b: &Foo = &a;
let b = TraitObject {
    // データを保存
    data: &a,
    // メソッドを保存
    vtable: &Foo_for_String_vtable
};

// let y: &Foo = x;
let y = TraitObject {
    // データを保存
    data: &x,
    // メソッドを保存
    vtable: &Foo_for_u8_vtable
};

// b.method();
(b.vtable.method)(b.data);

// y.method();
(y.vtable.method)(y.data);
```

オブジェクトの安全性

全てのトレイトがトレイトオブジェクトとして使えるわけではありません。例えば、ベクタは `Clone` を実装していますが、トレイトオブジェクトを作ろうとすると、

```
let v = vec![1, 2, 3];
let o = &v as &Clone;
```

エラーが発生します。

```
error: cannot convert to a trait object because trait `core::clone::Clone` is not obje
   ct-safe [E0038]
let o = &v as &Clone;
        ^~
note: the trait cannot require that `Self : Sized`
let o = &v as &Clone;
        ^~
```

エラーは `Clone` が「オブジェクト安全」(object-safe) でないと言っています。トレイトオブジェクトにできるのはオブジェクト安全なトレイトのみです。以下の両方が真であるならばトレイトはオブジェクト安全であるといえます。

- トレイトが `Self: Sized` を要求しないこと
- トレイトのメソッド全てがオブジェクト安全であること

では何がメソッドをオブジェクト安全にするのでしょう？　各メソッドは `Self: Sized` を要求するか、以下の全てを満足しなければなりません。

- どのような型パラメータも持ってはならない
- `Self` を使ってはならない

ひゃー！　見ての通り、これらルールのほとんどは `Self` について話しています。「特別な状況を除いて、トレイトのメソッドで `Self` を使うとオブジェクト安全ではなくなる」と考えるのが良いでしょう。

クロージャ

しばしば、関数と **自由変数**を一つにまとめておくことがコードの明確さや再利用に役立つ場合が有ります。自由変数は外部のスコープから来て、関数中で使われるときに「閉じ込め」られます。そのためそのようなまとまりを「クロージャ」と呼び、Rust はこれから見ていくようにクロージャの非常に良い実装を提供しています。

構文

クロージャは以下のような見た目です:

```
let plus_one = |x: i32| x + 1;

assert_eq!(2, plus_one(1));
```

束縛 `plus_one` を作成し、クロージャを代入しています。クロージャの引数はパイプ (`|`) の間に書きます、そしてクロージャの本体は式です、 この場合は `x + 1` がそれに当たります。 `{ }` が式であることを思い出して下さい、そのため複数行のクロージャを作成することも可能です:

```
let plus_two = |x| {
    let mut result: i32 = x;

    result += 1;
    result += 1;

    result
};

assert_eq!(4, plus_two(2));
```

いくつかクロージャと通常の `fn` で定義される関数との間の違いに気がつくことでしょう。一つ目はクロージャの引数や返り値の型を示す必要が無いことです。型を以下のように示すことも可能です:

```
let plus_one = |x: i32| -> i32 { x + 1 };

assert_eq!(2, plus_one(1));
```

しかし、このように型を示す必要はありません。なぜでしょう？ 一言で言えば、これは使いやすさのためです。名前の有る関数の型を全て指定するのはドキュメンテーションや型推論の役に立ちますが、クロージャの型は殆ど示されません、これはクロージャたちが匿名であり、さらに名前付きの関数が引き起こすと思われるような定義から離れた箇所で発生するエラーの要因ともならないためです。

通常の関数との違いの二つ目は、構文が大部分は似ていますがほんの少しだけ違うという点です。比較がしやすいようにスペースを適宜補って以下に示します:

```
fn  plus_one_v1   (x: i32) -> i32 { x + 1 }
let plus_one_v2 = |x: i32| -> i32 { x + 1 };
let plus_one_v3 = |x: i32|          x + 1  ;
```

小さな違いは有りますが殆どの部分は同じです。

クロージャとクロージャの環境

クロージャの環境は引数やローカルな束縛に加えてクロージャを囲んでいるスコープ中の束縛を含むことができます。例えば以下のようになります:

```
let num = 5;
let plus_num = |x: i32| x + num;

assert_eq!(10, plus_num(5));
```

クロージャ plus_num はスコープ内の let 束縛 num を参照しています。 より厳密に言うと、クロージャ plus_num は束縛を借用しています。もし、この束縛と衝突する処理を行うとエラーが発生します。例えば、以下のようなコードでは:

```
let mut num = 5;
let plus_num = |x: i32| x + num;

let y = &mut num;
```

以下のエラーを発生させます:

```
error: cannot borrow `num` as mutable because it is also borrowed as immutable
    let y = &mut num;
                ^~~
note: previous borrow of `num` occurs here due to use in closure; the immutable
  borrow prevents subsequent moves or mutable borrows of `num` until the borrow
  ends
    let plus_num = |x| x + num;
                   ^~~~~~~~~~
```

```
note: previous borrow ends here
fn main() {
    let mut num = 5;
    let plus_num = |x| x + num;

    let y = &mut num;
}
^
```

冗長ですが役に立つエラーメッセージです! エラーが示しているように、クロージャが既に num を借用しているために、 num の変更可能な借用を取得することはできません。もしクロージャがスコープ外になるようにした場合以下のようにできます:

```
let mut num = 5;
{
    let plus_num = |x: i32| x + num;

} // plus_num がスコープ外に出て、num の借用が終わります

let y = &mut num;
```

もしクロージャが num を要求した場合、Rust は借用する代わりに環境の所有権を取りムーブします。そのため、以下のコードは動作しません:

```
let nums = vec![1, 2, 3];

let takes_nums = || nums;

println!("{:?}", nums);
```

このコードは以下の様なエラーを発生させます:

```
note: `nums` moved into closure environment here because it has type
  `[closure(()) -> collections::vec::Vec<i32>]`, which is non-copyable
let takes_nums = || nums;
```

```
                    ^~~~~~
```

Vec<T> はその要素に対する所有権を持っています、それゆえそれらの要素をクロージャ内で参照した場合、 nums の所有権を取ることになります。 これは nums を nums の所有権を取る関数に渡した場合と同じです。

move クロージャ

move キーワードを用いることで、クロージャに環境の所有権を取得することを強制することができます。

```
let num = 5;

let owns_num = move |x: i32| x + num;
```

このようにすると move というキーワードにもかかわらず、変数は通常の move のセマンティクスに従います。この場合、 5 は Copy を実装しています、 そのため owns_num は num のコピーの所有権を取得します。では、なにが異なるのでしょうか？

```
let mut num = 5;

{
    let mut add_num = |x: i32| num += x;

    add_num(5);
}

assert_eq!(10, num);
```

このケースでは、クロージャは num の変更可能な参照を取得し、add_num を呼び出した時、期待通りに num の値を変更します。 またクロージャ add_num はその環境を変更するため mut として宣言する必要があります。

もしクロージャを move に変更した場合、結果が異なります:

```
let mut num = 5;
```

```
{
    let mut add_num = move |x: i32| num += x;

    add_num(5);
}

assert_eq!(5, num);
```

結果は 5 になります。 num の変更可能な借用を取得するのではなく、 num のコピーの所有権を取得します。

move クロージャを捉えるもう一つの観点は: move クロージャは独自のスタックフレームを持っているという点です。move クロージャは自己従属していますが、 move でないクロージャはクロージャを作成したスタックフレームと紐付いています。これは一般的に、move でないクロージャを関数から返すことはできないということを意味しています。

クロージャを引数や返り値にすることについて説明する間に、クロージャの実装についてもう少し説明する必要があります。システム言語として Rust はコードの動作についてコントロールする方法を大量に提供しています、そしてそれはクロージャも例外ではありません。

クロージャの実装

Rust におけるクロージャの実装は他の言語とは少し異なります。Rust におけるクロージャは実質的にトレイトへの糖衣構文です。続きの説明を読む前に トレイト やトレイトオブジェクトについてのセクションを学ぶ前に読みたくなるでしょう。

よろしいですか？　では、続きを説明いたします。

クロージャの内部的な動作を理解するための鍵は少し変わっています: 関数を呼び出すのに () を 例えば foo() の様に使いますが、この () はオーバーロード可能な演算子です。この事実から残りの全てを正しく理解することができます。Rust では、トレイトを演算子のオーバーロードに利用します。それは関数の呼び出しも例外ではありません。 () をオーバーロードするのに利用可能な、3 つの異なるトレイトが存在します:

```
pub trait Fn<Args> : FnMut<Args> {
    extern "rust-call" fn call(&self, args: Args) -> Self::Output;
}
```

```
pub trait FnMut<Args> : FnOnce<Args> {
    extern "rust-call" fn call_mut(&mut self, args: Args) -> Self::Output;
}

pub trait FnOnce<Args> {
    type Output;

    extern "rust-call" fn call_once(self, args: Args) -> Self::Output;
}
```

これらのトレイトの間のいくつかの違いに気がつくことでしょう、しかし大きな違いは `self` についてです: `Fn` は `&self` を引数に取ります、 `FnMut` は `&mut self` を引数に取ります、そして `FnOnce` は `self` を引数に取ります。 これは通常のメソッド呼び出しにおける `self` のすべての種類をカバーしています。しかし、これら `self` の各種類を一つの大きなトレイトにまとめるのではなく異なるトレイトに分けています。このようにすることで、どのような種類のクロージャを取るのかについて多くをコントロールすることができます。

クロージャの構文 `|| {}` は上述の 3 つのトレイトへの糖衣構文です。 Rust は環境用の構造体を作成し、適切なトレイトを `impl` し、それを利用します。

クロージャを引数に取る

クロージャが実際にはトレイトであることを学んだので、クロージャを引数としたり返り値としたりする方法を既に知っていることになります: 通常のトレイトと全く同様に行うのです!

これは、静的ディスパッチと動的ディスパッチを選択することができるということも意味しています。手始めに呼び出し可能な何かを引数にとり、それを呼び出し、結果を返す関数を書いてみましょう:

```
fn call_with_one<F>(some_closure: F) -> i32
    where F : Fn(i32) -> i32 {

    some_closure(1)
}

let answer = call_with_one(|x| x + 2);
```

```
assert_eq!(3, answer);
```

クロージャ `|x| x + 2` を `call_with_one` に渡しました。 `call_with_one` はその関数名から推測される処理を行います: クロージャに `1` を与えて呼び出します。

`call_with_one` のシグネチャを詳細に見ていきましょう:

```
fn call_with_one<F>(some_closure: F) -> i32
```

型 `F` の引数を 1 つ取り、返り値として `i32` を返します。この部分は特に注目には値しません。次の部分は:

```
    where F : Fn(i32) -> i32 {
```

`Fn` がトレイトであるために、ジェネリックの境界として `Fn` を指定することができます。 この場合はクロージャは `i32` を引数として取り、 `i32` を返します、そのためジェネリックの境界として `Fn(i32) -> i32` を指定します。

キーポイントがほかにもあります: ジェネリックをトレイトで境界を指定したために、この関数は単相化され、静的ディスパッチをクロージャに対して行います。これはとても素敵です。多くの言語では、クロージャは常にヒープにアロケートされ、常に動的ディスパッチが行われます。Rust ではスタックにクロージャの環境をアロケートし、呼び出しを静的ディスパッチすることができます。これは、しばしばクロージャを引数として取る、イテレータやそれらのアダプタにおいて頻繁に行われます。

もちろん、動的ディスパッチを行いたいときは、そうすることもできます。そのような場合もトレイトオブジェクトが通常どおりに対応します:

```
fn call_with_one(some_closure: &Fn(i32) -> i32) -> i32 {
    some_closure(1)
}

let answer = call_with_one(&|x| x + 2);

assert_eq!(3, answer);
```

トレイトオブジェクト `&Fn` を引数にとります。 また `call_with_one` にクロージャを渡すときに参照を

利用するようにしました、 そのため&|| を利用しています。

関数ポインタとクロージャ

関数ポインタは環境を持たないクロージャのようなものです。そのため、クロージャを引数として期待している関数に関数ポインタを渡すことができます。

```
fn call_with_one(some_closure: &Fn(i32) -> i32) -> i32 {
    some_closure(1)
}

fn add_one(i: i32) -> i32 {
    i + 1
}

let f = add_one;

let answer = call_with_one(&f);

assert_eq!(2, answer);
```

この例では、中間の変数 f が必ずしも必要なわけではありません、関数名を指定することでもきちんと動作します:

```
let answer = call_with_one(&add_one);
```

クロージャを返す

関数を用いたスタイルのコードでは、クロージャを返すことは非常によく見られます。もし、クロージャを返すことを試みた場合、エラーが発生します。これは一見奇妙に思われますが、理解することができます。以下は、関数からクロージャを返すことを試みた場合のコードです:

```
fn factory() -> (Fn(i32) -> i32) {
    let num = 5;

    |x| x + num
}

let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

このコードは以下の長いエラーを発生させます:

```
error: the trait bound `core::ops::Fn(i32) -> i32 : core::marker::Sized` is not satisf
   ied [E0277]
fn factory() -> (Fn(i32) -> i32) {
                ^~~~~~~~~~~~~~~
note: `core::ops::Fn(i32) -> i32` does not have a constant size known at compile-time
fn factory() -> (Fn(i32) -> i32) {
                ^~~~~~~~~~~~~~~
error: the trait bound `core::ops::Fn(i32) -> i32 : core::marker::Sized` is not satisf
   ied [E0277]
let f = factory();
    ^
note: `core::ops::Fn(i32) -> i32` does not have a constant size known at compile-time
let f = factory();
    ^
```

関数から何かを返すにあたって、Rust は返り値の型のサイズを知る必要があります。しかし、 `Fn` はトレイトであるため、そのサイズや種類は多岐にわたることになります: 多くの異なる型が `Fn` を実装できます。何かにサイズを与える簡単な方法は、それに対する参照を取得する方法です、参照は既知のサイズを持っています。そのため、以下のように書くことができます:

```
fn factory() -> &(Fn(i32) -> i32) {
    let num = 5;

    |x| x + num
}

let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

しかし、他のエラーが発生してしまいます:

```
error: missing lifetime specifier [E0106]
fn factory() -> &(Fn(i32) -> i32) {
                ^~~~~~~~~~~~~~~~
```

ふむ。これはリファレンスを利用したので、ライフタイムを指定する必要が有るためです。しかし、`factory()` 関数は引数を何も取りません、 そのためライフタイムの省略 は実施されません。では、どのような選択肢が有るのでしょうか？ `'static` を試してみましょう:

```
fn factory() -> &'static (Fn(i32) -> i32) {
    let num = 5;

    |x| x + num
}

let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

しかし、以下の別のエラーが発生します:

```
error: mismatched types:
```

```
 expected `&'static core::ops::Fn(i32) -> i32`,
    found `[closure@<anon>:7:9: 7:20]`
(expected &-ptr,
    found closure) [E0308]
         |x| x + num
         ^~~~~~~~~~
```

このエラーは `&'static Fn(i32) -> i32` ではなく、`[closure@<anon>:7:9: 7:20]` を使ってしまっているということを伝えています。ちょっと待ってください、一体これはどういう意味でしょう？

それぞれのクロージャはそれぞれの環境用の `struct` を生成し、`Fn` やそれに準ずるものを実装するため、それぞれの型は匿名となります。それらの型はそれらのクロージャのためだけに存在します。そのため Rust はそれらの型を自動生成された名前の代わりに `closure@<anon>` と表示します。

また、このエラーは返り値の型が参照であることを期待しているが、上のコードではそうなっていないということについても指摘しています。もうちょっというと、直接的に `'static` ライフタイムをオブジェクトに割り当てることはできません。そこで、`Fn` をボックス化することで「トレイトオブジェクト」を返すという方法を取ります。そうすると、動作するまであと一歩のところまで来ます:

```
fn factory() -> Box<Fn(i32) -> i32> {
    let num = 5;

    Box::new(|x| x + num)
}
let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

最後に残されたエラーは以下のとおりです:

```
error: closure may outlive the current function, but it borrows `num`,
which is owned by the current function [E0373]
Box::new(|x| x + num)
         ^~~~~~~~~~
```

以前説明したように、クロージャはその環境を借用します。今回の場合は、環境はスタックにアロケー

トされた 5 に束縛された num からできていることから、環境の借用はスタックフレームと同じライフタイムを持っています。そのため、もしこのクロージャを返り値とした場合、 そのあと factory() 関数の処理は終了し、スタックフレームが取り除かれクロージャはゴミとなったメモリを参照することになります! 上のコードに最後の修正を施すことによって動作させることができるようになります:

```
fn factory() -> Box<Fn(i32) -> i32> {
    let num = 5;

    Box::new(move |x| x + num)
}
let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

factory() 内のクロージャを move Fn にすることで、新しいスタックフレームをクロージャのために生成します。そしてボックス化することによって、既知のサイズとなり、現在のスタックフレームから抜けることが可能になります。

共通の関数呼出構文

しばしば、同名の関数が存在する時があります。たとえば、以下のコードでは:

```
trait Foo {
    fn f(&self);
}

trait Bar {
    fn f(&self);
}

struct Baz;

impl Foo for Baz {
```

```
    fn f(&self) { println!("Baz's impl of Foo"); }
}

impl Bar for Baz {
    fn f(&self) { println!("Baz's impl of Bar"); }
}

let b = Baz;
```

もしここで、 `b.f()` を呼びだそうとすると、以下の様なエラーが発生します:

```
error: multiple applicable methods in scope [E0034]
b.f();
  ^~~
note: candidate #1 is defined in an impl of the trait `main::Foo` for the type
`main::Baz`
    fn f(&self) { println!("Baz's impl of Foo"); }
    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
note: candidate #2 is defined in an impl of the trait `main::Bar` for the type
`main::Baz`
    fn f(&self) { println!("Baz's impl of Bar"); }
    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

このような場合は、どのメソッドを呼び出す必要があるのかについて曖昧性を排除する手段が必要です。そのようなフィーチャーは「共通の関数呼び出し構文」と呼ばれ、以下のように書けます:

```
Foo::f(&b);
Bar::f(&b);
```

部分的に見ていきましょう。

```
Foo::
Bar::
```

まず、呼び出しのこの部分は 2 つのトレイト `Foo` と `Bar` の型を表しています。この部分が、実際にどち

らのトレイトのメソッドを呼び出しているのかを指定し、曖昧性を排除している箇所になります。

```
f(&b)
```

`b.f()` のようにメソッド構文を利用して呼び出した時、Rust は `f()` が `&self` を引数に取る場合自動的に `b` を借用します。今回の場合は、そのようには呼び出していないので、明示的に `&b` を渡してやる必要があります。

山括弧形式

すぐ上で説明した、以下のような共通の関数呼び出し構文:

```
Trait::method(args);
```

これは短縮形であり、時々必要になる以下の様な展開された形式もあります:

```
<Type as Trait>::method(args);
```

`<>::` という構文は型のヒントを意味しており、`<>` のなかに型が入ります。 この場合、型は `Type as Trait` となり、 `Trait` のバージョンの `method` が呼ばれる事を期待していることを意味しています。`as Trait` という部分は、曖昧でない場合は省略可能です。山括弧についても同様に省略可能であり、なので先程のさらに短い形になるのです。

長い形式を用いたサンプルコードは以下の通りです:

```
trait Foo {
    fn foo() -> i32;
}

struct Bar;

impl Bar {
    fn foo() -> i32 {
        20
    }
}
```

```
impl Foo for Bar {
    fn foo() -> i32 {
        10
    }
}

fn main() {
    assert_eq!(10, <Bar as Foo>::foo());
    assert_eq!(20, Bar::foo());
}
```

山括弧構文を用いることでトレイトのメソッドを継承されたメソッドの代わりに呼び出すことができます。

クレートとモジュール

プロジェクトが大きくなり始めた際に、コードを小さなまとまりに分割しそれらでプロジェクトを組み立てるのはソフトウェア工学における優れた慣例だと考えられています。幾つかの機能をプライベートとし、また幾つかをパブリックとできるように、はっきりと定義されたインターフェースも重要となります。こういった事柄を容易にするため、Rust はモジュールシステムを有しています。

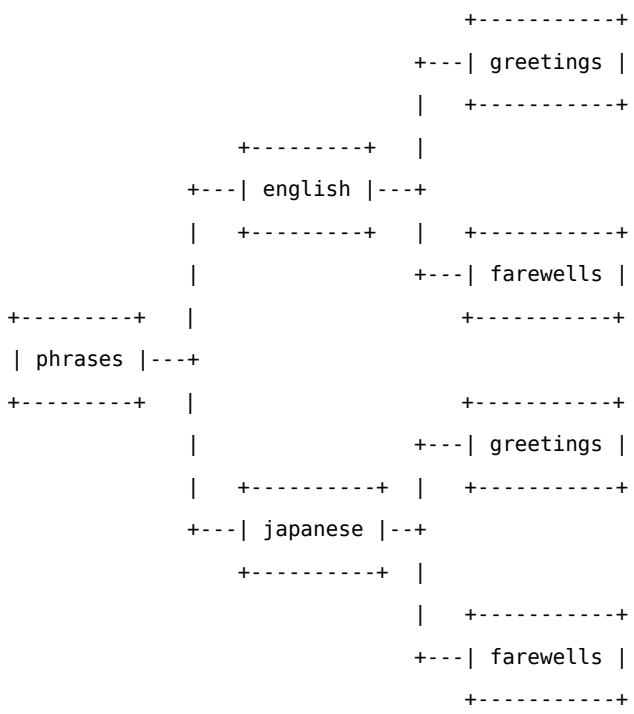
基本的な用語: クレートとモジュール

Rust はモジュールシステムに関連する「クレート」(crate) と「モジュール」(module) という 2 つの用語を明確に区別しています。クレートは他の言語における「ライブラリ」や「パッケージ」と同じ意味です。このことから Rust のパッケージマネジメントツールの名前を「Cargo」としています。(訳注: crate とは枠箱のことであり、cargo は船荷を指します) Cargo を使ってクレートを出荷し他のユーザに公開するわけです。クレートは実行形式かライブラリをプロジェクトに応じて作成できます。

各クレートは自身のコードが入っている **ルートモジュール** (root module) を暗黙的に持っています。そしてルートモジュール下にはサブモジュールの木が定義できます。モジュールによりクレート内でコードを分割できます。

例として、 *phrases* クレートを作ってみます。これに異なる言語で幾つかのフレーズを入れます。問題

の単純さを保つために、2 種類のフレーズ「greetings」と「farewells」のみとし、これらフレーズを表すための 2 つの言語として英語と日本語を使うことにします。モジュールのレイアウトは以下のようになります。



この例において、 phrases がクレートの名前で、それ以外の全てはモジュールです。それらが木の形をしており、クレートの**ルート** から枝分かれしていることが分かります。そして木のルートは phrases それ自身です。

ここまでで計画は立ちましたから、これらモジュールをコードで定義しましょう。始めるために、Cargo で新しいクレートを生成します。

```
$ cargo new phrases
$ cd phrases
```

聡明な読者ならご記憶かと思いますが Cargo が単純なプロジェクトを生成してくれます。

```
$ tree .
.
├── Cargo.toml
└── src
    └── lib.rs

1 directory, 2 files
```

src/lib.rs はクレートのルートであり、先程の図における phrases に相当します。

モジュールを定義する

それぞれのモジュールを定義するために、 mod キーワードを使います。 src/lib.rs を以下のようにしましょう。

```
mod english {
    mod greetings {
    }

    mod farewells {
    }
}

mod japanese {
    mod greetings {
    }

    mod farewells {
    }
}
```

mod キーワードの後、モジュールの名前を与えます。モジュール名は Rust の他の識別子の慣例に従います。つまり lower_snake_case です。各モジュールの内容は波括弧 ({}) の中に書きます。

与えられた mod 内で、サブ mod を定義することができます。サブモジュールは 2 重コロン (::) 記

法で参照できます。ネストされた 4 つのモジュールは `english::greetings` 、 `english::farewells` 、`japanese::greetings` 、そして `japanese::farewells` です。これらサブモジュールは親モジュール配下の名前空間になるため、名前は競合しません。つまり `english::greetings` と `japanese::greetings` は例え両名が `greetings` であったとしても、明確に区別されます。

このクレートは `main()` 関数を持たず、 `lib.rs` と名付けられているため、Cargo はこのクレートをライブラリとしてビルドします。

```
$ cargo build
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
$ ls target/debug
build  deps  examples  libphrases-a7448e02a0468eaa.rlib  native
```

`libphrases-hash.rlib` はコンパイルされたクレートです。このクレートを他のクレートから使う方法を見る前に、複数のファイルに分割してみましょう。

複数のファイルによるクレート

各クレートがただ 1 つのファイルからなるのであれば、これらファイルは非常に大きくなってしまうでしょう。クレートを複数のファイルに分けた方が楽になるため、Rust は 2 つの方法でこれをサポートしています。

以下のようなモジュールを宣言する代わりに、

```
mod english {
    // モジュールの内容はここに
}
```

以下のようなモジュールが宣言できます。

```
mod english;
```

こうすると、Rust は `english.rs` ファイルか、`english/mod.rs` ファイルのどちらかにモジュールの内容があるだろうと予想します。

それらのファイルの中でモジュールの再宣言を行う必要がないことに気をつけて下さい。先の `mod` 宣言にてそれは済んでいます。

これら 2 つのテクニックを用いて、クレートを 2 つのディレクトリと 7 つのファイルに分解できます。

```
$ tree .
.
├── Cargo.lock
├── Cargo.toml
├── src
│   ├── english
│   │   ├── farewells.rs
│   │   ├── greetings.rs
│   │   └── mod.rs
│   ├── japanese
│   │   ├── farewells.rs
│   │   ├── greetings.rs
│   │   └── mod.rs
│   └── lib.rs
└── target
    └── debug
        ├── build
        ├── deps
        ├── examples
        ├── libphrases-a7448e02a0468eaa.rlib
        └── native
```

`src/lib.rs` はクレートのルートで、以下のようになっています。

```
mod english;
mod japanese;
```

これら 2 つの宣言は Rust へ書き手の好みに合わせて `src/english.rs` と `src/japanese.rs` 、または `src/english/mod.rs` と `src/japanese/mod.rs` のどちらかを見よと伝えています。今回の場合、サブモジュールがあるため私たちは後者を選択しました。`src/english/mod.rs` と `src/japanese/mod.rs` は両方とも以下のようになっています。

```
mod greetings;
mod farewells;
```

繰り返すと、これら宣言は Rust へ src/english/greetings.rs 、src/english/farewells.rs 、src/japanese/greetings.rs 、src/japanese/farewells.rs 、または src/english/greetings/mod.rs 、src/english/farewells/mod.rs 、src/japanese/greetings/mod.rs 、src/japanese/farewells/mod.rs のどちらかを見よと伝えています。これらサブモジュールは自身配下のサブモジュールを持たないため、私たちは src/english/greetings.rs 、 src/english/farewells.rs 、 src/japanese/greetings.rs 、src/japanese/farewells.rs を選びました。ヒュー！

src/english/greetings.rs 、 src/english/farewells.rs 、 src/japanese/greetings.rs 、src/japanese/farewells.rs の中身は現在全て空です。幾つか関数を追加しましょう。

src/english/greetings.rs に以下を入力します。

```
fn hello() -> String {
    ”Hello!”.to_string()
}
```

src/english/farewells.rs に以下を入力します。

```
fn goodbye() -> String {
    ”Goodbye.”.to_string()
}
```

src/japanese/greetings.rs に以下を入力します。

```
fn hello() -> String {
    ” こんにちは”.to_string()
}
```

勿論、この web ページからコピー&ペーストしたり、他の何かをタイプしても構いません。モジュールシステムを学ぶのに「konnichiwa」と入力するのは重要なことではありません。

src/japanese/farewells.rs に以下を入力します。

```
fn goodbye() -> String {
    ”さようなら”.to_string()
}
```

（英語だと「Sayōnara」と表記するようです、御参考まで。）

ここまででクレートに幾つかの機能が実装されました、それでは他のクレートから使ってみましょう。

外部クレートのインポート

前節でライブラリクレートができました。インポートしこのライブラリを用いた実行形式クレートを作成しましょう。

`src/main.rs` を作成し配置します。（このコンパイルはまだ通りません）

```
extern crate phrases;

fn main() {
    println!(”Hello in English: {}”, phrases::english::greetings::hello());
    println!(”Goodbye in English: {}”, phrases::english::farewells::goodbye());

    println!(”Hello in Japanese: {}”, phrases::japanese::greetings::hello());
    println!(”Goodbye in Japanese: {}”, phrases::japanese::farewells::goodbye());
}
```

`extern crate` 宣言は Rust にコンパイルして `phrases` クレートをリンクせよと伝えます。するとこのクレート内で `phrases` モジュールが使えます。先に述べていたように、2 重コロンでサブモジュールとその中の関数を参照できます。

（注意: Rust の識別子として適切でない「like-this」のような、名前の中にダッシュが入ったクレートをインポートする場合、ダッシュがアンダースコアへ変換されるため `extern crate like_this;` と記述します。）

また、Cargo は `src/main.rs` がライブラリクレートではなくバイナリクレートのルートだと仮定します。パッケージには今 2 つのクレートがあります。`src/lib.rs` と `src/main.rs` です。ほとんどの機能をライブラリクレート内に置き、実行形式クレートから利用するのがよくあるパターンです。この方法

なら他のプログラムがライブラリクレートを使うこともできますし、素敵な関心の分離 (separation of concerns) にもなります。

けれどこのままではまだ動作しません。以下に似た 4 つのエラーが発生します。

```
$ cargo build
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
src/main.rs:4:38: 4:72 error: function `hello` is private
src/main.rs:4     println!("Hello in English: {}", phrases::english::greetings::hello(
   ));
                                                    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
note: in expansion of format_args!
<std macros>:2:25: 2:58 note: expansion site
<std macros>:1:1: 2:62 note: in expansion of print!
<std macros>:3:1: 3:54 note: expansion site
<std macros>:1:1: 3:58 note: in expansion of println!
phrases/src/main.rs:4:5: 4:76 note: expansion site
```

デフォルトでは、Rust における全てがプライベートです。それではもっと詳しく説明しましょう。

パブリックなインターフェースのエクスポート

Rust はインターフェースのパブリックである部分をきっちりと管理します。そのためプライベートがデフォルトです。パブリックにするためには `pub` キーワードを使います。まずは `english` モジュールに焦点を当てたいので、 `src/main.rs` をこれだけに減らしましょう。

```
extern crate phrases;

fn main() {
    println!("Hello in English: {}", phrases::english::greetings::hello());
    println!("Goodbye in English: {}", phrases::english::farewells::goodbye());
}
```

`src/lib.rs` 内にて、 `english` モジュールの宣言に `pub` を加えましょう。

```
pub mod english;
mod japanese;
```

また src/english/mod.rs にて、両方とも pub にしましょう。

```
pub mod greetings;
pub mod farewells;
```

src/english/greetings.rs にて、 fn の宣言に pub を加えましょう。

```
pub fn hello() -> String {
    "Hello!".to_string()
}
```

また src/english/farewells.rs にもです。

```
pub fn goodbye() -> String {
    "Goodbye.".to_string()
}
```

これでクレートはコンパイルできますが、 japanese 関数が使われていないという旨の警告が発生します。

```
$ cargo run
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
src/japanese/greetings.rs:1:1: 3:2 warning: function is never used: `hello`, #[warn(de
   ad_code)] on by default
src/japanese/greetings.rs:1 fn hello() -> String {
src/japanese/greetings.rs:2     "こんにちは".to_string()
src/japanese/greetings.rs:3 }
src/japanese/farewells.rs:1:1: 3:2 warning: function is never used: `goodbye`, #[warn(
   dead_code)] on by default
src/japanese/farewells.rs:1 fn goodbye() -> String {
src/japanese/farewells.rs:2     "さようなら".to_string()
src/japanese/farewells.rs:3 }
```

```
     Running `target/debug/phrases`
Hello in English: Hello!
Goodbye in English: Goodbye.
```

pub は struct やそのメンバのフィールドにも使えます。Rust の安全性に対する傾向に合わせ、単に struct をパブリックにしてもそのメンバまでは自動的にパブリックになりません。個々のフィールドに pub を付ける必要があります。

関数がパブリックになり、呼び出せるようになりました。素晴らしい！けれども phrases::english::greetings::hello() を打つのは非常に長くて退屈ですね。Rust にはもう 1 つ、現在のスコープに名前をインポートするキーワードがあるので、それを使えば短い名前で参照できます。では use について説明しましょう。

use でモジュールをインポートする

Rust には use キーワードがあり、ローカルスコープの中に名前をインポートできます。src/main.rs を以下のように変えてみましょう。

```
extern crate phrases;

use phrases::english::greetings;
use phrases::english::farewells;

fn main() {
    println!("Hello in English: {}", greetings::hello());
    println!("Goodbye in English: {}", farewells::goodbye());
}
```

2 つの use の行はローカルスコープの中に各モジュールをインポートしているため、とても短い名前で関数を参照できます。慣習では関数をインポートする場合、関数を直接インポートするよりもモジュール単位でするのがベストプラクティスだと考えられています。言い換えれば、こうすることもできる わけです。

```
extern crate phrases;

use phrases::english::greetings::hello;
```

```
use phrases::english::farewells::goodbye;

fn main() {
    println!("Hello in English: {}", hello());
    println!("Goodbye in English: {}", goodbye());
}
```

しかしこれは慣用的ではありません。名前の競合を引き起こす可能性が非常に高まるからです。この短いプログラムだと大したことではありませんが、長くなるにつれ問題になります。名前が競合するとコンパイルエラーになります。例えば、japanese 関数をパブリックにして、以下を試してみます。

```
extern crate phrases;

use phrases::english::greetings::hello;
use phrases::japanese::greetings::hello;

fn main() {
    println!("Hello in English: {}", hello());
    println!("Hello in Japanese: {}", hello());
}
```

Rust はコンパイル時にエラーを起こします。

```
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
src/main.rs:4:5: 4:40 error: a value named `hello` has already been imported in this m
   odule [E0252]
src/main.rs:4 use phrases::japanese::greetings::hello;
                  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
error: aborting due to previous error
Could not compile `phrases`.
```

同じモジュールから複数の名前をインポートする場合、二度同じ文字を打つ必要はありません。以下の代わりに、

```
use phrases::english::greetings;
use phrases::english::farewells;
```

このショートカットが使えます。

```
use phrases::english::{greetings, farewells};
```

pub use による再エクスポート

use は識別子を短くするためだけに用いるのではありません。他のモジュール内の関数を再エクスポートするためにクレートの中で使うこともできます。これを使って内部のコード構成そのままではない外部向けインターフェースを提供できます。

例を見てみましょう。 src/main.rs を以下のように変更します。

```
extern crate phrases;

use phrases::english::{greetings,farewells};
use phrases::japanese;

fn main() {
    println!("Hello in English: {}", greetings::hello());
    println!("Goodbye in English: {}", farewells::goodbye());

    println!("Hello in Japanese: {}", japanese::hello());
    println!("Goodbye in Japanese: {}", japanese::goodbye());
}
```

そして、 src/lib.rs の japanese モジュールをパブリックに変更します。

```
pub mod english;
pub mod japanese;
```

続いて、2 つの関数をパブリックにします。始めに src/japanese/greetings.rs を、

```
pub fn hello() -> String {
    "こんにちは".to_string()
}
```

そして src/japanese/farewells.rs を、

```
pub fn goodbye() -> String {
    "さようなら".to_string()
}
```

最後に、 src/japanese/mod.rs を以下のように変更します。

```
pub use self::greetings::hello;
pub use self::farewells::goodbye;

mod greetings;
mod farewells;
```

pub use 宣言は関数をモジュール階層 phrases::japanese のスコープへ持ち込みます。japanese モジュールの中で pub use したため、phrases::japanese::greetings::hello() と phrases::japanese::farewells::goodbye() にコードがあるのにも関わらず、phrases::japanese::hello() 関数と phrases::japanese::goodbye() 関数が使えるようになります。内部の構成で外部向けのインターフェースが決まるわけではありません。

pub use によって各関数を japanese スコープの中に持ち込めるようになりました。 greetings から現在のスコープへ全てをインクルードする代わりに、pub use self::greetings::* とすることでワイルドカード構文が使えます。

self とはなんでしょう? ええっと、デフォルトでは、 use 宣言はクレートのルートから始まる絶対パスです。 self は代わりに現在位置からの相対パスにします。 use にはもう 1 つ特別な形式があり、現在位置から 1 つ上へのアクセスに use super:: が使えます。多くのシェルにおけるカレントディレクトリと親ディレクトリの表示になぞらえ、. が self で、 .. が super であるという考え方を好む人もそれなりにいます。

use でなければ、パスは相対です。foo::bar() は私達のいる場所から相対的に foo の内側の関数を参照します。::foo::bar() のように :: から始まるのであれば、クレートのルートからの絶対パスで、先程とは異なる foo を参照します。

これはビルドして実行できます。

```
$ cargo run
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
     Running `target/debug/phrases`
Hello in English: Hello!
Goodbye in English: Goodbye.
Hello in Japanese: こんにちは
Goodbye in Japanese: さようなら
```

複合的なインポート

extern crate 及び use 文に対し、Rust は簡潔さと利便性を付加できる上級者向けオプションを幾つか提供しています。以下が例になります。

```
extern crate phrases as sayings;

use sayings::japanese::greetings as ja_greetings;
use sayings::japanese::farewells::*;
use sayings::english::{self, greetings as en_greetings, farewells as en_farewells};

fn main() {
    println!("Hello in English; {}", en_greetings::hello());
    println!("And in Japanese: {}", ja_greetings::hello());
    println!("Goodbye in English: {}", english::farewells::goodbye());
    println!("Again: {}", en_farewells::goodbye());
    println!("And in Japanese: {}", goodbye());
}
```

何が起きているでしょう?

第一に、インポートされているものを extern crate と use 双方でリネームしています。そのため「phrases」という名前のクレートであっても、ここでは「sayings」として参照することになります。同様に、始めの use 文はクレートから japanese::greetings を引き出していますが、単純な greetings ではなく ja_greetings で利用できるようにしています。これは異なる場所から同じ名前のアイテムをイ

ンポートする際、曖昧さを回避するのに役立ちます。

第二の use 文では sayings::japanese::farewells モジュールから全てのパブリックなシンボルを持ってくるためにスターグロブを使っています。ご覧の通り、最後にモジュールの修飾無しで日本語の goodbye 関数を参照できています。この類のグロブは慎重に使うべきです。スターグロブにはパブリックなシンボルをインポートするだけの機能しかありません、例えグロブするコードが同一のモジュール内であったとしてもです。

第三の use 文はもっと詳しい説明が必要です。これは 3 つの use 文を 1 つに圧縮するグロブ「中括弧展開」を用いています（以前に Linux のシェルスクリプトを書いたことがあるなら、この類の構文は慣れていることでしょう）。この文を展開した形式は以下のようになります。

```
use sayings::english;
use sayings::english::greetings as en_greetings;
use sayings::english::farewells as en_farewells;
```

ご覧の通り、波括弧は同一パス下にある幾つかのアイテムに対する use 文を圧縮します。また、この文脈における self はパスの 1 つ手前を参照します。（訳注: sayings::english::{self} の self が指す 1 つ手前は sayings::english です）注意: 波括弧はネストできず、スターグロブと混ぜるとこともできません。

const と static

Rust では const を用いることで定数を定義できます:

```
const N: i32 = 5;
```

let による束縛とは異なり、const を用いるときは型を明示する必要があります。

定数はプログラム全体のライフタイムの間生きています。さらに言えば、Rust プログラム中で定数はメモリ中に固定のアドレスを持ちません。これは定数が利用されている時にそれらが効率的にインライン化されるためです。このため、同じ定数への参照が必ずしも同じアドレスを指しているとは保証されません。

static

Rust は「グローバル変数」と呼ばれる静的アイテムを提供します。「グローバル変数」は定数と似ていますが、静的アイテムは使用にあたってインライン化は行われません。これは、「グローバル変数」にはそれぞれに対しただひとつのインスタンスのみが存在することを意味し、メモリ上に固定の位置を持つことになります。

以下に例を示します:

```
static N: i32 = 5;
```

let による束縛とは異なり、static を用いるときは型を明示する必要があります。

静的アイテムはプログラム全体のライフタイムの間生きています。そのため定数に保存されている参照は static ライフタイムを持ちます:

```
static NAME: &'static str = ”Steve”;
```

ミュータビリティ

mut を利用することでミュータビリティを導入できます:

```
static mut N: i32 = 5;
```

この静的な変数 N はミュータブルであるため、別のスレッドから読まれている間に変更される可能性があり、メモリの不安全性の原因となります。そのため static mut な変数にアクセスを行うことは unsafe であり、 unsafe ブロック中で行う必要があります。

```
unsafe {
    N += 1;

    println!(”N: {}”, N);
}
```

さらに言えば、 `static` な変数に格納される値の型は `Sync` を実装しており、かつ `Drop` は実装していない必要があります。

初期化

`const` 、 `static` どちらも値に対してそれらが定数式でなければならないという要件があります。言い換えると、関数の呼び出しのような複雑なものや実行時の値を指定することはできないということです。

どちらを使うべきか

大抵の場合、`static` か `const` で選ぶときは `const` を選ぶと良いでしょう。定数を定義したい時に、そのメモリロケーションが固定であることを必要とする場面は珍しく、また `const` を用いることで定数伝播によってあなたのクレートだけでなく、それを利用するクレートでも最適化が行われます。

アトリビュート

Rust では以下のように「アトリビュート」によって宣言を修飾することができます。

```
#[test]
```

または以下のように:

```
#![test]
```

２つの違いは `!` に有ります、 `!` はアトリビュートが適用されるものを変更します:

```
#[foo]
struct Foo;

mod bar {
    #![bar]
}
```

`#[foo]` アトリビュートは次のアイテムに適用され、この場合は `struct` 宣言に適用されます。 `#![bar]`

アトリビュートは `#![bar]` アトリビュートを囲んでいるアイテムに適用され、この場合は `mod` 宣言に適用されます。その他の点については同じであり、どちらも適用されたアイテムの意味を変化させます。

例を挙げると、たとえば以下の様な関数では:

```
#[test]
fn check() {
    assert_eq!(2, 1 + 1);
}
```

この関数は `#[test]` によってマークされており、これはテストを走らせた時に実行されるという特別な意味になります。通常通りにコンパイルをした場合は、コンパイル結果に含まれません。この関数は今やテスト関数なのです。

アトリビュートは以下のように、追加のデータを持つことができます:

```
#[inline(always)]
fn super_fast_fn() {
```

また、キーと値についても持つことができます:

```
#[cfg(target_os = "macos")]
mod macos_only {
```

Rust のアトリビュートは様々なことに利用されます。すべてのアトリビュートのリストはリファレンス[*39]に載っています。現在は、Rust コンパイラによって定義されている以外の独自のアトリビュートを作成することは許可されていません。

type エイリアス

`type` キーワードを用いることで他の型へのエイリアスを宣言することができます:

```
type Name = String;
```

このようにすると定義した型を実際の型であるかのように利用することができます:

[*39] http://doc.rust-lang.org/reference.html#attributes

```
type Name = String;

let x: Name = "Hello".to_string();
```

しかしながら、これはあくまで **エイリアス**であって、新しい型ではありません。言い換えると、Rust は強い型付け言語であるため、異なる型同士の比較が失敗することを期待するでしょう。例えば:

```
let x: i32 = 5;
let y: i64 = 5;

if x == y {
   // ...
}
```

このようなコードは以下のエラーを発生させます:

```
error: mismatched types:
 expected `i32`,
    found `i64`
(expected i32,
    found i64) [E0308]
     if x == y {
             ^
```

一方で、エイリアス用いた場合は:

```
type Num = i32;

let x: i32 = 5;
let y: Num = 5;

if x == y {
   // ...
}
```

このコードはエラーを起こすこと無くコンパイルを通ります。 Num 型の値は i32 型の値とすべての面において等価です。本当に新しい型がほしい時はタプル構造体を使うことができます。

また、エイリアスをジェネリクスと共に利用する事もできます:

```
use std::result;

enum ConcreteError {
    Foo,
    Bar,
}

type Result<T> = result::Result<T, ConcreteError>;
```

このようにすると Result 型の Result<T, E> の E として常に ConcreteError を持っている特殊化されたバージョンが定義されます。このような方法は標準ライブラリで細かく分類されたエラーを定義するために頻繁に使われています。一例を上げると io::Result[*40] がそれに当たります。

型間のキャスト

Rust は安全性に焦点を合わせており、異なる型の間を互いにキャストするために二つの異なる方法を提供しています。一つは as であり、これは安全なキャストに使われます。 逆に transmute は任意のキャストに使え、Rust における最も危険なフィーチャの一つです!

型強制

型強制は暗黙に行われ、それ自体に構文はありませんが、as で書くこともできます。

型強制が現れるのは、 let ・ const ・ static 文、関数呼び出しの引数、構造体初期化の際のフィールド値、そして関数の結果です。

一番よくある型強制は、参照からミュータビリティを取り除くものです。

- &mut T から &T へ

[*40] http://doc.rust-lang.org/std/io/type.Result.html

似たような変換としては、 生ポインタ[*41]からミュータビリティを取り除くものがあります。

- `*mut T` から `*const T` へ

参照も同様に、生ポインタへ型強制できます。

- `&T` から `*const T` へ
- `&mut T` から `*mut T` へ

`Deref`[*42] によって、カスタマイズされた型強制が定義されることもあります。

型強制は推移的です。

as

`as` というキーワードは安全なキャストを行います。

```
let x: i32 = 5;

let y = x as i64;
```

安全なキャストは大きく三つに分類されます。明示的型強制、数値型間のキャスト、そして、ポインタキャストです。

キャストは推移的ではありません。 `e as U1 as U2` が正しい式であったとしても、 `e as U2` が必ずしも正しいとは限らないのです。 (実際、この式が正しくなるのは、`U1` が `U2` へ型強制されるときのみです。)

明示的型強制

`e as U` というキャストは、 `e` が型 `T` を持ち、かつ `T` が `U` に 型強制されるとき、有効です。

数値キャスト

`e as U` というキャストは、以下のどの場合でも有効です。

[*41] raw-pointers.md
[*42] deref-coercions.md

- `e` が型 `T` を持ち、 `T` と `U` が数値型であるとき; *numeric-cast*
- `e` が C-like な列挙型であり (つまり、ヴァリアントがデータを持っておらず)、`U` が整数型であるとき; *enum-cast*
- `e` の型が `bool` か `char` であり、 `U` が整数型であるとき; *prim-int-cast*
- `e` が型 `u8` を持ち、 `U` が `char` であるとき; *u8-char-cast*

例えば、

