The Rust Programming Language Rust 编程语言

The Rust Team，hainuo

2015-05-15

Contents

1

Introduction 介绍说明

Welcome! This book will teach you about the Rust Programming Language[1].Rust is a systems programming language focused on three goals: safety, speed,and concurrency. It maintains these goals without having a garbage collector,making it a useful language for a number of use cases other languages aren't good at: embedding in other languages, programs with specific space and time requirements, and writing low-level code, like device drivers and operating systems. It improves on current languages targeting this space by having a number of compile-time safety checks that produce no runtime overhead, while eliminating all data races. Rust also aims to achieve 'zero-cost abstractions' even though some of these abstractions feel like those of a high-level language. Even then, Rust still allows precise control like a low-level language would.

欢迎你了解 Rust！这本书将教你关于 Rust 编程语言[2]的知识。Rust 是一个系统编程语言，它的诞生是为了做到三个目标：安全、快速、和并发。它没有使用垃圾回收器，来实现这些目标；这使它变得非常有用，尤其是在其他语言并不擅长的用例上：在特殊空间和时间要求嵌入到其他语言中，和编写底层代码，比如设陪驱动程序和操作系统。在清除了所有数据 races 的同时，通过没有任何运行时开销的多个编译时的安全检查操作改善当前语言的性能，Rust 同样致力于实现"零成本抽象概念"，尽管有些抽象概念像是高级语言。然而及时这样，Rust 仍然向一个低级语言一样允许精确控制。

"The Rust Programming Language" is split into seven sections. This introduction is the first. After this:

"Rust 编程语言"分为 7 大部分，这个说明是第一部分，然后是：

- Getting started **开始使用** (chapter 2, page 17) - Set up your computer for Rust development. **配置你的 Rust 开发环境。**
- Learn Rust **学习** Rust (chapter 3, page 27) - Learn Rust programming through small projects. **通过几个小例子来学习 Rust 编程。**

---

[1]http://rust-lang.org

[2]http://rust-lang.org

- Effective Rust **高效的** Rust (chapter 4, page 65) - Higher-level concepts for writing excellent Rust code. **编写优质** Rust **代码的高级理论。**
- Syntax and Semantics **语法与语义** (chapter 5, page 147) - Each bit of Rust, broken down into small chunks.Rust **的每一部分，分解成一个个小块。**
- Nightly Rust **每日** Rust **构建** (chapter 6, page 277) - Cutting-edge features that aren't in stable builds yet. **尚未稳定构建的顶端特性。**
- Glossary **术语表** (chapter 7, page 301) - A reference of terms used in the book. **本书中一些术语的引用**
- Academic Research **学术研究** (chapter 8, page 303) - Literature that influenced Rust. **影响了** Rust **的文献。**

After reading this introduction, you'll want to dive into either 'Learn Rust' or 'Syntax and Semantics', depending on your preference: 'Learn Rust' if you want to dive in with a project, or 'Syntax and Semantics' if you prefer to start small, and learn a single concept thoroughly before moving onto the next.Copious cross-linking connects these parts together.

在阅读完本介绍之后，你可以根据你的喜好，随意选择 Learn Rust **学习** Rust **或者** Syntax and Semantics **语法与语义章节：如果你想要从一个项目开始，那么选择** Learn Rust **学习** Rust**，或者如果你喜欢从小事做起，通过移动到下一页，一个概念一个概念的学习，将这些广泛的关联的只是连接起来。**

1.1 Contributing **社区贡献**

The source files from which this book is generated can be found on Github: github.com/rust-lang/rust/tree/master/src/doc/trpl[3]

生成本书的源代码的被放在 github **上：** github.com/rust-lang/rust/tree/master/src/doc/trpl[4]

1.2 A brief introduction to Rust Rust **简单说明**

Is Rust a language you might be interested in? Let's examine a few small code samples to show off a few of its strengths.

Rust **是你可能感兴趣的语言吗？让我们看着一些小的代码例子，展示它的一些优势。**

The main concept that makes Rust unique is called 'ownership'. Consider this small example:

使 Rust **独一无二的主要概念被称作** "ownership **所有权**"**。参考如下例子：**

```
fn main() {
  let mut x = vec!["Hello", "world"];
}
```

---

[3]https://github.com/rust-lang/rust/tree/master/src/doc/trpl
[4]https://github.com/rust-lang/rust/tree/master/src/doc/trpl

This program makes a variable binding (section 5.1, page 147) named x. The value of this binding is a Vec<T>, a 'vector', that we create through a macro (section 5.35, page 256) defined in the standard library. This macro is called vec, and we invoke macros with a !. This follows a general principle of Rust: make things explicit. Macros can do significantly more complicated things than function calls, and so they're visually distinct. The ! also helps with parsing, making tooling easier to write, which is also important.

