Draft. Please do not include this page.

Single chapter compiled via LaTeX.

Shing Lyu, June 18, 2022. Git commit: 65314af

Notes for the production team

Whitespace marker

You will see pink underline markers like these: `a␣b␣c`. One marker indicates one whitespace. So there is exactly one whitespace between "a" and "b" and one whitespace between "b" and "c". You'll also see them in code blocks like this:

```
#[derive(StructOpt)]
struct␣Options␣{
␣␣␣␣//␣...
␣␣␣␣#[structopt(short␣=␣"f",␣long␣=␣"file",␣parse(from_os_str))]
␣␣␣␣///␣Load␣the␣cat␣picture␣from␣the␣specified␣file
␣␣␣␣catfile:␣Option<std::path::PathBuf>,
}
```

These markers are only for you to see how many whitespaces there are. Please keep the exact number of whitespaces for inline code and code blocks. **Please do NOT include the markers during layout**.

The layout output should look like this: `a b c`

```
#[derive(StructOpt)]
struct Options {
    // ...
    #[structopt(short = "f", long = "file", parse(from_os_str))]
    /// Load the cat picture from the specified file
    catfile: Option<std::path::PathBuf>,
}
```

Auto line wrapping marker

You'll also see arrows near the end of the line and beginning of the second line:

```
This is a very very very very very very very very very very ↵
    ↳very very very long line that was wrapped automatically.
```

This indicates the original line was too long and it was automatically wrapped. **Please include these arrows.**

CHAPTER 10

■ ■ ■

What else can you do with Rust?

10.1 The end is just the beginning

We've walked through an exciting journey through the world of Rust. We've learned how to build a CLI, a GUI, a game, physical devices, and machine learning models. What next steps can you take? What other exciting applications can you build with Rust? We'll briefly walk you through some other areas that we didn't cover in this book.

10.2 The web

The World Wide Web (or *the web*) is probably one of the most influential areas in programming that shaped the modern world. People have high hopes about Rust in this area. There are so many interesting topics about the web; they can fill a book of their own. In today's industry, web programmers are categorized into two categories: backend and frontend. Backend Engineers are responsible for the server-side, while the frontend developer handles the client-side, usually the browser. But people often forgot that the browser and web crawlers, which consumes the backend on behalf of the frontend, is also a software.

Backend

Rust has a great potential to be a backend language because it provides high performance, no garbage collection, and security. The Rust backend ecosystem might not be as mature as other languages like PHP, Ruby, Python, Go, or Node.js, but there are already some frameworks worth checking out. The "Are we web yet?" page[1] curates a list of various web-related crates. For a full web framework, you might want to check out Rocket[2], Gotham[3], or Actix-web[4].

One of the most common operations in a backend is to access databases. There are

[1] https://www.arewewebyet.org/
[2] https://crates.io/crates/rocket
[3] https://crates.io/crates/gotham
[4] https://crates.io/crates/actix-web

database drivers like `mysql`[5] and `postgres`[6]. If you don't like to write raw SQL queries, you can use an ORM (Object Relational Mapping) that maps your Rust object to database operations. Popular ORMs includes `Diesel`[7] and `rustorm`[8].

Most of these frameworks let you build a web server on top of a single physical machine. But managing and operating physical servers is an expensive, sometimes even painful job. Unless you have the budget and staff to operate a data center, most people will go with a cloud provider. The level of service can range from a simple virtual private server, a hosted Docker container cluster, or even fully-managed serverless environments. Serverless computing like AWS (Amazon Web Service) Lambda allows you to run code without worrying about server provisioning. AWS Lambda gives you a way to run your Rust code in a pay-as-you-go model. You only need to write the Rust function that handles web request or does computational tasks, AWS takes care of the maintenance of physical servers, virtual machines, all the way to the Rust runtime environment. AWS started to experiment with Rust on Lambda[9]. You can also run your Rust code in an AWS EC2 (Elastic Compute Cloud) server or ECS (Elastic Container Service) container. AWS also provides many hosted services like DynamoDB (NoSQL database), SQS (message queue), and S3 (file storage). You can access all of these services through the Rust SDK `rusoto`[10].

