欢迎来到 Comprehensive Rust 

这是由 Android 团队开发的免费 Rust 课程。该课程涵盖了 Rust 的全部范围，从基本语法到高级主题如泛型和错误处理。

> 如需查看课程的最新版本，请访问 https://google.github.io/comprehensive-rust/。如果你是在其他地方阅读，请查看此网址了解是否有更新。

本课程的目标是教授你 Rust。我们假设你对 Rust 一无所知，并希望能够：

- 帮助你全面理解 Rust 的语法和语言。
- 使你能够修改现有的程序并用 Rust 编写新程序。
- 展示常见的 Rust 习语。

我们将前三天的课程称为“Rust 基础知识”。

在此基础上，你可以选择深入学习一个或多个专门的主题：

- Android：为期半天的课程，介绍如何在 Android 平台开发中使用 Rust（AOSP）。课程内容包括与 C、C++ 和 Java 的互操作性。
- Bare-metal：为期一天的课程，介绍如何使用 Rust 进行裸机（嵌入式）开发。课程内容涵盖微控制器和应用处理器。
- 并发：为期一天的课程，介绍 Rust 中的并发性。我们将涵盖传统并发（使用线程和互斥锁进行抢占式调度）和 async/await 并发（使用 futures 进行协作式多任务处理）。

非目标

Rust 是一门庞大的语言，我们无法在几天内涵盖所有内容。本课程的一些非目标包括：

- 了解如何开发宏，请参阅 Rust Book 中的第 19.5 章 和 Rust by Examples。

前提假设

本课程假设你已经具备编程知识。Rust 是一种静态类型语言，我们有时会与 C 和 C++ 进行比较，以更好地解释或对比 Rust 的方法。

如果你已经了解如 Python 或 JavaScript 等动态类型语言的编程，那么你也能够很好地跟上本课程。

▼ *Speaker Notes*

这是演讲者备注的示例。我们将使用这些备注来为幻灯片添加额外的信息。这可能包括讲师应该涵盖的关键点，以及课堂上常见问题的答案。

授课

> 本页面适用于课程教师。

以下是有关 Google 内部授课方式的一些背景信息。

上课时间通常是从上午 10:00 到下午 4:00，中间有 1 小时的午餐休息时间。这样，上午和下午各留了 2.5 小时的上课时间。请注意，这仅是建议：您也可以上午上课 3 小时，让学员有更多的时间进行练习。上课时间较长的缺点是，学员上了整整 6 小时的课，到了下午可能会非常疲倦。

在授课之前，你需要完成以下事项：

1. 熟悉课程资料。我们添加了演讲者备注，借此强调要点（请帮个忙，多多贡献演讲者备注！）。演示幻灯片时，你应确保在弹出式窗口中打开演讲者备注（点击对应的链接，在“演讲者备注”旁边有一个小箭头）。这样，你就可以确保屏幕整洁有序，更好地向全班学员展示课程内容。

2. 确定培训日期。由于本课程至少需要三天的时间，因此我们建议你安排两周以上的时间。课程学员曾表示，在每堂课之间留一段间隔会很有帮助，因为这有利于他们吸收我们所提供的所有信息。

3. 找一间足以容纳全体线下学员的大教室。我们建议你将课程人数控制在 15-25 人之间。这样，人数足够少，不仅便于学员提问问题，配备的一位教师也有时间答疑解惑。确保教室备有供你和学生使用的“课桌”：你们都需要能够坐下来并操作各自的笔记本电脑。特别是身为教师，你现场要进行大量编码，所以讲台对你来说用处不大。

4. 在开课当天，请提前一点到教室，设置好教学设备。我们建议你直接在笔记本电脑上运行 `mdbook serve` 来演示课程内容（请参阅安装说明）。这样可以确保你在切换页面时没有延迟，演示效果更好。当你或课程学员发现拼写错误时，你也可以使用笔记本电脑及时更正。

5. 让学员采取小组形式或独立解题。通常，我们会在上午和下午各安排 30-45 分钟的练习时间（包括查看解决方案的时间）。请务必询问学员是否遇到困难，或是否需要任何帮助。如果你看到多位学员遇到同样的问题，请在班级集体进行讲解，并提供相应的解决方案，例如告诉大家在标准库的什么位置可以找到相关信息。

今天的分享就是这些，祝你授课顺利！希望你和我们一样，乐在其中！

欢迎你课后提供反馈，帮助我们不断改进课程。我们非常期待了解哪些方面做得不错，哪些方面还需要改进。同时非常欢迎学生们向我们发送反馈！

课程结构

> 本页面适用于课程教师。

Rust 基础知识

我们会在头三天介绍 Rust 基础知识。这几天的步调会稍快,因为我们要探讨许多层面:

- 第 1 天：Rust 基础知识、语法、控制流、创建及使用值。
- 第 2 天：内存管理、所有权、复合数据类类型及标准库。
- 第 3 天：泛型、trait、错误处理、测试和不安全 Rust。

深入探究

除了为期 3 天的“Rust 基础知识”课程外，我们还推出了一些专题课程：

Android 中的 Rust

深入探究 Android课程为期半天，旨在介绍如何使用 Rust 进行 Android 平台开发。其中包括与 C、C++ 和 Java 的互操作性。

你将需要签出 AOSP。在同一机器上签出课程库， 然后将 `src/android/` 目录移至所签出的 AOSP 的根目录。这将确保 Android 构建系统能检测到 `src/android/` 中的 `Android.bp` 文件。

确保 `adb sync` 适用于你的模拟器或实际设备， 并使用 `src/android/build_all.sh` 预构建所有 Android 示例。请阅读脚本， 查看它所运行的命令，并确保这些命令能在你手动运行时正确执行。

裸金属 Rust

深入探究裸金属 Rust课程为期一天，旨在介绍如何使用 Rust 进行裸金属（嵌入式）开发。其中涵盖了微控制器和应用处理器。

对于微控制器部分，你需要提前购买 BBC micro:bit 第 2 版开发板。每个人都需要安装多个软件包， 具体如欢迎页面中所述。

Rust 中的并发

深入探究并发课程为期一天，旨在介绍传统并发和 `async` / `await` 并发。

你需要设置一个新 crate，下载所需的依赖项， 做好课前准备。然后，你可以将示例复制/粘贴到 `src/main.rs` 中， 以便对以下代码进行实验：

```
cargo init concurrency
cd concurrency
cargo add tokio --features full
cargo run
```

课程形式

本课程的互动性非常强， 建议你以问题驱动探索 Rust！

键盘快捷键

mdBook 中有一些实用键盘快捷键：

- `向左箭头`：转到上一页。
- `向右箭头`：转到下一页。
- `Ctrl + Enter`：执行具有焦点的代码示例。
- `s`：激活搜索栏。

翻译

一批优秀的志愿者已将本课程翻译成其他语言：

- 巴西葡萄牙语版本 译者：@rastringer、@hugojacob、@joaovicmendes 和 @henrif75。
- 韩语版本 译者：@keispace、@jiyongp 和 @jooyunghan。
- 西班牙语版本 译者：@deavid。

使用右上角的语言选择器切换语言。

未完成的翻译

多数语言版本仍在翻译中。我们会提供最近更新的翻译的链接：

- 孟加拉语版本 译者：@raselmandol。
- 繁体中文版本 译者：@hueich、@victorhsieh、@mingyc 和 @johnathan79717。
- 简体中文版本 译者：@suetfei、@wnghl, @anlunx、@kongy, @noahdragon 和 @superwhd。
- 法语版本 译者：@KookaS 和 @vcaen。
- 德语版本 译者：@Throvn 和 @ronaldfw。
- 日语版本 译者：@CoinEZ-JPN 和 @momotaro1105。

如果你想帮助我们，请参阅我们的说明，了解如何开始翻译。翻译工作将通过问题跟踪器跟踪。

使用 Cargo

开始了解 Rust 后，你很快就会遇到 Cargo，这是 Rust 生态系统中用于构建和运行 Rust 应用的标准工具。在这里，我们想简要介绍一下什么是 Cargo、它如何融入更广泛的生态系统， 以及我们如何在本培训中合理利用 Cargo。

安装

> **请按照 https://rustup.rs/ 上的说明操作。**

这将为你提供 Cargo 构建工具 (`cargo`)和 Rust 编译器 (`rustc`)。你还将获得 `rustup` ，这是一个命令行实用程序,你可以用它来安装不同的编译器版本。

安装Rust之后，你应当配置你的编辑器或 IDE 以开始使用 Rust。大多数编辑器使用rust-analyzer以达成此目的。它为VS Code、Emacs、Vim/Neovim及其他许多编辑器提供了自动补全及定义跳转的功能。同样也可以用一个叫RustRover的IDE。

▼ *Speaker Notes*

- 在 Debian/Ubuntu 上，你也可以通过 `apt` 安装 Cargo、Rust 源代码和 Rust 格式化工具。但是，这样会得到一个过时的 Rust 版本，这可能会导致意外的行为。命令如下:

  ```
  sudo apt install cargo rust-src rustfmt
  ```

Rust 生态系统

Rust 生态系统由许多工具组成，其中的主要工具包括：

- `rustc`： Rust 编译器，可将 `.rs` 文件转换为二进制文件和其他中间格式。
- `cargo`： Rust 依赖项管理器和构建工具。Cargo 知道如何下载托管在 https://crates.io 上的依赖项,并在构建项目时将它们传递给 `rustc` 。Cargo 还附带一个内置的测试运行程序，用于执行单元测试。
- `rustup`： Rust 工具链安装程序和更新程序。发布新版本 Rust 时，此工具用于安装并更新 `rustc` 和 `cargo` 。 此外， `rustup` 还可以下载标准库的文档。你可以同时安装多个版本的 Rust，并且 `rustup` 可让你根据需要在这些版本之间切换。

▼ *Speaker Notes*

关键点：

- Rust 有一个快速发布时间表，每六周就会发布一次 新版本。新版本保持与旧版本的向后兼容性，还添加了新功能。
- 共有三个发布阶段：“稳定版”、“Beta 版”和“夜间版”。
- 我们会在“夜间版”上测试新功能，每六周将“Beta 版”升级为“稳定版”。
- 您也可以通过备用的注册数据库、git及文件夹等资源来解析依赖项。
- Rust 也有三个[版本]：当前版本是 Rust 2021。之前的版本是 Rust 2015 和 Rust 2018。
    - 这些版本支持对语言进行向后不兼容的更改。
    - 为防止破坏代码，你可以自行选择版本： 通过 `Cargo.toml` 文件为 crate 选择合适的版本。
    - 为免分割生态系统，Rust 编译器可以混合使用为不同版本编写的代码。
    - 提及不通过 `cargo` 而直接使用编译器的情况相当少见（大多数用户从不这样做）。
    - 值得注意的是，Cargo 本身就是一个功能强大且全面的工具。它能够实现许多高级功能，包括但不限于：
        - 项目/软件包结构
        - [工作区]
        - 开发依赖项和运行时依赖项管理/缓存
        - [构建脚本]
        - [全局安装]]
        - 它还可以使用子命令插件（例如 cargo clippy）进行扩展。

- 如需了解详情，请参阅[Cargo 官方图书]

本培训中的代码示例

在本培训中，我们将主要通过示例探索 Rust 语言，这些示例可通过浏览器执行。这能大大简化设置过程， 并确保所有人都能获得一致的体验。

我们仍然建议你安装 Cargo：它有助于你更轻松地完成练习。在最后一天，我们要做一个更大的练习， 向你展示如何使用依赖项，因此你需要安装 Cargo。

本课程中的代码块是完全交互式的：

```
fn main() {
    println!("Edit me!");
}
```

当文本框为焦点时，你可以使用`Ctrl + Enter` to execute the code when focus is in the text box.

▼ *Speaker Notes*

大多数代码示例都可修改（如上图所示）。少数代码示例可能会因各种原因而不可修改：

- 嵌入式 Playground 无法执行单元测试。将代码复制并粘贴到实际 Playground 中，以演示单元测试。
- 嵌入式 Playground 会在你离开页面后立即丢失其状态！正因如此，学员应使用本地安装的 Rust 或通过 Playground 解题。

使用 Cargo 在本地运行代码

如果你想在自己的系统上对代码进行实验， 则需要先安装 Rust。为此，请按照 Rust 图书中的说明操作。这应会为你提供一个有效的 `rustc` 和 `cargo` 。在撰写本文时，最新的 Rust 稳定版具有以下版本号：

```
% rustc --version
rustc 1.69.0 (84c898d65 2023-04-16)
% cargo --version
cargo 1.69.0 (6e9a83356 2023-04-12)
```

您也可以使用任何更高版本，因为 Rust 保持向后兼容性。

了解这些信息后，请按照以下步骤从本培训中的 一个示例中构建 Rust 二进制文件：

1. 在你要复制的示例上点击“复制到剪贴板”按钮。
2. 使用 `cargo new exercise` 为你的代码新建一个 `exercise/` 目录：

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

3. 导航至 `exercise/` 并使用 `cargo run` 构建并运行你的二进制文件：

   ```
   $ cd exercise
   $ cargo run
      Compiling exercise v0.1.0 (/home/mgeisler/tmp/exercise)
       Finished dev [unoptimized + debuginfo] target(s) in 0.75s
        Running `target/debug/exercise`
   Hello, world!
   ```

4. 将 `src/main.rs` 中的样板代码替换为你自己的代码。例如， 使用上一页中的示例，将 `src/main.rs` 改为：

   ```
   fn main() {
       println!("Edit me!");
   }
   ```

5. 使用 `cargo run` 构建并运行你更新后的二进制文件：

```
$ cargo run
   Compiling exercise v0.1.0 (/home/mgeisler/tmp/exercise)
    Finished dev [unoptimized + debuginfo] target(s) in 0.24s
     Running `target/debug/exercise`
Edit me!
```

6. 使用 `cargo check` 快速检查项目是否存在错误；使用 `cargo build` 只进行编译，而不运行。你可以在 `target/debug/` 中找到常规调试 build 的输出。使用 `cargo build --release` 在 `target/release/` 中生成经过优化的 发布 build。

7. 你可以通过修改 `Cargo.toml` 为项目添加依赖项。当你 运行 `cargo` 命令时，系统会自动为你下载和编译缺失 的依赖项。

▼ *Speaker Notes*

尽量鼓励全班学员安装 Cargo 并使用 本地编辑器。这能为他们营造常规 开发环境，让工作变得更加轻松。

欢迎来到第一天

今天是学习 Comprehensive Rust 的第一天。我们会涉及很多内容：

- Rust 基本语法：变量、标量（scalar）和复合（compound）类型、枚举（Enum）、结构体（struct）、引用、函数和方法。
- 控制流的构造：`if`、`if let`、`while`、`while let`、`break` 和 `continue`。
- 模式匹配：解构枚举、结构体和数组（array）。

▼ *Speaker Notes*

请提醒学生：

- 他们可以随时提问，不需要留到最后。
- 这个课程本应该是互动的，我们鼓励大家积极讨论。
  - As an instructor, you should try to keep the discussions relevant, i.e., keep the discussions related to how Rust does things vs some other language. It can be hard to find the right balance, but err on the side of allowing discussions since they engage people much more than one-way communication.
- 有些问题会导致我们提前谈到后面的内容。
  - 这完全没有问题！重复是学习的一个重要方法。请记住，这些幻灯片只是一种辅助，你可以选择性地跳过。

第一天的主要目标是要谈到著名的 borrow checker，其他方面点到为止。Rust 处理内存的方式是其主要特点，这点我们应该尽早展示给学生。

如果你是在教室里教授此课程，不妨在这里介绍一下时间安排。 这边建议是把每天分成两部分（跟着幻灯片来）：

- 早上：9:00 到 12:00，
- 下午：13:00 到 16:00。

当然你也可以看情况调整时间。但是请务必记得提供休息时间。我们建议每个小时休息一次！

什么是 Rust?

Rust 是一门新的编程语言，它的1.0 版本于 2015 年发布：

- Rust 是一门静态编译语言，其功能定位与 C++ 相似
  - `rustc` 使用 LLVM 作为它的后端。
- Rust 支持多种平台和架构:
  - x86、ARM、WebAssembly……
  - Linux、Mac、Windows……
- Rust 被广泛用于各种设备中：
  - 固件和引导程序，
  - 智能显示器，
  - 手机，
  - 桌面，
  - 服务器。

▼ *Speaker Notes*

Rust 和 C++ 适用于类似的场景：

- 极高的灵活性。
- 高度的控制能力。
- 能够在资源匮乏的设备（如手机）上运行。
- 没有运行时和垃圾收集。
- 关注程序可靠性和安全性，而不会牺牲任何性能。

Hello World!

让我们进入最简单的 Rust 程序，一个经典的 Hello World 程序：

```
1  fn main() {
2      println!("Hello 🌍!");
3  }
```

你看到的：

- 函数以 `fn` 开头。
- 像 C 和 C++ 一样，块由花括号分隔。
- `main` 函数是程序的入口点。
- Rust 有卫生宏 (hygienic macros)， `println!` 就是一个例子。
- Rust 字符串是 UTF-8 编码的，可以包含任何 Unicode 字符。

▼ *Speaker Notes*

This slide tries to make the students comfortable with Rust code. They will see a ton of it over the next three days so we start small with something familiar.

关键点：

- Rust is very much like other languages in the C/C++/Java tradition. It is imperative and it doesn't try to reinvent things unless absolutely necessary.
- Rust 是一门现代编程语言，它完全支持 Unicode 等特性。
- 在需要处理可变数量的参数的情况下，Rust 使用宏（没有函数重载）。
- 宏是"卫生的"，这意味着它们不会意外地捕获它们所在作用域中的标识符。实际上，Rust 的宏只是部分卫生。
- Rust 是多范式编程语言。例如，它具有强大的面向对象的编程功能，虽然它不是函数式语言，但包括一系列的函数概念。

简短示例

以下是一个简短的 Rust 示例程序：

```rust
fn main() {                 // Program entry point
    let mut x: i32 = 6;     // Mutable variable binding
    print!("{x}");          // Macro for printing, like printf
    while x != 1 {          // No parenthesis around expression
        if x % 2 == 0 {     // Math like in other languages
            x = x / 2;
        } else {
            x = 3 * x + 1;
        }
        print!(" -> {x}");
    }
    println!();
}
```

▼ *Speaker Notes*

这段代码实现了 Collatz 猜想：猜想认为该循环总是会结束，但该猜想还没有被证明。可以编辑代码来尝试不同的输入。

关键点：

- 说明所有变量的类型都是静态的。尝试删除 `i32` 来触发类型推断。尝试使用 `i8` 来触发运行时整数溢出。
- 将 `let mut x` 改为 `let x`，讨论出现的编译错误。
- 展示 `print!` 在参数与格式字符串不匹配时产生的编译错误。
- 展示如何使用 `{}` 作为占位符，来输出比单个变量更复杂的表达式。
- 向学生展示标准库，展示如何搜索 `std::fmt`，其中包含用于格式化字符串的微型语言规则。要点是让学生熟悉在标准库中搜索的过程。
    - 在 shell 中，运行 `rustup doc std::fmt` 会在浏览器中打开本地 std::fmt 文档

为什么选择 Rust?

Rust 有一些独特的卖点：

- 编译期内存安全。
- 没有运行时未定义行为。
- 现代的编程语言特性。

▼ *Speaker Notes*

应该问问学生们都使用过哪些语言。根据答案侧重讲解 Rust 的不同特性：

- 使用过 C 或 C++：Rust 利用借用检查消除了一类 *运行时错误* 。你可以达到堪比 C 和 C++ 的性能，而没有内存不安全的问题。并且你还可以得到些现代的语言构造，比如模式匹配和内置依赖管理。

- 使用过 Java、Go、Python、JavaScript……：你可以得到和这些语言相同的内存安全特性，并拥有类似的使用高级语言的感受。同时你可以得到类似 C 和 C++ 的高速且可预测的执行性能（无垃圾回收机制），以及在需要时对底层硬件的访问。

C语言示例

让我们查看以下 C 语言的“最小错误示例”程序：

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/stat.h>
4
5  int main(int argc, char* argv[]) {
6      char *buf, *filename;
7      FILE *fp;
8      size_t bytes, len;
9      struct stat st;
10
11     switch (argc) {
12         case 1:
13             printf("Too few arguments!\n");
14             return 1;
15
16         case 2:
17             filename = argv[argc];
18             stat(filename, &st);
19             len = st.st_size;
20
21             buf = (char*)malloc(len);
22             if (!buf)
23                 printf("malloc failed!\n", len);
24                 return 1;
25
26             fp = fopen(filename, "rb");
27             bytes = fread(buf, 1, len, fp);
28             if (bytes = st.st_size)
29                 printf("%s", buf);
30             else
31                 printf("fread failed!\n");
32
33         case 3:
34             printf("Too many arguments!\n");
35             return 1;
36     }
37
38     return 0;
39 }
40
```

你发现了多少 bug?

▼ *Speaker Notes*

尽管该 C 语言示例仅有29行代码，但它却包含了至少11个严重 bug：

1. 使用赋值 `=` 而非判断相等 `==` （第28行）
2. `printf` 有多余参数（第23行）

3. 文件描述符泄露（第26行之后）
4. 多行 `if` 语句缺少花括号（第22行）
5. `switch` 语句忘记添加 `break` （第32行）
6. `buf` 字符串忘记NUL终止符，从而导致缓冲区溢出（第29行）
7. 未释放由 `malloc` 分配的缓冲区，从而导致内存泄漏（第21行）
8. 越界访问（第17行）
9. `switch` 语句存在未检查的情况（第11行）
10. `stat` 和 `fopen` 存在未检查的返回值（第18行及第26行）

即使对于 C 语言编译器，这些bug难道不应该是显而易见的吗？
惊人的是，即便使用最新版本的GCC（截至撰文时为13.2），在默认警告等级下编译代码时也不出现任何警告。

这是非常极端的示例吗？
当然不是。这些类型的bug在过去曾引发一系列的安全漏洞，比如以下案例：

- 使用赋值 `=` 而非判断相等 `==`：2003年Linux后门尝试
- 多行 `if` 语句缺少花括号：Apple goto失败漏洞
- `switch` 语句忘记添加 `break`：破坏sudo的break

Rust在这些方面表现得怎么样？
安全Rust使这些bug的出现变得不可能：

1. 不支持 `if` 语句内赋值。
2. 编译时检查格式化字符串。
3. 在作用域末尾，Rust通过 `Drop` trait 来释放资源。
4. 所有 `if` 语句必须有花括号。
5. `match` 语句（在 Rust 中相当于 `switch` ）并不会落空，因此你不会意外忘记一个 `break` 。
6. 缓冲区切片自带它们的大小，且不依赖NUL终止符。
7. 当相关 `Box` 离开作用域时，Rust 通过 `Drop` trait 释放堆分配内存。
8. 越界访问会导致程序发生 panic 而终止，也可以用 `get` 方法来检查一个序列是否越界。
9. `match` 语句规定要处理所有情况。
10. 可出错的 Rust 函数返回的 `Result` 值需要拆箱并检查是否成功。此外，如果你忽略检查标注为 `#[must_use]` 的函数的返回值，编译器会发出警告。

编译期保障

编译期静态内存管理：

- 不存在未初始化的变量。
- 不存在内存泄漏（*通常情况下*，见注释）。
- 不存在“双重释放”。
- 不存在“释放后使用”。
- 不存在 `NULL` 指针。
- 不存在被遗忘的互斥锁。
- 不存在线程之间的数据竞争。
- 不存在迭代器失效。

▼ *Speaker Notes*

在（安全的）Rust 中也有可能产生内存泄漏。例如：

- 可以使用 `Box::leak` 来泄漏一个指针。该方法可以用于得到在运行时决定大小和初始化的静态变量
- 可以使用 `std::mem::forget` 来让编译器“忘记”一个值（即其析构函数不会被执行）。
- 可以使用 `Rc` 或 `Arc` 意外创建一个循环引用（reference cycle）。
- 实际上，有人认为无限填充一个集合也是一种内存泄漏，而 Rust 对此没有保护。

就本课程而言，“不存在内存泄漏”应理解为“几乎没有 *意外* 内存泄漏”。

运行时保障

Rust 没有运行时未定义行为：

- 数组访问有边界检查。
- 整数溢出有明确定义（panic 或回绕）。

▼ *Speaker Notes*

关键点：

- 整数溢出的行为由编译时的标志指定。可以选择 panic（一种受控的程序崩溃）或使用“回绕（wrap-around）”语义。默认情况下，使用调试模式编译（ `cargo build` ）的行为为 panic，使用发布模式编译（ `cargo build --release` ）的行为为“回绕”。
- 边界检查不能使用编译标志禁用，也不能直接通过 `unsafe` 关键字禁用。然而， `unsafe` 允许你调用 `slice::get_unchecked` 等不做边界检查的函数。

现代特性

Rust 建立于过去几十年来所获得的经验之上。

语言特性

- 枚举和模式匹配。
- 泛型。
- 无额外开销的外部函数接口（FFI）。
- 零成本抽象。

工具

- 强大的编译器错误提示。
- 内置依赖管理器。
- 对测试的内置支持。
- 优秀的语言服务协议（Language Server Protocol）支持。

▼ *Speaker Notes*

关键点：

- 与 C++ 类似的零成本抽象，意味着你不需要为高级程序语言的结构“付出”更多的内存和 CPU。例如使用 `for` 循环与使用 `.iter().fold()` 结构应该会生成大致相同的底层指令。
- 值得一提的是，Rust 的枚举是“代数数据类型”（也叫“和类型”）。它使得类型系统可以表示 `Option<T>` 和 `Result<T, E>` 等结构。
- 提醒学生去阅读编译错误 — 许多开发者已经习惯去忽略冗长的编译器输出。Rust 编译器会比其它编译器更健谈。它通常会提供 *可操作的* 反馈，可以直接复制粘贴到代码中。
- 相比 Java、Python 和 Go 等语言，Rust 标准库较为精简。Rust 并没有内置一些你可能认为标准和必要的功能：
  - 随机数生成器，可以使用 rand 替代。
  - SSL 和 TLS 支持，可以使用 rusttls 替代。
  - JSON 支持，可以使用 serde_json 替代。 Rust 这么做的原因是标准库中的功能是无法去除的，因此该功能必须非常稳定。对于以上例子，Rust 社区仍在寻找最佳解决方案 — 甚至对一些情况可能没有单一的“最佳解决方案”。 Rust 内置了一个包管理器 Cargo，使得下载和编译第三方 crate 变得简单。这也导致标准库可以更加精简。

发现高质量的第三方 crate 也许是一个问题。 https://lib.rs/ 等网站对此问题有所帮助。它能帮你比较 crate 的健康指标，以找到一个高质量并受信任的 crate。

- rust-analyzer 是一个受到广泛支持的 LSP 实现，被主流的 IDE 和文本编辑器所使用。

基本语法

Rust 的许多语法与 C、C++ 和 Java 的语法相似：

- 代码块和作用域都是由花括号来界定的。
- 行内注释以 `//` 起始，块注释使用 `/* ... */` 来界定。
- `if` 和 `while` 等关键词作用与以上语言一致。
- 变量赋值使用 `=`，值之间比较使用 `==`。

标量类型

	类型	字面量
有符号整数	`i8` 、 `i16` 、 `i32` 、 `i64` 、 `i128` 、 `isize`	`-10` 、 `0` 、 `1_000` 、 `123_i64`
无符号整数	`u8` 、 `u16` 、 `u32` 、 `u64` 、 `u128` 、 `usize`	`0` 、 `123` 、 `10_u16`
浮点数	`f32` 、 `f64`	`3.14` 、 `-10.0e20` 、 `2_f32`
字符串	`&str`	`"foo"` 、 “两\n行”
Unicode 标量类型	`char`	`'a'` 、 `'α'` 、 `'∞'`
布尔值	`bool`	`true` 、 `false`

各类型占用的空间为：

- `iN` , `uN` 和 `fN` 占用 *N* 位，
- `isize` 和 `usize` 占用一个指针大小的空间，
- `char` 占用 32 位空间，
- `bool` 占用 8 位空间。

▼ *Speaker Notes*

上表中还有一些未提及的语法：

- 原始字符串可在创建 `&str` 时禁用转义： `r"\n" == "\\n"` 。可以在外层引号两侧添加相同数量的 `#` ，以在字符串中嵌入双引号：

```
fn main() {
    println!(r#"<a href="link.html">link</a>"#);
    println!("<a href=\"link.html\">link</a>");
}
```

- 字节串可以用于直接创建 `&[u8]` 类型的值：

```
fn main() {
    println!("{:?}", b"abc");
    println!("{:?}", &[97, 98, 99]);
}
```

- 数字中的所有下划线均可忽略，它们只是为了方便辨识。因此， `1_000` 可以写为 `1000` （或 `10_00` ），而 `123_i64` 可以写为 `123i64` 。

复合类型

	类型	字面量
数组（Arrays）	`[T; N]`	`[20, 30, 40]`, `[0; 3]`
元组（Tuples）	`()`, `(T,)`, `(T1, T2)`, ...	`()`, `('x',)`, `('x', 1.2)`, ...

数组的赋值和访问操作：

```
1  fn main() {
2      let mut a: [i8; 10] = [42; 10];
3      a[5] = 0;
4      println!("a: {:?}", a);
5  }
```

元组的赋值和访问操作：

```
1  fn main() {
2      let t: (i8, bool) = (7, true);
3      println!("t.0: {}", t.0);
4      println!("t.1: {}", t.1);
5  }
```

▼ *Speaker Notes*

关键点：

数组：

- A value of the array type `[T; N]` holds `N` (a compile-time constant) elements of the same type `T`. Note that the length of the array is *part of its type*, which means that `[u8; 3]` and `[u8; 4]` are considered two different types.
- 我们可以使用字面量来为数组赋值。
- 在主函数中，打印（print）语句使用 `?` 格式请求调试实现。 使用参数 `{}` 打印默认输出，`{:?}` 表示以调试格式输出。 我们也可以不在格式化字符串后面指定变量值，直接使用 `{a}` 和 `{a:?}` 进行输出。
- 添加 `#`，比如 `{a:#?}`，会输出“美观打印（pretty printing）”格式，这种格式可能会更加易读。

元组：

- 和数组一样，元组也具有固定的长度。
- 元组将不同类型的值组成一个复合类型。
- 元组中的字段可以通过英文句号加上值的下标进行访问比如： `t.0`，`t.1` 。

- 空元组 `()` 也被称作 “单元（unit）类型”. 它既是一个类型， 也是这种类型的唯一值——也就是说它的类型和它的值都被表示为 `()`。它通常用于表示，比如，一个函数或表达式没有返回值，我们会在后续的幻灯片种见到这种用法。
    - 你可以将其理解为你可能在其他编程语言中比较熟悉的 `void` 类型。

引用

如同 C++ 一样，Rust 也提供了引用类型。

```
fn main() {
    let mut x: i32 = 10;
    let ref_x: &mut i32 = &mut x;
    *ref_x = 20;
    println!("x: {x}");
}
```

一些注意事项：

- 就像 C 与 C++ 中的指针一样，对引用 `ref_x` 进行赋值时，我们必须对其解引用。
- Rust 有时会进行自动解引用。比如调用方法 `ref_x.count_ones()` 时，ref_x 会被解引用。
- 如果引用值被声明为 `mut`（可变引用），那么这个引用值可以在它的生命周期内被绑定为不同的值。

▼ *Speaker Notes*

关键点：

- 注意 `let mut ref_x: &i32` 与 `let ref_x: &mut i32` 之间的区别。第一条语句声明了一个可变引用，所以我们可以修改这个引用所绑定的值；第二条语句声明了一个指向可变变量的引用。

悬垂引用

Rust 会静态禁止悬垂引用：

```
fn main() {
    let ref_x: &i32;
    {
        let x: i32 = 10;
        ref_x = &x;
    }
    println!("ref_x: {ref_x}");
}
```

- 一个引用被认为是“借用（borrow）”了它指向的值。
- Rust 会跟踪所有引用的生命周期，以确保这些值的存活时间足够长。
- 我们会在讲到所有权（ownership）时详细讨论借用（borrow）。

切片

切片 (slice) 的作用是提供对集合 (collection) 的视图 (view):

```
fn main() {
    let mut a: [i32; 6] = [10, 20, 30, 40, 50, 60];
    println!("a: {a:?}");

    let s: &[i32] = &a[2..4];

    println!("s: {s:?}");
}
```

- 切片从被切片的类型中借用 (borrow) 数据。
- Question: What happens if you modify `a[3]` right before printing `s`?

▼ *Speaker Notes*

- We create a slice by borrowing `a` and specifying the starting and ending indexes in brackets.
- If the slice starts at index 0, Rust's range syntax allows us to drop the starting index, meaning that `&a[0..a.len()]` and `&a[..a.len()]` are identical.
- The same is true for the last index, so `&a[2..a.len()]` and `&a[2..]` are identical.
- To easily create a slice of the full array, we can therefore use `&a[..]`.
- `s` is a reference to a slice of `i32` s. Notice that the type of `s` (`&[i32]`) no longer mentions the array length. This allows us to perform computation on slices of different sizes.
- Slices always borrow from another object. In this example, `a` has to remain 'alive' (in scope) for at least as long as our slice.
- 关于修改"a[3]"的问题可能会引发一些有趣的讨论，但正解是，出于内存安全方面的原因，您无法在执行作业的这个时间点通过"a"来进行此修改，但可以从"a"和"s"安全地读取数据。它会在您创建 Slice 之前运作，在"println"之后（不再使用 Slice 时）再次运作。更多详情会在"借用检查器"部分中加以说明。

“String”与“str”的区别

现在我们就可以理解 Rust 中的两种字符串类型：

```
fn main() {
    let s1: &str = "World";
    println!("s1: {s1}");

    let mut s2: String = String::from("Hello ");
    println!("s2: {s2}");
    s2.push_str(s1);
    println!("s2: {s2}");

    let s3: &str = &s2[6..];
    println!("s3: {s3}");
}
```

Rust 术语：

- `&str` 是一个指向字符串片段的不可变引用。
- `String` 是一个可变字符串缓冲区。

▼ *Speaker Notes*

- `&str` 引入了一个字符串切片，它是一个指向保存在内存块中的 UTF-8 编码字符串数据的不可变引用。 字符串字面量（ ”Hello” ）会保存在程序的二进制文件中。
- Rust 的 `String` 类型是一个字节 vector 的封装。和 `Vec<T>` 一样，它是拥有所有权的。
- 和其他类型一样， `String::from()` 会从字符串字面量创建一个字符串； `String::new()` 会创建一个新的空字符串， 之后可以使用 `push()` 和 `push_str()` 方法向其中添加字符串数据。
- `format!()` 宏可以方便地动态生成拥有所有权的字符串。它接受和 `println!()` 相同的格式规范。
- 你可以通过 `&` 和可选的范围选择从 `String` 中借用 `&str` 切片。
- 对于 C++ 程序员：可以把 `&str` 当作 C++ 中的 `const char*` ，但是它总是指向内存中的一个有效字符串。 Rust 的 `String` 大致相当于 C++ 中 `std::string` （主要区别：它只能包含 UTF-8 编码的字节， 并且永远不会使用小字符串优化（small-string optimization））。

函数

一个 Rust 版本的著名 FizzBuzz 面试题：

```rust
fn main() {
    print_fizzbuzz_to(20);
}

fn is_divisible(n: u32, divisor: u32) -> bool {
    if divisor == 0 {
        return false;
    }
    n % divisor == 0
}

fn fizzbuzz(n: u32) -> String {
    let fizz = if is_divisible(n, 3) { "fizz" } else { "" };
    let buzz = if is_divisible(n, 5) { "buzz" } else { "" };
    if fizz.is_empty() && buzz.is_empty() {
        return format!("{n}");
    }
    format!("{fizz}{buzz}")
}

fn print_fizzbuzz_to(n: u32) {
    for i in 1..=n {
        println!("{}", fizzbuzz(i));
    }
}
```

▼ *Speaker Notes*

- 我们在 `main` 中引用了下面编写的一个函数。不需要提前声明或添加头文件。
- 类型跟随在声明的参数后（与某些编程语言相反），然后是返回类型。
- 函数体（或任何块）中的最后一个表达式将成为返回值。只需省略表达式末尾的 `;` 即可。
- 有些函数没有返回值，会返回“单元类型（unit type）”`()`。如果省略了 `-> ()` 的返回类型，编译器将会自动推断。
- `print_fizzbuzz_to()` 函数中 `for` 循环的范围表达式（range expression）包含 `=n`，这会导致它包括上限。

Rustdoc

Rust 中的所有语言元素都可以通过特殊的 `///` 语法进行文档化。

```rust
/// Determine whether the first argument is divisible by the second argument.
///
/// If the second argument is zero, the result is false.
///
/// # Example
/// ```
/// assert!(is_divisible_by(42, 2));
/// ```
fn is_divisible_by(lhs: u32, rhs: u32) -> bool {
    if rhs == 0 {
        return false;  // Corner case, early return
    }
    lhs % rhs == 0     // The last expression in a block is the return value
}
```

The contents are treated as Markdown. All published Rust library crates are automatically documented at `docs.rs` using the rustdoc tool. It is idiomatic to document all public items in an API using this pattern. Code snippets can document usage and will be used as unit tests.

▼ *Speaker Notes*

- 向学生展示在 `docs.rs/rand` 中为 `rand` crate 生成的文档。
- 本课程的幻灯片中不包含 rustdoc，这是为了节省空间，但是在实际的代码中，应当编写相关的程序文档。
- 内部文档注释将在稍后（在讲解模块的页面）讨论，这里无需进行说明。
- Rustdoc 注释可以包含我们可使用“cargo test”运行和测试的代码段。我们将在“测试”部分中讨论这些测试。

方法

方法是与某种类型关联的函数。方法的 `self` 参数是与其关联类型的一个实例：

```
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }

    fn inc_width(&mut self, delta: u32) {
        self.width += delta;
    }
}

fn main() {
    let mut rect = Rectangle { width: 10, height: 5 };
    println!("old area: {}", rect.area());
    rect.inc_width(5);
    println!("new area: {}", rect.area());
}
```

- 我们将在今天的练习和明天的课程中更深入地学习方法相关的概念。

▼ *Speaker Notes*

- Add a static method called `Rectangle::new` and call this from `main`:

  ```
  fn new(width: u32, height: u32) -> Rectangle {
      Rectangle { width, height }
  }
  ```

- 虽然从技术层面来讲，Rust 没有自定义构造函数，但静态方法通常用于初始化结构体（但并非必须这样做）。您可以直接调用实际构造函数“Rectangle { width, height }”。请参阅 Rust 秘典。

- Add a `Rectangle::square(width: u32)` constructor to illustrate that such static methods can take arbitrary parameters.

函数重载

不支持重载：

- 每一个函数都只有一种实现：
  - 始终接受固定个数的形参。
  - 始终接受一组形参类型。
- 不支持提供默认值：
  - 实参的数量在所有调用的地方都是一样的。
  - 有时可以用宏（Macro）作为替代。

然而，函数形参可以是泛型（generics）：

```
fn pick_one<T>(a: T, b: T) -> T {
    if std::process::id() % 2 == 0 { a } else { b }
}

fn main() {
    println!("coin toss: {}", pick_one("heads", "tails"));
    println!("cash prize: {}", pick_one(500, 1000));
}
```

▼ *Speaker Notes*

- 标准库中的 `Into<T>` 通过泛型参数提供了一种具有有限多态性的参数类型。详见之后的章节。

第一天上午习题

在这些习题中，我们将探索 Rust 的两个部分：

- 类型之间的隐式转换。
- 数组和 `for` 循环。

▼ *Speaker Notes*

在解题时要考虑几件事：

- 最好使用本地安装的 Rust，以实现在编辑器中自动补全。关于安装 Rust 的细节，请参见 [使用 Cargo] 页面。
- 也可以使用 Rust Playground 作为替代。

页面内嵌的代码片段是不可编辑的：因为离开页面后内嵌代码片段中的修改会丢失。

After looking at the exercises, you can look at the solutions provided.

隐式类型转换

与 C++ 不同，Rust 不会自动进行 *隐式类型转换*。例如，下面的程序中不存在隐式类型转换：

```
fn multiply(x: i16, y: i16) -> i16 {
    x * y
}

fn main() {
    let x: i8 = 15;
    let y: i16 = 1000;

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

Rust 的整数类型都实现了 `From<T>` 和 `Into<T>` trait，使得我们可以在它们之间进行转换。`From<T>` trait 包含 `from()` 方法，`Into<T>` trait 包含 `into()` 方法。类型通过实现这些 trait 来表达它将被如何转换为另一个类型。

标准库中包含 `From<i8> for i16` 的实现，即我们可以通过调用 `i16::from(x)` 来将 `i8` 类型的变量 `x` 转换为 `i16`。或者也可以简单地使用 `x.into()`，因为 `From<i8> for i16` 的实现会自动创建 `Into<i16> for i8` 的实现。

这同样也适用于自定义类型的 `From` 实现，只需实现 `From` 就可以自动得到对应的 `Into` 实现。

1. 执行上述程序，并查看对应的编译错误。
2. 修改代码，使用 `into()` 进行类型转换。
3. 修改 `x` 和 `y` 的类型（例如 `f32`，`bool`，`i128` 等）来了解哪些类型之间可以相互转换。尝试将较小的类型转换为较大的类型和将较大的类型转换为较小的类型。阅读 标准库文档 来了解对于你所尝试的两个类型 `From<T>` 是否已被实现。

数组与 for 循环

我们可以这样声明一个数组：

```
let array = [10, 20, 30];
```

你可以使用 `{:?}` 来打印这种数组的调试格式：

```
fn main() {
    let array = [10, 20, 30];
    println!("array: {array:?}");
}
```

在 Rust 中，可以使用 `for` 关键词遍历数组和区间等元素：

```
fn main() {
    let array = [10, 20, 30];
    print!("Iterating over array:");
    for n in &array {
        print!(" {n}");
    }
    println!();

    print!("Iterating over range:");
    for i in 0..3 {
        print!(" {}", array[i]);
    }
    println!();
}
```

使用以上知识，写一个用易读的格式输出矩阵的 `pretty_print` 函数，以及一个对矩阵进行转置（将行和列互换）的 `transpose` 函数：

$$\text{transpose}\left(\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{bmatrix}\right) == \begin{bmatrix}1 & 4 & 7\\ 2 & 5 & 8\\ 3 & 6 & 9\end{bmatrix}$$

硬编码这两个函数，让它们处理 3 × 3 的矩阵。

将下面的代码复制到 https://play.rust-lang.org/ 并实现上述函数：

```rust
// TODO: remove this when you're done with your implementation.
#![allow(unused_variables, dead_code)]

fn transpose(matrix: [[i32; 3]; 3]) -> [[i32; 3]; 3] {
    unimplemented!()
}

fn pretty_print(matrix: &[[i32; 3]; 3]) {
    unimplemented!()
}

fn main() {
    let matrix = [
        [101, 102, 103], // <-- the comment makes rustfmt add a newline
        [201, 202, 203],
        [301, 302, 303],
    ];

    println!("matrix:");
    pretty_print(&matrix);

    let transposed = transpose(matrix);
    println!("transposed:");
    pretty_print(&transposed);
}
```

附加题

是否可以使用 `&[i32]` 切片而不是硬编码的 3 × 3 矩阵作为函数的参数和返回类型？例如使用 `&[&[i32]]` 表示一个二维的切片的切片。为什么这样做是可行或不可行的？

参考 `ndarray` crate 以了解该功能满足生产环境质量的实现。

▼ *Speaker Notes*

题目解答和附加题的答案在 题解 章节中。

在“for n in &array”中使用引用“&array”这一做法巧妙地预先展示了下午将谈到的所有权问题。

如果不使用“&”...

- 循环将会是一个使用数组的循环。这是一项在 2021 年版中引入的变更。
- 会发生隐式数组复制。由于“i32”是复制类型，因此“[i32; 3]”也是复制类型。

控制流

正如我们所知， `if` 是 Rust 中的一个表达式。它用于有条件地 评估两个块中的一个，但这些块可以有一个值， 然后成为 `if` 表达式的值。其他控制流表达式在 Rust 中也有类似 的运作方式。

块

A block in Rust contains a sequence of expressions. Each block has a value and a type, which are those of the last expression of the block:

```
fn main() {
    let x = {
        let y = 10;
        println!("y: {y}");
        let z = {
            let w = {
                3 + 4
            };
            println!("w: {w}");
            y * w
        };
        println!("z: {z}");
        z - y
    };
    println!("x: {x}");
}
```

If the last expression ends with `;`, then the resulting value and type is `()`.

同样的规则也适用于函数：函数主体的值 是返回值：

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

fn main() {
    println!("double: {}", double(7));
}
```

▼ *Speaker Notes*

关键点：

- 这张幻灯片的重点是说明在 Rust 中，块有类型和值。
- 你可以通过更改块的最后一行，来展示块值的变化情况。例如，添加/移除分号或使用 `return`。

if 表达式

`if` 表达式 的用法与其他语言中的 `if` 语句完全一样。

```
fn main() {
    let mut x = 10;
    if x % 2 == 0 {
        x = x / 2;
    } else {
        x = 3 * x + 1;
    }
}
```

此外，你还可以将 `if` 用作一个表达式。每个块的最后一个表达式 将成为 `if` 表达式的值：

```
fn main() {
    let mut x = 10;
    x = if x % 2 == 0 {
        x / 2
    } else {
        3 * x + 1
    };
}
```

▼ *Speaker Notes*

由于 `if` 是一个表达式且必须有一个特定的类型，因此它的两个分支块必须有相同的类型。考虑在第二个示例中将 `;` 添加到 `x / 2` 的后面，看看会出现什么情况。

for 循环

The `for` loop is closely related to the `while let` loop. It will automatically call `into_iter()` on the expression and then iterate over it:

```
fn main() {
    let v = vec![10, 20, 30];

    for x in v {
        println!("x: {x}");
    }

    for i in (0..10).step_by(2) {
        println!("i: {i}");
    }
}
```

你可以在此照常使用 `break` 和 `continue` 。

▼ *Speaker Notes*

- 在这种情况下，索引迭代在 Rust 中并不是一个特殊的语法。
- `(0..10)` 是实现 `Iterator` trait 的区间。
- `step_by` 是返回另一个 `Iterator` 的方法，用于逐一跳过所有其他元素。
- 修改矢量中的元素并说明编译器错误。将矢量 `v` 改为可变，并将 for 循环改为 `for x in v.iter_mut()` 。

while 循环

`while` 关键字 的工作方式与其他语言非常相似：

```
fn main() {
    let mut x = 10;
    while x != 1 {
        x = if x % 2 == 0 {
            x / 2
        } else {
            3 * x + 1
        };
    }
    println!("x: {x}");
}
```

break 和 continue

- 如果你想提前退出循环，请使用 `break`，
- 如果需要立即启动 下一次迭代，请使用 `continue`。

`continue` 和 `break` 都可以选择接受一个标签参数，用来 终止嵌套循环：

```
fn main() {
    let v = vec![10, 20, 30];
    let mut iter = v.into_iter();
    'outer: while let Some(x) = iter.next() {
        println!("x: {x}");
        let mut i = 0;
        while i < x {
            println!("x: {x}, i: {i}");
            i += 1;
            if i == 3 {
                break 'outer;
            }
        }
    }
}
```

在本示例中，我们会在内循环 3 次迭代后终止外循环。

loop 表达式

最后是用于创建无限循环的 `loop` 关键字 。

在下例中，你必须 `break` 或 `return` 才能停止循环：

```
fn main() {
    let mut x = 10;
    loop {
        x = if x % 2 == 0 {
            x / 2
        } else {
            3 * x + 1
        };
        if x == 1 {
            break;
        }
    }
    println!("x: {x}");
}
```

▼ *Speaker Notes*

- 用一个值（例如 `break 8` ）来中断 `loop` 并将其输出。
- 请注意， `loop` 是唯一返回有意义的值的循环结构。 这是因为它保证至少被输入一次（与 `while` 和 `for` 循环不同）。

变量

Rust 通过静态类型实现了类型安全。变量绑定默认是不可变的：

```
fn main() {
    let x: i32 = 10;
    println!("x: {x}");
    // x = 20;
    // println!("x: {x}");
}
```

▼ *Speaker Notes*

- 由于类型推导， `i32` 可以省略。随着课程推进，我们会越来越少地看到类型声明。

类型推导

Rust 会根据变量的使用来确定其类型：

```
fn takes_u32(x: u32) {
    println!("u32: {x}");
}

fn takes_i8(y: i8) {
    println!("i8: {y}");
}

fn main() {
    let x = 10;
    let y = 20;

    takes_u32(x);
    takes_i8(y);
    // takes_u32(y);
}
```

▼ *Speaker Notes*

这张幻灯片演示了 Rust 编译器是如何根据变量声明和用法来推导其类型的。

需要重点强调的是这样声明的变量并非像那种动态类型语言中可以持有任何数据的“任何类型”。这种声明所生成的机器码与明确类型声明完全相同。编译器进行类型推导能够让我们编写更简略的代码。

The following code tells the compiler to copy into a certain generic container without the code ever explicitly specifying the contained type, using `_` as a placeholder:

```
fn main() {
    let mut v = Vec::new();
    v.push((10, false));
    v.push((20, true));
    println!("v: {v:?}");

    let vv = v.iter().collect::<std::collections::HashSet<_>>();
    println!("vv: {vv:?}");
}
```

`collect` relies on `FromIterator`, which `HashSet` implements.

静态变量和常量

静态变量和常量是创建全局范围值的两种不同方法，这类值在程序执行期间无法移动或重新分配。

const

系统会在编译时对常量变量进行求值；无论在何处使用，其值都会被内嵌：

```
const DIGEST_SIZE: usize = 3;
const ZERO: Option<u8> = Some(42);

fn compute_digest(text: &str) -> [u8; DIGEST_SIZE] {
    let mut digest = [ZERO.unwrap_or(0); DIGEST_SIZE];
    for (idx, &b) in text.as_bytes().iter().enumerate() {
        digest[idx % DIGEST_SIZE] = digest[idx % DIGEST_SIZE].wrapping_add(b
    }
    digest
}

fn main() {
    let digest = compute_digest("Hello");
    println!("digest: {digest:?}");
}
```

根据 Rust RFC Book 这些变量在使用时是内联 (inlined) 的。

在编译时只能调用标记为“const”的函数以生成“const”值。不过，可在运行时调用“const”函数。

static

静态变量在程序的整个执行过程中始终有效，因此不会移动：

```
static BANNER: &str = "Welcome to RustOS 3.14";

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

As noted in the Rust RFC Book, these are not inlined upon use and have an actual associated memory location. This is useful for unsafe and embedded code, and the variable lives through the entirety of the program execution. When a globally-scoped value does not have a reason to need object identity, `const` is generally preferred.