这段程序创建了一个叫做 x **的** variable binding **变量绑定** (section 5.1, page 147)。**这个绑定的值是一个** Vec<T>——**一个向量** vector，**他是在标准库中通过一个** macro **宏** (section 5.35, page 256) **来定义的。这个宏被称作** Vec，**我们使用**! **来强调宏。这遵循了** Rust **的一般原则：把事情明确。宏可以做比函数调用更加显著更加复杂的事情，这样使他们在视觉上区分开来。**! **同样帮助解析，是的工具更容易编写，所以也很重要。**

We used mut to make x mutable: bindings are immutable by default in Rust.We'll be mutating this vector later in the example.

在 Rust **语言中，默认变量是不可变的，我们通过使用** mut **使得变量** x **能够被改变。在稍后的例子中，我们将改变这个向量。**

It's also worth noting that we didn't need a type annotation here: while Rust is statically typed, we didn't need to explicitly annotate the type. Rust has type inference to balance out the power of static typing with the verbosity of annotating types.

值得一提的是，我们不需要在这里声明一个类型，因为 Rust **是静态类型的，我们不需要明确声明一个类型。**Rust **拥有类型推断，能够平衡类型标记型静态类型的能量。**

Rust prefers stack allocation to heap allocation: x is placed directly on the stack. However, the Vec<T> type allocates space for the elements of the vector on the heap. If you're not familiar with this distinction, you can ignore it for now, or check out 'The Stack and the Heap' (section 4.1, page 65). As a systems programming language, Rust gives you the ability to control how your memory is allocated, but when we're getting started, it's less of a big deal.

比起堆内存管理，Rust **语言更喜欢栈内存管理：** x **变量直接被分配在栈上，然而** Vec<T> **类型因为向量的元素组成被分配在堆上。如果你不熟悉这个地方，现在可以忽略它，或者点击链接** 'The Stack and the Heap **栈和堆**' (section 4.1, page 65) **了解它。作为一门系统编程语言，**Rust **给我们控制内存分配的能力，但是，当我们开始使用** Rust **时，它不再是一个大麻烦。**

Earlier, we mentioned that 'ownership' is the key new concept in Rust. In Rust parlance, x is said to 'own' the vector. This means that when x goes out of scope, the vector's memory will be deallocated. This is done deterministically by the Rust compiler, rather than through a mechanism such as a garbage collector. In other words, in Rust, you don't call functions like malloc and free yourself: the compiler statically determines when you need to allocate or deallocate memory, and inserts those calls itself. To err is to be human,but compilers never forget.

前面我们说过 ownership **所有权在** Rust **语言中是一个关键的新概念。在** Rust **语言中，** x **可以被说成是自己的向量。也就是说，当** x **超出作用域范围后，向量的内存空间将被释放。这是由** Rust **语言编译器决定的，而不是通过一个像垃圾回收器之类的机制来决定。换句话说，在** Rust **中，你不需要像** malloc **和** free **自己的方式来调用函数：编译器明确的决定了什么时候你需要分配或者释放内存，并且自动插入这些调用本身。人会忘记，但是编译器永远不会。**

Let's add another line to our example:

让我们在例子中增加一行：

```rust
fn main() {
  let mut x = vec!["Hello", "world"];

  let y = &x[0];
}
```

We've introduced another binding, y. In this case, y is a 'reference' to the first element of the vector. Rust's references are similar to pointers in other languages, but with additional compile-time safety checks. References interact with the ownership system by 'borrowing' (section 5.9, page 170) what they point to, rather than owning it. The difference is, when the reference goes out of scope, it will not deallocate the underlying memory. If it did, we'd de-allocate twice, which is bad!