■ **NOTE** You might see the term "Async I/O[11]" a lot when browsing the backend crates and their documentations. We'll briefly explain why Async I/O is vital for the backend.

A web server usually has to serve a large number of requests simultaneously. These requests are network I/O-bound tasks that might take a long time. To prevent a request blocking all the others to be served, we need to introduce concurrency. The Rust's native way of doing concurrency is to create OS threads. Whenever a thread is waiting for a slow I/O operation, it can return the control to the OS kernel. The kernel can then let other threads do their work, and swap back to the waiting thread when its I/O operation is done. However, OS threads have a lot of overheads, and context-switching between them is computationally expensive. When we have thousands of requests to serve, spawning thousands of threads is simply too expensive and might overwhelm the OS. To solve this problem, we can try to run multiple tasks on one single OS thread. Whenever a task makes a slow I/O request, instead of synchronously waiting for it, it makes an asynchronous call and gives the control to other tasks on the same thread. Some kind of mechanism has to keep monitoring the I/O operation and continue to run the task after the I/O is ready.

The concept is easy, but it requires two new things: a new syntax and a runtime. The

[5] https://crates.io/crates/mysql
[6] https://crates.io/crates/postgres
[7] https://crates.io/crates/diesel
[8] https://crates.io/crates/rustorm
[9] https://aws.amazon.com/blogs/opensource/rust-runtime-for-aws-lambda/
[10] https://crates.io/crates/rusoto
[11] I/O stands for input/output

new syntax is `Futures`[12] and `async-await`. `Futures` provide a way to describe a pending task, and `async-await` gives us a way to say where the waiting happens. This syntax has been discussed for years and are finally available in stable Rust in the version 1.36.0 (`Future`) and 1.39.0 (`async-await`). There is a "Are we async yet?" page[13] that tracks this progress. The `futures-rs`[14] crate also provides extra traits and tooling around the bare-minimum `std::future::Future`.

To coordinate the execution of `Futures`, we also need a runtime. Since the design philosophy of Rust is to keep the core runtime as minimal as possible, an async runtime is not included[15]. The two most popular umbrella project that provides a runtime are `tokio`[16] and `async-std`[17]. Since both projects started before the `Future` and `async-await` syntax are stabilized, there were many experimental implementations based on legacy syntax and crates. But with the new syntax in stable Rust now, they are all moving towards a more unified future.

Frontend

In today's frontend world, JavaScript has an absolute monopoly. Although JavaScript is easy to learn (but hard to master), it's performance is not very satisfying. Unlike Rust, JavaScript uses garbage collection, which means there will be pauses in the execution. JavaScript engines also use methods like Just-in-time compilation (JIT) to speed up the execution, but it leads to unpredictable performances. Asm.js was created to address this problem. Asm.js is a subset of JavaScript that is easier to optimize. You can write in languages like C or C++ and compile it to asm.js. Because asm.js deliberately chose a small subset of JavaScript, it can run much faster and without garbage collection.

The learnings from asm.js eventually lead to the development of WebAssembly, sometimes abbreviated as WASM. WebAssembly is a standard for a binary code format that languages like C, C++, and Rust can compile to. The compiled program can then be run in a virtual machine inside the web browser[18], alongside JavaScript. It can provide high-performance applications in the web browser. WebAssembly is not intended to replace JavaScript, but to augment it. For example, we can rewrite the performance bottleneck in Rust and make it work together with the JavaScript around it.

The Rust official website has a dedicated page for WebAssembly: https://www.rust-lang.org/what/wasm. There is also a WASM working group[19] that is actively developing

[12] In JavaScript, this concept is called `Promise`.

[13] https://areweasyncyet.rs/

[14] https://crates.io/crates/futures

[15] Before Rust 1.0, there was support for green-thread, which is another pattern for handling the "run multiple tasks on one thread" problem, but it was removed before 1.0.

[16] https://tokio.rs/

[17] https://async.rs/

[18] WebAssembly can theoretically run in other host environments. However, most of the focus is on running in web browsers at the moment.

[19] https://rustwasm.github.io/

the WebAssembly ecosystem in Rust. They maintain an excellent tool called `wasm-pack` that is a one-stop-shop for building, testing, and publishing your Rust to WebAssembly projects. When a Rust project is compiled to WebAssmebly and packaged by `wasm-pack`, it can be published to npm, the JavaScript package registry.

