Rust 语言学习笔记

感谢 RustPrimer 和 Rust 程序设计语言简体中文版

bradyjoestar@gmail.com

目录

序

这份学习笔记是在学习 Rust 的过程中的记录，主要基于《Rust 程序设计语言-简体中文版》和《RustPrimer》两份开源书籍。

上面两本书籍相对全面，但对部分初学者可能更为深奥一些，尤其是之前没有接触过 C/C++ 和 Rust 语言的读者。在学习过程中我对二份开源书籍进行了适合的整理，又加入了一些自己学习过程中新添加的内容，于是就有了这份学习笔记。

我个人对 Rust 语言设计的看法主要是：重大创新，却又博采众长。

Rust 为了解决内存安全问题重新设计了类型系统，提出了所有权的概念，同时为了能够解决当前大多数语言无法检测到的运行时错误，rust 创造性地设计了无畏并发。同时还可以看到 Rust 借鉴了很多优秀语言的设计理念，同时再加快速迭代的社区，都是 Rust 受到赞赏的重要因素。

Rust 是一门系统级编程语言，被设计为保证内存和线程安全，并防止段错误。作为系统级编程语言，它的基本理念是 “零开销抽象”。理论上来说，它的速度与 C / C++ 同级。

Rust 可以被归为通用的、多范式、编译型的编程语言，类似 C 或者 C++。与这两门编程语言不同的是，Rust 是线程安全的！

Rust 编程语言的目标是，创建一个高度安全和并发的软件系统。它强调安全性、并发和内存控制。尽管 Rust 借用了 C 和 C++ 的语法，它不允许空指针和悬挂指针，二者是 C 和 C++ 中系统崩溃、内存泄露和不安全代码的根源。

Rust 中有诸如 if else 和循环语句 for 和 while 的通用控制结构。和 C 和 C++ 风格的编程语言一样，代码段放在花括号中。

Rust 使用实现（implementation）、特征（trait）和结构化类型（structured type）而不是类（class）。这点，与基于继承的 OO 语言 C++, Java 有相当大的差异。而跟 Ocaml, Haskell 这类函数式语言更加接近。

Rust 做到了内存安全而无需 .NET 和 Java 编程语言中实现自动垃圾收集器的开销，这是通过所有权/借用机制、生命周期、以及类型系统来达到的。

Rust 程序设计语言的本质在于**赋能**（*empowerment*）：无论你现在编写的是何种代码，Rust 能让你在更为广泛的编程领域走得更远，写出自信。

比如，“系统层面”（“systems-level”）的工作，涉及内存管理、数据表示和并发等底层细节。从传统角度来看，这是一个神秘的编程领域，只为浸淫多年的极少数人所触及，也只有他们能避开那些臭名昭著的陷阱。即使谨慎的实践者，亦唯恐代码出现漏洞、崩溃或损坏。

Rust 破除了这些障碍，其消除了旧的陷阱并提供了伴你一路同行的友好、精良的工具。想要 “深入” 底层控制的程序员可以使用 Rust，无需冒着常见的崩

溃或安全漏洞的风险，也无需学习时常改变的工具链的最新知识。其语言本身更是被设计为自然而然的引导你编写出在运行速度和内存使用上都十分高效的可靠代码。

参考：

https://rustcc.gitbooks.io/rustprimer/content/ 《RustPrimer》

https://kaisery.github.io/trpl-zh-cn/ 《Rust 程序设计语言-简体中文版》

第一章 包管理系统与版本管理工具

包管理系统是所有语言向工程化方向走必须考虑的事情。

rust 的包管理系统和 go 的包管理系统以及 java 的包管理系统大大不同，很容易给人造成困惑。

最主要原因是：

1.Rust 的模块支持层级结构，但这种层级结构本身与文件系统目录的层级结构是解耦的。

因为 Rust 本身可用于操作系统的开发。

开发者需要自己去定义路径，定义 mod 的层级关系，配合 rust 的默认约定。这点和 java，go 开发完全不同，在面向 vm 的语言中这些都不需要考虑。

2.Rust 的包管理系统中使用了大量的默认约定，很容易使人头昏脑乱。

而在开发中我们又必须建立模块层级系统，rust 给出了如下方案，在给出了一些模块的最基本规则外，由开发者更大范围地自定义模块的存在。

首先在一个 rust 项目中，首先定义了 crate 和 module。

1.1 Crate

1.crate 编译后会形成一个库(例如.so)或二进制可执行文件。crate 分为两种：lib crate 和 bin crate。

2. 一个包可以带有零个或一个 lib crate 和任意多个 bin crate。一个包中必须有 crate，至少一个，（lib crate 或 bin crate 都可以）

3.通常写 rust 项目时非常依赖 crate，很多重要的信息都是配置在 cargo.toml 文件中，不仅仅包括 lib 和 crate 的入口文件，后面还有很多的 attribute。

4.rust 对于 crate 的 layout 有一些默认的约定：

i.Cargo 约定如果在代表包的 Cargo.toml 的同级目录下包含 src 目录且其中包含 main.rs 文件的话，Cargo 就知道这个包带有一个与包同名的 bin crate，且 src/main.rs 就是 crate 根。不用在写 cargo.toml 的时候精确到文件。

ii.另一个约定如果包目录中包含 src/lib.rs，则包带有与其同名的 lib crate，且 src/lib.rs 是 crate 根。同样不需要精确到文件。

iii. 包可以带有多个二进制 crate，默认将文件置于 src/bin 目录，但是也可以自由配置。

举例：

[[bin]]

name = “base_language_demo”

会自动去寻找 src/bin/base_language_demo.rs 作为 bin crate 的编译入口。

[[bin]]

name = “src/bin_build_demo/bin_test.rs”

非常清晰地指明了文件名，直接以 src/bin_build_demo/bin_test.rs 作为编译入口。

1.1.1 Cargo

rust 官方参考了现有语言管理工具的优点，于是就产生了 cargo。主要是为了减少复杂的项目管理配置参数。cargo 工具是官方正统出身。

在 cargo.toml 中不配置唯一的 lib crate 和 bin crate name 的话，会自动去根据 package 进行命名。

约定的补充：

cargo.toml 和 cargo.lock 文件总是位于项目根目录下。

源代码位于 src 目录下。

默认的库入口文件是 src/lib.rs。

默认的可执行程序入口文件是 src/main.rs。

其他可选的可执行文件位于 src/bin/*.rs(这里每一个 rs 文件均对应一个可执行文件)。

外部测试源代码文件位于 tests 目录下。

示例程序源代码文件位于 examples。

基准测试源代码文件位于 benches 目录下。

cargo.toml 是 cargo 特有的项目数据描述文件，对于猿们而言，cargo.toml 文件存储了项目的所有信息，它直接面向 rustacean，如果想让自己的 rust 项目能够按照期望的方式进行构建、测试和运行，那么，必须按照合理的方式构建’cargo.toml’。

而 cargo.lock 文件则不直接面向开发者，也不需要直接去修改这个文件。lock 文件是 cargo 工具根据同一项目的 toml 文件生成的项目依赖详细清单文件。

Cargo 字段：

1.[package]段落描述了软件开发者对本项目的各种元数据描述信息。

2.[dependency]

3.单元测试主要通过在项目代码的测试代码部分前用#[test]属性来描述，而集成测试，则一般都会通过 toml 文件中的[[test]]段落进行描述

4.example 用例的描述以及 bin 用例的描述。其描述方法和 test 用例描述方法类似。不过，这时候段落名称'[[test]]'分别替换为：'[[example]]'或者'[[bin]]'

1.2 module

Rust 提供了一个关键字 mod，它主要起到两个用途，在一个文件中**定义**一个模块，或者**引用**另外一个文件中的模块。

模块也有一些默认的约定：

1.每个 crate 中，默认实现了一个隐式的根模块（root module）；

2.模块的命名风格也是 lower_snake_case，跟其它的 Rust 的标识符一样；

3.模块可以嵌套；

4.模块中可以写任何合法的 Rust 代码；

为了让外部能使用模块中 item，需要使用 pub 关键字。外部引用的时候，使用 use 关键字。

1.2.1 module 的可见性

为了让外部能使用模块中 item，需要使用 pub 关键字。外部引用的时候，使用 use 关键字。

规则很简单，一个 item（函数，绑定，Trait 等），前面加了 pub，那么就它变成对外可见（访问，调用）的了。

1.2.2 引用外部文件模块

通常，我们会在单独的文件中写模块内容，然后使用 mod 关键字来加载那个文件作为我们的模块。

比如，我们在 src 下新建了文件 aaa.rs。现在目录结构是下面这样子：

```
foo
├── Cargo.toml
└── src
    └── aaa.rs
    └── main.rs
```

我们在 aaa.rs 中，写上：

```rust
pub fn print_aaa() {
    println!("{}", 25);
}
```

在 main.rs 中，写上：

```rust
mod aaa;

use self::aaa::print_aaa;

fn main () {
    print_aaa();
}
```

编译后，生成一个可执行文件。

细心的朋友会发现，aaa.rs 中，没有使用 mod xxx {} 这样包裹起来，是因为 mod xxx; 相当于把 xxx.rs 文件用 mod xxx {} 包裹起来了。（又一个约定）初学者往往会多加一层，请注意。

1.2.3 多文件模块的层级关系

Rust 的模块支持层级结构，但这种层级结构本身与文件系统目录的层级结构是解耦的。

mod xxx; 这个 xxx 不能包含::号。也即在这个表达形式中，是没法引用多层结构下的模块的。也即，你不可能直接使用 mod a::b::c::d;的形式来引用 a/b/c/d.rs 这个模块。

换句话说，必须依靠 rust 的默认约定去由开发去建立层级关系。rust 的层级关系是我们自己依靠默认规则自己定义出来的！

那么，Rust 的多层模块的定义查询遵循如下两条规则：

1.优先查找 xxx.rs 文件

2.main.rs、lib.rs、mod.rs 中的 mod xxx;默认**优先查找**同级目录下的 xxx.rs 文件；

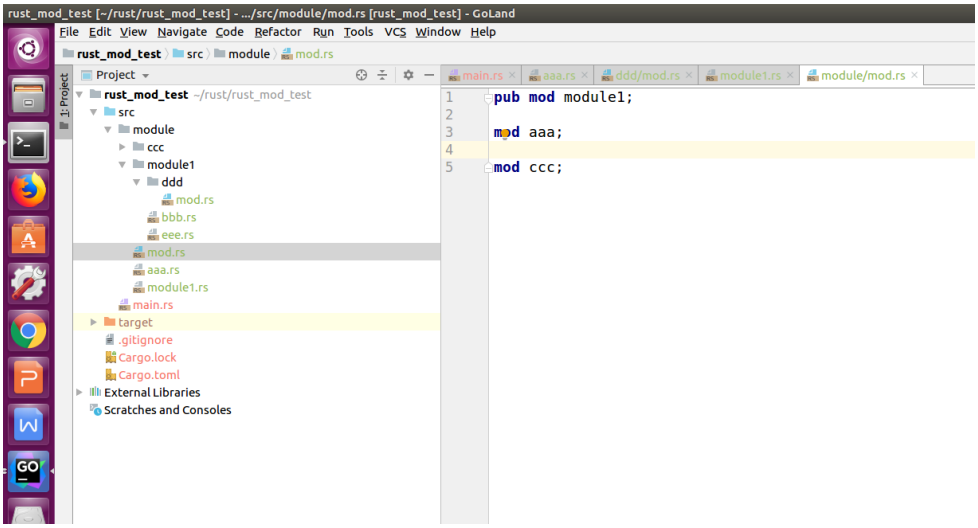
其他文件 yyy.rs 中的 mod xxx;默认**优先查找**同级目录的 yyy 目录下的 xxx.rs 文件；

如果 xxx.rs 不存在，则**查找** xxx/mod.rs 文件，即 xxx 目录下的 mod.rs 文件。

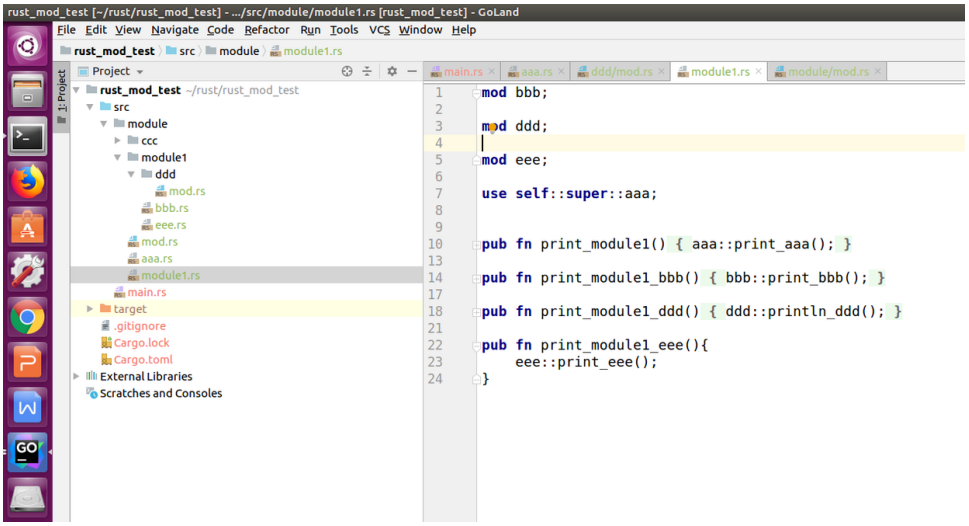
先不要去考虑默认不默认的问题，优先考虑尽可能不要让定义的 mod xxx 有两种解释。另外每个 xxx 只定义一次。

例子：

1.默认优先查找 module1.rs 文件



2. module1.rs 中的 mod.xxx 查找 module1 目录下的 xxx.rs 文件。



可以看到，module1 下没有 mod.rs 文件，避免歧义。

1.2.4 module 路径

前面我们提到，一个 crate 是一个独立的可编译单元。它有一个入口文件，这个入口文件是这个 crate（里面可能包含若干个 module）的模块根路径。整个模块的引用，形成一个链，每个模块，都可以用一个精确的路径（比如：a::b::c::d）来表示；

与文件系统概念类似，模块路径也有相对路径和绝对路径的概念。为此，Rust 提供了 self 和 super 两个关键字。

路径是自定义出来的！

super 表示，当前模块路径的上一级路径，可以理解成父模块。

另外，还有一种特殊的路径形式：

```
::xxx::yyy
```
它表示，引用根路径下的 xxx::yyy，这个根路径，指的是当前 crate 的根路径。

1.2.5 Re-exporting

我们可以结合使用 pub use 来实现 Re-exporting。Re-exporting 的字面意思就是 重新导出。它的意思是这样的，把深层的 item 导出到上层目录中，使调用的时候，更方便。接口设计中会大量用到这个技术。

还是举上面那个 a::b::c::d 的例子。我们在 main.rs 中，要调用 d，得使用 use a::b::c::d; 来调用。而如果我们修改 a/mod.rs 文件为： a/mod.rs 文件内容：

```
pub mod b;
pub use b::c::d;
```
那么，我们在 main.rs 中，就可以使用 use a::d; 来调用了。
baidu/rust-sgx-sdk 中的 SgxMutex 就使用了 re-exporting.

1.2.6 加载外部库

外部库是通过

```
extern crate xxx;
```
这样来引入的。

至于为何 Rust 要这样设计，有以下几个原因：

1.Rust 本身模块的设计是与操作系统文件系统目录解耦的，因为 Rust 本身可用于操作系统的开发；

2.Rust 中的一个文件内，可包含多个模块，直接将 a::b::c::d 映射到 a/b/c/d.rs 会引起一些歧义；

3.Rust 一切从安全性、显式化立场出发，要求引用路径中的每一个节点，都是一个有效的模块，比如上例，d 是一个有效的模块的话，那么，要求 c, b, a 分别都是有效的模块，可单独引用。

1.2.7 prelude

Rust 的标准库，有一个 prelude 子模块，这里面包含了默认导入（std 库是默认导入的，然后 std 库中的 prelude 下面的东西也是默认导入的）的所有符号。

大体上有下面一些内容：

```
std::marker::{Copy, Send, Sized, Sync}
std::ops::{Drop, Fn, FnMut, FnOnce}
std::mem::drop
std::boxed::Box
std::borrow::ToOwned
```

```
std::clone::Clone
std::cmp::{PartialEq, PartialOrd, Eq, Ord}
std::convert::{AsRef, AsMut, Into, From}
std::default::Default
std::iter::{Iterator, Extend, IntoIterator, DoubleEndedIterator, ExactSizeIterator}
std::option::Option::{self, Some, None}
std::result::Result::{self, Ok, Err}
std::slice::SliceConcatExt
std::string::{String, ToString}
std::vec::Vec
```

在 baidu/rust-sgx-sdk 这些都需要重新引入。

1.2.8 pub restricted

在 rust 中后来引入了支持使 item 仅仅在其能够指定想要的作用域(可见范围)可见。这块的内容可以查看 https://rustcc.gitbooks.io/rustprimer/content/module/pub-restricted.html 相关内容。

理性看待 rust 语言的升级。只是升级频度高一些，这样的升级在 java 和 go 中也普遍存在。go 中的感知稍微小一些。

每次升级都要更新相应的工具链。保证最新的编译器和链接器可以将新生成的程序生成出来。

Rust 的包管理系统非常明显地体现了它的与众不同。

1.3 版本管理工具

作为一门更新快速的语言，rust 开发了专用的版本管理工具 rustup。

对于 go 而言，不需要对这些东西进行了解，只需要下载包安装到环境变量中即可。

而 rust 的开发中经常会遇到配置不同的 toolchain 等需求，因此官方开发了 rustup。rustup 功能如下：

1. 管理安装多个官方版本的 Rust 二进制程序。

2. 配置基于目录的 Rust 工具链。

3. 安装和更新来自 Rust 的发布通道: nightly, beta 和 stable。

4. 接收来自发布通道更新的通知。

5. 从官方安装历史版本的 nightly 工具链。

6. 通过指定 stable 版本来安装。

7. 安装额外的 std 用于交叉编译。

8.安装自定义的工具链。

rustup 常用命令：

1.rustup default <toolchain> 配置默认工具链。

2. rustup show 显示当前安装的工具链信息。

3. rustup update 检查安装更新。

4. rustup toolchain [SUBCOMMAND] 配置工具链

更多细节请查看 rustprimer。

1.4 rust 编译运行

```
ps: cargo build 普通编译
ps: cargo build --release # 这个属于优化编译
ps: ./target/debug/hellorust.exe
ps: ./target/release/hellorust.exe # 如果前面是优化编译，则这样运行
ps: cargo run # 编译和运行合在一起
ps: cargo run --release # 同上，区别是是优化编译的
```

第二章 Rust 基本语法

2.1 前置知识

2.1.1 表达式和语句

其它语言中：

表达式是用来表达某含义的。可以包括定义某值，或判断某物，最终会有一个“值”的体现，“Anything that has a value″。
比如说 var a=b 就是表达式，是把 b 的值赋给 a，或者 if (a == b)其中 if()内的也是表达式。
而表达式语句就是程序识别的一条执行表达式的语句。
例如 var a=b; 这条是赋值语句，这里微小的差别就是加上了分号;作为语句结束符。
其实表达式简单的可以理解成某语言的语法，而由这些语法构成的一条执行语句则是表达式语句。最直观的根据是否有分号来判断。

但是在 rust 语言中有点不同，

“let a = 5”是表达式语句，即使没有分号。

let y = (let a = 4);会返回下面的错误：

variable declaration using `let` is a statement。

2.1.2 rust doc

rust 提供了从注释到可浏览网页文档的生成方法，通过 cargo doc 的方式。

注释主要有三种：行注释，模块注释，文档注释。

Rust 也有特定的用于文档的注释类型，通常被称为 文档注释（documentation comments），他们会生成 HTML 文档。这些 HTML 展示公有 API 文档注释的内容，他们意在让对库感兴趣的程序员理解如何 使用 这个 crate，而不是它是如何被 实现 的。

文档注释使用三斜杠 /// 而不是两斜杆并支持 Markdown 注解来格式化文本。文档注释就位于需要文档的项的之前。

模块注释使用 //! ，行注释使用 //

模块注释和文档注释用起来非常舒服的，远比/* */舒服，IDEA 对它支持很好，提供自动换行。

2.2 条件表达式

在 rust 中的条件表达式没有 switch 语句。

2.2.1 if 表达式

相对于 C 系语言，Rust 的 if 表达式的显著特点是：

1.判断条件不用小括号括起来。

2.它是表达式，而不是语句。

即使花括号里面有再多的表达式语句，它最后仍然需要返回一个值，是表达式。下面的写法是正确的：

```
let x = 5;

let y = if x == 5 {
          println!("hello test”)
    10
} else {
    15
}; // y: i32
```

需要说明的是 if 中条件判断必须是 bool 类型，不能写出 if 5 这种判断条件。

2.2.2 match 语句

Rust 中没有类似于 C 的 switch 关键字，但它有用于模式匹配的 match，能实现同样的功能，并且强大太多。

match 的使用非常简单，举例如下：

```
let x = 5;

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

2.2.3 if let 表达式

```
let y = 5;

if let y = 5 {

     println!("{}", y); // 这里输出为：5

}
```

if let 实际上是一个 match 的简化用法。设计这个特性的目的是，在条件判断的时候，直接做一次模式匹配，方便代码书写，使代码更紧凑。

2.3 循环表达式

2.3.1 for 循环

Rust 的 for 循环实际上和 C 语言的循环语句是不同的。这是为什么呢？因为，for 循环不过是 Rust 编译器提供的语法糖！在 rust 中，for 语句用于遍历一个迭代器。

Rust 迭代器返回一系列的元素，每个元素是循环中的一次重复。然后它的值与 var 绑定，它在循环体中有效。每当循环体执行完后，我们从迭代器中取出下一个值，然后我们再重复一遍。当迭代器中不再有值时，for 循环结束。

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

没有 c 中的这种用法：

```
// C 语言的 for 循环例子
for (x = 0; x < 10; x++) {
    printf( "%d\n", x );
}
```

设计目的：

1. 简化边界条件的确定，减少出错；

2. 减少运行时边界检查，提高性能。

当你需要记录你已经循环了多少次了的时候，你可以使用 .enumerate() 函数。比如：

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

2.3.2 while 循环

Rust 提供了 while 语句，条件表达式为真时，执行语句体。当你不确定应该循环多少次时可选择 while。

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

while !done {
    x += x - 3;

    println!("{}", x);

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

while 条件判断的类型必须是 bool 类型。

2.3.3 loop

loop 专用于无限循环，好处是对编译和运行进行了优化，跳过了表达式检查。

2.3.4 break 和 continue

Rust 也提供了 break 和 continue 两个关键字用来控制循环的流程。

1.break 用来跳出当前层的循环；

2.continue 用来执行当前层的下一次迭代

break 和 continue 可以大大简化代码，不论在哪一门语言中。

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

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

和

```
let mut x = 5;

loop {
    x += x - 3;

    println!("{}", x);

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

2.3.5 label

你也许会遇到这样的情形，当你有嵌套的循环而希望指定你的哪一个 break 或 continue 该起作用。就像大多数语言，默认 break 或 continue 将会作用于当前层的循环。当你想要一个 break 或 continue 作用于一个外层循环，你可以使用标签来指定你的 break 或 continue 语句作用的循环。

如下代码只会在 x 和 y 都为奇数时打印他们：

```
'outer: for x in 0..10 {
    'inner: for y in 0..10 {
        if x % 2 == 0 { continue 'outer; } // continues the loop over x
        if y % 2 == 0 { continue 'inner; } // continues the loop over y
        println!("x: {}, y: {}", x, y);
    }
}
```