我们声明了另一个变量，y。在案例中，y 是向量第一个元素的一个 reference 地址引用。Rust 的引用跟其他语言的指针比较相似，但是增加了编译时安全检查。地址引用通过他们指向的 borrowing 引用 (section 5.9, page 170) 来影响所有权系统，而不是通过拥有它来影响。不同的是，当地址引用超出了作用域范围，他不会释放原本的内存。如果他做了，我们就会释放两次，这是很坏的事情！

Let's add a third line. It looks innocent enough, but causes a compiler error:

现在增加第三行代码。他看起来很无辜，但是会引起编译错误：

```rust
fn main() {
  let mut x = vec!["Hello", "world"];

  let y = &x[0];

  x.push("foo");
}
```

push is a method on vectors that appends another element to the end of the vector. When we try to compile this program, we get an error:

push 是向量一个方法，向向量的末尾追加另一个元素。当我们试图编译程序时，我们得到错误信息：

```
error: cannot borrow `x` as mutable because it is also borrowed as immutable
  x.push("foo");
  ^
note: previous borrow of `x` occurs here; the immutable borrow prevents
subsequent moves or mutable borrows of `x` until the borrow ends
  let y = &x[0];
        ^
note: previous borrow ends here
fn main() {

}
^
```

Whew! The Rust compiler gives quite detailed errors at times, and this is one of those times. As the error explains, while we made our binding mutable, we still cannot call push. This is because we already have a reference to an element of the vector, y. Mutating something while another reference exists is dangerous, because we may invalidate the reference. In this specific case, when we create the vector, we may have only allocated space for three elements.Adding a fourth would mean allocating a new chunk of memory for all those elements,copying the old values over, and updating the internal pointer to that memory.That all works just fine.The problem is that y wouldn't get updated, and so we'd have a 'dangling pointer'. That's bad. Any use of y would be an error in this case, and so the compiler has caught this for us.

哦！Rust 编译器有时会给出非常详细的错误信息，这就是那些情况中的一种。错误说明，当我们创建了一个可变的变量时，我们仍然不能够调用 push 方法。这是因为我们已经有了一个向量元素的地址引用——y。当有元素进行地址引用时，改变一些内容是非常危险的，因为我们可能会废止地址引用。在特殊情况下，我们创建一个向量，我们只给他分配了三个元素的空间。增加第四个元素，将意味着分配一个新的内存块给所有的元素，将旧的的内存覆盖，并更新内部指针到那部分内存。它们做的刚刚好。问题是，y 没有被更新，所以我们就会哟一个' 空指针'。这是很坏的。在本案例中，y 的任何使用都是错误的，所以编译器抓取这个信息给我们。

So how do we solve this problem? There are two approaches we can take. The first is making a copy rather than using a reference:

```rust
fn main() {
    let mut x = vec!["Hello", "world"];

    let y = x[0].clone();

    x.push("foo");
}
```

Rust has move semantics (subsection 5.8.3, page 168) by default, so if we want to make a copy of some data, we call the clone() method. In this example, y is no longer a reference to the vector stored in x, but a copy of its first element, "Hello". Now that we don't have a reference, our push() works just fine.

Rust 默认有 move 语义 (subsection 5.8.3, page 168)，所以我们想要复制一些数据，我们可以调用 clone() 方法。在上述例子中，y 不再是在 x 中保存的向量的一个地址引用，而是他第一个元素"Hello" 的复制品。现在我们使用地址引用，所以 push() 方法运行良好。

If we truly want a reference, we need the other option: ensure that our reference goes out of scope before we try to do the mutation. That looks like this:

如果我们真的想使用一个地址引用，我们需要另一个方式：在试图改变变量之前确定地址引用不在当前作用域内。就像下面的例子：

```rust
fn main() {
    let mut x = vec!["Hello", "world"];

    {
```

```
    let y = &x[0];
  }

  x.push("foo");
}
```

We created an inner scope with an additional set of curly braces. y will go out of scope before we call push(), and so we're all good.

我们使用额外的一组花括号构建了一个内置作用域。在我们调用 push() 之前 y 已经超出了作用域范围，所以代码非常好。

This concept of ownership isn't just good for preventing dangling pointers, but an entire set of related problems, like iterator invalidation, concurrency, and more.

所有权的概念不止是能够很好的避免空指针，还有一整套的相关难点，比如迭代器失效、并发等等。

2

Getting Started 开始使用 Rust

This first section of the book will get you going with Rust and its tooling. First, we'll install Rust. Then, the classic 'Hello World' program. Finally, we'll talk about Cargo, Rust's build system and package manager.

2.1 Installing Rust 安装 Rust

The first step to using Rust is to install it! There are a number of ways to install Rust, but the easiest is to use the rustup script. If you're on Linux or a Mac, all you need to do is this (note that you don't need to type in the $s, they just indicate the start of each command):

$ curl -sf -L https://static.rust-lang.org/rustup.sh | sh

If you're concerned about the potential insecurity[1] of using curl|sh, please keep reading and see our disclaimer below. And feel free to use a two-step version of the installation and examine our installation script:

$ curl -f -L https://static.rust-lang.org/rustup.sh -O
$ sh rustup.sh

If you're on Windows, please download either the 32-bit installer[2] or the 64-bit installer[3] and run it.