由于“static”变量可从任何线程访问，因此它们必须是“Sync”。内部可变性可通过“互斥量”、原子性或类似对象实现。也可能具有可变静态项，但它们需要手动同步，因此对它们的任何访问都需要

“unsafe”代码。我们将在“不安全 Rust”章节中探讨可变静态项。

▼ *Speaker Notes*

- 值得一提的是， `const` 在语义上与C++的 `constexpr` 类似。
- 另一方面， `static` 远远更类似于C++中的 `const` 或可改变的全局变量。
- `static` provides object identity: an address in memory and state as required by types with interior mutability such as `Mutex<T>` .
- 虽然需要使用在运行中求值的常量的情况并不是很常见，但是它是有帮助的，而且比使用静态变量更安全。
- 可以使用宏“std::thread_local”创建“thread_local”数据。

属性表：

属性	Static	常量
在内存中有地址	是	否（内嵌）
Lives for the entire duration of the program	是	否
可变	是（不安全）	否
Evaluated at compile time	是（在编译时被初始化）	是
内嵌在使用它的任何位置	否	是

作用域和遮蔽（Shadowing）

你可以隐藏变量，位于外部作用域的变量和 相同作用域的变量都可以：

```
fn main() {
    let a = 10;
    println!("before: {a}");

    {
        let a = "hello";
        println!("inner scope: {a}");

        let a = true;
        println!("shadowed in inner scope: {a}");
    }

    println!("after: {a}");
}
```

▼ *Speaker Notes*

- 定义: 遮蔽和变更（mutation）不同，因为在遮蔽之后，两个变量都会同时存在于内存的不同位置中。在同一个名字下的两个变量都是可以被使用的，但是你在代码的哪里使用会最终决定你使用哪一个变量。
- 一个遮蔽变量可以具有不同的类型。
- 隐藏起初看起来会有些晦涩，但是它很便于存 `.unwrap()` 之后的得到的值。
- 以下代码说明了为什么在作用域内隐藏一个不可变的变量时，即使是在变量类型没有改变的情况下，编译器也不能简单地重复利用之前的内存位置。

```
fn main() {
    let a = 1;
    let b = &a;
    let a = a + 1;
    println!("{a} {b}");
}
```

枚举

`enum` 关键字允许创建具有几个 不同变体的类型：

```
fn generate_random_number() -> i32 {
    // Implementation based on https://xkcd.com/221/
    4  // Chosen by fair dice roll. Guaranteed to be random.
}

#[derive(Debug)]
enum CoinFlip {
    Heads,
    Tails,
}

fn flip_coin() -> CoinFlip {
    let random_number = generate_random_number();
    if random_number % 2 == 0 {
        return CoinFlip::Heads;
    } else {
        return CoinFlip::Tails;
    }
}

fn main() {
    println!("You got: {:?}", flip_coin());
}
```

▼ *Speaker Notes*

关键点：

- 枚举允许你从一种类型下收集一组值
- This page offers an enum type `CoinFlip` with two variants `Heads` and `Tails`. You might note the namespace when using variants.
- 这可能是比较结构体和枚举的好时机：
  - 在这两者中，你可以获得一个不含字段的简单版本（单位结构体），或一个包含不同类型字段的版本（变体载荷）。
  - 在这两者中，关联的函数都在 `impl` 块中定义。
  - 你甚至可以使用单独的结构体实现枚举的不同变体，但这样一来，如果它们都已在枚举中定义，类型与之前也不一样。

变体载荷

你可以定义更丰富的枚举，其中变体会携带数据。然后，你可以使用 `match` 语句从每个变体中提取数据：

```
enum WebEvent {
    PageLoad,                 // Variant without payload
    KeyPress(char),           // Tuple struct variant
    Click { x: i64, y: i64 }, // Full struct variant
}

#[rustfmt::skip]
fn inspect(event: WebEvent) {
    match event {
        WebEvent::PageLoad       => println!("page loaded"),
        WebEvent::KeyPress(c)    => println!("pressed '{c}'"),
        WebEvent::Click { x, y } => println!("clicked at x={x}, y={y}"),
    }
}

fn main() {
    let load = WebEvent::PageLoad;
    let press = WebEvent::KeyPress('x');
    let click = WebEvent::Click { x: 20, y: 80 };

    inspect(load);
    inspect(press);
    inspect(click);
}
```

▼ *Speaker Notes*

- 枚举变体中的值只有在被模式匹配后，才可访问。模式将引用绑定到 `=>` 之后的“match 分支”中的字段。
  - 表达式会从上到下与模式匹配。没有像 C 或 C++ 中那样的跳转。
  - 匹配表达式拥有一个值。值是 match 分支中被执行的最后一个表达式。
  - 从顶部开始，查找与该值匹配的模式，然后沿箭头运行代码。一旦找到匹配，我们便会停止。
- 展示搜索不详尽时会发生的情况。请注意 Rust 编译器的优势，即确认所有情况何时都得到了处理。
- `match` 会检查 `enum` 中的隐藏的判别字段。
- 可以通过调用 `std::mem::discriminant()` 来检索判别
  - 这很有用，例如如果为结构体实现 `PartialEq`，比较字段值不会影响等式。
- `WebEvent::Click { ... }` 与含顶层 `struct Click { ... }` 的 `WebEvent::Click(Click)` 不完全相同。例如，内嵌版本无法实现 trait。

枚举大小

Rust 枚举被紧密地打包，考虑到了对齐的影响，因此存在一些限制：

```rust
use std::any::type_name;
use std::mem::{align_of, size_of};

fn dbg_size<T>() {
    println!("{}: size {} bytes, align: {} bytes",
        type_name::<T>(), size_of::<T>(), align_of::<T>());
}

enum Foo {
    A,
    B,
}

fn main() {
    dbg_size::<Foo>();
}
```

- 请参阅 Rust 引用。

▼ *Speaker Notes*

关键点：

- Internally Rust is using a field (discriminant) to keep track of the enum variant.
- You can control the discriminant if needed (e.g., for compatibility with C):

  ```rust
  #[repr(u32)]
  enum Bar {
      A,  // 0
      B = 10000,
      C,  // 10001
  }

  fn main() {
      println!("A: {}", Bar::A as u32);
      println!("B: {}", Bar::B as u32);
      println!("C: {}", Bar::C as u32);
  }
  ```

  Without `repr`, the discriminant type takes 2 bytes, because 10001 fits 2 bytes.

- Try out other types such as
  - `dbg_size!(bool)`: size 1 bytes, align: 1 bytes,
  - `dbg_size!(Option<bool>)`: size 1 bytes, align: 1 bytes (niche optimization, see below),

    - `dbg_size!(&i32)` : size 8 bytes, align: 8 bytes (on a 64-bit machine),
    - `dbg_size!(Option<&i32>)` : size 8 bytes, align: 8 bytes (null pointer optimization, see below).

- Niche optimization: Rust will merge unused bit patterns for the enum discriminant.
- Null pointer optimization: For some types, Rust guarantees that `size_of::<T>()` equals `size_of::<Option<T>>()` .

  Example code if you want to show how the bitwise representation *may* look like in practice. It's important to note that the compiler provides no guarantees regarding this representation, therefore this is totally unsafe.

```
use std::mem::transmute;

macro_rules! dbg_bits {
    ($e:expr, $bit_type:ty) => {
        println!("- {}: {:#x}", stringify!($e), transmute::<_, $bit_ty
    };
}

fn main() {
    unsafe {
        println!("bool:");
        dbg_bits!(false, u8);
        dbg_bits!(true, u8);

        println!("Option<bool>:");
        dbg_bits!(None::<bool>, u8);
        dbg_bits!(Some(false), u8);
        dbg_bits!(Some(true), u8);

        println!("Option<Option<bool>>:");
        dbg_bits!(Some(Some(false)), u8);
        dbg_bits!(Some(Some(true)), u8);
        dbg_bits!(Some(None::<bool>), u8);
        dbg_bits!(None::<Option<bool>>, u8);

        println!("Option<&i32>:");
        dbg_bits!(None::<&i32>, usize);
        dbg_bits!(Some(&0i32), usize);
    }
}
```

  如果您想讨论将 256 多个“Option”链在一起时会发生什么情况，可以使用下方这个更复杂的示例。

```rust
#![recursion_limit = "1000"]

use std::mem::transmute;

macro_rules! dbg_bits {
    ($e:expr, $bit_type:ty) => {
        println!("- {}: {:#x}", stringify!($e), transmute::<_, $bit_ty
    };
}

// Macro to wrap a value in 2^n Some() where n is the number of "@" si
// Increasing the recursion limit is required to evaluate this macro.
macro_rules! many_options {
    ($value:expr) => { Some($value) };
    ($value:expr, @) => {
        Some(Some($value))
    };
    ($value:expr, @ $($more:tt)+) => {
        many_options!(many_options!($value, $($more)+), $($more)+)
    };
}

fn main() {
    // TOTALLY UNSAFE. Rust provides no guarantees about the bitwise
    // representation of types.
    unsafe {
        assert_eq!(many_options!(false), Some(false));
        assert_eq!(many_options!(false, @), Some(Some(false)));
        assert_eq!(many_options!(false, @@), Some(Some(Some(Some(false

        println!("Bitwise representation of a chain of 128 Option's.")
        dbg_bits!(many_options!(false, @@@@@@@), u8);
        dbg_bits!(many_options!(true, @@@@@@@), u8);

        println!("Bitwise representation of a chain of 256 Option's.")
        dbg_bits!(many_options!(false, @@@@@@@@), u16);
        dbg_bits!(many_options!(true, @@@@@@@@), u16);

        println!("Bitwise representation of a chain of 257 Option's.")
        dbg_bits!(many_options!(Some(false), @@@@@@@@), u16);
        dbg_bits!(many_options!(Some(true), @@@@@@@@), u16);
        dbg_bits!(many_options!(None::<bool>, @@@@@@@@), u16);
    }
}
```

新式控制流

Rust 有几个与其他语言不同的控制流结构。它们用于模式匹配：

- `if let` 表达式
- `while let` expressions
- `match` 表达式

if let 表达式

`if let` 表达式 能让你根据某个值是否与模式相匹配来执行不同的代码：

```
fn main() {
    let arg = std::env::args().next();
    if let Some(value) = arg {
        println!("Program name: {value}");
    } else {
        println!("Missing name?");
    }
}
```

如需详细了解 Rust 中 的模式，请参阅模式匹配。

▼ *Speaker Notes*

- Unlike `match`, `if let` does not have to cover all branches. This can make it more concise than `match`.
- 使用 `Option` 时，常见的做法是处理 `Some` 值。
- 与 `match` 不同的是， `if let` 不支持模式匹配的 guard 子句。
- Since 1.65, a similar let-else construct allows to do a destructuring assignment, or if it fails, execute a block which is required to abort normal control flow (with `panic` / `return` / `break` / `continue`):

```
fn main() {
    println!("{:?}", second_word_to_upper("foo bar"));
}

fn second_word_to_upper(s: &str) -> Option<String> {
    let mut it = s.split(' ');
    let (Some(_), Some(item)) = (it.next(), it.next()) else {
        return None;
    };
    Some(item.to_uppercase())
}

```

while let 循环

与 `if let` 一样， `while let` 变体会针对一个模式重复测试一个值：

```
fn main() {
    let v = vec![10, 20, 30];
    let mut iter = v.into_iter();

    while let Some(x) = iter.next() {
        println!("x: {x}");
    }
}
```

Here the iterator returned by `v.into_iter()` will return a `Option<i32>` on every call to `next()`. It returns `Some(x)` until it is done, after which it will return `None`. The `while let` lets us keep iterating through all items.

如需详细了解 Rust 中 的模式，请参阅模式匹配。

▼ *Speaker Notes*

- 指出只要值与模式匹配， `while let` 循环就会一直进行下去。
- 你可以使用 if 语句将 `while let` 循环重写为无限循环，当 `iter.next()` 没有值可以解封时中断。 `while let` 为上述情况提供了语法糖。

match 表达式

`match` 关键字 用于将一个值与一个或多个模式进行匹配。从这个意义上讲，它的工作方式 类似于一系列的 `if let` 表达式：

```
fn main() {
    match std::env::args().next().as_deref() {
        Some("cat") => println!("Will do cat things"),
        Some("ls")  => println!("Will ls some files"),
        Some("mv")  => println!("Let's move some files"),
        Some("rm")  => println!("Uh, dangerous!"),
        None        => println!("Hmm, no program name?"),
        _           => println!("Unknown program name!"),
    }
}
```

与 `if let` 类似，每个匹配分支必须有相同的类型。该类型是块的最后一个 表达式（如有）。在上例中，类型是 `()` 。

如需详细了解 Rust 中 的模式，请参阅模式匹配。

▼ *Speaker Notes*

- 将 match 表达式保存到一个变量中并输出结果。
- 移除 `.as_deref()` 并说明错误。
  - `std::env::args().next()` 会返回 `Option<String>` ，但无法与 `String` 进行匹配。
  - `as_deref()` 会将 `Option<T>` 转换为 `Option<&T::Target>` 。在我们的示例中，这会将 `Option<String>` 转换为 `Option<&str>` 。
  - 现在，我们可以使用模式匹配来匹配 `Option` 中的 `&str` 。

模式匹配

使用关键词 `match` 对一个值进行模式匹配。进行匹配时，会从上至下依次进行比较，并选定第一个匹配成功的结果。

模式（pattern）可以是简单的值，其用法类似于 C 与 C++ 中的 `switch` 。

```
fn main() {
    let input = 'x';

    match input {
        'q'                   => println!("Quitting"),
        'a' | 's' | 'w' | 'd' => println!("Moving around"),
        '0'..='9'             => println!("Number input"),
        _                     => println!("Something else"),
    }
}
```

模式 `_` 是外卡 (wildcard) 模式。它可以匹配任何值。

▼ *Speaker Notes*

关键点：

- You might point out how some specific characters are being used when in a pattern
  - `|` as an `or`
  - `..` can expand as much as it needs to be
  - `1..=5` represents an inclusive range
  - “_”是通配符
- 展示绑定的运作方式可能会很有帮助，例如通过用变量替换通配符或移除“q”周围的引号来展示。
- 您可以在参照项上演示如何匹配。
- 这时可能很适合提到“不可反驳的模式”这个概念，因为这个术语可能会出现在错误消息中。

解构枚举

模式还可用于将变量绑定到值的某些部分。这是您检查类型结构的方式。我们先从简单的“enum”类型开始：

```
enum Result {
    Ok(i32),
    Err(String),
}

fn divide_in_two(n: i32) -> Result {
    if n % 2 == 0 {
        Result::Ok(n / 2)
    } else {
        Result::Err(format!("cannot divide {n} into two equal parts"))
    }
}

fn main() {
    let n = 100;
    match divide_in_two(n) {
        Result::Ok(half) => println!("{n} divided in two is {half}"),
        Result::Err(msg) => println!("sorry, an error happened: {msg}"),
    }
}
```

在这里，我们使用了分支来解构“Result”值。在第一个分支中，“half”被绑定到“Ok”变体中的值。在第二个分支中，“msg”被绑定到错误消息。

▼ *Speaker Notes*

关键点：

- “if”/“else”表达式将返回一个枚举，该枚举之后会使用“match”进行解封装。
- 您可以尝试在枚举定义中添加第三个变体，并在运行代码时显示错误。指出代码现在有哪些地方还不详尽，并说明编译器会如何尝试给予提示。

解构结构体

您还可以解构“structs”：

```
struct Foo {
    x: (u32, u32),
    y: u32,
}

#[rustfmt::skip]
fn main() {
    let foo = Foo { x: (1, 2), y: 3 };
    match foo {
        Foo { x: (1, b), y } => println!("x.0 = 1, b = {b}, y = {y}"),
        Foo { y: 2, x: i }   => println!("y = 2, x = {i:?}"),
        Foo { y, .. }        => println!("y = {y}, other fields were ignored
    }
}
```

▼ *Speaker Notes*

- 更改“foo”中的字面量值以与其他模式相匹配。
- 向“Foo”添加一个新字段，并根据需要更改模式。
- 捕获和常量表达式之间的区别可能很难发现。尝试将第二个分支中的“2”更改为一个变量，可以看到它几乎无法运作了。将它更改为“const”，可以看到它又正常运作了。

解构数组

你可以通过元素匹配来解构数组、元组和切片：

```rust
#[rustfmt::skip]
fn main() {
    let triple = [0, -2, 3];
    println!("Tell me about {triple:?}");
    match triple {
        [0, y, z] => println!("First is 0, y = {y}, and z = {z}"),
        [1, ..]   => println!("First is 1 and the rest were ignored"),
        _         => println!("All elements were ignored"),
    }
}
```

▼ *Speaker Notes*

- 对未知长度的切片进行解构也可以使用固定长度的模式。

```rust
fn main() {
    inspect(&[0, -2, 3]);
    inspect(&[0, -2, 3, 4]);
}

#[rustfmt::skip]
fn inspect(slice: &[i32]) {
    println!("Tell me about {slice:?}");
    match slice {
        &[0, y, z] => println!("First is 0, y = {y}, and z = {z}"),
        &[1, ..]   => println!("First is 1 and the rest were ignored")
        _          => println!("All elements were ignored"),
    }
}
```

- 使用 `_` 创建一个新的模式来代表一个元素。
- 向数组中添加更多的值。
- 指出 `..` 是如何扩展以适应不同数量的元素的。
- 展示使用模式 `[.., b]` 和 `[a@..,b]` 来匹配切片的尾部。

匹配守卫

匹配时，您可以向模式中添加“守卫”。这是一个任意布尔表达式，如果模式匹配，就会执行该表达式：

```rust
#[rustfmt::skip]
fn main() {
    let pair = (2, -2);
    println!("Tell me about {pair:?}");
    match pair {
        (x, y) if x == y     => println!("These are twins"),
        (x, y) if x + y == 0 => println!("Antimatter, kaboom!"),
        (x, _) if x % 2 == 1 => println!("The first one is odd"),
        _                    => println!("No correlation..."),
    }
}
```

▼ *Speaker Notes*

关键点：

- 有些想法比模式本身所允许的程度更加复杂，如果我们希望简要地表达这些想法，就必须把匹配守卫视为独立的语法功能。
- 它们与匹配分支中的单独“if”表达式不同。选择匹配分支后，分支块内（在“=>”之后）会出现“if”表达式。如果该分支块内的“if”条件失败，系统不会考虑原始“match”表达式的其他分支。
- 您可以在 if 表达式中使用模式中定义的变量。
- 只要表达式在包含“|”的模式中，就会适用守卫定义的条件。

第 1 天：下午练习

我们将关注以下两方面：

- The Luhn algorithm,
- An exercise on pattern matching.

▼ *Speaker Notes*

After looking at the exercises, you can look at the solutions provided.

卢恩算法

卢恩算法用于验证信用卡号。该算法将字符串作为输入内容，并执行以下操作来验证信用卡号：

- 忽略所有空格。拒绝少于两位的号码。
- 从**右到左**，将偶数位的数字乘二。对于数字“1234”，我们将“3”和“1”乘二；对于数字“98765”，将“6”和“8”乘二。
- 将一个数字乘二后，如果结果大于 9，则将每位数字相加。因此，将“7”乘二得“14”，然后“1 + 4 = 5”。
- 将所有未乘二和已乘二的数字相加。
- 如果总和以“0”结尾，则信用卡号有效。

Copy the code below to https://play.rust-lang.org/ and implement the function.

使用“for”循环和整数，先尝试以简单的方式解决问题。然后，再次查看该解决方案，并尝试使用迭代器来实现它。

```rust
// TODO: remove this when you're done with your implementation.
#![allow(unused_variables, dead_code)]

pub fn luhn(cc_number: &str) -> bool {
    unimplemented!()
}

#[test]
fn test_non_digit_cc_number() {
    assert!(!luhn("foo"));
    assert!(!luhn("foo 0 0"));
}

#[test]
fn test_empty_cc_number() {
    assert!(!luhn(""));
    assert!(!luhn(" "));
    assert!(!luhn("  "));
    assert!(!luhn("    "));
}

#[test]
fn test_single_digit_cc_number() {
    assert!(!luhn("0"));
}

#[test]
fn test_two_digit_cc_number() {
    assert!(luhn(" 0 0 "));
}

#[test]
fn test_valid_cc_number() {
    assert!(luhn("4263 9826 4026 9299"));
    assert!(luhn("4539 3195 0343 6467"));
    assert!(luhn("7992 7398 713"));
}

#[test]
fn test_invalid_cc_number() {
    assert!(!luhn("4223 9826 4026 9299"));
    assert!(!luhn("4539 3195 0343 6476"));
    assert!(!luhn("8273 1232 7352 0569"));
}

#[allow(dead_code)]
fn main() {}
```

Exercise: Expression Evaluation

Let’s write a simple recursive evaluator for arithmetic expressions.

```rust
/// An operation to perform on two subexpressions.
#[derive(Debug)]
enum Operation {
    Add,
    Sub,
    Mul,
    Div,
}

/// An expression, in tree form.
#[derive(Debug)]
enum Expression {
    /// An operation on two subexpressions.
    Op {
        op: Operation,
        left: Box<Expression>,
        right: Box<Expression>,
    },

    /// A literal value
    Value(i64),
}

/// The result of evaluating an expression.
#[derive(Debug, PartialEq, Eq)]
enum Res {
    /// Evaluation was successful, with the given result.
    Ok(i64),
    /// Evaluation failed, with the given error message.
    Err(String),
}
// Allow `Ok` and `Err` as shorthands for `Res::Ok` and `Res::Err`.
use Res::{Err, Ok};

fn eval(e: Expression) -> Res {
    todo!()
}

#[test]
fn test_value() {
    assert_eq!(eval(Expression::Value(19)), Ok(19));
}

#[test]
fn test_sum() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(Expression::Value(10)),
            right: Box::new(Expression::Value(20)),
        }),
        Ok(30)
    );
}

#[test]
fn test_recursion() {
```

```
    let term1 = Expression::Op {
        op: Operation::Mul,
        left: Box::new(Expression::Value(10)),
        right: Box::new(Expression::Value(9)),
    };
    let term2 = Expression::Op {
        op: Operation::Mul,
        left: Box::new(Expression::Op {
            op: Operation::Sub,
            left: Box::new(Expression::Value(3)),
            right: Box::new(Expression::Value(4)),
        }),
        right: Box::new(Expression::Value(5)),
    };
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(term1),
            right: Box::new(term2),
        }),
        Ok(85)
    );
}

#[test]
fn test_error() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Div,
            left: Box::new(Expression::Value(99)),
            right: Box::new(Expression::Value(0)),
        }),
        Err(String::from("division by zero"))
    );
}
```

The `Box` type here is a smart pointer, and will be covered in detail later in the course. An expression can be "boxed" with `Box::new` as seen in the tests. To evaluate a boxed expression, use the deref operator to "unbox" it: `eval(*boxed_expr)`.

Some expressions cannot be evaluated and will return an error. The `Res` type represents either a successful value or an error with a message. This is very similar to the standard-library `Result` which we will see later.

Copy and paste the code into the Rust playground, and begin implementing `eval`. The final product should pass the tests. It may be helpful to use `todo!()` and get the tests to pass one-by-one.

If you finish early, try writing a test that results in an integer overflow. How could you handle this with `Res::Err` instead of a panic?

欢迎来到第二天

现在我们已经了解了相当多的Rust，接下来我们将学习：

- 内存管理：栈与堆，手动内存管理，基于作用域的内存管理，以及垃圾回收。
- 所有权：移动（move）的语义，复制（copy）和克隆（clone），借用（borrow），以及生命周期。
- Structs and methods.
- 标准库: 字符串（`String`），选项（`Option`） 和 结果（`Result`），动态数组（`Vec`），散列表（`HashMap`），引用计数（`Rc`） 和 共享引用计数（`Arc`）。
- 模块: 可见性, 路径和文件系统的层次结构。

内存管理

传统上，语言分为两大类：

- 通过手动内存管理实现完全控制：C、C++、Pascal...
- 运行时通过自动内存管理实现完全安全：Java、Python、Go、Haskell...

Rust 提供了一个全新的组合：

> 通过编译时强制执行正确的内存>管理来实现完全控制与安全。

它通过一个明确的所有权（ownership）概念来实现此目的。

首先，我们回顾一下内存管理的工作原理。

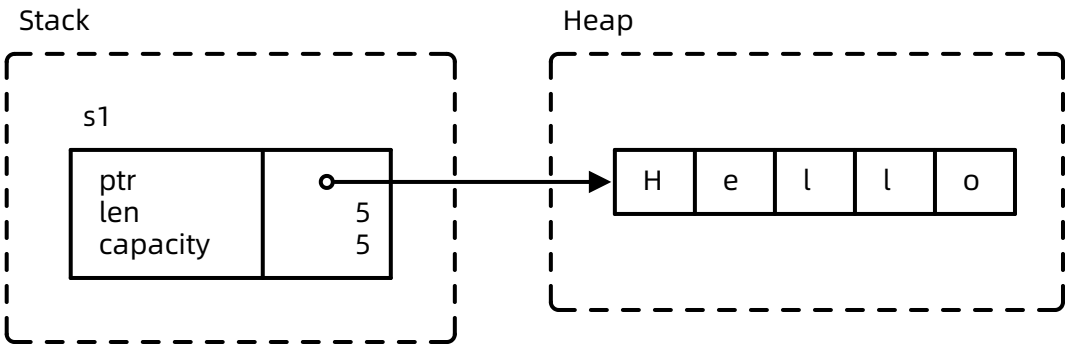
栈与堆

- 栈：局部变量的连续内存区域。
    - 值在编译时具有已知的固定大小。
    - 速度极快：只需移动一个栈指针。
    - 易于管理：遵循函数调用规则。
    - 优秀的内存局部性。
- 堆：函数调用之外的值的存储。
    - 值具有动态大小，具体大小需在运行时确定。
    - 比栈稍慢：需要向系统申请空间。
    - 不保证内存局部性。

Stack and Heap Example

Creating a `String` puts fixed-sized metadata on the stack and dynamically sized data, the actual string, on the heap:

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



▼ *Speaker Notes*

- 指出 `String` 底层由 `Vec` 实现，因此它具有容量和长度，如果值可变，则可以通过在堆上重新分配存储空间进行增长。
- 如果学员提出相关问题，你可以提及我们不仅能使用[系统分配器]在堆上分配底层内存，还能使用 Allocator API 实现自定义分配器
- 我们可以使用 `unsafe` 代码检查内存布局。不过，你应该指出，这种做法不安全！

```
fn main() {
    let mut s1 = String::from("Hello");
    s1.push(' ');
    s1.push_str("world");
    // DON'T DO THIS AT HOME! For educational purposes only.
    // String provides no guarantees about its layout, so this could l
    // undefined behavior.
    unsafe {
        let (ptr, capacity, len): (usize, usize, usize) = std::mem::tr
        println!("ptr = {ptr:#x}, len = {len}, capacity = {capacity}")
    }
}
```

手动内存管理

你自己实现堆内存分配和释放。

稍有不慎，这可能会导致崩溃、bug、安全漏洞和内存泄漏。

C++ 示例

你必须对使用 `malloc` 分配的每个指针调用 `free`：

```
void foo(size_t n) {
    int* int_array = malloc(n * sizeof(int));
    //
    // ... lots of code
    //
    free(int_array);
}
```

Memory is leaked if the function returns early between `malloc` and `free`: the pointer is lost and we cannot deallocate the memory. Worse, freeing the pointer twice, or accessing a freed pointer can lead to exploitable security vulnerabilities.

基于作用域的内存管理

构造函数和析构函数让你可以钩入对象的生命周期。

通过将指针封装在对象中，你可以在该对象 被销毁时释放内存。编译器可保证这一点的实现，即使引发了异常也不例外。

这通常称为“资源获取即初始化 (resource acquisition is initialization, RAII)”， 并为你提供智能指针。

C++ 示例

```
void say_hello(std::unique_ptr<Person> person) {
  std::cout << "Hello " << person->name << std::endl;
}
```

- `std::unique_ptr` 对象在栈上分配内存，并指向在堆上分配的内存。
- 在 `say_hello` 结束时， `std::unique_ptr` 析构函数将运行。
- 析构函数释放它所指向的 `Person` 对象。

将所有权传递给函数时，使用特殊的 move 构造函数：

```
std::unique_ptr<Person> person = find_person("Carla");
say_hello(std::move(person));
```

自动内存管理

自动内存管理是手动和基于作用域的内存管理 的替代方案：

- 程序员从不显式分配或取消分配内存。
- 垃圾回收器找到未使用的内存，并为程序员将其取消分配。

Java 示例

`sayHello` 返回后， `person` 对象未被取消分配：

```
void sayHello(Person person) {
  System.out.println("Hello " + person.getName());
}
```

Rust 中的内存管理

Rust 中的内存管理是一种混合模式：

- 像 Java 一样安全又正确，但没有垃圾回收器。
- 像 C++ 一样基于作用域，但编译器会强制完全遵循规则。
- Rust 用户可以根据具体情况选择合适的抽象，有些甚至没有像 C 那样的运行时开销。

Rust achieves this by modeling *ownership* explicitly.

▼ *Speaker Notes*

- 如果此时被问及如何操作，你可以提及在 Rust 中，这通常由 RAII 封装容器类型（例如 Box、Vec、Rc 或 Arc）处理。这些类型通过各种方式封装了所有权和内存分配，并防止了 C 中潜在错误的发生。
- 你可能会被问及析构函数，此处 Drop trait 是 Rust 等效项。

所有权

所有变量绑定都有一个有效的“作用域”，使用 超出其作用域的变量是错误的：

```
struct Point(i32, i32);

fn main() {
    {
        let p = Point(3, 4);
        println!("x: {}", p.0);
    }
    println!("y: {}", p.1);
}
```

- 作用域结束时，变量会“被丢弃”，数据会被释放。
- 析构函数可在此运行以释放资源。
- 指出变量“拥有”值。

移动语义

An assignment will transfer *ownership* between variables:

```
fn main() {
    let s1: String = String::from("Hello!");
    let s2: String = s1;
    println!("s2: {s2}");
    // println!("s1: {s1}");
}
```

- 将 `s1` 赋值给 `s2`，即转移了所有权。
- When `s1` goes out of scope, nothing happens: it does not own anything.
- 当 `s2` 离开作用域时，字符串数据被释放。
- 变量绑定在任一时刻有且“只有”一个值。

▼ *Speaker Notes*

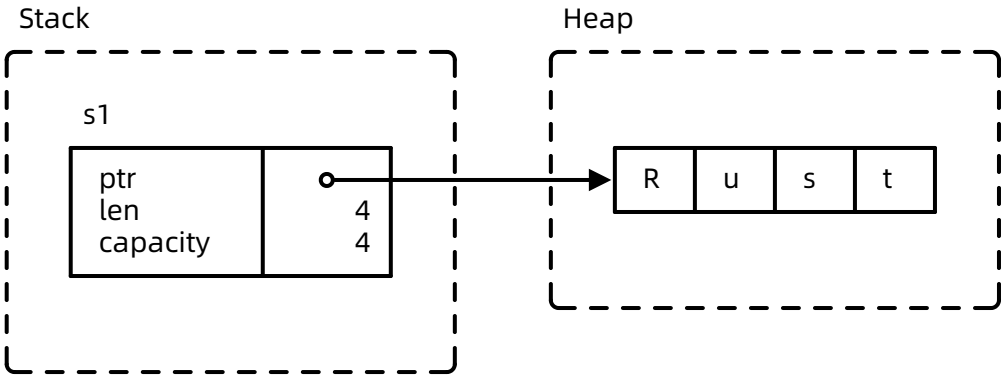
- 指出这与 C++ 中的默认值相反。除非你使用 `std::move`（并已定义 move 构造函数！），否则 C++ 中的默认值是按值复制的。
- 只有所有权发生了转移。是否会生成任何机器码来操控数据本身是一个优化方面的问题，系统会主动优化此类副本。
- 简单的值（例如整数）可以标记为“Copy”（请看后续幻灯片）。
- 在 Rust 中，克隆是显式的（通过使用 `clone`）。

Rust 中移动的字符串

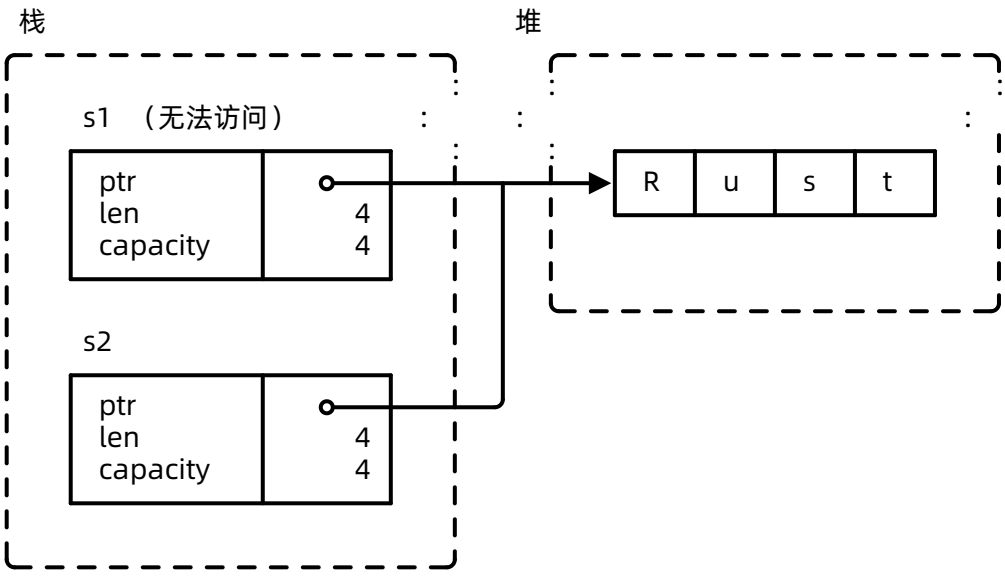
```rust
fn main() {
    let s1: String = String::from("Rust");
    let s2: String = s1;
}
```

- `s1` 中的堆数据会被 `s2` 重复使用。
- 当 `s1` 离开作用域时，什么都不会发生（它已被移出）。

移动到 `s2` 中之前：



移动到 `s2` 中之后：



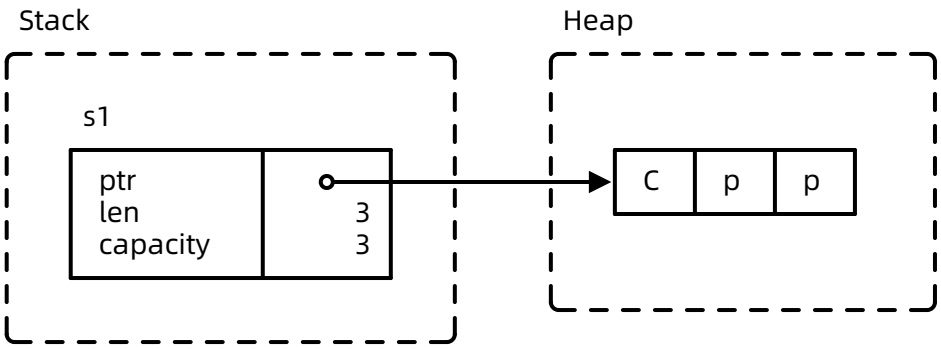
Defensive Copies in Modern C++

现代 C++ 以不同的方式解决此问题：

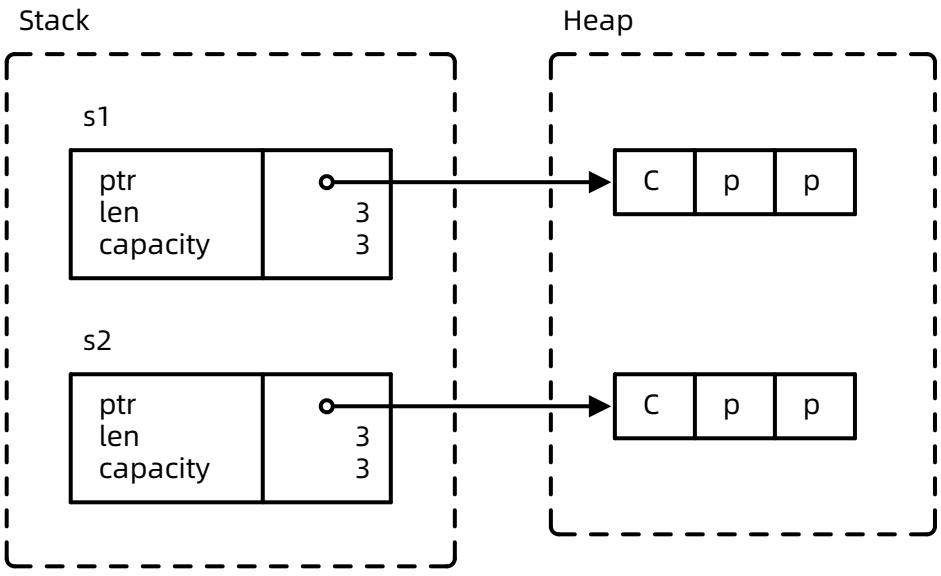
```
std::string s1 = "Cpp";
std::string s2 = s1;  // Duplicate the data in s1.
```

- `s1` 中的堆数据被复制，`s2` 获得自己的独立副本。
- 当 `s1` 和 `s2` 离开作用域时，它们会各自释放自己的内存。

复制-赋值之前：



复制-赋值之后：



▼ *Speaker Notes*

关键点：

- C++ 做出了与 Rust 略有不同的选择。由于“=”会复制数据，因此必须克隆字符串数据。否则，当任一字符串超出范围时，便会出现二次释放。

- C++ 还包含“std::move”，它用于指示何时可以移动某个值。如果示例为“s2 = std::move(s1)”，则不会发生堆分配。移动后，“s1”将处于有效但未指定的状态。与 Rust 不同，程序员可以继续使用“s1”。

- 与 Rust 不同，使用 C++ 时，“=”可以运行任意代码，具体取决于要复制或移动的类型。

函数调用中的移动

你将值传递给函数时，该值会被赋给函数 参数。这就转移了所有权：

```
fn say_hello(name: String) {
    println!("Hello {name}")
}

fn main() {
    let name = String::from("Alice");
    say_hello(name);
    // say_hello(name);
}
```

▼ *Speaker Notes*

- 首次调用 `say_hello` 时， `main` 便放弃了 `name` 的所有权。此后， `main` 中不能再使用 `name` 。
- 在 `say_hello` 函数结束时，系统会释放为 `name` 分配的堆内存。
- 如果 `main` 将 `name` 作为引用（`&name`）传递过去，且 `say_hello` 接受作为参数的引用，则可保留所有权。
- 此外， `main` 也可以在首次调用时传递 `name` 的克隆（`name.clone()`）。
- 相较于 C++，Rust 通过将移动语义设为默认值，并强制程序员进行显式克隆，更难以无意中创建副本。

复制和克隆

虽然移动语义是默认的，但默认情况下会复制某些类型：

```
fn main() {
    let x = 42;
    let y = x;
    println!("x: {x}");
    println!("y: {y}");
}
```

这些类型实现了 `Copy` trait。

你可以选择自己的类型来使用复制语义：

```
#[derive(Copy, Clone, Debug)]
struct Point(i32, i32);

fn main() {
    let p1 = Point(3, 4);
    let p2 = p1;
    println!("p1: {p1:?}");
    println!("p2: {p2:?}");
}
```

- 赋值之后， `p1` 和 `p2` 都拥有自己的数据。
- 我们还可以使用 `p1.clone()` 显式复制数据。

▼ *Speaker Notes*

复制和克隆是两码事：

- 复制是指内存区域的按位复制，不适用于任意对象。
- 复制不允许自定义逻辑（不同于 C++ 中的复制构造函数）。
- 克隆是一种更通用的操作，也允许通过实现 `Clone` trait 来自定义行为。
- 复制不适用于实现 `Drop` trait 的类型。

在上述示例中，请尝试以下操作：

- 在 `struct Point` 中添加 `String` 字段。由于 `String` 不属于 `Copy` 类型，因此无法编译。
- 从 `derive` 属性中移除 `Copy` 。现在，编译器错误位于 `p1` 的 `println!` 中。
- 指出如果你改为克隆 `p1` ，则可按预期运行。

如果学员问起 `derive` ，只需说这是一种 在编译时生成 Rust 代码的方法。在这种情况下，系统会生成 `Copy` 和 `Clone` trait 的默认实现。

借用

调用函数时，你可以让 函数“借用”值，而不是转移所有权：

```
#[derive(Debug)]
struct Point(i32, i32);

fn add(p1: &Point, p2: &Point) -> Point {
    Point(p1.0 + p2.0, p1.1 + p2.1)
}

fn main() {
    let p1 = Point(3, 4);
    let p2 = Point(10, 20);
    let p3 = add(&p1, &p2);
    println!("{p1:?} + {p2:?} = {p3:?}");
}
```

- `add` 函数“借用”两个点并返回一个新点。
- 调用方会保留输入的所有权。

▼ *Speaker Notes*

关于栈返回的说明：

- Demonstrate that the return from `add` is cheap because the compiler can eliminate the copy operation. Change the above code to print stack addresses and run it on the Playground or look at the assembly in Godbolt. In the “DEBUG” optimization level, the addresses should change, while they stay the same when changing to the “RELEASE” setting:

  ```
  #[derive(Debug)]
  struct Point(i32, i32);

  fn add(p1: &Point, p2: &Point) -> Point {
      let p = Point(p1.0 + p2.0, p1.1 + p2.1);
      println!("&p.0: {:p}", &p.0);
      p
  }

  pub fn main() {
      let p1 = Point(3, 4);
      let p2 = Point(10, 20);
      let p3 = add(&p1, &p2);
      println!("&p3.0: {:p}", &p3.0);
      println!("{p1:?} + {p2:?} = {p3:?}");
  }
  ```

- Rust 编译器能够执行返回值优化 (RVO)。

- In C++, copy elision has to be defined in the language specification because constructors can have side effects. In Rust, this is not an issue at all. If RVO did not happen, Rust will always perform a simple and efficient `memcpy` copy.

共享和唯一的借用

Rust 限制了借用值的方式：

- 在任何给定时间，你都可以有一个或多个 `&T` 值，或者
- 你可以有且只有一个 `&mut T` 值。

```
fn main() {
    let mut a: i32 = 10;
    let b: &i32 = &a;

    {
        let c: &mut i32 = &mut a;
        *c = 20;
    }

    println!("a: {a}");
    println!("b: {b}");
}
```

▼ *Speaker Notes*

- 上述代码无法编译，因为 `a` 同时作为可变值（通过 `c` ）和不可变值（通过 `b` ）被借用。
- 将 `b` 的 `println!` 语句移到引入 `c` 的作用域之前，这段代码就可以编译。
- 这样更改后，编译器会发现 `b` 只在通过 `c` 对 `a` 进行新可变借用之前使用过。这是借用检查器的一个功能，名为“非词法作用域生命周期”。

生命周期

借用的值是有“生命周期”的：

- 生命周期可以是隐式的：add(p1: &Point, p2: &Point) -> Point`。
- 生命周期也可以是显式的： `&'a Point` 、 `&'document str` 。
- 将 `&'a Point` 读取为“借用的 `Point，至少 在` a` 生命周期内有效。
- 生命周期始终由编译器推断出来：你不能自行 分配生命周期。
    - 生命周期注释会创建约束条件；编译器会验证 是否存在有效的解决方案。
- Lifetimes for function arguments and return values must be fully specified, but Rust allows lifetimes to be elided in most cases with a few simple rules.