2.4 Rust 类型系统

2.4.1 可变性

rust 在声明变量时，在变量前面加入 mut 关键字，变量就会成为可变绑定的变量。

通过可变绑定可以直接通过标识符修改内存中的变量的值。

在绑定后仍然可以重新修改绑定类型。

例子：

```
fn main() {
    let mut a: f64 = 1.0;
    let b = 2.0f32;

    //改变 a 的绑定
    a = 2.0;
    println!("{:?}", a);

    //重新绑定为不可变
    let a = a;

    //不能赋值
    //a = 3.0;

    //类型不匹配
    //assert_eq!(a, b);
}
```

2.4.2 原生类型

在所有 rust 的类型中，比较复杂的是字符串类型。当然不仅仅在 rust 中，包括 golang 等其它语言中，字符串类型和字符类型都是值得推敲的地方。

在本部分内容中，单独拿出字符类型和字符串类型放到最后进行讨论。

字符类型和字符串类型最好和其它语言对比讨论，例如 go。

2.4.2.1 bool 类型

Rust 自带了 `bool` 类型，其可能值为 `true` 或者 `false`

```
let is_she_love_me = false;
let mut is_he_love_me: bool = true;
```

当然，bool 类型被用的最多的地方就是在 if 表达式里了

2.4.2.2 数字类型

和其他类 C 系的语言不一样，Rust 用一种*符号+位数*的方式来表示其基本的数字类型。

可用的符号有　i、f、u
可用的位数，都是 2 的 n 次幂，分别为 8、16、32、64 及 size。
因为浮点类型最少只能用 32 位来表示，因此只能有 `f32` 和 `f64` 来表示。

2.4.2.3 自适应类型

isize 和 usize 取决于你的操作系统的位数。简单粗暴一点比如 64 位电脑上就是 64 位，32 位电脑上就是 32 位。

但是需要注意的是，不能因为电脑是 64 位的，而强行将它等同于 64，也就是说 isize != i64，任何情况下你都需要强制转换。

减少使用 isize 和 usize，因为它会降低代码可移植性。

2.4.2.4 数组 array

Rust 的数组是被表示为[T;N]。其中 N 表示数组大小，并且这个大小一定是个编译时就能获得的整数值，T 表示泛型类型，即任意类型。我们可以这么来声明和使用一个数组:

```
let a = [8, 9, 10];
let b: [u8;3] = [8, 6, 5];
print!("{}", a[0]);
```

和 Golang 一样，Rust 的数组中的 N（大小）也是类型的一部分，即[u8; 3] != [u8; 4]。
Rust 大小是固定的。

2.4.2.5 slice

Slice 从直观上讲，是对一个 Array 的切片，通过 Slice，你能获取到一个 Array 的部分或者全部的访问权限。和 Array 不同，Slice 是可以动态的，但是呢，其范围是不能超过 Array 的大小，这点和 Golang 是不一样的。Golang slice 可以超出 Array 的大小是存在一些问题的。

一个 Slice 的表达式可以为如下:　&[T]　或者　&mut [T]。

这里&符号是一个难点，我们不妨放开这个符号，简单的把它看成是 Slice 的规定。另外，同样的，Slice 也是可以通过下标的方式访问其元素，下标也是从 0 开始。　可以这么声明并使用一个 Slice：

```
let arr = [1, 2, 3, 4, 5, 6];
let slice_complete = &arr[..]; // 获取全部元素
let slice_middle = &arr[1..4]; // 获取中间元素，最后取得的 Slice 为 [2, 3, 4] 。切片遵循左闭右开原则。
let slice_right = &arr[1..]; // 最后获得的元素为[2, 3, 4, 5, 6]，长度为 5。
let slice_left = &arr[..3]; // 最后获得的元素为[1, 2, 3]，长度为 3。
```

似乎没有看到 slice 像 go 的动态 append 的例子。

2.4.2.6 动态 Vec

在 Rust 里，`Vec` 被表示为 `Vec<T>`， 其中 T 是一个泛型。

```
let mut v1: Vec<i32> = vec![1, 2, 3]; // 通过 vec!宏来声明
let v2 = vec![0; 10]; // 声明一个初始长度为 10 的值全为 0 的动态数组
println!("{}", v1[0]); // 通过下标来访问数组元素

for i in &v1 {
    print!("{}", i); // &Vec<i32> 可以通过 Deref 转换成 &[i32]
}

println!("");

for i in &mut v1 {
    *i = *i+1;
    print!("{}", i); // 可变访问
}
```

2.4.2.7 函数类型

相似于 golang，在 rust 中函数也是一种类型，例子如下：

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

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

assert_eq!(11, x(10));
```

2.4.2.8 枚举类型

```
struct Student{
    name: String,
}

enum Message{
    School(String),
    Location{x:i32,y:i32},
    ChangeColor(i32, i32, i32),
    Name(Student),
    ExitColor,
}

fn main(){
    let m = Message::ExitColor;
    match m {
        Message::ExitColor=>println!("{}","exited color"),
        Message::ChangeColor(x,y,z) => println!("{}",x),
        Message::Name(s) => println!("{}",s.name),
        Message::School(s) => println!("{}",s),
        Message::Location {x,y} => println!("{}",x),
    }
}
```

与结构体一样，枚举中的元素默认不能使用关系运算符进行比较（如 ==, !=, >=)， 也不支持像+和*这样的双目运算符，需要自己实现，或者使用 match 进行匹配。

枚举默认也是私有的，如果使用 pub 使其变为公有，则它的元素也都是默认公有的。 这一点是与结构体不同的：即使结构体是公有的，它的域仍然是默认私有的。

rust 枚举与其他语言的枚举不同的是在指定枚举元素时定义它元素是由什么组成的。

2.4.2.9 字符串类型

如果要说 rust 的字符串类型，就不得不先提 go 的字符串类型。

2.4.2.9.1 Golang 中的字符串类型

Go 没有专门的字符类型，**存储字符直接使用 byte 来存储**，字符串就是一串固定长度的字符连接起来字符序列。与其他编程语言不同之处在于，**Go 的字符串是字节组成，**而其他的编程语言是字符组成。

Go 的字符串可以把它当成字节数组来用，并且是不可变的。

Rust 的字符串底层也是 Vec<u8>动态数组，这点和 Go 的字符串有点类似，不同的是 Go 的字符串是定长的，无法修改的。

Rust 和 Go 原声在字符串里面支持 unicode，这就导致了很大某方面的不同。

Go 中字符串的例子：

```
package main

import (
          "fmt"
          "reflect"
)

func main() {
          fmt.Println("test")

          {
                    fmt.Println("example 1")
                    a := "testb"
                    fmt.Println(a[0])  //output 116
                    fmt.Println(len(a))

                    b := "testa 你好"
                    fmt.Println(len(b))

                    c := b[0:8]  // c is string type,alouth is slice of string
                    fmt.Println("c type is ",reflect.TypeOf(c))
                    fmt.Println(c)

                    if c == "testa 你" {
                              fmt.Println("yes")
                    } else {
                              fmt.Println("no")
                    }
          }

          {
                    fmt.Println("example 2")
                    var str []byte
```

```
                str = []byte("waht")
                fmt.Println(string(str))

                for i := 0; i < 10; i++ {
                        str = append(str, byte(i+100))
                }
                fmt.Println(string(str))
                fmt.Println(str)
        }

        {
                fmt.Println("example 3")
                str1 := "नमस्ते";
                fmt.Println(len(str1))
        }

        {
                fmt.Println("example 4")
                str2 := "hello world";
                str2slice := str2[0:5]
                str2slice = str2slice+" hello"

                fmt.Println(str2slice)
                fmt.Println(str2)
        }
        {
                a := []int{1, 2, 3, 4}
                fmt.Println(len(a))

                b := a[0:1]
                b = append(b, 5)
                fmt.Println(len(b))

                for _, value := range a {
                        fmt.Println(value)
                }

                str1 := "hello world"
                fmt.Println(str1)

                str1slice := str1[0:8]
                fmt.Println(str1slice)

                //compile error
                //# command-line-arguments
                //./main.go:78:21: cannot assign to ([]byte)(str1slice)

                //[]byte(str1slice) = append([]byte(str1slice),byte('a'))
                //fmt.Println(str1slice)
        }
}
```

输出结果：

```
example 1
116
5
11
c type is  string
testa 你
yes
example 2
waht
wahtdefghijklm
[119 97 104 116 100 101 102 103 104 105 106 107 108 109]
example 3
18
example 4
hello hello
```

```
hello world
```

从上面的例子中，可以看出存储 unicode 字符最大的特点是不同的字符串所占的字节数不同。例如梵文占 6 个字节，汉字占 3 个字节。

另外 golang 中的 string slice 不能进行 append 操作，可以把它当成一个固定的 string 类型来使用，例如可以比较二者包含的字符串是否相等。

另外 String 类型底层是 byte 数组，但是底层的 byte 数组没有暴露出来，所以无法修改长度。

2.4.2.9.2 Rust 中的字符串类型

常用 rust 字符串类型为&str 和 String，前者是字符串的引用，后者是基于堆创建的，可增长的字符串。

2.4.2.9.2.1 &str

let s ="hello world";那 s 的类型就是&str，右边称为字符串字面量 literal，程序编译成二进制文件后，这个字符串会被保存在文件内部，所以 s 是特定位置字符串的引用，这就是为什么 s 是&str 类型。

str 生命周期是 static，但是引用是有生命周期限制的。

可以在字符串字面量前加上 r 来避免转义

```
//没有转义序列
let d: &'static str = r"abc \n abc";
//等价于
let c: &'static str = "abc \\n abc";
```

2.4.2.9.2.2 String

这时候，一种在堆上声明的字符串 String 被设计了出来。 它能动态的去增长或者缩减，那么怎么声明它呢？

```
let x:&'static str = "hello";

let mut y:String = x.to_string();
println!("{}", y);
y.push_str(", world");
println!("{}", y);
```

那么如何将一个 String 重新变成&str 呢？用 &* 符号

```
fn use_str(s: &str) {
    println!("I am: {}", s);
}

fn main() {
    let s = "Hello".to_string();
    use_str(&*s);
}
```

&*是两个符号&和*的组合，按照 Rust 的运算顺序，先对 String 进行 Deref，也就是*操作。

由于 String 实现了 impl Deref<Target=str> for String，这相当于一个运算符重

载，所以你就能通过*获得一个 str 类型。但是我们知道，单独的 str 是不能在 Rust 里直接存在的，因此，我们需要先给他进行&操作取得&str 这个结果。

涵盖大部分&str 和 String 的例子。

```
fn main() {
    println!("Hello, world!");
    {
        println!("example 1");

        let mut str1 = String::from("你好 hello");
        str1.push_str(" str1");

        let mut str2 = &mut str1;
        str2.push_str(" test ");
        str2.push('a');
        println!("{:?}", str1);

        let mut str3 = &mut str1;
        str3.push_str(" test3 ");
        str3.push('3');
        println!("{:?}", str1);

        // compile error
        // error[E0502]: cannot borrow `str1` as immutable because it is also borrowed
as mutable
        //  --> src/main.rs:14:25
        //   |
        //11 |         let mut str3  = &mut str1;
        //   |                         --------- mutable borrow occurs here
        //...
        //14 |         println!("{:?}",str1);
        //   |                         ^^^^ immutable borrow occurs here
        //...
        //17 |         println!("{:?}",str3);
        //   |                         ---- mutable borrow later used here
        // println!("{:?}",str3);
    }

    {
        println!("example 2");

        let mut v1 = vec![1, 2, 3, 4];
        let mut v2 = &mut v1;
        v2.push(100);
        println!("{:?}", v2);

        let mut v3 = &mut v1;
        v3.push(20);

        //compile error
        println!("{:?}", v1);

        //error[E0502]: cannot borrow `v1` as immutable because it is also borrowed as
mutable
        //  --> src/main.rs:41:25
        //   |
        //37 |         let mut v3 = &mut v1;
        //   |                      ------- mutable borrow occurs here
        //...
        //41 |         println!("{:?}",v1);
        //   |                         ^^ immutable borrow occurs here
        //42 |         println!("{:?}",v3);
        //   |                         -- mutable borrow later used here

        //  println!("{:?}",v3);
    }
```

```
    {
        println!("example 3");

        let mut str1 = String::from("hello");

        //error[E0277]: the type `std::string::String` cannot be indexed by
`{integer}`
        //  --> src/main.rs:59:23
        //   |
        //59 |         let answer = &str1[0];
        //   |                       ^^^^^^^ `std::string::String` cannot be indexed
by `{integer}`
        //   |
        //   = help: the trait `std::ops::Index<{integer}>` is not implemented for
`std::string::String`

        // compile error

        //let answer = &str1[0];
    }

    {
        println!("example 4");

        //how to loop the string
        for c in "□□□□□□".chars() {
            println!("{}", c);
        }
    }

    {
        println!("example 5");

        let mut a = String::from("testa 你好");
        let mut b = &a[0..4];
        println!("{}", b);

        //This will panic
        //thread 'main' panicked at 'byte index 6 is not a char boundary; it is inside
'你' (bytes 5..8) of `testa 你好`', src/libcore/str/mod.rs:2027:5

        //let mut c = &a[0..6];
        //println!("{}",c);
    }

    {
        println!("example 6");

        let mut a = String::from("testa 你好");
        println!("the length of a is {}", a.len());
    }

    {
        println!("example 7");

        let mut a = String::from("test");
        let mut b = &mut a;
        b.push_str(" str");
        println!("{}", b);
        println!("{}", a);

        //compile error

        //error[E0502]: cannot borrow `a` as immutable because it is also borrowed as
mutable
        //   --> src/main.rs:102:23
        //    |
        //99  |         let mut b = &mut a;
        //    |                     ------ mutable borrow occurs here
```

```
        //...
        //102 |         println!("{}",a);
        //    |                         ^ immutable borrow occurs here
        //...
        //106 |         println!("{}",b);
        //    |                         - mutable borrow later used here
        //
        //error[E0502]: cannot borrow `a` as immutable because it is also borrowed as
mutable
        //   --> src/main.rs:105:23
        //    |
        //99  |         let mut b = &mut a;
        //    |                     ------ mutable borrow occurs here
        //...
        //105 |         println!("{}",a);
        //    |                         ^ immutable borrow occurs here
        //106 |         println!("{}",b);
        //    |                         - mutable borrow later used here

        //        println!("{}",a);
        //        println!("{}",b);
    }

    {
        println!("example 8");

        let s1 = String::from("Hello, ");
        let s2 = String::from("world!");
        let s3 = s1 + &s2; // 注意 s1 被移动了，不能继续使用

        // write bug
        // error[E0369]: binary operation `+` cannot be applied to type
`&std::string::String`

        //let s3 = &s1 + &s2; // 注意 s1 被移动了，不能继续使用


        println!("{}", s3);

        // error[E0382]: borrow of moved value: `s1`
        //   --> src/main.rs:141:23
        //    |
        //135 |         let s1 = String::from("Hello, ");
        //    |             -- move occurs because `s1` has type `std::string::String`,
which does not implement the `Copy` trait
        //136 |         let s2 = String::from("world!");
        //137 |         let s3 = s1 + &s2; // 注意 s1 被移动了，不能继续使用
        //    |                  -- value moved here
        //...
        //141 |         println!("{}",s1);
        //    |                         ^^ value borrowed here after move

        // compile error
        println!("{}", s2);
    }
}
```

输出结果：

```
example 1
"你好 hello str1 test a"
"你好 hello str1 test a test3 3"
example 2
[1, 2, 3, 4, 100]
[1, 2, 3, 4, 100, 20]
example 3
example 4
example 5
test
example 6
```

```
the length of a is 11
example 7
test str
test str
example 8
Hello, world!
world!
```

需要注意的主要就是：String 类型底层实现是 vec<u8>,unicode 类型，并且拿着引用可以改变 String 内容。有点类似中在 go 做一个特殊的 String 类型，并且内部包着一个 byte 数组。

第三章 所有权 引用借用 生命周期

一个 C 语言的例子：

```
int* foo() {
    int a;          // 变量 a 的作用域开始
    a = 100;
    char *c = "xyz";   // 变量 c 的作用域开始
    return &a;
}                   // 变量 a 和 c 的作用域结束
```

尽管可以编译通过，但这是一段非常糟糕的代码，变量 a 和 c 都是局部变量，函数结束后将局部变量 a 的地址返回，但局部变量 a 存在栈中，在离开作用域后，局部变量所申请的栈上内存都会被系统回收，从而造成了 Dangling Pointer 的问题。这是一个非常典型的内存安全问题。很多编程语言都存在类似这样的内存安全问题。

再来看变量 c，c 的值是常量字符串，存储于常量区，可能这个函数我们只调用了一次，我们可能不再想使用这个字符串，但 xyz 只有当整个程序结束后系统才能回收这片内存，这点让程序员是不是也很无奈？

所以，内存安全和内存管理通常是程序员眼中的两大头疼问题。令人兴奋的是，Rust 却不再让你担心内存安全问题，也不用再操心内存管理的麻烦，那 Rust 是如何做到这一点的？通过所有权。

3.1 所有权

3.1.1 绑定

首先必须强调下，准确地说 Rust 中并没有变量这一概念，而应该称为标识符，目标资源(内存，存放 value)绑定到这个标识符。

```
{
    let x: i32;       // 标识符 x，没有绑定任何资源
    let y: i32 = 100; // 标识符 y，绑定资源 100
}
```

Rust 并不会像其他语言一样可以为变量默认初始化值，Rust 明确规定变量的初始值必须由程序员自己决定。

上述代码中，let 关键字并不只是声明变量的意思，它还有一层特殊且重要的概念-绑定。通俗的讲，let 关键字可以把一个标识符和一段内存区域做“绑定”，绑定后，这段内存就被这个标识符所拥有，这个标识符也成为这段内存的唯一所有者。

3.1.2 作用域

rust 有着和其它语言类型的定义作用域的规则。

像 C 语言一样，在局部变量离开作用域后，变量随即会被销毁；但不同是，Rust 会连同变量绑定的内存，不管是否为常量字符串，连同所有者变量一起被销毁释放。

3.1.3 移动语义

在 Rust 中，和“绑定”概念相辅相成的另一个机制就是“转移 move 所有权”，意思是，可以把资源的所有权(ownership)从一个绑定转移（move）成另一个绑定，这个操作同样通过 let 关键字完成，和绑定不同的是，=两边的左值和右值均为两个标识符：

```
语法：
    let 标识符 A = 标识符 B;  // 把“B”绑定资源的所有权转移给“A”
```

move 前后的内存示意如下：

```
Before move:
a <=> 内存(地址：A，内容："xyz")
After move:
a
b <=> 内存(地址：A，内容："xyz")
```

move 后，如果变量 A 和变量 B 离开作用域，所对应的内存会不会造成“Double Free”的问题？答案是否定的，Rust 规定，只有资源的所有者销毁后才释放内存，而无论这个资源是否被多次 move，同一时刻只有一个 owner，所以该资源的内存也只会被 free 一次。

3.1.4 Copy 特性

举例如下：

```
  let a: i32 = 100;
   let b = a;
   println!("{}", a);
```

编译确实可以通过，输出为 100。这是为什么呢，是不是跟 move 小节里的结论相悖了？ 其实不然，这其实是根据变量类型是否实现 Copy 特性决定的。对于实现 Copy 特性的变量，在 move 时会拷贝资源到新内存区域，并把新内存区域的资源 binding 为 b。特性有点类似于 interface，但是在 rust 中必须显式实现。

```
Before move:
a <=> 内存(地址：A，内容：100)
After move:
a <=> 内存(地址：A，内容：100)
b <=> 内存(地址：B，内容：100)
```

在 Rust 中，基本数据类型(Primitive Types)均实现了 Copy 特性，包括 i8, i16, i32, i64, usize, u8, u16, u32, u64, f32, f64, (), bool, char 等等。

3.1.5 浅拷贝与深拷贝

对于基本数据类型来说，“深拷贝”和“浅拷贝“产生的效果相同。对于引用对象类型来说，”浅拷贝“更像仅仅拷贝了对象的内存地址。 如果我们想实现对 String 的”深拷贝“怎么办？ 可以直接调用 String 的 Clone 特性实现对内存的值拷贝而不是简单的地址拷贝。

3.1.6 高级 copy

一旦一种类型实现了 Copy 特性，这就意味着这种类型可以通过的简单的位(bits)拷贝实现拷贝。从前面知识我们知道“绑定”存在 move 语义（所有权转移），但是，一旦这种类型实现了 Copy 特性，会先拷贝内容到新内存区域，然后把新内存区域和这个标识符做绑定。

哪些情况下我们自定义的类型（如某个 Struct 等）可以实现 Copy 特性？ 只要这种类型的属性类型都实现了 Copy 特性，那么这个类型就可以实现 Copy 特性。 例如：

```
struct Foo {  //可实现 Copy 特性
    a: i32,
    b: bool,
}

struct Bar {  //不可实现 Copy 特性
    l: Vec<i32>,
}
```

因为 Foo 的属性 a 和 b 的类型 i32 和 bool 均实现了 Copy 特性，所以 Foo 也是可以实现 Copy 特性的。但对于 Bar 来说，它的属性 l 是 Vec<T>类型，这种类型并没有实现 Copy 特性，所以 Bar 也是无法实现 Copy 特性的。

那么我们如何来实现 Copy 特性呢？有两种方式可以实现。

3.1.6.1.通过 `derive` 让 Rust 编译器自动实现

```
#[derive(Copy, Clone)]
 struct Foo {
     a: i32,
     b: bool,
 }
```

3.1.6.2 手动实现 不推荐 会进入 unsafe rust

3.1.7 Copy trait 与 Clone trait

Copy 内部没有方法，Clone 内部有两个方法。

1.Copy trait 是给编译器用的，告诉编译器这个类型默认采用 copy 语义，而不是 move 语义。Clone trait 是给程序员用的，我们必须手动调用 clone 方法，它才能发挥作用。

2.Copy trait 不是你想实现就实现，它对类型是有要求的，有些类型就不可能 impl Copy。Clone trait 没有什么前提条件，任何类型都可以实现（unsized 类型除外）。

3.Copy trait 规定了这个类型在执行变量绑定、函数参数传递、函数返回等场景下的操作方式。即这个类型在这种场景下，必然执行的是“简单内存拷贝”操作，这是由编译器保证的，程序员无法控制。Clone trait 里面的 clone 方法究竟会执行什么操作，则是取决于程序员自己写的逻辑。一般情况下，clone 方法应该执行一个“深拷贝”操作，但这不是强制的，如果你愿意，也可以在里面启动一个人工智能程序，都是有可能的。

5.如果你确实需要 Clone trait 执行“深拷贝”操作，编译器帮我们提供了一个工具，我们可以在一个类型上添加#[derive(Clone)]，来让编译器帮我们自动生成那些重复的代码。

正因为如此，在希望让一个类型具有 Copy 性质的时候，一般使用 #[derive(Copy, Clone)] 这种方式，这种情况下它们俩最好一起出现，避免手工实现 Clone 导致错误。

3.1.8 高级 move

move 关键字常用在闭包中，强制闭包获取所有权。

```
fn main() {
    let mut x: String = String::from("abc");
    let mut some_closure = move |c: char| x.push(c);
    let y = some_closure('d');
    println!("x={:?}", x);
}
```

上述代码会报错。

这是因为 move 关键字，会把闭包中的外部变量的所有权 move 到包体内，发生了所有权转移的问题，所以 println 访问 x 会如上错误。如果我们去掉 println 就可以编译通过。

那么，如果我们想在包体外依然访问 x，即 x 不失去所有权，怎么办？