```
let one = true as u8;
let at_sign = 64 as char;
let two_hundred = -56i8 as u8;
```

数値キャストのセマンティクスは以下の通りです。

- サイズの同じ二つの整数間のキャスト (例えば、i32 -> u32) は何も行いません
- サイズの大きい整数から小さい整数へのキャスト (例えば、u32 -> u8) では切り捨てを行います
- サイズの小さい整数から大きい整数へのキャスト (例えば、u8 -> u32) では、
  - 元の整数が符号無しならば、ゼロ拡張を行います
  - 元の整数が符号付きならば、符号拡張を行います
- 浮動小数点数から整数へのキャストでは、0 方向への丸めを行います
  - **注意: 現在、丸められた値がキャスト先の整数型で扱えない場合、このキャストは未定義動作を引き起こします。**[*43]これには Inf や NaN も含まれます。これはバグであり、修正される予定です。
- 整数から浮動小数点数へのキャストでは、必要に応じて丸めが行われて、その整数を表す浮動小数点数がつくられます (丸め戦略は指定されていません)
- f32 から f64 へのキャストは完全で精度は落ちません
- f64 から f32 へのキャストでは、表現できる最も近い値がつくられます (丸め戦略は指定されていません)
  - **注意: 現在、値が有限でありながら f32 で表現できる最大 (最小) の有限値より大きい (小さい) 場合、このキャストは未定義動作を引き起こします。**[*44]これはバグであり、修正される予定です。

[*43] https://github.com/rust-lang/rust/issues/10184
[*44] https://github.com/rust-lang/rust/issues/15536

ポインタキャスト

驚くかもしれませんが、いくつかの制約のもとで、生ポインタ[*45]と整数の間のキャストや、ポインタと他の型の間のキャストは安全です。安全でないのはポインタの参照外しだけなのです。

```
let a = 300 as *const char; // 300 番地へのポインタ
let b = a as u32;
```

`e as U` が正しいポインタキャストであるのは、以下の場合です。

- `e` が型 `*T` を持ち、 `U` が `*U_0` であり、 `U_0: Sized` または `unsize_kind(T) == unsize_kind(U_0)` である場合; *ptr-ptr-cast*
- `e` が型 `*T` を持ち、 `U` が数値型で、`T: Sized` である場合; *ptr-addr-cast*
- `e` が整数、`U` が `*U_0` であり、`U_0: Sized` である場合; *addr-ptr-cast*
- `e` が型 `&[T; n]` を持ち、 `U` が`*const T` である場合; *array-ptr-cast*
- `e` が関数ポインタ型であり、 `U` が `*T` であって、`T: Sized` の場合; *fptr-ptr-cast*
- `e` が関数ポインタ型であり、 `U` が整数型である場合; *fptr-addr-cast*

transmute

`as` は安全なキャストしか許さず、例えば 4 つのバイト値を `u32` へキャストすることはできません。

```
let a = [0u8, 0u8, 0u8, 0u8];

let b = a as u32; // 4 つの 8 で 32 になる
```

これは以下のようなメッセージがでて、エラーになります。

[*45] raw-pointers.md

```
error: non-scalar cast: `[u8; 4]` as `u32`
let b = a as u32; // 4 つの 8 で 32 になる
        ^~~~~~~~
```

これは「non-scalar cast」であり、複数の値、つまり配列の 4 つの要素、があることが原因です。この種類のキャストはとても危険です。なぜなら、複数の裏に隠れた構造がどう実装されているかについて仮定をおいているからです。そのためもっと危険なものが必要になります。

`transmute` 関数は コンパイラ intrinsic によって提供されており、やることはとてもシンプルながら、とても恐ろしいです。この関数は、Rust に対し、ある型の値を他の型であるかのように扱うように伝えます。これは型検査システムに関係なく行われ、完全に使用者頼みです。

先ほどの例では、4 つの `u8` からなる配列が ちゃんと `u32` を表していることを知った上で、キャストを行おうとしました。これは、`as` の代わりに `transmute` を使うことで、次のように書けます。

```
use std::mem;

unsafe {
    let a = [0u8, 0u8, 0u8, 0u8];

    let b = mem::transmute::<[u8; 4], u32>(a);
}
```

コンパイルを成功させるために、この操作は `unsafe` ブロックでくるんであります。 技術的には、`mem::transmute` の呼び出しのみをブロックに入れればいいのですが、今回はどこを見ればよいかわかるよう、関連するもの全部を囲んでいます。この例では `a` に関する詳細も重要であるため、ブロックにいれてあります。ただ、文脈が離れすぎているときは、こう書かないこともあるでしょう。そういうときは、コード全体を `unsafe` でくるむことは良い考えではないのです。

`transmute` はほとんどチェックを行わないのですが、最低限、型同士が同じサイズかの確認はします。そのため、次の例はエラーになります。

```
use std::mem;

unsafe {
    let a = [0u8, 0u8, 0u8, 0u8];
```

```
    let b = mem::transmute::<[u8; 4], u64>(a);
}
```

エラーメッセージはこうです。

```
error: transmute called with differently sized types: [u8; 4] (32 bits) to u64
(64 bits)
```

ただそれ以外に関しては、自己責任です!

関連型

関連型は、Rust 型システムの強力な部分です。関連型は、「型族」という概念と関連があり、言い換えると、複数の型をグループ化するものです。この説明はすこし抽象的なので、実際の例を見ていきましょう。 例えば、`Graph` トレイトを定義したいとしましょう、このときジェネリックになる 2 つの型: 頂点の型、辺の型 が存在します。 そのため、以下のように `Graph<N, E>` と書きたくなるでしょう:

```
trait Graph<N, E> {
    fn has_edge(&self, &N, &N) -> bool;
    fn edges(&self, &N) -> Vec<E>;
    // etc
}
```

たしかに上のようなコードは動作しますが、この `Graph` の定義は少し扱いづらいです。 たとえば、任意の `Graph` を引数に取る関数は、 さらに 頂点 `N` と辺 `E` の型についてもジェネリックになる必要があります:

```
fn distance<N, E, G: Graph<N, E>>(graph: &G, start: &N, end: &N) -> u32 { ... }
```

この距離を計算する関数 distance は、辺の型に関わらず動作します、そのためシグネチャに含まれる `E` に関連する部分は邪魔になります。

本当に表現したいのは、それぞれの `Graph` は、辺 `E` と頂点 `N` で構成されていることです。それは、以下のように関連型を用いて表現できます:

```
trait Graph {
    type N;
    type E;

    fn has_edge(&self, &Self::N, &Self::N) -> bool;
    fn edges(&self, &Self::N) -> Vec<Self::E>;
    // etc
}
```

こうすると、使う側では、個々の Graph をより抽象的なものとして扱えます:

```
fn distance<G: Graph>(graph: &G, start: &G::N, end: &G::N) -> u32 { ... }
```

ここでは、頂点 E 型を扱わずに済んでいます！

もっと詳しく見ていきましょう。

関連型を定義する

早速、Graph トレイトを定義しましょう。以下がその定義です:

```
trait Graph {
    type N;
    type E;

    fn has_edge(&self, &Self::N, &Self::N) -> bool;
    fn edges(&self, &Self::N) -> Vec<Self::E>;
}
```

非常にシンプルですね。関連型には type キーワードを使い、そしてトレイトの本体にある関数で利用します。

これらの type 宣言は、関数で利用できるものと同じものが全て利用できます。たとえば、頂点を表示するため N 型には Display を実装してほしいなら、以下のように指定できます:

```
use std::fmt;

trait Graph {
    type N: fmt::Display;
    type E;

    fn has_edge(&self, &Self::N, &Self::N) -> bool;
    fn edges(&self, &Self::N) -> Vec<Self::E>;
}
```

関連型を実装する

通常のトレイトと同様に、関連型を使っているトレイトは実装するために `impl` を利用します。 以下は、シンプルな Graph の実装例です:

```
struct Node;

struct Edge;

struct MyGraph;

impl Graph for MyGraph {
    type N = Node;
    type E = Edge;

    fn has_edge(&self, n1: &Node, n2: &Node) -> bool {
        true
    }

    fn edges(&self, n: &Node) -> Vec<Edge> {
        Vec::new()
    }
}
```

この、いささか単純過ぎる実装では、常に `true` と空の `Vec<Edge>` を返します。しかし、関連型をどう定義したらよいのかを教えてくれます。まず、はじめに 3 つの `struct` が必要です。グラフのためにひとつ、頂点のためにひとつ、辺のためにひとつです。もし異なる型を利用するのが適切ならば、そうしても構いません。今回はこの 3 つの `struct` を用います。

次は `impl` の行です。これは他のトレイトを実装するときと同様です。

そして、`=` を関連型を定義するために利用します。トレイトが利用する名前は `=` の左側にある名前で、実装に用いる具体的な型は右側にあるものになります。最後に、具体的な型を関数の宣言に利用します。

関連型を伴うトレイト

すこし触れておきたい構文のひとつに、トレイトオブジェクトがあります。もし、トレイトオブジェクトを以下のように関連型を持つトレイトから作成しようとした場合:

```
let graph = MyGraph;
let obj = Box::new(graph) as Box<Graph>;
```

以下の様なエラーが発生します:

```
error: the value of the associated type `E` (from the trait `main::Graph`) must
be specified [E0191]
let obj = Box::new(graph) as Box<Graph>;
          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
24:44 error: the value of the associated type `N` (from the trait
`main::Graph`) must be specified [E0191]
let obj = Box::new(graph) as Box<Graph>;
          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

上のようにしてトレイトオブジェクトを作ることはできません。なぜなら関連型について知らないからです。 代わりに以下のように書けます:

```
let graph = MyGraph;
let obj = Box::new(graph) as Box<Graph<N=Node, E=Edge>>;
```

`N=Node` 構文を用いて型パラメータ `N` に対して具体的な型 `Node` を指定できます。 `E=Edge` についても同様です。もしこの制約を指定しなかった場合、このトレイトオブジェクトに対してどの `impl` がマッチす

るのか定まりません。

サイズ不定型

ほとんどの型はコンパイル時に知れる、バイト数で測った、サイズがあります。たとえば、 `i32` 型は、32 ビット (4 バイト) というサイズです。しかしながら、表現のためには便利であってもサイズが定まっていない型が存在します。そのような型を 「サイズ不定」または「動的サイズ」型と呼びます。一例を上げると `[T]` 型は 一定のサイズの `T` のシーケンスを意味していますが、その要素数については規定されていないため、サイズは不定となります。

Rust はいくつかのそのような型を扱うことができますが、それらには以下の様な 3 つの制約が存在します:

1. サイズ不定型はポインタを通してのみ操作できます。 たとえば、`&[T]` は大丈夫ですが、 `[T]` はそうではありません。
2. 変数や引数は動的なサイズを持つことはできません。
3. `struct` の最後のフィールドのみ、動的なサイズを持つことが許されます。その他のフィールドはサイズが不定であってはなりません。また、Enum のバリアントはデータとして動的なサイズの型を持つ事はできません。

なぜこんなにややこしいのでしょうか？ これは、`[T]` はポインタを通してのみ操作可能であるため、もし言語がサイズ不定型をサポートしていなかった場合、以下のようなコードを書くことは不可能となります:

```
impl Foo for str {
```

また、以下の様なコードも:

```
impl<T> Foo for [T] {
```

このように書く代わりに、以下のように書くことになるでしょう:

```
impl Foo for &str {
```

このように書いたとすると、このコードは参照に対してのみ動作するようになり、他のポインタ型に対しては動作しないことになります。`impl for str` のように書くことで、すべてのポインタ、ユーザーの

定義した独自のスマートポインタ（いくつかの点についてバグがあるので、それをまずは直さなくてはなりませんが）もこの `impl` を利用可能になります。

?Sized

もし動的サイズ型を引数に取れるような関数を定義したい場合、特別な境界`?Sized` を利用できます:

```
struct Foo<T: ?Sized> {
    f: T,
}
```

`?` は 「T は `Sized` かもしれない」と読みます。 これは`?` が特別な境界であり、より小さいカインドとマッチするのではなく、より大きいカインドとマッチすることを意味しています。 これは、すべての `T` は暗黙的に `T : Sized` という制限がかけられていて、 `?` はその制限を解除するようなものです。

演算子とオーバーロード

Rust は制限された形式での演算子オーバーロードを提供しており、オーバーロード可能な演算子がいくつか存在します。型同士の間の演算子をサポートするためのトレイトが存在し、それらを実装することで演算子をオーバーロードできます。

たとえば、 `+` の演算子は `Add` トレイトでオーバーロードできます:

```
use std::ops::Add;

#[derive(Debug)]
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() {
    let p1 = Point { x: 1, y: 0 };
    let p2 = Point { x: 2, y: 3 };

    let p3 = p1 + p2;

    println!("{:?}", p3);
}
```

main 中で、２つの Point に対して + を使えます。 これは Point に対して Add<Output=Point> を実装したためです。

同じ方法でオーバーロード可能な演算子が多数あります。それらに対応したトレイトは std::ops[*46] モジュール内に存在します。全てのオーバーロード可能な演算子と対応するトレイトについては std::ops[*47] のドキュメントを読んで確認して下さい。

それらのトレイトの実装は、ある一つのパターンに従います。Add[*48] トレイトを詳しく見ていきましょう:

```
pub trait Add<RHS = Self> {
    type Output;

    fn add(self, rhs: RHS) -> Self::Output;
}
```

関連する３つの型が存在します: impl Add を実装するもの、デフォルトが Self の RHS、 そして Output です。 たとえば、式 let z = x + y においては x は Self 型 y は RHS、 z は Self::Output 型となります。

[*46] http://doc.rust-lang.org/std/ops/index.html
[*47] http://doc.rust-lang.org/std/ops/index.html
[*48] http://doc.rust-lang.org/std/ops/trait.Add.html

```
impl Add<i32> for Point {
    type Output = f64;

    fn add(self, rhs: i32) -> f64 {
        // i32 を Point に加算し f64 を返す
    }
}
```

上のコードによって以下の様に書けるようになります:

```
let p: Point = // ...
let x: f64 = p + 2i32;
```

オペレータトレイトをジェネリック構造体で使う

オペレータトレイトがどのように定義されているかを学びましたので、トレイトについての章の `HasArea` トレイトと `Square` 構造体をさらに一般的に定義できます:

```
use std::ops::Mul;

trait HasArea<T> {
    fn area(&self) -> T;
}

struct Square<T> {
    x: T,
    y: T,
    side: T,
}

impl<T> HasArea<T> for Square<T>
        where T: Mul<Output=T> + Copy {
    fn area(&self) -> T {
        self.side * self.side
```

```
    }
}

fn main() {
    let s = Square {
        x: 0.0f64,
        y: 0.0f64,
        side: 12.0f64,
    };

    println!("Area of s: {}", s.area());
}
```

HasArea と Square について、型パラメータ T を宣言し f64 で置換しました。 impl はさらに関連する修正を必要とします:

```
impl<T> HasArea<T> for Square<T>
        where T: Mul<Output=T> + Copy { ... }
```

area メソッドは辺を掛けることが可能なことを必要としています。そのため型 T が std::ops::Mul を実装していなければならないと宣言しています。 上で説明した Add と同様に、Mul は Output パラメータを取ります: 数値を掛け算した時に型が変わらないことを知っていますので、Output も T と設定します。 また T は、Rust が self.side を返り値にムーブするのを試みないようにコピーをサポートしている必要があります。

Deref による型強制

標準ライブラリは特別なトレイト Deref[*49] を提供します。 Deref は通常、参照外し演算子 * をオーバーロードするために利用されます。

```
use std::ops::Deref;

struct DerefExample<T> {
```

[*49] http://doc.rust-lang.org/std/ops/trait.Deref.html

```
    value: T,
}

impl<T> Deref for DerefExample<T> {
    type Target = T;

    fn deref(&self) -> &T {
        &self.value
    }
}

fn main() {
    let x = DerefExample { value: 'a' };
    assert_eq!('a', *x);
}
```

このように、 Deref はカスタマイズしたポインタ型を定義するのに便利です。一方で、Deref に関連する機能がもう一つ有ります: 「deref による型強制」です。 これは、Deref<Target=T> を実装している型 U があるときに、 &U が自動的に &T に型強制されるというルールです。 例えば:

```
fn foo(s: &str) {
    // 一瞬だけ文字列を借用します
}

// String は Deref<Target=str> を実装しています
let owned = "Hello".to_string();

// なので、以下のコードはきちんと動作します:
foo(&owned);
```

値の前にアンパサンド (&) をつけることによってその値への参照を取得することができます。なので、owned は String であり、 &owned は&String であり、 そして、 String が Deref<Target=str> を実装しているために、&String は foo() が要求している &str に型強制されます。

以上です! このルールは Rust が自動的に変換を行う唯一の箇所の一つです。これによって、多くの柔軟

性が手にはいります。 例えば Rc<T> は Deref<Target=T> を実装しているため、以下のコードは正しく動作します:

```
use std::rc::Rc;

fn foo(s: &str) {
      // 文字列を一瞬だけ借用します
}

// String は Deref<Target=str>を実装しています
let owned = "Hello".to_string();
let counted = Rc::new(owned);

// ゆえに、以下のコードは正しく動作します:
foo(&counted);
```

先ほどのコードとの変化は String を Rc<T> でラッピングした点ですが、 依然 Rc<String> を String が必要なところに渡すことができます。 foo のシグネチャは変化していませんが、どちらの型についても正しく動作します。この例は 2 つの変換を含んでいます: Rc<String> が String に変換され、次に String が &str に変換されます。Rust はこのような変換を型がマッチするまで必要なだけ繰り返します。

標準ライブラリに頻繁に見られるその他の実装は例えば以下の様なものが有ります:

```
fn foo(s: &[i32]) {
     // スライスを一瞬だけ借用します
}

// Vec<T> は Deref<Target=[T]> を実装しています
let owned = vec![1, 2, 3];

foo(&owned);
```

ベクタはスライスに Deref することができます。

Deref とメソッド呼び出し

Deref はメソッド呼び出し時にも自動的に呼びだされます。例えば以下の様なコードを見てみましょう:

```
struct Foo;

impl Foo {
    fn foo(&self) { println!("Foo"); }
}

let f = &&Foo;

f.foo();
```

f は &&Foo であり、 foo は &self を引数に取るにも関わらずこのコードは動作します。これは、以下が全て等価なことによります:

```
f.foo();
(&f).foo();
(&&f).foo();
(&&&&&&&&f).foo();
```

&&&&&&&&&&&&&&&&Foo 型の値は Foo で定義されているメソッドを呼び出すことができます。これは、コンパイラが自動的に必要なだけ * 演算子を補うことによります。そして * が補われることによって Deref が利用される事になります。

マクロ

Rust が提供している多くのコードの再利用や抽象化に利用できるツールを学びました。それらのコードの再利用のユニットは豊富な意味論的構造を持っています。例えば、関数は型シグネチャ、型パラメータはトレイト境界、オーバーロードされた関数はトレイトに所属していなければならない等です。

このような構造は Rust のコアの抽象化が強力なコンパイル時の正確性のチェックを持っているという事を意味しています。しかし、それは柔軟性の減少というコストを払っています。もし、視覚的に繰り

返しているコードのパターンを発見した時に、それらをジェネリックな関数やトレイトや、他の Rust のセマンティクスとして表現することが困難であると気がつくかもしれません。

マクロは構文レベルでの抽象化をすることを可能にします。マクロ呼出は「展開された」構文への短縮表現です。展開はコンパイルの初期段階、すべての静的なチェックが実行される前に行われます。その結果として、マクロは Rust のコアの抽象化では不可能な多くのパターンのコードの再利用を可能としています。

マクロベースのコードの欠点は、組み込みルールの少なさに由来するそのコードの理解のしづらさです。普通の関数と同じように、良いマクロはその実装について理解しなくても使うことができます。しかしながら、そのような良いマクロを設計するのは困難です！ 加えて、マクロコード中のコンパイルエラーは開発者が書いたソースレベルではなく、展開した結果のコードの中の問題について書かれているために、とても理解しづらいです。

これらの欠点はマクロを「最終手段となる機能」にしています。これは、マクロが良くないものだと言っているわけではありません、マクロは Rust の一部です、なぜならばマクロを使うことで簡潔になったり、適切な抽象化が可能になる場面がしばしば存在するからです。ただ、このトレードオフを頭に入れておいて欲しいのです。

マクロを定義する

`vec!` マクロを見たことがあるでしょう、ベクタを任意の要素で初期化するために使われていました。

```
let x: Vec<u32> = vec![1, 2, 3];
```

`vec!` は通常の関数として定義することはできません、なぜなら `vec!` は任意の個数の引数を取るためです。 しかし、 `vec!` を以下のコードの構文上の短縮形であると考えることができます:

```
let x: Vec<u32> = {
    let mut temp_vec = Vec::new();
    temp_vec.push(1);
    temp_vec.push(2);
    temp_vec.push(3);
    temp_vec
};
```

このような短縮形をマクロ: [*50]を用いることで実装することができます

```
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

ワオ！ たくさんの新しい構文が現れました！ 細かく見ていきましょう。

```
macro_rules! vec { ... }
```

これは、新しいマクロ `vec` を定義していることを意味しています、`vec` という関数を定義するときに `fn vec` と書くのと同じです。非公式ですが、実際には、マクロ名をエクスクラメーションマーク (!) と共に記述します、例えば: `vec!` のように示します。エクスクラメーションマークはマクロ呼び出しの構文の一部で、マクロと通常の関数の区別をつけるためのものです。

マッチング

マクロは、幾つかのパターンマッチのケースを利用したルールに従って定義されています、上のコード中では、以下の様なパターンが見られました:

```
( $( $x:expr ),* ) => { ... };
```

これは `match` 式の腕に似ていますが、Rust の構文木に対してコンパイル時にマッチします。セミコロンはケースの末尾でだけ使うことのでき、省略可能です。`=>` の左辺にある「パターン」は「マッチャ」として知られています。 マッチャは小さなマッチャ独自の構文[*51]を持っています。

[*50] `vec!` の libcollections における実際の実装と、ここで示したコードは効率性や再利用性のために異なります。

[*51] http://doc.rust-lang.org/reference.html#macros

マッチャ `$x:expr` は任意の Rust の式にマッチし、マッチした構文木を「メタ変数」 `$x` に束縛します。識別子 `expr` は「フラグメント指定子」です。全てのフラグメント指定子の一覧はこの章で後ほど紹介します。マッチャを `$(...),*` で囲むと 0 個以上のコンマで句切られた式にマッチします。

特別なマッチャ構文は別にして、マッチャ中に登場するその他の任意のトークンはそれ自身に正確にマッチする必要があります。例えば:

```
macro_rules! foo {
    (x => $e:expr) => (println!("mode X: {}", $e));
    (y => $e:expr) => (println!("mode Y: {}", $e));
}

fn main() {
    foo!(y => 3);
}
```

上のコードは以下の様な出力をします

```
mode Y: 3
```

また、以下のようなコードでは

```
foo!(z => 3);
```

以下の様なコンパイルエラーが発生します

```
error: no rules expected the token `z`
```

展開

マクロルールの右辺は大部分が通常の Rust の構文です。しかし、マッチャによってキャプチャされた構文を繋げる事ができます。最初に示した `vec!` の例を見てみましょう:

```
$(
    temp_vec.push($x);
)*
```

`$x` にマッチしたそれぞれの式はマクロ展開中に `push` 文を生成します。マクロ展開中の繰り返しはマッチャ中の繰り返しと足並みを揃えて実行されます (これについてはもう少し説明します)。

`$x` が既に式にマッチすると宣言されているために、`=>` の右辺では `:expr` を繰り返しません。また、区切りのコンマは繰り返し演算子の一部には含めません。そのかわり、繰り返しブロックをセミコロンを用いて閉じます。

そのほかの詳細としては: `vec!` マクロは *2* つの括弧のペアを右辺に含みます。それらの括弧はよく以下のように合せられます:

```
macro_rules! foo {
    () => {{
        ...
    }}
}
```

外側の括弧は `macro_rules!` 構文の一部です。事実、`()` や `[]` をかわりに使うことができます。括弧は単純に右辺を区切るために利用されています。

内側の括弧は展開結果の一部です。 `vec!` マクロは式を必要としているコンテキストで利用されていることを思いだしてください。複数の文や、 `let` 束縛を含む式を書きたいときにはブロックを利用します。もし、マクロが単一の式に展開されるときは、追加の括弧は必要ありません。

マクロが式を生成すると **宣言**した事はないという点に注意してください。事実、それはマクロを式として利用するまでは決定されません。注意深くすれば、複数のコンテキストで適切に展開されるマクロを書く事ができます。例えば、データ型の短縮形は、式としてもパターンとしても正しく動作します。

繰り返し

繰り返し演算子は以下の 2 つの重要なルールに従います:

1. `$(...)*` は繰り返しの一つの「レイヤ」上で動作し、レイヤが含んでいる `$name` について足並みを揃えて動作します。
2. それぞれの `$name` はマッチしたときと同じ個数の`$(...)*` の内側になければなりません。 もし更に多くの`$(...)*` の中に表われた際には適切に複製されます。

以下の複雑なマクロは一つ外の繰り返しのレベルから値を複製している例です:

```
macro_rules! o_O {
    (
        $(
            $x:expr; [ $( $y:expr ),* ]
        );*
    ) => {
        &[ $($( $x + $y ),*),* ]
    }
}

fn main() {
    let a: &[i32]
        = o_O!(10; [1, 2, 3];
               20; [4, 5, 6]);

    assert_eq!(a, [11, 12, 13, 24, 25, 26]);
}
```

上のコードはほとんどのマッチャの構文を利用しています。この例では 0 個以上にマッチする `$(...)*` を利用しています、1 つ以上にマッチさせたい場合は `$(...)+` を代わりに利用する事ができます。また、どちらも補助的に区切りを指定する事ができます。区切りには、`+` と `*` 以外の任意のトークンを指定することが可能です。

このシステムは: “Macro-by-Example[*52]” (PDF リンク) に基づいています。

健全性

いくつかの言語に組込まれているマクロは単純なテキストの置換を用いています、しかしこれは多くの問題を発生させます。例えば、以下の C 言語のプログラムは期待している `25` の代わりに `13` と出力します:

```
#define FIVE_TIMES(x) 5 * x
```

[*52] https://www.cs.indiana.edu/ftp/techreports/TR206.pdf

```
int main() {
    printf(”%d\n”, FIVE_TIMES(2 + 3));
    return 0;
}
```

展開した結果は `5 * 2 + 3` となり、乗算は加算よりも優先度が高くなります。もし C 言語のマクロを頻繁に利用しているなら、この問題を避けるためのイディオムを 5、6 個は知っているでしょう。Rust ではこのような問題を恐れる必要はありません。

```
macro_rules! five_times {
    ($x:expr) => (5 * $x);
}

fn main() {
    assert_eq!(25, five_times!(2 + 3));
}
```

メタ変数 `$x` は一つの式の頂点としてパースされ、構文木上の位置は置換されたあとも保存されます。

他のマクロシステムで良くみられる問題は、「変数のキャプチャ」です。以下の C 言語のマクロは GNU C 拡張[*53] を Rust の式のブロックをエミュレートするために利用しています。

```
#define LOG(msg) ({ \
    int state = get_log_state(); \
    if (state > 0) { \
        printf(”log(%d): %s\n”, state, msg); \
    } \
})
```

以下はこのマクロを利用したときにひどい事になる単純な利用例です:

```
const char *state = ”reticulating splines”;
LOG(state)
```

このコードは以下のように展開されます

[*53] https://gcc.gnu.org/onlinedocs/gcc/Statement-Exprs.html

```
const char *state = ”reticulating splines”;
{
    int state = get_log_state();
    if (state > 0) {
        printf(”log(%d): %s\n”, state, state);
    }
 }
```

2 番目の変数 `state` は 1 つめの `state` を隠してしまいます。この問題は、print 文が両方の変数を参照する必要があるために起こります。

Rust における同様のマクロは期待する通りの動作をします。

```
macro_rules! log {
    ($msg:expr) => {{
        let state: i32 = get_log_state();
        if state > 0 {
            println!(”log({}): {}”, state, $msg);
        }
    }};
}

fn main() {
    let state: &str = ”reticulating splines”;
    log!(state);
}
```

このマクロは Rust が健全なマクロシステム[*54]を持っているためです。それぞれのマクロ展開は分離された「構文コンテキスト」で行なわれ、それぞれの変数はその変数が導入された構文コンテキストでタグ付けされます。これは、 `main` 中の `state` がマクロの中の `state` とは異なる「色」で塗られているためにコンフリクトしないという風に考える事ができます。

この健全性のシステムはマクロが新しい束縛を呼出時に導入する事を制限します。以下のようなコードは動作しません:

[*54] https://en.wikipedia.org/wiki/Hygienic_macro

```
macro_rules! foo {
    () => (let x = 3);
}

fn main() {
    foo!();
    println!("{}", x);
}
```

呼出時に渡す事で正しい構文コンテキストでタグ付けされるように、代わりに変数名を呼出時に渡す必要があります。

```
macro_rules! foo {
    ($v:ident) => (let $v = 3);
}

fn main() {
    foo!(x);
    println!("{}", x);
}
```

このルールは `let` 束縛やループについても同様ですが、アイテム[*55]については適用されません。 そのため、以下のコードはコンパイルが通ります:

```
macro_rules! foo {
    () => (fn x() { });
}

fn main() {
    foo!();
    x();
}
```

[*55] http://doc.rust-lang.org/reference.html#items

再帰的マクロ

マクロの展開は、展開中のマクロ自身も含めたその他のマクロ呼出しを含んでいることが可能です。そのような再帰的なマクロは、以下の (単純化した)HTML の短縮形のような、木構造を持つ入力の処理に便利です:

```
macro_rules! write_html {
    ($w:expr, ) => (());

    ($w:expr, $e:tt) => (write!($w, "{}", $e));

    ($w:expr, $tag:ident [ $($inner:tt)* ] $($rest:tt)*) => {{
        write!($w, "<{}>", stringify!($tag));
        write_html!($w, $($inner)*);
        write!($w, "</{}>", stringify!($tag));
        write_html!($w, $($rest)*);
    }};
}

fn main() {
    use std::fmt::Write;
    let mut out = String::new();

    write_html!(&mut out,
        html[
            head[title["Macros guide"]]
            body[h1["Macros are the best!"]]
        ]);

    assert_eq!(out,
        "<html><head><title>Macros guide</title></head>\
         <body><h1>Macros are the best!</h1></body></html>");
}
```

マクロをデバッグする

マクロの展開結果を見るには、 `rustc –pretty expanded` を実行して下さい。出力結果はクレートの全体を表しています、そのため出力結果を再び `rustc` に与えることができます、そのようにすると時々、直接コンパイルした場合よりもより良いエラーメッセージを得ることができます。しかし、 `–pretty expanded` は同じ名前の変数 (構文コンテキストは異なる) が同じスコープに複数存在する場合、出力結果のコード自体は、元のコードと意味が変わってくる場合があります。そのようになってしまう場合、`–pretty expanded,hygiene` のようにすることで、構文コンテキストについて知ることができます。

`rustc` はマクロのデバッグを補助する 2 つの構文拡張を提供しています。今のところは、それらの構文は不安定であり、フィーチャーゲートを必要としています。

- `log_syntax!(...)` は与えられた引数をコンパイル時に標準入力に出力し、展開結果は何も生じません。

- `trace_macros!(true)` はマクロが展開されるたびにコンパイラがメッセージを出力するように設定できます、`trace_macros!(false)` を展開の終わりごろに用いることで、メッセージの出力をオフにできます。

構文的な要求

Rust のコードに展開されていないマクロが含まれていても、構文木 としてパースすることができます。このような特性はテキストエディタや、その他のコードを処理するツールにとって非常に便利です。また、このような特性は Rust のマクロシステムの設計にも影響を及ぼしています。

一つの影響としては、マクロ呼出をパースした時、マクロが以下のどれを意味しているかを判定する必要があります:

- 0 個以上のアイテム
- 0 個以上のメソッド
- 式
- 文
- パターン

ブロック中でのマクロ呼出は、幾つかのアイテムや、一つの式 / 文に対応します。Rust はこの曖昧性を判定するために単純なルールを利用します。アイテムに対応しているマクロ呼出は以下のどちらかでなければなりません

- 波括弧で区切られている 例: `foo! { ... }`
- セミコロンで終了している 例: `foo!(...);`

その他の展開前にパース可能である事による制約はマクロ呼出は正しい Rust トークンで構成されている必要があるというものです。そのうえ、括弧や、角カッコ、波括弧はマクロ呼出し中でバランスしてなければなりません。例えば: `foo!([)` は禁止されています。これによって Rust はマクロ呼出しがどこで終わっているかを知ることができます。

もっと厳密に言うと、マクロ呼出しの本体は「トークンの木」のシーケンスである必要があります。トークンの木は以下のいずれかの条件により再帰的に定義されています

- マッチャ、 `()` 、 `[]` または `{}` で囲まれたトークンの木、あるいは、
- その他の単一のトークン

マッチャ内部ではそれぞれのメタ変数はマッチする構文を指定する「フラグメント指定子」を持っています。

- `ident`: 識別子。 例: `x`; `foo`
- `path`: 修飾された名前。例: `T::SpecialA`
- `expr`: 式。 例: `2 + 2`; `if true { 1 } else { 2 }`; `f(42)`
- `ty`: 型。 例: `i32`; `Vec<(char, String)>`; `&T`
- `pat`: パターン。 例: `Some(t)`; `(17, 'a')`; `_`
- `stmt`: 単一の文。 例: `let x = 3`
- `block`: 波括弧で区切られた文のシーケンスと場合によっては式も付く。例: `{ log(error, "hi"); return 12 }`
- `item`: アイテム[*56]。 例: `fn foo() { }`; `struct Bar;`
- `meta`: アトリビュートで見られるような「メタアイテム」。 例: `cfg(target_os = "windows")`
- `tt`: 単一のトークンの木

またメタ変数の次のトークンについて以下のルールが存在します:

[*56] http://doc.rust-lang.org/reference.html#items

- expr 変数と stmt 変数は => , ; のどれか一つのみが次に現れます
- ty と path 変数は=> , = | ; : > [{ as where のどれか一つのみが次に現れます
- pat 変数は => , = | if in のどれか一つのみが次に現れます
- その他の変数は任意のトークンが次に現れます

これらのルールは既存のマクロを破壊すること無く Rust の構文を拡張するための自由度を与えます。

マクロシステムはパースの曖昧さについては何も対処しません。 例えば、$($i:ident)* $e:expr は常にパースが失敗します、なぜならパーサーは $i をパースするか、 $e をパースするかを選ぶことを強制されるためです。呼出構文を変更して識別可能なトークンを先頭につけることでこの問題は回避することができます。そのようにする場合、例えば $(I $i:ident)* E $e:expr のように書くことができます。

スコープとマクロのインポート/エクスポート

マクロはコンパイルの早い段階、名前解決が行われる前に展開されます。一つの悪い側面としては、言語中のその他の構造とは異なり、マクロではスコープが少し違って動作するということです。

マクロの定義と展開はクレートの字面上の順序どおりに単一の深さ優先探索で行われます。そのため、モジュールスコープで定義されたマクロは、 後続する子供の mod アイテムも含む、同じモジュール中のコードから見えます。

fn の本体の中やその他のモジュールのスコープでない箇所で定義されたマクロはそのアイテム中でしか見えません。

もし、モジュールが macro_use アトリビュートを持っていた場合、それらのマクロは子供の mod アイテムの後で、親モジュールからも見えます。 もし親モジュールが同様に macro_use アトリビュートを持っていた場合、親の親モジュールから親の mod アイテムが終わった後に見えます。その後についても同様です。

また、 macro_use アトリビュートは extern crate の上でも利用することができます。 そのようにした場合、macro_use アトリビュートは外部のクレートからどのマクロをロードするのかを指定します。以下がその例です:

```
#[macro_use(foo, bar)]
extern crate baz;
```

もしアトリビュートが単純に #[macro_use] という形で指定されていた場合、全てのマクロがロードされます。 もし、#[macro_use] が指定されていなかった場合、#[macro_export] アトリビュートとともに

定義されているマクロ以外は、どのマクロもロードされません。

クレートのマクロを出力にリンクさせずにロードするには、`#[no_link]` を利用して下さい。

一例としては:

```
macro_rules! m1 { () => (()) }

// ここで見えるのは: m1

mod foo {
    // ここで見えるのは: m1

    #[macro_export]
    macro_rules! m2 { () => (()) }

    // ここで見えるのは: m1、m2
}

// ここで見えるのは: m1

macro_rules! m3 { () => (()) }

// ここで見えるのは: m1、m3

#[macro_use]
mod bar {
    // ここで見えるのは: m1、m3

    macro_rules! m4 { () => (()) }

    // ここで見えるのは: m1、m3、m4
}

// ここで見えるのは: m1、m3、m4
```

ライブラリが `#[macro_use]` と共に外部のクレートをロードした場合、 `m2` だけがインポートされます。

Rust のリファレンスはマクロに関連するアトリビュートの一覧[*57]を掲載しています。

`$crate` 変数

さらなる困難はマクロが複数のクレートで利用された時に発生します。`mylib` が以下のように定義されているとしましょう

```
pub fn increment(x: u32) -> u32 {
    x + 1
}

#[macro_export]
macro_rules! inc_a {
    ($x:expr) => ( ::increment($x) )
}

#[macro_export]
macro_rules! inc_b {
    ($x:expr) => ( ::mylib::increment($x) )
}
```

`inc_a` は `mylib` の中でだけ動作します、かたや `inc_b` は `mylib` の外部でだけ動作します。さらにいえば、 `inc_b` はユーザーが `mylib` を異なる名前でインポートした際には動作しません。

Rust は (まだ) 健全なクレートの参照の仕組みを持っていません、しかし、この問題に対する簡単な対処方法を提供しています。`foo` というクレートからインポートされたマクロ中において、特別なマクロ変数 `$crate` は `::foo` に展開されます。対照的に、マクロが同じクレートの中で定義され利用された場合、`$crate` は何にも展開されません。これはつまり以下のように書けることを意味しています:

```
#[macro_export]
macro_rules! inc {
    ($x:expr) => ( $crate::increment($x) )
```

[*57] http://doc.rust-lang.org/reference.html#macro-related-attributes

```
}
```

これは、ライブラリの中でも外でも動作するマクロを定義しています。関数の名前は `::increment` または `::mylib::increment` に展開されます。

このシステムを簡潔で正しく保つために、`#[macro_use] extern crate ...` はクレートのルートにしか登場せず、 `mod` の中には現れません。

最難関部

入門のチャプタで再帰的なマクロについて言及しました、しかしそのチャプタでは詳細について話していませんでした。再帰的なマクロが便利な他の理由は、それぞれの再帰的な呼出はマクロに与えられた引数にたいしてパターンマッチを行える可能性を与えてくれることです。

極端な例としては、 望ましくはありませんが、Bitwise Cyclic Tag[*58] のオートマトンを Rust のマクロで実装する事が可能です。

```
macro_rules! bct {
    // cmd 0:  d ... => ...
    (0, $($ps:tt),* ; $_d:tt)
        => (bct!($($ps),*, 0 ; ));
    (0, $($ps:tt),* ; $_d:tt, $($ds:tt),*)
        => (bct!($($ps),*, 0 ; $($ds),*));

    // cmd 1p:  1 ... => 1 ... p
    (1, $p:tt, $($ps:tt),* ; 1)
        => (bct!($($ps),*, 1, $p ; 1, $p));
    (1, $p:tt, $($ps:tt),* ; 1, $($ds:tt),*)
        => (bct!($($ps),*, 1, $p ; 1, $($ds),*, $p));

    // cmd 1p:  0 ... => 0 ...
    (1, $p:tt, $($ps:tt),* ; $($ds:tt),*)
        => (bct!($($ps),*, 1, $p ; $($ds),*));
```

[*58] https://esolangs.org/wiki/Bitwise_Cyclic_Tag

```
    // 空のデータ文字列で停止します
    ( $($ps:tt),* ; )
        => (());
}
```

演習: マクロを使って上の `bct!` マクロの定義の重複している部分を減らしてみましょう。

よく見られるマクロ

以下は、Rust コード中でよく見られるマクロたちです。

panic!

このマクロは現在のスレッドをパニック状態にします。パニック時のメッセージを指定することができます。

```
panic!("oh no!");
```

vec!

`vec!` マクロはこの本のなかで使われてきましたので、すでに見たことがあるでしょう。 `vec!` マクロは `Vec<T>` を簡単に作成できます:

```
let v = vec![1, 2, 3, 4, 5];
```

また、値の繰り返しのベクタを作成することも可能です。たとえば、以下は 100 個の 0 を含むベクタの例です:

```
let v = vec![0; 100];
```

assert! と assert_eq!

この 2 つのマクロはテスト時に利用されています。 `assert!` は真偽値を引数に取ります。 `assert_eq!` は 2 つの等価性をチェックする値を引数に取ります。 `true` ならばパスし、 `false` だった場合 `panic!` を

起こします:

```
// Ok です！

assert!(true);
assert_eq!(5, 3 + 2);

// 駄目だぁ :(

assert!(5 < 3);
assert_eq!(5, 3);
```

try!

try! はエラーハンドリングのために利用されています。try! は Result<T, E> を返す何らかの物を引数に取り、もし Result<T, E> が Ok<T> だった場合 T を返し、そうでなく Err(E) だった場合はそれを return します。例えば以下のように利用します:

```
use std::fs::File;

fn foo() -> std::io::Result<()> {
    let f = try!(File::create("foo.txt"));

    Ok(())
}
```

このコードは以下のコードよりも綺麗です:

```
use std::fs::File;

fn foo() -> std::io::Result<()> {
    let f = File::create("foo.txt");

    let f = match f {
        Ok(t) => t,
```

```
        Err(e) => return Err(e),
    };

    Ok(())
}
```

unreachable!

このマクロはあるコードが絶対に実行されるべきでないと考えている時に利用します。

```
if false {
    unreachable!();
}
```

時々、コンパイラによって絶対に呼び出されるはずがないと考えているブランチを作成することになる時があります。そういった時には、このマクロを利用しましょう、そうすることでもし何か誤ってしまった時に、panic! で知ることができます。

```
let x: Option<i32> = None;

match x {
    Some(_) => unreachable!(),
    None => println!("I know x is None!"),
}
```

unimplemented!

unimplemented! マクロはもし関数の本体の実装はしていないが、型チェックだけは行いたいという時に利用します。このような状況の一つの例としては複数のメソッドを必要としているトレイトのメソッドの一つを実装しようと試みている時などです。残りのメソッドたちの実装に取り掛かれるようになるまで unimplemented! として定義しましょう。

手続きマクロ

もし Rust のマクロシステムでは必要としていることができない場合、コンパイラプラグインを代わりに書きたくなるでしょう。 コンパイラプラグインは `macro_rules!` マクロとくらべて、更に多くの作業が必要になり、インタフェースはかなり不安定であり、バグはさらに追跡が困難になります。引き換えに、任意のコードをコンパイラ中で実行できるという自由度を得ることができます。構文拡張プラグインがしばしば「手続きマクロ」と呼ばれるのはこのためです。

生ポインタ

Rust は標準ライブラリに異なるスマートポインタの型を幾つか用意していますが、更に特殊な型が 2 つあります。Rust の安全性の多くはコンパイル時のチェックに依るものですが、生ポインタを用いるとそういった保証が得られないため unsafe です。

`*const T` と `*mut T` は Rust において「生ポインタ」と呼ばれます。時々、ある種のライブラリを書く際に、あなたは何らかの理由で Rust が行う安全性の保証を避けなければならないこともあります。このようなケースでは、ユーザに安全なインターフェースを提供しつつ、ライブラリの実装に生ポインタを使用できます。例えば、`*` ポインタはエイリアスとして振る舞うこともできるので、所有権を共有する型を書くのに用いたり、スレッドセーフな共有メモリ型でさえも実装できます。(`Rc<T>` と `Arc<T>` 型は完全に Rust のみで実装されています)

以下は覚えておくべき生ポインタとその他のポインタ型との違いです。

- 有効なメモリを指していることが保証されないどころか、null でないことも保証されない (`Box` と `&` では保証される)
- `Box` とは異なり、自動的な後処理が一切行われないため、手動のリソース管理が必要
- plain-old-data であるため、Rust コンパイラは use-after-free のようなバグから保護できない
- `&` と異なり、ライフタイムの機能が無効化されるため、コンパイラはダングリングポインタを推論できない
- また、 `*const T` を直接介した変更は拒むが、それ以外のエイリアシングやミュータビリティに関する保証はない

基本

生ポインタを作成すること自体は絶対に安全です。

```
let x = 5;
let raw = &x as *const i32;

let mut y = 10;
let raw_mut = &mut y as *mut i32;
```

しかしながら参照外しは安全ではありません。以下は動作しないでしょう。

```
let x = 5;
let raw = &x as *const i32;

println!("raw points at {}", *raw);
```

このコードは以下のエラーが発生します。

```
error: dereference of raw pointer requires unsafe function or block [E0133]
     println!("raw points at {}", *raw);
                                  ^~~~
```

生ポインタを参照外しする時、ポインタが間違った場所を指していないことに対して責任を負うことになります。そういう時は、unsafe を付けなければなりません。

```
let x = 5;
let raw = &x as *const i32;

let points_at = unsafe { *raw };

println!("raw points at {}", points_at);
```

生ポインタの操作に関する詳細は、API ドキュメント[*59]を参照してください。

FFI

生ポインタは FFI を使う際に役立ちます。Rust の `*const T` と`*mut T` はそれぞれ C 言語の `const T*` と `T*` に似ているからです。これの使い方に関する詳細は、FFI の章 を参照してください。

参照と生ポインタ

実行時において、同じデータを指す生ポインタ `*` と参照は内部的に同一です。事実、 `unsafe` 外の安全なコードにおいて `&T` 参照は `*const T` 生ポインタへ暗黙的に型強制されますし、 `mut` の場合でも同様です。(これら型強制は、それぞれ `value as *const T` と `value as *mut T` のように、明示的に行うこともできます。)

逆に、`*const` から 参照 `&` へ遡るのは安全ではありません。`&T` は常に有効であるため、最低でも `*const T` は型 `T` の有効な実体を指さなければならないのです。その上、ポインタは参照のエイリアシングとミュータビリティの規則も満たす必要があります。コンパイラはあらゆる参照についてこれらの性質が真であると仮定しており、その生成方法に依らず適用するため、生ポインタからのいかなる変換も、参照先の値が上記の性質を満たすと表明していることになります。プログラマがこのことを保証**しなければならない** のです。

おすすめの変換の方法は以下のとおりです。

```
// 明示的キャスト
let i: u32 = 1;
let p_imm: *const u32 = &i as *const u32;

// 暗黙的キャスト
let mut m: u32 = 2;
let p_mut: *mut u32 = &mut m;

unsafe {
    let ref_imm: &u32 = &*p_imm;
```

[*59] http://doc.rust-lang.org/std/primitive.pointer.html

```
    let ref_mut: &mut u32 = &mut *p_mut;
}
```

&*x 参照外し方式は transmute を用いるよりも好ましいです。後者は必要以上に強力ですから、より用途が限定されている操作の方が間違って使いにくいでしょう。例えば、前者の方法は x がポインタである必要があります。(transmute とは異なります)

unsafe

Rust の主たる魅力は、プログラムの動作についての強力で静的な保証です。しかしながら、安全性検査は本来保守的なものです。すなわち、実際には安全なのに、そのことがコンパイラには検証できないプログラムがいくらか存在します。その類のプログラムを書くためには、制約を少し緩和するようコンパイラに対して伝えることが要ります。そのために、Rust には unsafe というキーワードがあります。unsafe を使ったコードは、普通のコードよりも制約が少なくなります。

まずシンタックスをみて、それからセマンティクスについて話しましょう。unsafe は 4 つの場面で使われます。1 つめは、関数がアンセーフであることを印付ける場合です。