函数调用中的生命周期

除了借用其参数之外，函数还可以返回借用的值：

```
#[derive(Debug)]
struct Point(i32, i32);

fn left_most<'a>(p1: &'a Point, p2: &'a Point) -> &'a Point {
    if p1.0 < p2.0 { p1 } else { p2 }
}

fn main() {
    let p1: Point = Point(10, 10);
    let p2: Point = Point(20, 20);
    let p3: &Point = left_most(&p1, &p2);
    println!("p3: {p3:?}");
}
```

- `'a` 是一个泛型形参，由编译器推断出来。
- 以 `'` 和 `'a` 开头的生命周期是典型的默认名称。
- 将 `&'a Point` 读取为“借用的 `Point，至少 在 a`生命周期内有效。
    - 当参数在不同的作用域时，“至少”部分至关重要。

▼ *Speaker Notes*

在上述示例中，请尝试以下操作：

- 将 `p2` 和 `p3` 的声明移至新作用域（`{ ... }`），以产生以下代码：

```
#[derive(Debug)]
struct Point(i32, i32);

fn left_most<'a>(p1: &'a Point, p2: &'a Point) -> &'a Point {
    if p1.0 < p2.0 { p1 } else { p2 }
}

fn main() {
    let p1: Point = Point(10, 10);
    let p3: &Point;
    {
        let p2: Point = Point(20, 20);
        p3 = left_most(&p1, &p2);
    }
    println!("p3: {p3:?}");
}
```

请注意：由于 `p3` 的生命周期比 `p2` 长，因此无法编译。

- 重置工作区，然后将函数签名更改为 `fn left_most<'a, 'b>(p1: &'a Point, p2: &'a Point) -> &'b Point` 。这不会被编译，因为 `'a` 和 `'b` 生命周期之间的关系不明确。

- 另一种解释方式：

    - 对两个值的两个引用被一个函数借用，该函数返回 另一个引用。
    - 它必须是来自这两个输入中的一个（或来自一个全局变量）。
    - 是哪一个呢？编译器需要知道这一点，因此在调用点，返回的引用 的使用时间不会超过引用的来源中的变量。

数据结构中的生命周期

如果数据类型存储了借用的数据，则必须对其添加生命周期注释：

```
#[derive(Debug)]
struct Highlight<'doc>(&'doc str);

fn erase(text: String) {
    println!("Bye {text}!");
}

fn main() {
    let text = String::from("The quick brown fox jumps over the lazy dog.");
    let fox = Highlight(&text[4..19]);
    let dog = Highlight(&text[35..43]);
    // erase(text);
    println!("{fox:?}");
    println!("{dog:?}");
}
```

▼ *Speaker Notes*

- 在上述示例中， `Highlight` 注释会强制包含 `&str` 的底层数据的生命周期至少与使用该数据的任何 `Highlight` 实例一样长。
- 如果 `text` 在 `fox` （或 `dog` ）的生命周期结束前被消耗，借用检查器将抛出一个错误。
- 借用数据的类型会迫使用户保留原始数据。这对于创建轻量级视图很有用，但通常会使它们更难使用。
- 如有可能，让数据结构直接拥有自己的数据。
- 一些包含多个引用的结构可以有多个生命周期注释。除了结构体本身的生命周期之外，如果需要描述引用之间的生命周期关系，则可能需要这样做。这些都是非常高级的用例。

结构体

与 C 和 C++ 一样，Rust 支持自定义结构体：

```
struct Person {
    name: String,
    age: u8,
}

fn main() {
    let mut peter = Person {
        name: String::from("Peter"),
        age: 27,
    };
    println!("{} is {} years old", peter.name, peter.age);

    peter.age = 28;
    println!("{} is {} years old", peter.name, peter.age);

    let jackie = Person {
        name: String::from("Jackie"),
        ..peter
    };
    println!("{} is {} years old", jackie.name, jackie.age);
}
```

▼ *Speaker Notes*

关键点：

- 结构体的运作方式与使用 C 或 C++ 时类似。
  - 不需要 typedef 即可定义类型，这与使用 C++ 类似，但与使用 C 不同。
  - 与使用 C++ 不同的是，结构体之间没有继承关系。
- 方法是在“impl”块中进行定义的，我们将在后面的幻灯片中看到。
- 这时可能很适合告诉学员存在不同类型的结构体。
  - 针对某类型实现 trait 时，可能会使用大小为零的结构体 `e.g., struct Foo;`，但其中没有任何您要储存在值本身中的数据。
  - 下一张幻灯片将介绍元组结构体，当字段名称不重要时使用。
- 通过语法“..peter”，我们可以从旧结构体复制大部分字段，而无需明确地输入所有字段。它必须始终是最后一个元素。

元组结构体

如果字段名称不重要，您可以使用元组结构体：

```
struct Point(i32, i32);

fn main() {
    let p = Point(17, 23);
    println!("({}, {})", p.0, p.1);
}
```

这通常用于单字段封装容器（称为 newtype）：

```
struct PoundsOfForce(f64);
struct Newtons(f64);

fn compute_thruster_force() -> PoundsOfForce {
    todo!("Ask a rocket scientist at NASA")
}

fn set_thruster_force(force: Newtons) {
    // ...
}

fn main() {
    let force = compute_thruster_force();
    set_thruster_force(force);
}

```

▼ *Speaker Notes*

- 如需对基元类型中的值的额外信息进行编码，使用 newtype 是一种非常好的方式，例如：
  - 数字会以某些单位来衡量：上方示例中为 `Newtons` 。
  - 值在创建时已通过一些验证，因此您不再需要在每次使用时都再次验证它：`PhoneNumber(String)` 或 `OddNumber(u32)` 。
- 展示如何通过访问 newtype 中的单个字段，将 `f64` 值添加到 `Newtons` 类型。
  - Rust 通常不喜欢不明确的内容，例如自动解封或将布尔值用作整数。
  - 运算符过载在第 3 天（泛型）讨论。
- 此示例巧妙地引用了火星气候探测者号 的失败事故。

字段简写语法

如果您已有名称正确的变量，则可以使用简写形式创建结构体：

```
#[derive(Debug)]
struct Person {
    name: String,
    age: u8,
}

impl Person {
    fn new(name: String, age: u8) -> Person {
        Person { name, age }
    }
}

fn main() {
    let peter = Person::new(String::from("Peter"), 27);
    println!("{peter:?}");
}
```

▼ *Speaker Notes*

- 在编写“new”函数时可以使用“Self”作为类型，因为它可以与结构体类型名称互换

```
#[derive(Debug)]
struct Person {
    name: String,
    age: u8,
}
impl Person {
    fn new(name: String, age: u8) -> Self {
        Self { name, age }
    }
}
```

- 为结构体实现“Default”trait。定义一些字段并对其他字段使用默认值。

```rust
#[derive(Debug)]
struct Person {
    name: String,
    age: u8,
}
impl Default for Person {
    fn default() -> Person {
        Person {
            name: "Bot".to_string(),
            age: 0,
        }
    }
}
fn create_default() {
    let tmp = Person {
        ..Person::default()
    };
    let tmp = Person {
        name: "Sam".to_string(),
        ..Person::default()
    };
}
```

- 方法是在“impl”块中进行定义的。
- 使用结构体更新语法以利用“peter”定义一个新结构。请注意，之后将无法再访问变量“peter”。
- 在输出结构体时，使用“{:#?}”来请求“Debug”表示法。

方法

Rust 允许您将函数与新类型相关联。您可以使用“impl”块来执行此操作：

```
#[derive(Debug)]
struct Person {
    name: String,
    age: u8,
}

impl Person {
    fn say_hello(&self) {
        println!("Hello, my name is {}", self.name);
    }
}

fn main() {
    let peter = Person {
        name: String::from("Peter"),
        age: 27,
    };
    peter.say_hello();
}
```

▼ *Speaker Notes*

关键点：

- 引入方法时，将方法与函数进行比较会很有帮助。
  - 在某种类型（例如结构体或枚举）的实例上调用方法，第一个参数将该实例表示为“self”。
  - 开发者可能会选择使用方法，以便利用方法接收器语法并让方法更有条理。通过使用方法，我们可以将所有实现代码保存在一个可预测的位置。
- 指出关键字“self”的用法，它是一种方法接收器。
  - 显示它是“self: Self”的缩写术语，或许要显示结构体名称的可能用法。
  - 说明“Self”是“impl”块所属类型的类型别名，可以在块中的其他位置使用。
  - 指出“self”的使用方式与其他结构体一样，并且可以使用点表示法来指代各个字段。
  - 这可能是演示“&self”和“self”差别的好时机，您只要修改代码并尝试执行 say_hello 两次即可。
- 下面，我们将介绍方法接收器之间的区别。

方法接收者

上面的“&self”表明该方法以不可变的方式借用了对象。还有其他可能的方法接收器：

- “&self”：使用不可变的共享引用从调用方借用对象。之后可以再次使用该对象。
- “&mut self”：使用唯一的可变引用从调用方借用对象。之后可以再次使用该对象。
- “self”：获取对象的所有权并将其从调用方移出。该方法会成为对象的所有者。除非明确转移对象的所有权，否则在该方法返回时，对象将被丢弃（取消分配）。具备完全所有权，不自动等同于具备可变性。
- `mut self`：同上，但该方法可以改变对象。
- 无接收器：这将变为结构体上的静态方法。通常用于创建构造函数，按惯例被称为“new”。

Beyond variants on `self`, there are also special wrapper types allowed to be receiver types, such as `Box<Self>`.

▼ *Speaker Notes*

建议强调“共享且不可变”和“唯一且可变”。由于借用检查器规则的原因，这些约束在 Rust 中总是一起出现，而“self”也不例外。您无法从多个位置引用结构体并对其调用一项改变（“&mut self”）方法。

示例

```rust
#[derive(Debug)]
struct Race {
    name: String,
    laps: Vec<i32>,
}

impl Race {
    fn new(name: &str) -> Race {  // No receiver, a static method
        Race { name: String::from(name), laps: Vec::new() }
    }

    fn add_lap(&mut self, lap: i32) {  // Exclusive borrowed read-write acce
        self.laps.push(lap);
    }

    fn print_laps(&self) {  // Shared and read-only borrowed access to self
        println!("Recorded {} laps for {}:", self.laps.len(), self.name);
        for (idx, lap) in self.laps.iter().enumerate() {
            println!("Lap {idx}: {lap} sec");
        }
    }

    fn finish(self) {  // Exclusive ownership of self
        let total = self.laps.iter().sum::<i32>();
        println!("Race {} is finished, total lap time: {}", self.name, total
    }
}

fn main() {
    let mut race = Race::new("Monaco Grand Prix");
    race.add_lap(70);
    race.add_lap(68);
    race.print_laps();
    race.add_lap(71);
    race.print_laps();
    race.finish();
    // race.add_lap(42);
}
```

▼ *Speaker Notes*

关键点：

- 这里的所有四种方法都使用一个不同的方法接收器。
  - 您可以指出这会如何改变函数可对变量值采取的操作，以及是否/如何能够在“main”中再次使用该函数。
  - 您可以展示在尝试调用“finish”两次时出现的错误。
- 请注意，尽管方法接收器不同，但是非静态函数在 main 函数体中的调用方式相同。Rust 支持在调用方法时自动引用和解引用，并会自动加入“&”“*”和“muts”以便该对象与方法签名匹

配。

- 您或许可以指出 `print_laps` 使用的是不断迭代的vector。我们将在下午详细说明这些vector。

第二天上午习题

我们将考虑以下两种场景：

- 存储图书和查询馆藏
- 跟踪患者的健康统计信息

▼ *Speaker Notes*

After looking at the exercises, you can look at the solutions provided.

书籍存储

We will learn much more about structs and the `Vec<T>` type tomorrow. For now, you just need to know part of its API:

```
fn main() {
    let mut vec = vec![10, 20];
    vec.push(30);
    let midpoint = vec.len() / 2;
    println!("middle value: {}", vec[midpoint]);
    for item in &vec {
        println!("item: {item}");
    }
}
```

Use this to model a library's book collection. Copy the code below to https://play.rust-lang.org/ and update the types to make it compile:

```rust
struct Library {
    books: Vec<Book>,
}

struct Book {
    title: String,
    year: u16,
}

impl Book {
    // This is a constructor, used below.
    fn new(title: &str, year: u16) -> Book {
        Book {
            title: String::from(title),
            year,
        }
    }
}

// Implement the methods below. Notice how the `self` parameter
// changes type to indicate the method's required level of ownership
// over the object:
//
// - `&self` for shared read-only access,
// - `&mut self` for unique and mutable access,
// - `self` for unique access by value.
impl Library {
    fn new() -> Library {
        todo!("Initialize and return a `Library` value")
    }

    fn len(&self) -> usize {
        todo!("Return the length of `self.books`")
    }

    fn is_empty(&self) -> bool {
        todo!("Return `true` if `self.books` is empty")
    }

    fn add_book(&mut self, book: Book) {
        todo!("Add a new book to `self.books`")
    }

    fn print_books(&self) {
        todo!("Iterate over `self.books` and print each book's title and year")
    }

    fn oldest_book(&self) -> Option<&Book> {
        todo!("Return a reference to the oldest book (if any)")
    }
}

fn main() {
    let mut library = Library::new();

    println!(
        "The library is empty: library.is_empty() -> {}",
```

```
        library.is_empty()
    );

    library.add_book(Book::new("Lord of the Rings", 1954));
    library.add_book(Book::new("Alice's Adventures in Wonderland", 1865));

    println!(
        "The library is no longer empty: library.is_empty() -> {}",
        library.is_empty()
    );

    library.print_books();

    match library.oldest_book() {
        Some(book) => println!("The oldest book is {}", book.title),
        None => println!("The library is empty!"),
    }

    println!("The library has {} books", library.len());
    library.print_books();
}
```

健康统计

你正在实现一个健康监控系统。作为其中的一部分，你需要对用户的健康统计数据进行追踪。

`User` 结构体的定义和 `impl` 块中一些函数的框架已经给出。你的目标是实现在 `impl` 块中定义的 `User struct` 的方法。

将以下代码复制到 https://play.rust-lang.org/，并填充缺失的方法：

```rust
// TODO: remove this when you're done with your implementation.
#![allow(unused_variables, dead_code)]

pub struct User {
    name: String,
    age: u32,
    height: f32,
    visit_count: usize,
    last_blood_pressure: Option<(u32, u32)>,
}

pub struct Measurements {
    height: f32,
    blood_pressure: (u32, u32),
}

pub struct HealthReport<'a> {
    patient_name: &'a str,
    visit_count: u32,
    height_change: f32,
    blood_pressure_change: Option<(i32, i32)>,
}

impl User {
    pub fn new(name: String, age: u32, height: f32) -> Self {
        todo!("Create a new User instance")
    }

    pub fn name(&self) -> &str {
        todo!("Return the user's name")
    }

    pub fn age(&self) -> u32 {
        todo!("Return the user's age")
    }

    pub fn height(&self) -> f32 {
        todo!("Return the user's height")
    }

    pub fn doctor_visits(&self) -> u32 {
        todo!("Return the number of time the user has visited the doctor")
    }

    pub fn set_age(&mut self, new_age: u32) {
        todo!("Set the user's age")
    }

    pub fn set_height(&mut self, new_height: f32) {
        todo!("Set the user's height")
    }

    pub fn visit_doctor(&mut self, measurements: Measurements) -> HealthReport
{
        todo!("Update a user's statistics based on measurements from a visit to
the doctor")
    }
```

```
}

fn main() {
    let bob = User::new(String::from("Bob"), 32, 155.2);
    println!("I'm {} and my age is {}", bob.name(), bob.age());
}

#[test]
fn test_height() {
    let bob = User::new(String::from("Bob"), 32, 155.2);
    assert_eq!(bob.height(), 155.2);
}

#[test]
fn test_set_age() {
    let mut bob = User::new(String::from("Bob"), 32, 155.2);
    assert_eq!(bob.age(), 32);
    bob.set_age(33);
    assert_eq!(bob.age(), 33);
}

#[test]
fn test_visit() {
    let mut bob = User::new(String::from("Bob"), 32, 155.2);
    assert_eq!(bob.doctor_visits(), 0);
    let report = bob.visit_doctor(Measurements {
        height: 156.1,
        blood_pressure: (120, 80),
    });
    assert_eq!(report.patient_name, "Bob");
    assert_eq!(report.visit_count, 1);
    assert_eq!(report.blood_pressure_change, None);

    let report = bob.visit_doctor(Measurements {
        height: 156.1,
        blood_pressure: (115, 76),
    });

    assert_eq!(report.visit_count, 2);
    assert_eq!(report.blood_pressure_change, Some((-5, -4)));
}
```

标准库

Rust 附带一个标准库，此库有助于建立一个供 Rust 库和程序 使用的常用类型集。这样一来，两个库便可顺畅地搭配运作， 因为它们使用相同的 `String` 类型。

常见的词汇类型包括：

- `Option` 和 `Result` 类型：用于可选值和 错误处理。
- `String`： 用于自有数据的默认字符串类型。
- `Vec`： 标准的可扩展矢量。
- `HashMap`： 采用可配置哈希算法的哈希映射 类型。
- `Box`： 适用于堆分配数据的自有指针。
- `Rc`： 适用于堆分配数据的共享引用计数指针。

▼ *Speaker Notes*

- Rust 实际上含有多个层级的标准库，分别是 `core` 、 `alloc` 和 `std` 。
- `core` 包括最基本的类型与函数，这些类型与函数不依赖于 `libc` 、分配器 或是否存在操作系统。
- `alloc` 包括需要全局堆分配器的类型，例如 `Vec` 、 `Box` 和 `Arc` 。
- 嵌入式 Rust 应用通常只使用 `core` ，偶尔会使用 `alloc` 。

Option 和 Result

这些类型表示可选数据：

```
fn main() {
    let numbers = vec![10, 20, 30];
    let first: Option<&i8> = numbers.first();
    println!("first: {first:?}");

    let arr: Result<[i8; 3], Vec<i8>> = numbers.try_into();
    println!("arr: {arr:?}");
}
```

▼ *Speaker Notes*

- `Option` 和 `Result` 的使用范围很广，不局限于标准库。
- 相较于 `&T`， `Option<&T>` 的空间开销为零。
- `Result` 是用于实现错误处理的标准类型，我们将在第 3 天的课程中介绍。
- `try_into` attempts to convert the vector into a fixed-sized array. This can fail:
  - If the vector has the right size, `Result::Ok` is returned with the array.
  - Otherwise, `Result::Err` is returned with the original vector.

String

`String` 是标准堆分配的可扩容 UTF-8 字符串缓冲区：

```
fn main() {
    let mut s1 = String::new();
    s1.push_str("Hello");
    println!("s1: len = {}, capacity = {}", s1.len(), s1.capacity());

    let mut s2 = String::with_capacity(s1.len() + 1);
    s2.push_str(&s1);
    s2.push('!');
    println!("s2: len = {}, capacity = {}", s2.len(), s2.capacity());

    let s3 = String::from(" c н ");
    println!("s3: len = {}, number of chars = {}", s3.len(),
             s3.chars().count());
}
```

`String` 会实现 `Deref<Target = str>`，这意味着您可以 对 `String` 调用所有 `str` 方法。

▼ *Speaker Notes*

- “String::new”会返回一个新的空字符串，如果您知道自己想要推送到字符串的数据量，请使用“String::with_capacity”。
- “String::len”会返回“String”的大小（以字节为单位，可能不同于以字符为单位的长度）。
- “String::chars”会针对实际字符返回一个迭代器。请注意，由于字素簇，“char”可能与人们所认为的“字符”有所不同。
- 当人们提到字符串时，可能是指“&str”或“String”。
- 当某个类型实现“Deref<Target = T>”时，编译器会让您以公开透明方式从“T”调用方法。
  - “String”会实现“Deref<Target = str>”，后者可公开透明地授予其访问“str”方法的权限。
  - 写下并比较“let s3 = s1.deref();”和“let s3 = &*s1;”。
- “String”是作为字节矢量的封装容器实现的，矢量上支持的许多操作在“String”上也受支持，但有一些额外保证。
- 比较将“String”编入索引的不同方式：
  - 使用“s3.chars().nth(i).unwrap()”转换为字符，其中“i”代表是否出界。
  - 通过使用“s3[0..4]”转换为子字符串，其中该 Slice 在或不在字符边界上。

Vec

`Vec` 是标准的可调整大小堆分配缓冲区：

```
fn main() {
    let mut v1 = Vec::new();
    v1.push(42);
    println!("v1: len = {}, capacity = {}", v1.len(), v1.capacity());

    let mut v2 = Vec::with_capacity(v1.len() + 1);
    v2.extend(v1.iter());
    v2.push(9999);
    println!("v2: len = {}, capacity = {}", v2.len(), v2.capacity());

    // Canonical macro to initialize a vector with elements.
    let mut v3 = vec![0, 0, 1, 2, 3, 4];

    // Retain only the even elements.
    v3.retain(|x| x % 2 == 0);
    println!("{v3:?}");

    // Remove consecutive duplicates.
    v3.dedup();
    println!("{v3:?}");
}
```

`Vec` 会实现 `Deref<Target = [T]>`，这意味着您可以对 `Vec` 调用 slice 方法。

▼ *Speaker Notes*

- “Vec”以及“String”和“HashMap”都是一种集合。它包含的数据会存储在堆上。这意味着在编译时不需要知道数据量。它可以在运行时增大或缩小。
- Notice how `Vec<T>` is a generic type too, but you don't have to specify `T` explicitly. As always with Rust type inference, the `T` was established during the first `push` call.
- “vec![...]”是用来代替“Vec::new()”的规范化宏，它支持向矢量添加初始元素。
- 如需将矢量编入索引，您可以使用“[' ']”方法，但如果超出边界，矢量将会 panic。此外，使用“get”将返回“Option”。“pop”函数会移除最后一个元素。
- 介绍如何迭代矢量并更改它的值：“for e in &mut v { *e += 50; }”

HashMap

标准的哈希映射，内含针对 HashDoS 攻击的保护措施：

```
use std::collections::HashMap;

fn main() {
    let mut page_counts = HashMap::new();
    page_counts.insert("Adventures of Huckleberry Finn".to_string(), 207);
    page_counts.insert("Grimms' Fairy Tales".to_string(), 751);
    page_counts.insert("Pride and Prejudice".to_string(), 303);

    if !page_counts.contains_key("Les Misérables") {
        println!("We know about {} books, but not Les Misérables.",
                 page_counts.len());
    }

    for book in ["Pride and Prejudice", "Alice's Adventure in Wonderland"] {
        match page_counts.get(book) {
            Some(count) => println!("{book}: {count} pages"),
            None => println!("{book} is unknown.")
        }
    }

    // Use the .entry() method to insert a value if nothing is found.
    for book in ["Pride and Prejudice", "Alice's Adventure in Wonderland"] {
        let page_count: &mut i32 = page_counts.entry(book.to_string()).or_ir
        *page_count += 1;
    }

    println!("{page_counts:#?}");
}
```

▼ *Speaker Notes*

- “HashMap”未在序言中定义，因此需要纳入范围中。
- 请尝试使用以下代码行。第一行将查看图书是否在 hashmap 中；如果不在，则返回替代值。如果未找到图书，第二行会在 hashmap 中插入替代值。

```
let pc1 = page_counts
    .get("Harry Potter and the Sorcerer's Stone ")
    .unwrap_or(&336);
let pc2 = page_counts
    .entry("The Hunger Games".to_string())
    .or_insert(374);
```

- 遗憾的是，与“vec!”不同，不存在标准的“hashmap!”宏。

    - 不过，从 Rust 1.56 开始，HashMap 实现了“From<[(K, V); N]>”，让我们能够轻松地从字面量数组初始化哈希映射：

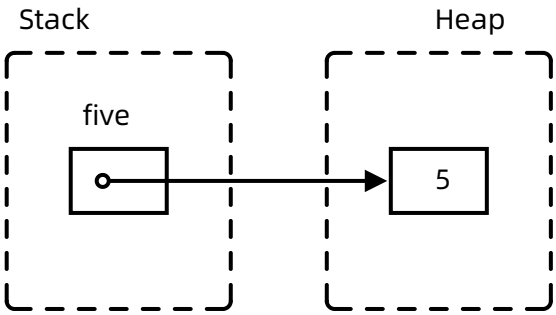
```
let page_counts = HashMap::from([
  ("Harry Potter and the Sorcerer's Stone".to_string(), 336),
  ("The Hunger Games".to_string(), 374),
]);
```

- 或者，HashMap 也可以基于任何可生成键-值元组的“Iterator”进行构建。
- 我们要展示“HashMap<String, i32>”，避免将“&str”用作键，以便简化示例。当然，可以在集合中使用引用，但可能会导致借用检查器出现复杂问题。
    - 尝试从上述示例中移除“to_string()”，看看它是否仍可编译。您认为我们可能会在哪些方面遇到问题？
- 此类型具有几种特定于方法的返回值类型，例如“std::collections::hash_map::Keys”。这些类型通常会出现在 Rust 文档的搜索结果中。向学员展示此类型的文档，以及指向“keys”方法的实用链接。

Box

`Box` 是指向堆上数据的自有指针：

```
1  fn main() {
2      let five = Box::new(5);
3      println!("five: {}", *five);
4  }
```



`Box<T>` 会实现 `Deref<Target = T>`，这意味着您可以直接在 `Box<T>` 上通过 `T` 调用相应方法。

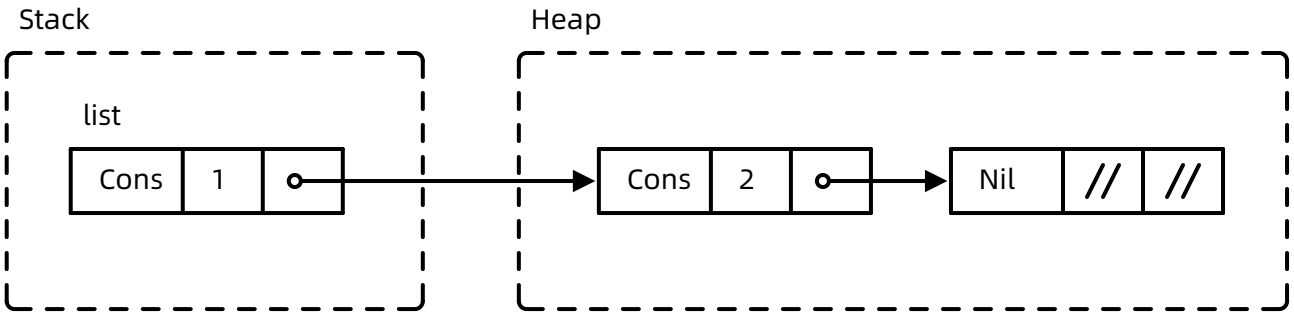
▼ *Speaker Notes*

- 在 C++ 中， `Box` 与 `std::unique_ptr` 类似，除了它一定会不为 null 以外。
- 在上面的示例中，因为有 `Deref`，您甚至可以在 `println!` 语句中省略 `*`。
- 在以下情况下， `Box` 可能会很实用：
  - 在编译时间遇到无法知晓大小的类型，但 Rust 编译器需要知道确切大小。
  - 想要转让大量数据的所有权。为避免在堆栈上复制大量数据，请改为将数据存储在 `Box` 中的堆上，以便仅移动指针。

包含递归数据结构的 Box

递归数据类型或具有动态大小的数据类型需要使用 `Box`：

```
#[derive(Debug)]
enum List<T> {
    Cons(T, Box<List<T>>),
    Nil,
}

fn main() {
    let list: List<i32> = List::Cons(1, Box::new(List::Cons(2, Box::new(List
    println!("{list:?}");
}
```



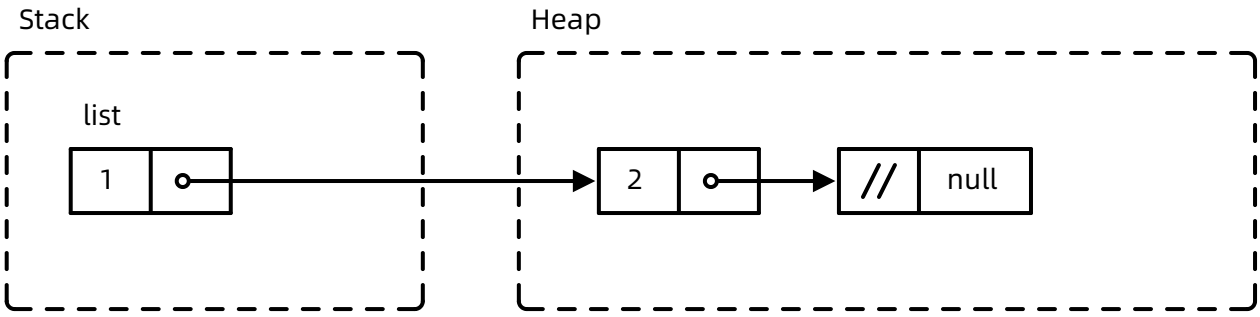
▼ *Speaker Notes*

- If `Box` was not used and we attempted to embed a `List` directly into the `List`, the compiler would not compute a fixed size of the struct in memory (`List` would be of infinite size).
- `Box` 大小与一般指针相同，并且只会指向堆中的下一个 `List` 元素， 因此可以解决这个问题。
- 将 `Box` 从 List 定义中移除后，画面上会显示编译器错误。如果您看到“Recursive with indirection”错误消息，这是在提示您使用 Box 或其他类型的引用，而不是直接储存值。

小众优化

```
#[derive(Debug)]
enum List<T> {
    Cons(T, Box<List<T>>),
    Nil,
}

fn main() {
    let list: List<i32> = List::Cons(1, Box::new(List::Cons(2, Box::new(List
    println!("{list:?}");
}
```

`Box` 不得为空，因此指针始终有效且非 `null` 。这样， 编译器就可以优化内存布局：



Rc

`Rc` 是引用计数的共享指针。如果您需要从多个位置 引用相同的数据，请使用此指针：

```
use std::rc::Rc;

fn main() {
    let mut a = Rc::new(10);
    let mut b = Rc::clone(&a);

    println!("a: {a}");
    println!("b: {b}");
}
```

- See `Arc` and `Mutex` if you are in a multi-threaded context.
- 您可以将共享指针_降级_为 `Weak` 指针， 以便创建之后会被舍弃的循环引用。

▼ *Speaker Notes*

- `Rc` 的计数可确保只要有引用，内含的值就会保持有效。
- Rust 中的“Rc”与 C++ 中的“std::shared_ptr”类似。
- `Rc::clone` 的成本很低：这个做法会创建指向相同分配的指针，并增加引用计数，而不会产生深层的克隆，排查代码性能问题时通常可以忽略。
- `make_mut` 实际上会在必要时克隆内部值（“clone-on-write”），并返回可变的引用。
- 使用 `Rc::strong_count` 可查看引用计数。
- `Rc::downgrade` gives you a *weakly reference-counted* object to create cycles that will be dropped properly (likely in combination with `RefCell`, on the next slide).

“Cell”和“RefCell”

`Cell` and `RefCell` implement what Rust calls *interior mutability:* mutation of values in an immutable context.

“Cell”通常用于简单类型，因为它需要复制或移动值。更复杂的内部类型通常使用“RefCell”，它会在运行时跟踪已共享和专有的引用，并在这些引用被滥用时 panic。

```
use std::cell::RefCell;
use std::rc::Rc;

#[derive(Debug, Default)]
struct Node {
    value: i64,
    children: Vec<Rc<RefCell<Node>>>,
}

impl Node {
    fn new(value: i64) -> Rc<RefCell<Node>> {
        Rc::new(RefCell::new(Node { value, ..Node::default() }))
    }

    fn sum(&self) -> i64 {
        self.value + self.children.iter().map(|c| c.borrow().sum()).sum::<i6
    }
}

fn main() {
    let root = Node::new(1);
    root.borrow_mut().children.push(Node::new(5));
    let subtree = Node::new(10);
    subtree.borrow_mut().children.push(Node::new(11));
    subtree.borrow_mut().children.push(Node::new(12));
    root.borrow_mut().children.push(subtree);

    println!("graph: {root:#?}");
    println!("graph sum: {}", root.borrow().sum());
}
```

▼ *Speaker Notes*

- 在此示例中，如果我们使用的是“Cell”而非“RefCell”，则必须将“Node”从“Rc”中移出以推送子项，然后再将其移回原位。这是安全的做法，因为单元格中总是有一个未引用的值，但这不符合人体工程学。
- 如需使用 Node 执行任何操作，您必须调用“RefCell”方法，通常为“borrow”或“borrow_mut”。
- 演示可以通过向“subtree.children”添加“root”来创建引用循环（不要尝试输出它！）。

- 为了演示运行时 panic，请添加一个会递增“self.value”并以相同方法调用其子项的“fn inc(&mut self)”。如果存在引用循环，就会 panic，并且“thread”“main”会因“already borrowed: BorrowMutError”而 panic。

模块

我们已看了“impl”块如何让我们将函数的命名空间建为一种类型。

同样，“mod”让我们可为类型和函数建立命名空间：

```
mod foo {
    pub fn do_something() {
        println!("In the foo module");
    }
}

mod bar {
    pub fn do_something() {
        println!("In the bar module");
    }
}

fn main() {
    foo::do_something();
    bar::do_something();
}
```

▼ *Speaker Notes*

- 包提供功能，并包含一个描述如何构建包含 1 个以上 crate 的捆绑包的“Cargo.toml”文件。
- crate 是一种模块树，其中的二进制 crate 会创建一个可执行文件，而库 crate 会编译为库。
- 模块定义了组织和范围，并且是本部分的重点。

可见性

模块是一种隐私边界：

- 默认情况下，模块项是私有的（隐藏实现详情）。
- 父项和同级子项始终可见。
- 换言之，如果某个项在模块“foo”中可见，那么该项在“foo”的所有后代中均可见。

```
mod outer {
    fn private() {
        println!("outer::private");
    }

    pub fn public() {
        println!("outer::public");
    }

    mod inner {
        fn private() {
            println!("outer::inner::private");
        }

        pub fn public() {
            println!("outer::inner::public");
            super::private();
        }
    }
}

fn main() {
    outer::public();
}
```

▼ *Speaker Notes*

- 使用“pub”关键字将模块设为公开。

此外，您还可以使用高级“pub(...)”说明符来限制公开可见的范围。

- 请参阅 Rust 参考。
- 配置“pub(crate)”可见性是一种常见模式。
- 您可以为特定路径授予可见性，这种情况不太常见。
- 在任何情况下，都必须向祖先模块（及其所有后代）授予可见性。

路径

路径解析如下：

1. 作为相对路径：
    - `foo` 或 `self::foo` 是指当前模块中的 `foo`，
    - “super::foo”是指父模块中的“foo”。
2. 作为绝对路径：
    - `crate::foo` 是指当前 crate 的根中的 `foo`，
    - “bar::foo”是指“bar”crate 中的“foo”。

一个模块可以使用“use”将另一个模块的符号全部纳入。您通常在每个模块的顶部会看到如下内容：

```
use std::collections::HashSet;
use std::mem::transmute;
```

文件系统层级结构

如果省略模块内容，则会指示 Rust 在另一个文件中查找：

```
mod garden;
```

这会告知 Rust 可以在“src/garden.rs”中找到“garden”模块内容。同样，您可以在“src/garden/vegetables.rs”中找到“garden::vegetables”模块。

“crate”根目录位于：

- “src/lib.rs”（对于库 crate）
- “src/main.rs”（对于二进制文件 crate）

也可以使用“内部文档注释”对文件中定义的模块进行记录。这些用于记录包含它们的项（在本例中为模块）。

```
//! This module implements the garden, including a highly performant germina
//! implementation.

// Re-export types from this module.
pub use seeds::SeedPacket;
pub use garden::Garden;

/// Sow the given seed packets.
pub fn sow(seeds: Vec<SeedPacket>) { todo!() }

/// Harvest the produce in the garden that is ready.
pub fn harvest(garden: &mut Garden) { todo!() }
```

▼ *Speaker Notes*

- 在 Rust 2018 之前的版本中，模块需要位于“module/mod.rs”而非“module.rs”中，对于 2018 年之后的版本而言，这仍是有效的替代方案。
- 引入“filename.rs”来替代“filename/mod.rs”的主要原因是，许多名为“mod.rs”的文件在 IDE 中可能难以区分。
- 即使主模块是文件，更深层的嵌套也可以使用文件夹：

```
src/
├── main.rs
├── top_module.rs
└── top_module/
    └── sub_module.rs
```

- Rust 寻找模块的位置可通过编译器指令更改：

```rust
#[path = "some/path.rs"]
mod some_module;
```

例如，如果您想将某个模块的测试放在名为“some_module_test.rs”的文件中（类似于 Go 中的惯例），这样做很有用。

第二天下午习题

今天下午的习题将重点关注字符串（string）和迭代器（iterator）。

▼ *Speaker Notes*

After looking at the exercises, you can look at the solutions provided.

迭代器和所有权

Rust 的所有权模式会影响许多 API。例如，“Iterator”和“IntoIterator” trait。

“Iterator”

trait 类似于接口：它们描述某类型的行为（方法）。 `Iterator` trait 只是告知您可以调用 `next`，直到返回 `None`：

```
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

您可以按如下方式使用此 trait：

```
fn main() {
    let v: Vec<i8> = vec![10, 20, 30];
    let mut iter = v.iter();

    println!("v[0]: {:?}", iter.next());
    println!("v[1]: {:?}", iter.next());
    println!("v[2]: {:?}", iter.next());
    println!("No more items: {:?}", iter.next());
}
```

迭代器返回的类型是什么？请在此测试您的答案：

```
fn main() {
    let v: Vec<i8> = vec![10, 20, 30];
    let mut iter = v.iter();

    let v0: Option<..> = iter.next();
    println!("v0: {v0:?}");
}
```

为什么要使用此类型？

“IntoIterator”

“Iterator”trait会告知您在创建迭代器后如何进行迭代。相关 trait“IntoIterator”会告知您如何创建迭代器：

```
pub trait IntoIterator {
    type Item;
    type IntoIter: Iterator<Item = Self::Item>;

    fn into_iter(self) -> Self::IntoIter;
}
```

这里的语法表示，“IntoIterator”的每个实现都必须声明两种类型：

- `Item`：我们迭代的类型，例如 `i8`，
- “IntoIter”：“into_iter”方法返回的“Iterator”类型。

Note that `IntoIter` and `Item` are linked: the iterator must have the same `Item` type, which means that it returns `Option<Item>`

和之前一样，迭代器返回的类型是什么？

```
fn main() {
    let v: Vec<String> = vec![String::from("foo"), String::from("bar")];
    let mut iter = v.into_iter();

    let v0: Option<..> = iter.next();
    println!("v0: {v0:?}");
}
```

“for”循环

现在，我们已了解了“Iterator”和“IntoIterator”，接下来可以构建“for”循环了。它们会针对表达式调用“into_iter()”，并对生成的迭代器进行迭代：

```
fn main() {
    let v: Vec<String> = vec![String::from("foo"), String::from("bar")];

    for word in &v {
        println!("word: {word}");
    }

    for word in v {
        println!("word: {word}");
    }
}
```

每个循环中的“word”是什么类型？

Experiment with the code above and then consult the documentation for `impl IntoIterator for &Vec<T>` and `impl IntoIterator for Vec<T>` to check your answers.

字符串和迭代器

在本练习中，您将实现 Web 服务器的路由组件。服务器配置有多个路径前缀，这些前缀与请求路径匹配。路径前缀可以包含与完整段匹配的通配符。请参阅下面的单元测试。

将以下代码复制到 https://play.rust-lang.org/，然后设法通过测试。请尽量避免为中间结果分配“Vec”：

```
// TODO: remove this when you're done with your implementation.
#![allow(unused_variables, dead_code)]

pub fn prefix_matches(prefix: &str, request_path: &str) -> bool {
    unimplemented!()
}

#[test]
fn test_matches_without_wildcard() {
    assert!(prefix_matches("/v1/publishers", "/v1/publishers"));
    assert!(prefix_matches("/v1/publishers", "/v1/publishers/abc-123"));
    assert!(prefix_matches("/v1/publishers", "/v1/publishers/abc/books"));

    assert!(!prefix_matches("/v1/publishers", "/v1"));
    assert!(!prefix_matches("/v1/publishers", "/v1/publishersBooks"));
    assert!(!prefix_matches("/v1/publishers", "/v1/parent/publishers"));
}

#[test]
fn test_matches_with_wildcard() {
    assert!(prefix_matches(
        "/v1/publishers/*/books",
        "/v1/publishers/foo/books"
    ));
    assert!(prefix_matches(
        "/v1/publishers/*/books",
        "/v1/publishers/bar/books"
    ));
    assert!(prefix_matches(
        "/v1/publishers/*/books",
        "/v1/publishers/foo/books/book1"
    ));

    assert!(!prefix_matches("/v1/publishers/*/books", "/v1/publishers"));
    assert!(!prefix_matches(
        "/v1/publishers/*/books",
        "/v1/publishers/foo/booksByAuthor"
    ));
}
```

欢迎参加第 3 天的课程

今天，我们将介绍一些更高级的 Rust 主题：

- trait：派生 trait、默认方法和重要的标准库 trait。
- 泛型：泛型数据类型、泛型方法、单态化和 trait 对象。
- 错误处理：panic、“Result”和 try 运算符“?”。
- 测试：单元测试、文档测试和集成测试。
- 不安全 Rust：原始指针、静态变量、不安全函数和外部函数。

泛型

Rust support generics, which lets you abstract algorithms or data structures (such as sorting or a binary tree) over the types used or stored.

通用数据类型

您可以使用泛型对具体字段类型进行抽象化处理：

```
#[derive(Debug)]
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
    println!("{integer:?} and {float:?}");
}
```

▼ *Speaker Notes*

- 尝试声明一个新变量“let p = Point { x: 5, y: 10.0 };”。
- 修正代码，以允许点具有不同类型的元素。

泛型方法

您可以在 `impl` 块中声明通用类型：

```
#[derive(Debug)]
struct Point<T>(T, T);

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.0  // + 10
    }

    // fn set_x(&mut self, x: T)
}

fn main() {
    let p = Point(5, 10);
    println!("p.x = {}", p.x());
}
```

▼ *Speaker Notes*

- *问：*为什么 `T` 在 `impl<T> Point<T> {}` 中指定了两次？这不是多余的吗？
  - 这是因为它是泛型类型的泛型实现部分。它们是独立的泛型内容。
  - 这意味着这些方法是针对所有 `T` 定义的。
  - 可以编写 `impl Point<u32> { .. }`。
    - `Point` 依然是一个泛型，并且您可以使用 `Point<f64>`，但此块中的方法将仅适用于 `Point<u32>`。

单态化

泛型代码根据调用位置转换为非泛型代码：

```
fn main() {
    let integer = Some(5);
    let float = Some(5.0);
}
```

具体行为与您所编写的一样

```
enum Option_i32 {
    Some(i32),
    None,
}

enum Option_f64 {
    Some(f64),
    None,
}

fn main() {
    let integer = Option_i32::Some(5);
    let float = Option_f64::Some(5.0);
}
```

这是零成本的抽象化处理：您得到的结果不会受到影响，也就是说，与在没有进行抽象化处理的情况下，对数据结构进行手动编码时的结果一样。

特征（Trait）

Rust 让您可以依据特征对类型进行抽象化处理。特征与接口类似：

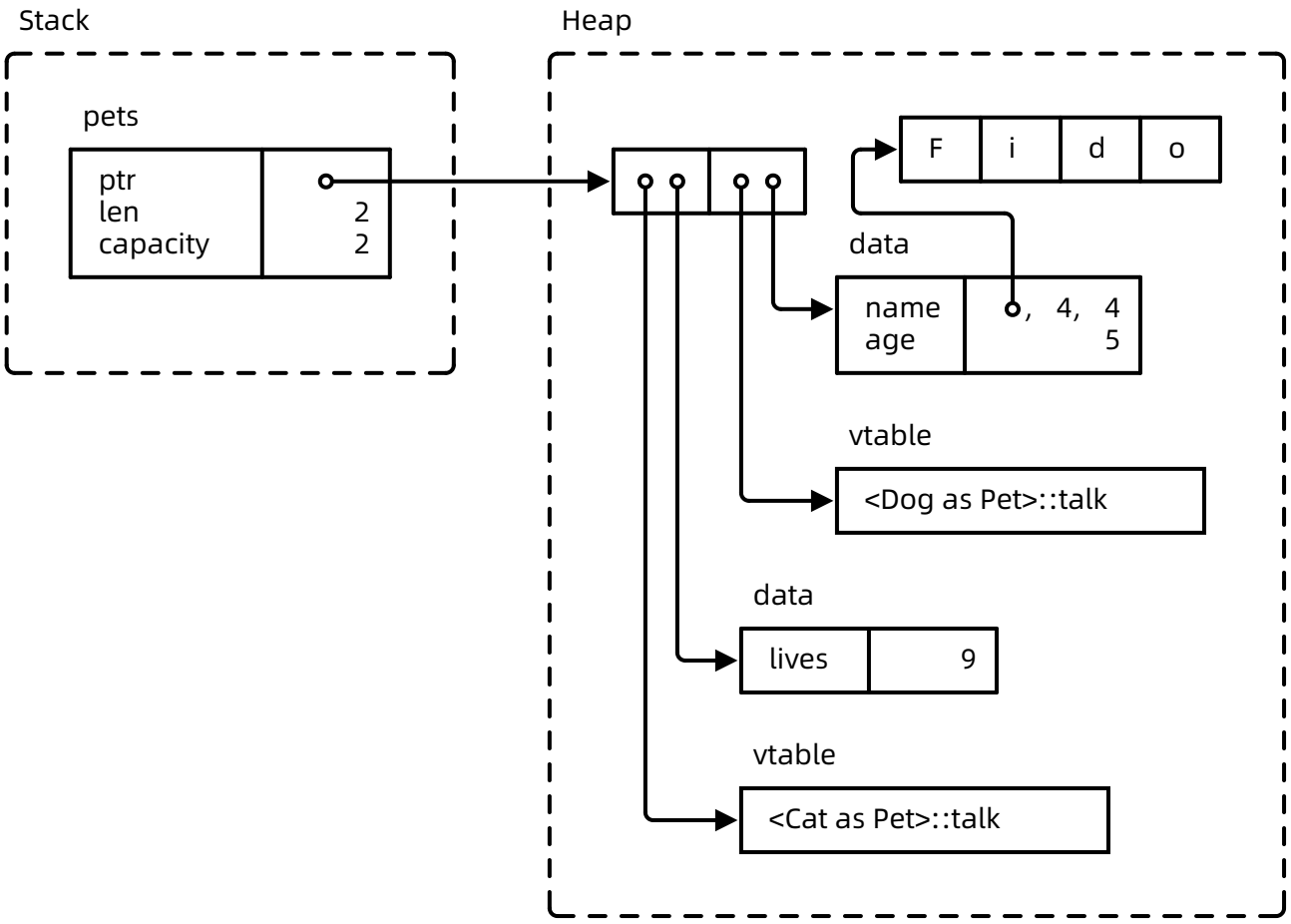
```
struct Dog { name: String, age: i8 }
struct Cat { lives: i8 } // No name needed, cats won't respond anyway.

trait Pet {
    fn talk(&self) -> String;
}

impl Pet for Dog {
    fn talk(&self) -> String { format!("Woof, my name is {}!", self.name) }
}

impl Pet for Cat {
    fn talk(&self) -> String { String::from("Miau!") }
}

fn greet<P: Pet>(pet: &P) {
    println!("Oh you're a cutie! What's your name? {}", pet.talk());
}

fn main() {
    let captain_floof = Cat { lives: 9 };
    let fido = Dog { name: String::from("Fido"), age: 5 };

    greet(&captain_floof);
    greet(&fido);
}
```

特征对象

特征（Trait）对象可接受不同类型的值，举例来说，在集合中会是这样：

```
struct Dog { name: String, age: i8 }
struct Cat { lives: i8 } // No name needed, cats won't respond anyway.

trait Pet {
    fn talk(&self) -> String;
}

impl Pet for Dog {
    fn talk(&self) -> String { format!("Woof, my name is {}!", self.name) }
}

impl Pet for Cat {
    fn talk(&self) -> String { String::from("Miau!") }
}

fn main() {
    let pets: Vec<Box<dyn Pet>> = vec![
        Box::new(Cat { lives: 9 }),
        Box::new(Dog { name: String::from("Fido"), age: 5 }),
    ];
    for pet in pets {
        println!("Hello, who are you? {}", pet.talk());
    }
}
```

以下是分配 `pets` 后的内存布局：



▼ *Speaker Notes*

- Types that implement a given trait may be of different sizes. This makes it impossible to have things like `Vec<dyn Pet>` in the example above.
- 可通过“dyn Pet”这个方法向编译器告知实现“Pet”的动态大小类型。
- In the example, `pets` is allocated on the stack and the vector data is on the heap. The two vector elements are *fat pointers*:
  - A fat pointer is a double-width pointer. It has two components: a pointer to the actual object and a pointer to the virtual method table (vtable) for the `Pet` implementation of that particular object.
  - The data for the `Dog` named Fido is the `name` and `age` fields. The `Cat` has a `lives` field.
- 比较上述示例中的这些输出：

```
    println!("{} {}", std::mem::size_of::<Dog>(), std::mem::size_of::
<Cat>());
    println!("{} {}", std::mem::size_of::<&Dog>(), std::mem::size_of::
<&Cat>());
    println!("{}", std::mem::size_of::<&dyn Pet>());
    println!("{}", std::mem::size_of::<Box<dyn Pet>>());
```

派生特征

Rust 派生宏的运作方式是自动生成代码，用于实现数据结构的指定 trait。

You can let the compiler derive a number of traits as follows:

```
#[derive(Debug, Clone, PartialEq, Eq, Default)]
struct Player {
    name: String,
    strength: u8,
    hit_points: u8,
}

fn main() {
    let p1 = Player::default();
    let p2 = p1.clone();
    println!("Is {:?}\nequal to {:?}?\nThe answer is {}!", &p1, &p2,
             if p1 == p2 { "yes" } else { "no" });
}
```

默认方法

特征可以依照其他特征方法来实现行为：

```
trait Equals {
    fn equals(&self, other: &Self) -> bool;
    fn not_equals(&self, other: &Self) -> bool {
        !self.equals(other)
    }
}