```
fn main() {
    let mut x: String = String::from("abc");
    {
        let mut some_closure = |c: char| x.push(c);
        some_closure('d');
    }
    println!("x={:?}", x);  //成功打印：x="abcd"
}
```

我们只是去掉了 move，去掉 move 后，包体内就会对 x 进行了可变借用，而不是“剥夺”x 的所有权，细心的同学还注意到我们在前后还加了{}大括号作用域，是为了作用域结束后让可变借用失效，这样 println 才可以成功访问并打印我们期待的内容。

最新的版本不加大括号也可以的？但是为了可读加上大括号比较好。尽可能满足作用域内部的规则。

具体内容需要查看闭包的可变借用。

3.2 引用和借用

所有权系统允许我们通过“Borrowing”的方式达到这个目的。这个机制非常像其他编程语言中的“读写锁”，即同一时刻，只能拥有一个“写锁”，或只能拥有多个“读锁”，不允许“写锁”和“读锁”在同一时刻同时出现。当然这也是数据读写过程中保障一致性的典型做法。只不过 Rust 是在编译中完成这个(Borrowing)检查的，而不是在运行时，这也就是为什么其他语言程序在运行过程中，容易出现死锁或者野指针的问题。

通过&符号完成 Borrowing：

```
fn main() {
    let x: Vec<i32> = vec!(1i32, 2, 3);
    let y = &x;
    println!("x={:?}, y={:?}", x, y);
}
```

Borrowing(&x)并不会发生所有权 moved，所以 println 可以同时访问 x 和 y。通过引用，就可以对普通类型完成修改。

```
fn main() {
    let mut x: i32 = 100;
    {
        let y: &mut i32 = &mut x;
        *y += 2;
    }
    println!("{}", x);
}
```

y 在大括号结束后会释放掉。

& 符号就是　引用，它们允许你使用值但不获取其所有权。将获取引用作为函数参数称为　借用（borrowing）。

3.2.1 借用和引用的规则

1. 同一作用域，特定数据最多只有一个可变借用（&mut T），或者 2。

2. 同一作用域，特定数据可有 0 个或多个不可变借用（&T），但不能有任何可变借用。

3. 借用在离开作用域后释放。

4. 在可变借用释放前不可访问源变量。

3.2.2 引用的可变性

Borrowing 也分“不可变借用”（默认，&T）和“可变借用”（&mut T）。

顾名思义，“不可变借用”是只读的，不可更新被引用的内容。

```
  fn main() {
    //源变量 x 可变性
    let mut x: Vec<i32> = vec!(1i32, 2, 3);

    //只能有一个可变借用
    let y = &mut x;
    // let z = &mut x; //错误
    y.push(100);

    //ok
    println!("{:?}", y);

    //错误，可变借用未释放，源变量不可访问
    // println!("{:?}", x);
}  //y 在此处销毁
```

另外一个例子：

```
fn main() {
    let mut x: Vec<i32> = vec!(1i32, 2, 3);

    //更新数组
    //push 中对数组进行了可变借用，并在 push 函数退出时销毁这个借用
    x.push(10);

    {
        //可变借用 1
        let mut y = &mut x;
        y.push(100);

        //可变借用 2，注意：此处是对 y 的借用，不可再对 x 进行借用，
        //因为 y 在此时依然存活。
        let z = &mut y;
        z.push(1000);

        println!("{:?}", z); //打印: [1, 2, 3, 10, 100, 1000]
    } //y 和 z 在此处被销毁，并释放借用。


    //访问 x 正常
    println!("{:?}", x); //打印: [1, 2, 3, 10, 100, 1000]
}
```

3.2.3 总结

1. 借用不改变内存的所有者（Owner），借用只是对源内存的临时引用。

2. 在借用周期内，借用方可以读写这块内存，所有者被禁止读写内存；且所有者保证在有“借用”存在的情况下，不会释放或转移内存。

3. 失去所有权的变量不可以被借用（访问）。

4. 在租借期内，内存所有者保证不会释放/转移/可变租借这块内存，但如果是在非可变租借的情况下，所有者是允许继续非可变租借出去的。

5. 借用周期满后，所有者收回读写权限。

6. 借用周期小于被借用者（所有者）的生命周期。

3.3 生命周期

几个概念：

1. Owner：资源的所有者 a

2. Borrower：资源的借用者 x

3. Scope：作用域，资源被借用/引用的有效期

无论是资源的所有者还是资源的借用/引用，都存在在一个有效的存活时间或区间，这个时间区间称为生命周期， 也可以直接以 Scope 作用域去理解。

例子：

```
fn main() {
    let a = 100_i32;

    {
        let x = &a;
    }  // x 作用域结束
    println!("{}", x);
}
```

生命周期/作用域图示：

```
            {    a    {    x    }    *    }
所有者 a:          |_____________________|
借用者 x:                    |____|            x = &a
  访问 x:                              |       失败：访问 x
```

分析：借用者 x 的生命周期是资源所有者 a 的生命周期的子集。但是 x 的生命周期在第一个 }时结束并销毁，在接下来的 println! 中再次访问便会发生严重的错误。

3.3.1 隐式 lifetime

我们经常会遇到参数或者返回值为引用类型的函数：

```
fn foo(x: &str) -> &str {
    x
}
```

foo 函数仅仅接受一个 &str 类型的参数（x 为对某个 string 类型资源 Something 的借用），并返回对资源 Something 的一个新的借用。

实际上，上面函数包含隐性的生命周期命名，这是由编译器自动推导的，相当于：

```
fn foo<'a>(x: &'a str) -> &'a str {
    x
}
```

3.4 高级所有权

前面三小节未大量涉及到关于所有权中的比较高级用法。

3.4.1.函数传递参数和返回参数类似于 let 语句

在 rust 中，函数是存放在函数栈，为了更为快速的运行。

函数输入参数的传递和返回参数的赋值类似于 let 语句，都看传递的参数和返回的参数是否实现了 copy trait。

如果实现了 copy trait，那么就不会夺走它的所有权，标识符在函数外部还可以继续访问。

如果没有实现 copy trait，那么它的所有权都会被夺走。

需要说明的是，从结果上来看，引用是值传递，和实现了 copy trait 的表征相同，同样可以外部继续使用。

举例而言：

```
pub fn test(){
    let a = vec![1,2,3,4,5];
    print_vec(a);

//error[E0382]: borrow of moved value: `a`
// --> src/fn_params.rs:6:21
//  |
//2 |     let a = vec![1,2,3,4,5];
//  |         - move occurs because `a` has type `std::vec::Vec<i32>`, which does not
implement the `Copy` trait
//3 |
//4 |     print_vec(a);
//  |               - value moved here
//5 |
//6 |     println!("{:?}",a);
//  |                     ^ value borrowed here after move
//    println!("{:?}",a);

    {
        let a = vec![1,2,3,4,5];
        let b = &a;
        print_vecs(b);
        //no error
        println!("{:?}",b);
    }
}
fn print_vec(a: Vec<i32>){
    println!("{:?}",a);
}
fn print_vecs(a:&Vec<i32>){
    println!("{:?}",a)
}
```

从上面可以看出，以 vec!为函数参数则发生了 move，在后面无法使用。

以&vec！为函数参数，后续仍然可以使用&vec！。

错误提示中也是说明根据是否实现 copy trait 来决定是否进行所有权的转移。

3.4.2 涉及到函数和结构体的借用检查器

在 rust 中引入引用后，我们需要使用引入借用检查器来保证引用的生命周期不会超过所有权的生命周期。

Rust 编译器有一个 借用检查器（borrow checker），它比较作用域来确保所有的借用都是有效的。避免出现类似 C 的悬挂指针等问题。借用检查器是在编译阶段进行工作的，将所有的无效借用识别出来。

大部分情况下，借用检查器可以正常工作，对于某些特殊情况，借用检查器无法识别，需要由开发人员显示标注生命周期。

主要分为以下三种情况：

1.函数定义中的生命周期注解

2.结构体定义中的生命周期注解

3.方法定义中的生命周期注解

上述三种类型的生命周期注解主要用来为借用检查器提供帮助，查看使用到它们的地方是否满足借用的生命周期规则，有没有悬挂指针的问题。

3.4.2.1 函数定义中的生命周期注解

举例而言：

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

rust 程序设计语言 简体中文版已经描述的非常到位了。

当在函数中使用生命周期注解时，这些注解出现在函数签名中，而不存在于函数体中的任何代码中。这是因为 Rust 能够分析函数中代码而不需要任何协助，不过当函数引用或被函数之外的代码引用时，让 Rust 自身分析出参数或返回值的生命周期几乎是不可能的。这些生命周期在每次函数被调用时都可能不同。这也就是为什么我们需要手动标记生命周期。

当具体的引用被传递给 longest 时，被 'a 所替代的具体生命周期是 x 的作用域与 y 的作用域相重叠的那一部分。换一种说法就是泛型生命周期 'a 的具体生命周期等同于 x 和 y 的生命周期中较小的那一个。因为我们用相同的生命周期参数 'a 标注了返回的引用值，所以返回的引用值就能保证在 x 和 y 中较短的那个生命周期结束之前保持有效。

要推导 Lifetime 是否合法，先明确两点：

1.输出值（也称为返回值）依赖哪些输入值

2.输入值的 Lifetime 大于或等于（可能依赖的输出值）的 Lifetime（准确来说：子集，而不是大于或等于）

Lifetime 推导公式： 当输出值 R 依赖输入值 X Y Z ...，当且仅当输出值的 Lifetime 为所有输入值的 Lifetime 交集的子集时，生命周期合法。

例子 1：

```
fn foo<'a>(x: &'a str, y: &'a str) -> &'a str {
    if true {
        x
    } else {
        y
    }
}
```

因为返回值同时依赖输入参数 x 和 y，所以

```
   Lifetime(返回值) ⊆ ( Lifetime(x) ∩ Lifetime(y) )

    即：

    'a ⊆ ('a ∩ 'a)  // 成立
```

例子 2：

```
fn foo<'a, 'b>(x: &'a str, y: &'b str) -> &'a str {
    if true {
        x
    } else {
        y
    }
}
```

因为返回值同时依赖 x 和 y，所以

```
   Lifetime(返回值) ⊆ ( Lifetime(x) ∩ Lifetime(y) )

    即：

    'a ⊆ ('a ∩ 'b)  //不成立
```

很显然，上面我们根本没法保证成立。

所以，这种情况下，我们可以显式地告诉编译器'b 比'a 长（'a 是'b 的子集），只需要在定义 Lifetime 的时候，在'b 的后面加上：'a，意思是'b 比'a 长，'a 是'b 的子集：

```
fn foo<'a, 'b: 'a>(x: &'a str, y: &'b str) -> &'a str {
    if true {
        x
    } else {
        y
    }
}
```

这里我们根据公式继续推导：

```
    条件：Lifetime(x) ⊆ Lifetime(y)
    推导：Lifetime(返回值) ⊆ ( Lifetime(x) ∩ Lifetime(y) )

    即：

    条件： 'a ⊆ 'b
    推导：'a ⊆ ('a ∩ 'b) // 成立
```

上面是成立的，所以可以编译通过。

3.4.2.2 结构体中的生命周期

在 struct 中 Lifetime 同样重要。关于 rust 中 struct 使用请查看第四章。

我们来定义一个 Person 结构体。

```
struct Person {
    age: &u8,
}
```

编译时我们会得到一个 error：

```
<anon>:2:8: 2:12 error: missing lifetime specifier [E0106]
<anon>:2     age: &str,
```

之所以会报错，这是因为 Rust 要确保 Person 的 Lifetime 不会比它的 age 借用长，不然会出现 Dangling Pointer 的严重内存问题。而在结构体中我们又没有进行指定。所以我们需要为 age 借用声明 Lifetime：

```
struct Person<'a> {
    age: &'a u8,
}
```

不需要对 Person 后面的<’a>感到疑惑，这里的’a 并不是指 Person 这个 struct 的 Lifetime，仅仅是一个泛型参数而已，struct 可以有多个 Lifetime 参数用来约束不同的 field，实际的 Lifetime 应该是所有 fieldLifetime 交集的子集。

```
fn main() {
    let x = 20_u8;
    let stormgbs = Person {
                        age: &x,
                    };
}
```

这里，生命周期/Scope 的示意图如下：

```
                  {   x    stormgbs      *     }
所有者 x:             |_____________________|
所有者 stormgbs:            |_____________|   'a
借用者 stormgbs.age:        |_____________|   stormgbs.age = &x
```

3.4.2.3 结构体方法定义的生命周期

既然<’a>作为 Person 的泛型参数，所以在为 Person 实现方法时也需要加上<’a>，不然：

```
impl Person {
    fn print_age(&self) {
        println!("Person.age = {}", self.age);
    }
}
```

报错：

```
<anon>:5:6: 5:12 error: wrong number of lifetime parameters: expected 1, found 0
[E0107]
<anon>:5 impl Person {
             ^~~~~~
```

正确的做法是：（age 是有生命周期的）

```
impl<'a> Person<'a> {
    fn print_age(&self) {
        println!("Person.age = {}", self.age);
    }
}
```
这样加上<’a>后就可以了。读者可能会疑问，为什么 print_age 中不需要加上’a？这是个好问题。因为 print_age 的输出参数为()，也就是可以不依赖任何输入参数，所以编译器此时可以不必关心和推导 Lifetime。即使是 fn print_age(&self, other_age: &i32) {...}也可以编译通过。

如果 Person 的方法存在输出值（借用）呢？

```
impl<'a> Person<'a> {
    fn get_age(&self) -> &u8 {
        self.age
    }
}
```
get_age 方法的输出值依赖一个输入值&self，这种情况下，Rust 编译器可以自动推导为：

```
impl<'a> Person<'a> {
    fn get_age(&'a self) -> &'a u8 {
        self.age
    }
}
```
如果输出值（借用）依赖了多个输入值呢？

```
impl<'a, 'b> Person<'a> {
    fn get_max_age(&'a self, p: &'a Person) -> &'a u8 {
        if self.age > p.age {
            self.age
        } else {
            p.age
        }
    }
}
```
类似之前的 Lifetime 推导章节，当返回值（借用）依赖多个输入值时，需显示声明 Lifetime。和函数 Lifetime 同理。

其他无论在函数还是在 struct 中，甚至在 enum 中，Lifetime 理论知识都是一样的。希望大家可以慢慢体会和吸收，做到举一反三。

第四章 面向对象编程

4.1 面向对象数据结构

4.1.1 元祖

元祖表示一个大小、类型固定的有序数据组。

```
let y = (2, "hello world");
let x: (i32, &str) = (3, "world hello");

// 然后呢，你能用很简单的方式去访问他们：

// 用 let 表达式
let (w, z) = y; // w=2, z="hello world"

// 用下标

let f = x.0; // f = 3
let e = x.1; // e = "world hello"
```

Rust 虽然函数只有一个返回值，只需要通过元祖，我们可以很容易地返回多个返回值的组合。例子如下：

```
pub fn tuple_test1() {
    let (number, persons) = demo_test();
    println!("{}",number);
    println!("{}",persons.name)
}
struct Person {
    name: String,
    age: u16,
}
fn demo_test() -> (u16, Person) {
    let person = Person {
        name: String::from("binwen"),
        age: 12,
    };
    (14,person)
}
```

4.1.2 结构体

在 Rust 中，结构体是一个跟 tuple 类似 的概念。我们同样可以将一些常用的数据、属性聚合在一起，就形成了一个结构体。

所不同的是，Rust 的结构体有三种最基本的形式。

1. 具名结构体：通常接触的基本都是这个类型的。

```
struct A {
    attr1: i32,
    atrr2: String,
}
```

内部每个成员都有自己的名字和类型。

2. 元祖类型结构体：元组类型结构体

```
struct B(i32, u16, bool);
```

它可以看作是一个有名字的元组，具体使用方法和一般的元组基本类似。

3. 空结构体

结构体内部也可以没有任何成员。

```
struct D;
```

空结构体的内存占用为 0。但是我们依然可以针对这样的类型实现它的“成员函数”。

4.1.3 结构体的方法

Rust 没有继承，它和 Golang 不约而同的选择了 trait(Golang 叫 Interface)作为其实现多态的基础。

不同的是，golang 是匿名继承，rust 是显式继承。如果需要实现匿名继承的话，可以通过隐藏实现类型可以由 generic 配合 trait 作出。

```
struct Person {
    name: String,
}

impl Person {
    fn new(n: &str) -> Person {
        Person {
            name: n.to_string(),
        }
    }

    fn greeting(&self) {
        println!("{} say hello .", self.name);
    }
}

fn main() {
    let peter = Person::new("Peter");
    peter.greeting();
}
```

上面的 impl 中，new 被 Person 这个结构体自身所调用，其特征是 :: 的调用，是一个类函数！ 而带有 self 的 greeting ，是一个成员函数。

4.1.4 再说结构体中引用的生命周期

本小节例子中，结构体的每个字段都是完整的属于自己的。也就是说，每个字段的 owner 都是这个结构体。每个字段的生命周期最终都不会超过这个结构体。

但是如果想要持有一个(可变)引用的值怎么办？例如

```
struct RefBoy {
    loc: &i32,
}
```

则会得到一个编译错误：

```
<anon>:6:14: 6:19 error: missing lifetime specifier [E0106]
<anon>:6         loc: & i32,
```

错误原因：

这种时候，你将持有一个值的引用，因为它本身的生命周期在这个结构体之外，所以对这个结构体而言，它无法准确的判断获知这个引用的生命周期，这在 Rust 编译器而言是不被接受的。

这个时候就需要我们给这个结构体人为的写上一个生命周期，并显式地表明这个引用的生命周期。这个引用需要被借用检查器进行检查。写法如下：

```
struct RefBoy<'a> {
    loc: &'a i32,
}
```

这里解释一下这个符号 <>，它表示的是一个 属于 的关系，无论其中描述的是 生命周期 还是 泛型 。即： RefBoy in 'a。最终我们可以得出个结论，RefBoy 这个结构体，其生命周期一定不能比 'a 更长才行。

需要知道两点：

1.结构体里的引用字段必须要有显式的生命周期。

2.一个被显式写出生命周期的结构体，其自身的生命周期一定小于等于其显式写出的任意一个生命周期。

关于第二点，其实生命周期是可以写多个的，用 , 分隔。

注：生命周期和泛型都写在 <> 里，先生命周期后泛型，用,分隔。

4.2.方法

Rust 中，通过 impl 可以对一个结构体添加成员方法。同时我们也看到了 self 这样的关键字。

impl 中的 self,常见的有三种形式：self、 &self、&mut self 。

虽然方法和 golang interface 非常相像，但是还是要加上类似于 java 的 self，主要原因在于 Rust 的所有权转移机制。

Rust 的笑话：“你调用了一下别人，然后你就不属于你了”。

例如下面代码：

```
struct A {
    a: i32,
}
impl A {
    pub fn show(self) {
        println!("{}", self.a);
    }
}

fn main() {
    let ast = A{a: 12i32};
    ast.show();
    println!("{}", ast.a);
}
```

错误：

```
13:25 error: use of moved value: `ast.a` [E0382]
<anon>:13     println!("{}", ast.a);
```

因为 Rust 本身，在你调用一个函数的时候，如果传入的不是一个引用，那么无疑，这个参数将被这个函数吃掉，即其 owner 将被 move 到这个函数的参数上。同理，impl 中的 self ，如果你写的不是一个引用的话，也是会被默认的 move 掉。

4.2.1 &self 与 &mut self

关于 ref 和 mut ref 的写法和被 move 的 self 写法类似，只不过多了一个引用修饰符号。

需要注意的一点是，你不能在一个 &self 的方法里调用一个 &mut ref ，任何情况下都不行！

```
#[derive(Copy, Clone)]
struct A {
    a: i32,
}
impl A {
    pub fn show(&self) {
        println!("{}", self.a);
        // compile error: cannot borrow immutable borrowed content
`*self` as mutable
        // self.add_one();
    }
    pub fn add_two(&mut self) {
        self.add_one();
        self.add_one();
        self.show();
    }
    pub fn add_one(&mut self) {
        self.a += 1;
    }
}

fn main() {
    let mut ast = A{a: 12i32};
    ast.show();
    ast.add_two();
}
```

需要注意的是，一旦你的结构体持有一个**可变引用**，你，只能在 &mut self 的实现里去改变他！例子：

```
struct Person<'a>{
    name :&'a mut Vec<i32>,
}

impl<'a> Person<'a>{
    fn println_name(&mut self){
        self.name.push(2090);
        println!("{:?}",self.name);
    }

    fn println_name2(&self){
```

```
        println!("{:?}",self.name);
    }

    // error[E0596]: cannot borrow `*self.name` as mutable, as it is
behind a `&` reference
    //  --> src/trait_test.rs:19:9
    //   |
    //18 |     fn println_name3(&self){
    //   |                      ----- help: consider changing this to be
a mutable reference: `&mut self`
    //19 |         self.name.push(2090);
    //   |         ^^^^^^^^^ `self` is a `&` reference, so the data it
refers to cannot be borrowed as mutable
    // fn println_name3(&self){
    //     self.name.push(2090);
    //     println!("{:?}",self.name);
    // }
}


pub fn trait_test1(){
    println!("trait_test1");

    let mut a = vec![1,2,3,4];
    let mut person = Person{
        name:&mut a,
    };
    person.name.push(120);

    person.println_name();
    person.println_name2();

    let a = &person;
    println!("{:?}",a.name);

    // cannot borrow `*a.name` as mutable, as it is behind a `&`
reference
    // a.name.push(200);
    //let a = &person;
    //|             ------- help: consider changing this to be a mutable
reference: `&mut person`
    //45 |     println!("{:?}",a.name);
    //46 |     a.name.push(200);
    //|     ^^^^^^ `a` is a `&` reference, so the data it refers to
cannot be borrowed as mutable

}
```

但是你可以在&self 的方法中读取它。类似于如果一个结构体持有一个可变引用 A，必须通过结构体的可变引用去改变 A 引用的值，而不能通过结构体的不可变引用去改变 A 引用的值，但是可以通过结构体的不可变引用去读取 A 引用的值。

稍微复杂的例子：

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

```
fn longest_with_an_announcement<'a, T>(x: &'a str, y: &'a str, ann: T) ->
&'a str
    where T: Display
{
    println!("Announcement! {}", ann);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

上面了例子引出本章重要的一部分内容：trait。

4.3.trait

trait 的简单使用：使用 **trait** 定义一个特征（可以定义多个）：

```
trait HasArea {
    fn area(&self) -> f64;
}
```

trait 里面的函数可以没有（也可以有）函数体，实现代码交给具体实现它的类型去补充：

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl HasArea for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

fn main() {
    let c = Circle {
        x: 0.0f64,
        y: 0.0f64,
        radius: 1.0f64,
    };
    println!("circle c has an area of {}", c.area());
}
```

4.3.1 泛型参数约束

我们知道泛型可以指任意类型，但有时这不是我们想要的，需要给它一些约束。

```
use std::fmt::Debug;
fn foo<T: Debug>(s: T) {
    println!("{:?}", s);
}
```

Debug 是 Rust 内置的一个 trait，为"{:?}"实现打印内容，函数 foo 接受一个泛型作为参数，并且约定其需要实现 Debug。

可以使用多个 trait 对泛型进行约束：

```
use std::fmt::Debug;
fn foo<T: Debug + Clone>(s: T) {
    s.clone();
```

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

<T: Debug + Clone>中 Debug 和 Clone 使用+连接，标示泛型 T 需要同时实现这两个 trait。

4.3.1.1 泛型参数约束简化（通过 where）