```
unsafe fn danger_will_robinson() {
    // 恐ろしいもの
}
```

たとえば、FFI から呼び出されるすべての関数は unsafe で印付けることが必要です。unsafe の 2 つめの用途は、アンセーフブロックです。

```
unsafe {
    // 恐ろしいもの
}
```

3 つめは、アンセーフトレイトです。

```
unsafe trait Scary { }
```

そして、4 つめは、そのアンセーフトレイトを実装する場合です。

```
unsafe impl Scary for i32 {}
```

大きな問題を引き起こすバグがあるかもしれないコードを明示できるのは重要なことです。もし Rust のプログラムがセグメンテーション違反を起こしても、バグは unsafe で印付けられた区間のどこかにあると確信できます。

「安全」とはどういう意味か?

Rust の文脈で、安全とは「どのようなアンセーフなこともしない」ことを意味します。

> 訳注: 正確には、安全とは「決して未定義動作を起こさない」ということです。そして、安全性が保証されていないことを「アンセーフ」と呼びます。つまり、未定義動作が起きるおそれがあるなら、それはアンセーフです。

知っておくべき重要なことに、たいていのコードにおいて望ましくないが、アンセーフ**ではない** とされている動作がいくらか存在するということがあります。

- デッドロック
- メモリやその他のリソースのリーク
- デストラクタを呼び出さないプログラム終了
- 整数オーバーフロー

Rust はソフトウェアが抱えるすべての種類の問題を防げるわけではありません。Rust でバグのあるコードを書くことはできますし、実際に書かれるでしょう。これらの動作は良いことではありませんが、特にアンセーフだとは見なされません。

さらに、Rust においては、次のものは未定義動作で、 unsafe コード中であっても、避ける必要があります。

> 訳注: 関数に付いている unsafe は「その関数の処理はアンセーフである」ということを表します。その一方で、ブロックに付いている unsafe は「ブロック中の個々の操作はアンセーフだが、全体としては安全な処理である」ということを表します。避ける必要があるのは、未定義動作が起こりうる処理をアンセーフブロックの中に書くことです。それは、アンセーフブロックの処理が安全であるために、その内部で未定義動作が決して起こらないことが必要だからです。アンセーフ

関数には安全性の保証が要らないので、未定義動作が起こりうるアンセーフ関数を定義することに問題はありません。

- データ競合
- ヌル・ダングリング生ポインタの参照外し
- undef[*60]（未初期化）メモリの読み出し
- 生ポインタによる pointer aliasing rules[*61] の違反
- `&mut T` と `&T` は、`UnsafeCell<U>` を含む `&T` を除き、LLVM のスコープ化された noalias[*62] モデルに従っています。アンセーフコードは、それら参照のエイリアシング保証を破ってはいけません。
- `UnsafeCell<U>` を持たないイミュータブルな値・参照の変更
- コンパイラ Intrinsic 経由の未定義挙動の呼び出し
- `std::ptr::offset` (`offset` intrinsic) を使って、オブジェクトの範囲外を指すこと。ただし、オブジェクトの最後より 1 バイト後を指すことは許されている。
- 範囲の重なったバッファに対して `std::ptr::copy_nonoverlapping_memory` (`memcpy32`/`memcpy64` intrinsics) を使う
- プリミティブ型の不正な値（プライベートなフィールドやローカル変数を含む）
- ヌルかダングリングである参照やボックス
- `bool` における、 `false` (0) か `true` (1) でない値
- `enum` の定義に含まれていない判別子
- `char` における、サロゲートか `char::MAX` を超えた値
- `str` における、UTF-8 でないバイト列
- 他言語から Rust への巻き戻しや、Rust から他言語への巻き戻し

アンセーフの能力

アンセーフ関数・アンセーフブロックでは、Rust は普段できない 3 つのことをさせてくれます。たった 3 つです。それは、

1. 静的ミュータブル変数のアクセスとアップデート。

[*60] http://llvm.org/docs/LangRef.html#undefined-values
[*61] http://llvm.org/docs/LangRef.html#pointer-aliasing-rules
[*62] http://llvm.org/docs/LangRef.html#noalias

2. 生ポインタの参照外し。
3. アンセーフ関数の呼び出し。これが最も強力な能力です。

以上です。 重要なのは、 unsafe が、たとえば「借用チェッカをオフにする」といったことを行わないことです。Rust のコードの適当な位置に unsafe を加えてもセマンティクスは変わらず、何でもただ受理するようになるということにはなりません。それでも、unsafe はルールのいくつかを破るコードを書けるようにはするのです。

また、unsafe キーワードは、Rust 以外の言語とのインターフェースを書くときに遭遇するでしょう。ライブラリの提供するメソッドの周りに、安全な、Rust ネイティブのインターフェースを書くことが推奨されています。

これから、その基本的な 3 つの能力を順番に見ていきましょう。

static mut のアクセスとアップデート。

Rust には「static mut」という、ミュータブルでグローバルな状態を実現する機能があります。これを使うことはデータレースが起こるおそれがあるので、本質的に安全ではありません。詳細は、この本の static セクションを参照してください。

生ポインタの参照外し

生ポインタによって任意のポインタ演算が可能になりますが、いくつもの異なるメモリ安全とセキュリティの問題が起こるおそれがあります。ある意味で、任意のポインタを参照外しする能力は行いうる操作のうち最も危険なもののひとつです。詳細は、この本の生ポインタに関するセクションを参照してください。

アンセーフ関数の呼び出し

この最後の能力は、unsafe の両面とともに働きます。すなわち、unsafe で印付けられた関数は、アンセーフブロックの内部からのみ呼び出すことができます。

この能力は強力で多彩です。Rust はいくらかの compiler intrinsics をアンセーフ関数として公開しており、また、いくつかのアンセーフ関数は安全性検査を回避することで、安全性とスピードを引き換えています。

繰り返しになりますが、アンセーフブロックと関数の内部で任意のことが**できる** としても、それをすべきだということを意味しません。コンパイラは、あなたが不変量を守っているかのように動作しますから、注意してください！

5

Effective Rust

これまで Rust コードの書き方について学んできましたが、 **とりあえず** Rust のコードが書けるということと **良い** Rust のコードが書けるということの間には違いがあります。

この章は比較的独立したチュートリアルから構成されており、あなたの Rust のコードをより良くするための方法が説明されています。頻出するパターンや標準ライブラリの機能が紹介されています。あなたの選んだ任意の順序でこれらのセクションをお読みください。

スタックとヒープ

Rust はシステム言語なので、低水準の操作を行います。もしあなたが高水準言語を使ってきたのであれば、システムプログラミングのいくつかの側面をよく知らないかもしれません。一番重要なのは、スタックとヒープと関連してメモリがどのように機能するかということです。もし C 言語のような言語でスタックアロケーションをどのように使っているかをよく知っているのであれば、この章は復習になるでしょう。そうでなければ、このより一般的な概念について、Rust 流の焦点の絞り方ではありますが、学んでゆくことになるでしょう。

ほとんどの物事と同様に、それらについて学ぶにあたって、まず簡略化したモデルを使って始めましょう。そうすることで、今は無関係な枝葉末節に足を取られることなく、基本を把握できます。これから使う例示は 100% 正確ではありませんが、現時点で学ぼうとするレベルのための見本になっています。ひとたび基本を飲み込めば、アロケータがどう実装されているかや仮想メモリなどの発展的なトピックを

学ぶことによって、この特殊な抽象モデルが取り漏らしているものが明らかになるでしょう。

メモリ管理

これら 2 つの用語はメモリ管理についてのものです。スタックとヒープは、いつメモリをアロケート・デアロケートするのかを決定するのを助ける抽象化です。

大まかに比較してみましょう:

スタックはとても高速で、Rust においてデフォルトでメモリが確保される場所です。しかし、このアロケーションはひとつの関数呼び出しに限られた局所的なもので、サイズに制限があります。一方、ヒープはより遅く、プログラムによって明示的にアロケートされます。しかし、事実上サイズに制限がなく、広域的にアクセス可能です。

スタック

次の Rust プログラムについて話しましょう:

```
fn main() {
    let x = 42;
}
```

このプログラムは変数 `x` の束縛をひとつ含んでいます。このメモリはどこかからアロケートされる必要があります。Rust はデフォルトで「スタックアロケート」、すなわち基本的な値を「スタックに置く」ということをします。それはどういう意味でしょうか。

関数が呼び出されたとき、関数中のローカル変数とそのほかの多少の情報のためにメモリがいくらかアロケートされます。これを「スタックフレーム」と呼びますが、このチュートリアルにおいては、余分な情報は無視して、アロケートするローカル変数だけを考えることにします。なので今回の場合は、`main()` が実行されるとき、スタックフレームとして 32 ビット整数をただ 1 つアロケートすることになります。これは、見ての通り自動的に取り扱われるので、特別な Rust コードか何かを書く必要はありません。

関数が終了するとき、スタックフレームはデアロケートされます。これもアロケーションと同様自動的に行われます。

これが、この単純なプログラムにあるものすべてです。ここで理解する鍵となるのは、スタックアロケーションはとても、とても高速だということです。ローカル変数はすべて事前にわかっているので、メモ

リを一度に確保できます。また、破棄するときも同様に、変数をすべて同時に破棄できるので、こちらもとても高速に済みます。

この話でよくないことは、単一の関数を超えて値が必要でも、その値を保持しつづけられないことです。また、「スタック」が何を意味するのかについてまだ話していませんでした。その点について見るために、もう少し複雑な例が必要です。

```
fn foo() {
    let y = 5;
    let z = 100;
}

fn main() {
    let x = 42;

    foo();
}
```

このプログラムには変数が `foo()` に 2 つ、 `main()` に 1 つで、全部で 3 つあります。 前の例と同様に `main()` が呼び出されたときは 1 つの整数がスタックフレームとしてアロケートされます。しかし、`foo()` が呼び出されたときに何が起こるかを話す前に、まずメモリ上に何が置いてあるかを図示する必要があります。オペレーティングシステムは、メモリをプログラムに対してとてもシンプルなものとして見せています。それは、0 からコンピュータが搭載している RAM の容量を表現する大きな数までのアドレスの巨大なリストです。たとえば、もしあなたのコンピュータに 1 ギガバイトの RAM がのっていれば、アドレスは `0` から `1,073,741,823` になります。この数値は、1 ギガバイトのバイト数である 2^{30} から来ています。[*1]

このメモリは巨大な配列のようなものです。すなわち、アドレスは 0 から始まり、最後の番号まで続いています。そして、これが最初のスタックフレームの図です:

Address	Name	Value
0	x	42

[*1] 「ギガバイト」が指すものには、10^9 と 2^{30} の 2 つがありえます。国際単位系 SI では「ギガバイト」は 10^9 を、「ギビバイト」は 2^{30} を指すと決めることで、この問題を解決しています。しかしながら、このような用語法を使う人はとても少なく、文脈で両者を区別しています。ここでは、その慣習に則っています。

この図から、 `x` はアドレス `0` に置かれ、その値は `42` だとわかります。

`foo()` が呼び出されると、新しいスタックフレームがアロケートされます:

Address	Name	Value
2	z	100
1	y	5
0	x	42

`0` は最初のフレームに取られているので、 `1` と `2` が `foo()` のスタックフレームのために使われます。これは、関数呼び出しが行われるたびに上に伸びていきます。

ここで注意しなければならない重要なことがいくつかあります。 `0`, `1`, `2` といった番号は単に解説するためのもので、コンピュータが実際に使うアドレス値とは関係がありません。特に、連続したアドレスは、実際にはそれぞれ数バイトずつ隔てられていて、その間隔は格納されている値のサイズより大きいこともあります。

`foo()` が終了した後、そのフレームはデアロケートされます:

Address	Name	Value
0	x	42

そして `main()` の後には、残っている値も消えてなくなります。簡単ですね!

「スタック」という名は、積み重ねたディナープレート（a stack of dinner plates）のように働くことに由来します。最初に置かれたプレートは、最後に取り去られるプレートです。そのため、スタックはしばしば「last in, first out queues」（訳注: 最後に入ったものが最初に出るキュー、LIFO と略記される）と呼ばれ、最後にスタックに積んだ値は最初にスタックから取り出す値になります。

3 段階の深さの例を見てみましょう:

```
fn italic() {
    let i = 6;
}
```

```
fn bold() {
    let a = 5;
    let b = 100;
    let c = 1;

    italic();
}

fn main() {
    let x = 42;

    bold();
}
```

分かりやすいようにちょと変な名前をつけています。

それでは、まず、 `main()` を呼び出します:

Address	Name	Value
0	x	42

次に、 `main()` は `bold()` を呼び出します:

Address	Name	Value
3	**c**	**1**
2	**b**	**100**
1	**a**	**5**
0	x	42

そして `bold()` は `italic()` を呼び出します:

Address	Name	Value
4	*i*	*6*
3	**c**	**1**
2	**b**	**100**
1	**a**	**5**
0	x	42

ふう、スタックが高く伸びましたね。

`italic()` が終了した後、そのフレームはデアロケートされて `bold()` と `main()` だけが残ります:

Address	Name	Value
3	**c**	**1**
2	**b**	**100**
1	**a**	**5**
0	x	42

そして `bold()` が終了すると `main()` だけが残ります:

Address	Name	Value
0	x	42

ついに、やりとげました。コツをつかみましたか? 皿を積み重ねるようなものです。つまり、一番上に追加し、一番上から取るんです。

ヒープ

さて、このやり方は結構うまくいくのですが、すべてがこのようにいくわけではありません。ときには、メモリを異なる関数間でやりとりしたり、1 回の関数実行より長く保持する必要があります。そのため

には、ヒープを使います。

Rust では、`Box<T>` 型[*2]を使うことで、メモリをヒープ上にアロケートできます。

```
fn main() {
    let x = Box::new(5);
    let y = 42;
}
```

`main()` が呼び出されたとき、メモリは次のようになります:

Address	Name	Value
1	y	42
0	x	??????

2 つの変数のために、スタック上に領域がアロケートされます。 通常通り、`y` は `42` になりますが、 `x` はどうなるのでしょうか? `x` は `Box<i32>` 型で、ボックスはヒープ上のメモリをアロケートします。このボックスの実際の値は、「ヒープ」へのポインタを持ったストラクチャです。関数の実行が開始され、`Box::new()` が呼び出されると、ヒープ上のメモリがいくらかアロケートされ、そこに `5` が置かれます。すると、メモリはこんな感じになります:

Address	Name	Value
(2^{30}) - 1		5
...	...	...
1	y	42
0	x	→ (2^{30}) - 1

今考えている 1GB の RAM を備えた仮想のコンピュータには (2^{30}) - 1 のアドレスがあります。また、スタックはゼロから伸びていますから、メモリをアロケートするのに一番楽なのは、反対側の端の場所です。ですから、最初の値はメモリのうち番号が一番大きい場所に置かれます。そして、 `x` にある構造

[*2] http://doc.rust-lang.org/std/boxed/index.html

体はヒープ上にアロケートした場所への生ポインタを持っているので、x の値は、今求めた位置 (2^{30}) - 1 です。

ここまでの話では、メモリをアロケート・デアロケートするということのこの文脈における意味を過剰に語ることはありませんでした。詳細を深く掘り下げるのはこのチュートリアルの目的範囲外なのですが、ここで重要なこととして指摘したいのは、ヒープは単にメモリの反対側から伸びるスタックなのではないということです。後ほど例を見ていきますが、ヒープはアロケート・デアロケートをどの順番にしてもよく、その結果「穴」のある状態になります。次の図は、とあるプログラムをしばらく実行していたときのメモリレイアウトです。

Address	Name	Value
(2^{30}) - 1		5
(2^{30}) - 2		
(2^{30}) - 3		
(2^{30}) - 4		42
...	...	...
3	y	→ (2^{30}) - 4
2	y	42
1	y	42
0	x	→ (2^{30}) - 1

この場合では、4 つのものをヒープにアロケートしていますが、2 つはすでにデアロケートされています。アドレス (2^{30}) - 1 と (2^{30}) - 4 の間には、現在使われていない隙間があります。このような隙間がなぜ、どのように起きるかの詳細は、どのようなヒープ管理戦略を使っているかによります。異なるプログラムには異なる「メモリアロケータ」というメモリを管理するライブラリを使うことができます。Rust のプログラムはこの用途に jemalloc[*3]を使います。

ともかく、私たちのプログラムの例に戻ります。 この（訳注: x のポインタが指す）メモリはヒープ上にあるので、ボックスをアロケートした関数よりも長い間生存しつづけることができます。しかし、この例ではそうではありません。[*4] 関数が終了したとき、 `main()` のためのスタックフレームを解放する必

[*3] http://www.canonware.com/jemalloc/

[*4] （「変数からのムーブアウト」とも呼ばれることもある）所有権の移動によって、メモリをより長い間生存させられます。よ

要があります。しかし、`Box<T>`には隠れた仕掛け、Drop があります。`Drop` トレイトの `Box` への実装は、ボックスが作られたときにアロケートされたメモリをデアロケートします。すばらしい! なので `x` が解放されるときには先にヒープ上にアロケートされたメモリを解放します。

Address	Name	Value
1	y	42
0	x	??????

その後スタックフレームが無くなることで、全てのメモリが解放されます。

引数と借用

ここまででスタックとヒープの基本的な例をいくつか学び進めましたが、関数の引数と借用についてはどうでしょうか? ここに小さな Rust プログラムがあります:

```
fn foo(i: &i32) {
    let z = 42;
}

fn main() {
    let x = 5;
    let y = &x;

    foo(y);
}
```

処理が `main()` に入ると、メモリはこんな感じになります:

Address	Name	Value
1	y	→ 0
0	x	5

り複雑な例は後ほど解説します。

`x` は普通の `5` で、 `y` は `x` への参照です。そのため、 `y` の値は `x` のメモリ上の位置で、今回は `0` です。

引数として `y` を渡している関数 `foo()` の呼び出しはどうなるのでしょうか?

Address	Name	Value
3	z	42
2	i	→ 0
1	y	→ 0
0	x	5

スタックフレームは単にローカルな束縛のために使われるだけでなく、引数のためにも使われます。なので、この例では、引数の `i` とローカル変数の束縛 `z` の両方が必要です。 `i` は引数 `y` のコピーです。`y` の値は `0` ですから、 `i` の値も `0` になります。

これは、変数を借用してもどのメモリもデアロケートされることがないことのひとつの理由になっています。つまり、参照の値はメモリ上の位置を示すポインタです。もしポインタが指しているメモリを取り去ってしまうと、ことが立ちゆかなくなってしまうでしょう。

複雑な例

それでは、次の複雑な例をステップ・バイ・ステップでやっていきましょう:

```
fn foo(x: &i32) {
    let y = 10;
    let z = &y;

    baz(z);
    bar(x, z);
}

fn bar(a: &i32, b: &i32) {
    let c = 5;
    let d = Box::new(5);
```

```
    let e = &d;

    baz(e);
}

fn baz(f: &i32) {
    let g = 100;
}

fn main() {
    let h = 3;
    let i = Box::new(20);
    let j = &h;

    foo(j);
}
```

まず、`main()` を呼び出します:

Address	Name	Value
(2^{30}) - 1		20
...	...	...
2	j	→ 0
1	i	→ (2^{30}) - 1
0	h	3

`j`, `i`, `h` のためのメモリをアロケートします。`i` が束縛されるボックスが確保する領域はヒープ上にあるので、`i` はそこを指す値を持っています。

つぎに、 `main()` の最後で、 `foo()` が呼び出されます:

Address	Name	Value
(2^{30}) - 1		20
…	…	…
5	z	→ 4
4	y	10
3	x	→ 0
2	j	→ 0
1	i	→ (2^{30}) - 1
0	h	3

`x`, `y`, `z` のための空間が確保されます。 引数 `x` は、渡された `j` と同じ値を持ちます。 `j` は `h` を指しているので、値は `0` アドレスを指すポインタです。

つぎに、 `foo()` は `baz()` を呼び出し、 `z` を渡します:

Address	Name	Value
(2^{30}) - 1		20
…	…	…
7	g	100
6	f	→ 4
5	z	→ 4
4	y	10
3	x	→ 0
2	j	→ 0
1	i	→ (2^{30}) - 1
0	h	3

`f` と `g` のためにメモリを確保しました。 `baz()` はとても短いので、 `baz()` の実行が終わったときに、そ

のスタックフレームを取り除きます。

Address	Name	Value
(2^{30}) - 1		20
…	…	…
5	z	→ 4
4	y	10
3	x	→ 0
2	j	→ 0
1	i	→ (2^{30}) - 1
0	h	3

次に、 `foo()` は `bar()` を `x` と `z` を引数にして呼び出します:

Address	Name	Value
(2^{30}) - 1		20
(2^{30}) - 2		5
…	…	…
10	e	→ 9
9	d	→ (2^{30}) - 2
8	c	5
7	b	→ 4
6	a	→ 0
5	z	→ 4
4	y	10
3	x	→ 0
2	j	→ 0
1	i	→ (2^{30}) - 1

Address	Name	Value
0	h	3

その結果、ヒープに値をもうひとつアロケートすることになるので、(2^{30}) - 1 から 1 を引かなくてはなりません。 そうすることは、`1,073,741,822` と書くよりは簡単です。いずれにせよ、いつものように変数を準備します。

`bar()` の最後で、 `baz()` を呼び出します:

Address	Name	Value
(2^{30}) - 1		20
(2^{30}) - 2		5
…	…	…
12	g	100
11	f	→ (2^{30}) - 2
10	e	→ 9
9	d	→ (2^{30}) - 2
8	c	5
7	b	→ 4
6	a	→ 0
5	z	→ 4
4	y	10
3	x	→ 0
2	j	→ 0
1	i	→ (2^{30}) - 1
0	h	3

こうして、一番深い所までやってきました! ふう! ここまで長い過程をたどってきて、お疲れ様でした。

`baz()` が終わったあとは、 `f` と `g` を取り除きます:

Address	Name	Value
(2^{30}) - 1		20
(2^{30}) - 2		5
…	…	…
10	e	→ 9
9	d	→ (2^{30}) - 2
8	c	5
7	b	→ 4
6	a	→ 0
5	z	→ 4
4	y	10
3	x	→ 0
2	j	→ 0
1	i	→ (2^{30}) - 1
0	h	3

次に、 `bar()` から戻ります。 ここで `d` は `Box<T>` 型なので、 `d` が指している (2^{30}) - 2 も一緒に解放されます。

Address	Name	Value
(2^{30}) - 1		20
…	…	…
5	z	→ 4
4	y	10
3	x	→ 0
2	j	→ 0

Address	Name	Value
1	i	→ (2^{30}) - 1
0	h	3

その後、 `foo()` から戻ります:

Address	Name	Value
(2^{30}) - 1		20
...	...	...
2	j	→ 0
1	i	→ (2^{30}) - 1
0	h	3

そして最後に `main()` から戻るところで、残っているものを除去します。 `i` が `Drop` されるとき、ヒープの最後の残りも除去されます。

他の言語では何をしているのか?

ガベージコレクタを備えた多くの言語はデフォルトでヒープアロケートします。つまり、すべての値がボックス化されています。そうなっている理由がいくつかあるのですが、それはこのチュートリアルの範囲外です。また、そのことが 100% 真であると言えなくなるような最適化もいくつか行われることがあります。メモリの解放のためにスタックと `Drop` を頼りにするかわりに、ガベージコレクタがヒープを取り扱います。

どちらを使えばいいのか?

スタックのほうが速くて管理しやすいというのであれば、なぜヒープが要るのでしょうか? 大きな理由のひとつは、スタックアロケーションだけしかないということはストレージの再利用に「Last In First Out（LIFO）（訳注: 後入れ先出し）」セマンティクスをとるしかないということだからです。ヒープア

ロケーションは厳密により普遍的で、ストレージを任意の順番でプールから取得したり、プールに返却することが許されているのですが、よりコストがかさみます。

一般的にはスタックアロケーションを選ぶべきで、そのため Rust はデフォルトでスタックアロケートします。スタックの LIFO モデルはより単純で、基本的なレベルに置かれています。このことは、実行時の効率性と意味論に大きな影響を与えています。

実行時の効率性

スタックのメモリを管理するのは些細なことです: 機械は「スタックポインタ」と呼ばれる単一の値を増減します。ヒープのメモリを管理するのは些細なことではありません: ヒープアロケートされたメモリは任意の時点で解放され、またヒープアロケートされたそれぞれのブロックは任意のサイズになりうるので、一般的にメモリマネージャは再利用するメモリを特定するためにより多くの仕事をします。

この事柄についてより詳しいことを知りたいのであれば、こちらの論文[*5]がよいイントロダクションになっています。

意味論への影響

スタックアロケーションは Rust の言語自体へ影響を与えており、したがって開発者のメンタルモデルにも影響しています。Rust 言語がどのように自動メモリ管理を取り扱うかは、LIFO セマンティクスに従っています。ヒープアロケートされユニークに所有されたボックスのデアロケーションさえも、スタックベースの LIFO セマンティクスに従っていることは、この章を通して論じてきたとおりです。非 LIFO セマンティクスの柔軟性（すなわち表現能力）は一般的に、いつメモリが解放されるべきなのかをコンパイラがコンパイル時に自動的に推論できなくなることを意味するので、デアロケーションを制御するために、ときに言語自体の外部に由来するかもしれない、動的なプロトコルに頼らなければなりません。（`Rc<T>` や `Arc<T>` が使っている参照カウントはその一例です。）

突き詰めれば、ヒープアロケーションによって増大した表現能力は（例えばガベージコレクタという形の）著しい実行時サポートか、（Rust コンパイラが提供していないような検証を必要とする明示的なメモリ管理呼び出しという形の）著しいプログラマの努力のいずれかのコストを引き起こすのです。

[*5] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.143.4688

テスト

> プログラムのテストはバグの存在を示すためには非常に効率的な方法ですが、バグの不存在を示すためには絶望的に不十分です。エドガー・W・ダイクストラ、『謙虚なプログラマ』(1972)

Rust のコードをテストする方法について話しましょう。ここでは Rust のコードをテストする正しい方法について議論するつもりはありません。テストを書くための正しい方法、誤った方法に関する流派はたくさんあります。それらの方法は全て、同じ基本的なツールを使うので、それらのツールを使うための文法をお見せしましょう。

test アトリビュート

Rust での一番簡単なテストは、 `test` アトリビュートの付いた関数です。 `adder` という名前の新しいプロジェクトを Cargo で作りましょう。

```
$ cargo new adder
$ cd adder
```

新しいプロジェクトを作ると、Cargo は自動的に簡単なテストを生成します。これが `src/lib.rs` の内容です。

```
#[test]
fn it_works() {
}
```

`#[test]` に注意しましょう。このアトリビュートは、この関数がテスト関数であるということを示します。今のところ、その関数には本文がありません。成功させるためにはそれで十分なのです! テストは `cargo test` で実行することができます。

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a
```

```
running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

Cargo はテストをコンパイルし、実行しました。ここでは 2 種類の結果が出力されています。1 つは書かれたテストについてのもの、もう 1 つはドキュメンテーションテストについてのものです。それらについては後で話しましょう。 とりあえず、この行を見ましょう。

```
test it_works ... ok
```

`it_works` に注意しましょう。 これは関数の名前に由来しています。

```
fn it_works() {
```

次のようなサマリも出力されています。

```
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured
```

なぜ何も書いていないテストがこのように成功するのでしょうか。`panic!` しないテストは全て成功で、`panic!` するテストは全て失敗なのです。 テストを失敗させましょう。

```
#[test]
fn it_works() {
    assert!(false);
}
```

`assert!` は Rust が提供するマクロで、1 つの引数を取ります。引数が `true` であれば何も起きません。引数が `false` であれば `panic!` します。 テストをもう一度実行しましょう。

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test it_works ... FAILED

failures:

---- it_works stdout ----
        thread 'it_works' panicked at 'assertion failed: false', /home/steve/tmp/adder
   /src/lib.rs:3



failures:
    it_works

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured

thread '<main>' panicked at 'Some tests failed', /home/steve/src/rust/src/libtest/lib.
   rs:247
```

Rust は次のとおりテストが失敗したことを示しています。

```
test it_works ... FAILED
```

そして、それはサマリにも反映されます。

```
test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured
```

ステータスコードも非 0 になっています。 OS X や Linux では `$?` を使うことができます。

```
$ echo $?
101
```

Windows では、 `cmd` を使っていればこうです。

```
> echo %ERRORLEVEL%
```

そして、PowerShell を使っていればこうです。

```
> echo $LASTEXITCODE # the code itself
> echo $? # a boolean, fail or succeed
```

これは `cargo test` を他のツールと統合したいときに便利です。

もう 1 つのアトリビュート、 `should_panic` を使ってテストの失敗を反転させることができます。

```
#[test]
#[should_panic]
fn it_works() {
    assert!(false);
}
```

今度は、このテストが `panic!` すれば成功で、完走すれば失敗です。試しましょう。

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

Rust はもう 1 つのマクロ、 `assert_eq!` を提供しています。これは 2 つの引数の等価性を調べます。

```
#[test]
#[should_panic]
fn it_works() {
    assert_eq!("Hello", "world");
}
```

このテストは成功でしょうか、失敗でしょうか。 `should_panic` アトリビュートがあるので、これは成功です。

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

`should_panic` を使ったテストは脆いテストです。なぜなら、テストが予想外の理由で失敗したのではないということを保証することが難しいからです。これを何とかするために、 `should_panic` アトリビュートにはオプションで `expected` パラメータを付けることができます。テストハーネスが、失敗したときの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。前述の例のもっと安全なバージョンはこうなります。

```
#[test]
#[should_panic(expected = "assertion failed")]
fn it_works() {
```

```
    assert_eq!(”Hello”, ”world”);
}
```

基本はそれだけです! 「リアルな」テストを書いてみましょう。

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[test]
fn it_works() {
    assert_eq!(4, add_two(2));
}
```

これは非常に一般的な `assert_eq!` の使い方です。いくつかの関数に結果の分かっている引数を渡して呼び出し、期待した結果と比較します。

ignore アトリビュート

ときどき、特定のテストの実行に非常に時間が掛かることがあります。そのようなテストは、 `ignore` アトリビュートを使ってデフォルトでは無効にすることができます。

```
#[test]
fn it_works() {
    assert_eq!(4, add_two(2));
}

#[test]
#[ignore]
fn expensive_test() {
    // 実行に 1 時間掛かるコード
}
```

テストを実行すると、 `it_works` が実行されることを確認できますが、今度は `expensive_test` は実行さ

れません。

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 2 tests
test expensive_test ... ignored
test it_works ... ok

test result: ok. 1 passed; 0 failed; 1 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

無効にされた高コストなテストは `cargo test -- --ignored` を使って明示的に実行することができます。

```
$ cargo test -- --ignored
     Running target/adder-91b3e234d4ed382a

running 1 test
test expensive_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

`--ignored` アトリビュートはテストバイナリの引数であって、Cargo のものではありません。コマンドが

`cargo test -- --ignored` となっているのはそういうことです。

tests モジュール

今までの例における手法は、慣用的ではありません。 `tests` モジュールがないからです。今までの例の慣用的な書き方はこのようになります。

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::add_two;

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
    }
}
```

ここでは、いくつかの変更点があります。 まず、 `cfg` アトリビュートの付いた `mod tests` を導入しました。このモジュールを使うと、全てのテストをグループ化することができます。また、必要であれば、ヘルパ関数を定義し、それをクレートの一部に含まれないようにすることもできます。`cfg` アトリビュートによって、テストを実行しようとしているときにだけテストコードがコンパイルされるようになります。これは、コンパイル時間を節約し、テストが通常のビルドに全く影響しないことを保証してくれます。

2 つ目の変更点は、 `use` 宣言です。ここは内部モジュールの中なので、テスト関数をスコープの中に持ち込む必要があります。モジュールが大きい場合、これは面倒かもしれないので、ここがグロブの一般的な使い所です。`src/lib.rs` をグロブを使うように変更しましょう。

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}
```

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
    }
}
```

use 行が変わったことに注意しましょう。さて、テストを実行します。

```
$ cargo test
    Updating registry `https://github.com/rust-lang/crates.io-index`
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

動きます!

現在の慣習では、 tests モジュールは「ユニット」テストを入れるために使うことになっています。単一の小さな機能の単位をテストするものは全て、ここに入れる意味があります。しかし、「結合」テストはどうでしょうか。 結合テストのためには、tests ディレクトリがあります。

tests ディレクトリ

結合テストを書くために、 tests ディレクトリを作りましょう。そして、その中に次の内容の tests/lib.rs ファイルを置きます。

```
extern crate adder;

#[test]
fn it_works() {
    assert_eq!(4, adder::add_two(2));
}
```

これは前のテストと似ていますが、少し違います。 今回は、extern crate adder を先頭に書いています。 これは、tests ディレクトリの中のテストが全く別のクレートであるため、ライブラリをインポートしなければならないからです。これは、なぜ tests が結合テストを書くのに適切な場所なのかという理由でもあります。そこにあるテストは、そのライブラリを他のプログラムと同じようなやり方で使うからです。

テストを実行しましょう。

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

     Running target/lib-c18e7d3494509e74

running 1 test
test it_works ... ok
```

```
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

今度は 3 つのセクションが出力されました。新しいテストが実行され、前に書いたテストも同様に実行されます。

tests ディレクトリについてはそれだけです。 tests モジュールはここでは必要ありません。全てのものがテストのためのものだからです。

最後に、3 つ目のセクションを確認しましょう。ドキュメンテーションテストです。

ドキュメンテーションテスト

例の付いたドキュメントほどよいものはありません。ドキュメントを書いた後にコードが変更された結果、実際に動かなくなった例ほど悪いものはありません。この状況を終わらせるために、Rust はあなたのドキュメント内の例の自動実行をサポートします（**注意**：これはライブラリクレートの中でのみ動作し、バイナリクレートの中では動作しません）。これが例を付けた具体的な src/lib.rs です。

```
//! `adder`クレートはある数値を数値に加える関数を提供する
//!
//! # Examples
//!
//! ```
//! assert_eq!(4, adder::add_two(2));
//! ```

/// この関数は引数に 2 を加える
///
/// # Examples
///
```

```
/// ```
/// use adder::add_two;
///
/// assert_eq!(4, add_two(2));
/// ```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
    }
}
```

モジュールレベルのドキュメントには `//!` を付け、関数レベルのドキュメントには `///` を付けていることに注意しましょう。Rust のドキュメントは Markdown 形式のコメントをサポートしていて、3 連バッククオートはコードブロックを表します。`# Examples` セクションを含めるのが慣習で、そのとおり、例が後に続きます。

テストをもう一度実行しましょう。

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/steve/tmp/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured
```

```
     Running target/lib-c18e7d3494509e74

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 2 tests
test add_two_0 ... ok
test _0 ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured
```

今回は全ての種類のテストを実行しています! ドキュメンテーションテストの名前に注意しましょう。`_0` はモジュールテストのために生成された名前で、 `add_two_0` は関数テストのために生成された名前です。例を追加するにつれて、それらの名前は `add_two_1` というような形で数値が増えていきます。

まだドキュメンテーションテストの書き方の詳細について、全てをカバーしてはいません。詳しくは ドキュメントの章を見てください。

条件付きコンパイル

Rust には `#[cfg]` という特別なアトリビュートがあり、コンパイラに渡されたフラグに合わせてコードをコンパイルすることを可能にします。`#[cfg]` アトリビュートは以下の 2 つの形式で利用することができます:

```
#[cfg(foo)]

#[cfg(bar = "baz")]
```

また、以下の様なヘルパが存在します:

```
#[cfg(any(unix, windows))]

#[cfg(all(unix, target_pointer_width = "32"))]

#[cfg(not(foo))]
```

ヘルパは以下のように自由にネストすることが可能です:

```
#[cfg(any(not(unix), all(target_os="macos", target_arch = "powerpc")))]
```

このようなスイッチの有効・無効の切り替えは Cargo を利用している場合 `Cargo.toml` 中の `[features]` セクション[*6] で設定できます。

```
[features]
# no features by default
default = []

# フィーチャ「secure-password」は bcrypt パッケージに依存しています
secure-password = ["bcrypt"]
```

もしこのように設定した場合、Cargo は `rustc` に以下のようにフラグを渡します:

```
--cfg feature="${feature_name}"
```

渡されたすべての `cfg` フラグによってどのフラグが有効に成るか決定され、それによってどのコードがコンパイルされるかも決定されます。以下のコードを見てみましょう:

```
#[cfg(feature = "foo")]
mod foo {
}
```

もしこのコードを `cargo build –features "foo"` としてコンパイルを行うと、 `–cfg features="foo"` が `rustc` に渡され、出力には `mod foo` が含まれます。もし標準的な `cargo build` でコンパイルを行った場合、`rustc` に追加のフラグは渡されず `foo` モジュールは存在しない事になります。

[*6] http://doc.crates.io/manifest.html#the-features-section

cfg_attr

また、`cfg_attr` を用いることで、`cfg` に設定された値によってアトリビュートを有効にすることができます:

```
#[cfg_attr(a, b)]
```

このようにすると、`cfg` アトリビュートによって `a` が有効になっている場合に限り `#[b]` と設定されている場合と同じ効果が得られます。

cfg!

`cfg!` 拡張構文は以下のようにコード中でフラグを利用することを可能にします:

```
if cfg!(target_os = "macos") || cfg!(target_os = "ios") {
    println!("Think Different!");
}
```

このようなコードは設定に応じてコンパイル時に `true` または `false` に置き換えられます。

ドキュメント

ドキュメントはどんなソフトウェアプロジェクトにとっても重要な部分であり、Rust においてはファーストクラスです。プロジェクトのドキュメントを作成するために、Rust が提供するツールについて話しましょう。

`rustdoc` について

Rust の配布物には `rustdoc` というドキュメントを生成するツールが含まれています。 `rustdoc` は `cargo doc` によって Cargo でも使われます。

ドキュメントは 2 通りの方法で生成することができます。ソースコードから、そして単体の Markdown ファイルからです。

ソースコードのドキュメントの作成

Rust のプロジェクトでドキュメントを書く 1 つ目の方法は、ソースコードに注釈を付けることで行います。ドキュメンテーションコメントはこの目的のために使うことができます。

```
/// 新しい `Rc<T>` の生成
///
/// # Examples
///
/// ```
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// ```
pub fn new(value: T) -> Rc<T> {
    // 実装が続く
}
```

このコードはこのような[*7]見た目のドキュメントを生成します。実装についてはそこにある普通のコメントのとおり、省略しています。

この注釈について注意すべき 1 つ目のことは、 `//` の代わりに`///` が使われていることです。3 連スラッシュはドキュメンテーションコメントを示します。

ドキュメンテーションコメントは Markdown で書きます。

Rust はそれらのコメントを把握し、ドキュメントを生成するときにそれらを使います。このことは次のように列挙型のようなもののドキュメントを作成するときに重要です。

```
/// `Option` 型。詳細は [モジュールレベルドキュメント](#sec--index) を参照
enum Option<T> {
    /// 値なし
    None,
    /// `T` 型の何らかの値
```

[*7] https://doc.rust-lang.org/nightly/std/rc/struct.Rc.html#method.new

```
    Some(T),
}
```

上記の例は動きますが、これは動きません。

```
/// `Option`型。詳細は[モジュールレベルドキュメント](#sec--index)を参照
enum Option<T> {
    None, /// 値なし
    Some(T), /// `T`型の何らかの値
}
```

次のようにエラーが発生します。

```
hello.rs:4:1: 4:2 error: expected ident, found `}`
hello.rs:4 }
           ^
```

この 残念なエラー[*8]は正しいのです。ドキュメンテーションコメントはそれらの後のものに適用されるところ、その最後のコメントの後には何もないからです。

■ドキュメンテーションコメントの記述

とりあえず、このコメントの各部分を詳細にカバーしましょう。

```
/// 新しい`Rc<T>`の生成
```

ドキュメンテーションコメントの最初の行は、その機能の短いサマリにすべきです。一文で。 基本だけを。 高レベルから。

```
///
/// `Rc<T>`の生成についてのその他の詳細。例えば、複雑なセマンティクスの説明、
/// 追加のオプションなどあらゆる種類のもの
///
```

[*8] https://github.com/rust-lang/rust/issues/22547

この例にはサマリしかありませんが、もしもっと書くべきことがあれば、新しい段落にもっと多くの説明を追加することができます。

特別なセクション

次は特別なセクションです。 それらには `#` が付いていて、ヘッダであることを示しています。一般的には、4 種類のヘッダが使われます。今のところそれらは特別な構文ではなく、単なる慣習です。

```
/// # Panics
```

Rust において、関数の回復不可能な誤用（つまり、プログラミングエラー）は普通、パニックによって表現されます。パニックは、少なくとも現在のスレッド全体の息の根を止めてしまいます。もし関数にこのような、パニックによって検出されたり強制されたりするような自明でない取決めがあるときには、ドキュメントを作成することは非常に重要です。

```
/// # Errors
```

もし関数やメソッドが `Result<T, E>` を戻すのであれば、それが `Err(E)` を戻したときの状況をドキュメントで説明するのはよいことです。 これは `Panics` のときに比べると重要性は少し下です。失敗は型システムによってコード化されますが、それでもまだそうすることはよいことだからです。

```
/// # Safety
```

もし関数が `unsafe` であれば、呼出元が動作を続けるためにはどの不変条件について責任を持つべきなのかを説明すべきです。

```
/// # Examples
///
/// ```
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// ```
```

4 つ目は `Examples` です。関数やメソッドの使い方の例を 1 つ以上含めてください。そうすればユーザから愛されることでしょう。それらの例はコードブロック注釈内に入れます。コードブロック注釈についてはすぐ後で話しますが、それらは 1 つ以上のセクションを持つことができます。

```
/// # Examples
///
/// 単純な `&str` パターン
///
/// ```
/// let v: Vec<&str> = "Mary had a little lamb".split(' ').collect();
/// assert_eq!(v, vec!["Mary", "had", "a", "little", "lamb"]);
/// ```
///
/// ラムダを使ったもっと複雑なパターン
///
/// ```
/// let v: Vec<&str> = "abc1def2ghi".split(|c: char| c.is_numeric()).collect();
/// assert_eq!(v, vec!["abc", "def", "ghi"]);
/// ```
```

それらのコードブロックの詳細について議論しましょう。

コードブロック注釈

コメント内に Rust のコードを書くためには、3 連バッククオートを使います。

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

もし Rust のコードではないものを書きたいのであれば、注釈を追加することができます。

```
/// ```c
/// printf("Hello, world\n");
/// ```
```

これは、使われている言語が何であるかに応じてハイライトされます。もし単なるプレーンテキストを書いているのであれば、 `text` を選択してください。

ここでは正しい注釈を選ぶことが重要です。なぜなら、 `rustdoc` はそれを興味深い方法で使うからです。

それらが実際のコードと不整合を起こさないように、ライブラリクレート内で実際にあなたの例をテストするために使うのです。もし例の中に C のコードが含まれているのに、あなたが注釈を付けるのを忘れてしまい、rustdoc がそれを Rust のコードだと考えてしまえば、rustdoc はドキュメントを生成しようとするときに怒るでしょう。

テストとしてのドキュメント

次のようなドキュメントにおける例について議論しましょう。

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

fn main() とかがここでは不要だということに気が付くでしょう。rustdoc は自動的に main() ラッパをコードの周りに、正しい場所へ配置するためのヒューリスティクスを使って追加します。例えば、こうです。

```
/// ```
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// ```
```

これが、テストのときには結局こうなります。

```
fn main() {
    use std::rc::Rc;
    let five = Rc::new(5);
}
```

これが rustdoc が例の前処理に使うアルゴリズムの全てです。

1. 前の方にある全ての #![foo] アトリビュートは、そのままクレートのアトリビュートとして置いておく
2. unused_variables 、 unused_assignments 、unused_mut 、 unused_attributes 、dead_code などのいくつかの一般的な allow アトリビュートを追加する。小さな例はしばしばこれらのリント

に引っ掛かる

3. もしその例が `extern crate` を含んでいなければ、`extern crate <mycrate>;` を挿入する（`#[macro_use]` がないことに注意する）
4. 最後に、もし例が `fn main` を含んでいなければ、テキストの残りの部分を `fn main() { your_code }` で囲む

こうして生成された `fn main` は問題になり得ます! もし `use` 文によって参照される例のコードに `extern crate` 文や `mod` 文が入っていれば、それらはステップ 4 を抑制するために少なくとも `fn main() {}` を含んでいない限り失敗します。`#[macro_use] extern crate` も同様に、クレートのルート以外では動作しません。そのため、マクロのテストには明示的な `main` が常に必要なのです。しかし、ドキュメントを散らかす必要はありません……続きを読みましょう!

しかし、これでは不十分なことがときどきあります。例えば、今まで話してきた全ての `///` の付いたコード例はどうだったでしょうか。生のテキストはこうなっています。

```
/// 何らかのドキュメント
# fn foo() {}
```

それは出力とは違って見えます。

```
/// 何らかのドキュメント
```

そうです。正解です。 `#` で始まる行を追加することで、コードをコンパイルするときには使われるけれども、出力はされないというようにすることができます。これは都合のよいように使うことができます。この場合、ドキュメンテーションコメントそのものを見せたいので、ドキュメンテーションコメントを何らかの関数に適用する必要があります。そのため、その後に小さい関数定義を追加する必要があります。同時に、それは単にコンパイラを満足させるためだけのものなので、それを隠すことで、例がすっきりするのです。長い例を詳細に説明する一方、テスト可能性を維持するためにこのテクニックを使うことができます。

例えば、このコードをドキュメントに書きたいとします。

```
let x = 5;
let y = 6;
println!("{}", x + y);
```

最終的にはこのように見えるドキュメントが欲しいのかもしれません。

まず、x に 5 をセットする

```
let x = 5;
# let y = 6;
# println!("{}", x + y);
```

次に、y に 6 をセットする

```
# let x = 5;
let y = 6;
# println!("{}", x + y);
```

最後に、x と y との合計を出力する

```
# let x = 5;
# let y = 6;
println!("{}", x + y);
```

各コードブロックをテスト可能な状態にしておくために、各ブロックにはプログラム全体が必要です。しかし、読者には全ての行を毎回見せたくはありません。ソースコードに挿入するものはこれです。

まず、`x`に 5 をセットする

````
```rust
let x = 5;
# let y = 6;
# println!("{}", x + y);
```
````

次に、`y`に 6 をセットする

````
```rust
# let x = 5;
let y = 6;
# println!("{}", x + y);
```
````

    最後に、`x`と`y`との合計を出力する

    ```rust
    # let x = 5;
    # let y = 6;
    println!("{}", x + y);
    ```

例の全体を繰り返すことで、例がちゃんとコンパイルされることを保証する一方、説明に関係する部分だけを見せることができます。

■マクロのドキュメントの作成

これはマクロのドキュメントの例です。