---

[1]http://curlpipesh.tumblr.com

[2]https://static.rust-lang.org/dist/rust-1.0.0-beta-i686-pc-windows-gnu.msi

[3]https://static.rust-lang.org/dist/rust-1.0.0-beta-x86_64-pc-windows-gnu.msi

Uninstalling

If you decide you don't want Rust anymore, we'll be a bit sad, but that's okay. Not every programming language is great for everyone. Just run the uninstall script:

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

If you used the Windows installer, just re-run the .msi and it will give you an uninstall option.

Some people, and somewhat rightfully so, get very upset when we tell you to curl|sh. Basically, when you do this, you are trusting that the good people who maintain Rust aren't going to hack your computer and do bad things. That's a good instinct! If you're one of those people, please check out the documentation on building Rust from Source[4], or the official binary downloads[5].

Oh, we should also mention the officially supported platforms:

- Windows (7, 8, Server 2008 R2)
- Linux (2.6.18 or later, various distributions), x86 and x86-64
- OSX 10.7 (Lion) or greater, x86 and x86-64

We extensively test Rust on these platforms, and a few others, too, like Android. But these are the ones most likely to work, as they have the most testing.

Finally, a comment about Windows. Rust considers Windows to be a first-class platform upon release, but if we're honest, the Windows experience isn't as integrated as the Linux/OS X experience is. We're working on it! If anything does not work, it is a bug. Please let us know if that happens. Each and every commit is tested against Windows just like any other platform.

If you've got Rust installed, you can open up a shell, and type this:

$ rustc --version

You should see the version number, commit hash, commit date and build date:

rustc 1.0.0-beta (9854143cb 2015-04-02) (built 2015-04-02)

If you did, Rust has been installed successfully! Congrats!

This installer also installs a copy of the documentation locally, so you can read it offline. On UNIX systems, /usr/local/share/doc/rust is the location. On Windows, it's in a share/doc directory, inside wherever you installed Rust to.

If not, there are a number of places where you can get help. The easiest is the #rust IRC channel on irc.mozilla.org[6], which you can access through Mibbit[7]. Click that link, and you'll be chatting

---

[4]https://github.com/rust-lang/rust#building-from-source
[5]http://www.rust-lang.org/install.html
[6]irc://irc.mozilla.org/#rust
[7]http://chat.mibbit.com/?server=irc.mozilla.org&channel=%23rust

with other Rustaceans (a silly nickname we call ourselves), and we can help you out. Other great resources include the user's forum[8], and Stack Overflow[9].

2.2 Hello, world!

Now that you have Rust installed, let's write your first Rust program. It's traditional to make your first program in any new language one that prints the text "Hello, world!" to the screen. The nice thing about starting with such a simple program is that you can verify that your compiler isn't just installed, but also working properly. And printing information to the screen is a pretty common thing to do.

The first thing that we need to do is make a file to put our code in. I like to make a projects directory in my home directory, and keep all my projects there. Rust does not care where your code lives.

This actually leads to one other concern we should address: this guide will assume that you have basic familiarity with the command line. Rust itself makes no specific demands on your editing tooling, or where your code lives. If you prefer an IDE to the command line, you may want to check out SolidOak[10], or wherever plugins are for your favorite IDE. There are a number of extensions of varying quality in development by the community. The Rust team also ships plugins for various editors[11]. Configuring your editor or IDE is out of the scope of this tutorial, so check the documentation for your setup, specifically.

With that said, let's make a directory in our projects directory.

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

If you're on Windows and not using PowerShell, the ~ may not work. Consult the documentation for your shell for more details.

Let's make a new source file next. We'll call our file main.rs. Rust files always end in a .rs extension. If you're using more than one word in your filename, use an underscore: hello_world.rs rather than helloworld.rs.

Now that you've got your file open, type this in:

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

Save the file, and then type this into your terminal window:

---

[8]http://users.rust-lang.org/
[9]http://stackoverflow.com/questions/tagged/rust
[10]https://github.com/oakes/SolidOak
[11]https://github.com/rust-lang/rust/blob/master/src/etc/CONFIGS.md

```
$ rustc main.rs
$ ./main # or main.exe on Windows
Hello, world!
```

Success! Let's go over what just happened in detail.

```
fn main() {

}
```

These lines define a *function* in Rust. The main function is special: it's the beginning of every Rust program. The first line says "I'm declaring a function named main which takes no arguments and returns nothing." If there were arguments, they would go inside the parentheses ((and)), and because we aren't returning anything from this function, we can omit the return type entirely. We'll get to it later.