```
use std::fmt::Debug;
fn foo<T: Clone, K: Clone + Debug>(x: T, y: K) {
    x.clone();
    y.clone();
    println!("{:?}", y);
}

// where 从句
fn foo<T, K>(x: T, y: K) where T: Clone, K: Clone + Debug {
    x.clone();
    y.clone();
    println!("{:?}", y);
}

// 或者
fn foo<T, K>(x: T, y: K)
    where T: Clone,
          K: Clone + Debug {
    x.clone();
    y.clone();
    println!("{:?}", y);
}
```

4.3.2 trait 与内置类型

内置类型如：i32, i64 等也可以添加 trait 实现，为其定制一些功能：

```
trait HasArea {
    fn area(&self) -> f64;
}

impl HasArea for i32 {
    fn area(&self) -> f64 {
        *self as f64
    }
}

5.area();
```

这样的做法是有限制的。Rust 有一个“孤儿规则”：当你为某类型实现某 trait 的时候，必须要求类型或者 trait 至少有一个是在当前 crate 中定义的。你不能为第三方的类型实现第三方的 trait 。

在调用 trait 中定义的方法的时候，一定要记得让这个 trait 可被访问。

4.3.3 trait 默认实现

```
trait Foo {
    fn is_valid(&self) -> bool;

    fn is_invalid(&self) -> bool { !self.is_valid() }
}
```

is_invalid 是默认方法，Foo 的实现者并不要求实现它，如果选择实现它，会覆盖掉它的默认行为。

4.3.4 trait 的继承

```
trait Foo {
    fn foo(&self);
}

trait FooBar : Foo {
    fn foobar(&self);
}
```

这样 FooBar 的实现者也要同时实现 Foo：

```
struct Baz;

impl Foo for Baz {
    fn foo(&self) { println!("foo"); }
}

impl FooBar for Baz {
    fn foobar(&self) { println!("foobar"); }
}
```

必须显式实现 Foo，这种写法是错误的：

```
impl FooBar for Baz {
    fn foobar(&self) { println!("foobar"); }

//    --> src/trait_test_three.rs:18:5
//    |
//    18 |     fn foo(&self) { println!("foo"); }
//    |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ not a member of trait `FooBar`

//    fn foo(&self) { println!("foo"); }
}
```

4.3.5 derive 属性

Rust 提供了一个属性 derive 来自动实现一些 trait，这样可以避免重复繁琐地实现他们，能被 derive 使用的 trait 包括：
Clone, Copy, Debug, Default, Eq, Hash, Ord, PartialEq, PartialOrd

```
#[derive(Debug)]
struct Foo;

fn main() {
    println!("{:?}", Foo);
}
```

4.3.6 impl Trait

impl Trait 语法适用于短小的例子，它不过是一个较长形式的语法糖。这被称为 trait bound.使用场景如下：

1.传参数

```
// before
fn foo<T: Trait>(x: T) {

// after
fn foo(x: impl Trait) {
```

2.返回参数

```
fn returns_summarizable() -> impl Summary {
    Tweet {
        username: String::from("horse_ebooks"),
        content: String::from("of course, as you probably already know, people"),
        reply: false,
        retweet: false,
    }
}
```

这个签名表明，“我要返回某个实现了 Summary trait 的类型，但是不确定其具体的类型”。在例子中返回了一个 Tweet，不过调用方并不知情。

3. 使用 trait bound 有条件地实现方法

通过使用带有 trait bound 的泛型参数的 impl 块，可以有条件地只为那些实现了特定 trait 的类型实现方法。例如，下面例子中的类型 Pair<T> 总是实现了 new 方法，不过只有那些为 T 类型实现了 PartialOrd trait（来允许比较） 和 Display trait （来启用打印）的 Pair<T> 才会实现 cmp_display 方法：

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

struct Pair<T> {
    x: T,
    y: T,
}

impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self {
            x,
            y,
        }
    }
}

impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("The largest member is x = {}", self.x);
        } else {
            println!("The largest member is y = {}", self.y);
        }
    }
```

4.闭包

```
// before
fn foo() -> Box<Fn(i32) -> i32> {
    Box::new(|x| x + 1)
}

// after
fn foo() -> impl Fn(i32) -> i32 {
    |x| x + 1
}
```

4.3.7 trait 对象

此外，trait 高级用法还有 trait 对象等等。这部分请查阅 rustPrimer 相应章节。

4.3.8 trait 定义中的生命周期和可变性声明

trait 特性中的可变性和生命周期泛型定义必须和实现它的方法完全一致！不能缺省！

例子：

```
trait HelloArea{
    fn areas<'a>(&mut self, a:&'a mut Vec<i32>, b:&'a mut Vec<i32>) -> &'a mut
Vec<i32>;
}

struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl HasArea for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

impl HelloArea for Circle{
    fn areas<'a>(&mut self, a:&'a mut Vec<i32>, b:&'a mut Vec<i32>) -> &'a mut
Vec<i32>{
        a.push(10000);
        return a;
    }
}

//compile_error!()

//error[E0053]: method `area` has an incompatible type for trait
//  --> src/trait_test_two.rs:23:13
//   |
//13 |     fn area(&mut self) -> f64;
//   |             --------- type in trait
//...
//23 |     fn area(&self) -> f64 {
//   |             ^^^^^ types differ in mutability
```

第五章 属性与 Cargo 配置

5.1 属性

属性（Attribute）是一种通用的用于表达元数据的特性。

在 rust 中大量使用属性，对于 go 程序员而言，属性的大量使用很容易造成困惑。

属性有些类似与 spring boot 中的注解，但又有很大不同。首先 java 就是一门平台无关性的语言。

属性只能用于修饰 rust 中的 item。

rust 中的 item 包括：

1. extern crate 声明

2. use 声明

3. 模块（模块是一个 Item 的容器）

4. 函数

5. type 定义

6. 结构体定义

7. 枚举类型定义

8. 常量定义

9. 静态变量定义

10. Trait 定义

11. 实现（Impl）

这些 Item 是可以互相嵌套的，比如在一个函数中定义一个静态变量、在一个模块中使用 use 声明或定义一个结构体。

5.1 属性

5.1.1 属性的语法

Rust 包括两种 attribute ：

#![attribute 属性描述]

第一种的含义是：

这种#后面有一个!的形式，表示这个属性值应用在当前的对象。通常，这样的 attribute 是针对当前文件设置的。当前文件可能包括两种对象，一个是 crate，另一个可能是 module。需要根据文件的上下文来判断。

判断的依据是，如果是 src/mod.rs 或者 src/lib.rs 那么这个设置就是针对 crate 的，因为这两个文件是 crate 的 root module。

#[attribute 属性描述]

第二种含义是：

这种前面只有#号的属性，表示属性的设置应用到属性下面的紧挨着的元素，即 item 上。

Rust attribute 正则表达式语法：

```
attribute : '#' '!' ? '[' meta_item ']' ;
meta_item : ident [ '=' literal  | '(' meta_seq ')' ] ? ;
meta_seq : meta_item [ ',' meta_seq ] ? ;
```

meta 可以进行无限嵌套。

参看一些 rust attribute 的例子去理解它的语法：

```
#[cfg(not(target_feature = "crt-static"))]
#[derive(PartialEq, Clone)]
#![crate_name = "mycrate"]
#[target_feature(enable = "avx2")]
#[link(name = "CoreFoundation", kind = "framework")]
#![allow(clippy::filter_map)]
#[cfg_attr(linux, path = "linux.rs")]
#[cfg_attr(windows, path = "windows.rs")]
```

按照语法可以写出一个这样例子：

```
#[target(windows,not(config=”true”,ubuntu(“config”=false,sixteen)))]
```

翻译过来的语法规则描述如下：

1.单个标识符代表的属性名，如#[unix]

2.单个标识符代表属性名，后面紧跟着一个=，然后再跟着一个 Value，组成一个键值对，如#[link(name = "openssl")]

3.单个标识符代表属性名，后面跟着一个逗号隔开的子属性的列表，如#[cfg(and(unix, not(windows)))]

5.1.2 几种常见的属性

官方的 attribute 文档：

https://doc.rust-lang.org/reference/attributes.html

RustPrimer 文档：

https://rustcc.gitbooks.io/rustprimer/content/attr-and-compiler-arg/attribute.html

几种常用的 attribute 如下：

5.1.2.1.应用于 crate 的属性

- `crate_name` - 指定 Crate 的名字。如`#[crate_name = "my_crate"]`则可以让编译出的库名字为 `libmy_crate.rlib`。
- `crate_type` - 指定 Crate 的类型，有以下几种选择
  - `"bin"` - 编译为可执行文件；
  - `"lib"` - 编译为库；
  - `"dylib"` - 编译为动态链接库；
  - `"staticlib"` - 编译为静态链接库；
  - `"rlib"` - 编译为 Rust 特有的库文件，它是一种特殊的静态链接库格式，它里面会含有一些元数据供编译器使用，最终会静态链接到目标文件之中。

  例`#![crate_type = "dylib"]`。

- `feature` - 可以开启一些不稳定特性，只可在 nightly 版的编译器中使用。
- `no_builtins` - 去掉内建函数。
- `no_main`- 不生成 `main` 这个符号，当你需要链接的库中已经定义了 `main` 函数时会用到。
- `no_start` - 不链接自带的 `native` 库。
- `no_std` - 不链接自带的 `std` 库。
- `plugin` - 加载编译器插件，一般用于加载自定义的编译器插件库。用法是

5.1.2.2 应用于函数的属性

- `main` - 把这个函数作为入口函数，替代 `fn main`，会被入口函数（Entry Point）调用。
- `plugin_registrar` - 编写编译器插件时用，用于定义编译器插件的入口函数。
- `start` - 把这个函数作为入口函数（Entry Point），改写 `start` language item。
- `test` - 指明这个函数为单元测试函数，在非测试环境下不会被编译。
- `should_panic` - 指明这个单元测试函数必然会 panic。
- `cold` - 指明这个函数很可能是不会被执行的，因此优化的时候特别对待它。

5.1.2.3 应用于 FFI 的属性

`extern` 块可以应用以下属性

- `link_args` - 指定链接时给链接器的参数，平台和实现相关。
- `link` - 说明这个块需要链接一个 native 库，它有以下参数：
  - `name` - 库的名字，比如 `libname.a` 的名字是 `name`；
  - `kind` - 库的类型，它包括
    - `dylib` - 动态链接库
    - `static` - 静态库
    - `framework` - OS X 里的 Framework

5.1.2.4 条件编译

有时候，我们想针对不同的编译目标来生成不同的代码，比如在编写跨平台模块时，针对 Linux 和 Windows 分别使用不同的代码逻辑。

条件编译基本上就是使用 `cfg` 这个属性，直接看例子

```
#[cfg(target_os = "macos")]
fn cross_platform() {
    // Will only be compiled on Mac OS, including Mac OS X
}

#[cfg(target_os = "windows")]
fn cross_platform() {
    // Will only be compiled on Windows
}

// 若条件`foo`或`bar`任意一个成立，则编译以下的 Item
#[cfg(any(foo, bar))]
fn need_foo_or_bar() {

}

// 针对 32 位的 Unix 系统
#[cfg(all(unix, target_pointer_width = "32"))]
fn on_32bit_unix() {

}

// 若`foo`不成立时编译
#[cfg(not(foo))]
fn needs_not_foo() {

}
```

其中，`cfg` 可接受的条件有

- `debug_assertions` - 若没有开启编译优化时就会成立。
- `target_arch = "..."` - 目标平台的 CPU 架构，包括但不限于 `x86`, `x86_64`, `mips`, `powerpc`, `arm` 或 `aarch64`。
- `target_endian = "..."` - 目标平台的大小端，包括 `big` 和 `little`。
- `target_env = "..."` - 表示使用的运行库，比如 `musl` 表示使用的是 MUSL 的 libc 实现, `msvc` 表示使用微软的 MSVC，`gnu` 表示使用 GNU 的实现。 但在部分平台这个数据是空的。
- `target_family = "..."` - 表示目标操作系统的类别，比如 `windows` 和 `unix`。这个属性可以直接作为条件使用，如`#[unix]`，`#[cfg(unix)]`。
- `target_os = "..."` - 目标操作系统，包括但不限于 `windows`, `macos`, `ios`, `linux`, `android`, `freebsd`, `dragonfly`, `bitrig`, `openbsd`, `netbsd`。
- `target_pointer_width = "..."` - 目标平台的指针宽度，一般就是 `32` 或 `64`。
- `target_vendor = "..."` - 生产商，例如 `apple`, `pc` 或大多数 Linux 系统的 `unknown`。
- `test` - 当启动了单元测试时（即编译时加了`--test` 参数，或使用 `cargo test`）。

还可以根据一个条件去设置另一个条件，使用 `cfg_attr`，如

```
#[cfg_attr(a, b)]
```

这表示若 `a` 成立，则这个就相当于`#[cfg(b)]`。

5.1.2.5 Linter 参数

目前的 Rust 编译器已自带的 Linter，它可以在编译时静态帮你检测不用的代码、死循环、编码风格等等。Rust 提供了一系列的属性用于控制 Linter 的行为

- `allow(C)` - 编译器将不会警告对于 `C` 条件的检查错误。
- `deny(C)` - 编译器遇到违反 `C` 条件的错误将直接当作编译错误。
- `forbit(C)` - 行为与 `deny(C)`一样，但这个将不允许别人使用 `allow(C)`去修改。

- warn(C) - 编译器将对于 C 条件的检查错误输出警告。

编译器支持的 Lint 检查可以通过执行 rustc -W help 来查看。

5.1.2.6 编译特性

在非稳定版的 Rust 编译器中，可以使用一些不稳定的功能，比如一些还在讨论中的新功能、正在实现中的功能等。Rust 编译器提供一个应用于 Crate 的属性 feature 来启用这些不稳定的功能，如

```
#![feature(advanced_slice_patterns, box_syntax, asm)]
```

具体可使用的编译器特性会因编译器版本的发布而不同，具体请阅读官方文档。

属性较为繁琐，以官方文档为主。

5.2 cargo 参数配置

5.2.1 package 配置

```
[package]
 # 软件包名称，如果需要在别的地方引用此软件包，请用此名称。
name = "hello_world"

# 当前版本号，这里遵循 semver 标准，也就是语义化版本控制标准。
version = "0.1.0"    # the current version, obeying semver

# 软件所有作者列表
authors = ["you@example.com"]

# 非常有用的一个字段，如果要自定义自己的构建工作流，
# 尤其是要调用外部工具来构建其他本地语言（C、C++、D 等）开发的软件包时。
# 这时，自定义的构建流程可以使用 rust 语言，写在"build.rs"文件中。
build = "build.rs"

# 显式声明软件包文件夹内哪些文件被排除在项目的构建流程之外，
# 哪些文件包含在项目的构建流程中
exclude = ["build/**/*.o", "doc/**/*.html"]
include = ["src/**/*", "Cargo.toml"]

# 当软件包在向公共仓库发布时出现错误时，使能此字段可以阻止此错误。
publish = false

# 关于软件包的一个简短介绍。
description = "..."

# 下面这些字段标明了软件包仓库的更多信息
documentation = "..."
homepage = "..."
repository = "..."

# 顾名思义，此字段指向的文件就是传说中的 ReadMe，
# 并且，此文件的内容最终会保存在注册表数据库中。
readme = "..."

# 用于分类和检索的关键词。
keywords = ["...", "..."]

# 软件包的许可证，必须是 cargo 仓库已列出的已知的标准许可证。
license = "..."

# 软件包的非标许可证书对应的文件路径。
```

```
license-file = "..."
```

5.2.2 依赖的详细配置：

```
# 注意，此处的 cfg 可以使用 not、any、all 等操作符任意组合键值对。
# 并且此用法仅支持 cargo 0.9.0（rust 1.8.0）以上版本。
# 如果是 windows 平台，则需要此依赖。
[target.'cfg(windows)'.dependencies]
winhttp = "0.4.0"

[target.'cfg(unix)'.dependencies]
openssl = "1.0.1"

#如果是 32 位平台，则需要此依赖。
[target.'cfg(target_pointer_width = "32")'.dependencies]
native = { path = "native/i686" }

[target.'cfg(target_pointer_width = "64")'.dependencies]
native = { path = "native/i686" }

# 另一种写法就是列出平台的全称描述
[target.x86_64-pc-windows-gnu.dependencies]
winhttp = "0.4.0"
[target.i686-unknown-linux-gnu.dependencies]
openssl = "1.0.1"

# 如果使用自定义平台，请将自定义平台文件的完整路径用双引号包含
[target."x86_64/windows.json".dependencies]
winhttp = "0.4.0"
[target."i686/linux.json".dependencies]
openssl = "1.0.1"
native = { path = "native/i686" }
openssl = "1.0.1"
native = { path = "native/x86_64" }

# [dev-dependencies]段落的格式等同于[dependencies]段落，
# 不同之处在于，[dependencies]段落声明的依赖用于构建软件包，
# 而[dev-dependencies]段落声明的依赖仅用于构建测试和性能评估。
# 此外，[dev-dependencies]段落声明的依赖不会传递给其他依赖本软件包的项目
[dev-dependencies]
iron = "0.2"
```

5.2.3 自定义编译器配置

cargo 内置五种编译器调用模板，分别为 dev、release、test、bench、doc，分别用于定义不同类型生成目标时的编译器参数，如果我们自己想改变这些编译模板，可以自己定义相应字段的值.

```
# 开发模板，对应`cargo build`命令
[profile.dev]
opt-level = 0  # 控制编译器的 --opt-level 参数，也就是优化参数
debug = true   # 控制编译器是否开启 `-g` 参数
rpath = false  # 控制编译器的 `-C rpath` 参数
lto = false    # 控制`-C lto` 参数，此参数影响可执行文件和静态库的生成，
debug-assertions = true  # 控制调试断言是否开启
```

```
codegen-units = 1 # 控制编译器的 `-C codegen-units` 参数。注意，当`lto = true`时，此字段值被忽略

# 发布模板，对应`cargo build --release`命令
[profile.release]
opt-level = 3
debug = false
rpath = false
lto = false
debug-assertions = false
codegen-units = 1

# 测试模板，对应`cargo test`命令
[profile.test]
opt-level = 0
debug = true
rpath = false
lto = false
debug-assertions = true
codegen-units = 1

# 性能评估模板，对应`cargo bench`命令
[profile.bench]
opt-level = 3
debug = false
rpath = false
lto = false
debug-assertions = false
codegen-units = 1

# 文档模板，对应`cargo doc`命令
[profile.doc]
opt-level = 0
debug = true
rpath = false
lto = false
debug-assertions = true
codegen-units = 1
```

5.2.4 feature 段落

[features]段落中的字段被用于条件编译选项或者是可选依赖。例如：

```
[package]
name = "awesome"

[features]
# 此字段设置了可选依赖的默认选择列表，
# 注意这里的"session"并非一个软件包名称，
# 而是另一个 featrue 字段 session
default = ["jquery", "uglifier", "session"]

# 类似这样的值为空的 feature 一般用于条件编译，
# 类似于`#[cfg(feature = "go-faster")]`。
go-faster = []

# 此 feature 依赖于 bcrypt 软件包，
# 这样封装的好处是未来可以对 secure-password 此 feature 增加可选项目。
secure-password = ["bcrypt"]

# 此处的 session 字段导入了 cookie 软件包中的 feature 段落中的 session 字段
session = ["cookie/session"]

[dependencies]
# 必要的依赖
cookie = "1.2.0"
```

```
oauth = "1.1.0"
route-recognizer = "=2.1.0"

# 可选依赖
jquery = { version = "1.0.2", optional = true }
uglifier = { version = "1.5.3", optional = true }
bcrypt = { version = "*", optional = true }
civet = { version = "*", optional = true }
```

如果其他软件包要依赖使用上述 awesome 软件包，可以在其描述文件中这样写：

```
[dependencies.awesome]
version = "1.3.5"
default-features = false # 禁用 awesome 的默认 features
features = ["secure-password", "civet"] # 使用此处列举的各项 features
```

使用 features 时需要遵循以下规则：

1. feature 名称在本描述文件中不能与出现的软件包名称冲突

2. 除了 default feature，其他所有的 features 均是可选的

3. features 不能相互循环包含

4. 开发依赖包不能包含在内

5. features 组只能依赖于可选软件包

features 的一个重要用途就是，当开发者需要对软件包进行最终的发布时，在进行构建时可以声明暴露给终端用户的 features，这可以通过下述命令实现：

```
$ cargo build --release --features "shumway pdf"
```

第六章 Rust 语言高级特性

6.1 函数式编程

6.1.1 闭包

闭包的定义有两种：

闭包（英语：Closure），又称词法闭包（Lexical Closure）或函数闭包（function closures），是引用了自由变量的函数。这个被引用的自由变量将和这个函数一同存在，即使已经离开了创造它的环境也不例外。

有另一种说法认为闭包是由函数和与其相关的引用环境组合而成的实体。闭包在运行时可以有多个实例，不同的引用环境和相同的函数组合可以产生不同的实例。

一个简单实用闭包的例子：

```
pub fn closure_test1(){
    println!("closure_test1");

    let mut a = 200;

    let plus_two = |mut x| {
        let mut result: i32 = x;

        result += 1;
        result += 1;

        x += 100;
        println!("{}",x);

        result
    };

    println!("{}",plus_two(a));

    a = 100;

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

```
计算结果：
closure_test1
300
202
100
```

rust 闭包的简化写法：

```
fn  add_one_v1   (x: u32) -> u32 { x + 1 }
let add_one_v2 = |x: u32| -> u32 { x + 1 };
let add_one_v3 = |x|             { x + 1 };
let add_one_v4 = |x|               x + 1  ;
```

第一行展示了一个函数定义，而第二行展示了一个完整标注的闭包定义。第三行闭包定义中省略了类型注解，而第四行去掉了可选的大括号，因为闭包体只有一行。这些都是有效的闭包定义，并在调用时产生相同的行为。

之所以把它称为“闭包”是因为它们“包含在环境中”（close over their environment）。这看起来像：

```
let num = 5;
let plus_num = |x: i32| x + num;

assert_eq!(10, plus_num(5));
```

6.1.2 闭包捕获周围环境的方式

闭包可以通过三种方式捕获其环境，他们直接对应函数的三种获取参数的方式：获取所有权，可变借用和不可变借用。这三种捕获值的方式被编码为如下三个 Fn trait：

1.FnOnce：消费从周围作用域捕获的变量，闭包周围的作用域被称为其 环境，environment。为了消费捕获到的变量，闭包必须获取其所有权并在定义闭包时将其移动进闭包。其名称的 Once 部分代表了闭包不能多次获取相同变量的所有权的事实，所以它只能被调用一次。

2.FnMut：获取可变的借用值所以可以改变其环境

3.Fn：从其环境获取不可变的借用值

Fn 获取&self，FnMut 获取&mut self，而 FnOnce 获取 self。这包含了所有 3 种通过通常函数调用语法的 self。

```
pub fn trait_test_four_main() {
    println!("trait_test_four");

    let vec = vec![1, 2, 3, 4];
    anonymous_fnonce();
    anonymous_fnonce_callback();
    anonymous_fnmut();
    anonymous_fnmut_callback();
    anonymous_fn();
    anonymous_fn_callback();
}

// 匿名函数中的 FnOnce/FnMut/Fn

// 首先 FnOnce/FnMut/Fn 这三个东西被称为 Trait,
// 默认情况下它们是交给 rust 编译器去推理的, 大致的推理原则是:
//     FnOnce: 当指定这个 Trait 时, 匿名函数内访问的外部变量必须拥有所有权.
//     FnMut: 当指定这个 Trait 时, 匿名函数可以改变外部变量的值.
//     Fn: 当指定这个 Trait 时, 匿名函数只能读取(borrow value immutably)变量值.

