# 前回 (2022-07-11) からの差分

## Rustup コマンドの更新がありました（Rust 本体の更新ではない）

Announcing Rustup 1.25.0 | Rust Blog
Announcing Rustup 1.25.1 | Rust Blog

- Windows でのインストールがより便利になった
- rustup の起動パフォーマンスがよくなった

更新しない理由はないです。

いま

```
$ rustup --version
rustup 1.24.3 (ce5817a94 2021-05-31)
```

更新する

```
$ rustup self update
```

（もし homebrew 経由で rustup をインストールしていた場合、これだと怒られるので、brew upgrade でアップデートする）

更新後

```
$ rustup --version
rustup 1.25.1 (bb60b1e89 2022-07-12)
```

# 今日のゴール

Cargo を覚えて、Cargo 経由で Rust プログラムを作っていけるようになる。

次回から本格的に Rust を頑張る予定。

# Cargo とは

Introduction - The Cargo Book
一番しっかりしてるページ（英語）

Hello, Cargo! - The Rust Programming Language 日本語版
さらっとしたページ（日本語）

Rust プログラムを作るための便利コマンドです。
外部のライブラリを使うときの動きは、Ruby の bundler と同じです。

Cargo.toml ファイルに、このプログラムに関する設定をなんでもかんでも書きます。

- $ cargo new xxx
  - プログラムのテンプレートを作ってくれる（スケルトン）
  - （使う機会は、作り始める最初だけ）
- $ cargo add xxx
  - 外部の xxx ライブラリを使えるようにする
  - （Rust 1.62 からの待望の新機能）
- $ cargo test
  - ユニットテストを実行する
- $ cargo clippy
  - 「Rust 的にはこう書いた方がいい」という部分を教えてくれる
  - （別に無視してもよい）
- $ cargo build
  - 最終成果物のバイナリを生成する
  - （内部で rustc している）
- $ cargo run
  - main 関数を実行する
  - その実態は、cargo build して、できたバイナリを実行しているだけのショートカット

ちなみに、Rust では ライブラリのことを「クレート」と呼ぶ。

## クレートのサイト

Crates.io

## cargo new hello_world する

好きなディレクトリで。

```
$ cargo new hello_world
      Created binary (application) `hello_world` package
```

（ちなみに、ライブラリを作るときは以下のように「--lib」をつける）

```
$ cargo new --lib tsuru
      Created library `tsuru` package
```

中にもうテンプレートが作られていて、実行もできる（スケルトン）。

```
.
├── .git
├── .gitignore
├── Cargo.toml
└── src
    └── main.rs
```

（Git が邪魔なら、「--vcs=none」もつけるとよい。私はよくつけている）

src/main.rs

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

```
$ cargo run
   Compiling hello_world v0.1.0
(/Users/r9/dev/src/sandbox/sandbox/rust-practice/hello_world)
    Finished dev [unoptimized + debuginfo] target(s) in 3.35s
    Running `target/debug/hello_world`
Hello, world!
```

コンパイル云々の文字がうっとおしければ「-q」をつける（quiet）。

```
$ cargo run -q
Hello, world!
```

## image クレートを使ってみる

[image - Rust]

```
$ cargo add image
```

（初回は結構時間がかかる）

Cargo.toml に自動的に追記されている。あと Cargo.lock ファイルが自動的に生成されている。

```
[dependencies]
image = "0.24.2"
```

処理を変更してみる。

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

    let input = "/Users/r9/Desktop/input.jpg";
    let output = "/Users/r9/Desktop/outnput.jpg";

    let mut img = image::open(input).unwrap();
    img.invert();
    img.save(output).unwrap();
}
```

input.jpg の画像（なんでもよい）を用意して、実行してみる。

※ image クレートはめちゃくちゃ遅いから本番用には必ず release モードでビルドしろと書いてあるくらい遅いです

```
$ cargo run --release
```

初回は結構時間がかかる。
（–release オプションは最適化してビルドするので、つけないときよりもサイズが小さくて実行速度が早くなるがコンパイルが遅い）

output.jpg の画像が出力されているのを確認する。

## example を作って実行する

本筋のプログラムは src ディレクトリ内に作っていくが、それとは別枠で「こうやって使えるよ」というサンプルプログラムを作ることができる。
もちろん、本筋の方の関数を呼び出して使える。

examples ディレクトリを作る。

examples/case1.rs

```
fn main() {
    println!("case1");
}
```

```
$ cargo run -q --example case1
case1
```

## src/bin に別の main() を作って実行する

普通に作る場合、生成できるバイナリは1つだけになるが、src/bin にファイルを置くことで複数のターゲットを対象にできる。
こちらも本筋の関数を使える。

src/bin/case2.rs

```
fn main() {
    println!("case2");
}
```

```
$ cargo run -q --bin case2
case2
```

（これを作ると、Cargo に動かす対象を教える必要があるようになる。Cargo.toml にデフォルトを教えることもできる）

```
$ cargo run
error: `cargo run` could not determine which binary to run. Use the
`--bin` option to specify a binary, or the `default-run` manifest key.
available binaries: case2, hello_world
```

```
[package]
default-run = "hello_world"
```

```
$ cargo run -q
Hello, world!
```

## benches ディレクトリとか tests ディレクトリとかもある

- benches
  - ベンチマーク用のコードを置くためのディレクトリ
- tests
  - ユニットテスト用のコードを置くためのディレクトリ
  - （Rust はプログラムの中に一緒にユニットテストを書くこともできるが、別居することもできる。このディレクトリはつまり別居先）

## Cargo.toml と Cargo.lock

Ruby でいう Gemfile と Gemfile.lock みたいなやつ。
Cargo.toml の方はもうちょっと色々細かいことも書ける。

Cargo.lock は、一回でもビルドとか実行すると自動的に生成されて、基本的に、自分で直接編集しない。

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

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

[dependencies]
image = "0.24.2"
```

[dependencies] のところが、明示された外部のクレート一覧となる（多分一番よく見ることになる）。

edition は、コンパイラにどのバージョンでコンパイルしてほしいかを書く。
いまのところ「2015」「2018」「2021」の3つがあって、新しいバージョンなら新しい Rust の文法や機能が使える。
昔のプログラムを触るならともかく、最初から作るのなら最新の 2021 を選ばない理由はない。

# まとめ

- $ cargo new でスケルトンを作る
- src ディレクトリ以下にコードを書く
- $ cargo add でクレートを追加
- $ cargo run で実行
- $ cargo build –release で最適化されたバイナリを生成（完成）