You'll also note that the function is wrapped in curly braces ({ and }). Rust requires these around all function bodies. It is also considered good style to put the opening curly brace on the same line as the function declaration, with one space in between.

Next up is this line:

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

This line does all of the work in our little program. There are a number of details that are important here. The first is that it's indented with four spaces, not tabs. Please configure your editor of choice to insert four spaces with the tab key. We provide some sample configurations for various editors[12].

The second point is the println!() part. This is calling a Rust macro (section 5.35, page 256), which is how metaprogramming is done in Rust. If it were a function instead, it would look like this: println(). For our purposes, we don't need to worry about this difference. Just know that sometimes, you'll see a !, and that means that you're calling a macro instead of a normal function. Rust implements println! as a macro rather than a function for good reasons, but that's an advanced topic. One last thing to mention: Rust's macros are significantly different from C macros, if you've used those. Don't be scared of using macros. We'll get to the details eventually, you'll just have to trust us for now.

Next, "Hello, world!" is a 'string'. Strings are a surprisingly complicated topic in a systems programming language, and this is a 'statically allocated' string. If you want to read further about allocation, check out the stack and the heap (section 4.1, page 65), but you don't need to right now if you don't want to. We pass this string as an argument to println!, which prints the string to the screen. Easy enough!

Finally, the line ends with a semicolon (;). Rust is an 'expression oriented' language, which means that most things are expressions, rather than statements. The ; is used to indicate that this expression is over, and the next one is ready to begin. Most lines of Rust code end with a ;.

---

[12]https://github.com/rust-lang/rust/tree/master/src/etc/CONFIGS.md

Finally, actually compiling and running our program. We can compile with our compiler, rustc, by passing it the name of our source file:

$ rustc main.rs

This is similar to gcc or clang, if you come from a C or C++ background. Rust will output a binary executable. You can see it with ls:

$ ls
main main.rs

Or on Windows:

$ dir
main.exe main.rs

There are now two files: our source code, with the .rs extension, and the executable (main.exe on Windows, main everywhere else)

$ ./main # or main.exe on Windows

This prints out our Hello, world! text to our terminal.

If you come from a dynamic language like Ruby, Python, or JavaScript, you may not be used to these two steps being separate. Rust is an 'ahead-of-time compiled language', which means that you can compile a program, give it to someone else, and they don't need to have Rust installed. If you give someone a .rb or .py or .js file, they need to have a Ruby/Python/JavaScript implementation installed, but you just need one command to both compile and run your program. Everything is a tradeoff in language design, and Rust has made its choice.

Congratulations! You have officially written a Rust program. That makes you a Rust programmer! Welcome.

Next, I'd like to introduce you to another tool, Cargo, which is used to write real-world Rust programs. Just using rustc is nice for simple things, but as your project grows, you'll want something to help you manage all of the options that it has, and to make it easy to share your code with other people and projects.

2.3   Hello, Cargo! Cargo 命令介绍

Cargo[13] is a tool that Rustaceans use to help manage their Rust projects. Cargo is currently in a pre-1.0 state, and so it is still a work in progress. However, it is already good enough to use for many Rust projects, and so it is assumed that Rust projects will use Cargo from the beginning.

---

[13]http://doc.crates.io

Cargo manages three things: building your code, downloading the dependencies your code needs, and building those dependencies. At first, your program doesn't have any dependencies, so we'll only be using the first part of its functionality. Eventually, we'll add more. Since we started off by using Cargo, it'll be easy to add later.

If you installed Rust via the official installers you will also have Cargo. If you installed Rust some other way, you may want to check the Cargo README[14] for specific instructions about installing it.

Converting to Cargo

Let's convert Hello World to Cargo.

To Cargo-ify our project, we need to do two things: Make a Cargo.toml configuration file, and put our source file in the right place. Let's do that part first:

```
$ mkdir src
$ mv main.rs src/main.rs
```

Note that since we're creating an executable, we used main.rs. If we want to make a library instead, we should use lib.rs. Custom file locations for the entry point can be specified with a [[lib]] or [[bin]][15] key in the TOML file described below.

Cargo expects your source files to live inside a src directory. That leaves the top level for other things, like READMEs, license information, and anything not related to your code. Cargo helps us keep our projects nice and tidy. A place for everything, and everything in its place.