// FnOnce inline way
// 以获取所有权的方式来获取其所在的环境的所有变量.
fn anonymous_fnonce() {
    let fn_name = "anonymous_fnonce";
    let mut b = String::from("hello");
    // 通过使用 move 的方式, 把所有权转移进来, rust 编译器
    // 会自动推理出这是一个 FnOnce Trait 匿名函数.
    let pushed_data = move || {
        // 由于所有权转移进来, 因此 b 已经被移除掉.
        // 因此这个匿名函数不可能在被执行第二遍.
        b.push_str(" world!");
        b
    };
    println!("{}: {}", fn_name, pushed_data()); // 这里只能运行一次.
```

```
                                                    // println!("{}: {}", fn_name,
pushed_data());     // 再次运行会报错.
}

// FnOnce callback way
fn anonymous_fnonce_callback() {
    let fn_name = "anonymous_fnonce_callback";
    fn calculate<T>(callback: T) -> i32
    where
        T: FnOnce() -> String,
    {
        let data = callback();
        data.len() as i32
    }

    let x = " world!";
    let mut y = String::from("hello");
    let result = calculate(|| {
        y.push_str(x);
        y
    });
    println!("{}: {}", fn_name, result);
}

// FnMut inline way
// 以 mutable 的方式获取其所在的环境的所有变量.
fn anonymous_fnmut() {
    let fn_name = "anonymous_fnmut";
    let mut b = String::from("hello");

    // rust 自动检测到 pushed_data 这个匿名函数要修改其外部的环境变量.
    // 因此自动推理出 pushed_data 是一个 FnMut 匿名函数.
    let pushed_data = || {
        b.push_str(" world!");

        // 由于 rust 的 mutable 原则是, 只允许一个 mut 引用, 因此 变量 b 不能
        // 再被其他代码引用, 所以这里要返回更改后的结果.
        b
    };
    let c = pushed_data();
    println!("{}: {}", fn_name, c);


    //error[E0382]: borrow of moved value: `b`
    //  --> src/trait_test_four.rs:77:42
    //   |
    //62 |     let mut b = String::from("hello");
    //   |         ----- move occurs because `b` has type `std::string::String`, which
does not implement the `Copy` trait
    //...
    //66 |     let pushed_data = || {
    //   |                       -- value moved into closure here
    //67 |         b.push_str(" world!");
    //   |         - variable moved due to use in closure
    //...
    //77 |     println!("b is borrowed as inmut {}",b);
    //   |                                         ^ value borrowed here after move

    //compile error
    //println!("b is borrowed as inmut {}",b);
}

// FnMut callback way.
fn anonymous_fnmut_callback() {
    let fn_name = "anonymous_fnmut_callback";
    fn calculate<T>(mut callback: T)
    where
        T: FnMut(),
    {
```

```
        callback()
    }

    let mut b = String::from("hello");
    calculate(|| {
        b.push_str(" world!");
    });
    println!("{}: {}", fn_name, b);
}

// Fn inline way
// 以 immutable 的方式获取其所在的环境的所有变量.
fn anonymous_fn() {
    let fn_name = "anonymous_fn";
    let mut a = String::from("hello");
    let b = String::from(" world!");
    let pushed_data = |x: &mut String| {
        // b 再这里被引用, 但是最后还能被打印, 证明它是被 immutable 引用.
        x.push_str(&*b);
        println!("{}: {}", fn_name, x);
        println!("b is borrowed as inmut {}",b);
    };
    pushed_data(&mut a);


    println!("{}: {}", fn_name, b);
}

// Fn callback way
fn anonymous_fn_callback() {
    let fn_name = "anonymous_fn_callback";
    fn calculate<T>(callback: T)
    where
        T: Fn(),
    {
        callback();
    }

    let a = String::from("hello");
    let b = String::from(" world!");
    calculate(|| {
        let joined = format!("{}{}", &*a, &*b);
        println!("{}: {}", fn_name, joined)
    })
}
```

上述例子中包含以 callback 为函数参数，实现功能。

6.1.3 函数指针

一个函数指针有点像一个没有环境的闭包。因此，你可以传递一个函数指针给任何函数除了作为闭包参数，下面的代码可以工作：

```
fn call_with_one(some_closure: &Fn(i32) -> i32) -> i32 {
    some_closure(1)
}

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

let f = add_one;

let answer = call_with_one(&f);

assert_eq!(2, answer);
```

在这个例子中，我们并不是严格的需要这个中间变量 f，函数的名字就可以了：

```
let answer = call_with_one(&add_one);
```
此外闭包的用法还有返回闭包，返回闭包使用价值不大，反而会引起困惑。直接返回函数就可以了。

6.2 unsafe 与原始指针

首先介绍原始指针（即裸指针）

6.2.1 裸指针

Rust 中的裸指针和 C++中的裸指针不同。

*const T 和*mut T 在 Rust 中被称为“裸指针”。它允许别名，允许用来写共享所有权的类型，甚至是内存安全的共享内存类型如：Rc<T>和 Arc<T>，但是赋予你更多权利的同时意味着你需要担当更多的责任：

1. 不能保证指向有效的内存，甚至不能保证是非空的

2. 没有任何自动清除，所以需要手动管理资源

3. 是普通旧式类型，也就是说，它不移动所有权，因此 Rust 编译器不能保证不出像释放后使用这种 bug

4. 缺少任何形式的生命周期，不像&，因此编译器不能判断出悬垂指针

5. 除了不允许直接通过*const T 改变外，没有别名或可变性的保障

从上面可以看到，使用裸指针基本和 rust 的静态类型检查告别。

几个比较好的例子可以说明用法。

```
let a = 1;
let b = &a as *const i32;

let mut x = 2;
let y = &mut x as *mut i32;
```
例子二：

```
let a = 1;
let b = &a as *const i32;
let c = unsafe { *b };
println!("{}", c);
```
例子三：

```
let a: Box<i32> = Box::new(10);
// 我们需要先解引用 a，再隐式把 & 转换成 *
let b: *const i32 = &*a;
// 使用 into_raw 方法
let c: *const i32 = Box::into_raw(a);
```
例子四：

```
struct Student {
    name: i32,
}
```

```
fn main() {
    {
        let mut x = 5;
        let raw = &mut x as *mut i32;
        let mut points_at = unsafe { *raw };
        points_at = 12;

        x = 1234;
        println!("{}", x);

        println!("raw points at {}", points_at);
    }

    {
        let a: Box<Vec<i32>> = Box::new(vec![1, 2, 3, 4]);
        // 我们需要先解引用 a，再隐式把 & 转换成 *
        let b: *const Vec<i32> = &*a;
        // 使用 into_raw 方法
        let c: *const Vec<i32> = Box::into_raw(a);
    }

    {
        let mut x = 2;
        let y = &mut x as *mut i32;
        let mut c = unsafe { *y };
        c = 100;
        //copy
        println!("{}", c);
        println!("{}", x);

        //output
        //100
        //2
    }

    //一般理解，*v 操作，是 &v 的反向操作，
    //即试图由资源的引用获取到资源的拷贝（如果资源类型实现了 Copy），或所有权（资源类型没有实现 Copy）。
    {
        let mut x = String::from("hello");
        let y = &mut x as *mut String;
        let mut c = unsafe { &mut *y };
        c.push('c');

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

        let z = &mut x as *mut String;
        let mut d = unsafe { &mut *z };
        d.push('d');
        println!("{}", c);
        println!("{}", d);
        println!("{}", x);

        //get the ownership
        //helloc
        //helloc
        //hellocd
        //hellocd
        //hellocd
    }
}
```

6.2.2 unsafe

Rust 的内存安全依赖于强大的类型系统和编译时检测，不过它并不能适应所有的场景。 首先，所有的编程语言都需要跟外部的“不安全”接口打交道，调用

外部库等，在“安全”的 Rust 下是无法实现的；其次，“安全”的 Rust 无法高效表示复杂的数据结构，特别是数据结构内部有各种指针互相引用的时候；再次， 事实上还存在着一些操作，这些操作是安全的，但不能通过编译器的验证。

因此在安全的 Rust 背后，还需要 unsafe 的支持。

unsafe 块能允许程序员做的额外事情有：

1.解引用一个裸指针*const T 和*mut T

```
let x = 5;
let raw = &x as *const i32;
let points_at = unsafe { *raw };
println!("raw points at {}", points_at);
```

2.读写一个可变的静态变量 static mut

```
static mut N: i32 = 5;
unsafe {
    N += 1;
    println!("N: {}", N);
}
```

3.调用一个不安全函数(FFI)

```
unsafe fn foo() {
    //实现
}
fn main() {
    unsafe {
        foo();
    }
}
```

6.2.3 Safe!= no bug

对于 Rust 来说禁止你做任何不安全的事是它的本职，不过有些是编写代码时的 bug，它们并不属于“内存安全”的范畴：

- 死锁
- 内存或其他资源溢出
- 退出未调用析构函数
- 整型溢出

使用 unsafe 时需要注意一些特殊情形：

- 数据竞争
- 解引用空裸指针和悬垂裸指针
- 读取未初始化的内存
- 使用裸指针打破指针重叠规则

- &mut T 和&T 遵循 LLVM 范围的 noalias 模型，除了如果&T 包含一个 UnsafeCell<U>的话。不安全代码必须不能违反这些重叠（aliasing）保证
- 不使用 UnsafeCell<U>改变一个不可变值/引用
- 通过编译器固有功能调用未定义行为：
  - 使用 std::ptr::offset（offset 功能）来索引超过对象边界的值，除了允许的末位超出一个字节
  - 在重叠（overlapping）缓冲区上使用 std::ptr::copy_nonoverlapping_memory（memcpy32/memcpy64 功能）
- 原生类型的无效值，即使是在私有字段/本地变量中：
  - 空/悬垂引用或装箱
  - bool 中一个不是 false（0）或 true（1）的值
  - enum 中一个并不包含在类型定义中判别式
  - char 中一个代理字（surrogate）或超过 char::MAX 的值
  - str 中非 UTF-8 字节序列
- 在外部代码中使用 Rust 或在 Rust 中使用外部语言。

6.3 FFI（Foreign Function Interface）

FFI(Foreign Function Interface)是用来与其它语言交互的接口，通常有两种类型：调用其他语言和供其他语言调用。

在 rust 中一般常见的主要是和 C 打交道。调用 C 语言写的代码和编程库供 C 调用。

使用 rust 调用 C 语言相对于使用于 C 语言调用 Rust 稍微复杂一些。

这和 cgo 正好相反。很大部分原因在于 rust 没有运行时。

github 上有一个非常好的例子：

https://github.com/alexcrichton/rust-ffi-examples

有简单的 rust 和各种语言交互的例子。

6.3.1 rust 调用 ffi 函数

rust 调用 ffi 稍微复杂一些。主要原因在于 rust 编程意义上的数据结构和传递给 c 语言的数据结构是不能通用的。

因此需要引入和 C 通信的 crate：libc。

由于 cffi 的数据类型与 rust 不完全相同，我们需要引入 libc 库来表达对应 ffi 函数中的类型。

在 Cargo.toml 中添加以下行:

```
[dependencies]
libc = "0.2.9"
```

主要有以下几步：

6.3.1.1 声明你的 ffi 函数

就像 c 语言需要#include 声明了对应函数的头文件一样，rust 中调用 ffi 也需要对对应函数进行声明。

```
use libc::c_int;
use libc::c_void;
use libc::size_t;

#[link(name = "yourlib")]
extern {
    fn your_func(arg1: c_int, arg2: *mut c_void) -> size_t; // 声明 ffi 函数
    fn your_func2(arg1: c_int, arg2: *mut c_void) -> size_t;
    static ffi_global: c_int; // 声明 ffi 全局变量
}
```

声明一个 ffi 库需要一个标记有#[link(name = "yourlib")]的 extern 块。name 为对应的库(so/dll/dylib/a)的名字。 如：如果你需要 snappy 库(libsnappy.so/libsnappy.dll/libsnappy.dylib/libsnappy.a)，则对应的 name 为 snappy。 在一个 extern 块中你可以声明任意多的函数和变量。

在有的程序中#[link(name = "yourlib")]不是必须的，只需保证可以最终 link 上即可。

6.3.1.2 调用 ffi 函数

声明完成后就可以进行调用了。 由于此函数来自外部的 c 库，所以 rust 并不能保证该函数的安全性。因此，调用任何一个 ffi 函数需要一个 unsafe 块。

```
let result: size_t = unsafe {
    your_func(1 as c_int, Box::into_raw(Box::new(3)) as *mut c_void)
};
```

6.3.1.3 封装 unsafe，暴露安全接口

作为一个库作者，对外暴露不安全接口是一种非常不合格的做法。在做 c 库的 rust binding 时，我们做的最多的将是将不安全的 c 接口封装成一个安全接口。通常做法是：在一个叫 ffi.rs 之类的文件中写上所有的 extern 块用以声明 ffi 函数。在一个叫 wrapper.rs 之类的文件中进行包装：

```
// ffi.rs
#[link(name = "yourlib")]
extern {
    fn your_func(arg1: c_int, arg2: *mut c_void) -> size_t;
}
```

与

```
// wrapper.rs
fn your_func_wrapper(arg1: i32, arg2: &mut i32) -> isize {
    unsafe { your_func(1 as c_int, Box::into_raw(Box::new(3)) as *mut c_void) } as
isize
}
```

对外暴露(pub use) your_func_wrapper 函数即可。

6.3.1.4 数据结构等映射

libc 为我们提供了很多原始数据类型，比如 c_int， c_float 等，但是对于自定义类型，如结构体，则需要我们自行定义。

数据结构是传递给 C 语言使用，并从 C 语言获取结果返回。

细节比较多，具体去查使用例子。这块相对而言还是比较复杂的。

6.3.1.5 静态库/动态库

前面提到了声明一个外部库的方式--#[link]标记，此标记默认为动态库。但如果是静态库，可以使用#[link(name = "foo", kind = "static")]来标记。 此外，对于 osx 的一种特殊库--framework，还可以这样标记#[link(name = "CoreFoundation", kind = "framework")].

6.3.1.6 bind-gen

是不是觉得把一个个函数和全局变量在 extern 块中去声明，对应的数据结构去手动创建特别麻烦？没关系，rust-bindgen 来帮你搞定。 rust-bindgen 是一个能从对应 c 头文件自动生成函数声明和数据结构的工具。创建一个绑定只需要./bindgen [options] input.h 即可。

例子：

```
typedef struct Doggo {
    int many;
    char wow;
} Doggo;

void eleven_out_of_ten_majestic_af(Doggo* pupper);
```

转换后的结果：

```
/* automatically generated by rust-bindgen */

#[repr(C)]
pub struct Doggo {
    pub many: ::std::os::raw::c_int,
    pub wow: ::std::os::raw::c_char,
}

extern "C" {
    pub fn eleven_out_of_ten_majestic_af(pupper: *mut Doggo);
}
```

6.3.1.7 一个简单的例子

build.rs(通过 cc 生成相应库文件)

```
extern crate cc;

fn main() {
    cc::Build::new()
        .file("src/double.c")
        .compile("libdouble.a");
}
```

Cargo.toml

```
[package]
name = "rust-to-c"
version = "0.1.0"
authors = ["Alex Crichton <alex@alexcrichton.com>"]
build = "build.rs"

[dependencies]
libc = "0.2"

[build-dependencies]
cc = "1.0"
```

double.c

```
int double_input(int input) {
    return input * 2;
}
```

main.rs

```
extern crate libc;

extern {
    fn double_input(input: libc::c_int) -> libc::c_int;
}

fn main() {
    let input = 4;
    let output = unsafe { double_input(input) };
    println!("{} * 2 = {}", input, output);
}
```

6.3.2 将 rust 编译成库

这一小节主要说明如何把 rust 编译成库让别的语言通过 cffi 调用。

6.3.2.1 调用约定和 Mangle

为了能让 rust 的函数通过 ffi 被调用，需要加上 extern "C"对函数进行修饰。

但由于 rust 支持重载，所以函数名会被编译器进行混淆，就像 c++一样。因此当你的函数被编译完毕后，函数名会带上一串表明函数签名的字符串。

比如：fn test() {}会变成_ZN4test20hf06ae59e934e5641haaE. 这样的函数名为 ffi 调用带来了困难，因此，rust 提供了#[no_mangle]属性为函数修饰。
对于带有#[no_mangle]属性的函数，rust 编译器不会为它进行函数名混淆。如：

```
#[no_mangle]
extern "C" fn test() {}
```

在 nm 中观察到为

```
...
00000000001a7820 T test
...
```

至此，test 函数将能够被正常的由 cffi 调用。

6.3.2.2 指定 Crate 类型

rustc 默认编译产生 rust 自用的 rlib 格式库，要让 rustc 产生动态链接库或者静态链接库，需要显式指定。

方法 1：在文件中指定。 在文件头加上#![crate_type = "foo"]，其中 foo 的可选类型有 bin， lib， rlib， dylib， staticlib.分别对应可执行文件，默认(将由 rustc 自己决定)， rlib 格式，动态链接库，静态链接库。

方法 2：编译时给 rustc 传--crate-type 参数。参数内容同上。

方法 3：使用 cargo，指定 crate-type = ["foo"]， foo 可选类型同 1。

6.3.2.3 反射的使用

由于在跨越 ffi 过程中，rust 类型信息会丢失，比如当用 rust 提供一个 OpaqueStruct 给别的语言时：

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

#[derive(Debug)]
struct Foo<T> {
  t: T
}

#[no_mangle]
extern "C" fn new_foo_vec() -> *const c_void {
    Box::into_raw(Box::new(Foo {t: vec![1,2,3]})) as *const c_void
}

#[no_mangle]
extern "C" fn new_foo_int() -> *const c_void {
    Box::into_raw(Box::new(Foo {t: 1})) as *const c_void
}

fn push_foo_element(t: &mut Foo<Vec<i32>>) {
    t.t.push(1);
}

#[no_mangle]
extern "C" fn push_foo_element_c(foo: *mut c_void){
    let foo2 = unsafe {
        &mut *(foo as *mut Foo<Vec<i32>>) // 这么确定是 Foo<Vec<i32>>? 万一 foo 是
Foo<i32>怎么办？
    };
    push_foo_element(foo3);
}
```

以上代码中完全不知道 foo 是一个什么东西。安全也无从说起了，只能靠文档。因此在 ffi 调用时往往会丧失掉 rust 类型系统带来的方便和安全。在这里提供一个小技巧:使用 Box<Box<Any>>来包装你的类型。

rust 的 Any 类型为 rust 带来了运行时反射的能力，使用 Any 跨越 ffi 边界将极大提高程序安全性。

```
use std::any::Any;

#[derive(Debug)]
struct Foo<T> {
  t: T
}
```

```
#[no_mangle]
extern "C" fn new_foo_vec() -> *const c_void {
    Box::into_raw(Box::new(Box::new(Foo {t: vec![1,2,3]}) as Box<Any>)) as *const
c_void
}

#[no_mangle]
extern "C" fn new_foo_int() -> *const c_void {
    Box::into_raw(Box::new(Box::new(Foo {t: 1}) as Box<Any>)) as *const c_void
}

fn push_foo_element(t: &mut Foo<Vec<i32>>) {
    t.t.push(1);
}

#[no_mangle]
extern "C" fn push_foo_element_c(foo: *mut c_void){
    let foo2 = unsafe {
        &mut *(foo as *mut Box<Any>)
    };
    let foo3: Option<&mut Foo<Vec<i32>>> = foo2.downcast_mut(); // 如果 foo2 不是*const
Box<Foo<Vec<i32>>>, 则 foo3 将会是 None
    if let Some(value) = foo3 {
      push_foo_element(value);
    }
}
```

上面的例子中可以接受调用方的*mut c_void 类型，然后进行运行时反射，对输入数据进行处理。

6.3.2.4 一个简单的例子

链接地址：

https://github.com/alexcrichton/rust-ffi-examples/tree/master/c-to-rust

Makefile

```
ifeq ($(shell uname),Darwin)
    LDFLAGS := -Wl,-dead_strip
else
    LDFLAGS := -Wl,--gc-sections -lpthread -ldl
endif

all: target/double
	target/double

target:
	mkdir -p $@

target/double: target/main.o target/debug/libdouble_input.a
	$(CC) -o $@ $^ $(LDFLAGS)

target/debug/libdouble_input.a: src/lib.rs Cargo.toml
	cargo build

target/main.o: src/main.c | target
	$(CC) -o $@ -c $<

clean:
	rm -rf target
```

Cargo.toml

```
[package]
name = "c-to-rust"
version = "0.1.0"
authors = ["Alex Crichton <alex@alexcrichton.com>"]

[lib]
name = "double_input"
crate-type = ["staticlib"]
```

lib.rs

```
#![crate_type = "staticlib"]

#[no_mangle]
pub extern fn double_input(input: i32) -> i32 {
    input * 2
}
```

main.c

```
#include <stdint.h>
#include <stdio.h>

extern int32_t double_input(int32_t input);

int main() {
    int input = 4;
    int output = double_input(input);
    printf("%d * 2 = %d\n", input, output);
    return 0;
}
```

Makefile 中会去链接 rust 生成的 lib，看起来这么简单的原因是 rust 内部提供的就是一个运算口子，捕获 C 传递的参数，执行相应的函数功能，并返回相应的 c 可以理解的结果。这些功能都内置了。开发上大大节省时间。

6.4 堆，栈，BOX

6.4.1 堆和栈

一般而言，在编译期间不能确定大小的数据类型都需要使用堆上内存，因为编译器无法在栈上分配 编译期未知大小 的内存，所以诸如 String, Vec 这些类型的内存其实是被分配在堆上的。换句话说，我们可以很轻松的将一个 Vec move 出作用域而不必担心消耗，因为数据实际上不会被复制。

对比一下 Vec<i32> 和 Vec<Box<i32>> 内存布局

```
Vec<i32>

(stack)    (heap)
┌──────┐   ┌───┐
│ vec1 │──→│ 1 │
└──────┘   ├───┤
           │ 2 │
           ├───┤
           │ 3 │
           ├───┤
           │ 4 │
           └───┘
```

和

```
Vec<Box<i32>>

(stack)    (heap)
┌──────┐   ┌──┐   ┌──┐
│ vec2 │ ──→ │  │ ──→ │ 1 │
└──────┘   ├──┤   └──┘
           │  │ ──→ ┌──┐
           ├──┤     │ 2 │
           │  │     └──┘
           ├──┤   ┌──┐
           │  │ ──→ │ 3 │
           ├──┤   └──┘
           │  │   ┌──┐
           └──┘ ──→ │ 4 │
                    └──┘
```

堆和栈区别：

栈内存从高位地址向下增长，且栈内存分配是连续的，一般操作系统对栈内存大小是有限制的，Linux/Unix 类系统上面可以通过 ulimit 设置最大栈空间大小，所以 C 语言中无法创建任意长度的数组。在 Rust 里，函数调用时会创建一个临时栈空间，调用结束后 Rust 会让这个栈空间里的对象自动进入 Drop 流程，最后栈顶指针自动移动到上一个调用栈顶，无需程序员手动干预，因而栈内存申请和释放是非常高效的。