```
/// 式が true と評価されない限り、与えられたメッセージとともにパニックする
///
/// # Examples
///
/// ```
/// # #[macro_use] extern crate foo;
/// # fn main() {
/// panic_unless!(1 + 1 == 2, “Math is broken.”);
/// # }
/// ```
///
/// ```should_panic
/// # #[macro_use] extern crate foo;
/// # fn main() {
/// panic_unless!(true == false, “I’m broken.”);
/// # }
/// ```
#[macro_export]
macro_rules! panic_unless {
```

```
    ($condition:expr, $($rest:expr),+) => ({ if ! $condition { panic!($($rest),+); } }
   );
}
```

3 つのことに気が付くでしょう。`#[macro_use]` アトリビュートを追加するために、自分で `extern crate` 行を追加しなければなりません。 2 つ目に、 `main()` も自分で追加する必要があります（理由は前述しました）。最後に、それらの 2 つが出力されないようにコメントアウトするという`#` の賢い使い方です。

`#` の使うと便利な場所のもう 1 つのケースは、エラーハンドリングを無視したいときです。次のようにしたいとしましょう。

```
/// use std::io;
/// let mut input = String::new();
/// try!(io::stdin().read_line(&mut input));
```

問題は `try!` が `Result<T, E>` を返すところ、テスト関数は何も返さないことで、これは型のミスマッチエラーを起こします。

```
/// try!を使ったドキュメンテーションテスト
///
/// ```
/// use std::io;
/// # fn foo() -> io::Result<()> {
/// let mut input = String::new();
/// try!(io::stdin().read_line(&mut input));
/// # Ok(())
/// # }
/// ```
```

これは関数内のコードをラッピングすることで回避できます。これはドキュメント上のテストが実行されるときに `Result<T, E>` を捕まえて飲み込みます。このパターンは標準ライブラリ内でよく現れます。

■ドキュメンテーションテストの実行

テストを実行するには、次のどちらかを使います。

```
$ rustdoc --test path/to/my/crate/root.rs
# or
$ cargo test
```

正解です。 `cargo test` は組み込まれたドキュメントもテストします。**しかし、`cargo test` がテストするのはライブラリクレートだけで、バイナリクレートはテストしません。**これは `rustdoc` の動き方によるものです。それはテストするためにライブラリをリンクしますが、バイナリには何もリンクするものがないからです。

`rustdoc` がコードをテストするときに正しく動作するのを助けるために便利な注釈があと少しあります。

```
/// ```ignore
/// fn foo() {
/// ```
```

`ignore` ディレクティブは Rust にコードを無視するよう指示します。これはあまりに汎用的なので、必要になることはほとんどありません。もしそれがコードではなければ、代わりに `text` の注釈を付けること、又は問題となる部分だけが表示された、動作する例を作るために`#` を使うことを検討してください。

```
/// ```should_panic
/// assert!(false);
/// ```
```

`should_panic` は、そのコードは正しくコンパイルされるべきではあるが、実際にテストとして成功する必要まではないということを `rustdoc` に教えます。

```
/// ```no_run
/// loop {
///     println!("Hello, world");
/// }
/// ```
```

`no_run` アトリビュートはコードをコンパイルしますが、実行はしません。これは「これはネットワークサービスを開始する方法です」というような例や、コンパイルされることは保証したいけれども、無限ループになってしまうような例にとって重要です!

■モジュールのドキュメントの作成

Rust には別の種類のドキュメンテーションコメント、 `//!` があります。このコメントは次に続く要素のドキュメントではなく、それを囲っている要素のドキュメントです。言い換えると、こうです。

```
mod foo {
    //! これは `foo` モジュールのドキュメントである
    //!
    //! # Examples

    // ...
}
```

`//!` を頻繁に見る場所がここ、モジュールドキュメントです。 もし `foo.rs` 内にモジュールを持っていれば、しばしばそのコードを開くとこれを見るでしょう。

```
//! `foo` で使われるモジュール
//!
//! `foo` モジュールに含まれているたくさんの便利な関数などなど
```

■ドキュメンテーションコメントのスタイル

ドキュメントのスタイルや書式についての全ての慣習を知るには RFC 505[*9] をチェックしてください。

その他のドキュメント

ここにある振舞いは全て、Rust 以外のソースコードファイルでも働きます。コメントは Markdown で書かれるので、しばしば `.md` ファイルになります。

ドキュメントを Markdown ファイルに書くとき、ドキュメントにコメントのプレフィックスを付ける必要はありません。例えば、こうする必要はありません。

[*9] https://github.com/rust-lang/rfcs/blob/master/text/0505-api-comment-conventions.md

```
/// # Examples
///
/// ```
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// ```
```

これは、こうします。

```
### Examples

```
use std::rc::Rc;

let five = Rc::new(5);
```
```

Markdown ファイルの中ではこうします。ただし、1 つだけ新しいものがあります。Markdown ファイルではこのように題名を付けなければなりません。

```
# % The title
% タイトル

# This is the example documentation.
これはサンプルのドキュメントです。
```

この % 行はそのファイルの一番先頭の行に書く必要があります。

doc アトリビュート

もっと深いレベルで言えは、ドキュメンテーションコメントはドキュメントアトリビュートの糖衣構文です。

```
/// this

#[doc="this"]
```

これらは同じもので、次のものも同じものです。

```
//! this

#![doc="this"]
```

このアトリビュートがドキュメントを書くために使われているのを見ることはそんなにないでしょう。しかし、これは何らかのオプションを変更したり、マクロを書いたりするときに便利です。

■再エクスポート

`rustdoc` はパブリックな再エクスポートがなされた場合に、両方の場所にドキュメントを表示します。

```
extern crate foo;

pub use foo::bar;
```

これは `bar` のドキュメントをクレートのドキュメントの中に生成するのと同様に、`foo` クレートのドキュメントの中にも生成します。同じドキュメントが両方の場所で使われます。

この振舞いは `no_inline` で抑制することができます。

```
extern crate foo;

#[doc(no_inline)]
pub use foo::bar;
```

ドキュメントの不存在

ときどき、プロジェクト内の公開されている全てのものについて、ドキュメントが作成されていることを確認したいことがあります。これは特にライブラリについて作業をしているときにあります。Rust で

は、要素にドキュメントがないときに警告やエラーを生成することができます。警告を生成するためには、 warn を使います。

```
#![warn(missing_docs)]
```

そしてエラーを生成するとき、 deny を使います。

```
#![deny(missing_docs)]
```

何かを明示的にドキュメント化されていないままにするため、それらの警告やエラーを無効にしたい場合があります。これは allow を使えば可能です。

```
#[allow(missing_docs)]
struct Undocumented;
```

ドキュメントから要素を完全に見えなくしたいこともあるかもしれません。

```
#[doc(hidden)]
struct Hidden;
```

■HTML の制御

rustdoc の生成する HTML のいくつかの外見は、#![doc] アトリビュートを通じて制御することができます。

```
#![doc(html_logo_url = "https://www.rust-lang.org/logos/rust-logo-128x128-blk-v2.png",
       html_favicon_url = "https://www.rust-lang.org/favicon.ico",
       html_root_url = "https://doc.rust-lang.org/")]
```

これは、複数の異なったオプション、つまりロゴ、お気に入りアイコン、ルートの URL をセットします。

■ドキュメンテーションテストの設定

rustdoc がドキュメントの例をテストする方法は、#[doc(test(..))] アトリビュートを通じて設定することができます。

```
#![doc(test(attr(allow(unused_variables), deny(warnings))))]
```

これによって例の中の使われていない値は許されるようになりますが、その他の全てのリントの警告に対してテストは失敗するようになるでしょう。

生成オプション

`rustdoc` はさらなるカスタマイズのために、その他にもコマンドラインのオプションをいくつか持っています。

- `--html-in-header FILE` : FILE の内容を`<head>...</head>` セクションの末尾に加える
- `--html-before-content FILE` : FILE の内容を`<body>` の直後、レンダリングされた内容（検索バーを含む）の直前に加える
- `--html-after-content FILE` : FILE の内容を全てのレンダリングされた内容の後に加える

セキュリティ上の注意

ドキュメンテーションコメント内の Markdown は最終的なウェブページの中に無修正で挿入されます。リテラルの HTML には注意してください。

```
/// <script>alert(document.cookie)</script>
```

イテレータ

ループの話をしましょう。

Rust の `for` ループを覚えていますか？ 以下が例です。

```
for x in 0..10 {
    println!("{}", x);
}
```

あなたはもう Rust に詳しいので、私たちはどのようにこれが動作しているのか詳しく話すことができます。レンジ (ranges、ここでは `0..10`) は「イテレータ」(iterators) です。イテレータは `.next()` メソッドを繰り返し呼び出すことができ、その都度順番に値を返すものです。

（ところで、 0..10 のようにドット 2 つのレンジは左端を含み（つまり 0 から始まる）右端を含みません（つまり 9 で終わる）。数学的な書き方をすれば「[0, 10)」です。10 まで含むレンジが欲しければ 0...10 と書きます。）

こんな風に:

```
let mut range = 0..10;

loop {
    match range.next() {
        Some(x) => {
            println!("{}", x);
        },
        None => { break }
    }
}
```

始めに変数 range へミュータブルな束縛を行っていますが、これがイテレータです。その次には中に match が入った loop があります。この match は range.next() を呼び出し、イテレータから得た次の値への参照を使用しています。next は Option<i32> を返します。このケースでは、次の値があればその値は Some(i32) であり、返ってくる値が無くなれば None が返ってきます。もし Some(i32) であればそれを表示し、 None であれば break によりループから脱出しています。

このコードは、基本的に for ループバージョンと同じ動作です。for ループはこの loop/ match / break で構成された処理を手軽に書ける方法というわけです。

しかしながら for ループだけがイテレータを使う訳ではありません。自作のイテレータを書く時は Iterator トレイトを実装する必要があります。それは本書の範囲外ですが、Rust は多様な反復処理を実現するために便利なイテレータを幾つか提供しています。ただその前に、少しばかりレンジの限界について言及しておきましょう。

レンジはとても素朴な機能ですから、度々別のより良い手段を用いることもあります。ここである Rust のアンチパターンについて考えてみましょう。それはレンジを C 言語ライクな for ループの再現に用いることです。例えばベクタの中身をイテレートする必要があったとしましょう。あなたはこう書きたくなるかもしれません。

```
let nums = vec![1, 2, 3];

for i in 0..nums.len() {
    println!("{}", nums[i]);
}
```

これは実際のイテレータの使い方からすれば全く正しくありません。あなたはベクタを直接反復処理できるのですから、こう書くべきです。

```
let nums = vec![1, 2, 3];

for num in &nums {
    println!("{}", num);
}
```

これには 2 つの理由があります。第一に、この方が書き手の意図をはっきり表せています。私たちはベクタのインデックスを作成してからその要素を繰り返し参照したいのではなく、ベクタ自体を反復処理したいのです。第二に、このバージョンのほうがより効率的です。1 つ目の例では `num[i]` というようにインデックスを介し参照しているため、余計な境界チェックが発生します。しかし、イテレータが順番にベクタの各要素への参照を生成していくため、2 つ目の例では境界チェックが発生しません。これはイテレータにとってごく一般的な性質です。不要な境界チェックを省くことができ、それでもなお安全なままなのです。

ここにはもう 1 つ、`println!` の動作という詳細が 100% はっきりしていない処理があります。 `num` は実際には `&i32` 型です。これは `i32` の参照であり、`i32` それ自体ではありません。 `println!` は上手い具合に参照外し (dereferencing) をしてくれますから、これ以上深追いはしないことにします。以下のコードも正しく動作します。

```
let nums = vec![1, 2, 3];

for num in &nums {
    println!("{}", *num);
}
```

今、私たちは明示的に `num` の参照外しを行いました。なぜ`&nums` は私たちに参照を渡すのでしょうか？

第一に、&を用いて私たちが明示的に要求したからです。第二に、もしデータそれ自体を渡す場合、私たちはデータの所有者でなければならないため、データの複製と、それを私たちに渡す操作が伴います。参照を使えば、データへの参照を借用して渡すだけで済み、ムーブを行う必要がなくなります。

ここまでで、多くの場合レンジはあなたの欲する物ではないとわかりましたから、あなたが実際に欲しているものについて話しましょう。

それは大きく分けてイテレータ、 **イテレータアダプタ** (iterator adaptors)、そして **コンシューマ** (consumers) の 3 つです。以下が定義となります。

- **イテレータ** は値のシーケンスを渡します。
- **イテレータアダプタ**はイテレータに作用し、出力の異なるイテレータを生成します。
- **コンシューマ**はイテレータに作用し、幾つかの最終的な値の組を生成します。

既にイテレータとレンジについて見てきましたから、初めにコンシューマについて話しましょう。

コンシューマ

コンシューマとはイテレータに作用し、何らかの値を返すものです。最も一般的なコンシューマは `collect()` です。このコードは全くコンパイルできませんが、意図するところは伝わるでしょう。

```
let one_to_one_hundred = (1..101).collect();
```

ご覧のとおり、ここではイテレータの `collect()` を呼び出しています。 `collect()` はイテレータが渡す沢山の値を全て受け取り、その結果をコレクションとして返します。それならなぜこのコードはコンパイルできないのでしょうか？ Rust はあなたが集めたい値の型を判断することができないため、あなたが欲しい型を指定する必要があります。以下のバージョンはコンパイルできます。

```
let one_to_one_hundred = (1..101).collect::<Vec<i32>>();
```

もしあなたが覚えているなら、 `::<>` 構文で型ヒント (type hint) を与え、整数型のベクタが欲しいと伝えることができます。かといって常に型をまるごとを書く必要はありません。 _ を用いることで部分的に推論してくれます。

```
let one_to_one_hundred = (1..101).collect::<Vec<_>>();
```

これは「値を `Vec<T>` の中に集めて下さい、しかし `T` は私のために推論して下さい」という意味です。こ

のため _ は「型プレースホルダ」(type placeholder) と呼ばれることもあります。

`collect()` は最も一般的なコンシューマですが、他にもあります。`find()` はそのひとつです。

```
let greater_than_forty_two = (0..100)
                             .find(|x| *x > 42);

match greater_than_forty_two {
    Some(_) => println!("Found a match!"),
    None => println!("No match found :("),
}
```

`find` はクロージャを引数にとり、イテレータの各要素の参照に対して処理を行います。ある要素が私たちの期待するものであれば、このクロージャは `true` を返し、そうでなければ `false` を返します。マッチングする要素が無いかもしれないので、 `find` は要素それ自体ではなく `Option` を返します。

もう一つの重要なコンシューマは `fold` です。こんな風になります。

```
let sum = (1..4).fold(0, |sum, x| sum + x);
```

`fold()` は `fold(base, |accumulator, element| ...)` というシグネチャのコンシューマで、2 つの引数を取ります。第 1 引数は *base* (基底) と呼ばれます。第 2 引数は 2 つ引数を受け取るクロージャです。クロージャの第 1 引数は *accumulator* (累積値) と呼ばれており、第 2 引数は *element* (要素) です。各反復毎にクロージャが呼び出され、その結果が次の反復の accumulator の値となります。反復処理の開始時は、base が accumulator の値となります。

ええ、ちょっとややこしいですね。ではこのイテレータを以下の値で試してみましょう。

base	accumulator	element	クロージャの結果
0	0	1	1
0	1	2	3
0	3	3	6

これらの引数で `fold()` を呼び出してみました。

```
.fold(0, |sum, x| sum + x);
```

というわけで、 `0` が base で、 `sum` が accumulator で、x が element です。1 度目の反復では、私たちは sum に 0 をセットし、`nums` の 1 つ目の要素 `1` が `x` になります。その後 `sum` と `x` を足し、 `0 + 1 = 1` を計算します。2 度目の反復では前回の `sum` が accumulator になり、element は値の列の 2 番目の要素 `2` になります。 `1 + 2 = 3` の結果は最後の反復処理における accumulator の値になります。最後の反復処理において、`x` は最後の要素 `3` であり、 `3 + 3 = 6` が最終的な結果となります。 `1 + 2 + 3 = 6` 、これが得られる結果となります。

ふぅ、ようやく説明し終わりました。 `fold` は初めのうちこそ少し奇妙に見えるかもしれませんが、一度理解すればあらゆる場面で使えるでしょう。何らかのリストを持っていて、そこから 1 つの結果を求めたい時ならいつでも、`fold` は適切な処理です。

イテレータにはまだ話していないもう 1 つの性質、遅延性があり、コンシューマはそれに関連して重要な役割を担っています。それではもっと詳しくイテレータについて話していきましょう、そうすればなぜコンシューマが重要なのか理解できるはずです。

イテレータ

前に言ったように、イテレータは `.next()` メソッドを繰り返し呼び出すことができ、その都度順番に値を返すものです。メソッドを繰り返し呼ぶ必要があることから、イテレータは *lazy* であり、前もって全ての値を生成できないことがわかります。このコードでは、例えば `1-99` の値は実際には生成されておらず、代わりにただシーケンスを表すだけの値を生成しています。

```
let nums = 1..100;
```

私たちはレンジを何にも使っていないため、値を生成しません。コンシューマを追加してみましょう。

```
let nums = (1..100).collect::<Vec<i32>>();
```

`collect()` は幾つかの数値を渡してくれるレンジを要求し、シーケンスを生成する作業を行います。

レンジは基本的な 2 つのイテレータのうちの 1 つです。もう片方は `iter()` です。 `iter()` はベクタを順に各要素を渡すシンプルなイテレータへ変換できます。

```
let nums = vec![1, 2, 3];

for num in nums.iter() {
   println!("{}", num);
}
```

これら 2 つの基本的なイテレータはあなたの役に立つはずです。無限を扱えるものも含め、より応用的なイテレータも幾つか用意されています。

これでイテレータについては十分でしょう。私たちがイテレータに関して最後に話しておくべき概念がイテレータアダプタです。それでは説明しましょう！

イテレータアダプタ

イテレータアダプタはイテレータを受け取って何らかの方法で加工し、新たなイテレータを生成します。`map` はその中でも最も単純なものです。

```
(1..100).map(|x| x + 1);
```

`map` は別のイテレータについて呼び出され、各要素の参照をクロージャに引数として与えた結果を新しいイテレータとして生成します。つまりこのコードは私たちに `2-100` の値を返してくれるでしょう。えーっと、厳密には少し違います！ もしこの例をコンパイルすると、こんな警告が出るはずです。

```
warning: unused result which must be used: iterator adaptors are lazy and
         do nothing unless consumed, #[warn(unused_must_use)] on by default
(1..100).map(|x| x + 1);
 ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

また遅延性にぶつかりました！ このクロージャは実行されませんね。例えば以下の例は何の数字も出力されません。

```
(1..100).map(|x| println!("{}", x));
```

もし副作用のためにイテレータに対してクロージャの実行を試みるのであれば、代わりに `for` を使いましょう。

他にも面白いイテレータアダプタは山ほどあります。 take(n) は元のイテレータの n 要素目までを実行するイテレータを返します。先程言及していた無限を扱えるイテレータで試してみましょう。

```
for i in (1..).take(5) {
    println!("{}", i);
}
```

これの出力は、

```
1
2
3
4
5
```

filter() は引数としてクロージャをとるアダプタです。このクロージャは true か false を返します。filter() が生成する新たなイテレータはそのクロージャが true を返した要素のみとなります。

```
for i in (1..100).filter(|&x| x % 2 == 0) {
    println!("{}", i);
}
```

これは 1 から 100 の間の偶数を全て出力します。（map と違って、filter に渡されたクロージャには要素そのものではなく要素への参照が渡されます。フィルタする述語は &x パターンを使って整数を取り出しています。フィルタするクロージャは要素ではなく true 又は false を返すので、 filter の実装は返り値のイテレータに所有権を渡すために要素への所有権を保持しておかないとならないのです。）

> 訳注: クロージャで用いられている &x パターンはパターンの章では紹介されていません。簡単に解説すると、何らかの参照 &T から **内容のみを取り出してコピー** するのが &x パターンです。参照をそのまま受け取る ref x パターンとは異なりますので注意して下さい。

あなたはここまでに説明された 3 つの概念を全て繋げることができます。イテレータから始まり、アダプタを幾つか繋ぎ、結果を消費するといった感じです。これを見て下さい。

```
(1..)
    .filter(|&x| x % 2 == 0)
    .filter(|&x| x % 3 == 0)
    .take(5)
    .collect::<Vec<i32>>();
```

これは `6, 12, 18, 24,` 、そして `30` が入ったベクタがあなたに渡されます。

イテレータ、イテレータアダプタ、そしてコンシューマがあなたの助けになることをほんの少しだけ体験できました。本当に便利なイテレータが幾つも用意されていますし、あなたがイテレータを自作することもできます。イテレータは全ての種類のリストに対し効率的な処理方法と安全性を提供します。これらは初めこそ珍しいかもしれませんが、もし使えばあなたは夢中になることでしょう。全てのイテレータとコンシューマのリストはイテレータモジュールのドキュメンテーション[*10]を参照して下さい。

並行性

並行性と並列性はコンピュータサイエンスにおいて極めて重要なトピックであり、現在では産業界でもホットトピックです。コンピュータはどんどん多くのコアを持つようになってきていますが、多くのプログラマはまだそれを十分に使いこなす準備ができていません。

Rust のメモリ安全性の機能は、Rust の並行性の話においても適用されます。Rust プログラムは並行であっても、メモリ安全でなければならず、データ競合を起こさないのです。Rust の型システムはこの問題を扱うことができ、並行なコードをコンパイル時に確かめるための強力な方法を与えます。

Rust が備えている並行性の機能について語る前に、理解しておくべき重要なことがあります。それは、Rust は十分にローレベルであるため、その大部分は、言語によってではなく、標準ライブラリによって提供されるということです。これは、もし Rust の並行性の扱い方に気に入らないところがあれば、代わりの方法を実装できるということを意味します。mio[*11] はこの原則を行動で示している実例です。

背景: `Send` と `Sync`

並行性を確かめるのは難しいことです。Rust には、コードを確かめるのを支援する強力で静的な型システムがあります。そして Rust は、並行になりうるコードの理解を助ける 2 つのトレイトを提供します。

[*10] http://doc.rust-lang.org/std/iter/index.html
[*11] https://github.com/carllerche/mio

■Send

最初に取り上げるトレイトは Send[*12] です。 型 T が Send を実装していた場合、この型のものはスレッド間で安全に受け渡しされる所有権を持てることを意味します。

これはある種の制約を強制させる際に重要です。例えば、もし 2 つのスレッドをつなぐチャネルがあり、そのチャネルを通じてデータを別のスレッドに送れるようにしたいとします。このときには、その型について Send が実装されているかを確かめます。

逆に、スレッドセーフでない FFI でライブラリを包んでいて、 Send を実装したくなかったとします。このときコンパイラは、そのライブラリが現在のスレッドの外にいかないよう強制することを支援してくれるでしょう。

■Sync

2 つ目のトレイトは Sync[*13] といいます。 型 T が Sync を実装していた場合、この型のものは共有された参照を通じて複数スレッドから並行に使われたとしても、必ずメモリ安全であることを意味します。そのため、 interior mutability を持たない型はもともと Sync であるといえます。そのような型としては、 u8 などの単純なプリミティブ型やそれらを含む合成型などがあります。

スレッドをまたいで参照を共有するために、Rust は Arc<T> というラッパ型を提供しています。T が Send と Sync の両方を実装している時かつその時に限り、Arc<T> は Send と Sync を実装します。 例えば、型 Arc<RefCell<U>> のオブジェクトをスレッドをまたいで受け渡すことはできません。 なぜなら、RefCell は Sync を実装していないため、Arc<RefCell<U>> は Send を実装しないためです。

これらの 2 つのトレイトのおかげで、コードの並行性に関する性質を強く保証するのに型システムを使うことができます。ただ、それがどうしてかということを示す前に、まずどうやって並行な Rust プログラムをつくるかということを学ぶ必要があります!

スレッド

Rust の標準ライブラリはスレッドのためのライブラリを提供しており、それにより Rust のコードを並列に走らせることができます。これが std::thread を使う基本的な例です。

[*12] http://doc.rust-lang.org/std/marker/trait.Send.html
[*13] http://doc.rust-lang.org/std/marker/trait.Sync.html

```
use std::thread;

fn main() {
    thread::spawn(|| {
        println!("Hello from a thread!");
    });
}
```

thread::spawn() というメソッドはクロージャを受け取り、それを新たなスレッドで実行します。そして、元のスレッドにハンドルを返します。このハンドルは、子スレッドが終了するのを待機しその結果を取り出すのに使うことが出来ます。

```
use std::thread;

fn main() {
    let handle = thread::spawn(|| {
        "Hello from a thread!"
    });

    println!("{}", handle.join().unwrap());
}
```

クロージャは環境から変数を捕捉出来るので、スレッドにデータを取り込もうとすることも出来ます。

```
use std::thread;

fn main() {
    let x = 1;
    thread::spawn(|| {
        println!("x is {}", x);
    });
}
```

しかし、これはエラーです。

```
5:19: 7:6 error: closure may outlive the current function, but it
                 borrows `x`, which is owned by the current function
...
5:19: 7:6 help: to force the closure to take ownership of `x` (and any other reference
   d variables),
         use the `move` keyword, as shown:
      thread::spawn(move || {
          println!(”x is {}”, x);
      });
```

これはクロージャはデフォルトで変数を参照で捕捉するためクロージャは *x への参照* のみを捕捉するからです。これは問題です。なぜならスレッドは x のスコープよに長生きするかもしれないのでダングリングポインタを招きかねません。

これを直すにはエラーメッセージにあるように move クロージャを使います。 move クロージャはこちらで詳細に説明されていますが、基本的には変数を環境からクロージャへムーブします。

```
use std::thread;

fn main() {
    let x = 1;
    thread::spawn(move || {
        println!(”x is {}”, x);
    });
}
```

多くの言語はスレッドを実行できますが、それはひどく危険です。 shared mutable state によって引き起こされるエラーをいかに防ぐかを丸々あつかった本もあります。Rust はこれについて型システムによって、コンパイル時にデータ競合を防ぐことで支援します。それでは、実際にどうやってスレッド間での共有を行うかについて話しましょう。

> 訳注: “shared mutable state” は「共有されたミュータブルな状態」という意味ですが、定型句として、訳さずそのまま使用しています。

安全な Shared Mutable State

Rust の型システムのおかげで、「安全な shared mutable state」という嘘のようにきこえる概念があらわれます。 shared mutable state がとてもとても悪いものであるということについて、多くのプログラマの意見は一致しています。

このようなことを言った人がいます。

> Shared mutable state is the root of all evil. Most languages attempt to deal with this problem through the 'mutable' part, but Rust deals with it by solving the 'shared' part.
>
> 訳: shared mutable state は諸悪の根源だ。 多くの言語は `mutable` の部分を通じてこの問題に対処しようとしている。 しかし、Rust は `shared` の部分を解決することで対処する。

ポインタの誤った使用の防止には所有権のシステムが役立ちますが、このシステムはデータ競合を排除する際にも同様に一役買います。データ競合は、並行性のバグの中で最悪なものの一つです。

例として、多くの言語で起こりうるようなデータ競合を含んだ Rust プログラムをあげます。これは、コンパイルが通りません。

```
use std::thread;
use std::time::Duration;

fn main() {
    let mut data = vec![1, 2, 3];

    for i in 0..3 {
        thread::spawn(move || {
            data[i] += 1;
        });
    }

    thread::sleep(Duration::from_millis(50));
}
```

以下のようなエラーがでます。

```
8:17 error: capture of moved value: `data`
        data[i] += 1;
        ^~~~
```

Rust はこれが安全でないだろうと知っているのです! もし、各スレッドに `data` への参照があり、スレッドごとにその参照の所有権があるとしたら、 3 人の所有者がいることになってしまうのです! `data` は最初の `spawn` の呼び出しで `main` からムーブしてしまっているので、ループ内の続く呼び出しはこの変数を使えないのです。

この例では配列の異ったインデックスにアクセスしているのでデータ競合は起きません。しかしこの分離性はコンパイル時に決定出来ませんし `i` が定数や乱数だった時にデータ競合が起きます。

そのため、1 つの値に対して 2 つ以上の所有権を持った参照を持てるような型が必要です。通常、この用途には `Rc<T>` を使います。これは所有権の共有を提供する参照カウントの型です。実行時にある程度の管理コストを払って、値への参照の数をカウントします。なので名前に参照カウント (reference count) が付いているのです。

`Rc<T>` に対して `clone()` を呼ぶと新たな所有権を持った参照を返し、内部の参照カウント数を増やします。スレッドそれぞれで `clone()` を取ります:

```
use std::thread;
use std::time::Duration;
use std::rc::Rc;

fn main() {
    let mut data = Rc::new(vec![1, 2, 3]);

    for i in 0..3 {
#       // create a new owned reference
        // 所有権を持った参照を新たに作る
        let data_ref = data.clone();

#       // use it in a thread
        // スレッド内でそれを使う
        thread::spawn(move || {
            data_ref[i] += 1;
        });
```

```
    }

    thread::sleep(Duration::from_millis(50));
}
```

これは動作せず、以下のようなエラーを出します:

```
13:9: 13:22 error: the trait bound `alloc::rc::Rc<collections::vec::Vec<i32>> : core::
   marker::Send`
            is not satisfied
...
13:9: 13:22 note: `alloc::rc::Rc<collections::vec::Vec<i32>>`
            cannot be sent between threads safely
```

エラーメッセージで言及があるように、 `Rc` は安全に別のスレッドに送ることが出来ません。これは内部の参照カウントがスレッドセーフに管理されていないのでデータ競合を起こし得るからです。

この問題を解決するために、 `Arc<T>` を使います。Rust の標準のアトミックな参照カウント型です。

「アトミック」という部分は `Arc<T>` が複数スレッドから安全にアクセスできることを意味しています。このためにコンパイラは、内部のカウントの更新には、データ競合が起こりえない分割不能な操作が用いられることを保証します。

要点は `Arc<T>` は **スレッド間**で所有権を共有可能にする型ということです。

```
use std::thread;
use std::sync::Arc;
use std::time::Duration;

fn main() {
    let mut data = Arc::new(vec![1, 2, 3]);

    for i in 0..3 {
        let data = data.clone();
        thread::spawn(move || {
            data[i] += 1;
        });
```

```
    }

    thread::sleep(Duration::from_millis(50));
}
```

前回と同様に clone() を使って所有権を持った新たなハンドルを作っています。そして、このハンドルは新たなスレッドに移動されます。

そうすると… まだ、エラーがでます。

```
<anon>:11:24 error: cannot borrow immutable borrowed content as mutable
<anon>:11                    data[i] += 1;
                             ^~~~
```

Arc<T> が保持する値はデフォルトでイミュータブルです。 スレッド間での**共有**はしてくれますがスレッドが絡んだ時の共有されたミュータブルなデータはデータ競合を引き起こし得ます。

通常イミュータブルな位置のものをミュータブルにしたい時は Cell<T> 又は RefCell<T> が実行時のチェックあるいは他の方法で安全に変更する手段を提供してくれる（参考: 保障を選ぶ）のでそれを使います。しかしながら Rc と同じくこれらはスレッドセーフではありません。これらを使おうとすると Sync でない旨のエラーが出てコンパイルに失敗します。

スレッド間で共有された値を安全に変更出来る型、例えばどの瞬間でも同時に 1 スレッドしか内容の値を変更できないことを保障する型が必要そうです。

そのためには、 Mutex<T> 型を使うことができます!

これが動くバージョンです。

```
use std::sync::{Arc, Mutex};
use std::thread;
use std::time::Duration;

fn main() {
    let data = Arc::new(Mutex::new(vec![1, 2, 3]));

    for i in 0..3 {
        let data = data.clone();
```

```
        thread::spawn(move || {
            let mut data = data.lock().unwrap();
            data[i] += 1;
        });
    }

    thread::sleep(Duration::from_millis(50));
}
```

i の値はクロージャへ束縛 (コピー) されるだけで、スレッド間で共有されるわけではないことに注意してください。

ここでは mutex を「ロック」しているのです。 mutex（「mutual exclusion（訳注: 相互排他）」の略）は前述の通り同時に 1 つのスレッドからのアクセスしか許しません。値にアクセスしようと思ったら、lock() を使います。これは値を使い終わるまで mutex を「ロック」して他のどのスレッドもロック出来ない（そして値に対して何も出来ない）ようにします。もし既にロックされている mutex をロックしようとすると別のスレッドがロックを解放するまで待ちます。

ここでの「解放」は暗黙的です。ロックの結果（この場合は data）がスコープを出ると、ロックは自動で解放されます

Mutex[*14]の lock[*15]メソッドは次のシグネチャを持つことを気をつけて下さい。

```
fn lock(&self) -> LockResult<MutexGuard<T>>
```

そして、 Send は MutexGuard<T> に対して実装されていないため、ガードはスレッドの境界をまたげず、ロックの獲得と解放のスレッドローカル性が保証されています。

それでは、スレッドの中身をさらに詳しく見ていきましょう。

```
thread::spawn(move || {
    let mut data = data.lock().unwrap();
    data[i] += 1;
});
```

[*14] http://doc.rust-lang.org/std/sync/struct.Mutex.html
[*15] http://doc.rust-lang.org/std/sync/struct.Mutex.html#method.lock

まず、`lock()` を呼び、mutex のロックを獲得します。これは失敗するかもしれないため、`Result<T, E>` が返されます。そして、今回は単なる例なので、データへの参照を得るためにそれを `unwrap()` します。実際のコードでは、ここでもっとちゃんとしたエラーハンドリングをするでしょう。そうしたら、ロックを持っているので、自由に値を変更できます。

最後の部分で、スレッドが実行されている間、短いタイマで待機しています。しかし、これはよろしくないです。というのも、ちょうどよい待機時間を選んでいた可能性より、必要以上に長い時間待ってしまっていたり、十分に待っていなかったりする可能性の方が高いからです。適切な待ち時間というのは、プログラムを実行した際に、実際に計算が終わるまでどれだけの時間がかかったかに依存します。

タイマに代わるより良い選択肢は、Rust 標準ライブラリによって提供されている、スレッドがお互いに同期するためのメカニズムを用いることです。それでは、そのようなものの一つについて話しましょう。チャネルです。

チャネル

このコードが、適当な時間を待つ代わりに、同期のためにチャネルを使ったバージョンです。

```
use std::sync::{Arc, Mutex};
use std::thread;
use std::sync::mpsc;

fn main() {
    let data = Arc::new(Mutex::new(0));

    // `tx` は送信（transmitter）
    // `rx` は受信（receiver）
    let (tx, rx) = mpsc::channel();

    for _ in 0..10 {
        let (data, tx) = (data.clone(), tx.clone());

        thread::spawn(move || {
            let mut data = data.lock().unwrap();
            *data += 1;
```

```
            tx.send(()).unwrap();
        });
    }

    for _ in 0..10 {
        rx.recv().unwrap();
    }
}
```

`mpsc::channel()` メソッドを使って、新たなチャネルを生成しています。 そして、ただの `()` をチャネルを通じて `send` し、それが１０個戻ってくるのを待機します。

このチャネルはシグナルを送っているだけですが、 `Send` であるデータならばなんでもこのチャネルを通じて送れます!

```
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    for i in 0..10 {
        let tx = tx.clone();

        thread::spawn(move || {
            let answer = i * i;

            tx.send(answer).unwrap();
        });
    }

    for _ in 0..10 {
        println!("{}", rx.recv().unwrap());
    }
}
```

ここでは、１０個のスレッドを生成し、それぞれに数値 (spawn() したときの i) の２乗を計算させ、その答えをチャネルを通じて send() で送り返させています。

パニック

panic! は現在実行中のスレッドをクラッシュさせます。Rust のスレッドは独立させるための単純なメカニズムとして使うことができます。

```
use std::thread;

let handle = thread::spawn(move || {
    panic!("oops!");
});

let result = handle.join();

assert!(result.is_err());
```

Thread.join() は Result を返し、これによってスレッドがパニックしたかどうかをチェックできます。

エラーハンドリング

他のほとんどのプログラミング言語と同様、Rust はプログラマに、ある決まった作法でエラーを扱うことを促します。一般的にエラーハンドリングは、例外、もしくは、戻り値を使ったものの、大きく 2 つに分類されます。Rust では戻り値を使います。

このセクションでは、Rust でのエラーハンドリングに関わる包括的な扱い方を提示しようと思います。単にそれだけではなく、エラーハンドリングのやり方を、ひとつひとつ、順番に積み上げていきます。こうすることで、全体がどう組み合わさっているのかの理解が進み、より実用的な知識が身につくでしょう。

もし素朴なやり方を用いたなら、Rust におけるエラーハンドリングは、冗長で面倒なものになり得ます。このセクションでは、エラーを処理する上でどのような課題があるかを吟味し、標準ライブラリを使うと、それがいかにシンプルでエルゴノミック（人間にとって扱いやすいもの）に変わるのかを紹介します。

目次

このセクションはとても長くなります。 というのは、直和型 (sum type) とコンビネータから始めることで、Rust におけるエラーハンドリングを徐々に改善していくための動機を与えるからです。このような構成ですので、もしすでに他の表現力豊かな型システムの経験があるプログラマでしたら、あちこち拾い読みしたくなるかもしれません。

  - 標準入力から読み込む
  - 独自のエラー型によるエラー処理
  - 機能を追加する
- まとめ

基礎

エラーハンドリングとは、ある処理が成功したかどうかを **場合分け** *(case analysis)* に基づいて判断するものだと考えられます。これから見ていくように、エラーハンドリングをエルゴノミックにするために重要なのは、プログラマがコードを合成可能 (composable) に保ったまま、明示的な場合分けの回数を、いかに減らしていくかということです。

コードを合成可能に保つのは重要です。なぜなら、もしこの要求がなかったら、想定外のことが起こる度に `panic`[*16] することを選ぶかもしれないからです。（`panic` は現タスクを巻き戻し (unwind) して、ほとんどの場合、プログラム全体をアボートします。）

```
// 1 から 10 までの数字を予想します。
// もし予想した数字に一致したら true を返し、そうでなければ false を返します。
fn guess(n: i32) -> bool {
    if n < 1 || n > 10 {
        panic!("Invalid number: {}", n);
    }
    n == 5
}

fn main() {
    guess(11);
}
```

訳注：文言の意味は

- Invalid number: {}：無効な数字です: {}

ですが、エディタの設定などによっては、ソースコード中のコメント以外の場所に日本語を使うとコンパイルできないことがあるので、英文のままにしてあります。

[*16] http://doc.rust-lang.org/std/macro.panic!.html

このコードを実行すると、プログラムがクラッシュして、以下のようなメッセージが表示されます。

```
thread '<main>' panicked at 'Invalid number: 11', src/bin/panic-simple.rs:5
```

次は、もう少し自然な例です。このプログラムは引数として整数を受け取り、2 倍した後に表示します。

```
use std::env;

fn main() {
    let mut argv = env::args();
    let arg: String = argv.nth(1).unwrap(); // エラー 1
    let n: i32 = arg.parse().unwrap(); // エラー 2
    println!("{}", 2 * n);
}
```

もし、このプログラムに引数を与えなかったら（エラー 1）、あるいは、最初の引数が整数でなかったら（エラー 2）、このプログラムは、最初の例と同じようにパニックするでしょう。

このようなスタイルのエラーハンドリングは、まるで、陶器店の中を駆け抜ける雄牛のようなものです。雄牛は自分の行きたいところへたどり着くでしょう。でも彼は、途中にある、あらゆるものを蹴散らしてしまいます。

アンラップ (unwrap) とは

先ほどの例で、プログラムが 2 つのエラー条件のいずれかを満たしたときに、パニックすると言いました。でもこのプログラムは、最初の例とは違って明示的に `panic` を呼び出してはいません。 実はパニックは `unwrap` の呼び出しの中に埋め込まれているのです。

Rust でなにかを「アンラップする」とき、こう言っているのと同じです。「計算結果を取り出しなさい。もしエラーになっていたのなら、パニックを起こしてプログラムを終了させなさい。」アンラップのコードはとてもシンプルなので、多分、それを見せたほうが早いでしょう。でもそのためには、まず `Option` と `Result` 型について調べる必要があります。 どちらの型にも `unwrap` という名前のメソッドが定義されています。

■Option 型

Option 型は標準ライブラリで定義されています[17]：

```
enum Option<T> {
    None,
    Some(T),
}
```

Option 型は、Rust の型システムを使って **不在の可能性**を示すためのものです。不在の可能性を型システムにエンコードすることは、重要なコンセプトです。なぜなら、その不在に対処することを、コンパイラがプログラマに強制させるからです。では、文字列から文字を検索する例を見てみましょう。

```
// `haystack`（干し草の山）から Unicode 文字 `needle`（縫い針）を検索します。
// もし見つかったら、文字のバイトオフセットを返します。見つからなければ、`None` を
// 返します。
fn find(haystack: &str, needle: char) -> Option<usize> {
    for (offset, c) in haystack.char_indices() {
        if c == needle {
            return Some(offset);
        }
    }
    None
}
```

この関数がマッチする文字を見つけたとき、単に offset を返すだけではないことに注目してください。その代わりに Some(offset) を返します。 Some は Option 型のヴァリアントの一つ、つまり **値コンストラクタ** です。 これは fn<T>(value: T) -> Option<T> という型の関数だと考えることもできます。 これに対応して None もまた値コンストラクタですが、こちらには引数がありません。 None は fn<T>() -> Option<T> という型の関数だと考えることもできます。

何もないことを表すのに、ずいぶん大げさだと感じるかもしれません。でもこれはまだ、話の半分に過ぎません。 残りの半分は、いま書いた find 関数を **使う** 場面です。これを使って、ファイル名から拡張子を見つけてみましょう。

[17] http://doc.rust-lang.org/std/option/enum.Option.html

```
fn main() {
    let file_name = "foobar.rs";
    match find(file_name, '.') {
        None => println!("No file extension found."),
        Some(i) => println!("File extension: {}", &file_name[i+1..]),
    }
}
```

訳注：

- No file extension found：ファイル拡張子は見つかりませんでした
- File extension: {}：ファイル拡張子：{}

このコードは `find` 関数が返した `Option<usize>` の **場合分け** に、パターンマッチ[*18]を使っています。実のところ、場合分けが、`Option<T>` に格納された値を取り出すための唯一の方法なのです。これは、`Option<T>` が `Some(t)` ではなく `None` だったとき、プログラマであるあなたが、このケースに対処しなければならないことを意味します。

でも、ちょっと待ってください。 さっき使った `unwrap` はどうだったでしょうか？ 場合分けはどこにもありませんでした！ 実は場合分けは `unwrap` メソッドの中に埋め込まれていたのです。もし望むなら、このように自分で定義することもできます：

```
enum Option<T> {
    None,
    Some(T),
}

impl<T> Option<T> {
    fn unwrap(self) -> T {
        match self {
            Option::Some(val) => val,
            Option::None =>
              panic!("called `Option::unwrap()` on a `None` value"),
        }
```

[*18] http://doc.rust-lang.org/book/patterns.html

```
    }
}
```

訳注：
called `Option::unwrap()` on a `None` value：`None` な値に対して `Option:unwrap()` が呼ばれました

`unwrap` メソッドは **場合分けを抽象化します**。このことは確かに `unwrap` をエルゴノミックにしています。しかし残念なことに、そこにある `panic!` が意味するものは、`unwrap` が合成可能ではない、つまり、陶器店の中の雄牛だということです。

■Option<T> 値を合成する

先ほどの例では、ファイル名から拡張子を見つけるために `find` をどのように使うかを見ました。 当然ながら全てのファイル名に `.` があるわけではなく、拡張子のないファイル名もあり得ます。 このような**不在の可能性** は `Option<T>` を使うことによって、型の中にエンコードされています。すなわち、コンパイラは、拡張子が存在しない可能性に対処することを、私たちに強制してくるわけです。今回は単に、そうなったことを告げるメッセージを表示するようにしました。

ファイル名から拡張子を取り出すことは一般的な操作ですので、それを関数にすることは理にかなっています。

```
// 与えられたファイル名の拡張子を返す。拡張子の定義は、最初の
// `.` に続く、全ての文字である。
// もし `file_name` に `.` がなければ、`None` が返される。
fn extension_explicit(file_name: &str) -> Option<&str> {
    match find(file_name, '.') {
        None => None,
        Some(i) => Some(&file_name[i+1..]),
    }
}
```

（プロ向けのヒント：このコードは使わず、代わりに標準ライブラリの `extension`[*19] メソッドを使ってください）

[*19] http://doc.rust-lang.org/std/path/struct.Path.html#method.extension

このコードはいたってシンプルですが、ひとつだけ注目して欲しいのは、find の型が不在の可能性について考慮することを強制していることです。これは良いことです。なぜなら、コンパイラが私たちに、ファイル名が拡張子を持たないケースを、うっかり忘れないようにしてくれるからです。しかし一方で、extension_explicit でしたような明示的な場合分けを毎回続けるのは、なかなか面倒です。

実は extension_explicit での場合分けは、ごく一般的なパターンである、Option<T> への *map* の適用に当てはめられます。 これは、もしオプションが None なら None を返し、そうでなければ、オプションの中の値に関数を適用する、というパターンです。

Rust はパラメトリック多相をサポートしていますので、このパターンを抽象化するためのコンビネータが簡単に定義できます：

```
fn map<F, T, A>(option: Option<T>, f: F) -> Option<A> where F: FnOnce(T) -> A {
    match option {
        None => None,
        Some(value) => Some(f(value)),
    }
}
```

もちろん map は、標準のライブラリの Option<T> でメソッドとして定義されています[*20]。メソッドなので、少し違うシグネチャを持っています。 メソッドは第一引数に self 、 &self あるいは &mut self を取ります。

新しいコンビネータを手に入れましたので、 extension_explicit メソッドを書き直して、場合分けを省きましょう：

```
// 与えられたファイル名の拡張子を返す。拡張子の定義は、最初の
// `.` に続く、全ての文字である。
// もし `file_name` に `.` がなければ、`None` が返される。
fn extension(file_name: &str) -> Option<&str> {
    find(file_name, '.').map(|i| &file_name[i+1..])
}
```

もう一つの共通のパターンは、Option の値が None だったときのデフォルト値を与えることです。例えばファイルの拡張子がないときは、それを rs とみなすようなプログラムを書きたくなるかもしれま

[*20] http://doc.rust-lang.org/std/option/enum.Option.html#method.map

せん。ご想像の通り、このような場合分けはファイルの拡張子に特有のものではありません。どんな Option<T> でも使えるでしょう：

```
fn unwrap_or<T>(option: Option<T>, default: T) -> T {
    match option {
        None => default,
        Some(value) => value,
    }
}
```

上記の map と同じように、標準ライブラリの実装はただの関数ではなくメソッドになっています。

ここでの仕掛けは、Option<T> に入れる値と同じ型になるよう、デフォルト値の型を制限していることです。これを使うのは、すごく簡単です：

```
fn main() {
    assert_eq!(extension("foobar.csv").unwrap_or("rs"), "csv");
    assert_eq!(extension("foobar").unwrap_or("rs"), "rs");
}
```

（unwrap_or は、標準のライブラリの Option<T> で、メソッドとして定義されています[*21]ので、いま定義したフリースタンディングな関数の代わりに、そちらを使いましょう。）