Under the hood, `wasm-pack` uses `wasm-bindgen`, the WASM/JavaScript binding generator. This crate can generate binding codes between Rust (compiled to WASM) and JavaScript so that you can pass data and function between the two languages without much hassle. They also provide two creates, `js-sys` and `web-sys`, to help you integrate your Rust code with JavaScript APIs. `Js-sys` contains Rust bindings for core JavaScript APIs, for example `Objects`, `Functions`, `Arrays`. But keep in mind that `js-sys` only contains JavaScript API that exists in all environments, including the browser and Node.js. The web APIs like the DOM (Document Object Model) and other things you'll find in your browser's global `window` object is exposed in the `web-sys` crate. With these tools and bindings, you can easily build Rust programs that can be compiled to WebAssembly and work seamlessly with JavaScript in any modern browser.

Web browser and crawler

When people discuss the web in terms of frontend and backend, they often omit what sits in between, the web browser. The reason people often omit it is because there is only a handful of browsers available on the market, so they are considered something set in stone. You might protest that there are hundreds of browsers you can find on Wikipedia[20], but in fact, most of the modern[21] browsers are powered by three browser engines:

- Blink: Chromium, Google Chrome, Microsoft Edge, Opera
- Gecko: Firefox
- WebKit: Safari

There is also a browser engine written in Rust from the ground up, called Servo[22]. Servo is one of the most significant projects written in Rust. The Servo project started in 2012, now has roughly 2.6 million lines of code (not all Rust, but still impressive). Servo started as a research project. But in 2017, the CSS engine it contains has matured and merged into Gecko, the browser engine that powers Mozilla's Firefox. The rendering component of Servo, called WebRender, was integrated into Firefox later as well. So if you are using Firefox right now, you are also executing a big chunk of Rust code.

Servo has a significant impact on Rust itself. The two projects shared some core developers, and the core contributors worked closely with each other because they were all started as research projects under Mozilla Research. Many of Servo's need drives

[20] Wikipeida page "List of web browsers": https://en.wikipedia.org/wiki/List_of_web_browsers

[21] Internet Explorer is powered by Trident, but Microsoft has stopped developing new versions of Internet Explorer and is encouraging users to switch to Microsoft Edge. See https://www.microsoft.com/en-us/microsoft-365/windows/end-of-ie-support.

[22] https://github.com/servo/servo

the development of new features in Rust, and Rust's design also heavily influenced how Servo is architectured. If you are interested in seeing Rust in large-scale projects, Servo is definitely a fun piece of work to dive into.

Browsers are for human beings. However, many programs also consume web pages. These programs are usually referred to as *Web Crawlers*, *Scrapers* or *Spiders*. They "crawl" through web pages and extract information from them. A use case might be when you want to compare prices listed on different e-commerce websites, but these websites don't provide APIs. We can utilize a crawler to crawl through their web pages and extract the price information from the HTML. There are a few frameworks for building web crawlers, for example `maman`[23], `spider`[24], and `url-crawler`[25]. If you want more fine-grain control over the crawling and parsing process, you can use the `reqwest`[26] HTTP client library to download the HTML page, and use an HTML parsing/querying library to parse the page and extract data. Some popular HTML parsing/querying library includes `html5ever`[27], `scraper`[28], and `select`[29].

Mobile

In Chapter 3, we talked about how to build a GUI for desktop. But we didn't talk about how to build GUIs for mobile devices (i.e., apps). The most dominant mobile platforms are Google's Android and Apple's iOS. Android apps are written with Java or Kotlin, while iOS apps are written in Objective-C or Swift. Sadly, Rust can't be a drop-in replacement for these natively supported languages. But both Android and iOS have some mechanism for invoking (and be invoked by) C or C++ libraries. These mechanisms are crucial for performance-critical applications, which builds the user interface in Java/Kotlin/Objective-C/Swift, and offload the computation-intensive part to C/C++ libraries. Since we can compile Rust to a library that looks and feels like a C library, we can also use this mechanism to build an app that has business logic written in Rust.