相对地，堆上内存则是从低位地址向上增长，堆内存通常只受物理内存限制，而且通常是不连续的，一般由程序员手动申请和释放的，如果想申请一块连续内存，则操作系统需要在堆中查找一块未使用的满足大小的连续内存空间，故其效率比栈要低很多，尤其是堆上如果有大量不连续内存时。另外内存使用完也必须由程序员手动释放，不然就会出现内存泄漏，内存泄漏对需要长时间运行的程序(例如守护进程)影响非常大。

rust 资源的管理比较复杂，资源和标志符要分开来看。

目前一个只包含 i32 的结构体可能放在栈上，一个包含字符串的结构体放在那里？不太确定，可以确定的是字符串肯定放在堆上，栈上数据不支持动态扩容。

6.4.2 BOX

Rust 可以强制把某些数据放到堆上。例如：

```
fn main() {
    let b = Box::new(5);
    println!("b = {}", b);
}
```

此外，除了数据被储存在堆上而不是栈上之外，box 没有性能损失。不过也没有很多额外的功能。它们多用于如下场景：

- 当有一个在编译时未知大小的类型，而又想要在需要确切大小的上下文中使用这个类型值的时候
- 当有大量数据并希望在确保数据不被拷贝的情况下转移所有权的时候
- 当希望拥有一个值并只关心它的类型是否实现了特定 trait 而不是其具体类型的时候（Box<Trait>，后来有了 impl trait 进行替换）

6.4.2.1 box 允许创建递归类型

```
#[derive(Debug)]
enum List {
    Cons(i32, Box<List>),
    Nil,
}
use List::{Cons, Nil};

struct BigStruct {
    one: i32,
    two: i32,
    // etc
    one_hundred: i32,
}

fn foo(x: Box<BigStruct>) -> BigStruct {
    *x
}

pub fn box_test_1() {
    println!("box test");

    let list = Cons(1, Box::new(Cons(2, Box::new(Cons(3, Box::new(Nil))))));

    println!("{:?}", list);

    let bs = BigStruct {
        one: 1,
        two: 2,
        one_hundred: 100,
    };

    let mut x = Box::new(bs);
    println!("{}", x.one_hundred);
    println!("{}", &x.one);
    x.one_hundred = 200;
    println!("{}", &x.one_hundred);

    let m = &x;

    let z = &mut x;
    z.one_hundred = 500;

    //error[E0502]: cannot borrow `x` as mutable because it is also borrowed as
immutable
    //  --> src/box_test.rs:41:13
    //   |
    //39 |     let m = &x;
    //   |             -- immutable borrow occurs here
    //40 |
    //41 |     let z = &mut x;
    //   |             ^^^^^^ mutable borrow occurs here
    //...
    //44 |     println!("{}",m.one_hundred);
    //   |                   ------------ immutable borrow later used here

    //compile error
    //println!("{}",m.one_hundred);

    let y = foo(x);
    println!("{}", y.one);
    println!("{}", y.one_hundred);
}
```

上面例子包含了递归类型 list。如果按照

```
enum List {
    Cons(i32, List),
```

```
    Nil,
}
```

去定义，则会报如下错误：

```
error[E0072]: recursive type `List` has infinite size
 --> src/main.rs:1:1
  |
1 | enum List {
  | ^^^^^^^^^ recursive type has infinite size
2 |     Cons(i32, List),
  |               ----- recursive without indirection
  |
  = help: insert indirection (e.g., a `Box`, `Rc`, or `&`) at some point to
  make `List` representable
```

这个错误表明这个类型 “有无限的大小”。其原因是 List 的一个成员被定义为是递归的：它直接存放了另一个相同类型的值。这意味着 Rust 无法计算为了存放 List 值到底需要多少空间。让我们一点一点来看：首先了解一下 Rust 如何决定需要多少空间来存放一个非递归类型。

box 指针和下面例子的 b 有点类似。

```
pub fn rust_ptr_test() {
    println!("rust_ptr_test");

    let mut a = vec![1, 2, 3, 4];

    println!("{:?}", a);

    let mut b = &mut a;
    println!("{:?}", b);

    let c = &b;
    println!("{:?}", c);

    let d = &mut b;
    println!("{:?}", d);
    d.push(100);
    println!("{:?}", d);

    //error[E0502]: cannot borrow `b` as mutable because it is also borrowed as
immutable
    //  --> src/rust_ptr_test.rs:15:13
    //   |
    //12 |     let c = &b;
    //   |             -- immutable borrow occurs here
    //...
    //15 |     let d = &mut b;
    //   |             ^^^^^^ mutable borrow occurs here
    //...
    //21 |     println!("{:?}",c);
    //   |                     - immutable borrow later used here

    //compile error
    //println!("{:?}",c);
}
```

6.5 智能指针

6.5.1 Rc 与 Arc

6.5.1.1 Rc 与 Rc Weak

Rc 用于同一线程内部，通过 use std::rc::Rc 来引入。它有以下几个特点：

1.用 Rc 包装起来的类型对象，是 immutable 的，即 不可变的。即你无法修改 Rc<T> 中的 T 对象，只能读（除非使用 RefCell。）；

2.一旦最后一个拥有者消失，则资源会被自动回收，这个生命周期是在编译期就确定下来的；

3.Rc 只能用于同一线程内部，不能用于线程之间的对象共享（不能跨线程传递）；

4.Rc 实际上是一个指针，它不影响包裹对象的方法调用形式（即不存在先解开包裹再调用值这一说）。和 Box 有点类似。

使用例子：

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

let five = Rc::new(5);
let five2 = five.clone();
let five3 = five.clone();
```

Weak 通过 use std::rc::Weak 来引入。

Rc 是一个引用计数指针，而 Weak 是一个指针，但不增加引用计数，是 Rc 的 weak 版。它有以下几个特点：

1.可访问，但不拥有。不增加引用计数，因此，不会对资源回收管理造成影响；

2.可由 Rc<T> 调用 downgrade 方法而转换成 Weak<T>；

3.Weak<T> 可以使用 upgrade 方法转换成 Option<Rc<T>>，如果资源已经被释放，则 Option 值为 None；

4.常用于解决循环引用的问题。

在一个线程中，Rc 和 RcWeak 可以同时存在，例如：

```
{
        let six = Rc::new(vec![1,2,3,4]);
        let six2 = six.clone();
        let six3 = six.clone();
        let six3 = Rc::downgrade(&six3);

        //output
        //[1, 2, 3, 4]
        //(Weak)
        //six3 is (Weak)

        println!("{:?}",six);
        println!("{:?}",six3);

        drop(six);
        drop(six2);

        //no compile error
        println!("six3 is {:?}",six3);
```

```
    }
```

可以看到，即使 six 和 six2 释放，six3 存在，编译时没有仍然没有报错。并且打印结果是 six3 is (Weak)。

6.5.1.2 Arc 与 Arc Weak

首先引出 Arc 前先罗列一个例子。搭配 mutex 进行使用，对共享内存中的内容进行修改。

```
use std::sync::{Mutex, Arc};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];


    for (i,j) in (0..20).enumerate() {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();
            println!("index is {}",i);
            *num += 1;
            println!("index is {} ended",i)
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

输出结果：

```
index is 0
index is 0 ended
index is 1
index is 1 ended
index is 2
index is 2 ended
index is 3
index is 3 ended
index is 5
index is 5 ended
index is 4
index is 4 ended
index is 6
index is 6 ended
index is 8
index is 8 ended
index is 9
index is 9 ended
index is 10
index is 10 ended
index is 12
index is 12 ended
index is 13
index is 13 ended
index is 11
index is 11 ended
index is 14
index is 14 ended
index is 7
index is 7 ended
index is 16
```

```
index is 16 ended
index is 17
index is 17 ended
index is 19
index is 19 ended
index is 18
index is 18 ended
index is 15
index is 15 ended
Result: 20
```

Arc

Arc 是原子引用计数，是 Rc 的多线程版本。Arc 通过 std::sync::Arc 引入。

它的特点：

1.Arc 可跨线程传递，用于跨线程共享一个对象；

2.用 Arc 包裹起来的类型对象，对可变性没有要求，可以搭配 mutex 或者 RWLock 进行修改；

3.一旦最后一个拥有者消失，则资源会被自动回收，这个生命周期是在编译期就确定下来的（如果搭配了锁的使用怎么确定？引入 mutex 可能会造成死锁。）；

4.Arc 实际上是一个指针，它不影响包裹对象的方法调用形式（即不存在先解开包裹再调用值这一说）；

Arc 对于多线程的共享状态几乎是必须的（减少复制，提高性能）

另外一个例子，多线程读：

```rust
use std::sync::Arc;
use std::thread;

fn main() {
    let numbers: Vec<_> = (0..100u32).collect();
    let shared_numbers = Arc::new(numbers);

    for _ in 0..10 {
        let child_numbers = shared_numbers.clone();

        thread::spawn(move || {
            let local_numbers = &child_numbers[..];

            // Work with the local numbers
        });
    }
}
```

Arc Weak

与 Rc 类似，Arc 也有一个对应的 Weak 类型，从 std::sync::Weak 引入。

意义与用法与 Rc Weak 基本一致，不同的点是这是多线程的版本。故不再赘述。

给出一个例子，单线程中如何多个对象同时引用另外一个对象：

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

struct Owner {
    name: String
}

struct Gadget {
    id: i32,
    owner: Rc<Owner>
}

fn main() {
    // Create a reference counted Owner.
    let gadget_owner : Rc<Owner> = Rc::new(
        Owner { name: String::from("Gadget Man") }
    );

    // Create Gadgets belonging to gadget_owner. To increment the
reference
    // count we clone the `Rc<T>` object.
    let gadget1 = Gadget { id: 1, owner: gadget_owner.clone() };
    let gadget2 = Gadget { id: 2, owner: gadget_owner.clone() };

    drop(gadget_owner);

    // Despite dropping gadget_owner, we're still able to print out the
name
    // of the Owner of the Gadgets. This is because we've only dropped
the
    // reference count object, not the Owner it wraps. As long as there
are
    // other `Rc<T>` objects pointing at the same Owner, it will remain
    // allocated. Notice that the `Rc<T>` wrapper around Gadget.owner
gets
    // automatically dereferenced for us.
    println!("Gadget {} owned by {}", gadget1.id, gadget1.owner.name);
    println!("Gadget {} owned by {}", gadget2.id, gadget2.owner.name);

    // At the end of the method, gadget1 and gadget2 get destroyed, and
with
    // them the last counted references to our Owner. Gadget Man now gets
    // destroyed as well.
}
```

6.5.2 Mutex 与 RwLock

6.5.2.1 Mutex

Mutex 意为互斥对象，用来保护共享数据。Mutex 有下面几个特征：

1.Mutex 会等待获取锁令牌(token)，在等待过程中，会阻塞线程。直到锁令牌得到。同时只有一个线程的 Mutex 对象获取到锁；

2.Mutex 通过 .lock() 或 .try_lock() 来尝试得到锁令牌，被保护的对象，必须通过这两个方法返回的 RAII 守卫来调用，不能直接操作；

3.当 RAII 守卫作用域（MutexGuard）结束后，锁会自动解开；

4.在多线程中，Mutex 一般和 Arc 配合使用。

例子如下：

```
use std::sync::mpsc::channel;
use std::sync::{Arc, Mutex, MutexGuard};
use std::thread;

const N: usize = 10;

pub fn main() {
    // Spawn a few threads to increment a shared variable (non-
atomically), and
    // let the main thread know once all increments are done.
    //
    // Here we're using an Arc to share memory among threads, and the
data inside
    // the Arc is protected with a mutex.
    let data = Arc::new(Mutex::new(0));

    let (tx, rx) = channel();
    for _ in 0..10 {
        let (data, tx) = (data.clone(), tx.clone());
        thread::spawn(move || {
            // The shared state can only be accessed once the lock is
held.
            // Our non-atomic increment is safe because we're the only
thread
            // which can access the shared state when the lock is held.
            //
            // We unwrap() the return value to assert that we are not
expecting
            // threads to ever fail while holding the lock.
            let mut data: MutexGuard<usize> = data.lock().unwrap();
            *data += 1;
            if *data == N {
                tx.send(*data).unwrap();
            }
            println!("{}", *data);
            // the lock is unlocked here when `data` goes out of scope.
        });
    }

    let result = rx.recv().unwrap();
    println!("result is {:?}", result);
}
```

mutex 的 lock 与 try_lock 的区别

.lock() 方法，会等待锁令牌，等待的时候，会阻塞当前线程。
而 .try_lock() 方法，只是做一次尝试操作，不会阻塞当前线程。

当 .try_lock() 没有获取到锁令牌时会返回 Err。因此，如果要使用 .try_lock()，需要对返回值做仔细处理（比如，在一个循环检查中）。

6.5.2.2 RwLock

RwLock 翻译成 读写锁。它的特点是：

1.同时允许多个读，最多只能有一个写；

2.读和写不能同时存在；

例子如下：

```
use std::sync::mpsc::channel;
use std::sync::{Arc, Mutex, MutexGuard, RwLock};
use std::thread;

const N: usize = 10;

pub fn main() {
    // Spawn a few threads to increment a shared variable (non-
atomically), and
    // let the main thread know once all increments are done.
    //
    // Here we're using an Arc to share memory among threads, and the
data inside
    // the Arc is protected with a mutex.
    let data = Arc::new(RwLock::new(0));

    let (tx, rx) = channel();
    for _ in 0..10 {
        let (data, tx) = (data.clone(), tx.clone());
        thread::spawn(move || {
            // The shared state can only be accessed once the lock is
held.
            // Our non-atomic increment is safe because we're the only
thread
            // which can access the shared state when the lock is held.
            //
            // We unwrap() the return value to assert that we are not
expecting
            // threads to ever fail while holding the lock.
            let mut data= data.write().unwrap();
            *data += 1;
            if *data == N {
                tx.send(*data).unwrap();
            }
            println!("{}", *data);
            // the lock is unlocked here when `data` goes out of scope.
        });
    }

    let result = rx.recv().unwrap();
    println!("result is {:?}", result);
}
```

所以，如果我们要对一个数据进行安全的多线程使用，最通用的做法就是使用 Arc<Mutex<T>> 或者 Arc<RwLock<T>>进行封装使用。

6.5.3 Cell 与 RefCell

Rust 通过其所有权机制，严格控制拥有和借用关系，来保证程序的安全，并且这种安全是在编译期可计算、可预测的。但是这种严格的控制，有时也会带来灵活性的丧失，有的场景下甚至还满足不了需求。

因此，Rust 标准库中，设计了这样一个系统的组件：Cell, RefCell，它们弥补了 Rust 所有权机制在灵活性上和某些场景下的不足。同时，又没有打破 Rust 的核心设计。它们的出现，使得 Rust 革命性的语言理论设计更加完整，更加实用。

具体是因为，它们提供了 内部可变性（相对于标准的 继承可变性 来讲的）。

通常，我们要修改一个对象，必须

1.成为它的拥有者，并且声明 mut；

2.或 以 &mut 的形式，借用；

而通过 Cell, RefCell，我们可以在需要的时候，就可以修改里面的对象。而不受编译期静态借用规则束缚。

6.5.3.1 Cell

Cell 有如下特点：

1.Cell<T> 只能用于 T 实现了 Copy 的情况；

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

let c = Cell::new(5);

let five = c.get();
```

和

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

let c = Cell::new(5);

c.set(10);
```

6.5.3.2 RefCell

相对于 Cell 只能包裹实现了 Copy 的类型，RefCell 用于更普遍的情况（其它情况都用 RefCell）。

相对于标准情况的 静态借用，RefCell 实现了 运行时借用，这个借用是临时的。这意味着，编译器对 RefCell 中的内容，不会做静态借用检查，也意味着，出了什么问题，用户自己负责。

可能会编译通过而运行时 panic 掉。

RefCell 的特点：

1.在不确定一个对象是否实现了 Copy 时，直接选 RefCell；

2.如果被包裹对象，同时被可变借用了两次，则会导致线程崩溃。所以需要用户自行判断；

3.RefCell 只能用于线程内部，不能跨线程；

4.RefCell 常常与 Rc 配合使用（都是单线程内部使用），搭配 Rc 又有些不同，Rc 本身就是获取同一资源同时有多个所有权拥有者；

例子：

```
use std::collections::HashMap;
use std::cell::RefCell;
use std::rc::Rc;
use std::thread;

fn main() {
    {
        let shared_map: Rc<RefCell<_>> =
Rc::new(RefCell::new(HashMap::new()));
        let shared_map2 = shared_map.clone();
        shared_map.borrow_mut().insert("africa", 92388);
        shared_map.borrow_mut().insert("kyoto", 11837);
        shared_map2.borrow_mut().insert("amy",23456);
        shared_map.borrow_mut().insert("piccadilly", 11826);
        shared_map.borrow_mut().insert("marbles", 38);
        println!("{}",shared_map.borrow().get("amy").unwrap());

        //output:23456
    }

    {
        let shared_map: Rc<RefCell<_>> =
Rc::new(RefCell::new(HashMap::new()));
        shared_map.borrow_mut().insert("africa", 92388);
        shared_map.borrow_mut().insert("kyoto", 11837);
        println!("{}",shared_map.borrow().get("kyoto").unwrap());
        shared_map.borrow_mut().insert("piccadilly", 11826);
        shared_map.borrow_mut().insert("marbles", 38);

        //output:11837
    }

    {

        let result = thread::spawn(move || {
            let c = RefCell::new(5);
            let m = c.borrow_mut();

            let b = c.borrow(); // this causes a panic
        }).join();

        //runtime error
```

```
        //thread '<unnamed>' panicked at 'already mutably borrowed:
BorrowError', src/libcore/result.rs:997:5
    }

}
```

.borrow()

不可变借用被包裹值。同时可存在多个不可变借用。

下面例子会崩溃：

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

let result = thread::spawn(move || {
   let c = RefCell::new(5);
   let m = c.borrow_mut();

   let b = c.borrow(); // this causes a panic
}).join();

assert!(result.is_err());
```

.borrow_mut()

可变借用被包裹值。同时只能有一个可变借用。

下面例子会崩溃：

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

let result = thread::spawn(move || {
   let c = RefCell::new(5);
   let m = c.borrow();

   let b = c.borrow_mut(); // this causes a panic
}).join();

assert!(result.is_err());
```

.into_inner()

取出包裹值。

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

let c = RefCell::new(5);

let five = c.into_inner();
```

6.5.4 综合例子

rust 是一门支持循环引用的语言，例如下面的 owner 和 Gadget 的关系。

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

struct Owner {
    name: String,
    gadgets: RefCell<Vec<Weak<Gadget>>>,
```

```
    // 其他字段
}

struct Gadget {
    id: i32,
    owner: Rc<Owner>,
    // 其他字段
}

fn main() {
    // 创建一个可计数的 Owner。
    // 注意我们将 gadgets 赋给了 Owner。
    // 也就是在这个结构体里， gadget_owner 包含 gadets
    let gadget_owner : Rc<Owner> = Rc::new(
        Owner {
            name: "Gadget Man".to_string(),
            gadgets: RefCell::new(Vec::new()),
        }
    );
    let gadget_onwer2 = gadget_owner.clone();

    // 首先，我们创建两个 gadget，他们分别持有 gadget_owner 的一个引用。
    let gadget1 = Rc::new(Gadget{id: 1, owner: gadget_owner.clone()});
    let gadget2 = Rc::new(Gadget{id: 2, owner: gadget_owner.clone()});

    // 我们将从 gadget_owner 的 gadgets 字段中持有其可变引用
    // 然后将两个 gadget 的 Weak 引用传给 owner。
    gadget_owner.gadgets.borrow_mut().push(Rc::downgrade(&gadget1));
    // gadget_owner2 is the ref of the same resource as rc
    // same pointer
    gadget_onwer2.gadgets.borrow_mut().push(Rc::downgrade(&gadget2));

    // 遍历 gadget_owner 的 gadgets 字段
    for gadget_opt in gadget_owner.gadgets.borrow().iter() {

        // gadget_opt 是一个 Weak<Gadget> 。 因为 weak 指针不能保证他所引用的对象
        // 仍然存在。所以我们需要显式的调用 upgrade() 来通过其返回值(Option<_>)来判
        // 断其所指向的对象是否存在。
        // 当然，这个 Option 为 None 的时候这个引用原对象就不存在了。
        let gadget = gadget_opt.upgrade().unwrap();
        println!("Gadget {} owned by {}", gadget.id, gadget.owner.name);
    }

    // 在 main 函数的最后， gadget_owner, gadget1 和 daget2 都被销毁。
    // 具体是，因为这几个结构体之间没有了强引用（`Rc<T>`），所以，当他们销毁的时候。
    // 首先 gadget1 和 gadget2 被销毁。
    // 然后因为 gadget_owner 的引用数量为 0，所以这个对象可以被销毁了。
    // 循环引用问题也就避免了
}
```

6.6 类型系统中常见的 trait

本章讲解 Rust 类型系统中的几个常见 trait。有 Into, From, AsRef, AsMut, Borrow, BorrowMut, ToOwned, Deref, Cow。

其中 Into，From，Cow 不是很常用，简单说明略过。Cow 是借鉴 linux 系统中的写时复制。

6.6.1 From，Into，Cow

From:

对于类型为 U 的对象 foo，如果它实现了 From<T>，那么，可以通过 let foo = U::from(bar) 来生成自己。这里，bar 是类型为 T 的对象。

Into:

对于一个类型为 U: Into<T> 的对象 foo，Into 提供了一个函数：.into(self) -> T，调用 foo.into() 会消耗自己（转移资源所有权），生成类型为 T 的另一个新对象 bar。

例子：

```
struct Person {
    name: String,
}

impl Person {
    fn new<S: Into<String>>(name: S) -> Person {
        Person { name: name.into() }
    }
}
fn main() {
    let person = Person::new("Herman");
    let person = Person::new("Herman".to_string());
}
```

Into<String>使传参又能够接受 String 类型，又能够接受 &str 类型。

标准库中，提供了 Into<T> 来为其做约束，以便方便而高效地达到我们的目的。

参数类型为 S， 是一个泛型参数，表示可以接受不同的类型。S: Into<String> 表示 S 类型必须实现了 Into<String>（约束）。而 &str 类型，符合这个要求。因此 &str 类型可以直接传进来。

而 String 本身也是实现了 Into<String> 的。当然也可以直接传进来。

下面 name: name.into() 这里也挺神秘的。它的作用是将 name 转换成 String 类型的另一个对象。当 name 是 &str 时，它会转换成 String 对象，会做一次字符串的拷贝（内存的申请、复制）。而当 name 本身是 String 类型时，name.into() 不会做任何转换，代价为零。

Cow: Cow 的设计目的是提高性能（减少复制）同时增加灵活性，因为大部分情况下，业务场景都是读多写少。利用 Cow，可以用统一，规范的形式实现，需要写的时候才做一次对象复制。这样就可能会大大减少复制的次数。

6.6.2 AsRef，AsMut

AsRef 提供了一个方法 .as_ref()。

对于一个类型为 T 的对象 foo，如果 T 实现了 AsRef<U>，那么，foo 可执行 .as_ref() 操作，即 foo.as_ref()。操作的结果，我们得到了一个类型为 &U 的新引用。

1.与 Into<T> 不同的是，AsRef<T> 只是类型转换，foo 对象本身没有被消耗；

2.T: AsRef<U> 中的 T，可以接受 资源拥有者（owned）类型，共享引用（shared referrence）类型 ，可变引用（mutable referrence）类型。

例子如下：