もうひとつ特筆すべきコンビネータがあります。それは and_then です。これを使うと **不在の可能性**を考慮しながら、別々の処理を簡単に組み合わせることができます。例えば、この節のほとんどのコードは、与えられたファイル名について拡張子を見つけだします。そのためには、まずファイル **パス**から取り出したファイル名が必要です。大抵のパスにはファイル名がありますが、 **全て**がというわけではありません。 例えば ., .., / などは例外です。

つまり、与えられたファイル **パス**から拡張子を見つけ出せるか、トライしなければなりません。まず明示的な場合分けから始めましょう：

```
fn file_path_ext_explicit(file_path: &str) -> Option<&str> {
    match file_name(file_path) {
        None => None,
```

[*21] http://doc.rust-lang.org/std/option/enum.Option.html#method.unwrap_or

```
        Some(name) => match extension(name) {
            None => None,
            Some(ext) => Some(ext),
        }
    }
}

fn file_name(file_path: &str) -> Option<&str> {
  // 実装は省略
  unimplemented!()
}
```

場合分けを減らすために単に map コンビネータを使えばいいと思うかもしれませんが、型にうまく適合しません。

```
fn file_path_ext(file_path: &str) -> Option<&str> {
    file_name(file_path).map(|x| extension(x)) //Compilation error
}
```

ここの map 関数は Option<_> 内で extension 関数が返した値をラップしていますが extension 関数自身が Option<&str> を返すので、式 file_name(file_path).map(|x| extension(x)) は実際は Option<Option<&str» を返すのです。

しかしながら file_path_ext は（Option<Option<&str» ではなく）ただの Option<&str> を返すのでコンパイルエラーとなるのです。

そして関数が返した値は **必ず** Some でラップされ直します 。つまりこの代わりに、 map に似た、しかし新たに Option<_> で包まずに直接呼び出し元に常に Option<_> を返すものが必要です。

これの汎用的な実装は map よりもシンプルです：

```
fn and_then<F, T, A>(option: Option<T>, f: F) -> Option<A>
        where F: FnOnce(T) -> Option<A> {
    match option {
        None => None,
        Some(value) => f(value),
```

```
    }
}
```

では、明示的な場合分けを省くように、 file_path_ext を書き直しましょう：

```
fn file_path_ext(file_path: &str) -> Option<&str> {
    file_name(file_path).and_then(extension)
}
```

補足: and_then は基本的に map のように振舞いますが Option<Option_» の代わりに Option<_> を返すので flatmap と呼ぶ言語もあります。

Option 型には、他にもたくさんのコンビネータが標準ライブラリで定義されています[*22]。 それらの一覧をざっと眺めて、なにがあるか知っておくといいでしょう。大抵の場合、場合分けを減らすのに役立ちます。それらのコンビネータに慣れるための努力は、すぐに報われるでしょう。なぜなら、そのほとんどは次に話す Result 型でも、（よく似たセマンティクスで）定義されているからです。

コンビネータは明示的な場合分けを減らしてくれるので、 Option のような型をエルゴノミックにします。 またこれらは **不在の可能性**を、呼び出し元がそれに合った方法で扱えるようにするので、合成可能だといえます。unwrap のようなメソッドは、Option<T> が None のときにパニックを起こすので、このような選択の機会を与えません。

Result 型

Result 型も標準ライブラリで定義されています[*23]。

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Result 型は Option 型の豪華版です。 Option のように **不在** の可能性を示す代わりに、Result は**エラー**になる可能性を示します。 通常 **エラー**は、なぜ処理が実行に失敗したのかを説明するために用いられま

[*22] http://doc.rust-lang.org/std/option/enum.Option.html
[*23] http://doc.rust-lang.org/std/result/

す。これは厳密には Option をさらに一般化した形式だといえます。以下のような型エイリアスがあるとしましょう。 これは全てにおいて、本物の Option<T> と等しいセマンティクスを持ちます。

```
type Option<T> = Result<T, ()>;
```

これは Result の 2 番目の型パラメータを () （「ユニット」または「空タプル」と発音します）に固定したものです。() 型のただ一つの値は () です。（そうなんです。型レベルと値レベルの項が、全く同じ表記法を持ちます!）

Result 型は、処理の結果がとりうる 2 つの可能性のうち、1 つを表すための方法です。慣例に従い、一方が期待されている結果、つまり「Ok」となり、もう一方が予想外の結果、つまり「Err」になります。

Option と全く同じように、Result 型も標準ライブラリで unwrap メソッドが定義されています[*24] 。 定義してみましょう：

```
impl<T, E: ::std::fmt::Debug> Result<T, E> {
    fn unwrap(self) -> T {
        match self {
            Result::Ok(val) => val,
            Result::Err(err) =>
              panic!("called `Result::unwrap()` on an `Err` value: {:?}", err),
        }
    }
}
```

訳注：
called Result::unwrap() on an Err value: {:?}“：Err 値 {:?} に対して Result::unwrap() が呼ばれました

これは実質的には私たちの Option::unwrap の定義と同じですが、 panic! メッセージにエラーの値が含まれているところが異なります。これはデバッグをより簡単にしますが、一方で、（エラーの型を表す）型パラメータ E に Debug[*25] 制約を付けることが求められます。 大半の型は Debug 制約を満たしているので、実際のところ、うまくいく傾向にあります。（Debug が型に付くということは、単にその型の値が、人間が読める形式で表示できることを意味しています。）

[*24] http://doc.rust-lang.org/std/result/enum.Result.html#method.unwrap
[*25] http://doc.rust-lang.org/std/fmt/trait.Debug.html

では、例を見ていきましょう。

■整数をパースする

Rust の標準ライブラリを使うと、文字列を整数に変換することが、すごく簡単にできます。あまりにも簡単なので、実際のところ、以下のように書きたいという誘惑に負けることがあります：

```
fn double_number(number_str: &str) -> i32 {
    2 * number_str.parse::<i32>().unwrap()
}

fn main() {
    let n: i32 = double_number("10");
    assert_eq!(n, 20);
}
```

すでにあなたは、unwrap を呼ぶことについて懐疑的になっているはずです。例えば、文字列が数字としてパースできなければ、パニックが起こります。

```
thread '<main>' panicked at 'called `Result::unwrap()` on an `Err` value: ParseIntErro
   r { kind: InvalidDigit }', /home/rustbuild/src/rust-buildbot/slave/beta-dist-rustc-l
   inux/build/src/libcore/result.rs:729
```

これは少し目障りです。もしあなたが使っているライブラリの中でこれが起こされたら、イライラするに違いありません。代わりに、私たちの関数の中でエラーを処理し、呼び出し元にどうするのかを決めさせるべきです。そのためには、double_number の戻り値の型（リターン型）を変更しなければなりません。 でも、一体何に？ ええと、これはつまり、標準ライブラリの parse メソッド[*26] のシグネチャを見ろということです。

```
impl str {
    fn parse<F: FromStr>(&self) -> Result<F, F::Err>;
}
```

うむ。最低でも Result を使わないといけないことはわかりました。もちろん、これが Option を戻すよ

[*26] http://doc.rust-lang.org/std/primitive.str.html#method.parse

うにすることも可能だったでしょう。結局のところ、文字列が数字としてパースできたかどうかが知りたいわけですよね？ それも悪いやり方ではありませんが、実装の内側では なぜ文字列が整数としてパースできなかったを、ちゃんと区別しています。（空の文字列だったのか、有効な数字でなかったのか、大きすぎたり、小さすぎたりしたのか。）従って、`Result` を使ってより多くの情報を提供するほうが、単に「不在」を示すことよりも理にかなっています。今後、もし `Option` と `Result` のどちらを選ぶという事態に遭遇したときは、このような理由付けのやり方を真似てみてください。もし詳細なエラー情報を提供できるのなら、多分、それをしたほうがいいでしょう。（後ほど別の例もお見せます。）

それでは、リターン型をどう書きましょうか？ 上の `parse` メソッドは一般化されているので、標準ライブラリにある、あらゆる数値型について定義されています。この関数を同じように一般化することもできますが（そして、そうするべきでしょうが）、今は明快さを優先しましょう。`i32` だけを扱うことにしますので、それの `FromStr` の実装がどうなっているか探しましょう[*27] 。 （ブラウザで `CTRL-F` を押して「FromStr」を探します。） そして関連型 (associated type)[*28] から `Err` を見つけます。こうすれば、具体的なエラー型が見つかります。 この場合、それは `std::num::ParseIntError`[*29] です。 これでようやく関数を書き直せます：

```
use std::num::ParseIntError;

fn double_number(number_str: &str) -> Result<i32, ParseIntError> {
    match number_str.parse::<i32>() {
        Ok(n) => Ok(2 * n),
        Err(err) => Err(err),
    }
}

fn main() {
    match double_number("10") {
        Ok(n) => assert_eq!(n, 20),
        Err(err) => println!("Error: {:?}", err),
    }
}
```

[*27] http://doc.rust-lang.org/std/primitive.i32.html
[*28] http://doc.rust-lang.org/book/associated-types.html
[*29] http://doc.rust-lang.org/std/num/struct.ParseIntError.html

これで少し良くなりましたが、たくさんのコードを書いてしまいました！ 場合分けに、またしてもやられたわけです。

コンビネータに助けを求めましょう！ ちょうど Option と同じように Result にもたくさんのコンビネータが、メソッドとして定義されています。Result と Option の間では、共通のコンビネータが数多く存在します。 例えば map も共通なものの一つです：

```
use std::num::ParseIntError;

fn double_number(number_str: &str) -> Result<i32, ParseIntError> {
    number_str.parse::<i32>().map(|n| 2 * n)
}

fn main() {
    match double_number("10") {
        Ok(n) => assert_eq!(n, 20),
        Err(err) => println!("Error: {:?}", err),
    }
}
```

Result でいつも候補にあがるのは unwrap_or[*30] と and_then[*31] です。 さらに Result は 2 つ目の型パラメータを取りますので、エラー型だけに影響を与える map_err[*32] （map に相当）と or_else[*33]（and_then に相当）もあります。

■Result 型エイリアスを用いたイディオム

標準ライブラリでは Result<i32> のような型をよく見ると思います。 でも、待ってください。2 つの型パラメータを取るように Result を定義したはずです 。 どうして、1 つだけを指定して済んだのでしょう？ 種を明かすと、Result の型エイリアスを定義して、一方の型パラメータを特定の型に 固定したのです。 通常はエラー型の方を固定します。例えば、先ほどの整数のパースの例は、こう書き換えることもできます。

[*30] http://doc.rust-lang.org/std/result/enum.Result.html#method.unwrap_or
[*31] http://doc.rust-lang.org/std/result/enum.Result.html#method.and_then
[*32] http://doc.rust-lang.org/std/result/enum.Result.html#method.map_err
[*33] http://doc.rust-lang.org/std/result/enum.Result.html#method.or_else

```
use std::num::ParseIntError;
use std::result;

type Result<T> = result::Result<T, ParseIntError>;

fn double_number(number_str: &str) -> Result<i32> {
    unimplemented!();
}
```

なぜ、こうするのでしょうか？ もし ParseIntError を返す関数をたくさん定義するとしたら、常に ParseIntError を使うエイリアスを定義したほうが便利だからです。こうすれば、同じことを何度も書かずに済みます。

標準ライブラリで、このイディオムが際立って多く使われている場所では、io::Result[*34] を用いています。 それらは通常 io::Result<T> のように書かれ、std::result のプレーンな定義の代わりに io モジュールの型エイリアスを使っていることが、明確にわかるようになっています。

小休止：アンラップは悪ではない

これまでの説明を読んだあなたは、 unwrap のような panic を起こし、プログラムをアボートするようなメソッドについて、私がきっぱりと否定する方針をとっていたことに気づいたかもしれません。**一般的には** これは良いアドバイスです。

しかしながら unwrap を使うのが賢明なこともあります。どんな場合に unwrap の使用を正当化できるのかについては、グレーな部分があり、人によって意見が分かれます。ここで、この問題についての、私の**個人的な意見**をまとめたいと思います。

- **即興で書いたサンプルコード。**サンプルコードや簡単なプログラムを書いていて、エラーハンドリングが単に重要でないこともあります。このようなときに unwrap の便利さは、とても魅力的に映るでしょう。これに打ち勝つのは難しいことです。

- **パニックがプログラムのバグの兆候となるとき。**コードの中の不変条件が、ある特定のケースの発生を未然に防ぐとき（例えば、空のスタックから取り出そうとしたなど）、パニックを起こしても差し支えありません。なぜなら、そうすることでプログラムに潜むバグが明るみに出るからで

[*34] http://doc.rust-lang.org/std/io/type.Result.html

> す。これは assert! の失敗のような明示的な要因によるものだったり、配列のインデックスが境界から外れたからだったりします。

これは多分、完全なリストではないでしょう。さらに Option を使うときは、ほとんどの場合で expect[*35] メソッドを使う方がいいでしょう。 expect は unwrap とほぼ同じことをしますが、 expect では与えられたメッセージを表示するところが異なります。この方が結果として起こったパニックを、少し扱いやすいものにします。なぜなら「 None な値に対してアンラップが呼ばれました」というメッセージの代わりに、指定したメッセージが表示されるからです。

私のアドバイスを突き詰めると、よく見極めなさい、ということです。私の書いた文章の中に「決して、X をしてはならない」とか「Y は有害だと考えよう」といった言葉が現れないのには、れっきとした理由があります。あるユースケースでこれが容認できるかどうかは、プログラマであるあなたの判断に委ねられます。私が目指していることは、あなたがトレードオフをできるかぎり正確に評価できるよう、手助けをすることなのです。

これで Rust におけるエラーハンドリングの基礎をカバーできました。また、アンラップについても解説しました。では標準ライブラリをもっと探索していきましょう。

複数のエラー型を扱う

これまで見てきたエラーハンドリングでは、Option<T> または Result<T, SomeError> が 1 つあるだけでした。 ではもし Option と Result の両方があったらどうなるでしょうか？ あるいは、Result<T, Error1> と Result<T, Error2> があったら？ **異なるエラー型の組み合わせ**を扱うことが、いま目の前にある次なる課題です。またこれが、このセクションの残りの大半に共通する、主要なテーマとなります。

Option と Result を合成する

これまで話してきたのは Option のために定義されたコンビネータと、 Result のために定義されたコンビネータについてでした。これらのコンビネータを使うと、様々な処理の結果を明示的な場合分けなしに組み合わせることができました。

もちろん現実のコードは、いつもこんなにクリーンではありません。 時には Option 型と Result 型が混在していることもあるでしょう。そんなときは、明示的な場合分けに頼るしかないのでしょうか？ それとも、コンビネータを使い続けることができるのでしょうか？

ここで、このセクションの最初の方にあった例に戻ってみましょう：

[*35] http://doc.rust-lang.org/std/option/enum.Option.html#method.expect

```
use std::env;

fn main() {
    let mut argv = env::args();
    let arg: String = argv.nth(1).unwrap(); // エラー 1
    let n: i32 = arg.parse().unwrap(); // エラー 2
    println!("{}", 2 * n);
}
```

これまでに獲得した知識、つまり Option 、Result と、それらのコンビネータに関する知識を動員して、これを書き換えましょう。エラーを適切に処理し、もしエラーが起こっても、プログラムがパニックしないようにするのです。

ここでの問題は argv.nth(1) が Option を返すのに、arg.parse() は Result を返すことです。これらを直接合成することはできません。 Option と Result の両方に出会ったときの **通常の** 解決策は Option を Result に変換することです。この例で（env::args() が）コマンドライン引数を返さなかったということは、ユーザーがプログラムを正しく起動しなかったことを意味します。エラーの理由を示すために、String を使うこともできます。試してみましょう：

```
use std::env;

fn double_arg(mut argv: env::Args) -> Result<i32, String> {
    argv.nth(1)
        .ok_or("Please give at least one argument".to_owned())
        .and_then(|arg| arg.parse::<i32>().map_err(|err| err.to_string()))
        .map(|n| 2 * n)
}

fn main() {
    match double_arg(env::args()) {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Error: {}", err),
    }
}
```

訳注：

Please give at least one argument：引数を最低 1 つ指定してください。

この例では、いくつか新しいことがあります。 ひとつ目は `Option::ok_or`[*36] コンビネータを使ったことです。 これは `Option` を `Result` へ変換する方法の一つです。 変換には `Option` が `None` のときに使われるエラーを指定する必要があります。他のコンビネータと同様に、その定義はとてもシンプルです：

```
fn ok_or<T, E>(option: Option<T>, err: E) -> Result<T, E> {
    match option {
        Some(val) => Ok(val),
        None => Err(err),
    }
}
```

ここで使った、もう一つの新しいコンビネータは `Result::map_err`[*37] です。 これは `Result::map` に似ていますが、 `Result` 値の**エラー** の部分に対して関数をマップするところが異なります。 もし `Result` の値が `Ok(...)` だったら、そのまま変更せずに返します。

`map_err` を使った理由は、（`and_then` の用法により）エラーの型を同じに保つ必要があったからです。ここでは（`argv.nth(1)` が返した）`Option<String>` を `Result<String, String>` に変換することを選んだため、`arg.parse()` が返した `ParseIntError` も `String` に変換しなければならなかったわけです。

コンビネータの限界

入出力と共に入力をパースすることは、非常によく行われます。そして私が Rust を使って個人的にやってきたことのほとんども、これに該当しています。ですから、ここでは（そして、この後も）IO と様々なパースを行うルーチンを、エラーハンドリングの例として扱っていきます。

まずは簡単なものから始めましょう。ここでのタスクは、ファイルを開き、その内容を全て読み込み、1 つの数値に変換することです。そしてそれに `2` を掛けて、結果を表示します。

いままで `unwrap` を使わないよう説得してきたわけですが、最初にコードを書くときには `unwrap` が便利に使えます。こうすることで、エラーハンドリングではなく、本来解決すべき課題に集中できます。それと同時に `unwrap` は、適切なエラーハンドリングが必要とされる場所を教えてくれます。ここから始

[*36] http://doc.rust-lang.org/std/option/enum.Option.html#method.ok_or
[*37] http://doc.rust-lang.org/std/result/enum.Result.html#method.map_err

めることをコーディングへの取っ掛かりとしましょう。その後、リファクタリングによって、エラーハンドリングを改善していきます。

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> i32 {
    let mut file = File::open(file_path).unwrap(); // エラー 1
    let mut contents = String::new();
    file.read_to_string(&mut contents).unwrap(); // エラー 2
    let n: i32 = contents.trim().parse().unwrap(); // エラー 3
    2 * n
}

fn main() {
    let doubled = file_double("foobar");
    println!("{}", doubled);
}
```

（備考： `AsRef<Path>` を使ったのは、`std::fs::File::open` で使われているものと同じ境界[*38] だからです。ファイルパスとして、どんな文字列でも受け付けるので、エルゴノミックになります。）

ここでは 3 種類のエラーが起こる可能性があります：

1. ファイルを開くときの問題
2. ファイルからデータを読み込むときの問題
3. データを数値としてパースするときの問題

最初の 2 つの問題は、`std::io::Error`[*39] 型で記述されます。 これは `std::fs::File::open`[*40] と `std::io::Read::read_to_string`[*41] のリターン型からわかります。（ちなみにどちらも、以前紹介した `Result` 型エイリアスのイディオム を用いています。 `Result` 型のところをクリックすると、いま言った

[*38] http://doc.rust-lang.org/std/fs/struct.File.html#method.open
[*39] http://doc.rust-lang.org/std/io/struct.Error.html
[*40] http://doc.rust-lang.org/std/fs/struct.File.html#method.open
[*41] http://doc.rust-lang.org/std/io/trait.Read.html#method.read_to_string

型エイリアスを見たり[*42]、必然的に、中で使われている `io::Error` 型も見ることになるでしょう。） 3 番目の問題は `std::num::ParseIntError`[*43] 型で記述されます。 特にこの `io::Error` 型は標準ライブラリ全体に**深く浸透しています** 。これからこの型を幾度となく見ることでしょう。

まず最初に `file_double` 関数をリファクタリングしましょう。この関数を、このプログラムの他の構成要素と合成可能にするためには、上記の問題のいずれかに遭遇しても、パニック**しない** ようにしなければなりません。これは実質的には、なにかの操作に失敗したときに、この関数が**エラーを返すべき** であることを意味します。ここでの問題は、`file_double` のリターン型が `i32` であるため、エラーの報告には全く役立たないことです。 従ってリターン型を `i32` から別の何かに変えることから始めましょう。

最初に決めるべきことは、 `Option` と `Result` のどちらを使うかです。 `Option` なら間違いなく簡単に使えます。もし 3 つのエラーのどれかが起こったら、単に `None` を返せばいいのですから。 これはたしかに動きますし、**パニックを起こすよりは良くなっています** 。とはいえ、もっと良くすることもできます。`Option` の代わりに、発生したエラーについての詳細を渡すべきでしょう。 ここでは**エラーの可能性**を示したいのですから、`Result<i32, E>` を使うのがよさそうです。 でも `E` を何にしたらいいのでしょうか？ 2 つの **異なる**型のエラーが起こり得ますので、これらを共通の型に変換する必要があります。そのような型の一つに `String` があります。この変更がコードにどんな影響を与えるか見てみましょう：

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, String> {
    File::open(file_path)
         .map_err(|err| err.to_string())
         .and_then(|mut file| {
              let mut contents = String::new();
              file.read_to_string(&mut contents)
                  .map_err(|err| err.to_string())
                  .map(|_| contents)
         })
         .and_then(|contents| {
              contents.trim().parse::<i32>()
                      .map_err(|err| err.to_string())
```

[*42] http://doc.rust-lang.org/std/io/type.Result.html
[*43] http://doc.rust-lang.org/std/num/struct.ParseIntError.html

```
        })
        .map(|n| 2 * n)
}

fn main() {
    match file_double("foobar") {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Error: {}", err),
    }
}
```

このコードは、やや難解になってきました。このようなコードを簡単に書けるようになるまでには、結構な量の練習が必要かもしれません。こういうものを書くときは **型に導かれる** ようにします。`file_double` のリターン型を `Result<i32, String>` に変更したらすぐに、それに合ったコンビネータを探し始めるのです。この例では `and_then`, `map`, `map_err` の、3 種類のコンビネータだけを使いました。

`and_then` は、エラーを返すかもしれない処理同士を繋いでいくために使います。ファイルを開いた後に、失敗するかもしれない処理が 2 つあります：ファイルから読み込む所と、内容を数値としてパースする所です。これに対応して `and_then` も 2 回呼ばれています。

`map` は `Result` の値が `Ok(...)` のときに関数を適用するために使います。 例えば、一番最後の `map` の呼び出しは、`Ok(...)` の値（ `i32` 型）に `2` を掛けます。 もし、これより前にエラーが起きたなら、この操作は `map` の定義に従ってスキップされます。

`map_err` は全体をうまく動かすための仕掛けです。`map_err` は `map` に似ていますが、 `Result` の値が `Err(...)` のときに関数を適用するところが異なります。今回の場合は、全てのエラーを `String` という同一の型に変換する予定でした。 `io::Error` と `num::ParseIntError` の両方が `ToString` を実装していたので、 `to_string()` メソッドを呼ぶことで変換できました。

説明し終わった後でも、このコードは難解なままです。コンビネータの使い方をマスタすることは重要ですが、コンビネータには限界もあるのです。次は、早期リターンと呼ばれる、別のアプローチを試してみましょう。

早期リターン

前の節で使ったコードを、 **早期リターン**を使って書き直してみようと思います。早期リターンとは、関数の途中で抜けることを指します。`file_double` のクロージャの中にいる間は、早期リターンはできな

いので、明示的な場合分けまでいったん戻る必要があります。

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, String> {
    let mut file = match File::open(file_path) {
        Ok(file) => file,
        Err(err) => return Err(err.to_string()),
    };
    let mut contents = String::new();
    if let Err(err) = file.read_to_string(&mut contents) {
        return Err(err.to_string());
    }
    let n: i32 = match contents.trim().parse() {
        Ok(n) => n,
        Err(err) => return Err(err.to_string()),
    };
    Ok(2 * n)
}

fn main() {
    match file_double("foobar") {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Error: {}", err),
    }
}
```

このコードが、コンビネータを使ったコードよりも良くなったのかについては、人によって意見が分かれるでしょう。でも、もしあなたがコンビネータによるアプローチに不慣れだったら、このコードのほうが読みやすいと思うかもしれません。ここでは明示的な場合分けを `match` と `if let` で行っています。もしエラーが起きたら関数の実行を打ち切って、エラーを（文字列に変換してから）返します。

でもこれって逆戻りしてませんか？ 以前は、エラーハンドリングをエルゴノミックにするために、明示

的な場合分けを減らすべきだと言っていました。それなのに、今は明示的な場合分けに戻ってしまっています。すぐにわかりますが、明示的な場合分けを減らす方法は **複数**あるのです。 コンビネータが唯一の方法ではありません。

try! マクロ

Rust でのエラー処理の基礎となるのは try! マクロです。try! マクロはコンビネータと同様、場合分けを抽象化します。しかし、コンビネータと異なるのは **制御フロー**も抽象化してくれることです。 つまり、先ほど見た **早期リターン**のパターンを抽象化できるのです。

try! マクロの簡略化した定義はこうなります：

```
macro_rules! try {
    ($e:expr) => (match $e {
        Ok(val) => val,
        Err(err) => return Err(err),
    });
}
```

（本当の定義[*44]はもっと洗練されています。 後ほど紹介します。）

try! マクロを使うと、最後の例をシンプルにすることが、とても簡単にできます。場合分けと早期リターンを肩代わりしてくれますので、コードが締まって読みやすくなります。

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, String> {
    let mut file = try!(File::open(file_path).map_err(|e| e.to_string()));
    let mut contents = String::new();
    try!(file.read_to_string(&mut contents).map_err(|e| e.to_string()));
    let n = try!(contents.trim().parse::<i32>().map_err(|e| e.to_string()));
    Ok(2 * n)
}
```

[*44] http://doc.rust-lang.org/std/macro.try!.html

```
fn main() {
    match file_double("foobar") {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Error: {}", err),
    }
}
```

今の私たちの try! の定義ですと、 map_err は今でも必要です。 なぜなら、エラー型を String に変換しなければならないからです。でも、いい知らせがあります。 map_err の呼び出しを省く方法をすぐに習うのです！ 悪い知らせは、map_err を省く前に、標準ライブラリのいくつかの重要なトレイトについて、もう少し学ぶ必要があるということです。

独自のエラー型を定義する

標準ライブラリのいくつかのエラートレイトについて学ぶ前に、これまでの例にあったエラー型における String の使用を取り除くことで、この節を締めくくりたいと思います。

これまでの例では String を便利に使ってきました。なぜなら、エラーは簡単に文字列へ変換できますし、問題が起こったその場で、文字列によるエラーを新たに作ることもできるからです。しかし String を使ってエラーを表すことには欠点もあります。

ひとつ目の欠点は、エラーメッセージがコードのあちこちに散らかる傾向があることです。エラーメッセージをどこか別の場所でまとめて定義することもできますが、特別に訓練された人でない限りは、エラーメッセージをコードに埋め込むことへの誘惑に負けてしまうでしょう。実際、私たちは 以前の例でも、その通りのことをしました。

ふたつ目の、もっと重大な欠点は、 String への変換で**情報が欠落する** ことです。もし全てのエラーを文字列に変換してしまったら、呼び出し元に渡したエラーが、オペーク（不透明）になってしまいます。呼び出し元が String のエラーに対してできる唯一妥当なことは、それをユーザーに表示することだけです。文字列を解析して、どのタイプのエラーだったか判断するのは、もちろん強固なやり方とはいえません。（この問題は、ライブラリの中の方が、他のアプリケーションのようなものよりも、間違いなく重大なものになるでしょう。）

例えば io::Error 型には io::ErrorKind[*45] が埋め込まれます。 これは **構造化されたデータ**で、IO 操作において何が失敗したのかを示します。エラーによって違った対応を取りたいこともあるので、この

[*45] http://doc.rust-lang.org/std/io/enum.ErrorKind.html

ことは重要です。（例： あなたのアプリケーションでは `BrokenPipe` エラーは正規の手順を踏んだ終了を意味し、 `NotFound` エラーはエラーコードと共に異常終了して、ユーザーにエラーを表示することを意味するかもしれません。）`io::ErrorKind` なら、呼び出し元でエラーの種類を調査するために、場合分けが使えます。これは `String` の中からエラーの詳細がなんだったのか探りだすことよりも、明らかに優れています。

ファイルから整数値を取り出す例で `String` をエラー型として用いた代わりに、独自のエラー型を定義し、**構造化されたデータ** によってエラー内容を表すことができます。呼び出し元が詳細を検査したいときに備え、大元のエラーについての情報を取りこぼさないよう、努力してみましょう。

多くの可能性のうちの一つ を表す理想的な方法は、 `enum` を使って独自の直和型を定義することです。このケースでは、エラーは `io::Error` もしくは `num::ParseIntError` でした。ここから思い浮かぶ自然な定義は：

```
use std::io;
use std::num;

// 全ての型は `Debug` を導出するべきでしょうから、ここでも `Debug` を導出します。
// これにより `CliError` 値について、人間が十分理解できる説明を得られます。
#[derive(Debug)]
enum CliError {
    Io(io::Error),
    Parse(num::ParseIntError),
}
```

コードの微調整はいとも簡単です。エラーを文字列に変換する代わりに、エラーに対応する値コンストラクタを用いて `CliError` 型に変換すればいいのです：

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, CliError> {
    let mut file = try!(File::open(file_path).map_err(CliError::Io));
    let mut contents = String::new();
    try!(file.read_to_string(&mut contents).map_err(CliError::Io));
```

```
    let n: i32 = try!(contents.trim().parse().map_err(CliError::Parse));
    Ok(2 * n)
}

fn main() {
    match file_double("foobar") {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Error: {:?}", err),
    }
}
```

ここでの変更点は、（エラーを文字列に変換する）`map_err(|e| e.to_string())` を、`map_err(CliError::Io)` や `map_err(CliError::Parse)` へ切り替えたことです。 こうして**呼び出し元**が、ユーザーに対してどの程度の詳細を報告するか決められるようになりました。`String` をエラー型として用いることは、事実上、呼び出し元からこうした選択肢を奪ってしまいます。`CliError` のような独自の `enum` エラー型を用いることは、**構造化されたデータ**によるエラーの説明だけでなく、これまでと同様の使いやすさをもたらします。

目安となる方法は独自のエラー型を定義することですが、 `String` エラー型も、いざというときに役立ちます。特にアプリケーションを書いているときなどはそうです。もしライブラリを書いているのなら、呼び出し元の選択肢を理由もなく奪わないために、独自のエラー型を定義することを強く推奨します。

標準ライブラリのトレイトによるエラー処理

標準ライブラリでは、エラーハンドリングに欠かせないトレイトが、2 つ定義されています：`std::error::Error`[*46] と `std::convert::From`[*47] です。 `Error` はエラーを総称的に説明することを目的に設計されているのに対し、`From` トレイトはもっと汎用的な、2 つの異なる型の間で値を変換する役割を担います。

`Error` トレイト

`Error` トレイトは標準ライブラリで定義されています[*48]：

[*46] http://doc.rust-lang.org/std/error/trait.Error.html
[*47] http://doc.rust-lang.org/std/convert/trait.From.html
[*48] http://doc.rust-lang.org/std/error/trait.Error.html

```
use std::fmt::{Debug, Display};

trait Error: Debug + Display {
  /// エラーの簡単な説明
  fn description(&self) -> &str;

  /// このエラーの一段下のレベルの原因（もしあれば）
  fn cause(&self) -> Option<&Error> { None }
}
```

このトレイトは極めて一般的です。 なぜなら、エラーを表す 全ての型で実装されることを目的としているからです。この後すぐ見るように、このことは合成可能なコードを書くのに間違いなく役立ちます。それ以外にも、このトレイトでは最低でも以下のようなことができます：

- エラーの Debug 表現を取得する。
- エラーのユーザー向け Display 表現を取得する。
- エラーの簡単な説明を取得する（description メソッド経由）。
- エラーの因果関係のチェーンが提供されているなら、それを調べる（cause メソッド経由）。

最初の 2 つは Error が Debug と Display の実装を必要としていることに由来します。 残りの 2 つは Error が定義している 2 つのメソッドに由来します。 Error の強力さは、実際に全てのエラー型が Error を実装していることから来ています。 このことは、全てのエラーを 1 つのトレイトオブジェクト[*49]として存在量化 (existentially quantify) できることを意味します。 これは Box<Error> または &Error と書くことで表明できます。 実際に cause メソッドは&Error を返し、これ自体はトレイトオブジェクトです。Error トレイトのトレイトオブジェクトとしての用例については、後ほど再び取りあげます。

Error トレイトの実装例を見せるには、いまはこのくらいで十分でしょう。前の節で定義したエラー型を使ってみましょう：

```
use std::io;
use std::num;

// 全ての型は `Debug` を導出するべきでしょうから、ここでも `Debug` を導出します。
// これにより `CliError` 値について、人間が十分理解できる説明を得られます。
```

[*49] http://doc.rust-lang.org/book/trait-objects.html

```
#[derive(Debug)]
enum CliError {
    Io(io::Error),
    Parse(num::ParseIntError),
}
```

このエラー型は 2 種類のエラー、つまり、IO を扱っているときのエラー、または、文字列を数値に変換するときのエラーが起こる可能性を示しています。enum 定義のヴァリアントを増やせば、エラーの種類をいくらでも表現できます。

Error を実装するのは実に単純な作業です。大抵は明示的な場合分けの繰り返しになります。

```
use std::error;
use std::fmt;

impl fmt::Display for CliError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match *self {
            // 下層のエラーは両方ともすでに `Display` を実装しているので、
            // それらの実装に従います。
            CliError::Io(ref err) => write!(f, "IO error: {}", err),
            CliError::Parse(ref err) => write!(f, "Parse error: {}", err),
        }
    }
}

impl error::Error for CliError {
    fn description(&self) -> &str {
        // 下層のエラーは両方ともすでに `Error` を実装しているので、
        // それらの実装に従います。
        match *self {
            CliError::Io(ref err) => err.description(),
            CliError::Parse(ref err) => err.description(),
        }
    }
```

```
    fn cause(&self) -> Option<&error::Error> {
        match *self {
            // 注意：これらは両方とも `err` を、その具象型（`&io::Error` か
            // `&num::ParseIntError` のいずれか）から、トレイトオブジェクト
            // `&Error` へ暗黙的にキャストします。どちらのエラー型も `Error` を
            // 実装しているので、問題なく動きます。
            CliError::Io(ref err) => Some(err),
            CliError::Parse(ref err) => Some(err),
        }
    }
}
```

これは極めて典型的な Error の実装だということに留意してください。このように、それぞれのエラー型にマッチさせて、description と cause のコントラクトを満たします。

From トレイト

std::convert::From は標準ライブラリで定義されています[*50]：

```
trait From<T> {
    fn from(T) -> Self;
}
```

嬉しいくらい簡単でしょ？ From は、ある特定の T という型 から、別の型へ変換するための汎用的な方法を提供するので大変便利です（この場合の「別の型」とは実装の主体、つまり Self です）。From を支えているのは標準ライブラリで提供される一連の実装です[*51]。

From がどのように動くか、いくつかの例を使って紹介しましょう：

```
let string: String = From::from("foo");
let bytes: Vec<u8> = From::from("foo");
let cow: ::std::borrow::Cow<str> = From::from("foo");
```

[*50] http://doc.rust-lang.org/std/convert/trait.From.html
[*51] http://doc.rust-lang.org/std/convert/trait.From.html

たしかに From が文字列を変換するのに便利なことはわかりました。でもエラーについてはどうでしょうか？ 結論から言うと、これが重要な実装です：

```
impl<'a, E: Error + 'a> From<E> for Box<Error + 'a>
```

この実装では、 Error を実装した **全て**の型は、トレイトオブジェクト Box<Error> に変換できると言っています。これは、驚きに値するものには見えないかもしれませんが、一般的なコンテキストで有用なのです。

さっき扱った 2 つのエラーを覚えてますか？ 具体的には io::Error と num::ParseIntError でした。どちらも Error を実装していますので From で動きます。

```
use std::error::Error;
use std::fs;
use std::io;
use std::num;

// エラーの値にたどり着くまで、何段階かのステップが必要です。
let io_err: io::Error = io::Error::last_os_error();
let parse_err: num::ParseIntError = "not a number".parse::<i32>().unwrap_err();

// では、こちらで変換します。
let err1: Box<Error> = From::from(io_err);
let err2: Box<Error> = From::from(parse_err);
```

ここに気づくべき、本当に重要なパターンがあります。 err1 と err2 の両方ともが **同じ型** になっているのです。なぜなら、それらが存在量化型、つまり、トレイトオブジェクトだからです。特にそれらの背後の型は、コンパイラの知識から **消去されます** ので、err1 と err2 が本当に同じに見えるのです。さらに私たちは同じ関数呼び出し From::from を使って err1 と err2 をコンストラクトしました。 これは From::from が引数とリターン型の両方でオーバーロードされているからです。

このパターンは重要です。なぜなら、私たちが前から抱えていた問題を解決するからです：同じ関数を使って、エラーを同一の型に変換する、確かな方法を提供するからです。

いよいよ、私たちの旧友 try! マクロを再訪するときが訪れました。

本当の try! マクロ

try! の定義は、以前このように提示されました：

```
macro_rules! try {
    ($e:expr) => (match $e {
        Ok(val) => val,
        Err(err) => return Err(err),
    });
}
```

これは本当の定義ではありません。 本当の定義は標準ライブラリの中にあります[*52]：

```
macro_rules! try {
    ($e:expr) => (match $e {
        Ok(val) => val,
        Err(err) => return Err(::std::convert::From::from(err)),
    });
}
```

文面上はわずかですが、非常に大きな違いがあります： エラーの値は From::from を経て渡されるのです。 これにより try! マクロは、はるかに強力になります。なぜなら、自動的な型変換をただで手に入れられるのですから。

強力になった try! マクロを手に入れたので、以前書いた、ファイルを読み込んで内容を整数値に変換するコードを見直してみましょう：

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, String> {
    let mut file = try!(File::open(file_path).map_err(|e| e.to_string()));
```

[*52] http://doc.rust-lang.org/std/macro.try!.html

```
    let mut contents = String::new();
    try!(file.read_to_string(&mut contents).map_err(|e| e.to_string()));
    let n = try!(contents.trim().parse::<i32>().map_err(|e| e.to_string()));
    Ok(2 * n)
}
```

以前 map_err の呼び出しを取り除くことができると約束しました。もちろんです。ここでしなければいけないのは From と共に動く型を一つ選ぶだけでよいのです。 前の節で見たように From の実装の一つは、どんなエラー型でも Box<Error> に変換できます：

```
use std::error::Error;
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, Box<Error>> {
    let mut file = try!(File::open(file_path));
    let mut contents = String::new();
    try!(file.read_to_string(&mut contents));
    let n = try!(contents.trim().parse::<i32>());
    Ok(2 * n)
}
```

理想的なエラーハンドリングまで、あと一歩です。私たちのコードには、エラーハンドリングを終えた後も、ごくわずかなオーバーヘッドしかありません。これは try! マクロが同時に 3 つのことをカプセル化するからです：

1. 場合分け
2. 制御フロー
3. エラー型の変換

これら 3 つが一つになったとき、コンビネータ、 unwrap の呼び出し、場合分けなどの邪魔者を排除したコードが得られるのです。

あとひとつ、些細なことが残っています：Box<Error> 型は **オペーク**なのです。 もし Box<Error> を呼

び出し元に返すと、呼び出し元では背後のエラー型が何であるかを、（簡単には）調べられません。この状況は `String` を返すよりは明らかに改善されてます。なぜなら、呼び出し元では `description`[*53] や `cause`[*54] といったメソッドを呼ぶこともできるからです。 しかし `Box<Error>` が不透明であるという制限は残ります。（注意：これは完全な真実ではありません。なぜなら Rust では実行時のリフレクションができるからです。この方法が有効なシナリオもありますが、このセクションで扱う範囲を超えています[*55]）

では、私たちの独自のエラー型 `CliError` に戻って、全てを一つにまとめ上げましょう。

独自のエラー型を合成する

前の節では `try!` マクロの本当の定義を確認し、それが `From::from` をエラーの値に対して呼ぶことで、自動的な型変換をする様子を見ました。特にそこでは、エラーを `Box<Error>` に変換しました。これはたしかに動きますが、呼び出し元にとって、型がオペークになってしまいました。

これを直すために、すでによく知っている改善方法である独自のエラー型を使いましょう。もう一度、ファイルの内容を読み込んで整数値に変換するコードです：

```
use std::fs::File;
use std::io::{self, Read};
use std::num;
use std::path::Path;

// 全ての型は `Debug` を導出するべきでしょうから、ここでも `Debug` を導出します。
// これにより `CliError` 値について、人間が十分理解できる説明を得られます。
#[derive(Debug)]
enum CliError {
    Io(io::Error),
    Parse(num::ParseIntError),
}

fn file_double_verbose<P: AsRef<Path>>(file_path: P) -> Result<i32, CliError> {
    let mut file = try!(File::open(file_path).map_err(CliError::Io));
```

[*53] http://doc.rust-lang.org/std/error/trait.Error.html#tymethod.description
[*54] http://doc.rust-lang.org/std/error/trait.Error.html#method.cause
[*55] https://crates.io/crates/error

```
    let mut contents = String::new();
    try!(file.read_to_string(&mut contents).map_err(CliError::Io));
    let n: i32 = try!(contents.trim().parse().map_err(CliError::Parse));
    Ok(2 * n)
}
```

map_err がまだあることに注目してください。 なぜって、try! と From の定義を思い出してください。ここでの問題は io::Error や num::ParseIntError といったエラー型を、私たち独自の CliError に変換できる From の実装が無いことです。 もちろん、これは簡単に直せます！ CliError を定義したわけですから、それに対して From を実装できます：

```
use std::io;
use std::num;

impl From<io::Error> for CliError {
    fn from(err: io::Error) -> CliError {
        CliError::Io(err)
    }
}

impl From<num::ParseIntError> for CliError {
    fn from(err: num::ParseIntError) -> CliError {
        CliError::Parse(err)
    }
}
```

これらの実装がしていることは、From に対して、どうやって他のエラー型を元に CliError を作るのか、教えてあげているだけです。このケースでは、単に対応する値コンストラクタを呼ぶことで構築しています。本当に **普通は** これくらい簡単にできてしまいます。

これでようやく file_double を書き直せます：

```
use std::fs::File;
use std::io::Read;
use std::path::Path;
```

```
fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, CliError> {
    let mut file = try!(File::open(file_path));
    let mut contents = String::new();
    try!(file.read_to_string(&mut contents));
    let n: i32 = try!(contents.trim().parse());
    Ok(2 * n)
}
```

ここでしたのは `map_err` を取り除くことだけです。 それらは `try!` マクロがエラーの値に対して `From::from` を呼ぶので、もう不要になりました。これで動くのは、起こりうる全てのエラー型に対して `From` の実装を提供したからです。

もし `file_double` 関数を変更して、なにか他の操作、例えば、文字列を浮動小数点数に変換させたいと思ったら、エラー型のヴァリアントを追加するだけです：

```
use std::io;
use std::num;

enum CliError {
    Io(io::Error),
    ParseInt(num::ParseIntError),
    ParseFloat(num::ParseFloatError),
}
```

そして、新しい `From` 実装を追加します：

```
use std::num;

impl From<num::ParseFloatError> for CliError {
    fn from(err: num::ParseFloatError) -> CliError {
        CliError::ParseFloat(err)
    }
}
```

これで完成です！

ライブラリ作者たちへのアドバイス

もし、あなたのライブラリがカスタマイズされたエラーを報告しなければならないなら、恐らく、独自のエラー型を定義するべきでしょう。エラーの表現を表にさらすか（例：`ErrorKind`[*56] ）、隠しておくか（例：`ParseIntError`[*57] ）は、あなたの自由です。 いずれかに関係なく、最低でも `String` による表現を超えたエラー情報を提供することが、ほとんどの場合、良い方法となるしょう。しかしこれは紛れもなく、ユースケースに深く依存します。

最低でも `Error`[*58] トレイトを実装するべきでしょう。 これにより、ライブラリの利用者にエラーを合成するための、最低ラインの柔軟性を与えます。`Error` トレイトを実装することは、利用者がエラーの文字列表現を取得できると保証することにもなります（なぜなら、こうすると `fmt::Debug` と `fmt::Display` の実装が必須になるからです）。

さらには、あなたのエラー型に対して `From` の実装を提供するのも便利かもしれません。このことは、（ライブラリ作者である）あなたと利用者が、より詳細なエラーを合成することを可能にします。 例えば `csv::Error`[*59] は `io::Error` と `byteorder::Error` の両方に `From` 実装を提供しています。

最後に、お好みで `Result` 型エイリアス を定義したくなるかもしれません。特にライブラリでエラー型を一つだけ定義しているときは当てはまります。この方法は標準ライブラリの `io::Result`[*60] や `fmt::Result`[*61] で用いられています。

ケーススタディ：人口データを読み込むプログラム

このセクションは長かったですね。あなたのバックグラウンドにもよりますが、内容が少し濃すぎたかもしれません。たくさんのコード例に、散文的な説明が添えられる形で進行しましたが、これは主に学習を助けるために、あえてこう構成されていたのでした。次はなにか新しいことをしましょう。ケーススタディです。

ここでは世界の人口データを問い合わせるための、コマンドラインプログラムを構築します。目標はシンプルです：プログラムに場所を与えると、人口を教えてくれます。シンプルにも関わらず、失敗しそ

[*56] http://doc.rust-lang.org/std/io/enum.ErrorKind.html
[*57] http://doc.rust-lang.org/std/num/struct.ParseIntError.html
[*58] http://doc.rust-lang.org/std/error/trait.Error.html
[*59] http://burntsushi.net/rustdoc/csv/enum.Error.html
[*60] http://doc.rust-lang.org/std/io/type.Result.html
[*61] http://doc.rust-lang.org/std/fmt/type.Result.html

うな所がたくさんあります！

ここで使うデータはデータサイエンスツールキット[*62]から取得したものです。これを元に演習で使うデータを準備しましたので、2 つのファイルのどちらかをダウンロードしてください：世界の人口データ[*63]（gzip 圧縮時 41MB、解凍時 145MB）と、アメリカ合衆国の人口データ[*64]（gzip 圧縮時 2.2MB、解凍時 7.2MB）があります。

いままで書いてきたコードでは、Rust の標準ライブラリだけを使うようにしてきました。今回のような現実のタスクでは、最低でも CSV データをパースする部分と、プログラムの引数をパースして、自動的に Rust の型にデコードする部分に何か使いたいでしょう。これには `csv`[*65] と `rustc-serialize`[*66] クレートを使います。

最初のセットアップ

Cargo を使ってプロジェクトをセットアップしますが、その方法はすでに Hello, Cargo! と Cargo のドキュメント[*67]でカバーされていますので、ここでは簡単に説明します。

何もない状態から始めるには、`cargo new —bin city-pop` を実行し、 `Cargo.toml` を以下のように編集します：

```
[package]
name = ”city-pop”
version = ”0.1.0”
authors = [”Andrew Gallant <jamslam@gmail.com>”]

[[bin]]
name = ”city-pop”

[dependencies]
csv = ”0.*”
rustc-serialize = ”0.*”
getopts = ”0.*”
```