Next, our configuration file:

```
$ editor Cargo.toml
```

Make sure to get this name right: you need the capital C!

Put this inside:

```
[package]

name = "hello_world"
version = "0.0.1"
authors = [ "Your name <you@example.com>" ]
```

This file is in the TOML[16] format. Let's let it explain itself to you:

---

[14]https://github.com/rust-lang/cargo#installing-cargo-from-nightlies
[15]http://doc.crates.io/manifest.html#configuring-a-target
[16]https://github.com/toml-lang/toml

> TOML aims to be a minimal configuration file format that's easy to read due to obvious semantics. TOML is designed to map unambiguously to a hash table. TOML should be easy to parse into data structures in a wide variety of languages.

TOML is very similar to INI, but with some extra goodies.

Once you have this file in place, we should be ready to build! Try this:

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

Bam! We build our project with cargo build, and run it with ./target/debug/hello_world. We can do both in one step with cargo run:

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

Notice that we didn't re-build the project this time. Cargo figured out that we hadn't changed the source file, and so it just ran the binary. If we had made a modification, we would have seen it do both:

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

This hasn't bought us a whole lot over our simple use of rustc, but think about the future: when our project gets more complex, we would need to do more things to get all of the parts to properly compile. With Cargo, as our project grows, we can just cargo build, and it'll work the right way.

When your project is finally ready for release, you can use cargo build --release to compile your project with optimizations.

You'll also notice that Cargo has created a new file: Cargo.lock.

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

This file is used by Cargo to keep track of dependencies in your application. Right now, we don't have any, so it's a bit sparse. You won't ever need to touch this file yourself, just let Cargo handle it.

That's it! We've successfully built hello_world with Cargo. Even though our program is simple, it's using much of the real tooling that you'll use for the rest of your Rust career. You can expect to do this to get started with virtually all Rust projects:

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

A New Project

You don't have to go through this whole process every time you want to start a new project! Cargo has the ability to make a bare-bones project directory in which you can start developing right away.

To start a new project with Cargo, use cargo new:

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

We're passing --bin because we're making a binary program: if we were making a library, we'd leave it off.

Let's check out what Cargo has generated for us:

```
$ cd hello_world
$ tree .
.
⊠──── Cargo.toml
⊠──── src
    ⊠──── main.rs

1 directory, 2 files
```

If you don't have the tree command, you can probably get it from your distribution's package manager. It's not necessary, but it's certainly useful.

This is all we need to get started. First, let's check out Cargo.toml:

```
[package]

name = "hello_world"
version = "0.0.1"
authors = ["Your Name <you@example.com>"]
```

Cargo has populated this file with reasonable defaults based off the arguments you gave it and your git global configuration. You may notice that Cargo has also initialized the hello_world directory as a git repository.

Here's what's in src/main.rs:

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

Cargo has generated a "Hello World!" for us, and you're ready to start coding! Cargo has its own guide[17] which covers Cargo's features in much more depth.

Now that you've got the tools down, let's actually learn more about the Rust language itself. These are the basics that will serve you well through the rest of your time with Rust.

You have two options: Dive into a project with 'Learn Rust (chapter 3, page 27)', or start from the bottom and work your way up with 'Syntax and Semantics (chapter 5, page 147)'. More experienced systems programmers will probably prefer 'Learn Rust', while those from dynamic backgrounds may enjoy either. Different people learn differently! Choose whatever's right for you.

---

[17]http://doc.crates.io/guide.html

3

Learn Rust Rust 入门

Welcome! This section has a few tutorials that teach you Rust through building projects. You'll get a high-level overview, but we'll skim over the details.

If you'd prefer a more 'from the ground up'-style experience, check out Syntax and Semantics (chapter 5, page 147).

3.1 Guessing Game 猜数字

For our first project, we'll implement a classic beginner programming problem: the guessing game. Here's how it works: Our program will generate a random integer between one and a hundred. It will then prompt us to enter a guess. Upon entering our guess, it will tell us if we're too low or too high. Once we guess correctly, it will congratulate us. Sounds good?

3.1.1 Set up

Let's set up a new project. Go to your projects directory. Remember how we had to create our directory structure and a Cargo.toml for hello_world? Cargo has a command that does that for us. Let's give it a shot:

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

We pass the name of our project to cargo new, and then the --bin flag, since we're making a binary, rather than a library.

Check out the generated Cargo.toml:

```
[package]

name = "guessing_game"
version = "0.0.1"
authors = ["Your Name <you@example.com>"]
```

Cargo gets this information from your environment. If it's not correct, go ahead and fix that.