```
fn is_hello<T: AsRef<str>>(s: T) {
   assert_eq!("hello", s.as_ref());
}

let s = "hello";
is_hello(s);

let s = "hello".to_string();
is_hello(s);
```

因为 String 和 &str 都实现了 AsRef<str>。

AsMut：

AsMut<T> 提供了一个方法 .as_mut()。它是 AsRef<T> 的可变（mutable）引用版本。

对于一个类型为 T 的对象 foo，如果 T 实现了 AsMut<U>，那么，foo 可执行 .as_mut() 操作，即 foo.as_mut()。操作的结果，我们得到了一个类型为 &mut U 的可变（mutable）引用。

注：在转换的过程中，foo 会被可变（mutable）借用。

6.6.3 Borrow，BorrowMut，ToOwned

6.6.3.1 Borrow

Borrow 提供了一个方法 .borrow()。

对于一个类型为 T 的值 foo，如果 T 实现了 Borrow<U>，那么，foo 可执行 .borrow() 操作，即 foo.borrow()。操作的结果，我们得到了一个类型为 &U 的新引用。

Borrow 可以认为是 AsRef 的严格版本，它对普适引用操作的前后类型之间附加了一些其它限制。

Borrow 的前后类型之间要求必须有内部等价性。不具有这个等价性的两个类型之间，不能实现 Borrow。

AsRef 更通用，更普遍，覆盖类型更多，是 Borrow 的超集。

6.6.3.2 BorrowMut

BorrowMut<T> 提供了一个方法 .borrow_mut()。它是 Borrow<T> 的可变（mutable）引用版本。

对于一个类型为 T 的值 foo，如果 T 实现了 BorrowMut<U>，那么，foo 可执行 .borrow_mut() 操作，即 foo.borrow_mut()。操作的结果我们得到类型为 &mut U 的一个可变（mutable）引用。

注：在转换的过程中，foo 会被可变（mutable）借用。

6.6.3.3 ToOwned

ToOwned 为 Clone 的普适版本。它提供了 .to_owned() 方法，用于类型转换。

有些实现了 Clone 的类型 T 可以从引用状态实例 &T 通过 .clone() 方法，生成具有所有权的 T 的实例。但是它只能由 &T 生成 T。而对于其它形式的引用，Clone 就无能为力了。

而 ToOwned trait 能够从任意引用类型实例，生成具有所有权的类型实例。

6.6.4 Deref

Deref 是 deref 操作符 * 的 trait，比如 *v。

一般理解，*v 操作，是 &v 的反向操作，即试图由资源的引用获取到资源的拷贝（如果资源类型实现了 Copy），或所有权（资源类型没有实现 Copy）。

6.6.4.1 强制解引

这种隐式转换的规则为：

一个类型为 T 的对象 foo，如果 T: Deref<Target=U>，那么，相关 foo 的某个智能指针或引用（比如 &foo）在应用的时候会自动转换成 &U。

Rust 编译器会在做 *v 操作的时候，自动先把 v 做引用归一化操作，即转换成内部通用引用的形式 &v，整个表达式就变成 *&v。这里面有两种情况：

1.把其它类型的指针（比如在库中定义的，Box, Rc, Arc, Cow 等），转成内部标准形式 &v；

2.把多重 & （比如：&&&&&&&v），简化成 &v（通过插入足够数量的 * 进行解引）。

简单了解一下即可。

第七章 多线程与线程通信

从结论上来说，rust 的编译器并不能防止所有的线程引起的问题，例如：

```
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();// mpsc 是多个发送者，一个接收者

    thread::spawn(move || {
        let val = String::from("hi");
//        tx.send(val).unwrap();
    });

    let received:String = rx.recv().unwrap(); // 阻塞等待，直到接收到一个消息
    println!("Got: {}", received);
}
```

例如上述代码编译通过，但是在运行过程中会死锁。而 rust 的编译器无法识别此类错误。Rust 对安全的理解是死锁不属于内存安全问题。

Rust 在并发上的改进是：

通过改进所有权和类型检查，Rust 很多并发错误都是 **编译时** 错误，而非运行时错误。

更强大的编译器和类型检查排除了很多并发问题，但并不能解决所有并发问题。所以在进行并发编程时仍然要小心谨慎。

7.1 线程

7.1.1 不同语言的线程实现

在大部分现代操作系统中，执行中程序的代码运行于一个 进程（process）中，操作系统则负责管理多个进程。在程序内部，也可以拥有多个同时运行的独立部分。这个运行这些独立部分的功能被称为 线程（threads）。

编程语言有一些不同的方法来实现线程。很多操作系统提供了创建新线程的 API。这种由编程语言调用操作系统 API 创建线程的模模型有时被称为 1:1，一个 OS 线程对应一个语言线程。

很多编程语言提供了自己特殊的线程实现。编程语言提供的线程被称为 绿色（green）线程，使用绿色线程的语言会在不同数量的 OS 线程的上下文中执行它们。为此，绿色线程模式被称为 M:N 模型：M 个绿色线程对应 N 个 OS 线程，这里 M 和 N 不必相同。

在当前上下文中，运行时 代表二进制文件中包含的由语言自身提供的代码。这些代码根据语言的不同可大可小，不过任何非汇编语言都会有一定数量的运行时代码。为此，通常人们说一个语言 “没有运行时”，一般意味着 “小运行时”。更小的运行时拥有更少的功能不过其优势在于更小的二进制输出，这使其易于在更多上下文中与其他语言相结合。虽然很多语言觉得增加运行时来

换取更多功能没有什么问题，但是 Rust 需要做到几乎没有运行时，同时为了保持高性能必需能够调用 C 语言，这点也是不能妥协的。

绿色线程的 M:N 模型更大的语言运行时来管理这些线程。为此，Rust 标准库只提供了 1:1 线程模型实现。Rust 是足够底层的语言，所以有相应的 crate 实现了 M:N 线程模型，如果你宁愿牺牲性能来换取例如更好的线程运行控制和更低的上下文切换成本。

7.1.2 使用 spawn 创建新线程

为了创建一个新线程，需要调用 thread::spawn 函数并传递一个闭包（第十三章学习了闭包），其包含希望在新线程运行的代码。

```
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

运行结果：

```
hi number 1 from the main thread!
hi number 1 from the spawned thread!
hi number 2 from the main thread!
hi number 2 from the spawned thread!
hi number 3 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the main thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
```

由于主线程结束，上面例子中的代码大部分时候不光会提早结束新建线程，甚至不能实际保证新建线程会被执行。其原因在于无法保证线程运行的顺序！

7.1.3 使用 join 等待所有线程结束

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap();
```

```
}
```

通过调用 handle 的 join 会阻塞当前线程直到 handle 所代表的线程结束。阻塞（Blocking） 线程意味着阻止该线程执行工作或退出。因为我们将 join 调用放在了主线程的 for 循环之后，运行示例 16-2 应该会产生类似这样的输出：

```
hi number 1 from the main thread!
hi number 2 from the main thread!
hi number 1 from the spawned thread!
hi number 3 from the main thread!
hi number 2 from the spawned thread!
hi number 4 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!
```

这两个线程仍然会交替执行，不过主线程会由于 handle.join() 调用会等待直到新建线程执行完毕。

7.1.4 线程与 move 闭包

thread::spawn 的闭包并没有任何参数：并没有在新建线程代码中使用任何主线程的数据。为了在新建线程中使用来自于主线程的数据，需要新建线程的闭包获取它需要的值。

```
use std::thread;

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

    let handle = thread::spawn(move || {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

线程中使用的闭包只能是 move 修饰。

7.2 消息传递

Rust 的通道(channel)可以把一个线程的消息(数据)传递到另一个线程，从而让信息在不同的线程中流动，从而实现协作。详情请参见 std::sync::mpsc。通道的两端分别是发送者(Sender)和接收者(Receiver)，发送者负责从一个线程发送消息，接收者则在另一个线程中接收该消息。

简单的例子：

```
use std::sync::mpsc;
use std::thread;

fn main() {
    // 创建一个通道
    let (tx, rx): (mpsc::Sender<i32>, mpsc::Receiver<i32>) =
        mpsc::channel();
```

```
    // 创建线程用于发送消息
    thread::spawn(move || {
        // 发送一个消息，此处是数字 id
        tx.send(1).unwrap();
    });

    // 在主线程中接收子线程发送的消息并输出
    println!("receive {}", rx.recv().unwrap());
}
```

运行结果：

```
receive 1
```

结果表明 main 所在的主线程接收到了新建线程发送的消息，用 Rust 在线程间传递消息就是这么简单！

虽然简单，但使用过其他语言就会知道，通道有多种使用方式，且比较灵活，为此我们需要进一步考虑关于 Rust 的 Channel 的几个问题：

1. 通道能保证消息的顺序吗？是否先发送的消息，先接收？
2. 通道能缓存消息吗？如果能的话能缓存多少？
3. 通道的发送者和接收者支持 N:1，1:N，N:M 模式吗？
4. 通道能发送任何数据吗？
5. 发送后的数据，在线程中继续使用没有问题吗？

下面各小节会对上面各部分问题进行回答。

7.2.1 通道与所有权的转移

在并发编程中避免错误是在整个 Rust 程序中必须思考所有权所换来的一大优势。

现在让我们做一个试验来看看通道与所有权如何一同协作以避免产生问题：我们将尝试在新建线程中的通道中发送完 val 值 之后 再使用它。

```
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
        println!("val is {}", val);
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

这里尝试在通过 tx.send 发送 val 到通道中之后将其打印出来。允许这么做是一个坏主意：一旦将值发送到另一个线程后，那个线程可能会在我们再次使用它之前就将其修改或者丢弃。其他线程对值可能的修改会由于不一致或不存在的数据而导致错误或意外的结果。

编译报错：

```
error[E0382]: use of moved value: `val`
  --> src/main.rs:10:31
   |
9  |         tx.send(val).unwrap();
   |                 --- value moved here
10 |         println!("val is {}", val);
   |                               ^^^ value used here after move
   |
   = note: move occurs because `val` has type `std::string::String`,
which does
not implement the `Copy` trait
```

7.2.2 通道保证发送的顺序

```
use std::thread;
use std::sync::mpsc;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let vals = vec![
            String::from("hi"),
            String::from("from"),
            String::from("the"),
            String::from("thread"),
        ];

        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    for received in rx {
        println!("Got: {}", received);
    }
}
```

这一次，在新建线程中有一个字符串 vector 希望发送到主线程。我们遍历他们，单独的发送每一个字符串并通过一个 Duration 值调用 thread::sleep 函数来暂停一秒。

在主线程中，不再显式调用 recv 函数：而是将 rx 当作一个迭代器。对于每一个接收到的值，我们将其打印出来。当通道被关闭时，迭代器也将结束。

```
打印结果：
Got: hi
Got: from
```

```
Got: the
Got: thread
```

7.2.3 通过克隆发送者来创建多个生产者

之前我们提到了 mpsc 是 multiple producer, single consumer 的缩写。可以运用 mpsc 以创建都向同一接收者发送值的多个线程。这可以通过克隆通道的发送端在来做到。

```
let (tx, rx) = mpsc::channel();

let tx1 = mpsc::Sender::clone(&tx);
thread::spawn(move || {
    let vals = vec![
        String::from("hi"),
        String::from("from"),
        String::from("the"),
        String::from("thread"),
    ];

    for val in vals {
        tx1.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
});

thread::spawn(move || {
    let vals = vec![
        String::from("more"),
        String::from("messages"),
        String::from("for"),
        String::from("you"),
    ];

    for val in vals {
        tx.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
});

for received in rx {
    println!("Got: {}", received);
}
```

这一次，在创建新线程之前，我们对通道的发送端调用了 clone 方法。这会给我们一个可以传递给第一个新建线程的发送端句柄。我们会将原始的通道发送端传递给第二个新建线程。这样就会有两个线程，每个线程将向通道的接收端发送不同的消息。

如果运行这些代码，你可能会看到这样的输出（两次的运行结果）：

```
Got: hi
Got: more
Got: from
```

```
Got: messages
Got: for
Got: the
Got: thread
Got: you
```

和

```
Got: hi
Got: more
Got: from
Got: messages
Got: the
Got: for
Got: you
Got: thread
```

虽然你可能会看到这些值以不同的顺序出现。在并发下，运行结果可能每次都不相同。以进入到 channel 的顺序为主。

7.2.4 异步通道与同步通道

Rust 的标准库其实提供了两种类型的通道：异步通道和同步通道。在前面使用的都是异步通道。异步通道指的是：不管接收者是否正在接收消息，消息发送者在发送消息时都不会阻塞。为了验证这一点，我们尝试多增加个线程来发送消息：

```rust
use std::sync::mpsc;
use std::thread;

// 线程数量
const THREAD_COUNT :i32 = 6;

fn main() {
    // 创建一个通道
    let (tx, rx): (mpsc::Sender<i32>, mpsc::Receiver<i32>) = mpsc::channel();

    // 创建线程用于发送消息
    for id in 0..THREAD_COUNT {
        // 注意 Sender 是可以 clone 的，这样就可以支持多个发送者
        let thread_tx = tx.clone();
        thread::spawn(move || {
            // 发送一个消息，此处是数字 id
            thread_tx.send(id + 1).unwrap();
            println!("send {}", id + 1);
        });
    }

    thread::sleep_ms(2000);
    println!("wake up");
    // 在主线程中接收子线程发送的消息并输出
    for _ in 0..THREAD_COUNT {
        println!("receive {}", rx.recv().unwrap());
    }
}
```

返回结果：

```
send 1
send 2
wake up
receive 1
receive 2
```

在代码中，我们故意让 main 所在的主线程睡眠 2 秒，从而让发送者所在线程优先执行，通过结果可以发现，发送者发送消息时确实没有阻塞。

异步通道具备消息缓存的功能，理论上是无穷的，直至内存耗光为止。

异步通道的具有良好的灵活性和扩展性，针对业务需要，可以灵活地应用于实际项目中。

同步通道：

同步通道在使用上同异步通道一样，接收端也是一样的，唯一的区别在于发送端，唯一不同的在于创建同步通道的那行代码。同步通道是 sync_channel，对应的发送者也变成了 SyncSender。为了显示出同步通道的区别，故意添加了一些打印。和异步通道相比，存在两点不同：

1. 同步通道是需要指定缓存的消息个数的，但需要注意的是，最小可以是 0，表示没有缓存。

2.发送者是会被阻塞的。当通道的缓存队列不能再缓存消息时，发送者发送消息时，就会被阻塞。当缓存队列有空间存放时，会从阻塞状态唤醒。

```
use std::sync::mpsc;
use std::thread;

fn main() {
    // 创建一个同步通道
    let (tx, rx): (mpsc::SyncSender<i32>, mpsc::Receiver<i32>) = mpsc::sync_channel(1);

    let tx1 = tx.clone();
    // 创建线程用于发送消息
    let new_thread = thread::spawn(move || {
        // 发送一个消息，此处是数字 id
        println!("before send");
        tx.send(1).unwrap();
        println!("after send");
    });

    let thread_two = thread::spawn(move || {
        // 发送一个消息，此处是数字 id
        println!("before send two");
        tx1.send(2).unwrap();
        println!("after send two");
    });

    println!("before sleep");
    thread::sleep_ms(5000);
    println!("after sleep");
    // 在主线程中接收子线程发送的消息并输出
    println!("receive {}", rx.recv().unwrap());
    println!("receive {}", rx.recv().unwrap());
    thread_two.join().unwrap();
}
```

7.2.5 可发送的消息类型

之前例子中我们传递的消息类型大部分为 i32，除了这种类型之外，是否还可以传递更多的原始类型，或者更复杂的类型，和自定义类型？下面我们尝试发送一个更复杂的 Rc 类型的消息：

```
use std::fmt;
use std::sync::mpsc;
use std::thread;
use std::rc::Rc;

pub struct Student {
    id: u32
}

impl fmt::Display for Student {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "student {}", self.id)
    }
}

fn main() {
    // 创建一个通道
    let (tx, rx): (mpsc::Sender<Rc<Student>>, mpsc::Receiver<Rc<Student>>) =
        mpsc::channel();

    // 创建线程用于发送消息
    thread::spawn(move || {
        // 发送一个消息，此处是数字 id
        tx.send(Rc::new(Student{
            id: 1,
        })).unwrap();
    });

    // 在主线程中接收子线程发送的消息并输出
    println!("receive {}", rx.recv().unwrap());
}
```

编译报错：

```
error: the trait `core::marker::Send` is not
implemented for the type `alloc::rc::Rc<Student>` [E0277]
note: `alloc::rc::Rc<Student>` cannot be sent between threads safely
```

将 Rc 改成 Arc，代码如下：

```
use std::fmt;
use std::sync::{mpsc, Arc};
use std::thread;
use std::rc::Rc;

pub struct Student {
    id: u32
}

impl fmt::Display for Student {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "student {}", self.id)
    }
}

fn main() {
    // 创建一个通道
    let (tx, rx): (mpsc::Sender<Arc<Student>>, mpsc::Receiver<Arc<Student>>) =
        mpsc::channel();

    // 创建线程用于发送消息
    thread::spawn(move || {
        // 发送一个消息，此处是数字 id
        tx.send(Arc::new(Student{
```

```
            id: 1,
        })).unwrap();
    });

    // 在主线程中接收子线程发送的消息并输出
    println!("receive {}", rx.recv().unwrap());
}
```

消息类型必须实现 marker trait Send。Rust 之所以这样强制要求，主要是为了解决并发安全的问题，再一次强调，安全是 Rust 考虑的重中之重。如果一个类型是 Send，则表明它可以在线程间安全的转移所有权(ownership)，当所有权从一个线程转移到另一个线程后，同一时间就只会存在一个线程能访问它，这样就避免了数据竞争，从而做到线程安全。

在上面两个例子中，Rc 这一类型的智能指针没有实现 trait Send，而 Arc 实现了。关于 send 与 sync 在下一节详细说明。

7.3 send 与 sync

7.3.1 send

Send 标记 trait 表明类型的所有权可以在线程间传递。几乎所有的 Rust 类型都是 Send 的，不过有一些例外，包括 Rc<T>：这是不能 Send 的，因为如果克隆了 Rc<T> 的值并尝试将克隆的所有权转移到另一个线程，这两个线程都可能同时更新引用计数。为此，Rc<T> 被实现为用于单线程场景，这时不需要为拥有线程安全的引用计数而付出性能代价。

因此，Rust 类型系统和 trait bound 确保永远也不会意外的将不安全的 Rc<T> 在线程间发送。当尝试这么做的时候，会得到错误 the trait Send is not implemented for Rc<Mutex<i32>>。而使用标记为 Send 的 Arc<T> 时，就没有问题了。

任何完全由 Send 的类型组成的类型也会自动被标记为 Send。几乎所有基本类型都是 Send 的，

7.3.2 sync

Sync 标记 trait 表明一个实现了 Sync 的类型可以安全的在多个线程中拥有其值的引用。换一种方式来说，对于任意类型 T，如果 &T（T 的引用）是 Send 的话 T 就是 Sync 的，这意味着其引用就可以安全的发送到另一个线程。类似于 Send 的情况，基本类型是 Sync 的，完全由 Sync 的类型组成的类型也是 Sync 的。

智能指针 Rc<T> 也不是 Sync 的，出于其不是 Send 相同的原因。RefCell<T>和 Cell<T> 系列类型不是 Sync 的。RefCell<T> 在运行时所进行的借用检查也不是线程安全的。Mutex<T> 是 Sync 的，正如 “在线程间共享 Mutex<T>” 部分所讲的它可以被用来在多线程中共享访问。

7.3.3 手动实现 send 和 sync 需要加上 unsafe

通常并不需要手动实现 Send 和 Sync trait，因为由 Send 和 Sync 的类型组成的类型，自动就是 Send 和 Sync 的。因为他们是标记 trait，甚至都不需要实现任何方法。他们只是用来加强并发相关的不可变性的。

7.4 共享内存

go 中虽然提供了 channel 方式进行通信，但同样提供了共享内存的方式。主要原因在于所有的数据交互通过 channel 方式通信过于复杂。在 rust 中同样提供了共享内存的方式和相应的锁等机制。

在 rust 中，共享内存主要有两种方式：

static 和堆。

7.4.1 static

Rust 语言中也存在 static 变量，其生命周期是整个应用程序，并且在内存中某个固定地址处只存在一份实例。所有线程都能够访问到它。这种方式也是最简单和直接的共享方式。

```
use std::thread;

static mut VAR: i32 = 5;

fn main() {
    // 创建一个新线程
    let new_thread = thread::spawn(move|| {
        unsafe {
            println!("static value in new thread: {}", VAR);
            VAR = VAR + 1;
        }
    });

    // 等待新线程先运行
    new_thread.join().unwrap();

    // compile error
    //error[E0133]: use of mutable static is unsafe and requires unsafe function or
block
    //        --> src/main.rs:16:49
    //        |
    //        16 |     println!("static value in main thread: {}", VAR);
    //    |                                                 ^^^ use of mutable static

    // println!("static value in main thread: {}", VAR);

    unsafe {
        println!("static value in main thread: {}", VAR);
    }

}
```

运行结果：

```
static value in new thread: 5
static value in main thread: 6
```

从结果来看 VAR 的值变了，从代码上来看，除了在 VAR 变量前面加了 mut 关键字外，更加明显的是在使用 VAR 的地方都添加了 unsafe 代码块。为什么？所有的线程都能访问 VAR，且它是可以被修改的，自然就是不安全的。上面的代码比较简单，同一时间只会有一个线程读写 VAR，不会有什么问题，所以用 unsafe 来标记就可以。如果是更多的线程，还是请使用接下来要介绍的同步机制来处理。

static 如此，那 const 呢？ const 会在编译时内联到代码中，所以不会存在某个固定的内存地址上，也不存在可以修改的情况，并不是内存共享的。

7.4.2 堆

由于现代操作系统的设计，线程寄生于进程，可以共享进程的资源，如果要在各个线程中共享一个变量，那么除了上面的 static，还有就是把变量保存在堆上了。

基于 Arc 创建资源就是使用了堆，可以从 Arc::New 查看到。

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

fn main() {
    let var : Arc<i32> = Arc::new(5);
    let share_var = var.clone();

    // 创建一个新线程
    let new_thread = thread::spawn(move|| {
        println!("share value in new thread: {}, address: {:p}", share_var,
&*share_var);
    });

    // 等待新建线程先执行
    new_thread.join().unwrap();
    println!("share value in main thread: {}, address: {:p}", var, &*var);
}
```

运行结果：

```
share value in new thread: 5, address: 0x2825070
share value in main thread: 5, address: 0x2825070
```

可以看出，它们的内存地址相同。

7.5 同步

如果是要在多个线程中使用，就需要面临两个关键问题：

1. 资源何时释放？

2. 线程如何安全的并发修改和读取？

由于上面两个问题的存在，这就是为什么我们不能直接用 Box 变量在线程中共享的原因，可以看出来，共享内存比消息传递机制似乎要复杂许多。Rust 用了引用计数的方式来解决第一个问题（即引入 Arc。）。

关于上面的第二个问题，Rust 语言及标准库提供了一系列的同步手段来解决。

同步指的是线程之间的协作配合，以共同完成某个任务。在整个过程中，需要注意两个关键点：一是共享资源的访问， 二是访问资源的顺序。

在前面的章节中描述了如何访问共享资源，在本节中重点说明访问资源的顺序。

7.5.1 控制访问顺序--等待与通知

等待有三种：

1.主动等待一段时间

2.主动放弃一段时间片

3.被动等待，被动唤醒

通知：

看是简单的通知，在编程时也需要注意以下几点：

1.通知必然是因为有等待，所以通知和等待几乎都是成对出现的，比如 std::sync::Condvar::wait 和 std::sync::Condvar::notify_one，std::sync::Condvar::notify_all。