#[derive(Debug)]
struct Centimeter(i16);

impl Equals for Centimeter {
    fn equals(&self, other: &Centimeter) -> bool {
        self.0 == other.0
    }
}

fn main() {
    let a = Centimeter(10);
    let b = Centimeter(20);
    println!("{a:?} equals {b:?}: {}", a.equals(&b));
    println!("{a:?} not_equals {b:?}: {}", a.not_equals(&b));
}
```

▼ *Speaker Notes*

- trait 或许可指定预实现（默认）方法，以及用户需要自行实现的方法。具有默认实现的方法可以依赖于必需的方法。
- 将方法“not_equals”移至新的 trait“NotEquals”。
- 将“Equals”设为“NotEquals”的超 trait。

```
trait NotEquals: Equals {
    fn not_equals(&self, other: &Self) -> bool {
        !self.equals(other)
    }
}
```

- 为“Equals”提供“NotEquals”的通用实现。

```
trait NotEquals {
    fn not_equals(&self, other: &Self) -> bool;
}

impl<T> NotEquals for T where T: Equals {
    fn not_equals(&self, other: &Self) -> bool {
        !self.equals(other)
    }
}
```

- 借助通用实现，您不再需要将“Equals”作为“NotEqual”的超 trait。

特征边界

使用泛型时，您通常会想要利用类型来实现某些特性， 这样才能调用此特征的方法。

您可以使用 `T: Trait` 或 `impl Trait` 执行此操作：

```
fn duplicate<T: Clone>(a: T) -> (T, T) {
    (a.clone(), a.clone())
}

// Syntactic sugar for:
//   fn add_42_millions<T: Into<i32>>(x: T) -> i32 {
fn add_42_millions(x: impl Into<i32>) -> i32 {
    x.into() + 42_000_000
}

// struct NotClonable;

fn main() {
    let foo = String::from("foo");
    let pair = duplicate(foo);
    println!("{pair:?}");

    let many = add_42_millions(42_i8);
    println!("{many}");
    let many_more = add_42_millions(10_000_000);
    println!("{many_more}");
}
```

▼ *Speaker Notes*

显示 `where` 子句，学员在阅读代码时会看到它。

```
fn duplicate<T>(a: T) -> (T, T)
where
    T: Clone,
{
    (a.clone(), a.clone())
}
```

- 它会在您有多个形参的情况下整理函数签名。
- 它具有额外功能，因此也更强大。
  - 如果有人提问，便阐明额外功能是指“:”左侧的类别可为任意值，例如 `Option<T>` 。

impl Trait

与特征边界类似， `impl Trait` 语法可以在函数形参 和返回值中使用：

```
use std::fmt::Display;

fn get_x(name: impl Display) -> impl Display {
    format!("Hello {name}")
}

fn main() {
    let x = get_x("foo");
    println!("{x}");
}
```

- `impl Trait` 让您可使用无法命名的类型。

▼ *Speaker Notes*

`impl Trait` 的意义因使用位置而略有不同。

- 对形参来说， `impl Trait` 就像是具有特征边界的匿名泛型形参。
- 对返回值类型来说，它则意味着返回值类型就是实现该特征的某具体类型， 无需为该类型命名。如果您不想在公共 API 中公开该具体类型，便可 使用此方法。

  在返回位置处进行推断有一定难度。会返回 `impl Foo` 的函数会挑选 自身返回的具体类型，而不必在来源中写出此信息。会返回 泛型类型（例如 `collect<B>() -> B` ）的函数则可返回符合 `B` 的任何类型，而调用方可能需要选择一个类型，例如使用 `let x: Vec<_> = foo.collect()` 或使用以下 Turbofish： `foo.collect::<Vec<_>>()` 。

这是一个非常棒的示例，因为它使用了两次 `impl Display` 。这有助于说明 此处没有任何项目会强制使用相同的 `impl Display` 类型。如果我们使用单个 `T: Display` ，它会强制限制输入 `T` 和返回 `T` 均为同一类型。 这并不适用于这个特定函数，因为我们预期作为输入的类型可能 不会是 `format!` 返回的值。如果我们希望通过 `: Display` 语法执行相同的操作，则需要两个 独立的泛型形参。

重要特征

现在，我们来看看 Rust 标准库的一些最常见的特征：

- `Iterator` 和 `IntoIterator` 用于 `for` 循环中，
- `From` 和 `Into` 用于转换值，
- `Read` 和 `Write` 用于实现 IO。
- `Add` 、 `Mul` 等用于实现运算符重载，
- `Drop` 用于定义析构函数。
- `Default` 用于构建相应类型的默认实例。

迭代器

您可以自行实现 `Iterator` 特征：

```
struct Fibonacci {
    curr: u32,
    next: u32,
}

impl Iterator for Fibonacci {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        let new_next = self.curr + self.next;
        self.curr = self.next;
        self.next = new_next;
        Some(self.curr)
    }
}

fn main() {
    let fib = Fibonacci { curr: 0, next: 1 };
    for (i, n) in fib.enumerate().take(5) {
        println!("fib({i}): {n}");
    }
}
```

▼ *Speaker Notes*

- `Iterator` 特征会对集合实现许多常见的函数程序操作，例如 `map` filter `和 reduce` 等。您可以通过此特征找到有关它们的所有 文档。在 Rust 中，这些函数应生成代码，且生成的代码应与等效命令式实现一样 高效。
- `IntoIterator` 是迫使 for 循环运作的特征。此特征由集合类型（例如 `Vec<T>` ）和相关引用（例如 `&Vec<T>` 和 `&[T]` ）而实现。此外，范围也会实现这项特征。因此，您可以使用 `for i in some_vec { .. }` 来遍历某矢量，但 `some_vec.next()` 不存在。

FromIterator

`FromIterator` 让您可通过 `Iterator` 构建一个集合。

```
fn main() {
    let primes = vec![2, 3, 5, 7];
    let prime_squares = primes
        .into_iter()
        .map(|prime| prime * prime)
        .collect::<Vec<_>>();
    println!("prime_squares: {prime_squares:?}");
}
```

▼ *Speaker Notes*

`Iterator` 会实现 `fn collect<B>(self) -> B where B: FromIterator<Self::Item>, Self: Sized`

还有一些实现，让您可执行一些很酷的操作，比如 将 `Iterator<Item = Result<V, E>>` 转换成 `Result<Vec<V>, E>` 。

From 和 Into

类型会实现 `From` 和 `Into` 以加快类型转换：

```
fn main() {
    let s = String::from("hello");
    let addr = std::net::Ipv4Addr::from([127, 0, 0, 1]);
    let one = i16::from(true);
    let bigger = i32::from(123i16);
    println!("{s}, {addr}, {one}, {bigger}");
}
```

实现 `From` 后，系统会自动实现 `Into`：

```
fn main() {
    let s: String = "hello".into();
    let addr: std::net::Ipv4Addr = [127, 0, 0, 1].into();
    let one: i16 = true.into();
    let bigger: i32 = 123i16.into();
    println!("{s}, {addr}, {one}, {bigger}");
}
```

▼ *Speaker Notes*

- 这就是为什么通常只需实现 `From`，因为您的类型也会实现 `Into`。
- 若要声明某个函数实参输入类型（例如“任何可转换成 `String` 的类型”），规则便会相反，此时应使用 `Into`。您的函数会接受可实现 `From` 的类型，以及那些仅实现 `Into` 的类型。

Read 和 Write

您可以使用 `Read` 和 `BufRead` 对 `u8` 来源进行抽象化处理：

```
use std::io::{BufRead, BufReader, Read, Result};

fn count_lines<R: Read>(reader: R) -> usize {
    let buf_reader = BufReader::new(reader);
    buf_reader.lines().count()
}

fn main() -> Result<()> {
    let slice: &[u8] = b"foo\nbar\nbaz\n";
    println!("lines in slice: {}", count_lines(slice));

    let file = std::fs::File::open(std::env::current_exe()?)?;
    println!("lines in file: {}", count_lines(file));
    Ok(())
}
```

您同样可使用 `Write` 对 `u8` 接收器进行抽象化处理：

```
use std::io::{Result, Write};

fn log<W: Write>(writer: &mut W, msg: &str) -> Result<()> {
    writer.write_all(msg.as_bytes())?;
    writer.write_all("\n".as_bytes())
}

fn main() -> Result<()> {
    let mut buffer = Vec::new();
    log(&mut buffer, "Hello")?;
    log(&mut buffer, "World")?;
    println!("Logged: {:?}", buffer);
    Ok(())
}
```

Drop 特征

用于实现 `Drop` 的值可以指定在超出范围时运行的代码：

```
struct Droppable {
    name: &'static str,
}

impl Drop for Droppable {
    fn drop(&mut self) {
        println!("Dropping {}", self.name);
    }
}

fn main() {
    let a = Droppable { name: "a" };
    {
        let b = Droppable { name: "b" };
        {
            let c = Droppable { name: "c" };
            let d = Droppable { name: "d" };
            println!("Exiting block B");
        }
        println!("Exiting block A");
    }
    drop(a);
    println!("Exiting main");
}
```

▼ *Speaker Notes*

- Note that `std::mem::drop` is not the same as `std::ops::Drop::drop`.
- Values are automatically dropped when they go out of scope.
- When a value is dropped, if it implements `std::ops::Drop` then its `Drop::drop` implementation will be called.
- All its fields will then be dropped too, whether or not it implements `Drop`.
- `std::mem::drop` is just an empty function that takes any value. The significance is that it takes ownership of the value, so at the end of its scope it gets dropped. This makes it a convenient way to explicitly drop values earlier than they would otherwise go out of scope.
    - This can be useful for objects that do some work on `drop`: releasing locks, closing files, etc.

讨论点：

- 为什么 `Drop::drop` 不使用 `self`？
    - 简答：如果这样的话，系统会在代码块结尾 调用 `std::mem::drop`，进而引发再一次调用 `Drop::drop`，并引发堆栈 溢出！

- 尝试用 `a.drop()` 替换 `drop(a)` 。

Default 特征

`Default` 特征会为类型生成默认值。

```
#[derive(Debug, Default)]
struct Derived {
    x: u32,
    y: String,
    z: Implemented,
}

#[derive(Debug)]
struct Implemented(String);

impl Default for Implemented {
    fn default() -> Self {
        Self("John Smith".into())
    }
}

fn main() {
    let default_struct = Derived::default();
    println!("{default_struct:#?}");

    let almost_default_struct = Derived {
        y: "Y is set!".into(),
        ..Derived::default()
    };
    println!("{almost_default_struct:#?}");

    let nothing: Option<Derived> = None;
    println!("{:#?}", nothing.unwrap_or_default());
}

```

▼ *Speaker Notes*

- 系统可以直接实现它，也可以通过 `#[derive(Default)]` 派生出它。
- A derived implementation will produce a value where all fields are set to their default values.
    - 这意味着，该结构体中的所有类型也都必须实现 `Default` 。
- 标准的 Rust 类型通常会以合理的值（例如 `0` “” `` ` `` 等）实现 Default`。
- 部分结构体副本可与默认值完美搭配运作。
- Rust 标准库了解类型可能会实现 `Default` ，因此提供了便利的使用方式。
- “..”语法被称为结构体更新语法

AddMul ``...

运算符重载是通过 `std::ops` 中的特征实现的：

```
#[derive(Debug, Copy, Clone)]
struct Point { x: i32, y: i32 }

impl std::ops::Add for Point {
    type Output = Self;

    fn add(self, other: Self) -> Self {
        Self {x: self.x + other.x, y: self.y + other.y}
    }
}

fn main() {
    let p1 = Point { x: 10, y: 20 };
    let p2 = Point { x: 100, y: 200 };
    println!("{:?} + {:?} = {:?}", p1, p2, p1 + p2);
}
```

▼ *Speaker Notes*

讨论点：

- 您可以针对 `&Point` 实现 `Add` 。此做法在哪些情况下可派上用场？
  - 回答： `Add:add` 会耗用 `self` 。如果您的运算符重载对象（即类型 `T` ）不是 `Copy` ，建议您也为 `&T` 重载运算符。这可避免调用点上存在不必要的 克隆任务。
- 为什么 `Output` 是关联类型？可将它用作该方法的类型形参吗？
  - 简答：函数类型形参是由调用方控管，但 `Output` 这类关联类型则由特征实现人员 控管。
- 您可以针对两种不同类型实现 `Add` ，例如， `impl Add<(i32, i32)> for Point` 会向 `Point` 中添加元组。

闭包

闭包或 lambda 表达式具有无法命名的类型。不过，它们会 实现特殊的 `Fn` ， `FnMut` 和 `FnOnce` 特征：

```
fn apply_with_log(func: impl FnOnce(i32) -> i32, input: i32) -> i32 {
    println!("Calling function on {input}");
    func(input)
}

fn main() {
    let add_3 = |x| x + 3;
    println!("add_3: {}", apply_with_log(add_3, 10));
    println!("add_3: {}", apply_with_log(add_3, 20));

    let mut v = Vec::new();
    let mut accumulate = |x: i32| {
        v.push(x);
        v.iter().sum::<i32>()
    };
    println!("accumulate: {}", apply_with_log(&mut accumulate, 4));
    println!("accumulate: {}", apply_with_log(&mut accumulate, 5));

    let multiply_sum = |x| x * v.into_iter().sum::<i32>();
    println!("multiply_sum: {}", apply_with_log(multiply_sum, 3));
}
```

▼ *Speaker Notes*

`Fn` （例如 `add_3` ）既不会耗用也不会修改捕获的值，或许 也不会捕获任何值。它可被并发调用多次。

`FnMut` （例如 `accumulate` ）可能会改变捕获的值。您可以多次调用它， 但不能并发调用它。

如果您使用 `FnOnce` （例如 `multiply_sum` ），或许只能调用它一次。它可能会耗用 所捕获的值。

`FnMut` 是 `FnOnce` 的子类型。 `Fn` 是 `FnMut` 和 `FnOnce` 的子类型。也就是说，您可以在任何 需要调用 `FnOnce` 的地方使用 `FnMut` ，还可在任何需要调用 `FnMut` 或 `FnOnce` 的地方 使用 `Fn` 。

编译器也会推断 `Copy` （例如针对 `add_3` ）和 `Clone` （例如 `multiply_sum` ）， 具体取决于闭包捕获的数据。

默认情况下，闭包会依据引用来捕获数据（如果可以的话）。 `move` 关键字则可让闭包依据值 来捕获数据。

```rust
fn make_greeter(prefix: String) -> impl Fn(&str) {
    return move |name| println!("{} {}", prefix, name)
}

fn main() {
    let hi = make_greeter("Hi".to_string());
    hi("there");
}
```

第 3 天：上午练习

我们将使用 trait 和 trait 对象设计一个经典的 GUI 库。

我们还将通过点和多边形的相关练习，探讨枚举调度情况。

▼ *Speaker Notes*

After looking at the exercises, you can look at the solutions provided.

Drawing A Simple GUI

Let us design a classical GUI library using our new knowledge of traits and trait objects. We'll only implement the drawing of it (as text) for simplicity.

我们的库中有许多 widget：

- “Window”：具有“title”且包含其他 widget。
- `Button`: has a `label`. In reality, it would also take a callback function to allow the program to do something when the button is clicked but we won't include that since we're only drawing the GUI.
- “Label”：具有“label”。

这些 widget 将实现“Widget”trait，如下所示。

将以下代码复制到 https://play.rust-lang.org/，然后填入缺少的“draw_into”方法，以便实现“Widget”trait：

```
// TODO: remove this when you're done with your implementation.
#![allow(unused_imports, unused_variables, dead_code)]

pub trait Widget {
    /// Natural width of `self`.
    fn width(&self) -> usize;

    /// Draw the widget into a buffer.
    fn draw_into(&self, buffer: &mut dyn std::fmt::Write);

    /// Draw the widget on standard output.
    fn draw(&self) {
        let mut buffer = String::new();
        self.draw_into(&mut buffer);
        println!("{buffer}");
    }
}

pub struct Label {
    label: String,
}

impl Label {
    fn new(label: &str) -> Label {
        Label {
            label: label.to_owned(),
        }
    }
}

pub struct Button {
    label: Label,
}

impl Button {
    fn new(label: &str) -> Button {
        Button {
            label: Label::new(label),
        }
    }
}

pub struct Window {
    title: String,
    widgets: Vec<Box<dyn Widget>>,
}

impl Window {
    fn new(title: &str) -> Window {
        Window {
            title: title.to_owned(),
            widgets: Vec::new(),
        }
    }

    fn add_widget(&mut self, widget: Box<dyn Widget>) {
        self.widgets.push(widget);
```

```
    }

    fn inner_width(&self) -> usize {
        std::cmp::max(
            self.title.chars().count(),
            self.widgets.iter().map(|w| w.width()).max().unwrap_or(0),
        )
    }
}


impl Widget for Label {
    fn width(&self) -> usize {
        unimplemented!()
    }

    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        unimplemented!()
    }
}

impl Widget for Button {
    fn width(&self) -> usize {
        unimplemented!()
    }

    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        unimplemented!()
    }
}

impl Widget for Window {
    fn width(&self) -> usize {
        unimplemented!()
    }

    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        unimplemented!()
    }
}

fn main() {
    let mut window = Window::new("Rust GUI Demo 1.23");
    window.add_widget(Box::new(Label::new("This is a small text GUI demo.")));
    window.add_widget(Box::new(Button::new(
        "Click me!"
    )));
    window.draw();
}
```

上述程序的输出可能非常简单，例如：

```
========
Rust GUI Demo 1.23
========

This is a small text GUI demo.

| Click me! |
```

如果要绘制对齐的文本，可以使用填充/对齐格式设置运算符。需要特别注意的是您填充不同字符（此处是“/”）的方式以及控制对齐的方式：

```rust
fn main() {
    let width = 10;
    println!("left aligned:  |{:/<width$}|", "foo");
    println!("centered:      |{:/^width$}|", "foo");
    println!("right aligned: |{:/>width$}|", "foo");
}
```

使用这些对齐技巧，您可以生成如下的输出内容：

```
+--------------------------------+
|       Rust GUI Demo 1.23       |
+================================+
| This is a small text GUI demo. |
| +------------+                 |
| | Click me! |                  |
| +------------+                 |
+--------------------------------+
```

多边形结构体

我们将创建一个包含一些点的“Polygon”结构体。将以下代码复制到 https://play.rust-lang.org/，然后填入缺少的方法，设法通过测试：

```rust
// TODO: remove this when you're done with your implementation.
#![allow(unused_variables, dead_code)]

pub struct Point {
    // add fields
}

impl Point {
    // add methods
}

pub struct Polygon {
    // add fields
}

impl Polygon {
    // add methods
}

pub struct Circle {
    // add fields
}

impl Circle {
    // add methods
}

pub enum Shape {
    Polygon(Polygon),
    Circle(Circle),
}

#[cfg(test)]
mod tests {
    use super::*;

    fn round_two_digits(x: f64) -> f64 {
        (x * 100.0).round() / 100.0
    }

    #[test]
    fn test_point_magnitude() {
        let p1 = Point::new(12, 13);
        assert_eq!(round_two_digits(p1.magnitude()), 17.69);
    }

    #[test]
    fn test_point_dist() {
        let p1 = Point::new(10, 10);
        let p2 = Point::new(14, 13);
        assert_eq!(round_two_digits(p1.dist(p2)), 5.00);
    }

    #[test]
    fn test_point_add() {
        let p1 = Point::new(16, 16);
        let p2 = p1 + Point::new(-4, 3);
```

```
        assert_eq!(p2, Point::new(12, 19));
    }

    #[test]
    fn test_polygon_left_most_point() {
        let p1 = Point::new(12, 13);
        let p2 = Point::new(16, 16);

        let mut poly = Polygon::new();
        poly.add_point(p1);
        poly.add_point(p2);
        assert_eq!(poly.left_most_point(), Some(p1));
    }

    #[test]
    fn test_polygon_iter() {
        let p1 = Point::new(12, 13);
        let p2 = Point::new(16, 16);

        let mut poly = Polygon::new();
        poly.add_point(p1);
        poly.add_point(p2);

        let points = poly.iter().cloned().collect::<Vec<_>>();
        assert_eq!(points, vec![Point::new(12, 13), Point::new(16, 16)]);
    }

    #[test]
    fn test_shape_perimeters() {
        let mut poly = Polygon::new();
        poly.add_point(Point::new(12, 13));
        poly.add_point(Point::new(17, 11));
        poly.add_point(Point::new(16, 16));
        let shapes = vec![
            Shape::from(poly),
            Shape::from(Circle::new(Point::new(10, 20), 5)),
        ];
        let perimeters = shapes
            .iter()
            .map(Shape::perimeter)
            .map(round_two_digits)
            .collect::<Vec<_>>();
        assert_eq!(perimeters, vec![15.48, 31.42]);
    }
}

#[allow(dead_code)]
fn main() {}
```

▼ *Speaker Notes*

由于问题语句中缺少方法签名，因此练习的关键部分是正确指定这些内容。您无需修改测试。

练习的其他有趣部分：

- 为某些结构体派生“Copy”trait，因为在测试中，方法有时不借用它们的参数。
- 发现必须实现“Add”trait 才能通过“+”添加两个对象。请注意，我们在第 3 天之前不会讨论泛型。

错误处理

Rust 中的错误处理是使用显式控制流来进行的：

- 包含错误的函数会在返回类型中列出相关信息。
- 此规则没有例外。

Panics

如果运行时发生严重错误，Rust 会触发 panic：

```
fn main() {
    let v = vec![10, 20, 30];
    println!("v[100]: {}", v[100]);
}
```

- Panic 用于指示不可恢复的意外错误。
  - Panic反映了程序中的 bug 问题。
- 如果崩溃不可接受，请使用不会触发 panic 的 API（例如 `Vec::get` ）。

捕获堆栈展开

默认情况下，panic 会导致堆栈展开。您可以捕获展开信息：

```rust
use std::panic;

fn main() {
    let result = panic::catch_unwind(|| {
        println!("hello!");
    });
    assert!(result.is_ok());

    let result = panic::catch_unwind(|| {
        panic!("oh no!");
    });
    assert!(result.is_err());
}
```

- 如果服务器需要持续运行（即使是在请求发生崩溃的情况下）， 此方法十分有用。
- 如果您在 `Cargo.toml` 中设置了 `panic = 'abort'`，此方法不会生效。

使用 Result 进行结构化错误处理

在前面，我们看到了 `Result` 枚举。在遇到正常操作产生的预期错误时， 我们常会用到此方法：

```
use std::fs;
use std::io::Read;

fn main() {
    let file = fs::File::open("diary.txt");
    match file {
        Ok(mut file) => {
            let mut contents = String::new();
            file.read_to_string(&mut contents);
            println!("Dear diary: {contents}");
        },
        Err(err) => {
            println!("The diary could not be opened: {err}");
        }
    }
}
```

▼ *Speaker Notes*

- 与 `Option` 方法相同，成功值位于 `Result` 方法内部， 开发者必须显示提取成功值。因此，建议进行错误检查。在绝不应出现错误的情况下， 可以调用 `unwrap()` 或 `expect()` 方法，这也是一种开发者意向信号。
- 我们建议阅读 `Result` 文档。虽然课程中不会涉及该文档，但是有必要提到它。 该文档中包含许多便捷的方法和函数，对于函数式编程很有帮助。

使用 ? 传播错误

try 操作符 `?` 用于将错误返回给调用方。它能把常用命令

```
match some_expression {
    Ok(value) => value,
    Err(err) => return Err(err),
}
```

转换成更简单的命令

```
some_expression?
```

We can use this to simplify our error handling code:

```
use std::{fs, io};
use std::io::Read;

fn read_username(path: &str) -> Result<String, io::Error> {
    let username_file_result = fs::File::open(path);
    let mut username_file = match username_file_result {
        Ok(file) => file,
        Err(err) => return Err(err),
    };

    let mut username = String::new();
    match username_file.read_to_string(&mut username) {
        Ok(_) => Ok(username),
        Err(err) => Err(err),
    }
}

fn main() {
    //fs::write("config.dat", "alice").unwrap();
    let username = read_username("config.dat");
    println!("username or error: {username:?}");
}
```

▼ *Speaker Notes*

关键点：

- `username` 变量可以是 `Ok(string)` 或 `Err(error)`。
- 可以使用 `fs::write` 调用来测试不同的场景：没有文件、空文件、包含用户名的文件。
- The return type of the function has to be compatible with the nested functions it calls. For instance, a function returning a `Result<T, Err>` can only apply the `?` operator on a function returning a `Result<AnyT, Err>`. It cannot apply the `?` operator on a function returning an `Option<AnyT>` or `Result<T, OtherErr>` unless `OtherErr`

implements `From<Err>`. Reciprocally, a function returning an `Option<T>` can only apply the `?` operator on a function returning an `Option<AnyT>`.

- 您可以使用其他“Option”和“Result”方法（例如“Option::ok_or”“Result::ok” “Result::err”）将不兼容的类型转换为另一种类型。

转换错误类型

`?` 的有效展开比前面介绍的内容略微复杂一些：

```
expression?
```

效果等同于

```
match expression {
    Ok(value) => value,
    Err(err)  => return Err(From::from(err)),
}
```

此处的 `From::from` 调用表示，我们尝试将错误类型转换为 函数返回的类型：

转换错误类型

```rust
use std::error::Error;
use std::fmt::{self, Display, Formatter};
use std::fs::{self, File};
use std::io::{self, Read};

#[derive(Debug)]
enum ReadUsernameError {
    IoError(io::Error),
    EmptyUsername(String),
}

impl Error for ReadUsernameError {}

impl Display for ReadUsernameError {
    fn fmt(&self, f: &mut Formatter) -> fmt::Result {
        match self {
            Self::IoError(e) => write!(f, "IO error: {e}"),
            Self::EmptyUsername(filename) => write!(f, "Found no username ir
        }
    }
}

impl From<io::Error> for ReadUsernameError {
    fn from(err: io::Error) -> ReadUsernameError {
        ReadUsernameError::IoError(err)
    }
}

fn read_username(path: &str) -> Result<String, ReadUsernameError> {
    let mut username = String::with_capacity(100);
    File::open(path)?.read_to_string(&mut username)?;
    if username.is_empty() {
        return Err(ReadUsernameError::EmptyUsername(String::from(path)));
    }
    Ok(username)
}

fn main() {
    //fs::write("config.dat", "").unwrap();
    let username = read_username("config.dat");
    println!("username or error: {username:?}");
}
```

▼ *Speaker Notes*

关键点：

- `username` 变量可以是 `Ok(string)` 或 `Err(error)`。
- 可以使用 `fs::write` 调用来测试不同的场景：没有文件、空文件、包含用户名的文件。

对所有不需要是“no_std”的错误类型来说，实现“std::error::Error”是一种很好的做法，而这需要“Debug”和“Display”。“core”的“Error”crate 仅在 nightly 提供，因此尚未与“no_std”完全兼容。

It’s generally helpful for them to implement `Clone` and `Eq` too where possible, to make life easier for tests and consumers of your library. In this case we can’t easily do so, because `io::Error` doesn’t implement them.

派生错误枚举

thiserror crate 是创建错误枚举的常用方法， 就像前一页中提供的示例一样：

```
use std::{fs, io};
use std::io::Read;
use thiserror::Error;

#[derive(Debug, Error)]
enum ReadUsernameError {
    #[error("Could not read: {0}")]
    IoError(#[from] io::Error),
    #[error("Found no username in {0}")]
    EmptyUsername(String),
}

fn read_username(path: &str) -> Result<String, ReadUsernameError> {
    let mut username = String::new();
    fs::File::open(path)?.read_to_string(&mut username)?;
    if username.is_empty() {
        return Err(ReadUsernameError::EmptyUsername(String::from(path)));
    }
    Ok(username)
}

fn main() {
    //fs::write("config.dat", "").unwrap();
    match read_username("config.dat") {
        Ok(username) => println!("Username: {username}"),
        Err(err)     => println!("Error: {err}"),
    }
}
```

▼ *Speaker Notes*

`thiserror` 的派生宏会自动实现 `std::error::Error`，并且可以选择性地实现 `Display`（如果提供了 `#[error(...)]` 属性）和 `From`（如果添加了 `#[from]` 属性）。此规则也适用于结构体。

但是，此规则不会影响公共 API，对于库而言，这非常理想。

动态错误类型

有时，我们需要允许返回任意类型的错误，但又不想自己手动编写枚举来涵盖所有不同的可能性。 `std::error::Error` 可以让我们轻松做到这一点。

```
use std::fs;
use std::io::Read;
use thiserror::Error;
use std::error::Error;

#[derive(Clone, Debug, Eq, Error, PartialEq)]
#[error("Found no username in {0}")]
struct EmptyUsernameError(String);

fn read_username(path: &str) -> Result<String, Box<dyn Error>> {
    let mut username = String::new();
    fs::File::open(path)?.read_to_string(&mut username)?;
    if username.is_empty() {
        return Err(EmptyUsernameError(String::from(path)).into());
    }
    Ok(username)
}

fn main() {
    //fs::write("config.dat", "").unwrap();
    match read_username("config.dat") {
        Ok(username) => println!("Username: {username}"),
        Err(err)     => println!("Error: {err}"),
    }
}
```

▼ *Speaker Notes*

虽然这可以省却编写代码的麻烦，但也会导致我们无法在程序中以不同的方式正常处理不同的 错误情况。因此，在库的公共 API 中使用 `Box<dyn Error>` 通常不是一个好主意。 但是对于您只需要在某处显示错误消息的程序来说，这不失为一个 很好的选择。

为错误添加背景信息

广泛使用的 anyhow crate 可以帮助我们为错误添加 背景信息，并减少自定义错误类型的 数量。

```
use std::{fs, io};
use std::io::Read;
use anyhow::{Context, Result, bail};

fn read_username(path: &str) -> Result<String> {
    let mut username = String::with_capacity(100);
    fs::File::open(path)
        .with_context(|| format!("Failed to open {path}"))?
        .read_to_string(&mut username)
        .context("Failed to read")?;
    if username.is_empty() {
        bail!("Found no username in {path}");
    }
    Ok(username)
}

fn main() {
    //fs::write("config.dat", "").unwrap();
    match read_username("config.dat") {
        Ok(username) => println!("Username: {username}"),
        Err(err)     => println!("Error: {err:?}"),
    }
}
```

▼ *Speaker Notes*

- `anyhow::Result<V>` is a type alias for `Result<V, anyhow::Error>`.
- `anyhow::Error` is essentially a wrapper around `Box<dyn Error>`. As such it's again generally not a good choice for the public API of a library, but is widely used in applications.
- Actual error type inside of it can be extracted for examination if necessary.
- Functionality provided by `anyhow::Result<T>` may be familiar to Go developers, as it provides similar usage patterns and ergonomics to `(T, error)` from Go.

测试

Rust 和 Cargo 随附了一个简单的单元测试框架：

- 单元测试在您的整个代码中都受支持。
- 您可以通过 `tests/` 目录来支持集成测试。

单元测试

使用 `#[test]` 标记单元测试：

```
fn first_word(text: &str) -> &str {
    match text.find(' ') {
        Some(idx) => &text[..idx],
        None => &text,
    }
}

#[test]
fn test_empty() {
    assert_eq!(first_word(""), "");
}

#[test]
fn test_single_word() {
    assert_eq!(first_word("Hello"), "Hello");
}

#[test]
fn test_multiple_words() {
    assert_eq!(first_word("Hello World"), "Hello");
}
```

使用 `cargo test` 查找并运行单元测试。

测试模块

单元测试通常会放在嵌套模块中（在 Playground 上运行测试）：

```
fn helper(a: &str, b: &str) -> String {
    format!("{a} {b}")
}

pub fn main() {
    println!("{}", helper("Hello", "World"));
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_helper() {
        assert_eq!(helper("foo", "bar"), "foo bar");
    }
}
```

- 这样一来，您可以对专用帮助程序进行单元测试。
- 仅当您运行 `cargo test` 时， `#[cfg(test)]` 属性才有效。

文档测试

Rust 本身就支持文档测试：

```rust
/// Shortens a string to the given length.
///
/// ```
/// # use playground::shorten_string;
/// assert_eq!(shorten_string("Hello World", 5), "Hello");
/// assert_eq!(shorten_string("Hello World", 20), "Hello World");
/// ```
pub fn shorten_string(s: &str, length: usize) -> &str {
    &s[..std::cmp::min(length, s.len())]
}
```

- `///` 注释中的代码块会自动被视为 Rust 代码。
- 代码会作为 `cargo test` 的一部分进行编译和执行。
- Adding `#` in the code will hide it from the docs, but will still compile/run it.
- 在 Rust Playground 上测试上述代码。

集成测试

如果您想要以客户的身份测试您的库，请使用集成测试。

在 `tests/` 下方创建一个 `.rs` 文件：

```
use my_library::init;

#[test]
fn test_init() {
    assert!(init().is_ok());
}
```

这些测试只能使用您的 crate 的公共 API。

用于编写测试的实用 crate

Rust 仅为编写测试提供基本支持。

下面列出了我们建议在编写测试时使用的一些其他 crate：

- googletest：遵从 GoogleTest for C++ 传统的综合测试断言库。
- proptest：基于属性的测试，适用于 Rust。
- rstest：支持固件和参数化测试。

不安全 Rust

Rust 语言包含两个部分：

- **安全 Rust：**内存安全，没有潜在的未定义行为。
- **不安全 Rust：**如果违反了前提条件，可能会触发未定义的行为。

本课程中出现的大多为“安全 Rust”，但是了解“不安全 Rust”的定义 非常重要。

不安全的代码通常内容很少而且与其他代码隔离， 其正确性也应得到仔细记录。这类代码通常封装在安全的抽象层中。

不安全 Rust 提供了五种新功能：

- 解引用原始指针。
- 访问或修改可变的静态变量。
- 访问 `union` 字段。
- 调用 `unsafe` 函数，包括 `extern` 函数。
- 实现 `unsafe` trait。

下面，我们将简要介绍这些不安全功能。如需了解完整详情，请参阅 《Rust 手册》第 19.1 章 和 Rustonomicon。

▼ *Speaker Notes*

不安全 Rust 并不意味着代码不正确，而是这意味着开发者已停用 编译器的安全功能，必须自行编写正确的 代码。也就是说，编译器不再强制执行 Rust 的内存安全规则。

解引用裸指针

创建指针是安全的操作，但解引用指针需要使用 `unsafe` 方法：

```
fn main() {
    let mut num = 5;

    let r1 = &mut num as *mut i32;
    let r2 = r1 as *const i32;

    // Safe because r1 and r2 were obtained from references and so are
    // guaranteed to be non-null and properly aligned, the objects underlyir
    // the references from which they were obtained are live throughout the
    // whole unsafe block, and they are not accessed either through the
    // references or concurrently through any other pointers.
    unsafe {
        println!("r1 is: {}", *r1);
        *r1 = 10;
        println!("r2 is: {}", *r2);
    }
}
```

▼ *Speaker Notes*

我们建议（而且 Android Rust 样式指南要求）为每个 `unsafe` 代码块编写一条注释， 说明该代码块中的代码如何满足其所执行的不安全操作的 安全要求。

对于指针解除引用，这意味着指针必须为 *valid*，即：

- 指针必须为非 null。
- 指针必须是 *dereferenceable*（在单个已分配对象的边界内）。
- 对象不得已取消分配。
- 不得并发访问相同位置。
- 如果通过转换引用类型来获取指针，则底层对象必须处于活跃状态， 而且不得使用任何引用来访问内存。

在大多数情况下，指针还必须正确对齐。

可变的静态变量

读取不可变的静态变量是安全的操作：

```
static HELLO_WORLD: &str = "Hello, world!";

fn main() {
    println!("HELLO_WORLD: {HELLO_WORLD}");
}
```

但是，读取和写入可变的静态变量是不安全的，因为这可能会 造成数据争用：

```
static mut COUNTER: u32 = 0;

fn add_to_counter(inc: u32) {
    unsafe { COUNTER += inc; }  // Potential data race!
}

fn main() {
    add_to_counter(42);

    unsafe { println!("COUNTER: {COUNTER}"); }  // Potential data race!
}
```

▼ *Speaker Notes*

通常，我们不建议使用可变的静态变量，但在某些情况下，在低层级 `no_std` 代码中可能需要这样做， 例如实现堆分配器或使用某些 C API。

联合体

联合体与枚举类似，但您需要自行跟踪活跃字段：

```
#[repr(C)]
union MyUnion {
    i: u8,
    b: bool,
}

fn main() {
    let u = MyUnion { i: 42 };
    println!("int: {}", unsafe { u.i });
    println!("bool: {}", unsafe { u.b });  // Undefined behavior!
}
```

▼ *Speaker Notes*

在 Rust 中很少需要用到联合体，因为您通常可以使用枚举。联合体只是偶尔用于 与 C 库 API 进行交互。

如果您只是想将字节重新解释为其他类型，则可能需要使用 `std::mem::transmute` 或 安全的封装容器，例如 `zerocopy` crate。

调用 Unsafe 函数

如果函数或方法具有额外的前提条件，您必须遵守这些前提条件来避免未定义的行为， 则可以将该函数或方法标记为 `unsafe`：

```
fn main() {
    let emojis = "🗻∈🌏";

    // Safe because the indices are in the correct order, within the bounds
    // the string slice, and lie on UTF-8 sequence boundaries.
    unsafe {
        println!("emoji: {}", emojis.get_unchecked(0..4));
        println!("emoji: {}", emojis.get_unchecked(4..7));
        println!("emoji: {}", emojis.get_unchecked(7..11));
    }

    println!("char count: {}", count_chars(unsafe { emojis.get_unchecked(0..

    // Not upholding the UTF-8 encoding requirement breaks memory safety!
    // println!("emoji: {}", unsafe { emojis.get_unchecked(0..3) });
    // println!("char count: {}", count_chars(unsafe { emojis.get_unchecked(
}

fn count_chars(s: &str) -> usize {
    s.chars().map(|_| 1).sum()
}
```

编写 Unsafe 函数

如果您自己编写的函数需要满足特定条件以避免未定义的行为， 您可以将这些函数标记为 `unsafe` 。

```
/// Swaps the values pointed to by the given pointers.
///
/// # Safety
///
/// The pointers must be valid and properly aligned.
unsafe fn swap(a: *mut u8, b: *mut u8) {
    let temp = *a;
    *a = *b;
    *b = temp;
}

fn main() {
    let mut a = 42;
    let mut b = 66;

    // Safe because ...
    unsafe {
        swap(&mut a, &mut b);
    }

    println!("a = {}, b = {}", a, b);
}
```

▼ *Speaker Notes*

实际上，我们不会这样使用指针，因为使用引用可以安全地达到相同的目的。

请注意，在不安全函数中，可以在没有 `unsafe` 代码块的情况下使用不安全代码。我们可以 使用 `#[deny(unsafe_op_in_unsafe_fn)]` 来禁止此行为。请尝试添加该命令，看看会出现什么情况。

调用外部代码

基于其他语言的函数可能会违反 Rust 的保证。因此， 调用这类函数是不安全的：

```
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    unsafe {
        // Undefined behavior if abs misbehaves.
        println!("Absolute value of -3 according to C: {}", abs(-3));
    }
}
```

▼ *Speaker Notes*

这个问题通常仅存在于使用指针执行违反 Rust 内存模型的操作的外部函数中。 但一般而言，任何 C 函数都有可能在任意情况下出现未定义行为。

本例中的“C”是 ABI；也可以使用其他 ABI。

实现 Unsafe Trait

与函数一样，如果您在实现某个 trait 时必须保证特定条件来避免未定义的行为， 您也可以将该 trait 标记为 `unsafe` 。

例如， `zerocopy` crate 包含一个不安全的 trait， 大致内容是这样的：

```
use std::mem::size_of_val;
use std::slice;

/// ...
/// # Safety
/// The type must have a defined representation and no padding.
pub unsafe trait AsBytes {
    fn as_bytes(&self) -> &[u8] {
        unsafe {
            slice::from_raw_parts(self as *const Self as *const u8, size_of_
        }
    }
}

// Safe because u32 has a defined representation and no padding.
unsafe impl AsBytes for u32 {}
```

▼ *Speaker Notes*

在 Rustdoc 中有关 trait 的章节下，有一个标题为 `# 安全` 的部分介绍了 安全实现 trait 的要求。

实际上，与 `AsBytes` 相关的安全说明远比这里展示的更详尽、更复杂。

内置的 `Send` 和 `Sync` trait 都是不安全的。

第 3 天：下午练习

让我们构建一个用于读取目录内容的安全封装容器！

在本练习中，我们建议您使用本地开发环境，而不是 Playground。这样，您就可以在自己的机器上运行二进制文件。

首先，请按照在本地运行中的说明操作。

▼ *Speaker Notes*

看过练习后，您可以查看所提供的解题方法。

安全 FFI 封装容器

Rust has great support for calling functions through a *foreign function interface* (FFI). We will use this to build a safe wrapper for the `libc` functions you would use from C to read the names of files in a directory.

建议您参考以下手册页面：

- `opendir(3)`
- `readdir(3)`
- `closedir(3)`

您还需要浏览“std::ffi”模块。在下方，您会发现完成这个练习所需的多种字符串类型：

类型	编码	使用
“str”和“String”	UTF-8	用 Rust 进行文本处理
“CStr”和“CString”	以空字符结尾	与 C 函数通信
“OsStr”和“OsString”	特定于操作系统	与操作系统通信

您将在以下所有类型之间进行转换：

- 将 `&str` 转换为 `CString`：您需要为尾随 `\0` 字符分配空格，
- 将 `CString` 转换为 `\*const i8`：您需要一个指针来调用 C 函数，
- 将 `\*const i8` 转换为 `&CStr`：您需要一些能够找到尾随 `\0` 字符的内容，
- 将 `&CStr` 转换为 `&\[u8\]`：一个字节 Slice 是“一些未知数据”的通用接口，
- 将 `&\[u8\]` 转换为 `&OsStr`：`&OsStr` 是向 `OsString` 迈进的一步，请使用 `OsStrExt` 来创建它，
- 将“&OsStr”转换为“OsString”：您需要克隆“&OsStr”中的数据，以便能够返回它并再次调用“readdir”。

秘典 中也有一个关于 FFI 的非常实用的章节。

将以下代码复制到 https://play.rust-lang.org/，并填入缺少的函数和方法：

```rust
// TODO: remove this when you're done with your implementation.
#![allow(unused_imports, unused_variables, dead_code)]

mod ffi {
    use std::os::raw::{c_char, c_int};
    #[cfg(not(target_os = "macos"))]
    use std::os::raw::{c_long, c_ulong, c_ushort, c_uchar};

    // Opaque type. See https://doc.rust-lang.org/nomicon/ffi.html.
    #[repr(C)]
    pub struct DIR {
        _data: [u8; 0],
        _marker: core::marker::PhantomData<(*mut u8,
core::marker::PhantomPinned)>,
    }

    // Layout according to the Linux man page for readdir(3), where ino_t and
    // off_t are resolved according to the definitions in
    // /usr/include/x86_64-linux-gnu/{sys/types.h, bits/typesizes.h}.
    #[cfg(not(target_os = "macos"))]
    #[repr(C)]
    pub struct dirent {
        pub d_ino: c_ulong,
        pub d_off: c_long,
        pub d_reclen: c_ushort,
        pub d_type: c_uchar,
        pub d_name: [c_char; 256],
    }

    // Layout according to the macOS man page for dir(5).
    #[cfg(all(target_os = "macos"))]
    #[repr(C)]
    pub struct dirent {
        pub d_fileno: u64,
        pub d_seekoff: u64,
        pub d_reclen: u16,
        pub d_namlen: u16,
        pub d_type: u8,
        pub d_name: [c_char; 1024],
    }

    extern "C" {
        pub fn opendir(s: *const c_char) -> *mut DIR;

        #[cfg(not(all(target_os = "macos", target_arch = "x86_64")))]
        pub fn readdir(s: *mut DIR) -> *const dirent;

        // See https://github.com/rust-lang/libc/issues/414 and the section on
        // _DARWIN_FEATURE_64_BIT_INODE in the macOS man page for stat(2).
        //
        // "Platforms that existed before these updates were available" refers
        // to macOS (as opposed to iOS / wearOS / etc.) on Intel and PowerPC.
        #[cfg(all(target_os = "macos", target_arch = "x86_64"))]
        #[link_name = "readdir$INODE64"]
        pub fn readdir(s: *mut DIR) -> *const dirent;

        pub fn closedir(s: *mut DIR) -> c_int;
```

```
    }
}

use std::ffi::{CStr, CString, OsStr, OsString};
use std::os::unix::ffi::OsStrExt;

#[derive(Debug)]
struct DirectoryIterator {
    path: CString,
    dir: *mut ffi::DIR,
}

impl DirectoryIterator {
    fn new(path: &str) -> Result<DirectoryIterator, String> {
        // Call opendir and return a Ok value if that worked,
        // otherwise return Err with a message.
        unimplemented!()
    }
}

impl Iterator for DirectoryIterator {
    type Item = OsString;
    fn next(&mut self) -> Option<OsString> {
        // Keep calling readdir until we get a NULL pointer back.
        unimplemented!()
    }
}

impl Drop for DirectoryIterator {
    fn drop(&mut self) {
        // Call closedir as needed.
        unimplemented!()
    }
}

fn main() -> Result<(), String> {
    let iter = DirectoryIterator::new(".")?;
    println!("files: {:#?}", iter.collect::<Vec<_>>());
    Ok(())
}
```

欢迎来到Android 中的Rust

Rust 支持Android 的原生平台开发。这意味着您可以在Rust 中编写新的操作系统服务，以及扩展现有服务。

> 今天我们会尝试在你自己的项目中调用Rust。 所以试着在你的代码中找一小段来改成Rust。 代码中越少依赖(dependencies)，越少“独特”的类型，越好。比如 一段解析原始字符的代码就很理想。

设置

我们将会使用Android 虚拟设备（Android Virtual Device）来测试我们的代码。 确保你有权限访问一个，或者用以下命令创建一个新的：

```
source build/envsetup.sh
lunch aosp_cf_x86_64_phone-userdebug
acloud create
```

更多细节请参考 Android Developer Codelab.

构建规则

Android 构建系统（Soong）通过一系列模块来支持Rust：

Module Type	描述
`rust_binary`	生成一个Rust二进制文件。
`rust_library`	生成一个 Rust 库，并提供 `rlib` 和 `dylib` 两种变体。
`rust_ffi`	生成一个可由 `cc` 模块使用的 Rust C 库，并提供静态和共享两种变体。
`rust_proc_macro`	Produces a `proc-macro` Rust library. These are analogous to compiler plugins.
`rust_test`	生成使用标准 Rust 测试框架的 Rust 测试二进制文件。
`rust_fuzz`	Produces a Rust fuzz binary leveraging `libfuzzer` .
`rust_protobuf`	生成源代码并生成为特定 protobuf 提供接口的 Rust 库。
`rust_bindgen`	生成源代码并生成包含 Rust 绑定到 C 库的 Rust 库。

下面我们来看看 `rust_binary` 和 `rust_library` 。

Rust 二进制文件

让我们从一个简单的应用程序开始。在 AOSP 签出的根目录下，创建以下文件：

hello_rust/Android.bp:

```
rust_binary {
    name: "hello_rust",
    crate_name: "hello_rust",
    srcs: ["src/main.rs"],
}
```

hello_rust/src/main.rs:

```
//! Rust demo.

/// Prints a greeting to standard output.
fn main() {
    println!("Hello from Rust!");
}
```

你现在可以构建、推送和运行二进制文件：

```
m hello_rust
adb push "$ANDROID_PRODUCT_OUT/system/bin/hello_rust /data/local/tmp"
adb shell /data/local/tmp/hello_rust


Hello from Rust!
```

Rust 库

您可以使用 `rust_library` 为 Android 创建一个新的 Rust 库。

在这里，我们声明了对两个库的依赖：

- `libgreeting`，我们在下面进行了定义，
- `libtextwrap`，一个已经在 `external/rust/crates/` 中提供的 crate。

hello_rust/Android.bp：

```
rust_binary {
    name: "hello_rust_with_dep",
    crate_name: "hello_rust_with_dep",
    srcs: ["src/main.rs"],
    rustlibs: [
        "libgreetings",
        "libtextwrap",
    ],
    prefer_rlib: true,
}

rust_library {
    name: "libgreetings",
    crate_name: "greetings",
    srcs: ["src/lib.rs"],
}
```

hello_rust/src/main.rs：

```
//! Rust demo.

use greetings::greeting;
use textwrap::fill;

/// Prints a greeting to standard output.
fn main() {
    println!("{}", fill(&greeting("Bob"), 24));
}
```

hello_rust/src/lib.rs：

```
//! Greeting library.