[*62] https://github.com/petewarden/dstkdata
[*63] http://burntsushi.net/stuff/worldcitiespop.csv.gz
[*64] http://burntsushi.net/stuff/uscitiespop.csv.gz
[*65] https://crates.io/crates/csv
[*66] https://crates.io/crates/rustc-serialize
[*67] http://doc.crates.io/guide.html

これでもう実行できるはずです：

```
cargo build --release
./target/release/city-pop
# 出力: Hello, world!
```

引数のパース

引数のパースができるようにしましょう。Getopts については、あまり深く説明しませんが、詳細を解説したドキュメント[*68]があります。簡単に言うと、Getopts はオプションのベクタから、引数のパーサーとヘルプメッセージを生成します（実際には、ベクタは構造体とメソッドの背後に隠れています）。パースが終わると、プログラムの引数を Rust の構造体へとデコードできます。そこから、例えば、フラグが指定されたかとか、フラグの引数がなんであったかといった、フラグの情報を取り出せるようになります。プログラムに適切な `extern crate` 文を追加して、Getopts の基本的な引数を設定すると、こうなります：

```
extern crate getopts;
extern crate rustc_serialize;

use getopts::Options;
use std::env;

fn print_usage(program: &str, opts: Options) {
    println!("{}", opts.usage(&format!("Usage: {} [options] <data-path> <city>", progr
   am)));
}

fn main() {
    let args: Vec<String> = env::args().collect();
    let program = &args[0];

    let mut opts = Options::new();
    opts.optflag("h", "help", "Show this usage message.");
```

[*68] http://doc.rust-lang.org/getopts/getopts/index.html

```
    let matches = match opts.parse(&args[1..]) {
        Ok(m)  => { m }
        Err(e) => { panic!(e.to_string()) }
    };
    if matches.opt_present("h") {
        print_usage(&program, opts);
        return;
    }
    let data_path = &args[1];
    let city = &args[2];

    // 情報を元にいろいろなことをする
}
```

訳注

- Usage: {} [options] ：使用法：{} [options]
- Show this usage message.：この使用法のメッセージを表示する。

このように、まず、このプログラムに渡された引数のベクタを取得します。次に、最初の要素、つまり、プログラムの名前を格納します。続いて引数フラグをセットアップしますが、今回はごく簡単なヘルプメッセージフラグが一つあるだけです。セットアップできたら `Options.parse` を使って引数のベクタをパースします（インデックス 0 はプログラム名ですので、インデックス 1 から始めます）。もしパースに成功したら、パースしたオブジェクトをマッチで取り出しますし、失敗したならパニックさせます。ここまでたどり着いたら、ヘルプフラグが指定されたか調べて、もしそうなら使用法のメッセージを表示します。ヘルプメッセージのオプションは Getopts により生成済みですので、使用法のメッセージを表示するために追加で必要なのは、プログラム名とテンプレートだけです。もしユーザーがヘルプフラグを指定しなかったなら、変数を用意して、対応する引数の値をセットします。

ロジックを書く

コードを書く順番は人それぞれですが、エラーハンドリングは最後に考えることが多いでしょう。これはプログラム全体の設計にとっては、あまり良いことではありません。しかし、ラピッドプロトタイピングには便利かもしれません。Rust は私たちにエラーハンドリングが明示的であることを（ `unwrap` を呼ばせることで）強制しますので、プログラムのどこがエラーを起こすかは、簡単にわかります。

このケーススタディでは、ロジックは非常にシンプルです。やることは、与えられた CSV データをパースして、マッチした行にあるフィールドを表示するだけです。やってみましょう。（ファイルの先頭に `extern crate csv;` を追加することを忘れずに。）

```
use std::fs::File;

// この構造体は CSV ファイルの各行のデータを表現します。
// 型に基づいたデコードにより、文字列を整数や浮動小数点数にパースして
// しまうといった、核心部分のエラーハンドリングの大半から解放されます。
#[derive(Debug, RustcDecodable)]
struct Row {
    country: String,
    city: String,
    accent_city: String,
    region: String,

    // 人口、経度、緯度などのデータは全ての行にあるわけではありません！
    // そこで、これらは不在の可能性を許す `Option` 型で表現します。
    // CSV パーサーは、これらを正しい値で埋めてくれます。
    population: Option<u64>,
    latitude: Option<f64>,
    longitude: Option<f64>,
}

fn print_usage(program: &str, opts: Options) {
    println!("{}", opts.usage(&format!("Usage: {} [options] <data-path> <city>", progr
   am)));
}

fn main() {
    let args: Vec<String> = env::args().collect();
    let program = &args[0];

    let mut opts = Options::new();
    opts.optflag("h", "help", "Show this usage message.");
```

```
    let matches = match opts.parse(&args[1..]) {
        Ok(m)  => { m }
        Err(e) => { panic!(e.to_string()) }
    };

    if matches.opt_present("h") {
        print_usage(&program, opts);
        return;
    }

    let data_path = &args[1];
    let city: &str = &args[2];

    let file = File::open(data_path).unwrap();
    let mut rdr = csv::Reader::from_reader(file);

    for row in rdr.decode::<Row>() {
        let row = row.unwrap();

        if row.city == city {
            println!("{}, {}: {:?}",
                row.city, row.country,
                row.population.expect("population count"));
        }
    }
}
```

訳注：
population count：人口のカウント

ここで、エラーの概要を把握しましょう。 まずは明白なところ、つまり `unwrap` が呼ばれている 3 ヶ所から始めます：

1. File::open[*69] が io::Error[*70] を返すかもしれない。
2. csv::Reader::decode[*71] は 1 度に 1 件のレコードをデコードし、レコードのデコード[*72]（Iterator の impl の Item 関連型を見てください）は csv::Error[*73] を起こすかもしれない。
3. もし row.population が None なら、 expect の呼び出しはパニックする。

他にもありますか？ もし一致する都市が見つからなかったら？ grep のようなツールはエラーコードを返しますので、ここでも、そうするべきかもしれません。つまり、IO エラーと CSV パースエラーの他に、このプログラム特有のエラーロジックがあるわけです。これらのエラーを扱うために、2 つのアプローチを試してみましょう。

まず Box<Error> から始めたいと思います。その後で、独自のエラー型を定義すると、どのように便利になるかを見てみましょう。

Box<Error> によるエラー処理

Box<Error> の良いところは**とにかく動く** ことです。 エラー型を定義して From を実装する、といったことは必要ありません。 これの欠点は Box<Error> がトレイトオブジェクトなので**型消去** され、コンパイラが背後の型を推測できなくなることです。

以前コードのリファクタリングを、関数の型を T から Result<T, **私たちのエラー型**> に変更することから始めました。 ここでは **私たちのエラー型** は単に Box<Error> です。 でも T は何になるでしょう？ それに main にリターン型を付けられるのでしょうか？

2 つ目の質問の答えはノーです。できません。つまり新しい関数を書くことになります。 では T は何になるでしょう？ 一番簡単にできるのは、マッチした Row 値のリストを Vec<Row> として返すことです。（もっと良いコードはイテレータを返すかもしれませんが、これは読者の皆さんへの練習問題とします。）

該当のコードを、専用の関数へとリファクタリングしましょう。 ただし unwrap の呼び出しはそのままにします。また、人口のカウントがない場合は、いまは単にその行を無視することに注意してください。

```
use std::path::Path;


struct Row {
```

[*69] http://doc.rust-lang.org/std/fs/struct.File.html#method.open
[*70] http://doc.rust-lang.org/std/io/struct.Error.html
[*71] http://burntsushi.net/rustdoc/csv/struct.Reader.html#method.decode
[*72] http://burntsushi.net/rustdoc/csv/struct.DecodedRecords.html
[*73] http://burntsushi.net/rustdoc/csv/enum.Error.html

```
    // 変更なし
}

struct PopulationCount {
    city: String,
    country: String,
    // これは `Option` から変更します。なぜなら、この型の値は
    // 人口のカウントがあるときだけ構築されるようになったからです。
    count: u64,
}

fn print_usage(program: &str, opts: Options) {
    println!("{}", opts.usage(&format!("Usage: {} [options] <data-path> <city>", progr
   am)));
}

fn search<P: AsRef<Path>>(file_path: P, city: &str) -> Vec<PopulationCount> {
    let mut found = vec![];
    let file = File::open(file_path).unwrap();
    let mut rdr = csv::Reader::from_reader(file);
    for row in rdr.decode::<Row>() {
        let row = row.unwrap();
        match row.population {
            None => { } // スキップする
            Some(count) => if row.city == city {
                found.push(PopulationCount {
                    city: row.city,
                    country: row.country,
                    count: count,
                });
            },
        }
    }
    found
```

```
}

fn main() {
    let args: Vec<String> = env::args().collect();
    let program = &args[0];

    let mut opts = Options::new();
    opts.optflag("h", "help", "Show this usage message.");

    let matches = match opts.parse(&args[1..]) {
        Ok(m)  => { m }
        Err(e) => { panic!(e.to_string()) }
    };
    if matches.opt_present("h") {
        print_usage(&program, opts);
        return;
    }

    let data_path = &args[1];
    let city = &args[2];
    for pop in search(data_path, city) {
        println!("{}, {}: {:?}", pop.city, pop.country, pop.count);
    }
}
```

`expect`（`unwrap` の少し良い版）の使用を 1 つ取り除くことができましたが、検索の結果が無いときのハンドリングは、依然として必要です。

このエラーを適切に処理するためには、以下のようにします：

1. `search` のリターン型を `Result<Vec<PopulationCount>, Box<Error>>` に変更する。
2. `try!` マクロを使用することで、プログラムをパニックする代わりに、エラーを呼び出し元に返す。
3. `main` でエラーをハンドリングする。

やってみましょう：

```
use std::error::Error;

// ここ以前の他のコードに変更なし

fn search<P: AsRef<Path>>
         (file_path: P, city: &str)
         -> Result<Vec<PopulationCount>, Box<Error+Send+Sync>> {
    let mut found = vec![];
    let file = try!(File::open(file_path));
    let mut rdr = csv::Reader::from_reader(file);
    for row in rdr.decode::<Row>() {
        let row = try!(row);
        match row.population {
            None => { } // スキップする
            Some(count) => if row.city == city {
                found.push(PopulationCount {
                    city: row.city,
                    country: row.country,
                    count: count,
                });
            },
        }
    }
    if found.is_empty() {
        Err(From::from("No matching cities with a population were found."))
    } else {
        Ok(found)
    }
}
```

訳注：
No matching cities with a population were found.：条件に合う人口データ付きの街は見つかりませんでした。

x.unwrap() の代わりに、今では try!(x) があります。私たちの関数が Result<T, E> を返すので、エラーの発生時、 try! マクロは関数の途中で戻ります。

このコードで 1 点注意があります：Box<Error> の代わりに Box<Error + Send + Sync> を使いました。こうすると、プレーンな文字列をエラー型に変換できます。この From 実装[*74] を使うために、このような追加の制限が必要でした。

```
// 上のコードでは `&'static str` に対して `From::from` を呼ぶことで、
// こちらの実装を使おうとしています。
impl<'a, 'b> From<&'b str> for Box<Error + Send + Sync + 'a>

// もし `format!` などを使ってエラーメッセージのために新しい文字列を
// 割り当てる場合は、こちらの実装も使えます。
impl From<String> for Box<Error + Send + Sync>
```

search が Result<T, E> を返すようになったため、 main は search を呼ぶときに場合分けをしなければなりません：

```
...
match search(&data_file, &city) {
    Ok(pops) => {
        for pop in pops {
            println!("{}, {}: {:?}", pop.city, pop.country, pop.count);
        }
    }
    Err(err) => println!("{}", err)
}
...
```

Box<Error> を使った適切なエラーハンドリングについて見ましたので、次は独自のエラー型による別のアプローチを試してみましょう。でもその前に、少しの間、エラーハンドリングから離れて、 stdin からの読み込みをサポートしましょう。

[*74] http://doc.rust-lang.org/std/convert/trait.From.html

標準入力から読み込む

このプログラムでは、入力としてファイルをただ一つ受け取り、1 回のパスでデータを処理しています。これは、標準入力からの入力を受け付けたほうがいいことを意味しているのかもしれません。でも、いまの方法も捨てがたいので、両方できるようにしましょう！

標準入力のサポートを追加するのは実に簡単です。 やることは 3 つだけです：

1. プログラムの引数を微修正して、唯一のパラメータとして「都市」を受け付け、人口データは標準入力から読み込むようにする。
2. プログラムを修正して、ファイルが標準入力に流し込まれなかったときに、`-f` オプションからファイルを得られるようにする。
3. `search` 関数を修正して、ファイルパスを **オプション**で受け取れるようにする。もし `None` なら標準入力から読み込む。

まず、使用法を変更します：

```
fn print_usage(program: &str, opts: Options) {
    println!("{}", opts.usage(&format!("Usage: {} [options] <city>", program)));
}
```

次のパートはやや難しくなります：

```
...
let mut opts = Options::new();
opts.optopt("f", "file", "Choose an input file, instead of using STDIN.", "NAME");
opts.optflag("h", "help", "Show this usage message.");
...
let file = matches.opt_str("f");
let data_file = &file.as_ref().map(Path::new);

let city = if !matches.free.is_empty() {
    &matches.free[0]
} else {
    print_usage(&program, opts);
```

```
    return;
};

match search(data_file, city) {
    Ok(pops) => {
        for pop in pops {
            println!("{}, {}: {:?}", pop.city, pop.country, pop.count);
        }
    }
    Err(err) => println!("{}", err)
}
...
```

訳注：
Choose an input file, instead of using STDIN : STDIN を使う代わりに、入力ファイルを選択する。

このコードでは（Option<String> 型の）file を受け取り、 search が使える型、つまり今回は &Option<AsRef<Path>» へ変換します。 そのためには file の参照を得て、それに対して Path::new をマップします。 このケースでは as_ref() が Option<String> を Option<&str> へ変換しますので、続いて、そのオプション値の中身に対して Path::new を実行することで、新しいオプション値を返します。ここまでできれば、残りは単に city 引数を取得して search を実行するだけです。

search の修正は少し厄介です。 csv トレイトは io::Read を実装している型[*75] からなら、いずれかを問わず、パーサーを構築できます。しかし両方の型に同じコードが使えるのでしょうか？ これを実際する方法は 2 つあります。 ひとつの方法は search を io::Read を満たす型パラメータ R に対するジェネリックとして書くことです。もうひとつの方法は、以下のように、トレイトオブジェクトを使うことです：

```
use std::io;

// ここ以前の他のコードに変更なし

fn search<P: AsRef<Path>>
```

[*75] http://burntsushi.net/rustdoc/csv/struct.Reader.html#method.from_reader

```
         (file_path: &Option<P>, city: &str)
         -> Result<Vec<PopulationCount>, Box<Error+Send+Sync>> {
    let mut found = vec![];
    let input: Box<io::Read> = match *file_path {
        None => Box::new(io::stdin()),
        Some(ref file_path) => Box::new(try!(File::open(file_path))),
    };
    let mut rdr = csv::Reader::from_reader(input);
    // これ以降は変更なし！
}
```

独自のエラー型によるエラー処理

以前、どうやって独自のエラー型を使ってエラーを合成するのか学びました。 そのときはエラー型を enum 型として定義して、Error と From を実装することで実現しました。

3 つの異なるエラー（IO、CSV のパース、検索結果なし）がありますので enum として 3 つのヴァリアントを定義しましょう：

```
#[derive(Debug)]
enum CliError {
    Io(io::Error),
    Csv(csv::Error),
    NotFound,
}
```

Display と Error を実装します：

```
impl fmt::Display for CliError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match *self {
            CliError::Io(ref err) => err.fmt(f),
            CliError::Csv(ref err) => err.fmt(f),
            CliError::NotFound => write!(f, "No matching cities with a \
                                             population were found."),
```

```
        }
    }
}

impl Error for CliError {
    fn description(&self) -> &str {
        match *self {
            CliError::Io(ref err) => err.description(),
            CliError::Csv(ref err) => err.description(),
            CliError::NotFound => "not found",
        }
    }

    fn cause(&self) -> Option<&error::Error> {
        match *self {
            CliError::Io(ref err) => Some(err),
            CliError::Parse(ref err) => Some(err),
            // 今回の自前のエラーは下流の原因となるエラーは持っていませんが
            // そのように変更することも可能です。
            CliError::NotFound() => None,
        }
    }
}
```

CliError を search 関数の型に使う前に、いくつかの From 実装を用意しなければなりません。どのエラーについて用意したらいいのでしょう？　ええと io::Error と csv::Error の両方を CliError に変換する必要があります。 外部エラーはこれだけですので、今は 2 つの From 実装だけが必要になるのです：

```
impl From<io::Error> for CliError {
    fn from(err: io::Error) -> CliError {
        CliError::Io(err)
    }
}
```

```
impl From<csv::Error> for CliError {
    fn from(err: csv::Error) -> CliError {
        CliError::Csv(err)
    }
}
```

try! がこのように定義 されているので、From の実装が重要になります。特にエラーが起こると、エラーに対して From::from が呼ばれますので、このケースでは、それらのエラーが私たち独自のエラー型 CliError へ変換されます。

From の実装ができましたので、search 関数に 2 つの小さな修正が必要です： リターン型と「not found」エラーです。全体はこうなります：

```
fn search<P: AsRef<Path>>
         (file_path: &Option<P>, city: &str)
         -> Result<Vec<PopulationCount>, CliError> {
    let mut found = vec![];
    let input: Box<io::Read> = match *file_path {
        None => Box::new(io::stdin()),
        Some(ref file_path) => Box::new(try!(File::open(file_path))),
    };
    let mut rdr = csv::Reader::from_reader(input);
    for row in rdr.decode::<Row>() {
        let row = try!(row);
        match row.population {
            None => { } // スキップする
            Some(count) => if row.city == city {
                found.push(PopulationCount {
                    city: row.city,
                    country: row.country,
                    count: count,
                });
            },
        }
    }
```

```
    if found.is_empty() {
        Err(CliError::NotFound)
    } else {
        Ok(found)
    }
}
```

これ以外の変更は不要です。

機能を追加する

汎用的なコードを書くのは素晴らしいことです。なぜなら、物事を汎用的にするのはクールですし、後になって役立つかもしれません。でも時には、その苦労の甲斐がないこともあります。最後のステップで何をしたか振り返ってみましょう：

1. 新しいエラー型を定義した。
2. `Error` と `Display` の実装を追加し、2 つのエラーに対して `From` も実装した。

ここでの大きな問題は、このプログラムは全体で見ると大して良くならなかったことです。`enum` でエラーを表現するには、多くの付随する作業が必要です。特にこのような短いプログラムでは、それが顕著に現れました。

ここでしたような独自のエラー型を使うのが便利といえる **一つの要素は、** `main` 関数がエラーによってどう対処するのかを選択できるようになったことです。以前の `Box<Error>` では、メッセージを表示する以外、選択の余地はほとんどありませんでした。いまでもそうですが、例えば、もし `–quiet` フラグを追加したくなったらどうでしょうか？ `–quiet` フラグは詳細な出力を抑止すべきです。

いま現在は、プログラムがマッチするものを見つけられなかったとき、それを告げるメッセージを表示します。これは、特にプログラムをシェルスクリプトから使いたいときなどは、扱いにくいかもしれません。

フラグを追加してみましょう。以前したように、使用法についての文字列を少し修正して、オプション変数にフラグを追加します。そこまですれば、残りは Getopts がやってくれます：

```
...
let mut opts = Options::new();
opts.optopt("f", "file", "Choose an input file, instead of using STDIN.", "NAME");
```

```
opts.optflag("h", "help", "Show this usage message.");
opts.optflag("q", "quiet", "Silences errors and warnings.");
...
```

後は「quiet」機能を実装するだけです。 `main` 関数の場合分けを少し修正します：

```
match search(&args.arg_data_path, &args.arg_city) {
    Err(CliError::NotFound) if args.flag_quiet => process::exit(1),
    Err(err) => panic!("{}", err),
    Ok(pops) => for pop in pops {
        println!("{}, {}: {:?}", pop.city, pop.country, pop.count);
    }
}
```

訳注：
Silences errors and warnings.：エラーや警告を抑止します。

もちろん、IO エラーが起こったり、データのパースに失敗したときは、エラーを抑止したくはありません。そこで場合分けを行い、エラータイプが `NotFound` かつ`--quiet` が指定されたかを検査しています。もし検索に失敗したら、今まで通り（ `grep` の動作にならい）なにも表示せず、exit コードと共に終了します。

もし `Box<Error>` で留まっていたら、`--quiet` 機能を実装するのは、かなり面倒だったでしょう。

これが、このケーススタディの締めくくりとなります。これからは外の世界に飛び出して、あなた自身のプログラムやライブラリを、適切なエラーハンドリングと共に書くことができるでしょう。

まとめ

このセクションは長いので、Rust におけるエラー処理について簡単にまとめたほうがいいでしょう。そこには「大まかな法則」が存在しますが、これらは命令的なものでは断固として**ありません** 。それぞれのヒューリスティックを破るだけの十分な理由もあり得ます！

- もし短いサンプルコードを書いていて、エラーハンドリングが重荷になるようなら、`unwrap` を

使っても大丈夫かもしれません（`Result::unwrap`[*76], `Option::unwrap`[*77], `Option::expect`[*78] のいずれかが使えます）。あなたのコードを参考にする人は、正しいエラーハンドリングについて知っているべきです。（そうでなければ、このセクションを紹介してください！）

- もし即興のプログラムを書いているなら `unwrap` を使うことに罪悪感を持たなくてもいいでしょう。ただし警告があります：もしそれが最終的に他の人たちの手に渡るなら、彼らが貧弱なエラーメッセージに動揺してもおかしくありません。
- もし即興のプログラムを書いていて、パニックすることに、どうしても後ろめたさを感じるなら、エラー型として `String` か `Box<Error + Send + Sync>` のいずれかを使ってください（`Box<Error + Send + Sync>` は `From` 実装がある[*79] ので使えます）。
- これらに該当しないなら、独自のエラー型を定義し、適切な `From`[*80] と `Error`[*81] を実装することで `try!`[*82] マクロをエルゴノミックにしましょう。
- もしライブラリを書いていて、そのコードがエラーを起こす可能性があるなら、独自のエラー型を定義し、`std::error::Error`[*83] トレイトを実装してください。 もし必要なら `From`[*84] を実装することで、ライブラリ自身と呼び出し元のコードを書きやすくしてください。（Rust の調和性規則 (coherence rule) により、呼び出し側では、あなたのエラー型に対して `From` を実装することはできません。 ライブラリでするべきです。）
- `Option`[*85] と `Result`[*86] で定義されているコンビネータについて学んでください。それだけを使うのは大変ですが、 `try!` とコンビネータを適度にミックスすることは、個人的には、とても魅力的な方法だと考えています。`and_then`, `map`, `unwrap_or` が私のお気に入りです。

保証を選ぶ

Rust の重要な特長の 1 つは、プログラムのコストと保証を制御することができるということです。

Rust の標準ライブラリには、様々な「ラッパ型」の抽象があり、それらはコスト、エルゴノミクス、保

[*76] http://doc.rust-lang.org/std/result/enum.Result.html#method.unwrap
[*77] http://doc.rust-lang.org/std/option/enum.Option.html#method.unwrap
[*78] http://doc.rust-lang.org/std/option/enum.Option.html#method.expect
[*79] http://doc.rust-lang.org/std/convert/trait.From.html
[*80] http://doc.rust-lang.org/std/convert/trait.From.html
[*81] http://doc.rust-lang.org/std/error/trait.Error.html
[*82] http://doc.rust-lang.org/std/macro.try!.html
[*83] http://doc.rust-lang.org/std/error/trait.Error.html
[*84] http://doc.rust-lang.org/std/convert/trait.From.html
[*85] http://doc.rust-lang.org/std/option/enum.Option.html
[*86] http://doc.rust-lang.org/std/result/enum.Result.html

証の間の多数のトレードオフをまとめています。それらのトレードオフの多くでは実行時とコンパイル時のどちらかを選ばせてくれます。このセクションでは、いくつかの抽象を選び、詳細に説明します。

先に進む前に、Rust における 所有権 と借用について読んでおくことを強く推奨します。

基本的なポインタ型

Box<T>

Box<T>[*87] は「所有される」ポインタ、すなわち「ボックス」です。ボックスは中身のデータへの参照を渡すことができますが、ボックスだけがそのデータの唯一の所有者です。特に、次のことを考えましょう。

```
let x = Box::new(1);
let y = x;
// ここではもう x にアクセスできない
```

ここで、そのボックスは y に **ムーブ** されました。x はもはやそれを所有していないので、これ以降、コンパイラはプログラマが x を使うことを許しません。同様に、ボックスはそれを返すことで関数の **外**にムーブさせることもできます。

（ムーブされていない）ボックスがスコープから外れると、デストラクタが実行されます。それらのデストラクタは中身のデータを解放して片付けます。

これは動的割当てのゼロコスト抽象化です。もしヒープにメモリを割り当てたくて、そのメモリへのポインタを安全に取り回したいのであれば、これは理想的です。コンパイル時にチェックされる通常の借用のルールに基づいてこれへの参照を共有することが許されているだけだということに注意しましょう。

&T と &mut T

参照にはイミュータブルな参照とミュータブルな参照がそれぞれあります。それらは「読み書きロック」パターンに従います。それは、あるデータへのミュータブルな参照を 1 つだけ持つこと、又は複数のイミュータブルな参照を持つことはあり得るが、その両方を持つことはあり得ないということです。この保証はコンパイル時に強制され、目に見えるような実行時のコストは発生しません。多くの場合、それら 2 つのポインタ型は低コストの参照をコードのセクション間で共有するには十分です。

それらのポインタを関連付けられているライフタイムを超えて有効になるような方法でコピーすることはできません。

[*87] http://doc.rust-lang.org/std/boxed/struct.Box.html

*const T と*mut T

関連付けられたライフタイムや所有権を持たない、C 的な生ポインタがあります。それらはメモリのある場所を何の制約もなく指示します。それらの提供する唯一の保証は、 unsafe であるとマークされたコードの外ではそれらが参照を外せないということです。

それらは Vec<T> のような安全で低コストな抽象を構築するときには便利ですが、安全なコードの中では避けるべきです。

Rc<T>

これは、本書でカバーする中では初めての、実行時にコストの発生するラッパです。

Rc<T>[*88] は参照カウンタを持つポインタです。言い換えると、これを使えば、あるデータを「所有する」複数のポインタを持つことができるようになるということです。そして、全てのポインタがスコープから外れたとき、そのデータは削除されます（デストラクタが実行されます）。

内部的には、それは共有「参照カウント」（「refcount」とも呼ばれます）を持っています。それは、Rc がクローンされる度に 1 増加し、 Rc がスコープから外れる度に 1 減少します。Rc<T> の主な役割は、共有データのデストラクタが呼び出されることを保証することです。

ここでの中身のデータはイミュータブルで、もし循環参照が起きてしまったら、そのデータはメモリリークを起こすでしょう。もし循環してもメモリリークを起こさないデータを求めるのであれば、ガーベジコレクタが必要です。

保証

ここで提供される主な保証は、それに対する全ての参照がスコープから外れるまではデータが破壊されないということです。

これは（読込専用の）あるデータを動的に割り当て、プログラムの様々な部分で共有したいときで、どの部分が最後にポインタを使い終わるのかがはっきりしないときに使われるべきです。それは &T が正しさを静的にチェックすることが不可能なとき、又はプログラマがそれを使うために開発コストを費やすことを望まないような極めて非エルゴノミックなコードを作っているときに、&T の有望な代替品です。

このポインタはスレッドセーフでは **ありません**。Rust はそれを他のスレッドに対して送ったり共有したりはさせません。これによって、それらが不要な状況でのアトミック性のためのコストを省くことができます。

[*88] http://doc.rust-lang.org/std/rc/struct.Rc.html

これの姉妹に当たるスマートポインタとして、Weak<T> があります。これは所有せず、借用もしないスマートポインタです。 それは &T とも似ていますが、ライフタイムによる制約がありません。Weak<T> は永遠に有効であり続けることができます。 しかし、これは所有する Rc のライフタイムを超えて有効である可能性があるため、中身のデータへのアクセスが失敗し、None を返すという可能性があります。これは循環するデータ構造やその他のものについて便利です。

コスト

メモリに関する限り、 Rc<T> の割当ては 1 回です。ただし、普通の Box<T> と比べると（「強い」参照カウントと「弱い」参照カウントのために）、2 ワード余分（つまり、2 つの usize の値）に割り当てます。

Rc<T> では、それをクローンしたりそれがスコープから外れたりする度に参照カウントを増減するための計算コストが掛かります。クローンはディープコピーではなく、それが単に内部の参照カウントを 1 増加させ、Rc<T> のコピーを返すだけだということに注意しましょう。

セル型

Cell は内的ミュータビリティを提供します。言い換えると、型がミュータブルな形式を持てないものであったとしても（例えば、それが& ポインタや Rc<T> の参照先であるとき）、操作できるデータを持つということです。

cell モジュールのドキュメントには、それらについての非常によい説明があります[*89]。

それらの型は 一般的には構造体のフィールドで見られますが、他の場所でも見られるかもしれません。

Cell<T>

Cell<T>[*90] はゼロコストで内的ミュータビリティを提供するものですが、 Copy 型のためだけのものです。コンパイラは含まれている値によって所有されている全てのデータがスタック上にあることを認識しています。そのため、単純にデータが置き換えられることによって参照先のデータがメモリリークを起こす（又はもっと悪いことも!）心配はありません。

このラッパを使うことで、維持されている不変性に違反してしまう可能性もあるので、それを使うときには注意しましょう。もしフィールドが Cell でラップされているならば、そのデータの塊はミュータブルで、最初にそれを読み込んだときとそれを使おうと思ったときで同じままだとは限らないということのよい目印になります。

[*89] http://doc.rust-lang.org/std/cell/
[*90] http://doc.rust-lang.org/std/cell/struct.Cell.html

```
use std::cell::Cell;

let x = Cell::new(1);
let y = &x;
let z = &x;
x.set(2);
y.set(3);
z.set(4);
println!("{}", x.get());
```

ここでは同じ値を様々なイミュータブルな参照から変更できるということに注意しましょう。

これには次のものと同じ実行時のコストが掛かります。

```
let mut x = 1;
let y = &mut x;
let z = &mut x;
x = 2;
*y = 3;
*z = 4;
println!("{}", x);
```

しかし、それには実際に正常にコンパイルできるという追加の利点があります。

保証

これは「ミュータブルなエイリアスはない」という制約を、それが不要な場所において緩和します。しかし、これはその制約が提供する保証をも緩和してしまいます。もし不変条件が `Cell` に保存されているデータに依存しているのであれば、注意すべきです。

これは `&` や `&mut` の静的なルールの下では簡単な方法がない場合に、プリミティブやその他の `Copy` 型を変更するのに便利です。

`Cell` によって安全な方法で自由に変更できるようなデータへの内部の参照を得られるわけではありません。

コスト

Cell<T> の使用に実行時のコストは掛かりません。ただし、もしそれを大きな（Copy の）構造体をラップするために使っているのであれば、代わりに個々のフィールドを Cell<T> でラップする方がよいかもしれません。そうしなければ、各書込みが構造体の完全コピーを発生させることになるからです。

RefCell<T>

RefCell<T>[*91] もまた内的ミュータビリティを提供するものですが、 Copy 型に限定されません。

その代わり、それには実行時のコストが掛かります。RefCell<T> は読み書きロックパターンを実行時に（シングルスレッドのミューテックスのように）強制します。この点が、それをコンパイル時に行う &T や &mut T とは異なります。 これは borrow() 関数と borrow_mut() 関数によって行われます。それらは内部の参照カウントを変更し、それぞれイミュータブル、ミュータブルに参照を外すことのできるスマートポインタを戻します。参照カウントはスマートポインタがスコープから外れたときに元に戻されます。このシステムによって、ミュータブルな借用が有効なときには決してその他の借用が有効にならないということを動的に保証することができます。もしプログラマがそのような借用を作ろうとすれば、スレッドはパニックするでしょう。

```
use std::cell::RefCell;

let x = RefCell::new(vec![1,2,3,4]);
{
    println!("{:?}", *x.borrow())
}

{
    let mut my_ref = x.borrow_mut();
    my_ref.push(1);
}
```

Cell と同様に、これは主に、借用チェッカを満足させることが困難、又は不可能な状況で便利です。一般的に、そのような変更はネストした形式では発生しないと考えられますが、それをチェックすることはよいことです。

大きく複雑なプログラムにとって、物事を単純にするために何かを RefCell の中に入れることは便利です。例えば、Rust コンパイラの内部の ctxt 構造体にあるたくさんのマップはこのラッパの中にありま

[*91] http://doc.rust-lang.org/std/cell/struct.RefCell.html

す。それらは（初期化の直後ではなく生成の過程で）一度だけ変更されるか、又はきれいに分離された場所で数回変更されます。しかし、この構造体はあらゆる場所で全般的に使われているので、ミュータブルなポインタとイミュータブルなポインタとをジャグリング的に扱うのは難しく（あるいは不可能で）、おそらく拡張の困難な& ポインタのスープになってしまいます。 一方、 `RefCell` はそれらにアクセスするための（ゼロコストではありませんが）低コストの方法です。将来、もし誰かが既に借用されたセルを変更しようとするコードを追加すれば、それは（普通は確定的に）パニックを引き起こすでしょう。これは、その違反した借用まで遡り得ます。

同様に、Servo の DOM ではたくさんの変更が行われるようになっていて、そのほとんどは DOM 型にローカルです。しかし、複数の DOM に縦横無尽にまたがり、様々なものを変更するものもあります。全ての変更をガードするために `RefCell` と `Cell` を使うことで、あらゆる場所でのミュータビリティについて心配する必要がなくなり、それは同時に、変更が**実際に** 起こっている場所を強調してくれます。

もし `&` ポインタを使ってもっと単純に解決できるのであれば、`RefCell` は避けるべきであるということに注意しましょう。

保証

`RefCell` はミュータブルなエイリアスを作らせないという**静的な** 制約を緩和し、それを **動的な** 制約に置き換えます。そのため、その保証は変わりません。

コスト

`RefCell` は割当てを行いませんが、データとともに（サイズ 1 ワードの）追加の「借用状態」の表示を持っています。

実行時には、各借用が参照カウントの変更又はチェックを発生させます。

同期型

前に挙げた型の多くはスレッドセーフな方法で使うことができません。 特に `Rc<T>` と `RefCell<T>` は両方とも非アトミックな参照カウント（ **アトミックな**参照カウントとは、データ競合を発生させることなく複数のスレッドから増加させることができるもののことです）を使っていて、スレッドセーフな方法で使うことができません。これによってそれらを低コストで使うことができるのですが、それらのスレッドセーフなバージョンも必要です。それらは `Arc<T>` 、`Mutex<T>` 、`RwLock<T>` という形式で存在します。

非スレッドセーフな型はスレッド間で送ることが **できません** 。これはコンパイル時にチェックされます。

sync[*92] モジュールには並行プログラミングのための便利なラッパがたくさんありますが、以下では有名なものだけをカバーします。

Arc<T>

Arc<T>[*93] はアトミックな参照カウントを使う Rc<T> の別バージョンです（そのため、「Arc」なのです）。これはスレッド間で自由に送ることができます。

C++ の shared_ptr は Arc と似ていますが、C++ の場合、中身のデータは常にミュータブルです。C++ と同じセマンティクスで使うためには、Arc<Mutex<T>>、Arc<RwLock<T>>、Arc<UnsafeCell<T>> を使うべきです [*94]（ UnsafeCell<T> はどんなデータでも持つことができ、実行時のコストも掛かりませんが、それにアクセスするためには unsafe ブロックが必要というセル型です）。最後のものは、その使用がメモリをアンセーフにしないことを確信している場合にだけ使うべきです。次のことを覚えましょう。構造体に書き込むのはアトミックな作業ではなく、vec.push() のような多くの関数は内部でメモリの再割当てを行い、アンセーフな挙動を引き起こす可能性があります。そのため単純な操作であるということだけでは UnsafeCell を正当化するには十分ではありません。

保証

Rc のように、これは最後の Arc がスコープから外れたときに（循環がなければ）中身のデータのためのデストラクタが実行されることを（スレッドセーフに）保証します。

コスト

これには参照カウントの変更（これは、それがクローンされたりスコープから外れたりする度に発生します）にアトミック性を使うための追加のコストが掛かります。シングルスレッドにおいて、データを Arc から共有するのであれば、可能な場合は & ポインタを共有する方が適切です。

Mutex<T> と RwLock<T>

Mutex<T>[*95] と RwLock<T>[*96] は RAII ガード（ガードとは、ロックのようにそれらのデストラクタが呼び出されるまである状態を保持するオブジェクトのことです）による相互排他を提供します。それらの

[*92] http://doc.rust-lang.org/std/sync/index.html
[*93] http://doc.rust-lang.org/std/sync/struct.Arc.html
[*94] Arc<UnsafeCell<T>> は Send や Sync ではないため、実際にはコンパイルできません。しかし、Arc<Wrapper<T>> を得るために、手動でそれを Send と Sync を実装した型でラップすることができます。ここでの Wrapper は struct Wrapper<T>(UnsafeCell<T>) です。
[*95] http://doc.rust-lang.org/std/sync/struct.Mutex.html
[*96] http://doc.rust-lang.org/std/sync/struct.RwLock.html

両方とも、その `lock()` を呼び出すまでミューテックスは不透明です。その時点で、スレッドはロックが得られ、ガードが戻されるまでブロックします。このガードを使うことで、中身のデータに（ミュータブルに）アクセスできるようになり、ロックはガードがスコープから外れたときに解放されます。

```
{
    let guard = mutex.lock();
    // ガードがミュータブルに内部の型への参照を外す
    *guard += 1;
} // デストラクタが実行されるときにロックは解除される
```

`RwLock` には複数の読込みを効率化するという追加の利点があります。それはライタのない限り常に、共有されたデータに対する複数のリーダを安全に持つことができます。そして、`RwLock` によってリーダは「読込みロック」を取得できます。このようなロックは並行に取得することができ、参照カウントによって追跡することができます。ライタは「書込みロック」を取得する必要があります。「書込みロック」はすべてのリーダがスコープから外れたときにだけ取得できます。

保証

それらのどちらもスレッド間での安全で共有されたミュータビリティを提供しますが、それらはデッドロックしがちです。型システムによって、ある程度の追加のプロトコルの安全性を得ることができます。

コスト

それらはロックを保持するために内部でアトミック的な型を使います。それにはかなりコストが掛かります（それらは仕事が終わるまで、プロセッサ中のメモリ読込み全てをブロックする可能性があります）。たくさんの並行なアクセスが起こる場合には、それらのロックを待つことが遅くなる可能性があります。

合成

Rust のコードを読むときに一般的な悩みは、`Rc<RefCell<Vec<T>>>` のような型（又はそのような型のもっと複雑な合成）です。その合成が何をしているのか、なぜ作者はこんなものを選んだのか（そして、自分のコード内でいつこんな合成を使うべきなのか）ということは、常に明らかなわけではありません。

普通、それは不要なコストを支払うことなく、必要とする保証を互いに組み合わせた場合です。

例えば、`Rc<RefCell<T>>` はそのような合成の 1 つです。 `Rc<T>` そのものはミュータブルに参照を外すことができません。`Rc<T>` は共有を提供し、共有されたミュータビリティはアンセーフな挙動に繋がる可能性があります。そのため、動的に証明された共有されたミュータビリティを得るために、`RefCell<T>`

を中に入れます。これで共有されたミュータブルなデータを持つことになりますが、それは（リーダはなしで）ライタが 1 つだけ、又はリーダが複数という方法で共有することになります。

今度は、これをさらに次の段階に進めると、`Rc<RefCell<Vec<T>>>` や `Rc<Vec<RefCell<T>>>` を持つことができます。それらは両方とも共有できるミュータブルなベクタですが、同じではありません。

1 つ目について、 `RefCell<T>` は `Vec<T>` をラップしているので、その `Vec<T>` 全体がミュータブルです。同時に、それらは特定の時間において `Vec` 全体の唯一のミュータブルな借用になり得ます。これは、コードがそのベクタの別の要素について、別の `Rc` ハンドルから同時には操作できないということを意味します。 しかし、`Vec<T>` に対するプッシュやポップは好きなように行うことができます。これは借用チェックが実行時に行われるという点で`&mut Vec<T>` と同様です。

2 つ目について、借用は個々の要素に対して行われますが、ベクタ全体がイミュータブルになります。そのため、異なる要素を別々に借用することができますが、ベクタに対するプッシュやポップを行うことはできません。これは `&mut [T]`[*97]と同じですが、やはり借用チェックは実行時に行われます。

並行プログラムでは、`Arc<Mutex<T>>` と似た状況に置かれます。それは共有されたミュータビリティと所有権を提供します。

それらを使ったコードを読むときには、1 行 1 行進み、提供される保証とコストを見ましょう。

合成された型を選択するときには、その逆に考えなければなりません。必要とする保証が何であるか、必要とする合成がどの点にあるのかを理解しましょう。例えば、もし `Vec<RefCell<T>>` と `RefCell<Vec<T>>` のどちらかを選ぶのであれば、前の方で行ったようにトレードオフを理解し、選ばなければなりません。

他言語関数インターフェース

導入

このガイドでは、他言語コードのためのバインディングを書く導入に snappy[*98] という圧縮・展開ライブラリを使います。Rust は現在、C++ ライブラリを直接呼び出すことができませんが、snappy は C のインターフェイスを持っています（ドキュメントが `snappy-c.h`[*99] にあります）。

[*97] `&[T]` と`&mut [T]` は **スライス**です。それらはポインタと長さを持ち、ベクタや配列の一部を参照することができます。`&mut [T]` ではその要素を変更できますが、その長さは変更することができません。

[*98] https://github.com/google/snappy

[*99] https://github.com/google/snappy/blob/master/snappy-c.h

libc についてのメモ

これらの例の多くは `libc` クレート[*100]を使っています。これは、主に C の様々な型の定義を提供するものです。もしこれらの例を自分で試すのであれば、次のように `libc` を `Cargo.toml` に追加する必要があるでしょう。

```
[dependencies]
libc = "0.2.0"
```

そして、クレートのルートに `extern crate libc;` を追加しましょう。

他言語関数の呼出し

次のコードは、snappy がインストールされていればコンパイルできる他言語関数を呼び出す最小の例です。

```
# #![feature(libc)]
extern crate libc;
use libc::size_t;

#[link(name = "snappy")]
extern {
    fn snappy_max_compressed_length(source_length: size_t) -> size_t;
}

fn main() {
    let x = unsafe { snappy_max_compressed_length(100) };
    println!("max compressed length of a 100 byte buffer: {}", x);
}
```

`extern` ブロックは他言語ライブラリの中の関数のシグネチャ、この例ではそのプラットフォーム上の C ABI によるもののリストです。 `#[link(...)]` アトリビュートは、シンボルが解決できるように、リンカに対して snappy のライブラリをリンクするよう指示するために使われています。

[*100] https://crates.io/crates/libc

他言語関数はアンセーフとみなされるので、それらを呼び出すには、この中に含まれているすべてのものが本当に安全であるということをコンパイラに対して約束するために、`unsafe {}` で囲まなければなりません。C ライブラリは、スレッドセーフでないインターフェイスを公開していることがありますし、ポインタを引数に取る関数のほとんどは、ポインタがダングリングポインタになる可能性を有しているので、すべての入力に対して有効なわけではありません。そして、生ポインタは Rust の安全なメモリモデルから外れてしまいます。

他言語関数について引数の型を宣言するとき、Rust のコンパイラはその宣言が正しいかどうかを確認することができません。それを正しく指定することは実行時にバインディングを正しく動作させるために必要なことです。

`extern` ブロックは snappy の API 全体をカバーするように拡張することができます。

```
# #![feature(libc)]
extern crate libc;
use libc::{c_int, size_t};

#[link(name = "snappy")]
extern {
    fn snappy_compress(input: *const u8,
                       input_length: size_t,
                       compressed: *mut u8,
                       compressed_length: *mut size_t) -> c_int;
    fn snappy_uncompress(compressed: *const u8,
                         compressed_length: size_t,
                         uncompressed: *mut u8,
                         uncompressed_length: *mut size_t) -> c_int;
    fn snappy_max_compressed_length(source_length: size_t) -> size_t;
    fn snappy_uncompressed_length(compressed: *const u8,
                                  compressed_length: size_t,
                                  result: *mut size_t) -> c_int;
    fn snappy_validate_compressed_buffer(compressed: *const u8,
                                         compressed_length: size_t) -> c_int;
}
# fn main() {}
```

安全なインターフェイスの作成

生の C API は、メモリの安全性を提供し、ベクタのようなもっと高レベルの概念を使うようにラップしなければなりません。ライブラリは安全で高レベルなインターフェイスのみを公開するように選択し、アンセーフな内部の詳細を隠すことができます。

バッファを期待する関数をラップするには、Rust のベクタをメモリへのポインタとして操作するために `slice::raw` モジュールを使います。Rust のベクタは隣接したメモリのブロックであることが保証されています。その長さは現在含んでいる要素の数で、容量は割り当てられたメモリの要素の合計のサイズです。長さは、容量以下です。

```
pub fn validate_compressed_buffer(src: &[u8]) -> bool {
    unsafe {
        snappy_validate_compressed_buffer(src.as_ptr(), src.len() as size_t) == 0
    }
}
```

上の `validate_compressed_buffer` ラッパは `unsafe` ブロックを使っていますが、関数のシグネチャを `unsafe` から外すことによって、その呼出しがすべての入力に対して安全であることが保証されています。

結果を保持するようにバッファを割り当てなければならないため、`snappy_compress` 関数と `snappy_uncompress` 関数はもっと複雑です。

`snappy_max_compressed_length` 関数は、圧縮後の結果を保持するために必要な最大の容量のベクタを割り当てるために使うことができます。そして、そのベクタは結果を受け取るための引数として `snappy_compress` 関数に渡されます。結果を受け取るための引数は、長さをセットするために、圧縮後の本当の長さを取得するためにも渡されます。

```
pub fn compress(src: &[u8]) -> Vec<u8> {
    unsafe {
        let srclen = src.len() as size_t;
        let psrc = src.as_ptr();

        let mut dstlen = snappy_max_compressed_length(srclen);
        let mut dst = Vec::with_capacity(dstlen as usize);
        let pdst = dst.as_mut_ptr();
```

```
        snappy_compress(psrc, srclen, pdst, &mut dstlen);
        dst.set_len(dstlen as usize);
        dst
    }
}
```

snappy は展開後のサイズを圧縮フォーマットの一部として保存していて、snappy_uncompressed_length が必要となるバッファの正確なサイズを取得するため、展開も同様です。

```
pub fn uncompress(src: &[u8]) -> Option<Vec<u8>> {
    unsafe {
        let srclen = src.len() as size_t;
        let psrc = src.as_ptr();

        let mut dstlen: size_t = 0;
        snappy_uncompressed_length(psrc, srclen, &mut dstlen);

        let mut dst = Vec::with_capacity(dstlen as usize);
        let pdst = dst.as_mut_ptr();

        if snappy_uncompress(psrc, srclen, pdst, &mut dstlen) == 0 {
            dst.set_len(dstlen as usize);
            Some(dst)
        } else {
            None // SNAPPY_INVALID_INPUT
        }
    }
}
```