Finally, Cargo generated a 'Hello, world!' for us. Check out src/main.rs:

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

Let's try compiling what Cargo gave us:

```
$ cargo build
   Compiling guessing_game v0.0.1 (file:///home/you/projects/guessing_game)
```

Excellent! Open up your src/main.rs again. We'll be writing all of our code in this file.

Before we move on, let me show you one more Cargo command: run. cargo run is kind of like cargo build, but it also then runs the produced executable. Try it out:

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

Great! The run command comes in handy when you need to rapidly iterate on a project. Our game is just such a project, we need to quickly test each iteration before moving on to the next one.

3.1.2   Processing a Guess

Let's get to it! The first thing we need to do for our guessing game is allow our player to input a guess. Put this in your src/main.rs:

```rust
use std::io;

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

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

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

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

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

There's a lot here! Let's go over it, bit by bit.

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

We'll need to take user input, and then print the result as output. As such, we need the io library from the standard library. Rust only imports a few things into every program, the 'prelude'[1]. If it's not in the prelude, you'll have to use it directly.

```
fn main() {
```

As you've seen before, the main() function is the entry point into your program. The fn syntax declares a new function, the ()s indicate that there are no arguments, and { starts the body of the function. Because we didn't include a return type, it's assumed to be (), an empty tuple[2].

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

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

We previously learned that println!() is a macro (section 5.35, page 256) that prints a string (section 5.18, page 203) to the screen.

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

Now we're getting interesting! There's a lot going on in this little line. The first thing to notice is that this is a let statement (section 5.1, page 147), which is used to create 'variable bindings'. They take this form:

```
let foo = bar;
```

This will create a new binding named foo, and bind it to the value bar. In many languages, this is called a 'variable', but Rust's variable bindings have a few tricks up their sleeves.

---

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

[2] primitive-types.html#tuples

For example, they're immutable (section 5.11, page 183) by default. That's why our example uses
mut: it makes a binding mutable, rather than immutable. let doesn't take a name on the left hand
side, it actually accepts a 'pattern (section 5.15, page 192)'. We'll use patterns more later. It's easy
enough to use for now:

```
let foo = 5; // immutable.
let mut bar = 5; // mutable
```

Oh, and // will start a comment, until the end of the line. Rust ignores everything in comments
(section 5.4, page 162).

So now we know that let mut guess will introduce a mutable binding named guess, but we have to
look at the other side of the = for what it's bound to: String::new().

String is a string type, provided by the standard library. A String[3] is a growable, UTF-8 encoded bit
of text.

The ::new() syntax uses :: because this is an 'associated function' of a particular type. That is to
say, it's associated with String itself, rather than a particular instance of a String. Some languages
call this a 'static method'.

This function is named new(), because it creates a new, empty String. You'll find a new() function
on many types, as it's a common name for making a new value of some kind.

Let's move forward:

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

That's a lot more! Let's go bit-by-bit. The first line has two parts. Here's the first:

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

Remember how we used std::io on the first line of the program? We're now calling an associated
function on it. If we didn't use std::io, we could have written this line as std::io::stdin().

This particular function returns a handle to the standard input for your terminal. More specifically,
a std::io::Stdin[4].

The next part will use this handle to get input from the user:

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

---

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

Here, we call the read_line()[5] method on our handle. Methods (section 5.16, page 197) are like associated functions, but are only available on a particular instance of a type, rather than the type itself. We're also passing one argument to read_line(): &mut guess.

Remember how we bound guess above? We said it was mutable. However, read_line doesn't take a String as an argument: it takes a &mut String. Rust has a feature called 'references (section 5.9, page 170)', which allows you to have multiple references to one piece of data, which can reduce copying. References are a complex feature, as one of Rust's major selling points is how safe and easy it is to use references. We don't need to know a lot of those details to finish our program right now, though. For now, all we need to know is that like let bindings, references are immutable by default. Hence, we need to write &mut guess, rather than &guess.

Why does read_line() take a mutable reference to a string? Its job is to take what the user types into standard input, and place that into a string. So it takes that string as an argument, and in order to add the input, it needs to be mutable.

But we're not quite done with this line of code, though. While it's a single line of text, it's only the first part of the single logical line of code:

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

When you call a method with the .foo() syntax, you may introduce a newline and other whitespace. This helps you split up long lines. We *could* have done:

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

But that gets hard to read. So we've split it up, three lines for three method calls. We already talked about read_line(), but what about ok() and expect()? Well, we already mentioned that read_line() puts what the user types into the &mut String we pass it. But it also returns a value: in this case, an io::Result[6]. Rust has a number of types named Result in its standard library: a generic Result[7], and then specific versions for sub-libraries, like io::Result.

The purpose of these Result types is to encode error handling information. Values of the Result type, like any type, have methods defined on them. In this case, io::Result has an ok() method, which says 'we want to assume this value is a successful one. If not, just throw away the error information'. Why throw it away? Well, for a basic program, we just want to print a generic error, as basically any issue means we can't continue. The ok() method[8] returns a value which has another method defined on it: expect(). The expect() method[9] takes a value it's called on, and if it isn't a successful one, panic! (section 4.7, page 121)s with a message you passed it. A panic! like this will cause our program to crash, displaying the message.

If we leave off calling these two methods, our program will compile, but we'll get a warning:

---

[5]http://doc.rust-lang.org/std/io/struct.Stdin.html#method.read_line
[6]http://doc.rust-lang.org/std/io/type.Result.html
[7]http://doc.rust-lang.org/std/result/enum.Result.html
[8]http://doc.rust-lang.org/std/result/enum.Result.html#method.ok
[9]http://doc.rust-lang.org/std/option/enum.Option.html#method.expect

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

Rust warns us that we haven't used the Result value. This warning comes from a special annotation that io::Result has. Rust is trying to tell you that you haven't handled a possible error. The right way to suppress the error is to actually write error handling. Luckily, if we just want to crash if there's a problem, we can use these two little methods. If we can recover from the error somehow, we'd do something else, but we'll save that for a future project.

There's just one line of this first example left:

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

This prints out the string we saved our input in. The {}s are a placeholder, and so we pass it guess as an argument. If we had multiple {}s, we would pass multiple arguments:

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

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

Easy.

Anyway, that's the tour. We can run what we have with cargo run:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running `target/debug/guessing_game`
Guess the number!
Please input your guess.
6
You guessed: 6
```