/// Greet `name`.
pub fn greeting(name: &str) -> String {
    format!("Hello {name}, it is very nice to meet you!")
}
```

您可以像之前一样构建、推送和运行二进制文件：

```
m hello_rust_with_dep
adb push "$ANDROID_PRODUCT_OUT/system/bin/hello_rust_with_dep /data/local/tmp"
adb shell /data/local/tmp/hello_rust_with_dep


Hello Bob, it is very
nice to meet you!
```

AIDL

Rust 支持 Android 接口定义语言 (AIDL)：

- Rust 代码可以调用现有的 AIDL 服务器，
- 您可以在 Rust 中创建新的 AIDL 服务器。

AIDL 接口

您可以使用 AIDL 接口声明您的服务的 API：

birthday_service/aidl/com/example/birthdayservice/IBirthdayService.aidl：

```
package com.example.birthdayservice;

/** Birthday service interface. */
interface IBirthdayService {
    /** Generate a Happy Birthday message. */
    String wishHappyBirthday(String name, int years);
}
```

birthday_service/aidl/Android.bp：

```
aidl_interface {
    name: "com.example.birthdayservice",
    srcs: ["com/example/birthdayservice/*.aidl"],
    unstable: true,
    backend: {
        rust: { // Rust is not enabled by default
            enabled: true,
        },
    },
}
```

如果供应商分区中的二进制文件使用了您的 AIDL 文件，请添加 `vendor_available: true` 。

服务实现

我们现在可以实现AIDL服务：

birthday_service/src/lib.rs:

```
//! Implementation of the `IBirthdayService` AIDL interface.
use
com_example_birthdayservice::aidl::com::example::birthdayservice::IBirthdayServ
ice::IBirthdayService;
use com_example_birthdayservice::binder;

/// The `IBirthdayService` implementation.
pub struct BirthdayService;

impl binder::Interface for BirthdayService {}

impl IBirthdayService for BirthdayService {
    fn wishHappyBirthday(&self, name: &str, years: i32) ->
binder::Result<String> {
        Ok(format!(
            "Happy Birthday {name}, congratulations with the {years} years!"
        ))
    }
}
```

birthday_service/Android.bp:

```
rust_library {
    name: "libbirthdayservice",
    srcs: ["src/lib.rs"],
    crate_name: "birthdayservice",
    rustlibs: [
        "com.example.birthdayservice-rust",
        "libbinder_rs",
    ],
}
```

AIDL 服务器

最后，我们可以创建一个暴露服务的服务器：

birthday_service/src/server.rs：

```
//! Birthday service.
use birthdayservice::BirthdayService;
use
com_example_birthdayservice::aidl::com::example::birthdayservice::IBirthdayServ
ice::BnBirthdayService;
use com_example_birthdayservice::binder;

const SERVICE_IDENTIFIER: &str = "birthdayservice";

/// Entry point for birthday service.
fn main() {
    let birthday_service = BirthdayService;
    let birthday_service_binder = BnBirthdayService::new_binder(
        birthday_service,
        binder::BinderFeatures::default(),
    );
    binder::add_service(SERVICE_IDENTIFIER,
birthday_service_binder.as_binder())
        .expect("Failed to register service");
    binder::ProcessState::join_thread_pool()
}
```

birthday_service/Android.bp：

```
rust_binary {
    name: "birthday_server",
    crate_name: "birthday_server",
    srcs: ["src/server.rs"],
    rustlibs: [
        "com.example.birthdayservice-rust",
        "libbinder_rs",
        "libbirthdayservice",
    ],
    prefer_rlib: true,
}
```

部署

我们现在可以构建、推送和启动服务：

```
m birthday_server
adb push "$ANDROID_PRODUCT_OUT/system/bin/birthday_server /data/local/tmp"
adb shell /data/local/tmp/birthday_server
```

在另一个终端中，检查该服务是否正在运行：

```
adb shell service check birthdayservice
```

```
Service birthdayservice: found
```

您还可以使用 `service call` 命令调用该服务：

```
adb shell service call birthdayservice 1 s16 Bob i32 24
```

```
Result: Parcel(
  0x00000000: 00000000 00000036 00610048 00700070 '....6...H.a.p.p.'
  0x00000010: 00200079 00690042 00740072 00640068 'y. .B.i.r.t.h.d.'
  0x00000020: 00790061 00420020 0062006f 0020002c 'a.y. .B.o.b.,. .'
  0x00000030: 006f0063 0067006e 00610072 00750074 'c.o.n.g.r.a.t.u.'
  0x00000040: 0061006c 00690074 006e006f 00200073 'l.a.t.i.o.n.s. .'
  0x00000050: 00690077 00680074 00740020 00650068 'w.i.t.h. .t.h.e.'
  0x00000060: 00320020 00200034 00650079 00720061 ' .2.4. .y.e.a.r.'
  0x00000070: 00210073 00000000                   's.!.....        ')
```

AIDL 客户端

最后，我们可以为我们的新服务创建一个 Rust 客户端。

birthday_service/src/client.rs:

```
//! Birthday service.
use
com_example_birthdayservice::aidl::com::example::birthdayservice::IBirthdayServ
ice::IBirthdayService;
use com_example_birthdayservice::binder;

const SERVICE_IDENTIFIER: &str = "birthdayservice";

/// Connect to the BirthdayService.
pub fn connect() -> Result<binder::Strong<dyn IBirthdayService>,
binder::StatusCode> {
    binder::get_interface(SERVICE_IDENTIFIER)
}

/// Call the birthday service.
fn main() -> Result<(), binder::Status> {
    let name = std::env::args()
        .nth(1)
        .unwrap_or_else(|| String::from("Bob"));
    let years = std::env::args()
        .nth(2)
        .and_then(|arg| arg.parse::<i32>().ok())
        .unwrap_or(42);

    binder::ProcessState::start_thread_pool();
    let service = connect().expect("Failed to connect to BirthdayService");
    let msg = service.wishHappyBirthday(&name, years)?;
    println!("{msg}");
    Ok(())
}
```

birthday_service/Android.bp:

```
rust_binary {
    name: "birthday_client",
    crate_name: "birthday_client",
    srcs: ["src/client.rs"],
    rustlibs: [
        "com.example.birthdayservice-rust",
        "libbinder_rs",
    ],
    prefer_rlib: true,
}
```

请注意，客户端不依赖于 `libbirthdayservice` 。

在您的设备上构建、推送并运行客户端：

```
m birthday_client
adb push "$ANDROID_PRODUCT_OUT/system/bin/birthday_client /data/local/tmp"
adb shell /data/local/tmp/birthday_client Charlie 60


Happy Birthday Charlie, congratulations with the 60 years!
```

更改 API

让我们扩展API以提供更多功能：我们希望允许客户端指定生日贺卡的行列表：

```
package com.example.birthdayservice;

/** Birthday service interface. */
interface IBirthdayService {
    /** Generate a Happy Birthday message. */
    String wishHappyBirthday(String name, int years, in String[] text);
}
```

日志记录

你应该使用 `log` crate 来自动记录日志到 `logcat` （设备上）或 `stdout` （主机上）：

hello_rust_logs/Android.bp:

```
rust_binary {
    name: "hello_rust_logs",
    crate_name: "hello_rust_logs",
    srcs: ["src/main.rs"],
    rustlibs: [
        "liblog_rust",
        "liblogger",
    ],
    prefer_rlib: true,
    host_supported: true,
}
```

hello_rust_logs/src/main.rs:

```
//! Rust logging demo.

use log::{debug, error, info};

/// Logs a greeting.
fn main() {
    logger::init(
        logger::Config::default()
            .with_tag_on_device("rust")
            .with_min_level(log::Level::Trace),
    );
    debug!("Starting program.");
    info!("Things are going fine.");
    error!("Something went wrong!");
}
```

在你的设备上构建，推送，并运行二进制文件：

```
m hello_rust_logs
adb push "$ANDROID_PRODUCT_OUT/system/bin/hello_rust_logs /data/local/tmp"
adb shell /data/local/tmp/hello_rust_logs
```

日志将会在 `adb logcat` 中显示：

```
adb logcat -s rust
```

```
09-08 08:38:32.454  2420  2420 D rust: hello_rust_logs: Starting program.
09-08 08:38:32.454  2420  2420 I rust: hello_rust_logs: Things are going fine.
09-08 08:38:32.454  2420  2420 E rust: hello_rust_logs: Something went wrong!
```

互操作性

Rust 对于与其他编程语言的互操作性有着出色的支持。这意味着您可以：

- 从其他语言调用 Rust 函数。
- 从 Rust 调用用其他语言编写的函数。

当您从外部语言调用函数时，我们称之为使用 **外部函数接口**（*Foreign Function Interface*， FFI）。

与 C 的互操作性

Rust 对使用 C 调用约定链接目标文件提供了完整的支持。同样地，你可以导出 Rust 函数并从 C 中调用它们。

如果你愿意的话，你可以手工完成它：

```
extern "C" {
    fn abs(x: i32) -> i32;
}

fn main() {
    let x = -42;
    let abs_x = unsafe { abs(x) };
    println!("{x}, {abs_x}");
}
```

我们已经在安全 FFI 封装容器练习中看到了这个例子。

> 这假设对目标平台拥有充分的了解，不建议用于生产环境。

接下来我们将探讨更好的选择。

使用 Bindgen

bindgen 工具可以自动生成 C 头文件的绑定代码。

首先创建一个小型的 C 语言库：

interoperability/bindgen/libbirthday.h:

```
typedef struct card {
  const char* name;
  int years;
} card;

void print_card(const card* card);
```

interoperability/bindgen/libbirthday.c:

```
#include <stdio.h>
#include "libbirthday.h"

void print_card(const card* card) {
  printf("+--------------\n");
  printf("| Happy Birthday %s!\n", card->name);
  printf("| Congratulations with the %i years!\n", card->years);
  printf("+--------------\n");
}
```

将该库添加到你的 `Android.bp` 文件中：

interoperability/bindgen/Android.bp:

```
cc_library {
    name: "libbirthday",
    srcs: ["libbirthday.c"],
}
```

为该库创建一个包装头文件（在此示例中不是必需的）：

interoperability/bindgen/libbirthday_wrapper.h:

```
#include "libbirthday.h"
```

您现在可以自动生成绑定代码：

interoperability/bindgen/Android.bp:

```
rust_bindgen {
    name: "libbirthday_bindgen",
    crate_name: "birthday_bindgen",
    wrapper_src: "libbirthday_wrapper.h",
    source_stem: "bindings",
    static_libs: ["libbirthday"],
}
```

最后，我们可以在 Rust 程序中使用这些绑定：

interoperability/bindgen/Android.bp:

```
rust_binary {
    name: "print_birthday_card",
    srcs: ["main.rs"],
    rustlibs: ["libbirthday_bindgen"],
}
```

interoperability/bindgen/main.rs:

```
//! Bindgen demo.

use birthday_bindgen::{card, print_card};

fn main() {
    let name = std::ffi::CString::new("Peter").unwrap();
    let card = card {
        name: name.as_ptr(),
        years: 42,
    };
    unsafe {
        print_card(&card as *const card);
    }
}
```

在你的设备上构建，推送，并运行二进制文件：

```
m print_birthday_card
adb push "$ANDROID_PRODUCT_OUT/system/bin/print_birthday_card /data/local/tmp"
adb shell /data/local/tmp/print_birthday_card
```

最后，我们可以运行自动生成的测试来确保绑定代码正常工作：

interoperability/bindgen/Android.bp:

```
rust_test {
    name: "libbirthday_bindgen_test",
    srcs: [":libbirthday_bindgen"],
    crate_name: "libbirthday_bindgen_test",
    test_suites: ["general-tests"],
    auto_gen_config: true,
    clippy_lints: "none", // Generated file, skip linting
    lints: "none",
}


atest libbirthday_bindgen_test
```

调用 Rust

将 Rust 函数和类型导出到 C 很简单：

interoperability/rust/libanalyze/analyze.rs

```
//! Rust FFI demo.
#![deny(improper_ctypes_definitions)]

use std::os::raw::c_int;

/// Analyze the numbers.
#[no_mangle]
pub extern "C" fn analyze_numbers(x: c_int, y: c_int) {
    if x < y {
        println!("x ({x}) is smallest!");
    } else {
        println!("y ({y}) is probably larger than x ({x})");
    }
}
```

interoperability/rust/libanalyze/analyze.h

```
#ifndef ANALYSE_H
#define ANALYSE_H

extern "C" {
void analyze_numbers(int x, int y);
}

#endif
```

interoperability/rust/libanalyze/Android.bp

```
rust_ffi {
    name: "libanalyze_ffi",
    crate_name: "analyze_ffi",
    srcs: ["analyze.rs"],
    include_dirs: ["."],
}
```

我们现在可以从一个 C 二进制文件中调用它：

interoperability/rust/analyze/main.c

```c
#include "analyze.h"

int main() {
  analyze_numbers(10, 20);
  analyze_numbers(123, 123);
  return 0;
}
```

interoperability/rust/analyze/Android.bp

```
cc_binary {
    name: "analyze_numbers",
    srcs: ["main.c"],
    static_libs: ["libanalyze_ffi"],
}
```

在你的设备上构建，推送，并运行二进制文件：

```
m analyze_numbers
adb push "$ANDROID_PRODUCT_OUT/system/bin/analyze_numbers /data/local/tmp"
adb shell /data/local/tmp/analyze_numbers
```

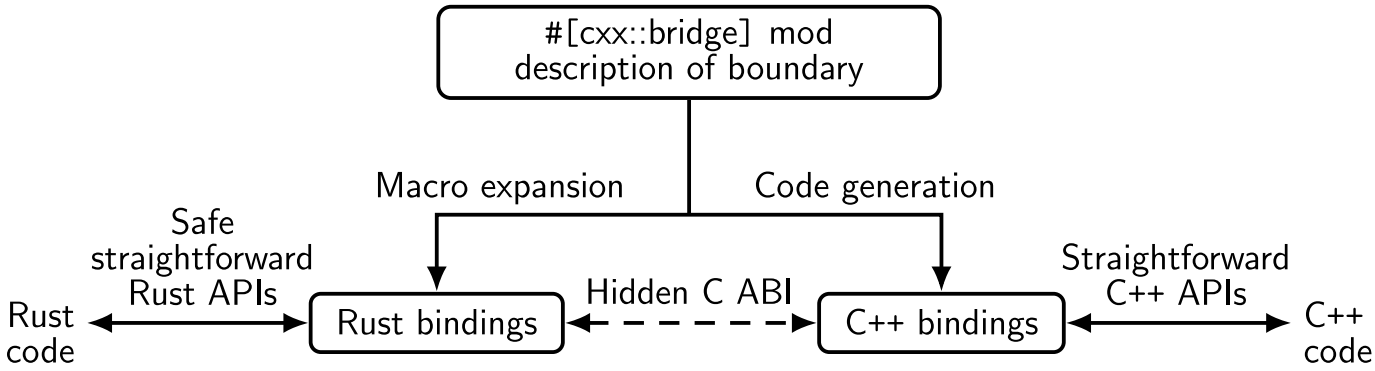
▼ *Speaker Notes*

`#[no_mangle]` 禁用了 Rust 通常的名称重整，因此导出的符号将仅为函数的名称。你还可以使用 `#[export_name = "some_name"]` 来指定任意你想要的名称。

与 C++ 交互

CXX crate 使得在 Rust 和 C++ 之间进行安全的互操作成为可能。

整体的方法如下：



请参阅 CXX tutorial ，了解有关使用的完整示例。

▼ *Speaker Notes*

- 此时，讲师应该该切换到 CXX tutorial。
- 逐步引导学生按照教程一步步操作。
- 突出展示 CXX 在 *两种语言* 中都提供了一个没有不安全代码的干净接口。
- 展示 Rust 和 C++类型 之间的对应关系：
  - 解释 Rust 的 `String` 无法直接映射到 C++ 的 `std::string` （后者不符合UTF-8不变性）。展示尽管类型不同，但在 C++ 中，可以很容易地从 C++ 的 `std::string` 构造 `rust::String` ，使得使用起来非常方便。
  - 解释 Rust 中返回 `Result<T, E>` 的函数在 C++ 中会变成抛出 `E` 异常的函数（反之亦然）。

与 Java 的互操作性

Java可以通过 Java 本地接口 (JNI) 加载共享对象。 `jni` crate 允许您创建一个兼容的库。

首先，我们创建一个可以导出到 Java 的 Rust 函数：

interoperability/java/src/lib.rs:

```
//! Rust <-> Java FFI demo.

use jni::objects::{JClass, JString};
use jni::sys::jstring;
use jni::JNIEnv;

/// HelloWorld::hello method implementation.
#[no_mangle]
pub extern "system" fn Java_HelloWorld_hello(
    env: JNIEnv,
    _class: JClass,
    name: JString,
) -> jstring {
    let input: String = env.get_string(name).unwrap().into();
    let greeting = format!("Hello, {input}!");
    let output = env.new_string(greeting).unwrap();
    output.into_inner()
}
```

interoperability/java/Android.bp:

```
rust_ffi_shared {
    name: "libhello_jni",
    crate_name: "hello_jni",
    srcs: ["src/lib.rs"],
    rustlibs: ["libjni"],
}
```

最后，我们可以从 Java 中调用这个函数：

interoperability/java/HelloWorld.java:

```
class HelloWorld {
    private static native String hello(String name);

    static {
        System.loadLibrary("hello_jni");
    }

    public static void main(String[] args) {
        String output = HelloWorld.hello("Alice");
        System.out.println(output);
    }
}
```

interoperability/java/Android.bp:

```
java_binary {
    name: "helloworld_jni",
    srcs: ["HelloWorld.java"],
    main_class: "HelloWorld",
    required: ["libhello_jni"],
}
```

最后，您可以构建、同步和运行二进制文件：

```
m helloworld_jni
adb sync  # requires adb root && adb remount
adb shell /system/bin/helloworld_jni
```

习题

这是一个小组练习：我们将查看你们正在处理的项目之一，并尝试将一些 Rust 代码集成进去。以下是一些建议：

- 使用 Rust 编写的客户端调用你的 AIDL 服务。
- 将你项目中的某个函数迁移到 Rust 中并调用它。

▼ *Speaker Notes*

此处没有提供解决方案，因为这是开放式的：它依赖于班级中是否有人有一段您可以即时转换成 Rust 的代码。

Welcome to Bare Metal Rust

This is a standalone one-day course about bare-metal Rust, aimed at people who are familiar with the basics of Rust (perhaps from completing the Comprehensive Rust course), and ideally also have some experience with bare-metal programming in some other language such as C.

Today we will talk about 'bare-metal' Rust: running Rust code without an OS underneath us. This will be divided into several parts:

- What is `no_std` Rust?
- Writing firmware for microcontrollers.
- Writing bootloader / kernel code for application processors.
- Some useful crates for bare-metal Rust development.

For the microcontroller part of the course we will use the BBC micro:bit v2 as an example. It's a development board based on the Nordic nRF51822 microcontroller with some LEDs and buttons, an I2C-connected accelerometer and compass, and an on-board SWD debugger.

To get started, install some tools we'll need later. On gLinux or Debian:

```
sudo apt install gcc-aarch64-linux-gnu gdb-multiarch libudev-dev picocom pkg-
config qemu-system-arm
rustup update
rustup target add aarch64-unknown-none thumbv7em-none-eabihf
rustup component add llvm-tools-preview
cargo install cargo-binutils cargo-embed
```

And give users in the `plugdev` group access to the micro:bit programmer:

```
echo 'SUBSYSTEM=="usb", ATTR{idVendor}=="0d28", MODE="0664", GROUP="plugdev"'
|\
  sudo tee /etc/udev/rules.d/50-microbit.rules
sudo udevadm control --reload-rules
```

On MacOS:

```
xcode-select --install
brew install gdb picocom qemu
brew install --cask gcc-aarch64-embedded
rustup update
rustup target add aarch64-unknown-none thumbv7em-none-eabihf
rustup component add llvm-tools-preview
cargo install cargo-binutils cargo-embed
```

no_std

core	alloc	std
• Slices, `&str`, `CStr` • `NonZeroU8` ... • `Option`, `Result` • `Display`, `Debug`, `write!` ... • "Iterator" • `panic!`, `assert_eq!` ... • `NonNull` and all the usual pointer-related functions • `Future` and `async` / `await` • `fence`, `AtomicBool`, `AtomicPtr`, `AtomicU32` ... • `Duration`	• `Box`, `Cow`, `Arc`, `Rc` • `Vec`, `BinaryHeap`, `BtreeMap`, `LinkedList`, `VecDeque` • `String`, `CString`, `format!`	• `Error` • `HashMap` • `Mutex`, `Condvar`, `Barrier`, `Once`, `RwLock`, `mpsc` • `File` and the rest of `fs` • `println!`, `Read`, `Write`, `Stdin`, `Stdout` and the rest of `io` • `Path`, `OsString` • `net` • `Command`, `Child`, `ExitCode` • `spawn`, `sleep` and the rest of `thread` • `SystemTime`, `Instant`

▼ *Speaker Notes*

- `HashMap` depends on RNG.
- `std` re-exports the contents of both `core` and `alloc`.

A minimal no_std program

```
#![no_main]
#![no_std]

use core::panic::PanicInfo;

#[panic_handler]
fn panic(_panic: &PanicInfo) -> ! {
    loop {}
}
```

▼ *Speaker Notes*

- This will compile to an empty binary.
- `std` provides a panic handler; without it we must provide our own.
- It can also be provided by another crate, such as `panic-halt`.
- Depending on the target, you may need to compile with `panic = "abort"` to avoid an error about `eh_personality`.
- Note that there is no `main` or any other entry point; it's up to you to define your own entry point. This will typically involve a linker script and some assembly code to set things up ready for Rust code to run.

alloc

To use `alloc` you must implement a global (heap) allocator.

```
#![no_main]
#![no_std]

extern crate alloc;
extern crate panic_halt as _;

use alloc::string::ToString;
use alloc::vec::Vec;
use buddy_system_allocator::LockedHeap;

#[global_allocator]
static HEAP_ALLOCATOR: LockedHeap<32> = LockedHeap::<32>::new();

static mut HEAP: [u8; 65536] = [0; 65536];

pub fn entry() {
    // Safe because `HEAP` is only used here and `entry` is only called once
    unsafe {
        // Give the allocator some memory to allocate.
        HEAP_ALLOCATOR
            .lock()
            .init(HEAP.as_mut_ptr() as usize, HEAP.len());
    }

    // Now we can do things that require heap allocation.
    let mut v = Vec::new();
    v.push("A string".to_string());
}
```

▼ *Speaker Notes*

- `buddy_system_allocator` is a third-party crate implementing a basic buddy system allocator. Other crates are available, or you can write your own or hook into your existing allocator.
- The const parameter of `LockedHeap` is the max order of the allocator; i.e. in this case it can allocate regions of up to 2**32 bytes.
- If any crate in your dependency tree depends on `alloc` then you must have exactly one global allocator defined in your binary. Usually this is done in the top-level binary crate.
- `extern crate panic_halt as _` is necessary to ensure that the `panic_halt` crate is linked in so we get its panic handler.
- This example will build but not run, as it doesn't have an entry point.

微控制器

The `cortex_m_rt` crate provides (among other things) a reset handler for Cortex M microcontrollers.

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

mod interrupts;

use cortex_m_rt::entry;

#[entry]
fn main() -> ! {
    loop {}
}
```

Next we'll look at how to access peripherals, with increasing levels of abstraction.

▼ *Speaker Notes*

- The `cortex_m_rt::entry` macro requires that the function have type `fn() -> !`, because returning to the reset handler doesn't make sense.
- Run the example with `cargo embed --bin minimal`

原始 MMIO

Most microcontrollers access peripherals via memory-mapped IO. Let’s try turning on an LED on our micro:bit:

```rust
#![no_main]
#![no_std]

extern crate panic_halt as _;

mod interrupts;

use core::mem::size_of;
use cortex_m_rt::entry;

/// GPIO port 0 peripheral address
const GPIO_P0: usize = 0x5000_0000;

// GPIO peripheral offsets
const PIN_CNF: usize = 0x700;
const OUTSET: usize = 0x508;
const OUTCLR: usize = 0x50c;

// PIN_CNF fields
const DIR_OUTPUT: u32 = 0x1;
const INPUT_DISCONNECT: u32 = 0x1 << 1;
const PULL_DISABLED: u32 = 0x0 << 2;
const DRIVE_S0S1: u32 = 0x0 << 8;
const SENSE_DISABLED: u32 = 0x0 << 16;

#[entry]
fn main() -> ! {
    // Configure GPIO 0 pins 21 and 28 as push-pull outputs.
    let pin_cnf_21 = (GPIO_P0 + PIN_CNF + 21 * size_of::<u32>()) as *mut u32
    let pin_cnf_28 = (GPIO_P0 + PIN_CNF + 28 * size_of::<u32>()) as *mut u32
    // Safe because the pointers are to valid peripheral control registers,
    // no aliases exist.
    unsafe {
        pin_cnf_21.write_volatile(
            DIR_OUTPUT | INPUT_DISCONNECT | PULL_DISABLED | DRIVE_S0S1 | SEN
        );
        pin_cnf_28.write_volatile(
            DIR_OUTPUT | INPUT_DISCONNECT | PULL_DISABLED | DRIVE_S0S1 | SEN
        );
    }

    // Set pin 28 low and pin 21 high to turn the LED on.
    let gpio0_outset = (GPIO_P0 + OUTSET) as *mut u32;
    let gpio0_outclr = (GPIO_P0 + OUTCLR) as *mut u32;
    // Safe because the pointers are to valid peripheral control registers,
    // no aliases exist.
    unsafe {
        gpio0_outclr.write_volatile(1 << 28);
        gpio0_outset.write_volatile(1 << 21);
    }

    loop {}
}
```

▼ *Speaker Notes*

- GPIO 0 pin 21 is connected to the first column of the LED matrix, and pin 28 to the first row.

Run the example with:

```
cargo embed --bin mmio
```

Peripheral Access Crates

`svd2rust` generates mostly-safe Rust wrappers for memory-mapped peripherals from CMSIS-SVD files.

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

use cortex_m_rt::entry;
use nrf52833_pac::Peripherals;

#[entry]
fn main() -> ! {
    let p = Peripherals::take().unwrap();
    let gpio0 = p.P0;

    // Configure GPIO 0 pins 21 and 28 as push-pull outputs.
    gpio0.pin_cnf[21].write(|w| {
        w.dir().output();
        w.input().disconnect();
        w.pull().disabled();
        w.drive().s0s1();
        w.sense().disabled();
        w
    });
    gpio0.pin_cnf[28].write(|w| {
        w.dir().output();
        w.input().disconnect();
        w.pull().disabled();
        w.drive().s0s1();
        w.sense().disabled();
        w
    });

    // Set pin 28 low and pin 21 high to turn the LED on.
    gpio0.outclr.write(|w| w.pin28().clear());
    gpio0.outset.write(|w| w.pin21().set());

    loop {}
}
```

▼ *Speaker Notes*

- SVD (System View Description) files are XML files typically provided by silicon vendors which describe the memory map of the device.
  - They are organised by peripheral, register, field and value, with names, descriptions, addresses and so on.
  - SVD files are often buggy and incomplete, so there are various projects which patch the mistakes, add missing details, and publish the generated crates.

- `cortex-m-rt` provides the vector table, among other things.
- If you `cargo install cargo-binutils` then you can run `cargo objdump --bin pac -- -d --no-show-raw-insn` to see the resulting binary.

Run the example with:

```
cargo embed --bin pac
```

HAL crates

HAL crates for many microcontrollers provide wrappers around various peripherals. These generally implement traits from `embedded-hal`.

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

use cortex_m_rt::entry;
use nrf52833_hal::gpio::{p0, Level};
use nrf52833_hal::pac::Peripherals;
use nrf52833_hal::prelude::*;

#[entry]
fn main() -> ! {
    let p = Peripherals::take().unwrap();

    // Create HAL wrapper for GPIO port 0.
    let gpio0 = p0::Parts::new(p.P0);

    // Configure GPIO 0 pins 21 and 28 as push-pull outputs.
    let mut col1 = gpio0.p0_28.into_push_pull_output(Level::High);
    let mut row1 = gpio0.p0_21.into_push_pull_output(Level::Low);

    // Set pin 28 low and pin 21 high to turn the LED on.
    col1.set_low().unwrap();
    row1.set_high().unwrap();

    loop {}
}
```

▼ *Speaker Notes*

- `set_low` and `set_high` are methods on the `embedded_hal` `OutputPin` trait.
- HAL crates exist for many Cortex-M and RISC-V devices, including various STM32, GD32, nRF, NXP, MSP430, AVR and PIC microcontrollers.

Run the example with:

```
cargo embed --bin hal
```

Board support crates

Board support crates provide a further level of wrapping for a specific board for convenience.

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

use cortex_m_rt::entry;
use microbit::hal::prelude::*;
use microbit::Board;

#[entry]
fn main() -> ! {
    let mut board = Board::take().unwrap();

    board.display_pins.col1.set_low().unwrap();
    board.display_pins.row1.set_high().unwrap();

    loop {}
}
```

▼ *Speaker Notes*

- In this case the board support crate is just providing more useful names, and a bit of initialisation.
- The crate may also include drivers for some on-board devices outside of the microcontroller itself.
    - `microbit-v2` includes a simple driver for the LED matrix.

Run the example with:

```
cargo embed --bin board_support
```

The type state pattern

```
#[entry]
fn main() -> ! {
    let p = Peripherals::take().unwrap();
    let gpio0 = p0::Parts::new(p.P0);

    let pin: P0_01<Disconnected> = gpio0.p0_01;

    // let gpio0_01_again = gpio0.p0_01; // Error, moved.
    let pin_input: P0_01<Input<Floating>> = pin.into_floating_input();
    if pin_input.is_high().unwrap() {
        // ...
    }
    let mut pin_output: P0_01<Output<OpenDrain>> = pin_input
        .into_open_drain_output(OpenDrainConfig::Disconnect0Standard1, Level
    pin_output.set_high().unwrap();
    // pin_input.is_high(); // Error, moved.

    let _pin2: P0_02<Output<OpenDrain>> = gpio0
        .p0_02
        .into_open_drain_output(OpenDrainConfig::Disconnect0Standard1, Level
    let _pin3: P0_03<Output<PushPull>> = gpio0.p0_03.into_push_pull_output(L

    loop {}
}
```

▼ *Speaker Notes*

- Pins don't implement `Copy` or `Clone`, so only one instance of each can exist. Once a pin is moved out of the port struct nobody else can take it.
- Changing the configuration of a pin consumes the old pin instance, so you can't keep use the old instance afterwards.
- The type of a value indicates the state that it is in: e.g. in this case, the configuration state of a GPIO pin. This encodes the state machine into the type system, and ensures that you don't try to use a pin in a certain way without properly configuring it first. Illegal state transitions are caught at compile time.
- You can call `is_high` on an input pin and `set_high` on an output pin, but not vice-versa.
- Many HAL crates follow this pattern.

`embedded-hal`

The `embedded-hal` crate provides a number of traits covering common microcontroller peripherals.

- GPIO
- ADC
- I2C, SPI, UART, CAN
- RNG
- Timers
- Watchdogs

Other crates then implement drivers in terms of these traits, e.g. an accelerometer driver might need an I2C or SPI bus implementation.

▼ *Speaker Notes*

- There are implementations for many microcontrollers, as well as other platforms such as Linux on Raspberry Pi.
- There is work in progress on an `async` version of `embedded-hal`, but it isn't stable yet.

probe-rs, cargo-embed

probe-rs is a handy toolset for embedded debugging, like OpenOCD but better integrated.

- SWD (Serial Wire Debug) and JTAG via CMSIS-DAP, ST-Link and J-Link probes
- GDB stub and Microsoft DAP (Debug Adapter Protocol) server
- Cargo integration

`cargo-embed` is a cargo subcommand to build and flash binaries, log RTT (Real Time Transfers) output and connect GDB. It's configured by an `Embed.toml` file in your project directory.

▼ *Speaker Notes*

- CMSIS-DAP is an Arm standard protocol over USB for an in-circuit debugger to access the CoreSight Debug Access Port of various Arm Cortex processors. It's what the on-board debugger on the BBC micro:bit uses.
- ST-Link is a range of in-circuit debuggers from ST Microelectronics, J-Link is a range from SEGGER.
- The Debug Access Port is usually either a 5-pin JTAG interface or 2-pin Serial Wire Debug.
- probe-rs is a library which you can integrate into your own tools if you want to.
- The Microsoft Debug Adapter Protocol lets VSCode and other IDEs debug code running on any supported microcontroller.
- cargo-embed is a binary built using the probe-rs library.
- RTT (Real Time Transfers) is a mechanism to transfer data between the debug host and the target through a number of ringbuffers.

调试

Embed.toml:

```
[default.general]
chip = "nrf52833_xxAA"

[debug.gdb]
enabled = true
```

In one terminal under `src/bare-metal/microcontrollers/examples/`:

```
cargo embed --bin board_support debug
```

In another terminal in the same directory:

```
gdb-multiarch target/thumbv7em-none-eabihf/debug/board_support --eval-
command="target remote :1337"
```

▼ *Speaker Notes*

In GDB, try running:

```
b src/bin/board_support.rs:29
b src/bin/board_support.rs:30
b src/bin/board_support.rs:32
c
c
c
```

Other projects

- RTIC
  - “Real-Time Interrupt-driven Concurrency”
  - Shared resource management, message passing, task scheduling, timer queue
- Embassy
  - `async` executors with priorities, timers, networking, USB
- TockOS
  - Security-focused RTOS with preemptive scheduling and Memory Protection Unit support
- Hubris
  - Microkernel RTOS from Oxide Computer Company with memory protection, unprivileged drivers, IPC
- Bindings for FreeRTOS
- Some platforms have `std` implementations, e.g. esp-idf.

▼ *Speaker Notes*

- RTIC can be considered either an RTOS or a concurrency framework.
  - It doesn’t include any HALs.
  - It uses the Cortex-M NVIC (Nested Virtual Interrupt Controller) for scheduling rather than a proper kernel.
  - Cortex-M only.
- Google uses TockOS on the Haven microcontroller for Titan security keys.
- FreeRTOS is mostly written in C, but there are Rust bindings for writing applications.

习题

We will read the direction from an I2C compass, and log the readings to a serial port.

▼ *Speaker Notes*

After looking at the exercises, you can look at the solutions provided.

罗盘

We will read the direction from an I2C compass, and log the readings to a serial port. If you have time, try displaying it on the LEDs somehow too, or use the buttons somehow.

Hints:

- Check the documentation for the `lsm303agr` and `microbit-v2` crates, as well as the micro:bit hardware.
- The LSM303AGR Inertial Measurement Unit is connected to the internal I2C bus.
- TWI is another name for I2C, so the I2C master peripheral is called TWIM.
- The LSM303AGR driver needs something implementing the `embedded_hal::blocking::i2c::WriteRead` trait. The `microbit::hal::Twim` struct implements this.
- You have a `microbit::Board` struct with fields for the various pins and peripherals.
- You can also look at the nRF52833 datasheet if you want, but it shouldn't be necessary for this exercise.

Download the exercise template and look in the `compass` directory for the following files.

src/main.rs:

```rust
#![no_main]
#![no_std]

extern crate panic_halt as _;

use core::fmt::Write;
use cortex_m_rt::entry;
use microbit::{hal::uarte::{Baudrate, Parity, Uarte}, Board};

#[entry]
fn main() -> ! {
    let board = Board::take().unwrap();

    // Configure serial port.
    let mut serial = Uarte::new(
        board.UARTE0,
        board.uart.into(),
        Parity::EXCLUDED,
        Baudrate::BAUD115200,
    );

    // Set up the I2C controller and Inertial Measurement Unit.
    // TODO

    writeln!(serial, "Ready.").unwrap();

    loop {
        // Read compass data and log it to the serial port.
        // TODO
    }
}
```

Cargo.toml (you shouldn't need to change this):

```
[workspace]

[package]
name = "compass"
version = "0.1.0"
edition = "2021"
publish = false

[dependencies]
cortex-m-rt = "0.7.3"
embedded-hal = "0.2.6"
lsm303agr = "0.2.2"
microbit-v2 = "0.13.0"
panic-halt = "0.2.0"
```

Embed.toml (you shouldn't need to change this):

```
[default.general]
chip = "nrf52833_xxAA"

[debug.gdb]
enabled = true

[debug.reset]
halt_afterwards = true
```

.cargo/config.toml (you shouldn't need to change this):

```
[build]
target = "thumbv7em-none-eabihf" # Cortex-M4F

[target.'cfg(all(target_arch = "arm", target_os = "none"))']
rustflags = ["-C", "link-arg=-Tlink.x"]
```

See the serial output on Linux with:

```
picocom --baud 115200 --imap lfcrlf /dev/ttyACM0
```

Or on Mac OS something like (the device name may be slightly different):

```
picocom --baud 115200 --imap lfcrlf /dev/tty.usbmodem14502
```

Use Ctrl+A Ctrl+Q to quit picocom.

Application processors

So far we've talked about microcontrollers, such as the Arm Cortex-M series. Now let's try writing something for Cortex-A. For simplicity we'll just work with QEMU's aarch64 'virt' board.

▼ *Speaker Notes*

- Broadly speaking, microcontrollers don't have an MMU or multiple levels of privilege (exception levels on Arm CPUs, rings on x86), while application processors do.
- QEMU supports emulating various different machines or board models for each architecture. The 'virt' board doesn't correspond to any particular real hardware, but is designed purely for virtual machines.

准备使用 Rust

Before we can start running Rust code, we need to do some initialisation.

```
.section .init.entry, "ax"
.global entry
entry:
    /*
     * Load and apply the memory management configuration, ready to enable MMU
and
     * caches.
     */
    adrp x30, idmap
    msr ttbr0_el1, x30

    mov_i x30, .Lmairval
    msr mair_el1, x30

    mov_i x30, .Ltcrval
    /* Copy the supported PA range into TCR_EL1.IPS. */
    mrs x29, id_aa64mmfr0_el1
    bfi x30, x29, #32, #4

    msr tcr_el1, x30

    mov_i x30, .Lsctlrval

    /*
     * Ensure everything before this point has completed, then invalidate any
     * potentially stale local TLB entries before they start being used.
     */
    isb
    tlbi vmalle1
    ic iallu
    dsb nsh
    isb

    /*
     * Configure sctlr_el1 to enable MMU and cache and don't proceed until this
     * has completed.
     */
    msr sctlr_el1, x30
    isb

    /* Disable trapping floating point access in EL1. */
    mrs x30, cpacr_el1
    orr x30, x30, #(0x3 << 20)
    msr cpacr_el1, x30
    isb

    /* Zero out the bss section. */
    adr_l x29, bss_begin
    adr_l x30, bss_end
0:  cmp x29, x30
    b.hs 1f
    stp xzr, xzr, [x29], #16
    b 0b

1:  /* Prepare the stack. */
    adr_l x30, boot_stack_end
    mov sp, x30
```

```
    /* Set up exception vector. */
    adr x30, vector_table_el1
    msr vbar_el1, x30

    /* Call into Rust code. */
    bl main

    /* Loop forever waiting for interrupts. */
2:  wfi
    b 2b
```

▼ *Speaker Notes*

- This is the same as it would be for C: initialising the processor state, zeroing the BSS, and setting up the stack pointer.
  - The BSS (block starting symbol, for historical reasons) is the part of the object file which containing statically allocated variables which are initialised to zero. They are omitted from the image, to avoid wasting space on zeroes. The compiler assumes that the loader will take care of zeroing them.
- The BSS may already be zeroed, depending on how memory is initialised and the image is loaded, but we zero it to be sure.
- We need to enable the MMU and cache before reading or writing any memory. If we don't:
  - Unaligned accesses will fault. We build the Rust code for the `aarch64-unknown-none` target which sets `+strict-align` to prevent the compiler generating unaligned accesses, so it should be fine in this case, but this is not necessarily the case in general.
  - If it were running in a VM, this can lead to cache coherency issues. The problem is that the VM is accessing memory directly with the cache disabled, while the host has cacheable aliases to the same memory. Even if the host doesn't explicitly access the memory, speculative accesses can lead to cache fills, and then changes from one or the other will get lost when the cache is cleaned or the VM enables the cache. (Cache is keyed by physical address, not VA or IPA.)
- For simplicity, we just use a hardcoded pagetable (see `idmap.S`) which identity maps the first 1 GiB of address space for devices, the next 1 GiB for DRAM, and another 1 GiB higher up for more devices. This matches the memory layout that QEMU uses.
- We also set up the exception vector (`vbar_el1`), which we'll see more about later.
- All examples this afternoon assume we will be running at exception level 1 (EL1). If you need to run at a different exception level you'll need to modify `entry.S` accordingly.

Inline assembly

Sometimes we need to use assembly to do things that aren't possible with Rust code. For example, to make an HVC (hypervisor call) to tell the firmware to power off the system:

```
#![no_main]
#![no_std]

use core::arch::asm;
use core::panic::PanicInfo;

mod exceptions;

const PSCI_SYSTEM_OFF: u32 = 0x84000008;

#[no_mangle]
extern "C" fn main(_x0: u64, _x1: u64, _x2: u64, _x3: u64) {
    // Safe because this only uses the declared registers and doesn't do
    // anything with memory.
    unsafe {
        asm!("hvc #0",
            inout("w0") PSCI_SYSTEM_OFF => _,
            inout("w1") 0 => _,
            inout("w2") 0 => _,
            inout("w3") 0 => _,
            inout("w4") 0 => _,
            inout("w5") 0 => _,
            inout("w6") 0 => _,
            inout("w7") 0 => _,
            options(nomem, nostack)
        );
    }

    loop {}
}
```

(If you actually want to do this, use the `smccc` crate which has wrappers for all these functions.)

▼ *Speaker Notes*

- PSCI is the Arm Power State Coordination Interface, a standard set of functions to manage system and CPU power states, among other things. It is implemented by EL3 firmware and hypervisors on many systems.
- The `0 => _` syntax means initialise the register to 0 before running the inline assembly code, and ignore its contents afterwards. We need to use `inout` rather than `in` because the call could potentially clobber the contents of the registers.
- This `main` function needs to be `#[no_mangle]` and `extern "C"` because it is called from our entry point in `entry.S`.

- `_x0` – `_x3` are the values of registers `x0` – `x3`, which are conventionally used by the bootloader to pass things like a pointer to the device tree. According to the standard aarch64 calling convention (which is what `extern "C"` specifies to use), registers `x0` – `x7` are used for the first 8 arguments passed to a function, so `entry.S` doesn't need to do anything special except make sure it doesn't change these registers.
- Run the example in QEMU with `make qemu_psci` under `src/bare-metal/aps/examples`.

Volatile memory access for MMIO

- Use `pointer::read_volatile` and `pointer::write_volatile`.
- Never hold a reference.
- `addr_of!` lets you get fields of structs without creating an intermediate reference.

▼ *Speaker Notes*

- Volatile access: read or write operations may have side-effects, so prevent the compiler or hardware from reordering, duplicating or eliding them.
  - Usually if you write and then read, e.g. via a mutable reference, the compiler may assume that the value read is the same as the value just written, and not bother actually reading memory.
- Some existing crates for volatile access to hardware do hold references, but this is unsound. Whenever a reference exist, the compiler may choose to dereference it.
- Use the `addr_of!` macro to get struct field pointers from a pointer to the struct.

Let's write a UART driver

The QEMU 'virt' machine has a PL011 UART, so let's write a driver for that.

```
const FLAG_REGISTER_OFFSET: usize = 0x18;
const FR_BUSY: u8 = 1 << 3;
const FR_TXFF: u8 = 1 << 5;

/// Minimal driver for a PL011 UART.
#[derive(Debug)]
pub struct Uart {
    base_address: *mut u8,
}

impl Uart {
    /// Constructs a new instance of the UART driver for a PL011 device at t
    /// given base address.
    ///
    /// # Safety
    ///
    /// The given base address must point to the 8 MMIO control registers of
    /// PL011 device, which must be mapped into the address space of the pro
    /// as device memory and not have any other aliases.
    pub unsafe fn new(base_address: *mut u8) -> Self {
        Self { base_address }
    }

    /// Writes a single byte to the UART.
    pub fn write_byte(&self, byte: u8) {
        // Wait until there is room in the TX buffer.
        while self.read_flag_register() & FR_TXFF != 0 {}

        // Safe because we know that the base address points to the control
        // registers of a PL011 device which is appropriately mapped.
        unsafe {
            // Write to the TX buffer.
            self.base_address.write_volatile(byte);
        }

        // Wait until the UART is no longer busy.
        while self.read_flag_register() & FR_BUSY != 0 {}
    }

    fn read_flag_register(&self) -> u8 {
        // Safe because we know that the base address points to the control
        // registers of a PL011 device which is appropriately mapped.
        unsafe { self.base_address.add(FLAG_REGISTER_OFFSET).read_volatile()
    }
}
```

▼ *Speaker Notes*

- Note that `Uart::new` is unsafe while the other methods are safe. This is because as long as the caller of `Uart::new` guarantees that its safety requirements are met (i.e. that there is only ever one instance of the driver for a given UART, and nothing else aliasing its address space), then it is always safe to call `write_byte` later because we can assume the necessary preconditions.
- We could have done it the other way around (making `new` safe but `write_byte` unsafe), but that would be much less convenient to use as every place that calls `write_byte` would need to reason about the safety
- This is a common pattern for writing safe wrappers of unsafe code: moving the burden of proof for soundness from a large number of places to a smaller number of places.

More traits

We derived the `Debug` trait. It would be useful to implement a few more traits too.

```
use core::fmt::{self, Write};

impl Write for Uart {
    fn write_str(&mut self, s: &str) -> fmt::Result {
        for c in s.as_bytes() {
            self.write_byte(*c);
        }
        Ok(())
    }
}

// Safe because it just contains a pointer to device memory, which can be
// accessed from any context.
unsafe impl Send for Uart {}
```

▼ *Speaker Notes*

- Implementing `Write` lets us use the `write!` and `writeln!` macros with our `Uart` type.
- Run the example in QEMU with `make qemu_minimal` under `src/bare-metal/aps/examples`.

A better UART driver

The PL011 actually has a bunch more registers, and adding offsets to construct pointers to access them is error-prone and hard to read. Plus, some of them are bit fields which would be nice to access in a structured way.

Offset	Register name	Width
0x00	DR	12
0x04	RSR	4
0x18	FR	9
0x20	ILPR	8
0x24	IBRD	16
0x28	FBRD	6
0x2c	LCR_H	8
0x30	CR	16
0x34	IFLS	6
0x38	IMSC	11
0x3c	RIS	11
0x40	MIS	11
0x44	ICR	11
0x48	DMACR	3

▼ *Speaker Notes*

- There are also some ID registers which have been omitted for brevity.

Bitflags

The `bitflags` crate is useful for working with bitflags.

```
use bitflags::bitflags;

bitflags! {
    /// Flags from the UART flag register.
    #[repr(transparent)]
    #[derive(Copy, Clone, Debug, Eq, PartialEq)]
    struct Flags: u16 {
        /// Clear to send.
        const CTS = 1 << 0;
        /// Data set ready.
        const DSR = 1 << 1;
        /// Data carrier detect.
        const DCD = 1 << 2;
        /// UART busy transmitting data.
        const BUSY = 1 << 3;
        /// Receive FIFO is empty.
        const RXFE = 1 << 4;
        /// Transmit FIFO is full.
        const TXFF = 1 << 5;
        /// Receive FIFO is full.
        const RXFF = 1 << 6;
        /// Transmit FIFO is empty.
        const TXFE = 1 << 7;
        /// Ring indicator.
        const RI = 1 << 8;
    }
}
```

▼ *Speaker Notes*

- The `bitflags!` macro creates a newtype something like `Flags(u16)`, along with a bunch of method implementations to get and set flags.

Multiple registers

We can use a struct to represent the memory layout of the UART's registers.