参考のために、ここで使った例は GitHub 上のライブラリ[*101]としても置いておきます。

[*101] https://github.com/thestinger/rust-snappy

デストラクタ

他言語ライブラリはリソースの所有権を呼出先のコードに手渡してしまうことがあります。そういうことが起きる場合には、安全性を提供し、それらのリソースが解放されることを保証するために、Rust のデストラクタを使わなければなりません（特にパニックした場合）。

デストラクタについて詳しくは、Drop トレイト[*102]を見てください。

C のコードから Rust の関数へのコールバック

外部のライブラリの中には、現在の状況や中間的なデータを呼出元に報告するためにコールバックを使わなければならないものがあります。Rust で定義された関数を外部のライブラリに渡すことは可能です。これをするために必要なのは、C のコードから呼び出すことができるように正しい呼出規則に従って、コールバック関数を `extern` としてマークしておくことです。

そして、登録呼出しを通じてコールバック関数を C のライブラリに送ることができるようになり、後でそれらから呼び出すことができるようになります。

基本的な例は次のとおりです。

これが Rust のコードです。

```
extern fn callback(a: i32) {
    println!(”I'm called from C with value {0}”, a);
}

#[link(name = ”extlib”)]
extern {
   fn register_callback(cb: extern fn(i32)) -> i32;
   fn trigger_callback();
}

fn main() {
    unsafe {
```

[*102] http://doc.rust-lang.org/std/ops/trait.Drop.html

```
        register_callback(callback);
# //        trigger_callback(); // Triggers the callback
        trigger_callback(); // コールバックをトリガする
    }
}
```

これが C のコードです。

```
typedef void (*rust_callback)(int32_t);
rust_callback cb;

int32_t register_callback(rust_callback callback) {
    cb = callback;
    return 1;
}

void trigger_callback() {
  cb(7); // Rust の callback(7) を呼び出す
}
```

この例では、Rust の `main()` が C の `trigger_callback()` を呼び出し、今度はそれが、Rust の `callback()` をコールバックしています。

Rust のオブジェクトを対象にしたコールバック

先程の例では、グローバルな関数を C のコードから呼ぶための方法を示してきました。しかし、特別な Rust のオブジェクトをコールバックの対象にしたいことがあります。これは、そのオブジェクトをそれぞれ C のオブジェクトのラッパとして表現することで可能になります。

これは、そのオブジェクトへの生ポインタを C ライブラリに渡すことで実現できます。そして、C のライブラリは Rust のオブジェクトへのポインタをその通知の中に含むことができるようになります。これにより、そのコールバックは参照される Rust のオブジェクトにアンセーフな形でアクセスできるようになります。

これが Rust のコードです。

```
#[repr(C)]
```

```
struct RustObject {
    a: i32,
#    // other members
    // その他のメンバ
}

extern "C" fn callback(target: *mut RustObject, a: i32) {
    println!("I'm called from C with value {0}", a);
    unsafe {
#        // Update the value in RustObject with the value received from the callback
        // コールバックから受け取った値で RustObject の中の値をアップデートする
        (*target).a = a;
    }
}

#[link(name = "extlib")]
extern {
   fn register_callback(target: *mut RustObject,
                        cb: extern fn(*mut RustObject, i32)) -> i32;
   fn trigger_callback();
}

fn main() {
#    // Create the object that will be referenced in the callback
    // コールバックから参照されるオブジェクトを作成する
    let mut rust_object = Box::new(RustObject { a: 5 });

    unsafe {
        register_callback(&mut *rust_object, callback);
        trigger_callback();
    }
}
```

これが C のコードです。

```
typedef void (*rust_callback)(void*, int32_t);
void* cb_target;
rust_callback cb;

int32_t register_callback(void* callback_target, rust_callback callback) {
    cb_target = callback_target;
    cb = callback;
    return 1;
}

void trigger_callback() {
  cb(cb_target, 7); // Rust の callback(&rustObject, 7) を呼び出す
}
```

非同期コールバック

先程の例では、コールバックは外部の C ライブラリへの関数呼出しに対する直接の反応として呼びだされました。実行中のスレッドの制御はコールバックの実行のために Rust から C へ、そして Rust へと切り替わりますが、最後には、コールバックはコールバックを引き起こした関数を呼び出したものと同じスレッドで実行されます。

外部のライブラリが独自のスレッドを生成し、そこからコールバックを呼び出すときには、事態はもっと複雑になります。そのような場合、コールバックの中の Rust のデータ構造へのアクセスは特にアンセーフであり、適切な同期メカニズムを使わなければなりません。ミューテックスのような古典的な同期メカニズムの他にも、Rust ではコールバックを呼び出した C のスレッドから Rust のスレッドにデータを転送するために（`std::sync::mpsc` の中の）チャネルを使うという手もあります。

もし、非同期のコールバックが Rust のアドレス空間の中の特別なオブジェクトを対象としていれば、それぞれの Rust のオブジェクトが破壊された後、C のライブラリからそれ以上コールバックが実行されないようにすることが絶対に必要です。これは、オブジェクトのデストラクタでコールバックの登録を解除し、登録解除後にコールバックが実行されないようにライブラリを設計することで実現できます。

リンク

extern ブロックの中の link アトリビュートは、rustc に対してネイティブライブラリをどのようにリンクするかを指示するための基本的な構成ブロックです。今のところ、2 つの形式の link アトリビュートが認められています。

- `#[link(name = "foo")]`
- `#[link(name = "foo", kind = "bar")]`

これらのどちらの形式でも、 foo はリンクするネイティブライブラリの名前で、2 つ目の形式の bar はコンパイラがリンクするネイティブライブラリの種類です。3 つのネイティブライブラリの種類が知られています。

- ダイナミック - `#[link(name = "readline")]`
- スタティック - `#[link(name = "my_build_dependency", kind = "static")]`
- フレームワーク - `#[link(name = "CoreFoundation", kind = "framework")]`

フレームワークは OSX ターゲットでのみ利用可能であることに注意しましょう。

異なる kind の値はリンク時のネイティブライブラリの使われ方の違いを意味します。リンクの視点からすると、Rust コンパイラは 2 種類の生成物を作ります。部分生成物（rlib/staticlib）と最終生成物（dylib/binary）です。ネイティブダイナミックライブラリとフレームワークの依存関係は最終生成物を作るときまで伝播され解決されますが、スタティックライブラリの依存関係は全く伝えません。なぜなら、スタティックライブラリはその後に続く生成物に直接統合されてしまうからです。

このモデルをどのように使うことができるのかという例は次のとおりです。

- ネイティブビルドの依存関係。ときどき、Rust のコードを書くときに C/C++ のグルーが必要になりますが、ライブラリの形式での C/C++ のコードの配布は重荷でしかありません。このような場合、 コードは libfoo.a にアーカイブされ、それから Rust のクレートで`#[link(name = "foo", kind = "static")]` によって依存関係を宣言します。

クレートの成果物の種類にかかわらず、ネイティブスタティックライブラリは成果物に含まれます。これは、ネイティブスタティックライブラリの配布は不要だということを意味します。

- 通常のダイナミックな依存関係。（ readline のような）一般的なシステムライブラリは多くのシステムで利用可能となっていますが、それらのライブラリのスタティックなコピーは見付からないことがしばしばあります。この依存関係を Rust のクレートに含めるときには、（rlib のような）部分生成物のターゲットはライブラリをリンクしませんが、（binary のような）最終生成物に rlib を含めるときには、ネイティブライブラリはリンクされます。

OSX では、フレームワークはダイナミックライブラリと同じ意味で振る舞います。

アンセーフブロック

生ポインタの参照外しやアンセーフであるとマークされた関数の呼出しなど、いくつかの作業はアンセーフブロックの中でのみ許されます。アンセーフブロックはアンセーフ性を隔離し、コンパイラに対してアンセーフ性がブロックの外に漏れ出さないことを約束します。

一方、アンセーフな関数はそれを全世界に向けて広告します。アンセーフな関数はこのように書きます。

```
unsafe fn kaboom(ptr: *const i32) -> i32 { *ptr }
```

この関数は unsafe ブロック又は他の unsafe な関数からのみ呼び出すことができます。

他言語のグローバル変数へのアクセス

他言語 API はしばしばグローバルな状態を追跡するようなことをするためのグローバル変数をエクスポートします。それらの値にアクセスするために、それらを extern ブロックの中で static キーワードを付けて宣言します。

```
# #![feature(libc)]
extern crate libc;

#[link(name = "readline")]
extern {
    static rl_readline_version: libc::c_int;
}

fn main() {
```

```
    println!("You have readline version {} installed.",
             rl_readline_version as i32);
}
```

あるいは、他言語インターフェイスが提供するグローバルな状態を変更しなければならないこともあるかもしれません。これをするために、スタティックな値を変更することができるように mut 付きで宣言することができます。

```
# #![feature(libc)]
extern crate libc;

use std::ffi::CString;
use std::ptr;

#[link(name = "readline")]
extern {
    static mut rl_prompt: *const libc::c_char;
}

fn main() {
    let prompt = CString::new("[my-awesome-shell] $").unwrap();
    unsafe {
        rl_prompt = prompt.as_ptr();

        println!("{:?}", rl_prompt);

        rl_prompt = ptr::null();
    }
}
```

static mut の付いた作用は全て、読込みと書込みの双方についてアンセーフであることに注意しましょう。グローバルでミュータブルな状態の扱いには多大な注意が必要です。

他言語呼出規則

他言語で書かれたコードの多くは C ABI をエクスポートしていて、Rust は他言語関数の呼出しのときのデフォルトとしてそのプラットフォーム上の C の呼出規則を使います。他言語関数の中には、特に Windows API ですが、他の呼出規則を使うものもあります。Rust にはコンパイラに対してどの規則を使うかを教える方法があります。

```
extern crate libc;

#[cfg(all(target_os = "win32", target_arch = "x86"))]
#[link(name = "kernel32")]
#[allow(non_snake_case)]
extern "stdcall" {
    fn SetEnvironmentVariableA(n: *const u8, v: *const u8) -> libc::c_int;
}
```

これは `extern` ブロック全体に適用されます。サポートされている ABI の規則は次のとおりです。

- `stdcall`
- `aapcs`
- `cdecl`
- `fastcall`
- `Rust`
- `rust-intrinsic`
- `system`
- `C`
- `win64`

このリストの ABI のほとんどは名前のとおりですが、 `system` ABI は少し変わっています。この規則はターゲットのライブラリを相互利用するために適切な ABI を選択します。例えば、x86 アーキテクチャの Win32 では、使われる ABI は `stdcall` になります。 しかし、x86_64 では、Windows は `C` の呼出規則を使うので、 `C` が使われます。 先程の例で言えば、`extern "system" { ... }` を使って、x86 のためだけではなく全ての Windows システムのためのブロックを定義することができるということです。

他言語コードの相互利用

`#[repr(C)]` アトリビュートが適用されている場合に限り、Rust は `struct` のレイアウトとそのプラットフォーム上の C での表現方法との互換性を保証します。`#[repr(C, packed)]` を使えば、パディングなしで構造体のメンバをレイアウトすることができます。`#[repr(C)]` は列挙型にも適用することができます。

Rust 独自のボックス（ `Box<T>` ）は包んでいるオブジェクトを指すハンドルとして非ヌルポインタを使います。しかし、それらは内部のアロケータによって管理されるため、手で作るべきではありません。参照は型を直接指す非ヌルポインタとみなすことが安全にできます。しかし、借用チェックやミュータブルについてのルールが破られた場合、安全性は保証されません。生ポインタについてはコンパイラは借用チェックやミュータブルほどには仮定を置かないので、必要なときには、生ポインタ（`*` ）を使いましょう。

ベクタと文字列は基本的なメモリレイアウトを共有していて、 `vec` モジュールと `str` モジュールの中のユーティリティは C API で扱うために使うことができます。 ただし、文字列は`\0` で終わりません。C と相互利用するために NUL 終端の文字列が必要であれば、 `std::ffi` モジュールの `CString` 型を使う必要があります。

crates.io の `libc` クレート[*103]は `libc` モジュール内に C の標準ライブラリの型の別名や関数の定義を含んでいて、Rust は `libc` と `libm` をデフォルトでリンクします。

「ヌルになり得るポインタの最適化」

いくつかの型は非 `null` であると定義されています。このようなものには、参照（ `&T` 、 `&mut T` ）、ボックス（ `Box<T>` ）、そして関数ポインタ（ `extern "abi" fn()` ）があります。C とのインターフェイスにおいては、ヌルになり得るポインタが使われることがしばしばあります。特別な場合として、ジェネリックな `enum` がちょうど 2 つのバリアントを持ち、そのうちの 1 つが値を持っていなくてもう 1 つが単一のフィールドを持っているとき、それは「ヌルになり得るポインタの最適化」の対象になります。そのような列挙型が非ヌルの型でインスタンス化されたとき、それは単一のポインタとして表現され、データを持っていない方のバリアントはヌルポインタとして表現されます。`Option<extern "C" fn(c_int) -> c_int>` は、C ABI で使われるヌルになり得る関数ポインタの表現方法の 1 つです。

[*103] https://crates.io/crates/libc

C からの Rust のコードの呼出し

Rust のコードを C から呼び出せる方法でコンパイルしたいときがあるかもしれません。これは割と簡単ですが、いくつか必要なことがあります。

```
#[no_mangle]
pub extern fn hello_rust() -> *const u8 {
    "Hello, world!\0".as_ptr()
}
```

extern は先程「他言語呼出規則」で議論したように、この関数を C の呼出規則に従うようにします。no_mangle アトリビュートは Rust による名前のマングリングをオフにして、リンクしやすいようにします。

FFI とパニック

FFI を扱うときに panic! に注意することは重要です。FFI の境界をまたぐ panic! の動作は未定義です。もしあなたがパニックし得るコードを書いているのであれば、他のスレッドで実行して、パニックが C に波及しないようにすべきです。

```
use std::thread;

#[no_mangle]
pub extern fn oh_no() -> i32 {
    let h = thread::spawn(|| {
        panic!("Oops!");
    });

    match h.join() {
        Ok(_) => 1,
        Err(_) => 0,
    }
}
```

オペーク構造体の表現

ときどき、C のライブラリが何かのポインタを要求してくるにもかかわらず、その要求されているものの内部的な詳細を教えてくれないことがあります。最も単純な方法は `void *` 引数を使うことです。

```
void foo(void *arg);
void bar(void *arg);
```

Rust ではこれを `c_void` 型で表現することができます。

```
extern crate libc;

extern "C" {
    pub fn foo(arg: *mut libc::c_void);
    pub fn bar(arg: *mut libc::c_void);
}
```

これはその状況に対処するための完全に正当な方法です。しかし、もっとよい方法があります。これを解決するために、いくつかの C ライブラリでは、代わりに `struct` を作っています。そこでは構造体の詳細とメモリレイアウトはプライベートです。これは型の安全性をいくらか満たします。それらの構造体は「オペーク」と呼ばれます。 これが C による例です。

```
struct Foo; /* Foo は構造体だが、その内容は公開インターフェイスの一部ではない */
struct Bar;
void foo(struct Foo *arg);
void bar(struct Bar *arg);
```

これを Rust で実現するために、`enum` で独自のオペーク型を作りましょう。

```
pub enum Foo {}
pub enum Bar {}

extern "C" {
    pub fn foo(arg: *mut Foo);
```

```
    pub fn bar(arg: *mut Bar);
}
```

バリアントなしの enum を使って、バリアントがないためにインスタンス化できないオペーク型を作ります。しかし、 Foo 型と Bar 型は異なる型であり、2 つものの間の型の安全性を満たすので、 Foo のポインタを間違って bar() に渡すことはなくなります。

Borrow と AsRef

Borrow[*104] トレイトと AsRef[*105] トレイトはとてもよく似ていますが違うものです。ここでは 2 つのトレイトの意味を簡単に説明します。

Borrow

Borrow トレイトはデータ構造を書いていて、所有型と借用型を同等に扱いたいときに使います。

例えば、HashMap[*106] には Borrow を使った get メソッド[*107] があります。

```
fn get<Q: ?Sized>(&self, k: &Q) -> Option<&V>
    where K: Borrow<Q>,
          Q: Hash + Eq
```

このシグネチャは少し複雑です。K パラメータに注目してください。これは以下のように HashMap 自身のパラメータになっています。

```
struct HashMap<K, V, S = RandomState> {
```

K パラメータは HashMap の「キー」を表す型です。ここで再び get() のシグネチャを見ると、キーが Borrow<Q> を実装しているときに get() を使えることが分かります。そのため、以下のように String をキーとした HashMap を検索するときに &str を使うことができます。

[*104] http://doc.rust-lang.org/std/borrow/trait.Borrow.html
[*105] http://doc.rust-lang.org/std/convert/trait.AsRef.html
[*106] http://doc.rust-lang.org/std/collections/struct.HashMap.html
[*107] http://doc.rust-lang.org/std/collections/struct.HashMap.html#method.get

```
use std::collections::HashMap;

let mut map = HashMap::new();
map.insert("Foo".to_string(), 42);

assert_eq!(map.get("Foo"), Some(&42));
```

これは標準ライブラリが `impl Borrow<str> for String` を提供しているためです。

所有型か借用型のどちらかを取りたい場合、たいていは `&T` で十分ですが、借用された値が複数種類ある場合 `Borrow` が役に立ちます。特に参照とスライスは `&T` と `&mut T` のいずれも取りうるため、そのどちらも受け入れたい場合は `Borrow` がよいでしょう。

```
use std::borrow::Borrow;
use std::fmt::Display;

fn foo<T: Borrow<i32> + Display>(a: T) {
    println!("a is borrowed: {}", a);
}

let mut i = 5;

foo(&i);
foo(&mut i);
```

上のコードは `a is borrowed: 5` を二度出力します。

AsRef

`AsRef` トレイトは変換用のトレイトです。ジェネリックなコードにおいて、値を参照に変換したい場合に使います。

```
let s = "Hello".to_string();

fn foo<T: AsRef<str>>(s: T) {
```

```
    let slice = s.as_ref();
}
```

どちらを使うべきか

ここまでで見てきた通り、2 つのトレイトは、どちらもある型の所有型バージョンと借用型バージョンの両方を扱う、という意味で同じような種類のものですが、少し違います。

いくつかの異なる種類の借用を抽象化したい場合や、ハッシュ化や比較のために所有型と借用型を同等に扱いたいデータ構造を作る場合は `Borrow` を使ってください。

ジェネリックなコードで値を参照に直接変換したい場合は `AsRef` を使ってください。

リリースチャネル

Rust プロジェクトではリリースを管理するために「リリースチャネル」という考え方を採用しています。どのバージョンの Rust を使用するか決めるためには、この考え方を理解することが重要になります。

概要

Rust のリリースには以下の 3 つのチャネルがあります。

- Nightly
- Beta
- Stable

新しい nightly リリースは毎日作られます。6 週間ごとに、最新の nightly リリースが「Beta」に格上げされます。これ以降は深刻なエラーを修正するパッチのみが受け付けられます。さらに 6 週間後、beta は「Stable」に格上げされ、次の「1.x」リリースになります。

このプロセスは並行して行われます。つまり 6 週間毎の同じ日に、nightly は beta に、beta は stable になります。「1.x」がリリースされると同時に「1.(x + 1)-beta」がリリースされ、nightly は「1.(x + 2)-nightly」の最初のバージョンになる、ということです。

バージョンを選ぶ

一般的に言って、特別な理由がなければ stable リリースチャネルを使うべきです。このリリースは一般のユーザ向けになっています。

しかし Rust に特に関心のある方は、代わりに nightly を選んでも構いません。基本的な交換条件は次の通りです。nightly チャネルを選ぶと不安定で新しいフィーチャを使うことができます。しかし、不安定なフィーチャは変更されやすく、新しい nightly リリースでソースコードが動かなくなってしまうかもしれません。stable リリースを使えば、実験的なフィーチャを使うことはできませんが、Rust のバージョンが上がっても破壊的な変更によるトラブルは起きないでしょう。

CI によるエコシステム支援

beta とはどういうチャネルでしょうか？ stable リリースチャネルを使う全てのユーザは、継続的インテグレーションシステムを使って beta リリースに対してもテストすることを推奨しています。こうすることで、突発的なリグレッションに備えることができます。

さらに、nightly に対してもテストすることでより早くリグレッションを捉えることができます。もし差し支えなければ、この 3 つ目のビルドを含めた全てのチャネルに対してテストしてもらえると嬉しいです。

例えば、多くの Rust プログラマが Travis[*108] をクレートのテストに使っています。(このサービスはオープンソースプロジェクトについては無料で使えます) Travis は Rust を直接サポート[*109]しており、「`.travis.yml`」に以下のように書くことですべてのチャネルに対するテストを行うことができます。

```
language: rust
rust:
  - nightly
  - beta
  - stable

matrix:
```

[*108] https://travis-ci.org/
[*109] http://docs.travis-ci.com/user/languages/rust/

```
  allow_failures:
    - rust: nightly
```

この設定で、Travis は 3 つ全てのチャネルに対してテストを行いますが、nightly で何かおかしくなったとしてもビルドが失敗にはなりません。他の CI システムでも同様の設定をお勧めします。詳細はお使いのシステムのドキュメントを参照してください。

標準ライブラリ無しで Rust を使う

Rust の標準ライブラリは多くの便利な機能を提供している一方で、スレッド、ネットワーク、ヒープアロケーション、その他の多くの機能をホストシステムが提供していることを前提としています。一方で、それらの機能を提供していないシステムも存在します。しかし、Rust はそれらの上でも利用できます！それは、Rust に標準ライブラリを利用しないということを`#![no_std]` アトリビュートを利用して伝えることで可能となります。

> メモ: このフィーチャーは技術的には安定していますが、いくつか注意点があります。例えば、`#![no_std]` を含んだ **ライブラリ** は安定版でビルド可能ですが、 **バイナリ** はビルド不可能です。標準ライブラリを利用しないバイナリについては`#![no_std]` についての不安定版のドキュメントを確認してください。

`#![no_std]` アトリビュートを利用するには、クレートのトップに以下のように追加します:

```
#![no_std]

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

標準ライブラリで提供されている多くの機能は `core` クレート[*110]を用いることでも利用できます。標準ライブラリを利用しているとき、Rust は自動的に `std` をスコープに導入し、標準ライブラリの機能を明示的にインポートすること無しに利用可能にします。それと同じように、もし `#![no_std]` を利用しているときは、Rust は自動的に `core` とそのプレリュード[*111]をスコープに導入します。これは、例えば

[*110] http://doc.rust-lang.org/core/
[*111] http://doc.rust-lang.org/core/prelude/v1/

多くの以下のようなコードが動作することを意味しています:

```
#![no_std]

fn may_fail(failure: bool) -> Result<(), &'static str> {
    if failure {
        Err("this didn't work!")
    } else {
        Ok(())
    }
}
```

6

Nightly Rust

Rust には nightly、beta、stable という 3 種類の配布用チャネルがあります。不安定なフィーチャは nightly の Rust でのみ使えます。詳細は「配布物の安定性[*1]」をご覧ください。

nightly の Rust をインストールするには `rustup.sh` を使って以下のようにします。

```
$ curl -s https://static.rust-lang.org/rustup.sh | sh -s -- --channel=nightly
```

もし `curl | sh` の使用による潜在的な危険性[*2]が気になる場合は、この後の注意書きまで読み進めてください。また、以下のように 2 段階のインストール方法を用い、インストールスクリプトを精査しても構いません。

```
$ curl -f -L https://static.rust-lang.org/rustup.sh -O
$ sh rustup.sh --channel=nightly
```

Windows の場合は 32bit 版インストーラ[*3]あるいは 64bit 版インストーラ[*4]をダウンロードして実行してください。

[*1] http://blog.rust-lang.org/2014/10/30/Stability.html
[*2] http://curlpipesh.tumblr.com
[*3] https://static.rust-lang.org/dist/rust-nightly-i686-pc-windows-gnu.msi
[*4] https://static.rust-lang.org/dist/rust-nightly-x86_64-pc-windows-gnu.msi

アンインストール

もし Rust が不要だと判断した場合、残念ですが仕方がありません。万人に気に入られるプログラミング言語ばかりとは限らないのです。アンインストールするには以下を実行します。

```
$ sudo /usr/local/lib/rustlib/uninstall.sh
```

Windows 用のインストーラを使用した場合は、 `.msi` を再実行しアンインストールオプションを選択してください。

`curl | sh` を行うように書いたことについて、(当然のことですが) 戸惑った方がいるかもしれません。基本的に `curl | sh` を行うということは、Rust をメンテナンスしている善良な人々がコンピュータのハッキングなど何か悪いことをしたりしないと信用する、ということです。ですので戸惑った方はよい直感を持っているといえます。そのような方はソースコードからの Rust ビルド[*5]あるいは公式バイナリのダウンロード[*6]を確認してください。

ここで公式にサポートするプラットフォームについても述べておきます。

- Windows (7、8、Server 2008 R2)
- Linux (2.6.18 以上の各ディストリビューション)、x86 及び x86-64
- OSX 10.7 (Lion) 以上、x86 及び x86-64

Rust ではこれらのプラットフォームの他、Android などいくつかのプラットフォームについても幅広いテストが行われています。しかし、後者については (訳注:実環境での使用実績が少ないという意味で) テストし尽くされているとは言えず、ほとんどの場合動くだろう、という状態です。

最後に Windows についてです。Rust は Windows をリリースする上での 1 級プラットフォームだと考えています。しかし、正直なところ、Windows での使い勝手は Linux/OS X のそれと同等というわけではありません。(そうなるように努力しています!) もしうまく動かないことがあれば、それはバグですのでお知らせください。全てのコミットは他のプラットフォームと同様に Windows に対してもテストされています。

Rust のインストールが終われば、シェルを開いて以下のように入力してください。

[*5] https://github.com/rust-lang/rust#building-from-source
[*6] https://www.rust-lang.org/install.html

```
$ rustc --version
```

バージョン番号、コミットハッシュ、コミット日時、ビルド日時が表示されるはずです。

```
rustc 1.0.0-nightly (f11f3e7ba 2015-01-04) (built 2015-01-06)
```

これで Rust のインストールはうまくいきました!おめでとう!

インストーラはドキュメントのローカルコピーもインストールするので、オフラインでも読むことが出来ます。UNIX システムでは`/usr/local/share/doc/rust` に、Windows では Rust をインストールしたディレクトリの `share\doc` ディレクトリに配置されます。

もし上手くいかないなら様々な場所で助けを得られます。最も簡単なのは Mibbit[*7] からアクセス出来る irc.mozilla.org にある#rust チャネル[*8]です。リンクをクリックしたら他の Rustacean 達 (我々のことをふざけてこう呼ぶのです) にチャットで助けを求めることができます。他にはユーザフォーラム[*9] や Stack Overflow[*10] などがあります。

コンパイラプラグイン

イントロダクション

`rustc` はコンパイラプラグイン、ユーザの提供する構文拡張や構文チェックなどのコンパイラの振舞を拡張するライブラリをロード出来ます。

プラグインとは `rustc` に拡張を登録するための、指定された**登録用** 関数を持った動的ライブラリのクレートです。他のクレートはこれらのプラグインを `#![plugin(...)]` クレートアトリビュートでロード出来ます。プラグインの定義、ロードの仕組みについて詳しくは `rustc_plugin` を参照して下さい。

`#![plugin(foo(... args ...))]` のように渡された引数があるなら、それらは rustc 自身によっては解釈されません。`Registry` の `args` メソッドを通じてプラグインに渡されます。

ほとんどの場合で、プラグインは `#![plugin]` を通じて**のみ** 使われるべきで、 `extern crate` を通じて使われるべきではありません。プラグインをリンクすると libsyntax と librustc の全てをクレートの依

[*7] http://chat.mibbit.com/?server=irc.mozilla.org&channel=%23rust
[*8] irc://irc.mozilla.org/#rust
[*9] https://users.rust-lang.org/
[*10] http://stackoverflow.com/questions/tagged/rust

存に引き込んでしまいます。これは別のプラグインを作っているのでもない限り一般的には望まぬ挙動です。plugin_as_library チェッカによってこのガイドラインは検査されます。

普通の慣行ではコンパイラプラグインはそれ専用のクレートに置かれて、macro_rules! マクロやコンシューマが使うライブラリのコードとは分けられます。

構文拡張

プラグインは Rust の構文を様々な方法で拡張出来ます。構文拡張の 1 つに手続的マクロがあります。これらは普通のマクロと同じように実行されますが展開は任意の構文木をコンパイル時に操作する Rust のコードが行います。

ローマ数字リテラルを実装する roman_numerals.rs[*11] を書いてみましょう。

```
#![crate_type="dylib"]
#![feature(plugin_registrar, rustc_private)]

extern crate syntax;
extern crate rustc;
extern crate rustc_plugin;

use syntax::codemap::Span;
use syntax::parse::token;
use syntax::ast::TokenTree;
use syntax::ext::base::{ExtCtxt, MacResult, DummyResult, MacEager};
# // use syntax::ext::build::AstBuilder;  // trait for expr_usize
use syntax::ext::build::AstBuilder;  // expr_usize のトレイト
use rustc_plugin::Registry;

fn expand_rn(cx: &mut ExtCtxt, sp: Span, args: &[TokenTree])
        -> Box<MacResult + 'static> {

    static NUMERALS: &'static [(&'static str, usize)] = &[
        ("M", 1000), ("CM", 900), ("D", 500), ("CD", 400),
```

[*11] https://github.com/rust-lang/rust/tree/master/src/test/auxiliary/roman_numerals.rs

```
        ("C",  100), ("XC",  90), ("L",  50), ("XL",  40),
        ("X",   10), ("IX",   9), ("V",   5), ("IV",   4),
        ("I",    1)];

    if args.len() != 1 {
        cx.span_err(
            sp,
            &format!("argument should be a single identifier, but got {} arguments", a
rgs.len()));
        return DummyResult::any(sp);
    }

    let text = match args[0] {
        TokenTree::Token(_, token::Ident(s, _)) => s.to_string(),
        _ => {
            cx.span_err(sp, "argument should be a single identifier");
            return DummyResult::any(sp);
        }
    };

    let mut text = &*text;
    let mut total = 0;
    while !text.is_empty() {
        match NUMERALS.iter().find(|&&(rn, _)| text.starts_with(rn)) {
            Some(&(rn, val)) => {
                total += val;
                text = &text[rn.len()..];
            }
            None => {
                cx.span_err(sp, "invalid Roman numeral");
                return DummyResult::any(sp);
            }
        }
    }
```

```
    MacEager::expr(cx.expr_usize(sp, total))
}

#[plugin_registrar]
pub fn plugin_registrar(reg: &mut Registry) {
    reg.register_macro("rn", expand_rn);
}
```

`rn!()` マクロを他の任意のマクロと同じように使えます。

```
#![feature(plugin)]
#![plugin(roman_numerals)]

fn main() {
    assert_eq!(rn!(MMXV), 2015);
}
```

単純な `fn(&str) -> u32` に対する利点は

- （任意に複雑な）変換がコンパイル時に行なわれる
- 入力バリデーションもコンパイル時に行なわれる
- パターンで使えるように拡張出来るので、実質的に任意のデータ型に対して新たなリテラル構文を与えられる

手続き的マクロに加えて `derive`[*12]ライクなアトリビュートや他の拡張を書けます。`Registry::register_syntax_extension` や `SyntaxExtension` 列挙型を参照して下さい。もっと複雑なマクロの例は `regex_macros`[*13]を参照して下さい。

ヒントと小技

マクロデバッグのヒントのいくつかが使えます。

`syntax::parse` を使うことでトークン木を式などの高レベルな構文要素に変換出来ます。

[*12] http://doc.rust-lang.org/reference.html#derive
[*13] https://github.com/rust-lang/regex/blob/master/regex_macros/src/lib.rs

```
fn expand_foo(cx: &mut ExtCtxt, sp: Span, args: &[TokenTree])
        -> Box<MacResult+'static> {

    let mut parser = cx.new_parser_from_tts(args);

    let expr: P<Expr> = parser.parse_expr();
```

libsyntax のパーサのコード[*14]を見るとパーサの基盤がどのように機能しているかを感られるでしょう。

パースしたものの Span は良いエラー報告のために保持しておきましょう。自分で作ったデータ構造に対しても Spanned でラップ出来ます。

ExtCtxt::span_fatal を呼ぶとコンパイルは即座に中断されます。ExtCtxt::span_err を呼んで DummyResult を返せばコンパイラはさらなるエラーを発見できるのでその方が良いでしょう。

構文の断片を表示するには span_note と syntax::print::pprust::*_to_string を使えば出来ます。

上記の例では AstBuilder::expr_usize を使って整数リテラルを作りました。 AstBuilder トレイトの代替として libsyntax は準クォートマクロを提供しています。ドキュメントがない上に荒削りです。しかしながらその実装は改良版の普通のプラグインライブラリのとっかかりにはほど良いでしょう。

構文チェックプラグイン

プラグインによって Rust の構文チェック基盤[*15]を拡張してコーディングスタイル、安全性などを検査するようにできます。では lint_plugin_test.rs[*16] プラグインを書いてみましょう。lintme という名前のアイテムについて警告を出すものです。

```
#![feature(plugin_registrar)]
#![feature(box_syntax, rustc_private)]

extern crate syntax;

# // Load rustc as a plugin to get macros
// macro を使うために rustc をプラグインとして読み込む
```

[*14] https://github.com/rust-lang/rust/blob/master/src/libsyntax/parse/parser.rs
[*15] http://doc.rust-lang.org/reference.html#lint-check-attributes
[*16] https://github.com/rust-lang/rust/blob/master/src/test/auxiliary/lint_plugin_test.rs

```
#[macro_use]
extern crate rustc;
extern crate rustc_plugin;

use rustc::lint::{EarlyContext, LintContext, LintPass, EarlyLintPass,
                  EarlyLintPassObject, LintArray};
use rustc_plugin::Registry;
use syntax::ast;

declare_lint!(TEST_LINT, Warn, "Warn about items named 'lintme'");

struct Pass;

impl LintPass for Pass {
    fn get_lints(&self) -> LintArray {
        lint_array!(TEST_LINT)
    }
}

impl EarlyLintPass for Pass {
    fn check_item(&mut self, cx: &EarlyContext, it: &ast::Item) {
        if it.ident.name.as_str() == "lintme" {
            cx.span_lint(TEST_LINT, it.span, "item is named 'lintme'");
        }
    }
}

#[plugin_registrar]
pub fn plugin_registrar(reg: &mut Registry) {
    reg.register_early_lint_pass(box Pass as EarlyLintPassObject);
}
```

そしたらこのようなコードは

```
#![plugin(lint_plugin_test)]
```

```
fn lintme() { }
```

コンパイラの警告を発生させます。

```
foo.rs:4:1: 4:16 warning: item is named 'lintme', #[warn(test_lint)] on by default
foo.rs:4 fn lintme() { }
         ^~~~~~~~~~~~~~~
```

構文チェックプラグインのコンポーネントは

- 1 回以上の `declare_lint!` の実行。それによって `Lint` 構造体が定義されます。
- 構文チェックパスで必要となる状態を保持する構造体（ここでは何もない）
- それぞれの構文要素をどうやってチェックするかを定めた `LintPass` の実装。 単一の `LintPass` は複数回 `span_lint` をいくつかの異なる `Lint` に対して呼ぶかもしれませんが、全て `get_lints` を通じて登録すべきです。

構文チェックパスは構文巡回ですが、型情報が得られる、コンパイルの終盤で走ります。`rustc` の組み込み構文チェック[*17]は殆どプラグインと同じ基盤を使っており、どうやって型情報にアクセスするかの例になっています。

プラグインによって定義された Lint は普通のアトリビュートとコンパイラフラグ[*18]例えば `#[allow(test_lint)]` や `-A test-lint` によってコントロールされます。 これらの識別子は `declare_lint!` の第一引数に由来しており、適切な名前に変換されます。

`rustc -W help foo.rs` を走らせることで `rustc` の知っている、及び `foo.rs` 内で定義されたコンパイラ構文チェックをロード出来ます。

インラインアセンブリ

極めて低レベルな技巧やパフォーマンス上の理由から、CPU を直接コントロールしたいと思う人もいるでしょう。Rust はそのような処理を行うために、インラインアセンブリを `asm!` マクロによってサポー

[*17] https://github.com/rust-lang/rust/blob/master/src/librustc/lint/builtin.rs
[*18] http://doc.rust-lang.org/reference.html#lint-check-attributes

トしています。

```
# // asm!(assembly template
asm!(アセンブリのテンプレート
# //   : output operands
   : 出力オペランド
# //   : input operands
   : 入力オペランド
# //   : clobbers
   : 破壊されるデータ
# //   : options
   : オプション
   );
```

asm のいかなる利用もフィーチャーゲートの対象です（利用するには#![feature(asm)] がクレートに必要になります）。そしてもちろん unsafe ブロックも必要です。

> メモ: ここでの例は x86/x86-64 のアセンブリで示されますが、すべてのプラットフォームがサポートされています。

アセンブリテンプレート

アセンブリテンプレートのみが要求されるパラメータであり、文字列リテラル (例: ””) である必要があります。

```
#![feature(asm)]

#[cfg(any(target_arch = ”x86”, target_arch = ”x86_64”))]
fn foo() {
    unsafe {
        asm!(”NOP”);
    }
}

// その他のプラットフォーム
```

```
#[cfg(not(any(target_arch = "x86", target_arch = "x86_64")))]
fn foo() { /* ... */ }

fn main() {
    // ...
    foo();
    // ...
}
```

以後は、 `feature(asm)` と `#[cfg]` は省略して示します。

出力オペランド、入力オペランド、破壊されるデータ、オプションはすべて省略可能ですが、省略する場合でも正しい数の: を書く必要があります。

```
asm!("xor %eax, %eax"
    :
    :
    : "{eax}"
   );
```

空白も必要ではありません:

```
asm!("xor %eax, %eax" ::: "{eax}");
```

オペランド

入力と出力のオペランドは、 `: "制約 1"(式 1), "制約 2"(式 2), ...` というフォーマットに従います。出力オペランドの式は変更可能な左辺値か、アサインされていない状態でなければなりません。

```
fn add(a: i32, b: i32) -> i32 {
    let c: i32;
    unsafe {
        asm!("add $2, $0"
             : "=r"(c)
             : "0"(a), "r"(b)
```

```
            );
    }
    c
}

fn main() {
    assert_eq!(add(3, 14159), 14162)
}
```

もし本当のオペランドをここで利用したい場合、波括弧 `{}` で利用したいレジスタの周りを囲む必要があり、また、オペランドの特有のサイズを書く必要があります。これは、どのレジスタを利用するかが重要となる、ごく低レベルのプログラミングで有用です。

```
let result: u8;
asm!("in %dx, %al" : "={al}"(result) : "{dx}"(port));
result
```

破壊されるデータ

いくつかのインストラクションは異なる値を持っている可能性のあるレジスタを変更する事があります。そのため、コンパイラがそれらのレジスタに格納された値が処理後にも有効であると思わないように、破壊されるデータのリストを利用します。

```
// eax に 0x200 を格納します
asm!("mov $$0x200, %eax" : /* 出力なし */ : /* 入力無し */ : "{eax}");
```

入力と出力のレジスタは変更される可能性があることが制約によってすでに伝わっているために、リストに載せる必要はありません。それ以外では、その他の暗黙的、明示的に利用されるレジスタをリストに載せる必要があります。

もしアセンブリが条件コードを変更する場合、レジスタ `cc` も破壊されるデータのリストに指定する必要があります。同様に、もしアセンブリがメモリを変更する場合 `memory` もリストに指定する必要があります。

オプション

最後のセクション、 `options` は Rust 特有のものです。`options` の形式は、コンマで区切られた文字列リテラルのリスト（例: `:"foo", "bar", "baz"`）です。これはインラインアセンブリについての追加の情報を指定するために利用されます:

現在有効なオプションは以下の通りです:

1. *volatile* - このオプションを指定することは、gcc/clang で`__asm__ __volatile__ (...)` を指定することと類似しています。
2. *alignstack* - いくつかのインストラクションはスタックが決まった方式（例: SSE）でアラインされていることを期待しています。このオプションを指定することはコンパイラに通常のスタックをアラインメントするコードの挿入を指示します。
3. *intel* - デフォルトの AT&T 構文の代わりにインテル構文を利用することを意味しています。

```
let result: i32;
unsafe {
   asm!("mov eax, 2" : "={eax}"(result) : : : "intel")
}
println!("eax is currently {}", result);
```

さらなる情報

現在の `asm!` マクロの実装は LLVM のインラインアセンブリ表現[*19]への直接的なバインディングです。そのため破壊されるデータのリストや、制約、その他の情報について LLVM のドキュメント[*20]を確認してください。

No stdlib

%No stdlib

Rust の標準ライブラリはたくさんの有用な機能を提供していますが、スレッド、ネットワーク、ヒープ割り当てなど、さまざまな機能についてホストシステムのサポートがあることを前提としています。し

[*19] http://llvm.org/docs/LangRef.html#inline-assembler-expressions
[*20] http://llvm.org/docs/LangRef.html#inline-assembler-expressions

かし、それらの機能を持たないシステムは存在しますし、Rust はそういったシステム上でも動作します。そのためには、`#![no_std]` アトリビュートを使って標準ライブラリを使用しないことを示します。

> 注記: このフィーチャは技術的には安定していますが、いくつかの注意点があります。例えば、`#![no_std]` を含んだ **ライブラリ**は安定版でビルドできますが、**バイナリ**はそうではありません。標準ライブラリを使用しないライブラリについての詳細は`#![no_std]` についてのドキュメントを参照してください。

当然のことですが、ライブラリだけが全てではなく、実行可能形式においても`#[no_std]` は使用できます。このときエントリポイントを指定する方法には 2 種類あり、1 つは`#[start]` アトリビュート、もう 1 つは C の `main` 関数の上書きです。

`#[start]` のついた関数へは C と同じフォーマットでコマンドライン引数が渡されます。

```
#![feature(lang_items)]
#![feature(start)]
#![no_std]

// crt0.o が必要としていると思われるもののためにシステムの libc ライブラリを参照します。
extern crate libc;

// このプログラムのエントリポイントです
#[start]
fn start(_argc: isize, _argv: *const *const u8) -> isize {
    0
}

// これらの関数とトレイトは必要最小限の hello world のようなプログラムが
// 使うのではなく、コンパイラが使います。通常これらは libstd により提供されます。
#[lang = "eh_personality"] extern fn eh_personality() {}
#[lang = "panic_fmt"] extern fn panic_fmt() -> ! { loop {} }
```

コンパイラによって挿入される `main` を上書きするには、まず`#![no_main]` によってコンパイラによる挿入を無効にします。その上で、正しい ABI と正しい名前を備えたシンボルを作成します。これにはコンパイラの名前マングリングを上書きする必要もあります。

```
#![feature(lang_items)]
#![feature(start)]
#![no_std]
#![no_main]

extern crate libc;

#[no_mangle] // 出力結果においてこのシンボル名が `main` になることを保証します。

pub extern fn main(argc: i32, argv: *const *const u8) -> i32 {
    0
}

#[lang = "eh_personality"] extern fn eh_personality() {}
#[lang = "panic_fmt"] extern fn panic_fmt() -> ! { loop {} }
```

今のところ、コンパイラは実行可能形式においていくつかのシンボルが呼び出し可能であるという前提を置いています。通常、これらの関数は標準ライブラリが提供しますが、それを使わない場合自分で定義しなければなりません。

2 つある関数のうち 1 つ目は `eh_personality` で、コンパイラの失敗メカニズムに使われます。これはしばしば GCC の personality 関数に割り当てられますが（詳細は libstd 実装[*21]を参照してください）、パニックを発生させないクレートではこの関数は呼ばれないことが保証されています。2 つ目の関数は `panic_fmt` で、こちらもコンパイラの失敗メカニズムのために使われます。

Intrinsic

> メモ: intrinsics のインタフェースは常に不安定です、intrinsics を直接利用するのではなく、libcore の安定なインタフェースを利用することを推奨します。

intrinsics は特別な ABI `rust-intrinsic` を用いて、FFI の関数で有るかのようにインポートされます。例えば、独立したコンテキストの中で型の間の `transmute` をしたい場合や、効率的なポインタ演算を行いたい場合、それらの関数を以下のような宣言を通してインポートします

[*21] https://github.com/rust-lang/rust/blob/master/src/libstd/sys/common/unwind/gcc.rs

```
#![feature(intrinsics)]

extern "rust-intrinsic" {
    fn transmute<T, U>(x: T) -> U;

    fn offset<T>(dst: *const T, offset: isize) -> *const T;
}
```

他の FFI 関数と同様に、呼出は常に `unsafe` です。

言語アイテム

> **注意**：言語アイテムは大抵 Rust の配布物内のクレートから提供されていますし言語アイテムのインターフェース自体安定していません。自身で言語アイテムを定義するのではなく公式の配布物のクレートを使うことが推奨されています。

`rustc` コンパイラはあるプラガブルな操作、つまり機能が言語にハードコードされているのではなくライブラリで実装されているものを持っており、特別なマーカによってそれが存在することをコンパイラに伝えます。マーカとは `#[lang = "..."]` アトリビュートで、 `...` には様々な値が、つまり様々な「言語アイテム」あります。

例えば、 `Box` ポインタは 2 つの言語アイテムを必要とします。1 つはアロケーションのため、もう 1 つはデアロケーションのため。フリースタンディング環境で動くプログラムは `Box` を `malloc` と `free` による動的アロケーションの糖衣として使います。

```
#![feature(lang_items, box_syntax, start, libc)]
#![no_std]

extern crate libc;

extern {
    fn abort() -> !;
}

#[lang = "owned_box"]
```

```
pub struct Box<T>(*mut T);

#[lang = "exchange_malloc"]
unsafe fn allocate(size: usize, _align: usize) -> *mut u8 {
    let p = libc::malloc(size as libc::size_t) as *mut u8;

    // malloc failed
    if p as usize == 0 {
        abort();
    }

    p
}

#[lang = "exchange_free"]
unsafe fn deallocate(ptr: *mut u8, _size: usize, _align: usize) {
    libc::free(ptr as *mut libc::c_void)
}

#[lang = "box_free"]
unsafe fn box_free<T>(ptr: *mut T) {
    deallocate(ptr as *mut u8, ::core::mem::size_of::<T>(), ::core::mem::align_of::<T>
   ());
}

#[start]
fn main(argc: isize, argv: *const *const u8) -> isize {
    let x = box 1;

    0
}

#[lang = "eh_personality"] extern fn eh_personality() {}
#[lang = "panic_fmt"] fn panic_fmt() -> ! { loop {} }
```

abort を使ってることに注意して下さい: exchange_malloc 言語アイテムは有効なポインタを返すものとされており、内部でその検査をする必要があるのです。

言語アイテムによって提供される機能には以下のようなものがあります。:

- トレイトによるオーバーロード可能な演算子: == 、< 、 参照外し（ * ） そして + （など）の演算子は全て言語アイテムでマークされています。これら 4 つはそれぞれ eq 、 ord 、 deref 、add です
- スタックの巻き戻しと一般の失敗は eh_personality 、fail そして fail_bounds_check 言語アイテムです。
- std::marker 内のトレイトで様々な型を示すのに使われています。send 、 sync そして copy 言語アイテム。
- マーカ型と std::marker にある変性指示子。covariant_type 、 contravariant_lifetime 言語アイテムなどなど。

言語アイテムはコンパイラによって必要に応じてロードされます、例えば、Box を一度も使わないなら exchange_malloc と exchange_free の関数を定義する必要はありません。rustc はアイテムが必要なのに現在のクレートあるいはその依存するクレート内で見付からないときにエラーを出します。

高度なリンキング

Rust におけるリンクの一般的なケースについては本書の前の方で説明しましたが、他言語から利用できるような幅広いリンクをサポートすることは、ネイティブライブラリとのシームレスな相互利用を実現するために、Rust にとって重要です。

リンク引数

どのようにリンクをカスタマイズするかを rustc に指示するために、1 つの方法があります。それは、link_args アトリビュートを使うことです。 このアトリビュートは extern ブロックに適用され、生成物を作るときにリンカに渡したいフラグをそのまま指定します。使い方の例は次のようになります。

```
#![feature(link_args)]

#[link_args = ”-foo -bar -baz”]
extern {}
```

```
# fn main() {}
```

これはリンクを実行するための認められた方法ではないため、この機能は現在 `feature(link_args)` ゲートによって隠されているということに注意しましょう。 今は `rustc` がシステムリンカ（多くのシステムでは `gcc` 、MSVC では `link.exe` ）に渡すので、追加のコマンドライン引数を提供することには意味がありますが、それが今後もそうだとは限りません。将来、 `rustc` がネイティブライブラリをリンクするために LLVM を直接使うようになるかもしれませんし、そのような場合には `link_args` は意味がなくなるでしょう。 `rustc` に`-C link-args` 引数をつけることで、 `link_args` アトリビュートと同じような効果を得ることができます。

このアトリビュートは使わ ないことが強く推奨されているので、代わりにもっと正式な`#[link(...)]` アトリビュートを `extern` ブロックに使いましょう。

スタティックリンク

スタティックリンクとは全ての必要なライブラリを含めた成果物を生成する手順のことで、そうすればコンパイルされたプロジェクトを使いたいシステム全てにライブラリをインストールする必要がなくなります。Rust のみで構築された依存関係はデフォルトでスタティックリンクされます。そのため、Rust をインストールしなくても、作成されたバイナリやライブラリを使うことができます。対照的に、ネイティブライブラリ（例えば `libc` や `libm` ）はダイナミックリンクされるのが普通です。しかし、これを変更してそれらを同様にスタティックリンクすることも可能です。

リンクは非常にプラットフォームに依存した話題であり、スタティックリンクのできないプラットフォームすらあるかもしれません! このセクションは選んだプラットフォームにおけるリンクについての基本が理解できていることを前提とします。

Linux

デフォルトでは、Linux 上の全ての Rust のプログラムはシステムの `libc` とその他のいくつものライブラリとリンクされます。 GCC と `glibc` （Linux における最も一般的な `libc` ）を使った 64 ビット Linux マシンでの例を見てみましょう。

```
$ cat example.rs
fn main() {}
$ rustc example.rs
$ ldd example
```

```
        linux-vdso.so.1 =>  (0x00007ffd565fd000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fa81889c000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007fa81867e000)
        librt.so.1 => /lib/x86_64-linux-gnu/librt.so.1 (0x00007fa818475000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007fa81825f000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fa817e9a000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fa818cf9000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fa817b93000)
```

古いシステムで新しいライブラリの機能を使いたいときや、実行するプログラムに必要な依存関係を満たさないシステムをターゲットにしたいときは、Linux におけるダイナミックリンクは望ましくないかもしれません。

スタティックリンクは代わりの `libc` である `musl`[*22] によってサポートされています。 以下の手順に従い、 `musl` を有効にした独自バージョンの Rust をコンパイルして独自のディレクトリにインストールすることができます。

```
$ mkdir musldist
$ PREFIX=$(pwd)/musldist
$
$ # Build musl
$ curl -O http://www.musl-libc.org/releases/musl-1.1.10.tar.gz
$ tar xf musl-1.1.10.tar.gz
$ cd musl-1.1.10/
musl-1.1.10 $ ./configure --disable-shared --prefix=$PREFIX
musl-1.1.10 $ make
musl-1.1.10 $ make install
musl-1.1.10 $ cd ..
$ du -h musldist/lib/libc.a
2.2M    musldist/lib/libc.a
$
$ # Build libunwind.a
$ curl -O http://llvm.org/releases/3.7.0/llvm-3.7.0.src.tar.xz
$ tar xf llvm-3.7.0.src.tar.xz
```

[*22] http://www.musl-libc.org/

```
$ cd llvm-3.7.0.src/projects/
llvm-3.7.0.src/projects $ curl http://llvm.org/releases/3.7.0/libunwind-3.7.0.src.tar.
   xz | tar xJf -
llvm-3.7.0.src/projects $ mv libunwind-3.7.0.src libunwind
llvm-3.7.0.src/projects $ mkdir libunwind/build
llvm-3.7.0.src/projects $ cd libunwind/build
llvm-3.7.0.src/projects/libunwind/build $ cmake -DLLVM_PATH=../../.. -DLIBUNWIND_ENABL
   E_SHARED=0 ..
llvm-3.7.0.src/projects/libunwind/build $ make
llvm-3.7.0.src/projects/libunwind/build $ cp lib/libunwind.a $PREFIX/lib/
llvm-3.7.0.src/projects/libunwind/build $ cd ../../../../
$ du -h musldist/lib/libunwind.a
164K    musldist/lib/libunwind.a
$
$ # Build musl-enabled rust
$ git clone https://github.com/rust-lang/rust.git muslrust
$ cd muslrust
muslrust $ ./configure --target=x86_64-unknown-linux-musl --musl-root=$PREFIX --prefix
   =$PREFIX
muslrust $ make
muslrust $ make install
muslrust $ cd ..
$ du -h musldist/bin/rustc
12K     musldist/bin/rustc
```

これで `musl` が有効になった Rust のビルドが手に入りました! 独自のプレフィックスを付けてインストールしたので、試しに実行するときにはシステムがバイナリと適切なライブラリを見付けられることを確かめなければなりません。

```
$ export PATH=$PREFIX/bin:$PATH
$ export LD_LIBRARY_PATH=$PREFIX/lib:$LD_LIBRARY_PATH
```

試してみましょう!