For Android, the process will be[30]

- Install the Android Studio (containing the Android SDK), this is the official development environment for Android apps.
- Install the Android NDK (Native Development Kit). This toolkit allows us to compile Rust into a library that can work on Android and interact with Java/Kotlin. There is a `cargo ndk`[31] command you can install to simplify the compilation

[23] https://crates.io/crates/maman
[24] https://crates.io/crates/spider
[25] https://crates.io/crates/url-crawler
[26] https://crates.io/crates/reqwest
[27] https://crates.io/crates/html5ever
[28] https://crates.io/crates/scraper
[29] https://crates.io/crates/select
[30] Mozilla published a post that guides you through the process step by step: https://mozilla.github.io/firefox-browser-architecture/experiments/2017-09-21-rust-on-android.html
[31] https://github.com/bbqsrc/cargo-ndk

process.

- Use `rustup` to install the android targets, for example `armv7-linux-androideabi`
- Build your Rust library project and compile it to a library file. We need to expose our Rust code to Java through JNI (Java Native Interface), there is the `jni`[32] crate that helps you with that.
- Import the Rust library into your Java/Kotlin Android app project, and call the library inside your Java/Kotlin code.

The steps for iOS[33] is very similar:

- Install Xcode, which is the official development environment for iOS apps.
- Use `rustup` to install the iOS targets, for example `armv7-apple-ios`
- Build your Rust library project and compile it to a library file. We also need to expose a C-style header file, so iOS can consume the Rust library as if it's a C library.
- Import the Rust library into your XCode and call the library inside your Objective-C/Swift code.

If you want to avoid building the same code twice, you might want to consider using Flutter. Flutter is a cross-platform UI toolkit developed by Google, and it will be the UI toolkit for Fuchsia, Google's upcoming OS. Currently, Flutter uses the Dart programming language for building the UI. Then it can run on both Android and iOS. Rust libraries can be packaged as Flutter plugins and be invoked from the Dart code, so we can build a Flutter frontend + Rust library architecture similar to the Android and iOS ones we've shown before. There is a `flutter-rs`[34] project that will help you integrate Rust with Flutter.

If you are looking for purely-Rust mobile applications, you can still build it the hard way on Android. You can use the `android-rs-glue`[35] project to package your Rust code into an APK file. You still need the Android SDK and NDK, but the `android-rs-glue` project provides a docker image, which contains all the necessary setup and wiring to cross-compile your Rust code into an Android-compatible library. Then it creates a thin wrapper APK that invokes the main function in your Rust library immediately. `Android-rs-glue` also provides an `android_glue`[36] crate, which gives the Rust code access

[32] https://crates.io/crates/jni

[33] Here is the Mozilla post on running Rust on iOS: https://mozilla.github.io/firefox-browser-architecture/experiments/2017-09-06-rust-on-ios.html.

[34] https://github.com/flutter-rs/flutter-rs

[35] https://github.com/rust-windowing/android-rs-glue

[36] https://crates.io/crates/android_glue

to JNI. This can be used to render to the screen and accept user inputs. But since this doesn't give us access to any native UI widgets, we have to render everything inside the Rust code via OpenGL or Vulkan, similar to most of the full-screen games app you may find.

■ **NOTE** The idea of compiling Rust into a shared library, and use it inside other programming languages using their FFI (foreign function interface) mechanism, can be applied not only in the mobile realm. The Rust FFI Omnibus website[37] collects such examples for various programming languages:

- C
- Ruby
- Python
- Haskell
- Node.js
- C#
- Julia

It can also work the other way around. Rust can call libraries written in other languages like C. You can reference the section *Using extern functions to call external code*[38] from the Rust book to learn more.

Operating systems and embedded devices

As we mentioned at the end of Chapter 8, there are many more things you can do on the hardware level than just blinking an LED. There are simply too many hardware platforms and peripherals out there, writing bare-metal Rust programs for each and every one of them from scratch is almost impossible. Thankfully there are some software abstractions already defined at various layers. At the most bottom layer, there are peripheral access crates that contain register definitions and low-level details of the microcontrollers. On top of that, there is the `embedded-hal` layer. The `-hal` suffix stands for Hardware Abstraction Layer. The `embedded-hal` is a few traits that define a hardware-agnostic interface between a specific HAL implementation and drivers. Drivers can be written against the `embedded-hal` traits without worrying about hardware-specific details. This enables developers to build portable drivers, firmware, and applications on top of this abstraction layer. You can find many `embedded-hal` crates and their implementations by search the keyword "embedded-hal" or "embedded-hal-impl" on crates.io.