2.等待所使用的对象，与通知使用的对象是同一个对象，从而该对象需要在多个线程之间共享，参见下面的例子。

3.除了 Condvar 之外，其实锁也是具有自动通知功能的，当持有锁的线程释放锁的时候，等待锁的线程就会自动被唤醒，以抢占锁。关于锁的介绍，在下面有详解。

4.通过条件变量和锁，还可以构建更加复杂的自动通知方式，比如 std::sync::Barrier。

5.通知也可以是 1:1 的，也可以是 1:N 的，Condvar 可以控制通知一个还是 N 个，而锁则不能控制，只要释放锁，所有等待锁的其他线程都会同时醒来，而不是只有最先等待的线程。

例子：

```
use std::sync::{Arc, Mutex, Condvar};
use std::thread;

fn main() {

    let pair = Arc::new((Mutex::new(false), Condvar::new()));
    let pair2 = pair.clone();

    // 创建一个新线程
    thread::spawn(move|| {
        let &(ref lock, ref cvar) = &*pair2;
        let mut started = lock.lock().unwrap();
        *started = true;
        cvar.notify_one();
        println!("notify main thread");
    });
```

```
    // 等待新线程先运行
    let &(ref lock, ref cvar) = &*pair;
    let mut started = lock.lock().unwrap();
    while !*started {
        println!("before wait");
        started = cvar.wait(started).unwrap();
        println!("after wait");
    }
}
```

输出结果：

```
before wait
notify main thread
after wait
```

这个例子展示了如何通过条件变量和锁来控制新建线程和主线程的同步，让主线程等待新建线程执行后，才能继续执行。从结果来看，功能上是实现了。对于上面这个例子，还有下面几点需要说明：

1. Mutex 是 Rust 中的一种锁。

2.Condvar 需要和 Mutex 一同使用，因为有 Mutex 保护，Condvar 并发才是安全的。

3.Mutex::lock 方法返回的是一个 MutexGuard，在离开作用域的时候，自动销毁，从而自动释放锁，从而避免锁没有释放的问题。

4.Condvar 在等待时，会释放锁的，被通知唤醒时，会重新获得锁，从而保证并发安全。

7.5.2 控制访问顺序的机制-原子类型与锁

7.5.2.1 原子类型锁

原子类型是最简单的控制共享资源访问的一种机制，相比较于后面将介绍的锁而言，原子类型不需要开发者处理加锁和释放锁的问题，同时支持修改，读取等操作，还具备较高的并发性能，从硬件到操作系统，到各个语言，基本都支持。在标准库 std::sync::atomic 中，你将在里面看到 Rust 现有的原子类型，包括 AtomicBool，AtomicIsize，AtomicPtr，AtomicUsize。这 4 个原子类型基本能满足百分之九十的共享资源安全访问的需要。下面我们就用原子类型，结合共享内存的知识，来展示一下一个线程修改，一个线程读取的情况：

```
use std::thread;
use std::sync::Arc;
use std::sync::atomic::{AtomicUsize, Ordering};

fn main() {
    let var : Arc<AtomicUsize> = Arc::new(AtomicUsize::new(5));
    let share_var = var.clone();

    // 创建一个新线程
    let new_thread = thread::spawn(move|| {
        println!("share value in new thread: {}",
share_var.load(Ordering::SeqCst));
        // 修改值
```

```
        share_var.store(9, Ordering::SeqCst);
    });

    // 等待新建线程先执行
    new_thread.join().unwrap();
    println!("share value in main thread: {}",
var.load(Ordering::SeqCst));
}
```

输出结果：

```
share value in new thread: 5
share value in main thread: 9
```

7.5.2.1 Mutex

为了保障锁使用的安全性问题，Rust 做了很多工作，但从效率来看还不如原子类型，那么锁是否就没有存在的价值了？显然事实不可能是这样的，既然存在，那必然有其价值。它能解决原子类型锁不能解决的那百分之十的问题。

```
use std::sync::{Arc, Mutex, Condvar};
use std::thread;

fn main() {

    let pair = Arc::new((Mutex::new(false), Condvar::new()));
    let pair2 = pair.clone();

    // 创建一个新线程
    thread::spawn(move|| {
        let &(ref lock, ref cvar) = &*pair2;
        let mut started = lock.lock().unwrap();
        *started = true;
        cvar.notify_one();
        println!("notify main thread");
    });

    // 等待新线程先运行
    let &(ref lock, ref cvar) = &*pair;
    let mut started = lock.lock().unwrap();
    while !*started {
        println!("before wait");
        started = cvar.wait(started).unwrap();
        println!("after wait");
    }
}
```

代码中的 Condvar 就是条件变量，它提供了 wait 方法可以主动让当前线程等待，同时提供了 notify_one 方法，让其他线程唤醒正在等待的线程。这样就能完美实现顺序控制了。看起来好像条件变量把事都做完了，要 Mutex 干嘛呢？为了防止多个线程同时执行条件变量的 wait 操作，因为条件变量本身也是需要被保护的，这就是锁能做，而原子类型做不到的地方。

在 Rust 中，Mutex 是一种独占锁，同一时间只有一个线程能持有这个锁。这种锁会导致所有线程串行起来，这样虽然保证了安全，但效率并不高。对于写少读多的情况来说，如果在没有写的情况下，都是读取，那么应该是可以并发执行的，为了达到这个目的，几乎所有的编程语言都提供了一种叫读写锁的机制，Rust 中也存在，叫 std::sync::RwLock。

7.6 并行

在 rust 中，并行借助第三方库实现 rayon。rayon 尽可能封装，但并行本身就比较复杂，这节略去不说。

目前的应用开发上应该很少会涉及。

7.7 总结

和像 Mutex<T> 和 Arc<T> 这样可以安全的用于并发上下文的智能指针。类型系统和借用检查器会确保这些场景中的代码，不会出现数据竞争和无效的引用。

所有权和生命周期 + Send 和 Sync（本质上为类型系统，向多线程发送的过程中首先进行检查，保证其在编译环境下多线程情况下绝对安全。）来为并发编程提供了安全可靠的基础设施。使得程序员可以放心在其上构建稳健的并发模型。

一旦代码可以编译了（除了 RefCell 和 unsafe 模块），我们就可以坚信这些代码可以正确的运行于多线程环境，而不会出现其他语言中经常出现的那些难以追踪的 bug。并发编程不再是什么可怕的概念：无所畏惧地并发吧！

无畏并发并不是保证没有 bug，代码有问题还是会出现死锁。

第八章 Rust 性能优化

目前中文关于 rust 性能优化的文档少之又少，github 上有一些可以值得参考借鉴的材料，

https://gist.github.com/jFransham/5c19171f898ca3e33eadb30bbb5e4fd6

https://gist.github.com/jFransham/369a86eff00e5f280ed25121454acec1

第九章 测试与评测

在 rust 中内部构建了测试和评测模块，虽然目前 bench 模块仍然在 nightly channel。

需要手动将 rust 切换到 nightly 版本，通过以下命令：

```
rustup default nightly
```

作为软件工程质量保障体系的重要一环，测试是应该引起我们充分注意并重视的事情。前面说过，Rust 语言的设计集成了最近十多年中总结出来的大量最佳工程实践，而对测试的原生集成也正体现了这一点。很大程度借鉴了 golang 的部分内容。

Rust 的测试特性按精细度划分，分为 3 个层次：

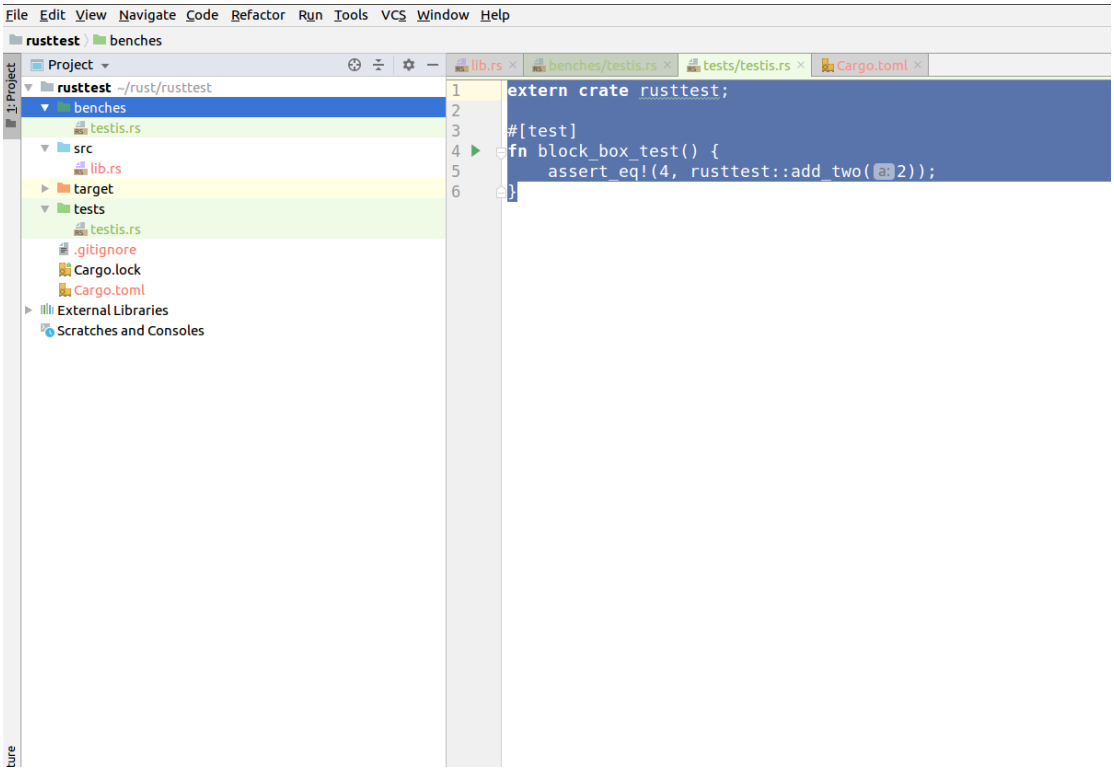
1. 函数级：主要通过#[test] 标识

2. 模块级：主要通过#[cfg(test)]标志。

3. 工程级：例如黑盒测试，放于 test 目录下。

另外，Rust 还支持对文档进行测试。

一个项目中路径如下：



之前提到过的 Cargo.toml 补充：

cargo.toml 和 cargo.lock 文件总是位于项目根目录下。

源代码位于 src 目录下。

默认的库入口文件是 src/lib.rs。

默认的可执行程序入口文件是 src/main.rs。

其他可选的可执行文件位于 src/bin/*.rs(这里每一个 rs 文件均对应一个可执行文件)。

外部测试源代码文件位于 tests 目录下。

示例程序源代码文件位于 examples。

基准测试源代码文件位于 benches 目录下。

9.1 函数级测试

当我们创建一个空的库项目时，打开 src/lib.rs 文件，可以看到如下代码：

```
#[test]
fn it_works() {
    // do test work
}
```

Rust 中，只需要在一个函数的上面，加上 #[test] 就标明这是一个测试用的函数。

有了这个属性之后，在使用 cargo build 编译时，就会忽略这些函数。使用 cargo test 可以运行这些函数。类似于如下效果：

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

此外，可以使用的属性还有：

#[ignore]

#[should_panic]

9.2 模块级测试

有时，我们会组织一批测试用例，这时，模块化的组织结构就有助于建立结构性的测试体系。Rust 中，可以类似如下写法：

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

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

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
    }
}
```

也即在 mod 的上面写上 #[cfg(test)] ，表明这个模块是个测试模块。一个测试模块中，可以包含若干测试函数，测试模块中还可以继续包含测试模块，即模块的嵌套。

如此，就形式了结构化的测试体系，甚是方便。

9.3 工程级测试（黑盒集成测试）

函数级和模块级的测试，代码是与要测试的模块（编译单元）写在相同的文件中，一般做的是白盒测试。工程的测试，一般做的就是黑盒集成测试了。

我们看上图截图工程的目录，在这个目录下，有一个 tests 文件夹。

```
extern crate rusttest;
#[test]
fn block_box_test() {
    assert_eq!(4, rusttest::add_two(2));
}
```

这里，比如，我们 src 中，写了一个库，提供了一个 add_two 函数，现在进行集成测试。

首先，用 extern crate 的方式，引入这个库，由于是同一个项目，cargo 会自动找。引入后，就按模块的使用方法调用就行了，其它的测试标识与前面相同。

9.4 基准测试

单元测试是用来校验程序的正确性的，然而，程序能正常运行后，往往还需要测试程序（一部分）的执行速度，这时，就需要用到性能测试。 通常来讲，所谓性能测试，指的是测量程序运行的速度，即运行一次要多少时间（通常是执行多次求平均值）。Rust 参照 go 实现了这部分内容。

推荐基准测试专门写在 benches 下，否则容易编译失败，在使用 stable channel 时。

例子如下：

```
#![feature(test)]
extern crate test;

extern crate rusttest;

#[cfg(test)]
mod testbinwen {
    use rusttest::add_two;
    use test::Bencher;

    #[bench]
    fn bench_add_two(b: &mut Bencher) {
        b.iter(|| add_two(2));
    }
}
```

评测函数 fn bench_add_two(b: &mut Bencher) {}上面使用#[bench]做标注，同时函数接受一个参数，b 就是 Rust 提供的评测器。这个写法是固定的。

执行

```
cargo bench
```

即可获得结果。

可以看出，rust 对测试的支持和 go 非常相似。

第十章 Rust 语法补充

10.1 Result 与错误处理

大部分错误并没有严重到需要程序完全停止执行。有时，一个函数会因为一个容易理解并做出反应的原因失败。例如，如果尝试打开一个文件不过由于文件并不存在而失败，此时我们可能想要创建这个文件而不是终止进程。

在 Rust 中，提供了 Result 枚举，它定义有如下两个成员，Ok 和 Err：

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

T 和 E 是泛型类型参数；T 代表成功时返回的 Ok 成员中的数据的类型，而 E 代表失败时返回的 Err 成员中的错误的类型。（T,E 这也是 Rust 中的枚举不同于其它语言枚举的地方）。

因为 Result 有这些泛型类型参数，我们可以将 Result 类型和标准库中为其定义的函数用于很多不同的场景，这些情况中需要返回的成功值和失败值可能会各不相同。

在下面的例子中，我们增加根据 File::open 返回值进行不同处理的逻辑。

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => {
            panic!("There was a problem opening the file: {:?}", error)
        },
    };
}
```

注意与 Option 枚举一样，Result 枚举和其成员也被导入到了 prelude 中，所以就不需要在 match 分支中的 Ok 和 Err 之前指定 Result::。

这里我们告诉 Rust 当结果是 Ok 时，返回 Ok 成员中的 file 值，然后将这个文件句柄赋值给变量 f。match 之后，我们可以利用这个文件句柄来进行读写。

match 的另一个分支处理从 File::open 得到 Err 值的情况。在这种情况下，我们选择调用 panic!宏。如果当前目录没有一个叫做 hello.txt 的文件，当运行这段代码时会看到如下来自 panic! 宏的输出。

10.1.1 匹配不同的错误

上面的例子中代码不管 File::open 是因为什么原因失败都会 panic!。我们真正希望的是对不同的错误原因采取不同的行为：如果 File::open 因为文件不存在而失败，我们希望创建这个文件并返回新文件的句柄。

```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("hello.txt") {
                Ok(fc) => fc,
                Err(e) => panic!("Tried to create file but there was a
problem: {:?}", e),
            },
            other_error => panic!("There was a problem opening the file:
{:?}", other_error),
        },
    };
}
```

File::open 返回的 Err 成员中的值类型 io::Error，它是一个标准库中提供的结构体。这个结构体有一个返回 io::ErrorKind 值的 kind 方法可供调用。io::ErrorKind 是一个标准库提供的枚举，它的成员对应 io 操作可能导致的不同错误类型。我们感兴趣的成员是 ErrorKind::NotFound，它代表尝试打开的文件并不存在。所以 match 的 f 匹配，不过对于 error.kind() 还有一个内部 match。

我们希望在匹配守卫中检查的条件是 error.kind() 的返回值是 ErrorKind 的 NotFound 成员。如果是，则尝试通过 File::create 创建文件。然而因为 File::create 也可能会失败，还需要增加一个内部 match 语句。当文件不能被打开，会打印出一个不同的错误信息。外部 match 的最后一个分支保持不变这样对任何除了文件不存在的错误会使程序 panic。

Result<T, E> 有很多接受闭包的方法，并采用 match 表达式实现。一个更老练的 Rustacean 可能会这么写：

```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let f = File::open("hello.txt").map_err(|error| {
        if error.kind() == ErrorKind::NotFound {
            File::create("hello.txt").unwrap_or_else(|error| {
                panic!("Tried to create file but there was a problem:
{:?}", error);
            })
        } else {
            panic!("There was a problem opening the file: {:?}", error);
        }
```

```
    });
}
```

10.1.2 unwrap 与 expect

match 能够胜任它的工作，不过它可能有点冗长并且不总是能很好的表明其意图。Result<T, E> 类型定义了很多辅助方法来处理各种情况。其中之一叫做 unwrap，如果 Result 值是成员 Ok，unwrap 会返回 Ok 中的值。如果 Result 是成员 Err，unwrap 会为我们调用 panic!。这里是一个实践 unwrap 的例子：

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt").unwrap();
}
```

还有另一个类似于 unwrap 的方法它还允许我们选择 panic! 的错误信息：expect。使用 expect 而不是 unwrap 并提供一个好的错误信息可以表明你的意图并更易于追踪 panic 的根源。expect 的语法看起来像这样：

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt").expect("Failed to open hello.txt");
}
```

expect 与 unwrap 的使用方式一样：返回文件句柄或调用 panic! 宏。expect 用来调用 panic! 的错误信息将会作为参数传递给 expect ，而不像 unwrap 那样使用默认的 panic! 信息。

11.1.3 传播错误与传播错误的简写

11.1.3.1 传播错误

当编写一个其实现会调用一些可能会失败的操作的函数时，除了在这个函数中处理错误外，还可以选择让调用者知道这个错误并决定该如何处理。这被称为 传播（propagating）错误，这样能更好的控制代码调用，因为比起你代码所拥有的上下文，调用者可能拥有更多信息或逻辑来决定应该如何处理错误。

例如，下面的例子中展示了一个从文件中读取用户名的函数。如果文件不存在或不能读取，这个函数会将这些错误返回给调用它的代码：

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

fn read_username_from_file() -> Result<String, io::Error> {
    let f = File::open("hello.txt");

    let mut f = match f {
        Ok(file) => file,
        Err(e) => return Err(e),
    };
```

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

    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}
```

调用这个函数的代码最终会得到一个包含用户名的 Ok 值，或者一个包含 io::Error 的 Err 值。我们无从得知调用者会如何处理这些值。例如，如果他们得到了一个 Err 值，他们可能会选择 panic! 并使程序崩溃、使用一个默认的用户名或者从文件之外的地方寻找用户名。我们没有足够的信息知晓调用者具体会如何尝试，所以将所有的成功或失败信息向上传播，让他们选择合适的处理方法。

这种传播错误的模式在 Rust 是如此的常见，以至于有一个更简便的专用语法：?。

11.1.3.2 传播错误的简写

下面例子展示了一个 read_username_from_file 的实现，它实现了与上面例子中的代码相同的功能，不过这个实现使用了问号运算符（运算符重载）：

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

fn read_username_from_file() -> Result<String, io::Error> {
    let mut f = File::open("hello.txt")?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    Ok(s)
}
```

File::open 调用结尾的 ? 将会把 Ok 中的值返回给变量 f。如果出现了错误，? 会提早返回整个函数并将一些 Err 值传播给调用者。同理也适用于 read_to_string 调用结尾的 ?。

? 消除了大量样板代码并使得函数的实现更简单。我们甚至可以在 ? 之后直接使用链式方法调用来进一步缩短代码，如下面的例子所示：

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

fn read_username_from_file() -> Result<String, io::Error> {
    let mut s = String::new();

    File::open("hello.txt")?.read_to_string(&mut s)?;

    Ok(s)
}
```

? 只能被用于返回值类型为 Result 的函数。

下面的例子中：

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt")?;
}
```

当编译这些代码，会得到如下错误信息：

```
error[E0277]: the `?` operator can only be used in a function that
returns `Result` or `Option` (or another type that implements
`std::ops::Try`)
 --> src/main.rs:4:13
  |
4 |     let f = File::open("hello.txt")?;
  |             ^^^^^^^^^^^^^^^^^^^^^^^^ cannot use the `?` operator in a
function that returns `()`
  |
  = help: the trait `std::ops::Try` is not implemented for `()`
  = note: required by `std::ops::Try::from_error`
```

错误指出只能在返回 Result 的函数中使用 ?。在不返回 Result 的函数中，当调用其他返回 Result 的函数时，需要使用 match 或 Result 的方法之一来处理，而不能用 ? 将潜在的错误传播给代码调用方。

10.2 Any 和反射

```
use std::any::Any;
use std::fmt::Debug ;

fn load_config<T:Any+Debug>(value: &T) -> Vec<String>{
    let mut cfgs: Vec<String>= vec![];
    let value = value as &Any;
    match value.downcast_ref::<String>() {
        Some(cfp) => cfgs.push(cfp.clone()),
        None => (),
    };

    match value.downcast_ref::<Vec<String>>() {
        Some(v) => cfgs.extend_from_slice(&v),
        None =>(),
    }

    if cfgs.len() == 0 {
        panic!("No Config File");
    }
    cfgs
}

fn main() {
    let cfp = "/etc/wayslog.conf".to_string();
    assert_eq!(load_config(&cfp), vec!["/etc/wayslog.conf".to_string()]);
    let cfps = vec!["/etc/wayslog.conf".to_string(),
                    "/etc/wayslog_sec.conf".to_string()];
    assert_eq!(load_config(&cfps),
               vec!["/etc/wayslog.conf".to_string(),
                    "/etc/wayslog_sec.conf".to_string()]);
}
```

熟悉 Java 的同学肯定对 Java 的反射能力记忆犹新，同样的，Rust 也提供了运行时反射的能力。但是，这里有点小小的不同，因为 Rust 不带 VM 不带 Runtime ，因此，其提供的反射更像是一种编译时反射。

因为，Rust 只能对 'static 生命周期的变量（常量）进行反射！

我们来重点分析一下中间这个函数：

```
fn load_config<T:Any+Debug>(value: &T) -> Vec<String>{..}
```

首先，这个函数接收一个泛型 T 类型，T 必须实现了 Any 和 Debug。

这里可能有同学疑问了，你不是说只能反射 'static 生命周期的变量么？我们来看一下 Any 限制：

```
pub trait Any: 'static + Reflect {
    fn get_type_id(&self) -> TypeId;
}
```

看，Any 在定义的时候就规定了其生命周期，而 Reflect 是一个 Marker，默认所有的 Rust 类型都会实现他！注意，这里不是所有原生类型，而是所有类型。

好的，继续，由于我们无法判断出传入的参数类型，因此，只能从运行时候反射类型。

```
let value = value as &Any;
```

首先，我们需要将传入的类型转化成一个 trait Object，当然了，你高兴的话用 UFCS 也是可以做的，参照本章最后的附录。

这样，value 就可以被堪称一个 Any 了。然后，我们通过 downcast_ref 来进行类型推断。如果类型推断成功，则 value 就会被转换成原来的类型。

有的同学看到这里有点懵，为什么你都转换成 Any 了还要转回来？

其实，转换成 Any 是为了有机会获取到他的类型信息，转换回来，则是为了去使用这个值本身。

最后，我们对不同的类型处以不同的处理逻辑。最终，一个反射函数就完成了。