```
$ echo 'fn main() { println!(”hi!”); panic!(”failed”); }' > example.rs
```

```
$ rustc --target=x86_64-unknown-linux-musl example.rs
$ ldd example
        not a dynamic executable
$ ./example
hi!
thread '<main>' panicked at 'failed', example.rs:1
```

成功しました! このバイナリは同じマシンアーキテクチャであればほとんど全ての Linux マシンにコピーして問題なく実行することができます。

`cargo build` も `–target` オプションを受け付けるので、あなたのクレートも普通にビルドできるはずです。ただし、リンクする前にネイティブライブラリを `musl` 向けにリコンパイルする必要はあるかもしれません。

ベンチマークテスト

Rust はコードのパフォーマンスをテストできるベンチマークテストをサポートしています。早速、`src/lib.rc` を以下のように作っていきましょう (コメントは省略しています):

```
#![feature(test)]

extern crate test;

pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;
    use test::Bencher;

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
```

```
    }

    #[bench]
    fn bench_add_two(b: &mut Bencher) {
        b.iter(|| add_two(2));
    }
}
```

不安定なベンチマークのフィーチャを有効にするため、 `test` フィーチャゲートを利用していることに注意して下さい。

ベンチマークテストのサポートを含んだ `test` クレートをインポートしています。 また、 `bench` アトリビュートのついた新しい関数を定義しています。引数を取らない通常のテストとは異なり、ベンチマークテストは`&mut Bencher` を引数に取ります。 `Bencher` はベンチマークしたいコードを含んだクロージャを引数に取る `iter` メソッドを提供しています。

ベンチマークテストは以下のように `cargo bench` のようにして実施できます:

```
$ cargo bench
   Compiling adder v0.0.1 (file:///home/steve/tmp/adder)
     Running target/release/adder-91b3e234d4ed382a

running 2 tests
test tests::it_works ... ignored
test tests::bench_add_two ... bench:         1 ns/iter (+/- 0)

test result: ok. 0 passed; 0 failed; 1 ignored; 1 measured
```

ベンチマークでないテストは無視されます。 `cargo bench` が `cargo test` よりも時間がかかることにお気づきになったかもしれません。これは、Rust がベンチマークをかなりの回数繰り返し実行し、その結果の平均を取るためです。今回のコードでは非常に小さな処理しか行っていないために、`1 ns/iter (+/- 0)` という結果を得ました、しかし、この結果は変動することがあるでしょう。

以下は、ベンチマークを書くときのアドバイスです:

- セットアップのコードを `iter` の外に移し、計測したい箇所のみを `iter` の中に書きましょう。

- それぞれの繰り返しでコードが「同じこと」をするようにし、集計をしたり状態を変更したりといったことはしないで下さい。
- 利用している外部の関数についても冪等にしましょう、ベンチマークはその関数をおそらく何度も実行します。
- 内側の `iter` ループを短く高速にしましょう、そうすることでベンチマークの実行は高速になり、キャリブレータは実行の長さをより良い精度で補正できるようになります。
- パフォーマンスの向上 (または低下) をピンポイントで突き止められるように、`iter` ループ中のコードの処理を簡潔にしましょう。

注意点: 最適化

ベンチマークを書くときに気をつけなければならないその他の点は: 最適化を有効にしてコンパイルしたベンチマークは劇的に最適化され、もはや本来ベンチマークしたかったコードとは異なるという点です。たとえば、コンパイラは幾つかの計算がなにも外部に影響を及ぼさないことを認識してそれらの計算を取り除くかもしれません。

```
#![feature(test)]

extern crate test;
use test::Bencher;

#[bench]
fn bench_xor_1000_ints(b: &mut Bencher) {
    b.iter(|| {
        (0..1000).fold(0, |old, new| old ^ new);
    });
}
```

このベンチマークは以下の様な結果となります

```
running 1 test
test bench_xor_1000_ints ... bench:         0 ns/iter (+/- 0)

test result: ok. 0 passed; 0 failed; 0 ignored; 1 measured
```

ベンチマークランナーはこの問題を避ける 2 つの手段を提供します。`iter` メソッドが受け取るクロージャは任意の値を返すことができ、オプティマイザに計算の結果が利用されていると考えさせ、その計算を取り除くことができないと保証することができます。これは、上のコードにおいて `b.iter` の呼出を以下のようにすることで可能です:

```
b.iter(|| {
    // `;` が無いことに注意して下さい (明示的な `return` を使うこともできます)。
    (0..1000).fold(0, |old, new| old ^ new)
});
```

もう一つの方法としては、ジェネリックな `test::black_box` 関数を呼び出すという手段が有ります、`test::black_box` 関数はオプティマイザにとって不透明な「ブラックボックス」であり、オプティマイザに引数のどれもが利用されていると考えさせることができます。

```
#![feature(test)]

extern crate test;

b.iter(|| {
    let n = test::black_box(1000);

    (0..n).fold(0, |a, b| a ^ b)
})
```

2 つの手段のどちらも値を読んだり変更したりせず、小さな値に対して非常に低コストです。大きな値は、オーバーヘッドを減らすために間接的に渡すことができます (例: `black_box(&huge_struct)`)。

上記のどちらかの変更を施すことでベンチマークの結果は以下のようになります

```
running 1 test
test bench_xor_1000_ints ... bench:       131 ns/iter (+/- 3)

test result: ok. 0 passed; 0 failed; 0 ignored; 1 measured
```

しかしながら、上のどちらかの方法をとったとしても依然オプティマイザはテストケースを望まない形で変更する場合があります。

Box 構文とパターン

今のところ、安定版において `Box` を作成する唯一の方法は `Box::new` メソッドです。安定版の Rust ではパターンマッチで `Box` を分解することもできません。不安定版の `box` キーワードは `Box` の作成と分解の両方に使えます。使い方は以下の通りです。

```
#![feature(box_syntax, box_patterns)]

fn main() {
    let b = Some(box 5);
    match b {
        Some(box n) if n < 0 => {
            println!("Box contains negative number {}", n);
        },
        Some(box n) if n >= 0 => {
            println!("Box contains non-negative number {}", n);
        },
        None => {
            println!("No box");
        },
        _ => unreachable!()
    }
}
```

注記: 将来的にこの構文は変わる可能性があるため、現時点でこれらのフィーチャは `box_syntax` (box の作成)、 `box_patterns` (分解とパターンマッチ) を明示しなければ使えません。

ポインタ返し

ポインタを持つ多くのプログラミング言語では、巨大なデータ構造のコピーを避けるため、関数からポインタを返してきました。例えば以下のように書くことができます。

```
struct BigStruct {
    one: i32,
    two: i32,
    // etc
    one_hundred: i32,
}

fn foo(x: Box<BigStruct>) -> Box<BigStruct> {
    Box::new(*x)
}

fn main() {
    let x = Box::new(BigStruct {
        one: 1,
        two: 2,
        one_hundred: 100,
    });

    let y = foo(x);
}
```

考え方としては、box で渡すことで `BigStruct` を構成する 100 個の `i32` の代わりにポインタのみのコピーで済む、というものです。

これは Rust ではアンチパターンです。代わりに以下のように書きます。

```
#![feature(box_syntax)]

struct BigStruct {
    one: i32,
    two: i32,
    // etc
    one_hundred: i32,
}
```

```
fn foo(x: Box<BigStruct>) -> BigStruct {
    *x
}

fn main() {
    let x = Box::new(BigStruct {
        one: 1,
        two: 2,
        one_hundred: 100,
    });

    let y: Box<BigStruct> = box foo(x);
}
```

このように書くことでパフォーマンスを犠牲にすることなく、柔軟性を確保することができます。

このコードはひどいパフォーマンス低下をもたらすと感じるかもしれません。値を返して即座に box に入れるなんて?! このパターンは悪いところどりになっているのでは? Rust はもう少し賢いため、このコードでコピーは発生しません。 `main` 内では `box` のために十分なメモリ領域を確保し、そのメモリへのポインタを `foo` へ `x` として渡します。 `foo` はその値を直接 `Box<T>` (訳注: すなわち `y`) の中に書き込みます。

重要なことなので繰り返しますが、ポインタを戻り値の最適化のために使うべきではありません。呼び出し側が戻り値をどのように使うかを選択できるようにしましょう。

スライスパターン

スライスや配列に対してマッチを行いたい場合、 `slice_patterns` フィーチャを有効にすると以下のように `&` を使うことができます。

```
#![feature(slice_patterns)]

fn main() {
    let v = vec!["match_this", "1"];
```

```
    match &v[..] {
        ["match_this", second] => println!("The second element is {}", second),
        _ => {},
    }
}
```

advanced_slice_patterns フィーチャを有効にすると、スライスにマッチするパターンの中で .. を使ってその要素の数が任意であることを示すことができます。このワイルドカードは与えるパターン配列の中で一度だけ使うことができます。もし.. の前に識別子 (訳注: 以下の例では inside) があれば、そのスライスの結果はその識別子名に束縛されます。例えば以下のようになります。

```
#![feature(advanced_slice_patterns, slice_patterns)]

fn is_symmetric(list: &[u32]) -> bool {
    match list {
        [] | [_] => true,
        [x, inside.., y] if x == y => is_symmetric(inside),
        _ => false
    }
}

fn main() {
    let sym = &[0, 1, 4, 2, 4, 1, 0];
    assert!(is_symmetric(sym));

    let not_sym = &[0, 1, 7, 2, 4, 1, 0];
    assert!(!is_symmetric(not_sym));
}
```

関連定数

associated_consts フィーチャを使うと、以下のように定数を定義することができます。

```
#![feature(associated_consts)]

trait Foo {
    const ID: i32;
}

impl Foo for i32 {
    const ID: i32 = 1;
}

fn main() {
    assert_eq!(1, i32::ID);
}
```

Foo を実装する場合、必ず ID を定義しなければなりません。もし以下のように定義がなかった場合

```
#![feature(associated_consts)]

trait Foo {
    const ID: i32;
}

impl Foo for i32 {
}
```

このようになります。

```
error: not all trait items implemented, missing: `ID` [E0046]
(訳注: エラー。トレイトの全ての要素が実装されていません。 `ID` が未実装です。)
     impl Foo for i32 {
     }
```

既定値についても以下のように実装できます。

```
#![feature(associated_consts)]

trait Foo {
    const ID: i32 = 1;
}

impl Foo for i32 {
}

impl Foo for i64 {
    const ID: i32 = 5;
}

fn main() {
    assert_eq!(1, i32::ID);
    assert_eq!(5, i64::ID);
}
```

上記の通り、 `Foo` トレイトを実装する際、 `i32` のように未実装のままにすることができます。この場合、既定値が使われます。一方 `i64` のように独自の定義を追加することもできます。

関連定数は必ずしもトレイトに関連付けられる必要はありません。`struct` や `enum` の `impl` ブロックにおいても使うことができます。

```
#![feature(associated_consts)]

struct Foo;

impl Foo {
    const FOO: u32 = 3;
}
```

カスタムアロケータ

メモリ割り当てが常に簡単に出来るとは限りません。ほとんどの場合、Rust が既定の方法でメモリ割り当てを行いますが、割り当て方法をカスタマイズする必要が出てくる場合があります。現在、コンパイラと標準ライブラリはコンパイル時に既定のグローバルアロケータを切り替えることが出来ます。詳細は RFC 1183[*23] に書かれていますが、ここではどのように独自のアロケータを作成するか順を追って説明します。

既定のアロケータ

現在コンパイラは `alloc_system` と `alloc_jemalloc` (jemalloc のないターゲットもあります) という 2 つの既定のアロケータを提供しています。これらのアロケータは単に普通の Rust のクレートで、メモリの割り当てと解放の手続きを実装しています。標準ライブラリはどちらか一方を前提としてコンパイルされているわけではありません。コンパイラは生成する成果物の種類に応じてどちらのアロケータを使用するかをコンパイル時に決定します。

バイナリを生成する場合、既定では (もし可能なら) `alloc_jemalloc` を使用します。この場合、コンパイラは最後のリンクにまで影響力を持っているという意味で、「全世界を支配」しています。従ってアロケータの選択はコンパイラに委ねることができます。

一方、動的あるいは静的ライブラリの場合、既定では `alloc_system` を使用します。他のアプリケーションや他の環境など、使用するアロケータの決定権がない世界において、Rust は「お客」に過ぎません。そのため、メモリの割り当てと解放を行うには、標準 API(例えば `malloc` と `free`) に頼ることになります。

アロケータの切り替え

コンパイラによる既定の選択はほとんどの場合うまく動きますが、しばしば多少の調整が必要になることがあります。コンパイラのアロケータ選択を上書きするには、単に希望のアロケータとリンクするだけです。

[*23] https://github.com/rust-lang/rfcs/blob/master/text/1183-swap-out-jemalloc.md

```
#![feature(alloc_system)]

extern crate alloc_system;

fn main() {
    let a = Box::new(4); // システムアロケータからのメモリ割り当て
    println!("{}", a);
}
```

この例で生成されるバイナリは既定の jemalloc とリンクする代わりに、システムアロケータを使います。逆に既定で jemalloc を使う動的ライブラリを生成するには次のようにします。

```
#![feature(alloc_jemalloc)]
#![crate_type = "dylib"]

extern crate alloc_jemalloc;

pub fn foo() {
    let a = Box::new(4); // jemalloc からのメモリ割り当て
    println!("{}", a);
}
```

カスタムアロケータを書く

時々 jemalloc とシステムアロケータの選択では足りず、全く新しいカスタムアロケータが必要になることがあります。この場合、アロケータ API(例えば `alloc_system` や `alloc_jemalloc` と同様のもの) を実装した独自のクレートを書くことになります。例として、`alloc_system` の簡素な注釈付きバージョンを見てみましょう。

```
// コンパイラがリンク時に他のアロケータ (例えば jemalloc) とリンクしてしまうことを防ぐため、
// このクレートがアロケータであることを示す必要があります。
#![feature(allocator)]
#![allocator]
```

```
// 循環依存を避けるため、アロケータはアロケータを使う標準ライブラリに依存してはいけません。
// しかし libcore については全ての機能を使用できます。
#![no_std]

// カスタムアロケータに固有の名前を付けてください。
#![crate_name = "my_allocator"]
#![crate_type = "rlib"]

// この独自アロケータは FFI バインディングのために Rust コンパイラのソースコードに
// 同梱されている libc クレートを使います。
// 注記: 現在の外部 libc(crate.io のもの) は標準ライブラリとリンクしているため使用できません
// ( `#![no_std]` がまだ stable ではないためです)。そのため特別に同梱版の libc が必要になります。
#![feature(libc)]
extern crate libc;

// 今のところ、カスタムアロケータには以下の 5 つのメモリ割り当て関数が必要です。
// コンパイラは現時点では関数のシグネチャとシンボル名の型検査を行いません。
// しかし、将来的には以下の型に一致する必要があります。
//
// 注記: 標準の `malloc` と `realloc` 関数へはアラインメントについての情報を
// 渡すことができません。そのためアラインメントを考慮するためにはこの実装は
// 改良の必要があります。

#[no_mangle]
pub extern fn __rust_allocate(size: usize, _align: usize) -> *mut u8 {
    unsafe { libc::malloc(size as libc::size_t) as *mut u8 }
}

#[no_mangle]
pub extern fn __rust_deallocate(ptr: *mut u8, _old_size: usize, _align: usize) {
    unsafe { libc::free(ptr as *mut libc::c_void) }
}

#[no_mangle]
```

```
pub extern fn __rust_reallocate(ptr: *mut u8, _old_size: usize, size: usize,
                                _align: usize) -> *mut u8 {
    unsafe {
        libc::realloc(ptr as *mut libc::c_void, size as libc::size_t) as *mut u8
    }
}

#[no_mangle]
pub extern fn __rust_reallocate_inplace(_ptr: *mut u8, old_size: usize,
                                        _size: usize, _align: usize) -> usize {
    old_size // この API は libc ではサポートされていません。
}

#[no_mangle]
pub extern fn __rust_usable_size(size: usize, _align: usize) -> usize {
    size
}
```

このクレートをコンパイルすると、次のように使えるようになります。

```
extern crate my_allocator;

fn main() {
    let a = Box::new(8); // カスタムアロケータによるメモリ割り当て
    println!("{}", a);
}
```

カスタムアロケータの制限

カスタムアロケータを使用する場合、コンパイルエラーの原因となりうるいくつかの制限があります。

- 1 つの成果物は高々 1 つのアロケータとしかリンクすることはできません。バイナリ、dynlib、staticlib は必ず 1 つのアロケータとリンクする必要があり、もし明示的に指定されなければコンパイラがアロケータを選択します。一方、rlib はアロケータとリンクする必要はありません (リン

クすることも可能です)。

- アロケータを使うコードは `#![needs_allocator]` でタグ付けされます (例えば現時点での `liballoc` クレート)。また、 `#[allocator]` がついたクレートはアロケータを使うクレートに直接的にも間接的にも依存することが出来ません (例えば、循環依存は許されていません)。このためアロケータは原則として libcore にしか依存しないようにする必要があります。

7
用語集

全ての Rustacean がシステムプログラミングあるいはコンピュータサイエンスのバックグラウンドを持つ訳でもないので馴染みがないかもしれない用語について説明を付与しました。

■抽象構文木

コンパイラがプログラムをコンパイルする時、様々なことをします。その 1 つがテキストを「抽象構文木」（または「AST」）に変換することです。この木はプログラムの構造を表します。例えば、 `2 + 3` は以下のような木に変換されます。

```
  +
 / \
2   3
```

そして `2 + (3 * 4)` はこのようになるでしょう。

```
  +
 / \
2   *
   / \
  3   4
```

■アリティ

アリティは関数、あるいは操作がとる引数の数を指します。

```
let x = (2, 3);
let y = (4, 6);
let z = (8, 2, 6);
```

上記の例では x と y はアリティ 2 を、 z はアリティ 3 を持ちます。

■境界

境界 (Bounds) とはある 1 つの型またはトレイトにおける制約のことです。例えば、ある関数がとる引数に境界が設定されたとすると、その関数に渡される型は設定された制約に必ず従わなければなりません。

■DST (Dynamically Sized Type)

A type without a statically known size or alignment. (more info[*1])

■式

コンピュータプログラミングに於いて、式は値、定数、変数、演算子、1 つの値へと評価される関数の組み合わせです。例えば、 `2 + (3 * 4)` は値 14 を返す式です。式が副作用を持ちうることに意味はありません。例えば、ある式に含まれる関数がただ値を返す以外にも何か作用をするかもしれません。

■式指向言語

早期のプログラミング言語では式と文は文法上違うものでした。式は値を持ち、文は何かをします。しかしながら後の言語ではこれらの溝は埋まり式で何かを出来るようになり文が値を持てるようになりました。式指向言語では (ほとんど) 全ての文が式であり値を返します。同時にこれらの式文はより大きな式の一部を成します。

■文

コンピュータプログラミングに於いて、文とはコンピュータに何かしらの作用をさせるプログラミング言語の最小要素です。

[*1] ../nomicon/exotic-sizes.html#dynamically-sized-types-dsts

8

構文の索引

キーワード

- as: プリミティブのキャスト。あるいはあるアイテムを含むトレイトの曖昧性の排除。[型間のキャスト (as)] 、 [共通の関数呼び出し構文 (山括弧形式)] 、 関連型参照。
- break: ループからの脱却。[ループ (反復の早期終了)] 参照。
- const: 定数および定数ポインタ。const と static 、 生ポインタ参照。
- continue: 次の反復への継続。 [ループ (反復の早期終了)] 参照。
- crate: 外部クレートのリンク。 [クレートとモジュール (外部クレートのインポート)] 参照
- else: if と if let が形成するフォールバック。 [if] 、 [if let] 参照。
- enum: 列挙型の定義。 列挙型参照。
- extern: 外部クレート、関数、変数のリンク。[クレートとモジュール (外部クレートのインポート)]、[他言語関数インターフェイス] 参照。
- false: ブーリアン型の偽値のリテラル。 [プリミティブ型 (ブーリアン型)] 参照。
- fn: 関数定義及び関数ポインタ型。関数 参照。
- for: イテレータループ、 トレイト impl 構文の一部、あるいは 高階ライフタイム構文。 [ループ (for)] 、メソッド構文 参照。
- if: 条件分岐 [if] 、 [if let] 参照。
- impl: 継承及びトレイト実装のブロック。メソッド構文 参照。
- in: for ループ構文の一部。 [ループ (for)] 参照。

- let: 変数束縛。変数束縛 参照。
- loop: 条件無しの無限ループ。 [ループ (loop)] 参照。
- match: パターンマッチ。 マッチ参照。
- mod: モジュール宣言。 [クレートとモジュール (モジュールを定義する)] 参照。
- move: クロージャ構文の一部。 [クロージャ (move クロージャ)] 参照。
- mut: ポインタ型とパターン束縛におけるミュータビリティを表す。ミュータビリティ 参照。
- pub: struct のフィールド、 impl ブロック、モジュールにおいて可視性を表す。 [クレートとモジュール (パブリックなインターフェースのエクスポート)] 参照。
- ref: 参照束縛。 [パターン (ref と ref mut)] 参照。
- return: 関数からのリターン。 [関数 (早期リターン)] 参照。
- Self: 実装者の型のエイリアス。トレイト 参照。
- self: メソッドの主語。 [メソッド構文 (メソッド呼び出し)] 参照。
- static: グローバル変数。 [const と static (static)] 参照。
- struct: 構造体定義。 構造体参照。
- trait: トレイト定義。トレイト 参照。
- true: ブーリアン型の真値のリテラル。 [プリミティブ型 (ブーリアン型)] 参照。
- type: 型エイリアス、または関連型定義。type エイリアス 、関連型 参照。
- unsafe: アンセーフなコード、関数、トレイト、そして実装を表す。 [Unsafe] 参照。
- use: スコープにシンボルをインポートする。[クレートとモジュール (use でモジュールをインポートする)] 参照。
- where: 型制約節。 [トレイト (where 節)] 参照。
- while: 条件付きループ。 [ループ (while)] 参照。

演算子とシンボル

- ! (ident!(…), ident!{…}, ident![…]): マクロ展開を表す。マクロ 参照
- ! (!expr): ビット毎、あるいは論理の補数。オーバロード可能 (Not)。
- != (var != expr): 非等価性比較。オーバーロード可能 (PartialEq)。
- % (expr % expr): 算術剰余算。オーバーロード可能 (Rem)。
- %= (var %= expr): 算術剰余算をして代入。オーバーロード可能 (RemAssign)。
- & (expr & expr):ビット毎の論理積。オーバーロード可能 (BitAnd)。
- & (&expr): 借用。参照と借用 参照
- & (&type, &mut type, &'a type, &'a mut type): 借用されたポインタの型。参照と借用 参照。

- `&=` (`var &= expr`): ビット毎の論理積をして代入。オーバーロード可能 (`BitAndAssign`)。
- `&&` (`expr && expr`): 論理積。
- `*` (`expr * expr`): 算術乗算。 オーバーロード可能 (`Mul`)。
- `*` (`*expr`): 参照外し。
- `*` (`*const type`, `*mut type`): 生ポインタ。生ポインタ 参照。
- `*=` (`var *= expr`): 算術乗算をして代入。オーバーロード可能 (`MulAssign`)。
- `+` (`expr + expr`): 算術加算。オーバーロード可能 (`Add`)。
- `+` (`trait + trait`, `'a + trait`): 合成型制約。[トレイト (複数のトレイト境界)] 参照。
- `+=` (`var += expr`): 算術加算をして代入。オーバーロード可能 (`AddAssign`)。
- `,`: 引数または要素の区切り。アトリビュート 、関数 、構造体 、ジェネリクス 、マッチ 、クロージャ 、[クレートとモジュール (`use` でモジュールをインポートする)] 参照。
- `-` (`expr - expr`): 算術減算。オーバーロード可能 (`Sub`)。
- `-` (`- expr`): 算術負。オーバーロード可能 (`Neg`)。
- `-=` (`var -= expr`): 算術減算をして代入。オーバーロード可能 (`SubAssign`)。
- `->` (`fn(…) -> type`, `|…| -> type`): 関数とクロージャの返り型。 関数、クロージャ 参照。
- `-> !` (`fn(…) -> !`, `|…| -> !`): 発散する関数またはクロージャ。発散する関数 参照。
- `.` (`expr.ident`): メンバへのアクセス。構造体 、メソッド構文 参照。
- `..` (`..`, `expr..`, `..expr`, `expr..expr`): 右に開な区間のリテラル。
- `..` (`..expr`): 構造体リテラルのアップデート構文。[構造体 (アップデート構文)] 参照。
- `..` (`variant(x, ..)`, `struct_type { x, .. }`): 「〜と残り」のパターン束縛。[パターン (束縛の無視)] 参照。
- `...` (`...expr`, `expr...expr`) **式内で**: 閉区間式。イテレータ 参照
- `...` (`expr...expr`) **パターン内で**: 閉区間パターン。 [パターン (レンジ)] 参照
- `/` (`expr / expr`): 算術除算。オーバーロード可能 (`Div`)。
- `/=` (`var /= expr`): 算術除算と代入。オーバーロード可能 (`DivAssign`)。
- `:` (`pat: type`, `ident: type`): 制約。変数束縛 、関数 、構造体 、トレイト 参照。
- `:` (`ident: expr`): 構造体のフィールドの初期化。構造体 参照。
- `:` (`'a: loop {…}`): ループラベル。 [ループ (ループラベル)] 参照。
- `;`: 文またはアイテムの区切り。
- `;` (`[…; len]`): 固定長配列構文の一部。[プリミティブ型 (配列)] 参照。
- `«` (`expr « expr`): 左シフト。オーバーロード可能 (`Shl`)。
- `«=` (`var «= expr`): 左シフトして代入。オーバーロード可能 (`ShlAssign`)。
- `<` (`expr < expr`): 「より小さい」の比較。オーバーロード可能 (`PartialOrd`)。

- <= (var <= expr): 「以下」の比較。オーバーロード可能 (PartialOrd)。
- = (var = expr, ident = type): 代入/等価比較。 変数束縛 、type エイリアス、ジェネリックパラメータのデフォルトを参照。
- == (var == expr): 等価性比較。オーバーロード可能 (PartialEq)。
- => (pat => expr): マッチの腕の構文の一部。 マッチ 参照。
- > (expr > expr): 「より大きい」の比較。オーバーロード可能 (PartialOrd)。
- >= (var >= expr): 「以上」の比較。オーバーロード可能 (PartialOrd)。
- » (expr » expr): 右シフト。オーバーロード可能 (Shr)。
- »= (var »= expr): 右シフトして代入。オーバーロード可能 (ShrAssign)。
- @ (ident @ pat): パターン束縛。 [パターン (束縛)] 参照。
- ^ (expr ^ expr): ビット毎の排他的論理和。オーバーロード可能 (BitXor)。
- ^= (var ^= expr): ビット毎の排他的論理和をして代入。オーバーロード可能 (BitXorAssign)。
- | (expr | expr): ビット毎の論理和。 オーバーロード可能 (BitOr)。
- | (pat | pat): パターンの「または」。 [パターン (複式パターン)] 参照
- | (|…| expr): クロージャ。クロージャ 参照。
- |= (var |= expr): ビット毎の論理和をして代入。オーバーロード可能 (BitOrAssign)。
- || (expr || expr): 論理和。
- _: 「無視」するパターン束縛。 [パターン (束縛の無視)]。

他の構文

- 'ident: 名前付きライフタイムまたはループラベル。ライフタイム 、 [ループ (ループラベル)] 参照。
- …u8, …i32, …f64, …usize, …: その型の数値リテラル。
- "…": 文字列リテラル。文字列 参照。
- r"…", r#"…"#, r##"…"##, …: 生文字列リテラル、エスケープ文字は処理されない。 [リファレンス (生文字列リテラル)] 参照。
- b"…": バイト列リテラル、文字列ではなく [u8] を作る。 [リファレンス (バイト列リテラル)] 参照。
- br"…", br#"…"#, br##"…"##, …: 生バイト列リテラル。生文字列とバイト列リテラルの組み合わせ。 [リファレンス (生バイト列リテラル)] 参照
- '…': 文字リテラル。 [プリミティブ型 (char)] 参照。
- b'…': ASCII バイトリテラル。

- `|…| expr`: クロージャ。クロージャ 参照。

- `ident::ident`: パス。[クレートとモジュール (モジュールを定義する)] 参照。
- `::path`: クレートのルートからの相対パス (つまり明示的な絶対パス)。 [クレートとモジュール (`pub use` による再エクスポート)] 参照。
- `self::path`: 現在のモジュールからの相対パス (つまり明示的な相対パス)。 [クレートとモジュール (`pub use` による再エクスポート)] 参照。
- `super::path`: 現在のモジュールの親からの相対パス。[クレートとモジュール (`pub use` による再エクスポート)] 参照。
- `type::ident`, `<type as trait>::ident`: 関連定数、関数、型。 関連型参照。
- `<type>::…`: 直接名前付けられない型の関連アイテム (例えば`<&T>::…` 、`<[T]>::…` 、 など)。関連型 参照。
- `trait::method(…)`: メソッドを定義したトレイトを指定することによるメソッド呼び出しの曖昧性排除。[共通の関数呼び出し構文] 参照。
- `type::method(…)`: そのメソッドが定義された型を指定することによるメソッド呼び出しの曖昧性排除。[共通の関数呼び出し構文] 参照。
- `<type as trait>::method(…)`: メソッドを定義したトレイト 及び型を指定することによるメソッド呼び出しの曖昧性排除。[共通の関数呼び出し構文 (山括弧形式)] 参照。

- `path<…>` (例えば `Vec<u8>`): 型でのジェネリック型のパラメータの指定。ジェネリクス 。
- `path::<…>`, `method::<…>` (例えば`"42".parse::<i32>()`): 式でのジェネリック型あるいは関数、メソッドの型の指定。
- `fn ident<…> …`: ジェネリック関数を定義。ジェネリクス 参照。
- `struct ident<…> …`: ジェネリック構造体を定義。ジェネリクス 参照。
- `enum ident<…> …`: ジェネリック列挙型を定義。ジェネリクス 参照。
- `impl<…> …`: ジェネリック実装を定義。
- `for<…> type`: 高階ライフタイム境界。
- `type<ident=type>` (例えば `Iterator<Item=T>`): 1 つ以上の関連型について指定のあるジェネリック型。関連型 参照。

- `T: U`: `U` を実装する型に制約されたジェネリックパラメータ `T` 。トレイト 参照。
- `T: 'a`: ジェネリック型 `T` はライフタイム `'a` より長生きしなければならない。ライフタイムが「長生きする」とは`'a` より短かい、いかなるライフタイムも推移的に含んでいないことを意味する。
- `T : 'static`: ジェネリック型 T は `'static` なもの以外の借用した参照を含んでいない。

- `'b: 'a`: ジェネリックライフタイム `'b` はライフタイム`'a` より長生きしなければならない。
- `T: ?Sized`: ジェネリック型パラメータが動的サイズ型になること許可する。[サイズ不定型 (`?Sized`)] 参照。
- `'a + trait`, `trait + trait`: 合成型制約。 [トレイト (複数のトレイト境界)] 参照。

- `#[meta]`: 外側のアトリビュート。アトリビュート 参照。
- `#![meta]`: 内側のアトリビュート。アトリビュート 参照。
- `$ident`: マクロでの置換。マクロ 参照。
- `$ident:kind`: マクロでの捕捉。マクロ 参照。
- `$(…)…`: マクロでの繰り返し。マクロ 参照。

- `//`: ラインコメント。コメント 参照。
- `//!`: 内側の行ドキュメントコメント。コメント 参照。
- `///`: 外側の行ドキュメントコメントコメント 参照。
- `/*…*/`: ブロックコメント。コメント 参照。
- `/*!…*/`: 内側のブロックドキュメントコメント。コメント 参照。
- `/**…*/`: 外側のブロックドキュメントコメント。コメント 参照。

- `()`: 空タプル (**あるいは** ユニット) の、リテラルと型両方。
- `(expr)`: 括弧付きの式。
- `(expr,)`: 1 要素タプルの式。 [プリミティブ型 (タプル)] 参照。
- `(type,)`: 1 要素タプルの型。 [プリミティブ型 (タプル)] 参照。
- `(expr, …)`: タプル式。 [プリミティブ型 (タプル)] 参照。
- `(type, …)`: タプル型。 [プリミティブ型 (タプル)] 参照。
- `expr(expr, …)`: 関数呼び出し式。また、 タプル `struct` 、 タプル `enum` のヴァリアントを初期化するのにも使われる。関数 参照。
- `ident!(…)`, `ident!{…}`, `ident![…]`: マクロの起動。マクロ 参照。
- `expr.0`, `expr.1`, `…`: タプルのインデックス。[プリミティブ型 (タプルのインデックス)] 参照。

- `{…}`: ブロック式。
- `Type {…}`: `struct` リテラル。構造体 参照。

- `[…]`: 配列リテラル。 [プリミティブ型 (配列)] 参照。
- `[expr; len]`: `len` 個の `expr` を要素に持つ配列リテラル。 [プリミティブ型 (配列)] 参照。

- `[type; len]`: `len` 個の `type` のインスタンスを要素に持つ配列型。[プリミティブ型 (配列)] 参照。
- `expr[expr]`: コレクションのインデックス。オーバーロード可能 (`Index`, `IndexMut`)。
- `expr[..]`, `expr[a..]`, `expr[..b]`, `expr[a..b]`: コレクションのスライスのようなコレクションのインデックス。`Range` 、 `RangeFrom` 、 `RangeTo` 、`RangeFull` を「インデックス」として使う。

9 関係書目

これは Rust に関連した読書一覧です。これらの中には（ある時点での）Rust の設計に影響を与えたものもあれば、Rust についての出版物もあります。

■型システム

- Region based memory management in Cyclone[*1]
- Safe manual memory management in Cyclone[*2]
- Typeclasses: making ad-hoc polymorphism less ad hoc[*3]
- Macros that work together[*4]
- Traits: composable units of behavior[*5]
- Alias burying[*6] — 似たようなことをしようとしましたがやめました
- External uniqueness is unique enough[*7]
- Uniqueness and Reference Immutability for Safe Parallelism[*8]

[*1] http://209.68.42.137/ucsd-pages/Courses/cse227.w03/handouts/cyclone-regions.pdf
[*2] http://www.cs.umd.edu/projects/PL/cyclone/scp.pdf
[*3] http://www.ps.uni-sb.de/courses/typen-ws99/class.ps.gz
[*4] https://www.cs.utah.edu/plt/publications/jfp12-draft-fcdf.pdf
[*5] http://scg.unibe.ch/archive/papers/Scha03aTraits.pdf
[*6] http://www.cs.uwm.edu/faculty/boyland/papers/unique-preprint.ps
[*7] http://www.cs.uu.nl/research/techreps/UU-CS-2002-048.html
[*8] https://research.microsoft.com/pubs/170528/msr-tr-2012-79.pdf

- Region Based Memory Management[*9]

■並行性

- Singularity: rethinking the software stack[*10]
- Language support for fast and reliable message passing in singularity OS[*11]
- Scheduling multithreaded computations by work stealing[*12]
- Thread scheduling for multiprogramming multiprocessors[*13]
- The data locality of work stealing[*14]
- Dynamic circular work stealing deque[*15] — Chase と Lev の deque
- Work-first and help-first scheduling policies for async-finish task parallelism[*16] — 完全正格 (fully-strict) なワークスティーリング (work stealing) より一般的
- A Java fork/join calamity[*17] — Java の fork/join ライブラリについての評価。特にワークスティーリングの非正格な計算への応用
- Scheduling techniques for concurrent systems[*18]
- Contention aware scheduling[*19]
- Balanced work stealing for time-sharing multicores[*20]
- Three layer cake for shared-memory programming[*21]
- Non-blocking steal-half work queues[*22]
- Reagents: expressing and composing fine-grained concurrency[*23]
- Algorithms for scalable synchronization of shared-memory multiprocessors[*24]

[*9] http://www.cs.ucla.edu/~palsberg/tba/papers/tofte-talpin-iandc97.pdf
[*10] https://research.microsoft.com/pubs/69431/osr2007_rethinkingsoftwarestack.pdf
[*11] https://research.microsoft.com/pubs/67482/singsharp.pdf
[*12] http://supertech.csail.mit.edu/papers/steal.pdf
[*13] http://www.eecis.udel.edu/%7Ecavazos/cisc879-spring2008/papers/arora98thread.pdf
[*14] http://www.aladdin.cs.cmu.edu/papers/pdfs/y2000/locality_spaa00.pdf
[*15] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.170.1097&rep=rep1&type=pdf
[*16] http://www.cs.rice.edu/%7Eyguo/pubs/PID824943.pdf
[*17] http://www.coopsoft.com/ar/CalamityArticle.html
[*18] http://www.stanford.edu/~ouster/cgi-bin/papers/coscheduling.pdf
[*19] http://www.blagodurov.net/files/a8-blagodurov.pdf
[*20] http://www.cse.ohio-state.edu/hpcs/WWW/HTML/publications/papers/TR-12-1.pdf
[*21] http://dl.acm.org/citation.cfm?id=1953616&dl=ACM&coll=DL&CFID=524387192&CFTOKEN=44362705
[*22] http://www.cs.bgu.ac.il/%7Ehendlerd/papers/p280-hendler.pdf
[*23] http://www.mpi-sws.org/~turon/reagents.pdf
[*24] https://www.cs.rochester.edu/u/scott/papers/1991_TOCS_synch.pdf

- Epoch-based reclamation[*25]

■その他

- Crash-only software[*26]
- Composing High-Performance Memory Allocators[*27]
- Reconsidering Custom Memory Allocation[*28]

■Rust についての論文

- GPU Programming in Rust: Implementing High Level Abstractions in a Systems Level Language[*29] — Eric Holk による初期の GPU 研究
- Parallel closures: a new twist on an old idea[*30] — 正確には Rust についてではないが、nmatsakis によるもの（訳注: nmatsakis は mozilla のデベロッパ）
- Patina: A Formalization of the Rust Programming Language[*31] — Eric Reed による初期の型システムのサブセットの形式化
- Experience Report: Developing the Servo Web Browser Engine using Rust[*32] — Lars Bergstrom によるもの
- Implementing a Generic Radix Trie in Rust[*33] — Michael Sproul の学部論文
- Reenix: Implementing a Unix-Like Operating System in Rust[*34] — Alex Light の学部論文
- Evaluation of performance and productivity metrics of potential programming languages in the HPC environment[*35] — Florian Wilkens の卒業論文。C、Go、Rust を比較する
- Nom, a byte oriented, streaming, zero copy, parser combinators library in Rust[*36] — Geoffroy Couprie による VLC のための研究

[*25] https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf
[*26] https://www.usenix.org/legacy/events/hotos03/tech/full_papers/candea/candea.pdf
[*27] http://people.cs.umass.edu/~emery/pubs/berger-pldi2001.pdf
[*28] http://people.cs.umass.edu/~emery/pubs/berger-oopsla2002.pdf
[*29] http://www.cs.indiana.edu/~eholk/papers/hips2013.pdf
[*30] https://www.usenix.org/conference/hotpar12/parallel-closures-new-twist-old-idea
[*31] ftp://ftp.cs.washington.edu/tr/2015/03/UW-CSE-15-03-02.pdf
[*32] http://arxiv.org/abs/1505.07383
[*33] https://michaelsproul.github.io/rust_radix_paper/rust-radix-sproul.pdf
[*34] http://scialex.github.io/reenix.pdf
[*35] http://octarineparrot.com/assets/mrfloya-thesis-ba.pdf
[*36] http://spw15.langsec.org/papers/couprie-nom.pdf

- Graph-Based Higher-Order Intermediate Representation[*37] — Rust に似た言語、Impala で実装された実験的中間表現
- Code Refinement of Stencil Codes[*38] — Impala を使った別の論文
- Parallelization in Rust with fork-join and friends[*39] — Linus Farnstrand の修士論文
- Session Types for Rust[*40] — Philip Munksgaard の修士論文。Servo のための研究
- Ownership is Theft: Experiences Building an Embedded OS in Rust - Amit Levy, et. al.[*41]
- You can't spell trust without Rust[*42] — Alexis Beingessner の修士論文

[*37] http://compilers.cs.uni-saarland.de/papers/lkh15_cgo.pdf
[*38] http://compilers.cs.uni-saarland.de/papers/ppl14_web.pdf
[*39] http://publications.lib.chalmers.se/records/fulltext/219016/219016.pdf
[*40] http://munksgaard.me/papers/laumann-munksgaard-larsen.pdf
[*41] http://amitlevy.com/papers/tock-plos2015.pdf
[*42] https://raw.githubusercontent.com/Gankro/thesis/master/thesis.pdf