[37] http://jakegoulding.com/rust-ffi-omnibus/basics/

[38] https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html#using-extern-functions-to-call-external-code

Building on top of `embedded-hal` are driver crates and board support crates. Drivers give you platform-agnostic support to a specific kind of device like sensors, modems, LCD controllers, etc. Board support crates give you support for a specific development board.

Many Rust developers also take on the challenge of building operating systems in Rust. One of the relatively mature ones is Redox OS[39] (Figure 10.1), which is designed with the microkernel architecture. It already has a GUI and some useful applications running on it. There is also Tock[40], which is targeting IoT (Internet-of-things) devices with low-memory and low-power constraints.

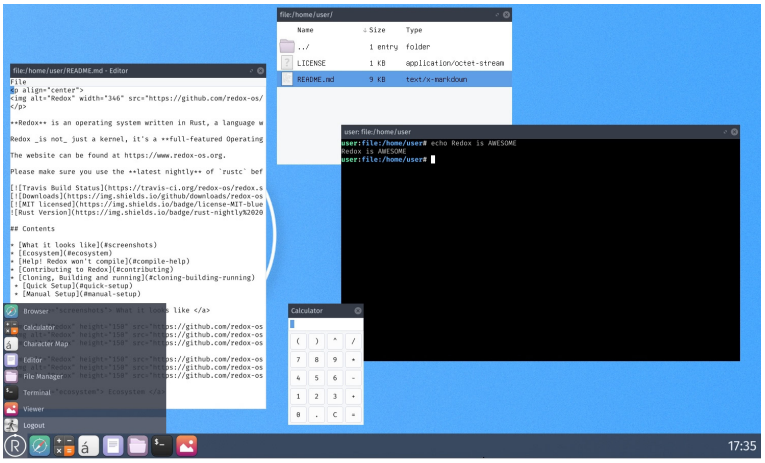

Figure 10.1: Redox OS. Image retrieved from the Redox OS GitLab repository. MIT License.

There are OSes for teaching purposes, like the intermezzOS[41] and Blog OS[42]. There are also a few less-active ones that take a language-based approach, which means their focus is to build a minimal OS that can run a Rust program. This includes rustboot, later forked into RustOS, then further forked into QuiltOS. But most of them are not actively developed since around 2017.

[39] https://www.redox-os.org/

[40] https://www.tockos.org

[41] https://intermezzos.github.io/

[42] https://os.phil-opp.com/ It's named Blog OS because it was a series of blog articles by Philip Opperman on how to build an OS in Rust.

Unlimited possibilities of Rust

Besides the files we talked about in previous chapters and sections, there are many more applications of Rust. Here is a non-exhaustive list[43]:

- Blockchain and Cryptocurrency: Facebook's Libra cryptocurrency is built with Rust.
- Compression
- Cryptography and Security: `ring`[44], `openssl`[45], `sodiumoxide`[46].
- Database implementations[47]
- Emulators: game consoles and other hardware.
- Multimedia: images, audio, and video manipulation; rendering 2D/3D content.
- Parser
- Science: mathematics, bio-informatics (e.g., Rust-Bio[48]), geo-information, physics, and chemistry simulation

Rust is a wonderful tool for building almost any kind of application. Although at this moment, some fields might not have a mature, production-ready Rust library and user base, but with the support from the passionate and friendly community, we can expect to see many more applications of Rust from the mini IoT sensors running on low-power microcontrollers, to cutting-edge AI running on massive supercomputers. Have you found anything you would like to build with Rust? Let's all work together to grow Rust and unleash its full potential!

[43] This list in alphabetical order. The order does not indicate popularity or maturity.
[44] https://crates.io/crates/ring
[45] https://crates.io/crates/openssl
[46] https://crates.io/crates/sodiumoxide
[47] https://crates.io/categories/database-implementations
[48] https://rust-bio.github.io/