```
#[repr(C, align(4))]
struct Registers {
    dr: u16,
    _reserved0: [u8; 2],
    rsr: ReceiveStatus,
    _reserved1: [u8; 19],
    fr: Flags,
    _reserved2: [u8; 6],
    ilpr: u8,
    _reserved3: [u8; 3],
    ibrd: u16,
    _reserved4: [u8; 2],
    fbrd: u8,
    _reserved5: [u8; 3],
    lcr_h: u8,
    _reserved6: [u8; 3],
    cr: u16,
    _reserved7: [u8; 3],
    ifls: u8,
    _reserved8: [u8; 3],
    imsc: u16,
    _reserved9: [u8; 2],
    ris: u16,
    _reserved10: [u8; 2],
    mis: u16,
    _reserved11: [u8; 2],
    icr: u16,
    _reserved12: [u8; 2],
    dmacr: u8,
    _reserved13: [u8; 3],
}
```

▼ *Speaker Notes*

- `#[repr(C)]` tells the compiler to lay the struct fields out in order, following the same rules as C. This is necessary for our struct to have a predictable layout, as default Rust representation allows the compiler to (among other things) reorder fields however it sees fit.

驱动程序

Now let’s use the new `Registers` struct in our driver.

```rust
/// Driver for a PL011 UART.
#[derive(Debug)]
pub struct Uart {
    registers: *mut Registers,
}

impl Uart {
    /// Constructs a new instance of the UART driver for a PL011 device at t
    /// given base address.
    ///
    /// # Safety
    ///
    /// The given base address must point to the 8 MMIO control registers of
    /// PL011 device, which must be mapped into the address space of the pro
    /// as device memory and not have any other aliases.
    pub unsafe fn new(base_address: *mut u32) -> Self {
        Self {
            registers: base_address as *mut Registers,
        }
    }

    /// Writes a single byte to the UART.
    pub fn write_byte(&self, byte: u8) {
        // Wait until there is room in the TX buffer.
        while self.read_flag_register().contains(Flags::TXFF) {}

        // Safe because we know that self.registers points to the control
        // registers of a PL011 device which is appropriately mapped.
        unsafe {
            // Write to the TX buffer.
            addr_of_mut!((*self.registers).dr).write_volatile(byte.into());
        }

        // Wait until the UART is no longer busy.
        while self.read_flag_register().contains(Flags::BUSY) {}
    }

    /// Reads and returns a pending byte, or `None` if nothing has been rece
    pub fn read_byte(&self) -> Option<u8> {
        if self.read_flag_register().contains(Flags::RXFE) {
            None
        } else {
            let data = unsafe { addr_of!((*self.registers).dr).read_volatile
            // TODO: Check for error conditions in bits 8-11.
            Some(data as u8)
        }
    }

    fn read_flag_register(&self) -> Flags {
        // Safe because we know that self.registers points to the control
        // registers of a PL011 device which is appropriately mapped.
        unsafe { addr_of!((*self.registers).fr).read_volatile() }
    }
}
```

▼ *Speaker Notes*

- Note the use of `addr_of!` / `addr_of_mut!` to get pointers to individual fields without creating an intermediate reference, which would be unsound.

Using it

Let's write a small program using our driver to write to the serial console, and echo incoming bytes.

```
#![no_main]
#![no_std]

mod exceptions;
mod pl011;

use crate::pl011::Uart;
use core::fmt::Write;
use core::panic::PanicInfo;
use log::error;
use smccc::psci::system_off;
use smccc::Hvc;

/// Base address of the primary PL011 UART.
const PL011_BASE_ADDRESS: *mut u32 = 0x900_0000 as _;

#[no_mangle]
extern "C" fn main(x0: u64, x1: u64, x2: u64, x3: u64) {
    // Safe because `PL011_BASE_ADDRESS` is the base address of a PL011 devi
    // and nothing else accesses that address range.
    let mut uart = unsafe { Uart::new(PL011_BASE_ADDRESS) };

    writeln!(uart, "main({x0:#x}, {x1:#x}, {x2:#x}, {x3:#x})").unwrap();

    loop {
        if let Some(byte) = uart.read_byte() {
            uart.write_byte(byte);
            match byte {
                b'\r' => {
                    uart.write_byte(b'\n');
                }
                b'q' => break,
                _ => {}
            }
        }
    }

    writeln!(uart, "Bye!").unwrap();
    system_off::<Hvc>().unwrap();
}
```

▼ *Speaker Notes*

- As in the inline assembly example, this `main` function is called from our entry point code in `entry.S`. See the speaker notes there for details.
- Run the example in QEMU with `make qemu` under `src/bare-metal/aps/examples`.

日志记录

It would be nice to be able to use the logging macros from the `log` crate. We can do this by implementing the `Log` trait.

```
use crate::pl011::Uart;
use core::fmt::Write;
use log::{LevelFilter, Log, Metadata, Record, SetLoggerError};
use spin::mutex::SpinMutex;

static LOGGER: Logger = Logger {
    uart: SpinMutex::new(None),
};

struct Logger {
    uart: SpinMutex<Option<Uart>>,
}

impl Log for Logger {
    fn enabled(&self, _metadata: &Metadata) -> bool {
        true
    }

    fn log(&self, record: &Record) {
        writeln!(
            self.uart.lock().as_mut().unwrap(),
            "[{}] {}",
            record.level(),
            record.args()
        )
        .unwrap();
    }

    fn flush(&self) {}
}

/// Initialises UART logger.
pub fn init(uart: Uart, max_level: LevelFilter) -> Result<(), SetLoggerError
    LOGGER.uart.lock().replace(uart);

    log::set_logger(&LOGGER)?;
    log::set_max_level(max_level);
    Ok(())
}
```

▼ *Speaker Notes*

- The unwrap in `log` is safe because we initialise `LOGGER` before calling `set_logger`.

Using it

We need to initialise the logger before we use it.

```
#![no_main]
#![no_std]

mod exceptions;
mod logger;
mod pl011;

use crate::pl011::Uart;
use core::panic::PanicInfo;
use log::{error, info, LevelFilter};
use smccc::psci::system_off;
use smccc::Hvc;

/// Base address of the primary PL011 UART.
const PL011_BASE_ADDRESS: *mut u32 = 0x900_0000 as _;

#[no_mangle]
extern "C" fn main(x0: u64, x1: u64, x2: u64, x3: u64) {
    // Safe because `PL011_BASE_ADDRESS` is the base address of a PL011 devi
    // and nothing else accesses that address range.
    let uart = unsafe { Uart::new(PL011_BASE_ADDRESS) };
    logger::init(uart, LevelFilter::Trace).unwrap();

    info!("main({x0:#x}, {x1:#x}, {x2:#x}, {x3:#x})");

    assert_eq!(x1, 42);

    system_off::<Hvc>().unwrap();
}

#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    error!("{info}");
    system_off::<Hvc>().unwrap();
    loop {}
}
```

▼ *Speaker Notes*

- Note that our panic handler can now log details of panics.
- Run the example in QEMU with `make qemu_logger` under `src/bare-metal/aps/examples`.

异常

AArch64 defines an exception vector table with 16 entries, for 4 types of exceptions (synchronous, IRQ, FIQ, SError) from 4 states (current EL with SP0, current EL with SPx, lower EL using AArch64, lower EL using AArch32). We implement this in assembly to save volatile registers to the stack before calling into Rust code:

```rust
use log::error;
use smccc::psci::system_off;
use smccc::Hvc;

#[no_mangle]
extern "C" fn sync_exception_current(_elr: u64, _spsr: u64) {
    error!("sync_exception_current");
    system_off::<Hvc>().unwrap();
}

#[no_mangle]
extern "C" fn irq_current(_elr: u64, _spsr: u64) {
    error!("irq_current");
    system_off::<Hvc>().unwrap();
}

#[no_mangle]
extern "C" fn fiq_current(_elr: u64, _spsr: u64) {
    error!("fiq_current");
    system_off::<Hvc>().unwrap();
}

#[no_mangle]
extern "C" fn serr_current(_elr: u64, _spsr: u64) {
    error!("serr_current");
    system_off::<Hvc>().unwrap();
}

#[no_mangle]
extern "C" fn sync_lower(_elr: u64, _spsr: u64) {
    error!("sync_lower");
    system_off::<Hvc>().unwrap();
}

#[no_mangle]
extern "C" fn irq_lower(_elr: u64, _spsr: u64) {
    error!("irq_lower");
    system_off::<Hvc>().unwrap();
}

#[no_mangle]
extern "C" fn fiq_lower(_elr: u64, _spsr: u64) {
    error!("fiq_lower");
    system_off::<Hvc>().unwrap();
}

#[no_mangle]
extern "C" fn serr_lower(_elr: u64, _spsr: u64) {
    error!("serr_lower");
    system_off::<Hvc>().unwrap();
}
```

▼ *Speaker Notes*

- EL is exception level; all our examples this afternoon run in EL1.

- For simplicity we aren't distinguishing between SP0 and SPx for the current EL exceptions, or between AArch32 and AArch64 for the lower EL exceptions.
- For this example we just log the exception and power down, as we don't expect any of them to actually happen.
- We can think of exception handlers and our main execution context more or less like different threads. `Send` and `Sync` will control what we can share between them, just like with threads. For example, if we want to share some value between exception handlers and the rest of the program, and it's `Send` but not `Sync`, then we'll need to wrap it in something like a `Mutex` and put it in a static.

Other projects

- oreboot
  - “coreboot without the C”
  - Supports x86, aarch64 and RISC-V.
  - Relies on LinuxBoot rather than having many drivers itself.
- Rust RaspberryPi OS tutorial
  - Initialisation, UART driver, simple bootloader, JTAG, exception levels, exception handling, page tables
  - Some dodginess around cache maintenance and initialisation in Rust, not necessarily a good example to copy for production code.
- `cargo-call-stack`
  - Static analysis to determine maximum stack usage.

▼ *Speaker Notes*

- The RaspberryPi OS tutorial runs Rust code before the MMU and caches are enabled. This will read and write memory (e.g. the stack). However:
  - Without the MMU and cache, unaligned accesses will fault. It builds with `aarch64-unknown-none` which sets `+strict-align` to prevent the compiler generating unaligned accesses so it should be alright, but this is not necessarily the case in general.
  - If it were running in a VM, this can lead to cache coherency issues. The problem is that the VM is accessing memory directly with the cache disabled, while the host has cacheable aliases to the same memory. Even if the host doesn’t explicitly access the memory, speculative accesses can lead to cache fills, and then changes from one or the other will get lost. Again this is alright in this particular case (running directly on the hardware with no hypervisor), but isn’t a good pattern in general.

实用 crate

We’ll go over a few crates which solve some common problems in bare-metal programming.

zerocopy

The `zerocopy` crate (from Fuchsia) provides traits and macros for safely converting between byte sequences and other types.

```
use zerocopy::AsBytes;

#[repr(u32)]
#[derive(AsBytes, Debug, Default)]
enum RequestType {
    #[default]
    In = 0,
    Out = 1,
    Flush = 4,
}

#[repr(C)]
#[derive(AsBytes, Debug, Default)]
struct VirtioBlockRequest {
    request_type: RequestType,
    reserved: u32,
    sector: u64,
}

fn main() {
    let request = VirtioBlockRequest {
        request_type: RequestType::Flush,
        sector: 42,
        ..Default::default()
    };

    assert_eq!(
        request.as_bytes(),
        &[4, 0, 0, 0, 0, 0, 0, 0, 42, 0, 0, 0, 0, 0, 0, 0]
    );
}
```

This is not suitable for MMIO (as it doesn't use volatile reads and writes), but can be useful for working with structures shared with hardware e.g. by DMA, or sent over some external interface.

▼ *Speaker Notes*

- `FromBytes` can be implemented for types for which any byte pattern is valid, and so can safely be converted from an untrusted sequence of bytes.
- Attempting to derive `FromBytes` for these types would fail, because `RequestType` doesn't use all possible u32 values as discriminants, so not all byte patterns are valid.
- `zerocopy::byteorder` has types for byte-order aware numeric primitives.

- Run the example with `cargo run` under `src/bare-metal/useful-crates/zerocopy-example/`. (It won't run in the Playground because of the crate dependency.)

aarch64-paging

The `aarch64-paging` crate lets you create page tables according to the AArch64 Virtual Memory System Architecture.

```
use aarch64_paging::{
    idmap::IdMap,
    paging::{Attributes, MemoryRegion},
};

const ASID: usize = 1;
const ROOT_LEVEL: usize = 1;

// Create a new page table with identity mapping.
let mut idmap = IdMap::new(ASID, ROOT_LEVEL);
// Map a 2 MiB region of memory as read-only.
idmap.map_range(
    &MemoryRegion::new(0x80200000, 0x80400000),
    Attributes::NORMAL | Attributes::NON_GLOBAL | Attributes::READ_ONLY,
).unwrap();
// Set `TTBR0_EL1` to activate the page table.
idmap.activate();
```

▼ *Speaker Notes*

- For now it only supports EL1, but support for other exception levels should be straightforward to add.
- This is used in Android for the Protected VM Firmware.
- There's no easy way to run this example, as it needs to run on real hardware or under QEMU.

buddy_system_allocator

`buddy_system_allocator` is a third-party crate implementing a basic buddy system allocator. It can be used both for `LockedHeap` implementing `GlobalAlloc` so you can use the standard `alloc` crate (as we saw before), or for allocating other address space. For example, we might want to allocate MMIO space for PCI BARs:

```
use buddy_system_allocator::FrameAllocator;
use core::alloc::Layout;

fn main() {
    let mut allocator = FrameAllocator::<32>::new();
    allocator.add_frame(0x200_0000, 0x400_0000);

    let layout = Layout::from_size_align(0x100, 0x100).unwrap();
    let bar = allocator
        .alloc_aligned(layout)
        .expect("Failed to allocate 0x100 byte MMIO region");
    println!("Allocated 0x100 byte MMIO region at {:#x}", bar);
}
```

▼ *Speaker Notes*

- PCI BARs always have alignment equal to their size.
- Run the example with `cargo run` under `src/bare-metal/useful-crates/allocator-example/`. (It won't run in the Playground because of the crate dependency.)

tinyvec

Sometimes you want something which can be resized like a `Vec`, but without heap allocation. `tinyvec` provides this: a vector backed by an array or slice, which could be statically allocated or on the stack, which keeps track of how many elements are used and panics if you try to use more than are allocated.

```
use tinyvec::{array_vec, ArrayVec};

fn main() {
    let mut numbers: ArrayVec<[u32; 5]> = array_vec!(42, 66);
    println!("{numbers:?}");
    numbers.push(7);
    println!("{numbers:?}");
    numbers.remove(1);
    println!("{numbers:?}");
}
```

▼ *Speaker Notes*

- `tinyvec` requires that the element type implement `Default` for initialisation.
- The Rust Playground includes `tinyvec`, so this example will run fine inline.

spin

`std::sync::Mutex` and the other synchronisation primitives from `std::sync` are not available in `core` or `alloc`. How can we manage synchronisation or interior mutability, such as for sharing state between different CPUs?

The `spin` crate provides spinlock-based equivalents of many of these primitives.

```
use spin::mutex::SpinMutex;

static counter: SpinMutex<u32> = SpinMutex::new(0);

fn main() {
    println!("count: {}", counter.lock());
    *counter.lock() += 2;
    println!("count: {}", counter.lock());
}
```

▼ *Speaker Notes*

- Be careful to avoid deadlock if you take locks in interrupt handlers.
- `spin` also has a ticket lock mutex implementation; equivalents of `RwLock`, `Barrier` and `Once` from `std::sync`; and `Lazy` for lazy initialisation.
- The `once_cell` crate also has some useful types for late initialisation with a slightly different approach to `spin::once::Once`.
- The Rust Playground includes `spin`, so this example will run fine inline.

Android

To build a bare-metal Rust binary in AOSP, you need to use a `rust_ffi_static` Soong rule to build your Rust code, then a `cc_binary` with a linker script to produce the binary itself, and then a `raw_binary` to convert the ELF to a raw binary ready to be run.

```
rust_ffi_static {
    name: "libvmbase_example",
    defaults: ["vmbase_ffi_defaults"],
    crate_name: "vmbase_example",
    srcs: ["src/main.rs"],
    rustlibs: [
        "libvmbase",
    ],
}

cc_binary {
    name: "vmbase_example",
    defaults: ["vmbase_elf_defaults"],
    srcs: [
        "idmap.S",
    ],
    static_libs: [
        "libvmbase_example",
    ],
    linker_scripts: [
        "image.ld",
        ":vmbase_sections",
    ],
}

raw_binary {
    name: "vmbase_example_bin",
    stem: "vmbase_example.bin",
    src: ":vmbase_example",
    enabled: false,
    target: {
        android_arm64: {
            enabled: true,
        },
    },
}
```

vmbase

For VMs running under crosvm on aarch64, the vmbase library provides a linker script and useful defaults for the build rules, along with an entry point, UART console logging and more.

```
#![no_main]
#![no_std]

use vmbase::{main, println};

main!(main);

pub fn main(arg0: u64, arg1: u64, arg2: u64, arg3: u64) {
    println!("Hello world");
}
```

▼ *Speaker Notes*

- The `main!` macro marks your main function, to be called from the `vmbase` entry point.
- The `vmbase` entry point handles console initialisation, and issues a PSCI_SYSTEM_OFF to shutdown the VM if your main function returns.

习题

We will write a driver for the PL031 real-time clock device.

▼ *Speaker Notes*

After looking at the exercises, you can look at the solutions provided.

RTC driver

The QEMU aarch64 virt machine has a PL031 real-time clock at 0x9010000. For this exercise, you should write a driver for it.

1. Use it to print the current time to the serial console. You can use the `chrono` crate for date/time formatting.
2. Use the match register and raw interrupt status to busy-wait until a given time, e.g. 3 seconds in the future. (Call `core::hint::spin_loop` inside the loop.)
3. *Extension if you have time:* Enable and handle the interrupt generated by the RTC match. You can use the driver provided in the `arm-gic` crate to configure the Arm Generic Interrupt Controller.
    - Use the RTC interrupt, which is wired to the GIC as `IntId::spi(2)`.
    - Once the interrupt is enabled, you can put the core to sleep via `arm_gic::wfi()`, which will cause the core to sleep until it receives an interrupt.

Download the exercise template and look in the `rtc` directory for the following files.

src/main.rs:

```rust
#![no_main]
#![no_std]

mod exceptions;
mod logger;
mod pl011;

use crate::pl011::Uart;
use arm_gic::gicv3::GicV3;
use core::panic::PanicInfo;
use log::{error, info, trace, LevelFilter};
use smccc::psci::system_off;
use smccc::Hvc;

/// Base addresses of the GICv3.
const GICD_BASE_ADDRESS: *mut u64 = 0x800_0000 as _;
const GICR_BASE_ADDRESS: *mut u64 = 0x80A_0000 as _;

/// Base address of the primary PL011 UART.
const PL011_BASE_ADDRESS: *mut u32 = 0x900_0000 as _;

#[no_mangle]
extern "C" fn main(x0: u64, x1: u64, x2: u64, x3: u64) {
    // Safe because `PL011_BASE_ADDRESS` is the base address of a PL011 device,
    // and nothing else accesses that address range.
    let uart = unsafe { Uart::new(PL011_BASE_ADDRESS) };
    logger::init(uart, LevelFilter::Trace).unwrap();

    info!("main({:#x}, {:#x}, {:#x}, {:#x})", x0, x1, x2, x3);

    // Safe because `GICD_BASE_ADDRESS` and `GICR_BASE_ADDRESS` are the base
    // addresses of a GICv3 distributor and redistributor respectively, and
    // nothing else accesses those address ranges.
    let mut gic = unsafe { GicV3::new(GICD_BASE_ADDRESS, GICR_BASE_ADDRESS) };
    gic.setup();

    // TODO: Create instance of RTC driver and print current time.

    // TODO: Wait for 3 seconds.

    system_off::<Hvc>().unwrap();
}

#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    error!("{info}");
    system_off::<Hvc>().unwrap();
    loop {}
}
```

src/exceptions.rs (you should only need to change this for the 3rd part of the exercise):

```
// Copyright 2023 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

use arm_gic::gicv3::GicV3;
use log::{error, info, trace};
use smccc::psci::system_off;
use smccc::Hvc;

#[no_mangle]
extern "C" fn sync_exception_current(_elr: u64, _spsr: u64) {
    error!("sync_exception_current");
    system_off::<Hvc>().unwrap();
}

#[no_mangle]
extern "C" fn irq_current(_elr: u64, _spsr: u64) {
    trace!("irq_current");
    let intid = GicV3::get_and_acknowledge_interrupt().expect("No pending
interrupt");
    info!("IRQ {intid:?}");
}

#[no_mangle]
extern "C" fn fiq_current(_elr: u64, _spsr: u64) {
    error!("fiq_current");
    system_off::<Hvc>().unwrap();
}

#[no_mangle]
extern "C" fn serr_current(_elr: u64, _spsr: u64) {
    error!("serr_current");
    system_off::<Hvc>().unwrap();
}

#[no_mangle]
extern "C" fn sync_lower(_elr: u64, _spsr: u64) {
    error!("sync_lower");
    system_off::<Hvc>().unwrap();
}

#[no_mangle]
extern "C" fn irq_lower(_elr: u64, _spsr: u64) {
    error!("irq_lower");
    system_off::<Hvc>().unwrap();
}
```

```
#[no_mangle]
extern "C" fn fiq_lower(_elr: u64, _spsr: u64) {
    error!("fiq_lower");
    system_off::<Hvc>().unwrap();
}

#[no_mangle]
extern "C" fn serr_lower(_elr: u64, _spsr: u64) {
    error!("serr_lower");
    system_off::<Hvc>().unwrap();
}
```

src/logger.rs (you shouldn't need to change this):

```
// Copyright 2023 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

// ANCHOR: main
use crate::pl011::Uart;
use core::fmt::Write;
use log::{LevelFilter, Log, Metadata, Record, SetLoggerError};
use spin::mutex::SpinMutex;

static LOGGER: Logger = Logger {
    uart: SpinMutex::new(None),
};

struct Logger {
    uart: SpinMutex<Option<Uart>>,
}

impl Log for Logger {
    fn enabled(&self, _metadata: &Metadata) -> bool {
        true
    }

    fn log(&self, record: &Record) {
        writeln!(
            self.uart.lock().as_mut().unwrap(),
            "[{}] {}",
            record.level(),
            record.args()
        )
        .unwrap();
    }

    fn flush(&self) {}
}

/// Initialises UART logger.
pub fn init(uart: Uart, max_level: LevelFilter) -> Result<(), SetLoggerError> {
    LOGGER.uart.lock().replace(uart);

    log::set_logger(&LOGGER)?;
    log::set_max_level(max_level);
    Ok(())
}
```

src/pl011.rs (you shouldn't need to change this):

```
// Copyright 2023 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

#![allow(unused)]

use core::fmt::{self, Write};
use core::ptr::{addr_of, addr_of_mut};

// ANCHOR: Flags
use bitflags::bitflags;

bitflags! {
    /// Flags from the UART flag register.
    #[repr(transparent)]
    #[derive(Copy, Clone, Debug, Eq, PartialEq)]
    struct Flags: u16 {
        /// Clear to send.
        const CTS = 1 << 0;
        /// Data set ready.
        const DSR = 1 << 1;
        /// Data carrier detect.
        const DCD = 1 << 2;
        /// UART busy transmitting data.
        const BUSY = 1 << 3;
        /// Receive FIFO is empty.
        const RXFE = 1 << 4;
        /// Transmit FIFO is full.
        const TXFF = 1 << 5;
        /// Receive FIFO is full.
        const RXFF = 1 << 6;
        /// Transmit FIFO is empty.
        const TXFE = 1 << 7;
        /// Ring indicator.
        const RI = 1 << 8;
    }
}
// ANCHOR_END: Flags

bitflags! {
    /// Flags from the UART Receive Status Register / Error Clear Register.
    #[repr(transparent)]
    #[derive(Copy, Clone, Debug, Eq, PartialEq)]
    struct ReceiveStatus: u16 {
        /// Framing error.
        const FE = 1 << 0;
        /// Parity error.
```

```
        const PE = 1 << 1;
        /// Break error.
        const BE = 1 << 2;
        /// Overrun error.
        const OE = 1 << 3;
    }
}

// ANCHOR: Registers
#[repr(C, align(4))]
struct Registers {
    dr: u16,
    _reserved0: [u8; 2],
    rsr: ReceiveStatus,
    _reserved1: [u8; 19],
    fr: Flags,
    _reserved2: [u8; 6],
    ilpr: u8,
    _reserved3: [u8; 3],
    ibrd: u16,
    _reserved4: [u8; 2],
    fbrd: u8,
    _reserved5: [u8; 3],
    lcr_h: u8,
    _reserved6: [u8; 3],
    cr: u16,
    _reserved7: [u8; 3],
    ifls: u8,
    _reserved8: [u8; 3],
    imsc: u16,
    _reserved9: [u8; 2],
    ris: u16,
    _reserved10: [u8; 2],
    mis: u16,
    _reserved11: [u8; 2],
    icr: u16,
    _reserved12: [u8; 2],
    dmacr: u8,
    _reserved13: [u8; 3],
}
// ANCHOR_END: Registers

// ANCHOR: Uart
/// Driver for a PL011 UART.
#[derive(Debug)]
pub struct Uart {
    registers: *mut Registers,
}

impl Uart {
    /// Constructs a new instance of the UART driver for a PL011 device at the
    /// given base address.
    ///
    /// # Safety
    ///
    /// The given base address must point to the MMIO control registers of a
    /// PL011 device, which must be mapped into the address space of the
process
```

```
    /// as device memory and not have any other aliases.
    pub unsafe fn new(base_address: *mut u32) -> Self {
        Self {
            registers: base_address as *mut Registers,
        }
    }

    /// Writes a single byte to the UART.
    pub fn write_byte(&self, byte: u8) {
        // Wait until there is room in the TX buffer.
        while self.read_flag_register().contains(Flags::TXFF) {}

        // Safe because we know that self.registers points to the control
        // registers of a PL011 device which is appropriately mapped.
        unsafe {
            // Write to the TX buffer.
            addr_of_mut!((*self.registers).dr).write_volatile(byte.into());
        }

        // Wait until the UART is no longer busy.
        while self.read_flag_register().contains(Flags::BUSY) {}
    }

    /// Reads and returns a pending byte, or `None` if nothing has been
received.
    pub fn read_byte(&self) -> Option<u8> {
        if self.read_flag_register().contains(Flags::RXFE) {
            None
        } else {
            let data = unsafe { addr_of!((*self.registers).dr).read_volatile()
};
            // TODO: Check for error conditions in bits 8-11.
            Some(data as u8)
        }
    }

    fn read_flag_register(&self) -> Flags {
        // Safe because we know that self.registers points to the control
        // registers of a PL011 device which is appropriately mapped.
        unsafe { addr_of!((*self.registers).fr).read_volatile() }
    }
}
// ANCHOR_END: Uart

impl Write for Uart {
    fn write_str(&mut self, s: &str) -> fmt::Result {
        for c in s.as_bytes() {
            self.write_byte(*c);
        }
        Ok(())
    }
}

// Safe because it just contains a pointer to device memory, which can be
// accessed from any context.
unsafe impl Send for Uart {}
```

Cargo.toml (you shouldn't need to change this):

```
[workspace]

[package]
name = "rtc"
version = "0.1.0"
edition = "2021"
publish = false

[dependencies]
arm-gic = "0.1.0"
bitflags = "2.0.0"
chrono = { version = "0.4.24", default-features = false }
log = "0.4.17"
smccc = "0.1.1"
spin = "0.9.8"

[build-dependencies]
cc = "1.0.73"
```

build.rs (you shouldn't need to change this):

```
// Copyright 2023 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

use cc::Build;
use std::env;

fn main() {
    #[cfg(target_os = "linux")]
    env::set_var("CROSS_COMPILE", "aarch64-linux-gnu");
    #[cfg(not(target_os = "linux"))]
    env::set_var("CROSS_COMPILE", "aarch64-none-elf");

    Build::new()
        .file("entry.S")
        .file("exceptions.S")
        .file("idmap.S")
        .compile("empty")
}
```

entry.S (you shouldn't need to change this):

```
/*
 * Copyright 2023 Google LLC
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

.macro adr_l, reg:req, sym:req
    adrp \reg, \sym
    add \reg, \reg, :lo12:\sym
.endm

.macro mov_i, reg:req, imm:req
    movz \reg, :abs_g3:\imm
    movk \reg, :abs_g2_nc:\imm
    movk \reg, :abs_g1_nc:\imm
    movk \reg, :abs_g0_nc:\imm
.endm

.set .L_MAIR_DEV_nGnRE, 0x04
.set .L_MAIR_MEM_WBWA,  0xff
.set .Lmairval, .L_MAIR_DEV_nGnRE | (.L_MAIR_MEM_WBWA << 8)

/* 4 KiB granule size for TTBR0_EL1. */
.set .L_TCR_TG0_4KB, 0x0 << 14
/* 4 KiB granule size for TTBR1_EL1. */
.set .L_TCR_TG1_4KB, 0x2 << 30
/* Disable translation table walk for TTBR1_EL1, generating a translation fault
instead. */
.set .L_TCR_EPD1, 0x1 << 23
/* Translation table walks for TTBR0_EL1 are inner sharable. */
.set .L_TCR_SH_INNER, 0x3 << 12
/*
 * Translation table walks for TTBR0_EL1 are outer write-back read-allocate
write-allocate
 * cacheable.
 */
.set .L_TCR_RGN_OWB, 0x1 << 10
/*
 * Translation table walks for TTBR0_EL1 are inner write-back read-allocate
write-allocate
 * cacheable.
 */
.set .L_TCR_RGN_IWB, 0x1 << 8
/* Size offset for TTBR0_EL1 is 2**39 bytes (512 GiB). */
.set .L_TCR_T0SZ_512, 64 - 39
.set .Ltcrval, .L_TCR_TG0_4KB | .L_TCR_TG1_4KB | .L_TCR_EPD1 | .L_TCR_RGN_OWB
.set .Ltcrval, .Ltcrval | .L_TCR_RGN_IWB | .L_TCR_SH_INNER | .L_TCR_T0SZ_512
```

```
/* Stage 1 instruction access cacheability is unaffected. */
.set .L_SCTLR_ELx_I, 0x1 << 12
/* SP alignment fault if SP is not aligned to a 16 byte boundary. */
.set .L_SCTLR_ELx_SA, 0x1 << 3
/* Stage 1 data access cacheability is unaffected. */
.set .L_SCTLR_ELx_C, 0x1 << 2
/* EL0 and EL1 stage 1 MMU enabled. */
.set .L_SCTLR_ELx_M, 0x1 << 0
/* Privileged Access Never is unchanged on taking an exception to EL1. */
.set .L_SCTLR_EL1_SPAN, 0x1 << 23
/* SETEND instruction disabled at EL0 in aarch32 mode. */
.set .L_SCTLR_EL1_SED, 0x1 << 8
/* Various IT instructions are disabled at EL0 in aarch32 mode. */
.set .L_SCTLR_EL1_ITD, 0x1 << 7
.set .L_SCTLR_EL1_RES1, (0x1 << 11) | (0x1 << 20) | (0x1 << 22) | (0x1 << 28) |
(0x1 << 29)
.set .Lsctlrval, .L_SCTLR_ELx_M | .L_SCTLR_ELx_C | .L_SCTLR_ELx_SA |
.L_SCTLR_EL1_ITD | .L_SCTLR_EL1_SED
.set .Lsctlrval, .Lsctlrval | .L_SCTLR_ELx_I | .L_SCTLR_EL1_SPAN |
.L_SCTLR_EL1_RES1

/**
 * This is a generic entry point for an image. It carries out the operations
required to prepare the
 * loaded image to be run. Specifically, it zeroes the bss section using
registers x25 and above,
 * prepares the stack, enables floating point, and sets up the exception
vector. It preserves x0-x3
 * for the Rust entry point, as these may contain boot parameters.
 */
.section .init.entry, "ax"
.global entry
entry:
    /* Load and apply the memory management configuration, ready to enable MMU
and caches. */
    adrp x30, idmap
    msr ttbr0_el1, x30

    mov_i x30, .Lmairval
    msr mair_el1, x30

    mov_i x30, .Ltcrval
    /* Copy the supported PA range into TCR_EL1.IPS. */
    mrs x29, id_aa64mmfr0_el1
    bfi x30, x29, #32, #4

    msr tcr_el1, x30

    mov_i x30, .Lsctlrval

    /*
     * Ensure everything before this point has completed, then invalidate any
potentially stale
     * local TLB entries before they start being used.
     */
    isb
    tlbi vmalle1
```

```
    ic iallu
    dsb nsh
    isb

    /*
     * Configure sctlr_el1 to enable MMU and cache and don't proceed until this
has completed.
     */
    msr sctlr_el1, x30
    isb

    /* Disable trapping floating point access in EL1. */
    mrs x30, cpacr_el1
    orr x30, x30, #(0x3 << 20)
    msr cpacr_el1, x30
    isb

    /* Zero out the bss section. */
    adr_l x29, bss_begin
    adr_l x30, bss_end
0:      cmp x29, x30
    b.hs 1f
    stp xzr, xzr, [x29], #16
    b 0b

1:      /* Prepare the stack. */
    adr_l x30, boot_stack_end
    mov sp, x30

    /* Set up exception vector. */
    adr x30, vector_table_el1
    msr vbar_el1, x30

    /* Call into Rust code. */
    bl main

    /* Loop forever waiting for interrupts. */
2:      wfi
    b 2b
```

exceptions.S (you shouldn't need to change this):

```
/*
 * Copyright 2023 Google LLC
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

/**
 * Saves the volatile registers onto the stack. This currently takes 14
 * instructions, so it can be used in exception handlers with 18 instructions
 * left.
 *
 * On return, x0 and x1 are initialised to elr_el2 and spsr_el2 respectively,
 * which can be used as the first and second arguments of a subsequent call.
 */
.macro save_volatile_to_stack
    /* Reserve stack space and save registers x0-x18, x29 & x30. */
    stp x0, x1, [sp, #-(8 * 24)]!
    stp x2, x3, [sp, #8 * 2]
    stp x4, x5, [sp, #8 * 4]
    stp x6, x7, [sp, #8 * 6]
    stp x8, x9, [sp, #8 * 8]
    stp x10, x11, [sp, #8 * 10]
    stp x12, x13, [sp, #8 * 12]
    stp x14, x15, [sp, #8 * 14]
    stp x16, x17, [sp, #8 * 16]
    str x18, [sp, #8 * 18]
    stp x29, x30, [sp, #8 * 20]

    /*
     * Save elr_el1 & spsr_el1. This such that we can take nested exception
     * and still be able to unwind.
     */
    mrs x0, elr_el1
    mrs x1, spsr_el1
    stp x0, x1, [sp, #8 * 22]
.endm

/**
 * Restores the volatile registers from the stack. This currently takes 14
 * instructions, so it can be used in exception handlers while still leaving 18
 * instructions left; if paired with save_volatile_to_stack, there are 4
 * instructions to spare.
 */
.macro restore_volatile_from_stack
    /* Restore registers x2-x18, x29 & x30. */
    ldp x2, x3, [sp, #8 * 2]
    ldp x4, x5, [sp, #8 * 4]
```

```
    ldp x6, x7, [sp, #8 * 6]
    ldp x8, x9, [sp, #8 * 8]
    ldp x10, x11, [sp, #8 * 10]
    ldp x12, x13, [sp, #8 * 12]
    ldp x14, x15, [sp, #8 * 14]
    ldp x16, x17, [sp, #8 * 16]
    ldr x18, [sp, #8 * 18]
    ldp x29, x30, [sp, #8 * 20]

    /* Restore registers elr_el1 & spsr_el1, using x0 & x1 as scratch. */
    ldp x0, x1, [sp, #8 * 22]
    msr elr_el1, x0
    msr spsr_el1, x1

    /* Restore x0 & x1, and release stack space. */
    ldp x0, x1, [sp], #8 * 24
.endm

/**
 * This is a generic handler for exceptions taken at the current EL while using
 * SP0. It behaves similarly to the SPx case by first switching to SPx, doing
 * the work, then switching back to SP0 before returning.
 *
 * Switching to SPx and calling the Rust handler takes 16 instructions. To
 * restore and return we need an additional 16 instructions, so we can
implement
 * the whole handler within the allotted 32 instructions.
 */
.macro current_exception_sp0 handler:req
    msr spsel, #1
    save_volatile_to_stack
    bl \handler
    restore_volatile_from_stack
    msr spsel, #0
    eret
.endm

/**
 * This is a generic handler for exceptions taken at the current EL while using
 * SPx. It saves volatile registers, calls the Rust handler, restores volatile
 * registers, then returns.
 *
 * This also works for exceptions taken from EL0, if we don't care about
 * non-volatile registers.
 *
 * Saving state and jumping to the Rust handler takes 15 instructions, and
 * restoring and returning also takes 15 instructions, so we can fit the whole
 * handler in 30 instructions, under the limit of 32.
 */
.macro current_exception_spx handler:req
    save_volatile_to_stack
    bl \handler
    restore_volatile_from_stack
    eret
.endm

.section .text.vector_table_el1, "ax"
.global vector_table_el1
```

```
.balign 0x800
vector_table_el1:
sync_cur_sp0:
    current_exception_sp0 sync_exception_current

.balign 0x80
irq_cur_sp0:
    current_exception_sp0 irq_current

.balign 0x80
fiq_cur_sp0:
    current_exception_sp0 fiq_current

.balign 0x80
serr_cur_sp0:
    current_exception_sp0 serr_current

.balign 0x80
sync_cur_spx:
    current_exception_spx sync_exception_current

.balign 0x80
irq_cur_spx:
    current_exception_spx irq_current

.balign 0x80
fiq_cur_spx:
    current_exception_spx fiq_current

.balign 0x80
serr_cur_spx:
    current_exception_spx serr_current

.balign 0x80
sync_lower_64:
    current_exception_spx sync_lower

.balign 0x80
irq_lower_64:
    current_exception_spx irq_lower

.balign 0x80
fiq_lower_64:
    current_exception_spx fiq_lower

.balign 0x80
serr_lower_64:
    current_exception_spx serr_lower

.balign 0x80
sync_lower_32:
    current_exception_spx sync_lower

.balign 0x80
irq_lower_32:
    current_exception_spx irq_lower

.balign 0x80
```

```
fiq_lower_32:
    current_exception_spx fiq_lower

.balign 0x80
serr_lower_32:
    current_exception_spx serr_lower
```

idmap.S (you shouldn't need to change this):

```
/*
 * Copyright 2023 Google LLC
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

.set .L_TT_TYPE_BLOCK, 0x1
.set .L_TT_TYPE_PAGE,  0x3
.set .L_TT_TYPE_TABLE, 0x3

/* Access flag. */
.set .L_TT_AF, 0x1 << 10
/* Not global. */
.set .L_TT_NG, 0x1 << 11
.set .L_TT_XN, 0x3 << 53

.set .L_TT_MT_DEV, 0x0 << 2                     // MAIR #0 (DEV_nGnRE)
.set .L_TT_MT_MEM, (0x1 << 2) | (0x3 << 8)      // MAIR #1 (MEM_WBWA), inner
shareable

.set .L_BLOCK_DEV, .L_TT_TYPE_BLOCK | .L_TT_MT_DEV | .L_TT_AF | .L_TT_XN
.set .L_BLOCK_MEM, .L_TT_TYPE_BLOCK | .L_TT_MT_MEM | .L_TT_AF | .L_TT_NG

.section ".rodata.idmap", "a", %progbits
.global idmap
.align 12
idmap:
    /* level 1 */
    .quad               .L_BLOCK_DEV | 0x0                  // 1 GiB of device
mappings
    .quad               .L_BLOCK_MEM | 0x40000000       // 1 GiB of DRAM
    .fill               254, 8, 0x0                     // 254 GiB of unmapped
VA space
    .quad               .L_BLOCK_DEV | 0x4000000000 // 1 GiB of device mappings
    .fill               255, 8, 0x0                     // 255 GiB of remaining
VA space
```

image.ld (you shouldn’t need to change this):

```
/*
 * Copyright 2023 Google LLC
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

/*
 * Code will start running at this symbol which is placed at the start of the
 * image.
 */
ENTRY(entry)

MEMORY
{
        image : ORIGIN = 0x40080000, LENGTH = 2M
}

SECTIONS
{
        /*
         * Collect together the code.
         */
        .init : ALIGN(4096) {
                text_begin = .;
                *(.init.entry)
                *(.init.*)
        } >image
        .text : {
                *(.text.*)
        } >image
        text_end = .;

        /*
         * Collect together read-only data.
         */
        .rodata : ALIGN(4096) {
                rodata_begin = .;
                *(.rodata.*)
        } >image
        .got : {
                *(.got)
        } >image
        rodata_end = .;

        /*
         * Collect together the read-write data including .bss at the end which
         * will be zero'd by the entry code.
```

```
         */
        .data : ALIGN(4096) {
                data_begin = .;
                *(.data.*)
                /*
                 * The entry point code assumes that .data is a multiple of 32
                 * bytes long.
                 */
                . = ALIGN(32);
                data_end = .;
        } >image

        /* Everything beyond this point will not be included in the binary. */
        bin_end = .;

        /* The entry point code assumes that .bss is 16-byte aligned. */
        .bss : ALIGN(16)  {
                bss_begin = .;
                *(.bss.*)
                *(COMMON)
                . = ALIGN(16);
                bss_end = .;
        } >image

        .stack (NOLOAD) : ALIGN(4096) {
                boot_stack_begin = .;
                . += 40 * 4096;
                . = ALIGN(4096);
                boot_stack_end = .;
        } >image

        . = ALIGN(4K);
        PROVIDE(dma_region = .);

        /*
         * Remove unused sections from the image.
         */
        /DISCARD/ : {
                /* The image loads itself so doesn't need these sections. */
                *(.gnu.hash)
                *(.hash)
                *(.interp)
                *(.eh_frame_hdr)
                *(.eh_frame)
                *(.note.gnu.build-id)
        }
}
```

Makefile (you shouldn't need to change this):

```
# Copyright 2023 Google LLC
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

UNAME := $(shell uname -s)
ifeq ($(UNAME),Linux)
    TARGET = aarch64-linux-gnu
else
    TARGET = aarch64-none-elf
endif
OBJCOPY = $(TARGET)-objcopy

.PHONY: build qemu_minimal qemu qemu_logger

all: rtc.bin

build:
    cargo build

rtc.bin: build
    $(OBJCOPY) -O binary target/aarch64-unknown-none/debug/rtc $@

qemu: rtc.bin
    qemu-system-aarch64 -machine virt,gic-version=3 -cpu max -serial mon:stdio
-display none -kernel $< -s

clean:
    cargo clean
    rm -f *.bin
```

.cargo/config.toml (you shouldn't need to change this):

```
[build]
target = "aarch64-unknown-none"
rustflags = ["-C", "link-arg=-Timage.ld"]
```

Run the code in QEMU with `make qemu`.

欢迎了解 Rust 中的并发

Rust 完全支持使用带有互斥锁和通道的操作系统线程进行并发。

Rust 类型系统能帮助我们把许多并发bug转换为编译期bug 发挥着重要作用。这通常称为“无畏并发”，因为你可以依靠编译器来确保 运行时的正确性。

线程

Rust 线程的运作方式与其他语言中的线程类似：

```rust
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("Count in thread: {i}!");
            thread::sleep(Duration::from_millis(5));
        }
    });

    for i in 1..5 {
        println!("Main thread: {i}");
        thread::sleep(Duration::from_millis(5));
    }
}
```

- 线程均为守护程序线程，主线程不会等待这些线程。
- 线程紧急警报 (panic) 是彼此独立的。
    - 紧急警报可以携带载荷，并可以使用 `downcast_ref` 对载荷进行解压缩。

▼ *Speaker Notes*

关键点：

- 请注意，线程在达到 10 之前就停止了，而主线程并 没有等待。
- 使用 `let handle = thread::spawn(...)` 和后面的 `handle.join()` 等待 线程完成。
- 在线程中触发紧急警报，并注意这为何不会影响到 `main` 。
- 使用 `handle.join()` 的 `Result` 返回值来获取对紧急警报 载荷的访问权限。现在有必要介绍一下 `Any` 了。

范围线程

常规线程不能从它们所处的环境中借用：

```
use std::thread;

fn foo() {
    let s = String::from("Hello");
    thread::spawn(|| {
        println!("Length: {}", s.len());
    });
}

fn main() {
    foo();
}
```

不过，你可以使用范围线程来实现此目的：

```
use std::thread;

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

    thread::scope(|scope| {
        scope.spawn(|| {
            println!("Length: {}", s.len());
        });
    });
}
```

▼ *Speaker Notes*

- 其原因在于，在 `thread::scope` 函数完成后，可保证所有线程都已联结在一起，使得线程能够返回借用的数据。
- 此时须遵守常规 Rust 借用规则：你可以通过一个线程以可变的方式借用，也可以通过任意数量的线程以不可变的方式借用。

通道

Rust 通道（Channel）包含两个部分： `Sender<T>` 和 `Receiver<T>` 。这两个部分 通过通道进行连接，但你只能看到端点。

```
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    tx.send(10).unwrap();
    tx.send(20).unwrap();

    println!("Received: {:?}", rx.recv());
    println!("Received: {:?}", rx.recv());

    let tx2 = tx.clone();
    tx2.send(30).unwrap();
    println!("Received: {:?}", rx.recv());
}
```

▼ *Speaker Notes*

- `mpsc` 代表多个生产方，单个使用方。 `Sender` 和 `SyncSender` 会实现 `Clone` （因此， 你可以设置多个生产方），但 `Receiver` 不会实现。
- `send()` 和 `recv()` 会返回 `Result` 。如果它们返回 `Err` ，则表示对应的 `Sender` 或 `Receiver` 已被丢弃，且通道已关闭。

无界通道

你可以使用 `mpsc::channel()` 获得无边界的异步通道：

```rust
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let thread_id = thread::current().id();
        for i in 1..10 {
            tx.send(format!("Message {i}")).unwrap();
            println!("{thread_id:?}: sent Message {i}");
        }
        println!("{thread_id:?}: done");
    });
    thread::sleep(Duration::from_millis(100));

    for msg in rx.iter() {
        println!("Main: got {msg}");
    }
}
```

有界通道

With bounded (synchronous) channels, `send` can block the current thread:

```
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::sync_channel(3);

    thread::spawn(move || {
        let thread_id = thread::current().id();
        for i in 1..10 {
            tx.send(format!("Message {i}")).unwrap();
            println!("{thread_id:?}: sent Message {i}");
        }
        println!("{thread_id:?}: done");
    });
    thread::sleep(Duration::from_millis(100));

    for msg in rx.iter() {
        println!("Main: got {msg}");
    }
}
```

▼ *Speaker Notes*

- Calling `send` will block the current thread until there is space in the channel for the new message. The thread can be blocked indefinitely if there is nobody who reads from the channel.
- A call to `send` will abort with an error (that is why it returns `Result`) if the channel is closed. A channel is closed when the receiver is dropped.
- A bounded channel with a size of zero is called a “rendezvous channel”. Every send will block the current thread until another thread calls `read`.

Send 和 Sync

How does Rust know to forbid shared access across threads? The answer is in two traits:

- `Send`：如果跨线程边界移动 `T` 是安全的，则类型 `T` 为 `Send`。
- `Sync`：如果跨线程边界移动 `&T` 是安全的，则类型 `T` 为 `Sync`。

`Send` 和 `Sync` 均为不安全特征。只要类型仅包含 `Send` 和 `Sync` 类型，编译器就会自动为类型派生 这两种特征。你也可以手动实现它们（如果你确定这样 有效的话）。

▼ *Speaker Notes*

- 不妨将这些特征视为类型包含某些线程安全属性的标记。
- 它们可以在泛型约束中作为常规特征使用。

Send

> 如果将 `T` 值移动到另一个线程是安全的，则类型 `T` 为 `Send` 。

将所有权转移到另一个线程的影响是，“析构函数”将在相应线程中 运行。因此，问题在于你何时可以在一个线程中分配某个值，然后在 另一个线程中取消分配该值。

▼ *Speaker Notes*

例如，与 SQLite 库的连接只能通过 单个线程访问。

Sync

> 如果同时从多个线程访问 `T` 值是安全的，则类型 `T` 为 `Sync` 。

更准确地说，定义是：

> 当且仅当 `&T` 为 `Send` 时， `T` 为 `Sync`

▼ *Speaker Notes*

该语句实质上是一种简写形式，表示如果某个类型对于共享使用是线程安全的，那么跨线程传递对该类型的引用也是线程安全的。

这是因为如果某个类型为 Sync，则意味着它可以在多个线程之间共享，而不存在数据争用或其他同步问题的风险，因此将其移动到另一个线程是安全的。对该类型的引用同样可以安全地移动到另一个线程，因为它引用的数据可以从任何线程安全地访问。

示例

Send + Sync

你遇到的类型大都属于 `Send + Sync`：

- `i8`、`f32`、`bool`、`char`、`&str` ...
- `(T1, T2)`、`[T; N]`、`&[T]`、`struct { x: T }` ...
- `String`、`Option<T>`、`Vec<T>`、`Box<T>` ...
- `Arc<T>`：明确通过原子引用计数实现线程安全。
- `Mutex<T>`：明确通过内部锁定实现线程安全。
- `AtomicBool`、`AtomicU8` ...：使用特殊的原子指令。

当类型参数为 `Send + Sync` 时，泛型类型通常 为 `Send + Sync` 。

Send + !Sync

这些类型可以移动到其他线程，但它们不是线程安全的。 这通常是由内部可变性造成的：

- `mpsc::Sender<T>`
- `mpsc::Receiver<T>`
- `Cell<T>`
- `RefCell<T>`

!Send + Sync

这些类型是线程安全的，但它们不能移动到另一个线程：

- `MutexGuard<T>`：使用操作系统级别的原语（必须在创建这些原语的线程上 取消分配）。

!Send + !Sync

这些类型不是线程安全的，不能移动到其他线程：

- `Rc<T>`：每个 `Rc<T>` 都具有对 `RcBox<T>` 的引用，其中包含 非原子引用计数。
- `*const T`、`*mut T`：Rust 会假定原始指针可能 在并发方面有特殊的注意事项。

共享状态

Rust 使用类型系统来强制同步共享数据。这主要 通过两种类型实现：

- `Arc<T>` ，对 `T` 进行原子计数：用于处理线程之间的共享，并负责 在最后一个引用被丢弃时取消分配 `T` 。
- `Mutex<T>` ：确保对 `T` 值的互斥访问。

Arc

`Arc<T>` 允许通过 `Arc::clone` 实现共享只读权限：

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

fn main() {
    let v = Arc::new(vec![10, 20, 30]);
    let mut handles = Vec::new();
    for _ in 1..5 {
        let v = Arc::clone(&v);
        handles.push(thread::spawn(move || {
            let thread_id = thread::current().id();
            println!("{thread_id:?}: {v:?}");
        }));
    }

    handles.into_iter().for_each(|h| h.join().unwrap());
    println!("v: {v:?}");
}
```

▼ *Speaker Notes*

- `Arc` 代表“原子引用计数”，它是使用原子操作的 `Rc` 的 线程安全版本。
- `Arc<T>` implements `Clone` whether or not `T` does. It implements `Send` and `Sync` if and only if `T` implements them both.
- `Arc::clone()` 在执行原子操作方面有开销，但在此之后， `T` 便可 随意使用，而没有任何开销。
- 请警惕引用循环， `Arc` 不会使用垃圾回收器检测引用循环。
  - `std::sync::Weak` 对此有所帮助。

互斥器 （Mutex）

`Mutex<T>` 能够确保互斥，并允许对只读接口 后面的 `T` 进行可变访问：

```
use std::sync::Mutex;

fn main() {
    let v = Mutex::new(vec![10, 20, 30]);
    println!("v: {:?}", v.lock().unwrap());

    {
        let mut guard = v.lock().unwrap();
        guard.push(40);
    }

    println!("v: {:?}", v.lock().unwrap());
}
```

请注意我们如何设置 `impl<T: Send> Sync for Mutex<T>` 通用 实现。

▼ *Speaker Notes*

- Rust 中的互斥器看起来就像只包含一个元素的集合，其中的元素就是受保护的数据。
  - 在访问受保护的数据之前不可能忘记获取互斥量。
- 你可以通过获取锁，从 `&Mutex<T>` 中获取 `&mut T` 。 `MutexGuard` 能够确保 `&mut T` 存在的时间不会比持有锁的时间更长。
- `Mutex<T>` implements both `Send` and `Sync` iff (if and only if) `T` implements `Send` .
- 读写锁版本 - `RwLock` 。
- 为什么 `lock()` 会返回 `Result` ？
  - 如果持有 `Mutex` 的线程发生panic， `Mutex` 便会“中毒”并发出信号， 表明其所保护的数据可能处于不一致状态。对中毒的互斥量调用 `lock()` 将会失败， 并将显示 `PoisonError` 。无论如何，你可以对该错误调用 `into_inner()` 来 恢复数据。

示例

让我们看看 `Arc` 和 `Mutex` 的实际效果：

```
use std::thread;
// use std::sync::{Arc, Mutex};

fn main() {
    let v = vec![10, 20, 30];
    let handle = thread::spawn(|| {
        v.push(10);
    });
    v.push(1000);

    handle.join().unwrap();
    println!("v: {v:?}");
}
```

▼ *Speaker Notes*

可能有用的解决方案：

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let v = Arc::new(Mutex::new(vec![10, 20, 30]));

    let v2 = Arc::clone(&v);
    let handle = thread::spawn(move || {
        let mut v2 = v2.lock().unwrap();
        v2.push(10);
    });

    {
        let mut v = v.lock().unwrap();
        v.push(1000);
    }

    handle.join().unwrap();

    println!("v: {v:?}");
}
```

值得注意的部分：

- `Arc` 和 `Mutex` 中都封装了 `v`，因为它们的关注点是正交的。
  - 将 `Mutex` 封装在 `Arc` 中是一种在线程之间共享可变状态的常见模式。
- `v: Arc<_>` 必须先克隆为 `v2`，然后才能移动到另一个线程中。请注意，lambda 签名中添加了 `move`。

- 我们引入了块，以尽可能缩小 `LockGuard` 的作用域。

习题

Let us practice our new concurrency skills with

- Dining philosophers: a classic problem in concurrency.
- Multi-threaded link checker: a larger project where you'll use Cargo to download dependencies and then check links in parallel.

▼ *Speaker Notes*

After looking at the exercises, you can look at the solutions provided.

哲学家就餐问题 (Dining philosophers problem)

The dining philosophers problem is a classic problem in concurrency:

> Five philosophers dine together at the same table. Each philosopher has their own place at the table. There is a fork between each plate. The dish served is a kind of spaghetti which has to be eaten with two forks. Each philosopher can only alternately think and eat. Moreover, a philosopher can only eat their spaghetti when they have both a left and right fork. Thus two forks will only be available when their two nearest neighbors are thinking, not eating. After an individual philosopher finishes eating, they will put down both forks.

You will need a local Cargo installation for this exercise. Copy the code below to a file called `src/main.rs`, fill out the blanks, and test that `cargo run` does not deadlock:

```
use std::sync::{mpsc, Arc, Mutex};
use std::thread;
use std::time::Duration;

struct Fork;

struct Philosopher {
    name: String,
    // left_fork: ...
    // right_fork: ...
    // thoughts: ...
}

impl Philosopher {
    fn think(&self) {
        self.thoughts
            .send(format!("Eureka! {} has a new idea!", &self.name))
            .unwrap();
    }

    fn eat(&self) {
        // Pick up forks...
        println!("{} is eating...", &self.name);
        thread::sleep(Duration::from_millis(10));
    }
}

static PHILOSOPHERS: &[&str] =
    &["Socrates", "Plato", "Aristotle", "Thales", "Pythagoras"];

fn main() {
    // Create forks

    // Create philosophers

    // Make each of them think and eat 100 times

    // Output their thoughts
}
```

You can use the following `Cargo.toml`:

```
[package]
name = "dining-philosophers"
version = "0.1.0"
edition = "2021"
```

多线程链接检查器

Let us use our new knowledge to create a multi-threaded link checker. It should start at a webpage and check that links on the page are valid. It should recursively check other pages on the same domain and keep doing this until all pages have been validated.

For this, you will need an HTTP client such as `reqwest`. Create a new Cargo project and `reqwest` it as a dependency with:

```
cargo new link-checker
cd link-checker
cargo add --features blocking,rustls-tls reqwest
```

> If `cargo add` fails with `error: no such subcommand`, then please edit the `Cargo.toml` file by hand. Add the dependencies listed below.

You will also need a way to find links. We can use `scraper` for that:

```
cargo add scraper
```

Finally, we'll need some way of handling errors. We use `thiserror` for that:

```
cargo add thiserror
```

The `cargo add` calls will update the `Cargo.toml` file to look like this:

```
[package]
name = "link-checker"
version = "0.1.0"
edition = "2021"
publish = false

[dependencies]
reqwest = { version = "0.11.12", features = ["blocking", "rustls-tls"] }
scraper = "0.13.0"
thiserror = "1.0.37"
```

You can now download the start page. Try with a small site such as `https://www.google.org/`.

Your `src/main.rs` file should look something like this:

```
use reqwest::{blocking::Client, Url};
use scraper::{Html, Selector};
use thiserror::Error;

#[derive(Error, Debug)]
enum Error {
    #[error("request error: {0}")]
    ReqwestError(#[from] reqwest::Error),
    #[error("bad http response: {0}")]
    BadResponse(String),
}

#[derive(Debug)]
struct CrawlCommand {
    url: Url,
    extract_links: bool,
}

fn visit_page(client: &Client, command: &CrawlCommand) -> Result<Vec<Url>,
Error> {
    println!("Checking {:#}", command.url);
    let response = client.get(command.url.clone()).send()?;
    if !response.status().is_success() {
        return Err(Error::BadResponse(response.status().to_string()));
    }

    let mut link_urls = Vec::new();
    if !command.extract_links {
        return Ok(link_urls);
    }

    let base_url = response.url().to_owned();
    let body_text = response.text()?;
    let document = Html::parse_document(&body_text);

    let selector = Selector::parse("a").unwrap();
    let href_values = document
        .select(&selector)
        .filter_map(|element| element.value().attr("href"));
    for href in href_values {
        match base_url.join(href) {
            Ok(link_url) => {
                link_urls.push(link_url);
            }
            Err(err) => {
                println!("On {base_url:#}: ignored unparsable {href:?}:
{err}");
            }
        }
    }
    Ok(link_urls)
}

fn main() {
    let client = Client::new();
    let start_url = Url::parse("https://www.google.org").unwrap();
    let crawl_command = CrawlCommand{ url: start_url, extract_links: true };
```

```
    match visit_page(&client, &crawl_command) {
        Ok(links) => println!("Links: {links:#?}"),
        Err(err) => println!("Could not extract links: {err:#}"),
    }
}
```

Run the code in `src/main.rs` with

```
cargo run
```

任务

- Use threads to check the links in parallel: send the URLs to be checked to a channel and let a few threads check the URLs in parallel.
- Extend this to recursively extract links from all pages on the `www.google.org` domain. Put an upper limit of 100 pages or so so that you don't end up being blocked by the site.

Async Rust

“Async” is a concurrency model where multiple tasks are executed concurrently by executing each task until it would block, then switching to another task that is ready to make progress. The model allows running a larger number of tasks on a limited number of threads. This is because the per-task overhead is typically very low and operating systems provide primitives for efficiently identifying I/O that is able to proceed.

Rust’s asynchronous operation is based on “futures”, which represent work that may be completed in the future. Futures are “polled” until they signal that they are complete.

Futures are polled by an async runtime, and several different runtimes are available.

Comparisons

- Python has a similar model in its `asyncio`. However, its `Future` type is callback-based, and not polled. Async Python programs require a “loop”, similar to a runtime in Rust.
- JavaScript’s `Promise` is similar, but again callback-based. The language runtime implements the event loop, so many of the details of Promise resolution are hidden.

async/await

从高层次上看，异步 Rust 代码与“正常”的顺序代码非常类似：

```
use futures::executor::block_on;

async fn count_to(count: i32) {
    for i in 1..=count {
        println!("Count is: {i}!");
    }
}

async fn async_main(count: i32) {
    count_to(count).await;
}

fn main() {
    block_on(async_main(10));
}
```

▼ *Speaker Notes*

关键点：

- 请注意，这只是一个简单的示例，用于展示语法。其中没有长时间运行的操作或任何真正的并发！
- 异步调用的返回类型是什么？
    - 在 `main` 中使用 `let future: () = async_main(10);` 来查看类型。
- “async” 关键字是语法糖。编译器会将返回类型替换为 future。
- 你不能将 `main` 声明为异步函数，除非在编译器中加入额外的指令来告诉它如何使用返回的 future。
- 你需要一个执行器来运行异步代码。 `block_on` 会阻塞当前线程，直到提供的future完成为止。
- `.await` 会异步地等待另一个操作的完成。与 `block_on` 不同， `.await` 不会阻塞当前线程。
- `.await` 只能在 `async` 函数（或块，这些稍后会介绍）中使用。

Futures

`Future` is a trait, implemented by objects that represent an operation that may not be complete yet. A future can be polled, and `poll` returns a `Poll`.

```
use std::pin::Pin;
use std::task::Context;

pub trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}

pub enum Poll<T> {
    Ready(T),
    Pending,
}
```

An async function returns an `impl Future`. It's also possible (but uncommon) to implement `Future` for your own types. For example, the `JoinHandle` returned from `tokio::spawn` implements `Future` to allow joining to it.

The `.await` keyword, applied to a Future, causes the current async function to pause until that Future is ready, and then evaluates to its output.

▼ *Speaker Notes*

- The `Future` and `Poll` types are implemented exactly as shown; click the links to show the implementations in the docs.
- We will not get to `Pin` and `Context`, as we will focus on writing async code, rather than building new async primitives. Briefly:
  - `Context` allows a Future to schedule itself to be polled again when an event occurs.
  - `Pin` ensures that the Future isn't moved in memory, so that pointers into that future remain valid. This is required to allow references to remain valid after an `.await`.

Runtimes

A *runtime* provides support for performing operations asynchronously (a *reactor*) and is responsible for executing futures (an *executor*). Rust does not have a "built-in" runtime, but several options are available:

- Tokio: performant, with a well-developed ecosystem of functionality like Hyper for HTTP or Tonic for gRPC.
- async-std: aims to be a "std for async", and includes a basic runtime in `async::task`.
- smol: simple and lightweight

Several larger applications have their own runtimes. For example, Fuchsia already has one.

▼ *Speaker Notes*

- Note that of the listed runtimes, only Tokio is supported in the Rust playground. The playground also does not permit any I/O, so most interesting async things can't run in the playground.
- Futures are "inert" in that they do not do anything (not even start an I/O operation) unless there is an executor polling them. This differs from JS Promises, for example, which will run to completion even if they are never used.

Tokio

Tokio provides:

- A multi-threaded runtime for executing asynchronous code.
- An asynchronous version of the standard library.
- A large ecosystem of libraries.

```
use tokio::time;

async fn count_to(count: i32) {
    for i in 1..=count {
        println!("Count in task: {i}!");
        time::sleep(time::Duration::from_millis(5)).await;
    }
}

#[tokio::main]
async fn main() {
    tokio::spawn(count_to(10));

    for i in 1..5 {
        println!("Main task: {i}");
        time::sleep(time::Duration::from_millis(5)).await;
    }
}
```

▼ *Speaker Notes*

- With the `tokio::main` macro we can now make `main` async.
- The `spawn` function creates a new, concurrent "task".
- Note: `spawn` takes a `Future`, you don't call `.await` on `count_to`.

Further exploration:

- Why does `count_to` not (usually) get to 10? This is an example of async cancellation. `tokio::spawn` returns a handle which can be awaited to wait until it finishes.
- Try `count_to(10).await` instead of spawning.
- Try awaiting the task returned from `tokio::spawn`.

任务

Rust has a task system, which is a form of lightweight threading.

A task has a single top-level future which the executor polls to make progress. That future may have one or more nested futures that its `poll` method polls, corresponding loosely to a call stack. Concurrency within a task is possible by polling multiple child futures, such as racing a timer and an I/O operation.

```
use tokio::io::{self, AsyncReadExt, AsyncWriteExt};
use tokio::net::TcpListener;

#[tokio::main]
async fn main() -> io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:6142").await?;
    println!("listening on port 6142");

    loop {
        let (mut socket, addr) = listener.accept().await?;

        println!("connection from {addr:?}");

        tokio::spawn(async move {
            if let Err(e) = socket.write_all(b"Who are you?\n").await {
                println!("socket error: {e:?}");
                return;
            }

            let mut buf = vec![0; 1024];
            let reply = match socket.read(&mut buf).await {
                Ok(n) => {
                    let name = std::str::from_utf8(&buf[..n]).unwrap().trim();
                    format!("Thanks for dialing in, {name}!\n")
                }
                Err(e) => {
                    println!("socket error: {e:?}");
                    return;
                }
            };

            if let Err(e) = socket.write_all(reply.as_bytes()).await {
                println!("socket error: {e:?}");
            }
        });
    }
}
```

▼ *Speaker Notes*

Copy this example into your prepared `src/main.rs` and run it from there.

- Ask students to visualize what the state of the example server would be with a few connected clients. What tasks exist? What are their Futures?
- This is the first time we've seen an `async` block. This is similar to a closure, but does not take any arguments. Its return value is a Future, similar to an `async fn`.
- Refactor the async block into a function, and improve the error handling using `?`.

异步通道

Several crates have support for asynchronous channels. For instance `tokio`:

```
use tokio::sync::mpsc::{self, Receiver};

async fn ping_handler(mut input: Receiver<()>) {
    let mut count: usize = 0;

    while let Some(_) = input.recv().await {
        count += 1;
        println!("Received {count} pings so far.");
    }

    println!("ping_handler complete");
}

#[tokio::main]
async fn main() {
    let (sender, receiver) = mpsc::channel(32);
    let ping_handler_task = tokio::spawn(ping_handler(receiver));
    for i in 0..10 {
        sender.send(()).await.expect("Failed to send ping.");
        println!("Sent {} pings so far.", i + 1);
    }

    drop(sender);
    ping_handler_task.await.expect("Something went wrong in ping handler tas
}
```

▼ *Speaker Notes*

- Change the channel size to `3` and see how it affects the execution.
- Overall, the interface is similar to the `sync` channels as seen in the morning class.
- Try removing the `std::mem::drop` call. What happens? Why?
- The Flume crate has channels that implement both `sync` and `async` `send` and `recv`. This can be convenient for complex applications with both IO and heavy CPU processing tasks.
- What makes working with `async` channels preferable is the ability to combine them with other `future`s to combine them and create complex control flow.

Futures Control Flow

Futures can be combined together to produce concurrent compute flow graphs. We have already seen tasks, that function as independent threads of execution.

- Join
- Select

加入

A join operation waits until all of a set of futures are ready, and returns a collection of their results. This is similar to `Promise.all` in JavaScript or `asyncio.gather` in Python.

```
use anyhow::Result;
use futures::future;
use reqwest;
use std::collections::HashMap;

async fn size_of_page(url: &str) -> Result<usize> {
    let resp = reqwest::get(url).await?;
    Ok(resp.text().await?.len())
}

#[tokio::main]
async fn main() {
    let urls: [&str; 4] = [
        "https://google.com",
        "https://httpbin.org/ip",
        "https://play.rust-lang.org/",
        "BAD_URL",
    ];
    let futures_iter = urls.into_iter().map(size_of_page);
    let results = future::join_all(futures_iter).await;
    let page_sizes_dict: HashMap<&str, Result<usize>> =
        urls.into_iter().zip(results.into_iter()).collect();
    println!("{:?}", page_sizes_dict);
}
```

▼ *Speaker Notes*

Copy this example into your prepared `src/main.rs` and run it from there.

- For multiple futures of disjoint types, you can use `std::future::join!` but you must know how many futures you will have at compile time. This is currently in the `futures` crate, soon to be stabilised in `std::future`.
- The risk of `join` is that one of the futures may never resolve, this would cause your program to stall.
- You can also combine `join_all` with `join!` for instance to join all requests to an http service as well as a database query. Try adding a `tokio::time::sleep` to the future, using `futures::join!`. This is not a timeout (that requires `select!`, explained in the next chapter), but demonstrates `join!`.

选择

A select operation waits until any of a set of futures is ready, and responds to that future's result. In JavaScript, this is similar to `Promise.race`. In Python, it compares to `asyncio.wait(task_set, return_when=asyncio.FIRST_COMPLETED)`.

Similar to a match statement, the body of `select!` has a number of arms, each of the form `pattern = future => statement`. When the `future` is ready, the `statement` is executed with the variables in `pattern` bound to the `future`'s result.

```rust
use tokio::sync::mpsc::{self, Receiver};
use tokio::time::{sleep, Duration};

#[derive(Debug, PartialEq)]
enum Animal {
    Cat { name: String },
    Dog { name: String },
}

async fn first_animal_to_finish_race(
    mut cat_rcv: Receiver<String>,
    mut dog_rcv: Receiver<String>,
) -> Option<Animal> {
    tokio::select! {
        cat_name = cat_rcv.recv() => Some(Animal::Cat { name: cat_name? }),
        dog_name = dog_rcv.recv() => Some(Animal::Dog { name: dog_name? })
    }
}

#[tokio::main]
async fn main() {
    let (cat_sender, cat_receiver) = mpsc::channel(32);
    let (dog_sender, dog_receiver) = mpsc::channel(32);
    tokio::spawn(async move {
        sleep(Duration::from_millis(500)).await;
        cat_sender
            .send(String::from("Felix"))
            .await
            .expect("Failed to send cat.");
    });
    tokio::spawn(async move {
        sleep(Duration::from_millis(50)).await;
        dog_sender
            .send(String::from("Rex"))
            .await
            .expect("Failed to send dog.");
    });

    let winner = first_animal_to_finish_race(cat_receiver, dog_receiver)
        .await
        .expect("Failed to receive winner");

    println!("Winner is {winner:?}");
}
```

▼ *Speaker Notes*

- In this example, we have a race between a cat and a dog. `first_animal_to_finish_race` listens to both channels and will pick whichever arrives first. Since the dog takes 50ms, it wins against the cat that take 500ms.
- You can use `oneshot` channels in this example as the channels are supposed to receive only one `send`.
- Try adding a deadline to the race, demonstrating selecting different sorts of futures.

- Note that `select!` drops unmatched branches, which cancels their futures. It is easiest to use when every execution of `select!` creates new futures.
    - An alternative is to pass `&mut future` instead of the future itself, but this can lead to issues, further discussed in the pinning slide.

Pitfalls of async/await

Async / await provides convenient and efficient abstraction for concurrent asynchronous programming. However, the async/await model in Rust also comes with its share of pitfalls and footguns. We illustrate some of them in this chapter:

- Blocking the Executor
- Pin
- Async Traits
- Cancellation

Blocking the executor

Most async runtimes only allow IO tasks to run concurrently. This means that CPU blocking tasks will block the executor and prevent other tasks from being executed. An easy workaround is to use async equivalent methods where possible.

```
use futures::future::join_all;
use std::time::Instant;

async fn sleep_ms(start: &Instant, id: u64, duration_ms: u64) {
    std::thread::sleep(std::time::Duration::from_millis(duration_ms));
    println!(
        "future {id} slept for {duration_ms}ms, finished after {}ms",
        start.elapsed().as_millis()
    );
}

#[tokio::main(flavor = "current_thread")]
async fn main() {
    let start = Instant::now();
    let sleep_futures = (1..=10).map(|t| sleep_ms(&start, t, t * 10));
    join_all(sleep_futures).await;
}
```

▼ *Speaker Notes*

- Run the code and see that the sleeps happen consecutively rather than concurrently.
- The `"current_thread"` flavor puts all tasks on a single thread. This makes the effect more obvious, but the bug is still present in the multi-threaded flavor.
- Switch the `std::thread::sleep` to `tokio::time::sleep` and await its result.
- Another fix would be to `tokio::task::spawn_blocking` which spawns an actual thread and transforms its handle into a future without blocking the executor.
- You should not think of tasks as OS threads. They do not map 1 to 1 and most executors will allow many tasks to run on a single OS thread. This is particularly problematic when interacting with other libraries via FFI, where that library might depend on thread-local storage or map to specific OS threads (e.g., CUDA). Prefer `tokio::task::spawn_blocking` in such situations.
- Use sync mutexes with care. Holding a mutex over an `.await` may cause another task to block, and that task may be running on the same thread.

固定

When you await a future, all local variables (that would ordinarily be stored on a stack frame) are instead stored in the Future for the current async block. If your future has pointers to data on the stack, those pointers might get invalidated. This is unsafe.

Therefore, you must guarantee that the addresses your future points to don't change. That is why we need to `pin` futures. Using the same future repeatedly in a `select!` often leads to issues with pinned values.

```rust
use tokio::sync::{mpsc, oneshot};
use tokio::task::spawn;
use tokio::time::{sleep, Duration};

// A work item. In this case, just sleep for the given time and respond
// with a message on the `respond_on` channel.
#[derive(Debug)]
struct Work {
    input: u32,
    respond_on: oneshot::Sender<u32>,
}

// A worker which listens for work on a queue and performs it.
async fn worker(mut work_queue: mpsc::Receiver<Work>) {
    let mut iterations = 0;
    loop {
        tokio::select! {
            Some(work) = work_queue.recv() => {
                sleep(Duration::from_millis(10)).await; // Pretend to work.
                work.respond_on
                    .send(work.input * 1000)
                    .expect("failed to send response");
                iterations += 1;
            }
            // TODO: report number of iterations every 100ms
        }
    }
}

// A requester which requests work and waits for it to complete.
async fn do_work(work_queue: &mpsc::Sender<Work>, input: u32) -> u32 {
    let (tx, rx) = oneshot::channel();
    work_queue
        .send(Work {
            input,
            respond_on: tx,
        })
        .await
        .expect("failed to send on work queue");
    rx.await.expect("failed waiting for response")
}

#[tokio::main]
async fn main() {
    let (tx, rx) = mpsc::channel(10);
    spawn(worker(rx));
    for i in 0..100 {
        let resp = do_work(&tx, i).await;
        println!("work result for iteration {i}: {resp}");
    }
}
```

▼ *Speaker Notes*

- You may recognize this as an example of the actor pattern. Actors typically call `select!` in a loop.

- This serves as a summation of a few of the previous lessons, so take your time with it.
    - Naively add a `_ = sleep(Duration::from_millis(100)) => { println!(..) }` to the `select!`. This will never execute. Why?
    - Instead, add a `timeout_fut` containing that future outside of the `loop`:

```
let mut timeout_fut = sleep(Duration::from_millis(100));
loop {
    select! {
        ..,
        _ = timeout_fut => { println!(..); },
    }
}
```

    - This still doesn't work. Follow the compiler errors, adding `&mut` to the `timeout_fut` in the `select!` to work around the move, then using `Box::pin`:

```
let mut timeout_fut = Box::pin(sleep(Duration::from_millis(100)));
loop {
    select! {
        ..,
        _ = &mut timeout_fut => { println!(..); },
    }
}
```

    - This compiles, but once the timeout expires it is `Poll::Ready` on every iteration (a fused future would help with this). Update to reset `timeout_fut` every time it expires.
- Box allocates on the heap. In some cases, `std::pin::pin!` (only recently stabilized, with older code often using `tokio::pin!`) is also an option, but that is difficult to use for a future that is reassigned.
- Another alternative is to not use `pin` at all but spawn another task that will send to a `oneshot` channel every 100ms.

异步特质

Async methods in traits are not yet supported in the stable channel (An experimental feature exists in nightly and should be stabilized in the mid term.)

The crate async_trait provides a workaround through a macro:

```
use async_trait::async_trait;
use std::time::Instant;
use tokio::time::{sleep, Duration};

#[async_trait]
trait Sleeper {
    async fn sleep(&self);
}

struct FixedSleeper {
    sleep_ms: u64,
}

#[async_trait]
impl Sleeper for FixedSleeper {
    async fn sleep(&self) {
        sleep(Duration::from_millis(self.sleep_ms)).await;
    }
}

async fn run_all_sleepers_multiple_times(sleepers: Vec<Box<dyn Sleeper>>, n_
    for _ in 0..n_times {
        println!("running all sleepers..");
        for sleeper in &sleepers {
            let start = Instant::now();
            sleeper.sleep().await;
            println!("slept for {}ms", start.elapsed().as_millis());
        }
    }
}

#[tokio::main]
async fn main() {
    let sleepers: Vec<Box<dyn Sleeper>> = vec![
        Box::new(FixedSleeper { sleep_ms: 50 }),
        Box::new(FixedSleeper { sleep_ms: 100 }),
    ];
    run_all_sleepers_multiple_times(sleepers, 5).await;
}
```

▼ *Speaker Notes*

- `async_trait` is easy to use, but note that it's using heap allocations to achieve this. This heap allocation has performance overhead.

- 对于 `async trait` 的语言支持中的挑战是深入 Rust的，并且可能不值得深入描述。如果您对深入了解感兴趣，Niko Matsakis 在这篇文章中对它们做了很好的解释。
- 尝试创建一个新的 sleeper 结构，使其随机休眠一段时间，并将其添加到 Vec 中。

消除

Dropping a future implies it can never be polled again. This is called *cancellation* and it can occur at any `await` point. Care is needed to ensure the system works correctly even when futures are cancelled. For example, it shouldn't deadlock or lose data.

```rust
use std::io::{self, ErrorKind};
use std::time::Duration;
use tokio::io::{AsyncReadExt, AsyncWriteExt, DuplexStream};

struct LinesReader {
    stream: DuplexStream,
}

impl LinesReader {
    fn new(stream: DuplexStream) -> Self {
        Self { stream }
    }

    async fn next(&mut self) -> io::Result<Option<String>> {
        let mut bytes = Vec::new();
        let mut buf = [0];
        while self.stream.read(&mut buf[..]).await? != 0 {
            bytes.push(buf[0]);
            if buf[0] == b'\n' {
                break;
            }
        }
        if bytes.is_empty() {
            return Ok(None)
        }
        let s = String::from_utf8(bytes)
            .map_err(|_| io::Error::new(ErrorKind::InvalidData, "not UTF-8")
        Ok(Some(s))
    }
}

async fn slow_copy(source: String, mut dest: DuplexStream) -> std::io::Resul
    for b in source.bytes() {
        dest.write_u8(b).await?;
        tokio::time::sleep(Duration::from_millis(10)).await
    }
    Ok(())
}

#[tokio::main]
async fn main() -> std::io::Result<()> {
    let (client, server) = tokio::io::duplex(5);
    let handle = tokio::spawn(slow_copy("hi\nthere\n".to_owned(), client));

    let mut lines = LinesReader::new(server);
    let mut interval = tokio::time::interval(Duration::from_millis(60));
    loop {
        tokio::select! {
            _ = interval.tick() => println!("tick!"),
            line = lines.next() => if let Some(l) = line? {
                print!("{}", l)
            } else {
                break
            },
        }
    }
    handle.await.unwrap()?;
    Ok(())
```

```
59  }
```

▼ *Speaker Notes*

- The compiler doesn't help with cancellation-safety. You need to read API documentation and consider what state your `async fn` holds.
- Unlike `panic` and `?`, cancellation is part of normal control flow (vs error-handling).
- The example loses parts of the string.
  - Whenever the `tick()` branch finishes first, `next()` and its `buf` are dropped.
  - `LinesReader` can be made cancellation-safe by making `buf` part of the struct:

```
struct LinesReader {
    stream: DuplexStream,
    bytes: Vec<u8>,
    buf: [u8; 1],
}

impl LinesReader {
    fn new(stream: DuplexStream) -> Self {
        Self { stream, bytes: Vec::new(), buf: [0] }
    }
    async fn next(&mut self) -> io::Result<Option<String>> {
        // prefix buf and bytes with self.
        // ...
        let raw = std::mem::take(&mut self.bytes);
        let s = String::from_utf8(raw)
        // ...
    }
}
```

- `Interval::tick` is cancellation-safe because it keeps track of whether a tick has been 'delivered'.
- `AsyncReadExt::read` is cancellation-safe because it either returns or doesn't read data.
- `AsyncBufReadExt::read_line` is similar to the example and *isn't* cancellation-safe. See its documentation for details and alternatives.

习题

为了练习您的异步 Rust 技能，我们再次为您提供了两个练习：

- 哲学家进餐：我们已经在上午看到了这个问题。这次你将使用异步 Rust 来实现它。
- 广播聊天应用：这是一个更大的项目，允许您尝试更高级的异步Rust功能。

▼ *Speaker Notes*

After looking at the exercises, you can look at the solutions provided.

哲学家进餐 - 异步

查看哲学家进餐以获取问题的描述。

与之前一样，您需要一个本地的 Cargo 安装来进行这个练习。将下面的代码复制到一个名为 `src/main.rs` 的文件中，填写空白部分，并测试确保 `cargo run` 不会死锁：

```
use std::sync::Arc;
use tokio::time;
use tokio::sync::mpsc::{self, Sender};
use tokio::sync::Mutex;

struct Fork;

struct Philosopher {
    name: String,
    // left_fork: ...
    // right_fork: ...
    // thoughts: ...
}

impl Philosopher {
    async fn think(&self) {
        self.thoughts
            .send(format!("Eureka! {} has a new idea!", &self.name)).await
            .unwrap();
    }

    async fn eat(&self) {
        // Pick up forks...
        println!("{} is eating...", &self.name);
        time::sleep(time::Duration::from_millis(5)).await;
    }
}

static PHILOSOPHERS: &[&str] =
    &["Socrates", "Plato", "Aristotle", "Thales", "Pythagoras"];

#[tokio::main]
async fn main() {
    // Create forks

    // Create philosophers

    // Make them think and eat

    // Output their thoughts
}
```

因为这次您正在使用异步Rust，您将需要一个 `tokio` 依赖。您可以使用以下的 `Cargo.toml`：

```toml
[package]
name = "dining-philosophers-async-dine"
version = "0.1.0"
edition = "2021"

[dependencies]
tokio = {version = "1.26.0", features = ["sync", "time", "macros", "rt-multi-thread"]}
```

另外，请注意，这次您必须使用来自 `tokio` 包的 `Mutex` 和 `mpsc` 模块。

▼ *Speaker Notes*

- 您可以单线程化您的实现吗？

广播聊天应用程序

在本练习中，我们想要使用我们的新知识来实现一个广播聊天应用。我们有一个聊天服务器，客户端连接到该服务器并发布他们的消息。客户端从标准输入读取用户消息，并将其发送到服务器。聊天服务器将收到的每条消息广播给所有客户端。

For this, we use a broadcast channel on the server, and `tokio_websockets` for the communication between the client and the server.

创建一个新的 Cargo 项目并添加以下依赖：

Cargo.toml:

```
[package]
name = "chat-async"
version = "0.1.0"
edition = "2021"

[dependencies]
futures-util = { version = "0.3.28", features = ["sink"] }
http = "0.2.9"
tokio = { version = "1.28.1", features = ["full"] }
tokio-websockets = { version = "0.4.0", features = ["client", "fastrand",
"server", "sha1_smol"] }
```

所需的API

You are going to need the following functions from `tokio` and `tokio_websockets`. Spend a few minutes to familiarize yourself with the API.

- StreamExt::next() implemented by `WebsocketStream`: for asynchronously reading messages from a Websocket Stream.
- SinkExt::send() 由 `WebsocketStream` 实现：用于在Websocket流上异步发送消息。
- Lines::next_line(): for asynchronously reading user messages from the standard input.
- Sender::subscribe()：用于订阅广播频道。

两个可执行文件

通常在一个 Cargo 项目中，你只能有一个二进制文件，和一个 `src/main.rs` 文件。在这个项目中，我们需要两个二进制文件。一个用于客户端，另一个用于服务器。你可能会考虑将它们制作成两个单独的 Cargo 项目，但我们将它们放在一个包含两个二进制文件的 Cargo 项目中。为了使其工作，客户端和服务器的代码应该放在 `src/bin` 下（参见文档）。

将以下服务器和客户端代码分别复制到 `src/bin/server.rs` 和 `src/bin/client.rs` 中。您的任务是按照下面的描述完成这些文件。

src/bin/server.rs:

```rust
use futures_util::sink::SinkExt;
use futures_util::stream::StreamExt;
use std::error::Error;
use std::net::SocketAddr;
use tokio::net::{TcpListener, TcpStream};
use tokio::sync::broadcast::{channel, Sender};
use tokio_websockets::{Message, ServerBuilder, WebsocketStream};

async fn handle_connection(
    addr: SocketAddr,
    mut ws_stream: WebsocketStream<TcpStream>,
    bcast_tx: Sender<String>,
) -> Result<(), Box<dyn Error + Send + Sync>> {

    // TODO: For a hint, see the description of the task below.

}

#[tokio::main]
async fn main() -> Result<(), Box<dyn Error + Send + Sync>> {
    let (bcast_tx, _) = channel(16);

    let listener = TcpListener::bind("127.0.0.1:2000").await?;
    println!("listening on port 2000");

    loop {
        let (socket, addr) = listener.accept().await?;
        println!("New connection from {addr:?}");
        let bcast_tx = bcast_tx.clone();
        tokio::spawn(async move {
            // Wrap the raw TCP stream into a websocket.
            let ws_stream = ServerBuilder::new().accept(socket).await?;

            handle_connection(addr, ws_stream, bcast_tx).await
        });
    }
}
```

src/bin/client.rs:

```rust
use futures_util::stream::StreamExt;
use futures_util::SinkExt;
use http::Uri;
use tokio::io::{AsyncBufReadExt, BufReader};
use tokio_websockets::{ClientBuilder, Message};

#[tokio::main]
async fn main() -> Result<(), tokio_websockets::Error> {
    let (mut ws_stream, _) =
        ClientBuilder::from_uri(Uri::from_static("ws://127.0.0.1:2000"))
            .connect()
            .await?;

    let stdin = tokio::io::stdin();
    let mut stdin = BufReader::new(stdin).lines();


    // TODO: For a hint, see the description of the task below.

}
```

运行可执行文件

使用以下命令运行服务器：

```
cargo run --bin server
```

and the client with:

```
cargo run --bin client
```

任务

- 在 `src/bin/server.rs` 中实现 `handle_connection` 函数。
  - 提示：使用 `tokio::select!` 在一个连续的循环中并发执行两个任务。一个任务从客户端接收消息并广播它们。另一个任务将服务器接收到的消息发送给客户端。
- 完成 `src/bin/client.rs` 中的 `main` 函数。
  - Hint: As before, use `tokio::select!` in a continuous loop for concurrently performing two tasks: (1) reading user messages from standard input and sending them to the server, and (2) receiving messages from the server, and displaying them for the user.
- Optional: Once you are done, change the code to broadcast messages to all clients, but the sender of the message.

谢谢！

Thank you for taking Comprehensive Rust  *!* We hope you enjoyed it and that it was useful.

组织这门课程让我们收获了很多乐趣。本课程并非完美无缺，因此，如果您发现任何错误或有任何改进建议，请在 GitHub 上与我们联系。我们期待收到您的宝贵意见。

词汇表

The following is a glossary which aims to give a short definition of many Rust terms. For translations, this also serves to connect the term back to the English original.

- allocate:
  Dynamic memory allocation on the heap.
- argument:
- Bare-metal Rust: See Bare-metal Rust.
- block:
  See Blocks and *scope*.
- borrow:
  See Borrowing.
- borrow checker:
  The part of the Rust compiler which checks that all borrows are valid.
- brace:
  `{` and `}`. Also called *curly brace*, they delimit *blocks*.
- build:
- call:
- channel:
  Used to safely pass messages between threads.
- Comprehensive Rust 🦀:
  The courses here are jointly called Comprehensive Rust 🦀.
- concurrency:
- Concurrency in Rust:
  See Concurrency in Rust.
- constant:
- control flow:
- crash:
- enumeration:
- error:
- error handling:
- exercise:
- function:
- garbage collector:
- generics:
- immutable:
- integration test:
- keyword:
- library:
- macro:
- main function:
- match:

- memory leak:
- method:
- module:
- move:
- 可变:
- ownership:
- panic:
- parameter:
- pattern:
- payload:
- program:
- programming language:
- receiver:
- reference counting:
- return:
- Rust:
- Rust Fundamentals:
  Days 1 to 3 of this course.
- Rust in Android:
  See Rust in Android.
- safe:
- scope:
- standard library:
- static:
- string:
- struct:
- test:
- thread:
- thread safety:
- trait:
- type:
- type inference:
- undefined behavior:
- union:
- unit test:
- 是（不安全）
- variable:\

其他 Rust 资源

Rust 社区已经创造了丰富的高质量免费资源在线提供。

官方文档

Rust 项目提供了许多资源。这些资源涵盖了 Rust 的一般内容：

- Rust 程序设计语言：一部有关 Rust 的免费权威图书。书中详细介绍了该语言，并包含一些可供读者构建的项目。
- 通过例子学 Rust：通过一系列展示不同结构的示例介绍 Rust 语法。有时会包括一些小练习，会要求您充分地阐述示例中的代码。
- Rust 标准库：Rust 标准库的完整文档。
- Rust 参考手册：一本未完成的书，介绍了 Rust 语法和内存模型。

Rust 官方网站上有更多专业指南：

- Rust 秘典：介绍了不安全 Rust，包括使用原始指针以及与其他语言 (FFI) 交互。
- Rust 中的异步编程：介绍了在《Rust 程序设计语言》成书后引入的新异步编程模型。
- 嵌入式 Rust 之书：介绍如何在没有操作系统的嵌入式设备上使用 Rust。

非官方学习资料

其他 Rust 指南和教程的小选集：

- Learn Rust the Dangerous Way（以危险的方式学 Rust）：从低级 C 语言程序员的角度介绍 Rust。
- 面向嵌入式 C 程序员的 Rust：从使用 C 语言编写固件的开发者的角度介绍 Rust。
- Rust for professionals（面向专业人士的 Rust）：通过与其他语言（例如 C、C++、Java、JavaScript 和 Python）进行并排比较，介绍 Rust 的语法。
- Rust on Exercism（在 Exercism 上学 Rust）：100 多项练习助您学习 Rust。
- Ferrous Teaching Material：一系列小演示文稿，涵盖 Rust 语言的基础知识和高级部分。还涵盖了 WebAssembly 和 async/await 等其他主题。
- 面向 Rust 的初学者系列和使用 Rust 迈出第一步：两个面向新手开发者的 Rust 指南。第一个指南包含 35 个视频，第二个指南包含 11 个模块，内容涵盖 Rust 语法和基本结构。
- 通过大量的链表学习Rust：通过实现几种不同类型的列表结构，深入探索 Rust 的内存管理规则。

如需更多 Rust 图书，请查看 Rust 小册。

鸣谢

本课中的资料以众多优秀的 Rust 文档资源为基础。 如需查看实用资源的完整列表， 请参阅关于其他资源的页面。

Rust 示例

部分示例和练习复制并 改编自Rust by Example。如需了解详情（包括许可 条款），请参阅 `third_party/rust-by-example/` 目录。

Rust on Exercism

部分练习复制并 改编自 Rust on Exercism。如需了解详情（包括许可 条款），请参阅 `third_party/rust-on-exercism/` 目录。

CXX

“与 C++ 的互操作性”部分引用了一张 来自 CXX 的图片。如需了解详情（包括许可条款）， 请参阅 `third_party/cxx/` 目录。

C语言示例

The Why Rust? - An Example in C section has been taken from the presentation slides of Colin Finck's Master Thesis. It has been relicensed under the terms of the Apache 2.0 license for this course by the author.

解答

您将在下面的页面找到练习的解答。

欢迎您在 GitHub 上提问关于解决方案的问题。如果您有与此处呈现的不同或更好的解决方案，请告诉我们。

第一天上午的练习

数组与 for 循环

(返回练习)

```rust
fn transpose(matrix: [[i32; 3]; 3]) -> [[i32; 3]; 3] {
    let mut result = [[0; 3]; 3];
    for i in 0..3 {
        for j in 0..3 {
            result[j][i] = matrix[i][j];
        }
    }
    return result;
}

fn pretty_print(matrix: &[[i32; 3]; 3]) {
    for row in matrix {
        println!("{row:?}");
    }
}

#[test]
fn test_transpose() {
    let matrix = [
        [101, 102, 103], //
        [201, 202, 203],
        [301, 302, 303],
    ];
    let transposed = transpose(matrix);
    assert_eq!(
        transposed,
        [
            [101, 201, 301], //
            [102, 202, 302],
            [103, 203, 303],
        ]
    );
}

fn main() {
    let matrix = [
        [101, 102, 103], // <-- the comment makes rustfmt add a newline
        [201, 202, 203],
        [301, 302, 303],
    ];

    println!("matrix:");
    pretty_print(&matrix);

    let transposed = transpose(matrix);
    println!("transposed:");
    pretty_print(&transposed);
}
```

附加问题

这需要更高级的概念。看起来，我们可以使用切片的切片（ `&[&[i32]]` ）作为输入类型来进行转置，从而使我们的函数能够处理任意大小的矩阵。然而，这很快就会崩溃：返回类型不能是 `&[&`

`[i32]]`，因为它需要拥有您返回的数据。

您可以尝试使用类似 `Vec<Vec<i32>>` 的方式，但这也无法直接工作：从 `Vec<Vec<i32>>` 转换为 `&[&[i32]]` 很困难，因此您现在也不能轻松使用 `pretty_print` 。

了解 trait 和泛型后，我们就可以使用“std::convert::AsRef”trait 来抽象化任何可作为 Slice 引用的内容了。

```
use std::convert::AsRef;
use std::fmt::Debug;

fn pretty_print<T, Line, Matrix>(matrix: Matrix)
where
    T: Debug,
    // A line references a slice of items
    Line: AsRef<[T]>,
    // A matrix references a slice of lines
    Matrix: AsRef<[Line]>
{
    for row in matrix.as_ref() {
        println!("{:?}", row.as_ref());
    }
}

fn main() {
    // &[&[i32]]
    pretty_print(&[&[1, 2, 3], &[4, 5, 6], &[7, 8, 9]]);
    // [[&str; 2]; 2]
    pretty_print([["a", "b"], ["c", "d"]]);
    // Vec<Vec<i32>>
    pretty_print(vec![vec![1, 2], vec![3, 4]]);
}
```

此外，类型本身不会强制要求子切片具有相同的长度，因此这样的变量可能包含一个无效的矩阵。

第一天下午的练习

卢恩算法

(返回练习)

```rust
pub fn luhn(cc_number: &str) -> bool {
    let mut digits_seen = 0;
    let mut sum = 0;
    for (i, ch) in cc_number.chars().rev().filter(|&ch| ch != ' ').enumerate()
{
        match ch.to_digit(10) {
            Some(d) => {
                sum += if i % 2 == 1 {
                    let dd = d * 2;
                    dd / 10 + dd % 10
                } else {
                    d
                };
                digits_seen += 1;
            }
            None => return false,
        }
    }

    if digits_seen < 2 {
        return false;
    }

    sum % 10 == 0
}

fn main() {
    let cc_number = "1234 5678 1234 5670";
    println!(
        "Is {cc_number} a valid credit card number? {}",
        if luhn(cc_number) { "yes" } else { "no" }
    );
}

#[test]
fn test_non_digit_cc_number() {
    assert!(!luhn("foo"));
    assert!(!luhn("foo 0 0"));
}

#[test]
fn test_empty_cc_number() {
    assert!(!luhn(""));
    assert!(!luhn(" "));
    assert!(!luhn("  "));
    assert!(!luhn("    "));
}

#[test]
fn test_single_digit_cc_number() {
    assert!(!luhn("0"));
}

#[test]
fn test_two_digit_cc_number() {
    assert!(luhn(" 0 0 "));
}
```

```rust
#[test]
fn test_valid_cc_number() {
    assert!(luhn("4263 9826 4026 9299"));
    assert!(luhn("4539 3195 0343 6467"));
    assert!(luhn("7992 7398 713"));
}

#[test]
fn test_invalid_cc_number() {
    assert!(!luhn("4223 9826 4026 9299"));
    assert!(!luhn("4539 3195 0343 6476"));
    assert!(!luhn("8273 1232 7352 0569"));
}
```

Pattern matching

```
/// An operation to perform on two subexpressions.
#[derive(Debug)]
enum Operation {
    Add,
    Sub,
    Mul,
    Div,
}

/// An expression, in tree form.
#[derive(Debug)]
enum Expression {
    /// An operation on two subexpressions.
    Op {
        op: Operation,
        left: Box<Expression>,
        right: Box<Expression>,
    },

    /// A literal value
    Value(i64),
}

/// The result of evaluating an expression.
#[derive(Debug, PartialEq, Eq)]
enum Res {
    /// Evaluation was successful, with the given result.
    Ok(i64),
    /// Evaluation failed, with the given error message.
    Err(String),
}
// Allow `Ok` and `Err` as shorthands for `Res::Ok` and `Res::Err`.
use Res::{Err, Ok};

fn eval(e: Expression) -> Res {
    match e {
        Expression::Op { op, left, right } => {
            let left = match eval(*left) {
                Ok(v) => v,
                Err(msg) => return Err(msg),
            };
            let right = match eval(*right) {
                Ok(v) => v,
                Err(msg) => return Err(msg),
            };
            Ok(match op {
                Operation::Add => left + right,
                Operation::Sub => left - right,
                Operation::Mul => left * right,
                Operation::Div => {
                    if right == 0 {
                        return Err(String::from("division by zero"));
                    } else {
                        left / right
```

```
                    }
                }
            })
        }
        Expression::Value(v) => Ok(v),
    }
}

#[test]
fn test_value() {
    assert_eq!(eval(Expression::Value(19)), Ok(19));
}

#[test]
fn test_sum() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(Expression::Value(10)),
            right: Box::new(Expression::Value(20)),
        }),
        Ok(30)
    );
}

#[test]
fn test_recursion() {
    let term1 = Expression::Op {
        op: Operation::Mul,
        left: Box::new(Expression::Value(10)),
        right: Box::new(Expression::Value(9)),
    };
    let term2 = Expression::Op {
        op: Operation::Mul,
        left: Box::new(Expression::Op {
            op: Operation::Sub,
            left: Box::new(Expression::Value(3)),
            right: Box::new(Expression::Value(4)),
        }),
        right: Box::new(Expression::Value(5)),
    };
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(term1),
            right: Box::new(term2),
        }),
        Ok(85)
    );
}

#[test]
fn test_error() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Div,
            left: Box::new(Expression::Value(99)),
            right: Box::new(Expression::Value(0)),
```

```
        }),
        Err(String::from("division by zero"))
    );
}
fn main() {
    let expr = Expression::Op {
        op: Operation::Sub,
        left: Box::new(Expression::Value(20)),
        right: Box::new(Expression::Value(10)),
    };
    println!("expr: {:?}", expr);
    println!("result: {:?}", eval(expr));
}
```

第二天上午的练习

设计一个库

(返回练习)

```rust
struct Library {
    books: Vec<Book>,
}

struct Book {
    title: String,
    year: u16,
}

impl Book {
    // This is a constructor, used below.
    fn new(title: &str, year: u16) -> Book {
        Book {
            title: String::from(title),
            year,
        }
    }
}

// Implement the methods below. Notice how the `self` parameter
// changes type to indicate the method's required level of ownership
// over the object:
//
// - `&self` for shared read-only access,
// - `&mut self` for unique and mutable access,
// - `self` for unique access by value.
impl Library {

    fn new() -> Library {
        Library { books: Vec::new() }
    }

    fn len(&self) -> usize {
        self.books.len()
    }

    fn is_empty(&self) -> bool {
        self.books.is_empty()
    }

    fn add_book(&mut self, book: Book) {
        self.books.push(book)
    }

    fn print_books(&self) {
        for book in &self.books {
            println!("{}, published in {}", book.title, book.year);
        }
    }

    fn oldest_book(&self) -> Option<&Book> {
        // Using a closure and a built-in method:
        // self.books.iter().min_by_key(|book| book.year)

        // Longer hand-written solution:
        let mut oldest: Option<&Book> = None;
        for book in self.books.iter() {
```

```
            if oldest.is_none() || book.year < oldest.unwrap().year {
                oldest = Some(book);
            }
        }

        oldest
    }
}

fn main() {
    let mut library = Library::new();

    println!(
        "The library is empty: library.is_empty() -> {}",
        library.is_empty()
    );

    library.add_book(Book::new("Lord of the Rings", 1954));
    library.add_book(Book::new("Alice's Adventures in Wonderland", 1865));

    println!(
        "The library is no longer empty: library.is_empty() -> {}",
        library.is_empty()
    );

    library.print_books();

    match library.oldest_book() {
        Some(book) => println!("The oldest book is {}", book.title),
        None => println!("The library is empty!"),
    }

    println!("The library has {} books", library.len());
    library.print_books();
}

#[test]
fn test_library_len() {
    let mut library = Library::new();
    assert_eq!(library.len(), 0);
    assert!(library.is_empty());

    library.add_book(Book::new("Lord of the Rings", 1954));
    library.add_book(Book::new("Alice's Adventures in Wonderland", 1865));
    assert_eq!(library.len(), 2);
    assert!(!library.is_empty());
}

#[test]
fn test_library_is_empty() {
    let mut library = Library::new();
    assert!(library.is_empty());

    library.add_book(Book::new("Lord of the Rings", 1954));
    assert!(!library.is_empty());
}

#[test]
```

```rust
fn test_library_print_books() {
    let mut library = Library::new();
    library.add_book(Book::new("Lord of the Rings", 1954));
    library.add_book(Book::new("Alice's Adventures in Wonderland", 1865));
    // We could try and capture stdout, but let us just call the
    // method to start with.
    library.print_books();
}

#[test]
fn test_library_oldest_book() {
    let mut library = Library::new();
    assert!(library.oldest_book().is_none());

    library.add_book(Book::new("Lord of the Rings", 1954));
    assert_eq!(
        library.oldest_book().map(|b| b.title.as_str()),
        Some("Lord of the Rings")
    );

    library.add_book(Book::new("Alice's Adventures in Wonderland", 1865));
    assert_eq!(
        library.oldest_book().map(|b| b.title.as_str()),
        Some("Alice's Adventures in Wonderland")
    );
}
```

健康统计

(back to exercise)

```rust
pub struct User {
    name: String,
    age: u32,
    height: f32,
    visit_count: usize,
    last_blood_pressure: Option<(u32, u32)>,
}

pub struct Measurements {
    height: f32,
    blood_pressure: (u32, u32),
}

pub struct HealthReport<'a> {
    patient_name: &'a str,
    visit_count: u32,
    height_change: f32,
    blood_pressure_change: Option<(i32, i32)>,
}

impl User {
    pub fn new(name: String, age: u32, height: f32) -> Self {
        Self {
            name,
            age,
            height,
            visit_count: 0,
            last_blood_pressure: None,
        }
    }

    pub fn name(&self) -> &str {
        &self.name
    }

    pub fn age(&self) -> u32 {
        self.age
    }

    pub fn height(&self) -> f32 {
        self.height
    }

    pub fn doctor_visits(&self) -> u32 {
        self.visit_count as u32
    }

    pub fn set_age(&mut self, new_age: u32) {
        self.age = new_age
    }

    pub fn set_height(&mut self, new_height: f32) {
        self.height = new_height
    }

    pub fn visit_doctor(&mut self, measurements: Measurements) -> HealthReport
{
```

```
        self.visit_count += 1;
        let bp = measurements.blood_pressure;
        let report = HealthReport {
            patient_name: &self.name,
            visit_count: self.visit_count as u32,
            height_change: measurements.height - self.height,
            blood_pressure_change: match self.last_blood_pressure {
                Some(lbp) => Some((
                    bp.0 as i32 - lbp.0 as i32,
                    bp.1 as i32 - lbp.1 as i32
                )),
                None => None,
            }
        };
        self.height = measurements.height;
        self.last_blood_pressure = Some(bp);
        report
    }
}

fn main() {
    let bob = User::new(String::from("Bob"), 32, 155.2);
    println!("I'm {} and my age is {}", bob.name(), bob.age());
}

#[test]
fn test_height() {
    let bob = User::new(String::from("Bob"), 32, 155.2);
    assert_eq!(bob.height(), 155.2);
}

#[test]
fn test_set_age() {
    let mut bob = User::new(String::from("Bob"), 32, 155.2);
    assert_eq!(bob.age(), 32);
    bob.set_age(33);
    assert_eq!(bob.age(), 33);
}

#[test]
fn test_visit() {
    let mut bob = User::new(String::from("Bob"), 32, 155.2);
    assert_eq!(bob.doctor_visits(), 0);
    let report = bob.visit_doctor(Measurements {
        height: 156.1,
        blood_pressure: (120, 80),
    });
    assert_eq!(report.patient_name, "Bob");
    assert_eq!(report.visit_count, 1);
    assert_eq!(report.blood_pressure_change, None);

    let report = bob.visit_doctor(Measurements {
        height: 156.1,
        blood_pressure: (115, 76),
    });

    assert_eq!(report.visit_count, 2);
```

```
    assert_eq!(report.blood_pressure_change, Some((-5, -4)));
}
```

第二天下午的练习

字符串和迭代器

(返回练习)

```rust
pub fn prefix_matches(prefix: &str, request_path: &str) -> bool {

    let mut request_segments = request_path.split('/');

    for prefix_segment in prefix.split('/') {
        let Some(request_segment) = request_segments.next() else {
            return false;
        };
        if request_segment != prefix_segment && prefix_segment != "*" {
            return false;
        }
    }
    true

    // Alternatively, Iterator::zip() lets us iterate simultaneously over
prefix
    // and request segments. The zip() iterator is finished as soon as one of
    // the source iterators is finished, but we need to iterate over all
request
    // segments. A neat trick that makes zip() work is to use map() and chain()
    // to produce an iterator that returns Some(str) for each pattern segments,
    // and then returns None indefinitely.
}

#[test]
fn test_matches_without_wildcard() {
    assert!(prefix_matches("/v1/publishers", "/v1/publishers"));
    assert!(prefix_matches("/v1/publishers", "/v1/publishers/abc-123"));
    assert!(prefix_matches("/v1/publishers", "/v1/publishers/abc/books"));

    assert!(!prefix_matches("/v1/publishers", "/v1"));
    assert!(!prefix_matches("/v1/publishers", "/v1/publishersBooks"));
    assert!(!prefix_matches("/v1/publishers", "/v1/parent/publishers"));
}

#[test]
fn test_matches_with_wildcard() {
    assert!(prefix_matches(
        "/v1/publishers/*/books",
        "/v1/publishers/foo/books"
    ));
    assert!(prefix_matches(
        "/v1/publishers/*/books",
        "/v1/publishers/bar/books"
    ));
    assert!(prefix_matches(
        "/v1/publishers/*/books",
        "/v1/publishers/foo/books/book1"
    ));

    assert!(!prefix_matches("/v1/publishers/*/books", "/v1/publishers"));
    assert!(!prefix_matches(
        "/v1/publishers/*/books",
        "/v1/publishers/foo/booksByAuthor"
    ));
}
```

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

第三天上午的练习

Drawing A Simple GUI

(返回练习)

```
pub trait Widget {
    /// Natural width of `self`.
    fn width(&self) -> usize;

    /// Draw the widget into a buffer.
    fn draw_into(&self, buffer: &mut dyn std::fmt::Write);

    /// Draw the widget on standard output.
    fn draw(&self) {
        let mut buffer = String::new();
        self.draw_into(&mut buffer);
        println!("{buffer}");
    }
}

pub struct Label {
    label: String,
}

impl Label {
    fn new(label: &str) -> Label {
        Label {
            label: label.to_owned(),
        }
    }
}

pub struct Button {
    label: Label,
}

impl Button {
    fn new(label: &str) -> Button {
        Button {
            label: Label::new(label),
        }
    }
}

pub struct Window {
    title: String,
    widgets: Vec<Box<dyn Widget>>,
}

impl Window {
    fn new(title: &str) -> Window {
        Window {
            title: title.to_owned(),
            widgets: Vec::new(),
        }
    }

    fn add_widget(&mut self, widget: Box<dyn Widget>) {
        self.widgets.push(widget);
    }

    fn inner_width(&self) -> usize {
```

```
        std::cmp::max(
            self.title.chars().count(),
            self.widgets.iter().map(|w| w.width()).max().unwrap_or(0),
        )
    }
}


impl Widget for Window {
    fn width(&self) -> usize {
        // Add 4 paddings for borders
        self.inner_width() + 4
    }

    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        let mut inner = String::new();
        for widget in &self.widgets {
            widget.draw_into(&mut inner);
        }

        let inner_width = self.inner_width();

        // TODO: after learning about error handling, you can change
        // draw_into to return Result<(), std::fmt::Error>. Then use
        // the ?-operator here instead of .unwrap().
        writeln!(buffer, "+-{:-<inner_width$}-+", "").unwrap();
        writeln!(buffer, "| {:^inner_width$} |", &self.title).unwrap();
        writeln!(buffer, "+={:=<inner_width$}=+", "").unwrap();
        for line in inner.lines() {
            writeln!(buffer, "| {:inner_width$} |", line).unwrap();
        }
        writeln!(buffer, "+-{:-<inner_width$}-+", "").unwrap();
    }
}

impl Widget for Button {
    fn width(&self) -> usize {
        self.label.width() + 8 // add a bit of padding
    }

    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        let width = self.width();
        let mut label = String::new();
        self.label.draw_into(&mut label);

        writeln!(buffer, "+{:-<width$}+", "").unwrap();
        for line in label.lines() {
            writeln!(buffer, "|{:^width$}|", &line).unwrap();
        }
        writeln!(buffer, "+{:-<width$}+", "").unwrap();
    }
}

impl Widget for Label {
    fn width(&self) -> usize {
        self.label
            .lines()
            .map(|line| line.chars().count())
```

```
            .max()
            .unwrap_or(0)
    }

    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        writeln!(buffer, "{}", &self.label).unwrap();
    }
}

fn main() {
    let mut window = Window::new("Rust GUI Demo 1.23");
    window.add_widget(Box::new(Label::new("This is a small text GUI demo.")));
    window.add_widget(Box::new(Button::new(
        "Click me!"
    )));
    window.draw();
}
```

点和多边形

(返回练习)

```rust
#[derive(Debug, Copy, Clone, PartialEq, Eq)]
pub struct Point {
    x: i32,
    y: i32,
}

impl Point {
    pub fn new(x: i32, y: i32) -> Point {
        Point { x, y }
    }

    pub fn magnitude(self) -> f64 {
        f64::from(self.x.pow(2) + self.y.pow(2)).sqrt()
    }

    pub fn dist(self, other: Point) -> f64 {
        (self - other).magnitude()
    }
}

impl std::ops::Add for Point {
    type Output = Self;

    fn add(self, other: Self) -> Self::Output {
        Self {
            x: self.x + other.x,
            y: self.y + other.y,
        }
    }
}

impl std::ops::Sub for Point {
    type Output = Self;

    fn sub(self, other: Self) -> Self::Output {
        Self {
            x: self.x - other.x,
            y: self.y - other.y,
        }
    }
}

pub struct Polygon {
    points: Vec<Point>,
}

impl Polygon {
    pub fn new() -> Polygon {
        Polygon { points: Vec::new() }
    }

    pub fn add_point(&mut self, point: Point) {
        self.points.push(point);
    }

    pub fn left_most_point(&self) -> Option<Point> {
        self.points.iter().min_by_key(|p| p.x).copied()
```

```
    }

    pub fn iter(&self) -> impl Iterator<Item = &Point> {
        self.points.iter()
    }

    pub fn length(&self) -> f64 {
        if self.points.is_empty() {
            return 0.0;
        }

        let mut result = 0.0;
        let mut last_point = self.points[0];
        for point in &self.points[1..] {
            result += last_point.dist(*point);
            last_point = *point;
        }
        result += last_point.dist(self.points[0]);
        result
        // Alternatively, Iterator::zip() lets us iterate over the points as
pairs
        // but we need to pair each point with the next one, and the last point
        // with the first point. The zip() iterator is finished as soon as one
of
        // the source iterators is finished, a neat trick is to combine
Iterator::cycle
        // with Iterator::skip to create the second iterator for the zip and
using map
        // and sum to calculate the total length.
    }
}

pub struct Circle {
    center: Point,
    radius: i32,
}

impl Circle {
    pub fn new(center: Point, radius: i32) -> Circle {
        Circle { center, radius }
    }

    pub fn circumference(&self) -> f64 {
        2.0 * std::f64::consts::PI * f64::from(self.radius)
    }

    pub fn dist(&self, other: &Self) -> f64 {
        self.center.dist(other.center)
    }
}

pub enum Shape {
    Polygon(Polygon),
    Circle(Circle),
}

impl From<Polygon> for Shape {
    fn from(poly: Polygon) -> Self {
```

```
        Shape::Polygon(poly)
    }
}

impl From<Circle> for Shape {
    fn from(circle: Circle) -> Self {
        Shape::Circle(circle)
    }
}

impl Shape {
    pub fn perimeter(&self) -> f64 {
        match self {
            Shape::Polygon(poly) => poly.length(),
            Shape::Circle(circle) => circle.circumference(),
        }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    fn round_two_digits(x: f64) -> f64 {
        (x * 100.0).round() / 100.0
    }

    #[test]
    fn test_point_magnitude() {
        let p1 = Point::new(12, 13);
        assert_eq!(round_two_digits(p1.magnitude()), 17.69);
    }

    #[test]
    fn test_point_dist() {
        let p1 = Point::new(10, 10);
        let p2 = Point::new(14, 13);
        assert_eq!(round_two_digits(p1.dist(p2)), 5.00);
    }

    #[test]
    fn test_point_add() {
        let p1 = Point::new(16, 16);
        let p2 = p1 + Point::new(-4, 3);
        assert_eq!(p2, Point::new(12, 19));
    }

    #[test]
    fn test_polygon_left_most_point() {
        let p1 = Point::new(12, 13);
        let p2 = Point::new(16, 16);

        let mut poly = Polygon::new();
        poly.add_point(p1);
        poly.add_point(p2);
        assert_eq!(poly.left_most_point(), Some(p1));
    }
```

```
    #[test]
    fn test_polygon_iter() {
        let p1 = Point::new(12, 13);
        let p2 = Point::new(16, 16);

        let mut poly = Polygon::new();
        poly.add_point(p1);
        poly.add_point(p2);

        let points = poly.iter().cloned().collect::<Vec<_>>();
        assert_eq!(points, vec![Point::new(12, 13), Point::new(16, 16)]);
    }

    #[test]
    fn test_shape_perimeters() {
        let mut poly = Polygon::new();
        poly.add_point(Point::new(12, 13));
        poly.add_point(Point::new(17, 11));
        poly.add_point(Point::new(16, 16));
        let shapes = vec![
            Shape::from(poly),
            Shape::from(Circle::new(Point::new(10, 20), 5)),
        ];
        let perimeters = shapes
            .iter()
            .map(Shape::perimeter)
            .map(round_two_digits)
            .collect::<Vec<_>>();
        assert_eq!(perimeters, vec![15.48, 31.42]);
    }
}

fn main() {}
```

第三天下午的练习

安全 FFI 封装容器

(返回练习)

```
mod ffi {
    use std::os::raw::{c_char, c_int};
    #[cfg(not(target_os = "macos"))]
    use std::os::raw::{c_long, c_ulong, c_ushort, c_uchar};

    // Opaque type. See https://doc.rust-lang.org/nomicon/ffi.html.
    #[repr(C)]
    pub struct DIR {
        _data: [u8; 0],
        _marker: core::marker::PhantomData<(*mut u8,
core::marker::PhantomPinned)>,
    }

    // Layout according to the Linux man page for readdir(3), where ino_t and
    // off_t are resolved according to the definitions in
    // /usr/include/x86_64-linux-gnu/{sys/types.h, bits/typesizes.h}.
    #[cfg(not(target_os = "macos"))]
    #[repr(C)]
    pub struct dirent {
        pub d_ino: c_ulong,
        pub d_off: c_long,
        pub d_reclen: c_ushort,
        pub d_type: c_uchar,
        pub d_name: [c_char; 256],
    }

    // Layout according to the macOS man page for dir(5).
    #[cfg(all(target_os = "macos"))]
    #[repr(C)]
    pub struct dirent {
        pub d_fileno: u64,
        pub d_seekoff: u64,
        pub d_reclen: u16,
        pub d_namlen: u16,
        pub d_type: u8,
        pub d_name: [c_char; 1024],
    }

    extern "C" {
        pub fn opendir(s: *const c_char) -> *mut DIR;

        #[cfg(not(all(target_os = "macos", target_arch = "x86_64")))]
        pub fn readdir(s: *mut DIR) -> *const dirent;

        // See https://github.com/rust-lang/libc/issues/414 and the section on
        // _DARWIN_FEATURE_64_BIT_INODE in the macOS man page for stat(2).
        //
        // "Platforms that existed before these updates were available" refers
        // to macOS (as opposed to iOS / wearOS / etc.) on Intel and PowerPC.
        #[cfg(all(target_os = "macos", target_arch = "x86_64"))]
        #[link_name = "readdir$INODE64"]
        pub fn readdir(s: *mut DIR) -> *const dirent;

        pub fn closedir(s: *mut DIR) -> c_int;
    }
}
```

```
use std::ffi::{CStr, CString, OsStr, OsString};
use std::os::unix::ffi::OsStrExt;

#[derive(Debug)]
struct DirectoryIterator {
    path: CString,
    dir: *mut ffi::DIR,
}

impl DirectoryIterator {
    fn new(path: &str) -> Result<DirectoryIterator, String> {
        // Call opendir and return a Ok value if that worked,
        // otherwise return Err with a message.
        let path = CString::new(path).map_err(|err| format!("Invalid path:
{err}"))?;
        // SAFETY: path.as_ptr() cannot be NULL.
        let dir = unsafe { ffi::opendir(path.as_ptr()) };
        if dir.is_null() {
            Err(format!("Could not open {:?}", path))
        } else {
            Ok(DirectoryIterator { path, dir })
        }
    }
}

impl Iterator for DirectoryIterator {
    type Item = OsString;
    fn next(&mut self) -> Option<OsString> {
        // Keep calling readdir until we get a NULL pointer back.
        // SAFETY: self.dir is never NULL.
        let dirent = unsafe { ffi::readdir(self.dir) };
        if dirent.is_null() {
            // We have reached the end of the directory.
            return None;
        }
        // SAFETY: dirent is not NULL and dirent.d_name is NUL
        // terminated.
        let d_name = unsafe { CStr::from_ptr((*dirent).d_name.as_ptr()) };
        let os_str = OsStr::from_bytes(d_name.to_bytes());
        Some(os_str.to_owned())
    }
}

impl Drop for DirectoryIterator {
    fn drop(&mut self) {
        // Call closedir as needed.
        if !self.dir.is_null() {
            // SAFETY: self.dir is not NULL.
            if unsafe { ffi::closedir(self.dir) } != 0 {
                panic!("Could not close {:?}", self.path);
            }
        }
    }
}

fn main() -> Result<(), String> {
    let iter = DirectoryIterator::new(".")?;
    println!("files: {:#?}", iter.collect::<Vec<_>>());
```

```
    Ok(())
}

#[cfg(test)]
mod tests {
    use super::*;
    use std::error::Error;

    #[test]
    fn test_nonexisting_directory() {
        let iter = DirectoryIterator::new("no-such-directory");
        assert!(iter.is_err());
    }

    #[test]
    fn test_empty_directory() -> Result<(), Box<dyn Error>> {
        let tmp = tempfile::TempDir::new()?;
        let iter = DirectoryIterator::new(
            tmp.path().to_str().ok_or("Non UTF-8 character in path")?,
        )?;
        let mut entries = iter.collect::<Vec<_>>();
        entries.sort();
        assert_eq!(entries, &[".", ".."]);
        Ok(())
    }

    #[test]
    fn test_nonempty_directory() -> Result<(), Box<dyn Error>> {
        let tmp = tempfile::TempDir::new()?;
        std::fs::write(tmp.path().join("foo.txt"), "The Foo Diaries\n")?;
        std::fs::write(tmp.path().join("bar.png"), "<PNG>\n")?;
        std::fs::write(tmp.path().join("crab.rs"), "//! Crab\n")?;
        let iter = DirectoryIterator::new(
            tmp.path().to_str().ok_or("Non UTF-8 character in path")?,
        )?;
        let mut entries = iter.collect::<Vec<_>>();
        entries.sort();
        assert_eq!(entries, &[".", "..", "bar.png", "crab.rs", "foo.txt"]);
        Ok(())
    }
}
```

裸机 Rust 上午练习

罗盘

(返回练习)

```rust
#![no_main]
#![no_std]

extern crate panic_halt as _;

use core::fmt::Write;
use cortex_m_rt::entry;
use core::cmp::{max, min};
use lsm303agr::{AccelOutputDataRate, Lsm303agr, MagOutputDataRate};
use microbit::display::blocking::Display;
use microbit::hal::prelude::*;
use microbit::hal::twim::Twim;
use microbit::hal::uarte::{Baudrate, Parity, Uarte};
use microbit::hal::Timer;
use microbit::pac::twim0::frequency::FREQUENCY_A;
use microbit::Board;

const COMPASS_SCALE: i32 = 30000;
const ACCELEROMETER_SCALE: i32 = 700;

#[entry]
fn main() -> ! {
    let board = Board::take().unwrap();

    // Configure serial port.
    let mut serial = Uarte::new(
        board.UARTE0,
        board.uart.into(),
        Parity::EXCLUDED,
        Baudrate::BAUD115200,
    );

    // Set up the I2C controller and Inertial Measurement Unit.
    writeln!(serial, "Setting up IMU...").unwrap();
    let i2c = Twim::new(board.TWIM0, board.i2c_internal.into(),
FREQUENCY_A::K100);
    let mut imu = Lsm303agr::new_with_i2c(i2c);
    imu.init().unwrap();
    imu.set_mag_odr(MagOutputDataRate::Hz50).unwrap();
    imu.set_accel_odr(AccelOutputDataRate::Hz50).unwrap();
    let mut imu = imu.into_mag_continuous().ok().unwrap();

    // Set up display and timer.
    let mut timer = Timer::new(board.TIMER0);
    let mut display = Display::new(board.display_pins);

    let mut mode = Mode::Compass;
    let mut button_pressed = false;

    writeln!(serial, "Ready.").unwrap();

    loop {
        // Read compass data and log it to the serial port.
        while !(imu.mag_status().unwrap().xyz_new_data
            && imu.accel_status().unwrap().xyz_new_data)
        {}
        let compass_reading = imu.mag_data().unwrap();
```

```
        let accelerometer_reading = imu.accel_data().unwrap();
        writeln!(
            serial,
            "{},{},{}\t{},{},{}",
            compass_reading.x,
            compass_reading.y,
            compass_reading.z,
            accelerometer_reading.x,
            accelerometer_reading.y,
            accelerometer_reading.z,
        )
        .unwrap();

        let mut image = [[0; 5]; 5];
        let (x, y) = match mode {
            Mode::Compass => (
                scale(-compass_reading.x, -COMPASS_SCALE, COMPASS_SCALE, 0, 4)
as usize,
                scale(compass_reading.y, -COMPASS_SCALE, COMPASS_SCALE, 0, 4)
as usize,
            ),
            Mode::Accelerometer => (
                scale(
                    accelerometer_reading.x,
                    -ACCELEROMETER_SCALE,
                    ACCELEROMETER_SCALE,
                    0,
                    4,
                ) as usize,
                scale(
                    -accelerometer_reading.y,
                    -ACCELEROMETER_SCALE,
                    ACCELEROMETER_SCALE,
                    0,
                    4,
                ) as usize,
            ),
        };
        image[y][x] = 255;
        display.show(&mut timer, image, 100);

        // If button A is pressed, switch to the next mode and briefly blink
all LEDs on.
        if board.buttons.button_a.is_low().unwrap() {
            if !button_pressed {
                mode = mode.next();
                display.show(&mut timer, [[255; 5]; 5], 200);
            }
            button_pressed = true;
        } else {
            button_pressed = false;
        }
    }
}

#[derive(Copy, Clone, Debug, Eq, PartialEq)]
enum Mode {
    Compass,
```

```
    Accelerometer,
}

impl Mode {
    fn next(self) -> Self {
        match self {
            Self::Compass => Self::Accelerometer,
            Self::Accelerometer => Self::Compass,
        }
    }
}

fn scale(value: i32, min_in: i32, max_in: i32, min_out: i32, max_out: i32) ->
i32 {
    let range_in = max_in - min_in;
    let range_out = max_out - min_out;
    cap(
        min_out + range_out * (value - min_in) / range_in,
        min_out,
        max_out,
    )
}

fn cap(value: i32, min_value: i32, max_value: i32) -> i32 {
    max(min_value, min(value, max_value))
}
```

嵌入式 Rust：进阶篇

RTC driver

(返回练习)

main.rs:

```
#![no_main]
#![no_std]

mod exceptions;
mod logger;
mod pl011;
mod pl031;

use crate::pl031::Rtc;
use arm_gic::gicv3::{IntId, Trigger};
use arm_gic::{irq_enable, wfi};
use chrono::{TimeZone, Utc};
use core::hint::spin_loop;
use crate::pl011::Uart;
use arm_gic::gicv3::GicV3;
use core::panic::PanicInfo;
use log::{error, info, trace, LevelFilter};
use smccc::psci::system_off;
use smccc::Hvc;

/// Base addresses of the GICv3.
const GICD_BASE_ADDRESS: *mut u64 = 0x800_0000 as _;
const GICR_BASE_ADDRESS: *mut u64 = 0x80A_0000 as _;

/// Base address of the primary PL011 UART.
const PL011_BASE_ADDRESS: *mut u32 = 0x900_0000 as _;

/// Base address of the PL031 RTC.
const PL031_BASE_ADDRESS: *mut u32 = 0x901_0000 as _;
/// The IRQ used by the PL031 RTC.
const PL031_IRQ: IntId = IntId::spi(2);

#[no_mangle]
extern "C" fn main(x0: u64, x1: u64, x2: u64, x3: u64) {
    // Safe because `PL011_BASE_ADDRESS` is the base address of a PL011 device,
    // and nothing else accesses that address range.
    let uart = unsafe { Uart::new(PL011_BASE_ADDRESS) };
    logger::init(uart, LevelFilter::Trace).unwrap();

    info!("main({:#x}, {:#x}, {:#x}, {:#x})", x0, x1, x2, x3);

    // Safe because `GICD_BASE_ADDRESS` and `GICR_BASE_ADDRESS` are the base
    // addresses of a GICv3 distributor and redistributor respectively, and
    // nothing else accesses those address ranges.
    let mut gic = unsafe { GicV3::new(GICD_BASE_ADDRESS, GICR_BASE_ADDRESS) };
    gic.setup();

    // Safe because `PL031_BASE_ADDRESS` is the base address of a PL031 device,
    // and nothing else accesses that address range.
    let mut rtc = unsafe { Rtc::new(PL031_BASE_ADDRESS) };
    let timestamp = rtc.read();
    let time = Utc.timestamp_opt(timestamp.into(), 0).unwrap();
    info!("RTC: {time}");

    GicV3::set_priority_mask(0xff);
    gic.set_interrupt_priority(PL031_IRQ, 0x80);
    gic.set_trigger(PL031_IRQ, Trigger::Level);
```

```
    irq_enable();
    gic.enable_interrupt(PL031_IRQ, true);

    // Wait for 3 seconds, without interrupts.
    let target = timestamp + 3;
    rtc.set_match(target);
    info!(
        "Waiting for {}",
        Utc.timestamp_opt(target.into(), 0).unwrap()
    );
    trace!(
        "matched={}, interrupt_pending={}",
        rtc.matched(),
        rtc.interrupt_pending()
    );
    while !rtc.matched() {
        spin_loop();
    }
    trace!(
        "matched={}, interrupt_pending={}",
        rtc.matched(),
        rtc.interrupt_pending()
    );
    info!("Finished waiting");

    // Wait another 3 seconds for an interrupt.
    let target = timestamp + 6;
    info!(
        "Waiting for {}",
        Utc.timestamp_opt(target.into(), 0).unwrap()
    );
    rtc.set_match(target);
    rtc.clear_interrupt();
    rtc.enable_interrupt(true);
    trace!(
        "matched={}, interrupt_pending={}",
        rtc.matched(),
        rtc.interrupt_pending()
    );
    while !rtc.interrupt_pending() {
        wfi();
    }
    trace!(
        "matched={}, interrupt_pending={}",
        rtc.matched(),
        rtc.interrupt_pending()
    );
    info!("Finished waiting");

    system_off::<Hvc>().unwrap();
}

#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    error!("{info}");
    system_off::<Hvc>().unwrap();
```

```
        loop {}
    }
```

pl031.rs:

```rust
use core::ptr::{addr_of, addr_of_mut};

#[repr(C, align(4))]
struct Registers {
    /// Data register
    dr: u32,
    /// Match register
    mr: u32,
    /// Load register
    lr: u32,
    /// Control register
    cr: u8,
    _reserved0: [u8; 3],
    /// Interrupt Mask Set or Clear register
    imsc: u8,
    _reserved1: [u8; 3],
    /// Raw Interrupt Status
    ris: u8,
    _reserved2: [u8; 3],
    /// Masked Interrupt Status
    mis: u8,
    _reserved3: [u8; 3],
    /// Interrupt Clear Register
    icr: u8,
    _reserved4: [u8; 3],
}

/// Driver for a PL031 real-time clock.
#[derive(Debug)]
pub struct Rtc {
    registers: *mut Registers,
}

impl Rtc {
    /// Constructs a new instance of the RTC driver for a PL031 device at the
    /// given base address.
    ///
    /// # Safety
    ///
    /// The given base address must point to the MMIO control registers of a
    /// PL031 device, which must be mapped into the address space of the
process
    /// as device memory and not have any other aliases.
    pub unsafe fn new(base_address: *mut u32) -> Self {
        Self {
            registers: base_address as *mut Registers,
        }
    }

    /// Reads the current RTC value.
    pub fn read(&self) -> u32 {
        // Safe because we know that self.registers points to the control
        // registers of a PL031 device which is appropriately mapped.
        unsafe { addr_of!((*self.registers).dr).read_volatile() }
    }

    /// Writes a match value. When the RTC value matches this then an interrupt
```

```
    /// will be generated (if it is enabled).
    pub fn set_match(&mut self, value: u32) {
        // Safe because we know that self.registers points to the control
        // registers of a PL031 device which is appropriately mapped.
        unsafe { addr_of_mut!((*self.registers).mr).write_volatile(value) }
    }

    /// Returns whether the match register matches the RTC value, whether or
not
    /// the interrupt is enabled.
    pub fn matched(&self) -> bool {
        // Safe because we know that self.registers points to the control
        // registers of a PL031 device which is appropriately mapped.
        let ris = unsafe { addr_of!((*self.registers).ris).read_volatile() };
        (ris & 0x01) != 0
    }

    /// Returns whether there is currently an interrupt pending.
    ///
    /// This should be true if and only if `matched` returns true and the
    /// interrupt is masked.
    pub fn interrupt_pending(&self) -> bool {
        // Safe because we know that self.registers points to the control
        // registers of a PL031 device which is appropriately mapped.
        let ris = unsafe { addr_of!((*self.registers).mis).read_volatile() };
        (ris & 0x01) != 0
    }

    /// Sets or clears the interrupt mask.
    ///
    /// When the mask is true the interrupt is enabled; when it is false the
    /// interrupt is disabled.
    pub fn enable_interrupt(&mut self, mask: bool) {
        let imsc = if mask { 0x01 } else { 0x00 };
        // Safe because we know that self.registers points to the control
        // registers of a PL031 device which is appropriately mapped.
        unsafe { addr_of_mut!((*self.registers).imsc).write_volatile(imsc) }
    }

    /// Clears a pending interrupt, if any.
    pub fn clear_interrupt(&mut self) {
        // Safe because we know that self.registers points to the control
        // registers of a PL031 device which is appropriately mapped.
        unsafe { addr_of_mut!((*self.registers).icr).write_volatile(0x01) }
    }
}

// Safe because it just contains a pointer to device memory, which can be
// accessed from any context.
unsafe impl Send for Rtc {}
```

并发编程：上午练习

哲学家就餐问题 (Dining philosophers problem)

(返回练习)

```rust
use std::sync::{mpsc, Arc, Mutex};
use std::thread;
use std::time::Duration;

struct Fork;

struct Philosopher {
    name: String,
    left_fork: Arc<Mutex<Fork>>,
    right_fork: Arc<Mutex<Fork>>,
    thoughts: mpsc::SyncSender<String>,
}

impl Philosopher {
    fn think(&self) {
        self.thoughts
            .send(format!("Eureka! {} has a new idea!", &self.name))
            .unwrap();
    }

    fn eat(&self) {
        println!("{} is trying to eat", &self.name);
        let left = self.left_fork.lock().unwrap();
        let right = self.right_fork.lock().unwrap();

        println!("{} is eating...", &self.name);
        thread::sleep(Duration::from_millis(10));
    }
}

static PHILOSOPHERS: &[&str] =
    &["Socrates", "Plato", "Aristotle", "Thales", "Pythagoras"];

fn main() {
    let (tx, rx) = mpsc::sync_channel(10);

    let forks = (0..PHILOSOPHERS.len())
        .map(|_| Arc::new(Mutex::new(Fork)))
        .collect::<Vec<_>>();

    for i in 0..forks.len() {
        let tx = tx.clone();
        let mut left_fork = Arc::clone(&forks[i]);
        let mut right_fork = Arc::clone(&forks[(i + 1) % forks.len()]);

        // To avoid a deadlock, we have to break the symmetry
        // somewhere. This will swap the forks without deinitializing
        // either of them.
        if i == forks.len() - 1 {
            std::mem::swap(&mut left_fork, &mut right_fork);
        }

        let philosopher = Philosopher {
            name: PHILOSOPHERS[i].to_string(),
            thoughts: tx,
            left_fork,
            right_fork,
```

```
        };

        thread::spawn(move || {
            for _ in 0..100 {
                philosopher.eat();
                philosopher.think();
            }
        });
    }

    drop(tx);
    for thought in rx {
        println!("{thought}");
    }
}
```

Link Checker

(back to exercise)

```rust
use std::{sync::Arc, sync::Mutex, sync::mpsc, thread};

use reqwest::{blocking::Client, Url};
use scraper::{Html, Selector};
use thiserror::Error;

#[derive(Error, Debug)]
enum Error {
    #[error("request error: {0}")]
    ReqwestError(#[from] reqwest::Error),
    #[error("bad http response: {0}")]
    BadResponse(String),
}

#[derive(Debug)]
struct CrawlCommand {
    url: Url,
    extract_links: bool,
}

fn visit_page(client: &Client, command: &CrawlCommand) -> Result<Vec<Url>,
Error> {
    println!("Checking {:#}", command.url);
    let response = client.get(command.url.clone()).send()?;
    if !response.status().is_success() {
        return Err(Error::BadResponse(response.status().to_string()));
    }

    let mut link_urls = Vec::new();
    if !command.extract_links {
        return Ok(link_urls);
    }

    let base_url = response.url().to_owned();
    let body_text = response.text()?;
    let document = Html::parse_document(&body_text);

    let selector = Selector::parse("a").unwrap();
    let href_values = document
        .select(&selector)
        .filter_map(|element| element.value().attr("href"));
    for href in href_values {
        match base_url.join(href) {
            Ok(link_url) => {
                link_urls.push(link_url);
            }
            Err(err) => {
                println!("On {base_url:#}: ignored unparsable {href:?}:
{err}");
            }
        }
    }
    Ok(link_urls)
}

struct CrawlState {
    domain: String,
```

```
    visited_pages: std::collections::HashSet<String>,
}

impl CrawlState {
    fn new(start_url: &Url) -> CrawlState {
        let mut visited_pages = std::collections::HashSet::new();
        visited_pages.insert(start_url.as_str().to_string());
        CrawlState {
            domain: start_url.domain().unwrap().to_string(),
            visited_pages,
        }
    }

    /// Determine whether links within the given page should be extracted.
    fn should_extract_links(&self, url: &Url) -> bool {
        let Some(url_domain) = url.domain() else {
            return false;
        };
        url_domain == self.domain
    }

    /// Mark the given page as visited, returning true if it had already
    /// been visited.
    fn mark_visited(&mut self, url: &Url) -> bool {
        self.visited_pages.insert(url.as_str().to_string())
    }
}

type CrawlResult = Result<Vec<Url>, (Url, Error)>;
fn spawn_crawler_threads(
    command_receiver: mpsc::Receiver<CrawlCommand>,
    result_sender: mpsc::Sender<CrawlResult>,
    thread_count: u32,
) {
    let command_receiver = Arc::new(Mutex::new(command_receiver));

    for _ in 0..thread_count {
        let result_sender = result_sender.clone();
        let command_receiver = command_receiver.clone();
        thread::spawn(move || {
            let client = Client::new();
            loop {
                let command_result = {
                    let receiver_guard = command_receiver.lock().unwrap();
                    receiver_guard.recv()
                };
                let Ok(crawl_command) = command_result else {
                    // The sender got dropped. No more commands coming in.
                    break;
                };
                let crawl_result = match visit_page(&client, &crawl_command) {
                    Ok(link_urls) => Ok(link_urls),
                    Err(error) => Err((crawl_command.url, error)),
                };
                result_sender.send(crawl_result).unwrap();
            }
        });
    }
```

```
}

fn control_crawl(
    start_url: Url,
    command_sender: mpsc::Sender<CrawlCommand>,
    result_receiver: mpsc::Receiver<CrawlResult>,
) -> Vec<Url> {
    let mut crawl_state = CrawlState::new(&start_url);
    let start_command = CrawlCommand { url: start_url, extract_links: true };
    command_sender.send(start_command).unwrap();
    let mut pending_urls = 1;

    let mut bad_urls = Vec::new();
    while pending_urls > 0 {
        let crawl_result = result_receiver.recv().unwrap();
        pending_urls -= 1;

        match crawl_result {
            Ok(link_urls) => {
                for url in link_urls {
                    if crawl_state.mark_visited(&url) {
                        let extract_links =
crawl_state.should_extract_links(&url);
                        let crawl_command = CrawlCommand { url, extract_links
};
                        command_sender.send(crawl_command).unwrap();
                        pending_urls += 1;
                    }
                }
            }
            Err((url, error)) => {
                bad_urls.push(url);
                println!("Got crawling error: {:#}", error);
                continue;
            }
        }
    }
    bad_urls
}

fn check_links(start_url: Url) -> Vec<Url> {
    let (result_sender, result_receiver) = mpsc::channel::<CrawlResult>();
    let (command_sender, command_receiver) = mpsc::channel::<CrawlCommand>();
    spawn_crawler_threads(command_receiver, result_sender, 16);
    control_crawl(start_url, command_sender, result_receiver)
}

fn main() {
    let start_url = reqwest::Url::parse("https://www.google.org").unwrap();
    let bad_urls = check_links(start_url);
    println!("Bad URLs: {:#?}", bad_urls);
}
```

并发编程：下午练习

哲学家进餐 - 异步

(返回练习)

```rust
use std::sync::Arc;
use tokio::time;
use tokio::sync::mpsc::{self, Sender};
use tokio::sync::Mutex;

struct Fork;

struct Philosopher {
    name: String,
    left_fork: Arc<Mutex<Fork>>,
    right_fork: Arc<Mutex<Fork>>,
    thoughts: Sender<String>,
}

impl Philosopher {
    async fn think(&self) {
        self.thoughts
            .send(format!("Eureka! {} has a new idea!", &self.name)).await
            .unwrap();
    }

    async fn eat(&self) {
        // Pick up forks...
        let _first_lock = self.left_fork.lock().await;
        // Add a delay before picking the second fork to allow the execution
        // to transfer to another task
        time::sleep(time::Duration::from_millis(1)).await;
        let _second_lock = self.right_fork.lock().await;

        println!("{} is eating...", &self.name);
        time::sleep(time::Duration::from_millis(5)).await;

        // The locks are dropped here
    }
}

static PHILOSOPHERS: &[&str] =
    &["Socrates", "Plato", "Aristotle", "Thales", "Pythagoras"];

#[tokio::main]
async fn main() {
    // Create forks
    let mut forks = vec![];
    (0..PHILOSOPHERS.len()).for_each(|_|
forks.push(Arc::new(Mutex::new(Fork))));

    // Create philosophers
    let (philosophers, mut rx) = {
        let mut philosophers = vec![];
        let (tx, rx) = mpsc::channel(10);
        for (i, name) in PHILOSOPHERS.iter().enumerate() {
            let left_fork = Arc::clone(&forks[i]);
            let right_fork = Arc::clone(&forks[(i + 1) % PHILOSOPHERS.len()]);
            // To avoid a deadlock, we have to break the symmetry
            // somewhere. This will swap the forks without deinitializing
            // either of them.
            if i  == 0 {
```

```
                std::mem::swap(&mut left_fork, &mut right_fork);
            }
            philosophers.push(Philosopher {
                name: name.to_string(),
                left_fork,
                right_fork,
                thoughts: tx.clone(),
            });
        }
        (philosophers, rx)
        // tx is dropped here, so we don't need to explicitly drop it later
    };

    // Make them think and eat
    for phil in philosophers {
        tokio::spawn(async move {
            for _ in 0..100 {
                phil.think().await;
                phil.eat().await;
            }
        });

    }

    // Output their thoughts
    while let Some(thought) = rx.recv().await {
        println!("Here is a thought: {thought}");
    }
}
```

广播聊天应用程序

(返回练习)

src/bin/server.rs:

```rust
use futures_util::sink::SinkExt;
use futures_util::stream::StreamExt;
use std::error::Error;
use std::net::SocketAddr;
use tokio::net::{TcpListener, TcpStream};
use tokio::sync::broadcast::{channel, Sender};
use tokio_websockets::{Message, ServerBuilder, WebsocketStream};

async fn handle_connection(
    addr: SocketAddr,
    mut ws_stream: WebsocketStream<TcpStream>,
    bcast_tx: Sender<String>,
) -> Result<(), Box<dyn Error + Send + Sync>> {

    ws_stream
        .send(Message::text("Welcome to chat! Type a message".into()))
        .await?;
    let mut bcast_rx = bcast_tx.subscribe();

    // A continuous loop for concurrently performing two tasks: (1) receiving
    // messages from `ws_stream` and broadcasting them, and (2) receiving
    // messages on `bcast_rx` and sending them to the client.
    loop {
        tokio::select! {
            incoming = ws_stream.next() => {
                match incoming {
                    Some(Ok(msg)) => {
                        if let Some(text) = msg.as_text() {
                            println!("From client {addr:?} {text:?}");
                            bcast_tx.send(text.into())?;
                        }
                    }
                    Some(Err(err)) => return Err(err.into()),
                    None => return Ok(()),
                }
            }
            msg = bcast_rx.recv() => {
                ws_stream.send(Message::text(msg?)).await?;
            }
        }
    }
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn Error + Send + Sync>> {
    let (bcast_tx, _) = channel(16);

    let listener = TcpListener::bind("127.0.0.1:2000").await?;
    println!("listening on port 2000");

    loop {
        let (socket, addr) = listener.accept().await?;
        println!("New connection from {addr:?}");
        let bcast_tx = bcast_tx.clone();
        tokio::spawn(async move {
            // Wrap the raw TCP stream into a websocket.
            let ws_stream = ServerBuilder::new().accept(socket).await?;
```

```
            handle_connection(addr, ws_stream, bcast_tx).await
        });
    }
}
```

src/bin/client.rs:

```
use futures_util::stream::StreamExt;
use futures_util::SinkExt;
use http::Uri;
use tokio::io::{AsyncBufReadExt, BufReader};
use tokio_websockets::{ClientBuilder, Message};

#[tokio::main]
async fn main() -> Result<(), tokio_websockets::Error> {
    let (mut ws_stream, _) =
        ClientBuilder::from_uri(Uri::from_static("ws://127.0.0.1:2000"))
            .connect()
            .await?;

    let stdin = tokio::io::stdin();
    let mut stdin = BufReader::new(stdin).lines();

    // Continuous loop for concurrently sending and receiving messages.
    loop {
        tokio::select! {
            incoming = ws_stream.next() => {
                match incoming {
                    Some(Ok(msg)) => {
                        if let Some(text) = msg.as_text() {
                            println!("From server: {}", text);
                        }
                    },
                    Some(Err(err)) => return Err(err.into()),
                    None => return Ok(()),
                }
            }
            res = stdin.next_line() => {
                match res {
                    Ok(None) => return Ok(()),
                    Ok(Some(line)) =>
ws_stream.send(Message::text(line.to_string())).await?,
                    Err(err) => return Err(err.into()),
                }
            }

        }
    }
}
```