All right! Our first part is done: we can get input from the keyboard, and then print it back out.

3.1.3   Generating a secret number

Next, we need to generate a secret number. Rust does not yet include random number functionality in its standard library. The Rust team does, however, provide a rand crate[10]. A 'crate' is a package

---

[10]https://crates.io/crates/rand

of Rust code. We've been building a 'binary crate', which is an executable. rand is a 'library crate', which contains code that's intended to be used with other programs.

Using external crates is where Cargo really shines. Before we can write the code using rand, we need to modify our Cargo.toml. Open it up, and add these few lines at the bottom:

```
[dependencies]

rand="0.3.0"
```

The [dependencies] section of Cargo.toml is like the [package] section: everything that follows it is part of it, until the next section starts. Cargo uses the dependencies section to know what dependencies on external crates you have, and what versions you require. In this case, we've used version 0.3.0. Cargo understands Semantic Versioning[11], which is a standard for writing version numbers. If we wanted to use the latest version we could use * or we could use a range of versions. Cargo's documentation[12] contains more details.

Now, without changing any of our code, let's build our project:

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

(You may see different versions, of course.)

Lots of new output! Now that we have an external dependency, Cargo fetches the latest versions of everything from the registry, which is a copy of data from Crates.io[13]. Crates.io is where people in the Rust ecosystem post their open source Rust projects for others to use.

After updating the registry, Cargo checks our [dependencies] and downloads any we don't have yet. In this case, while we only said we wanted to depend on rand, we've also grabbed a copy of libc. This is because rand depends on libc to work. After downloading them, it compiles them, and then compiles our project.

If we run cargo build again, we'll get different output:

```
$ cargo build
```

That's right, no output! Cargo knows that our project has been built, and that all of its dependencies are built, and so there's no reason to do all that stuff. With nothing to do, it simply exits. If we open up src/main.rs again, make a trivial change, and then save it again, we'll just see one line:

---

[11]http://semver.org
[12]http://doc.crates.io/crates-io.html
[13]https://crates.io

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

So, we told Cargo we wanted any 0.3.x version of rand, and so it fetched the latest version at the time this was written, v0.3.8. But what happens when next week, version v0.3.9 comes out, with an important bugfix? While getting bugfixes is important, what if 0.3.9 contains a regression that breaks our code?

The answer to this problem is the Cargo.lock file you'll now find in your project directory. When you build your project for the first time, Cargo figures out all of the versions that fit your criteria, and then writes them to the Cargo.lock file. When you build your project in the future, Cargo will see that the Cargo.lock file exists, and then use that specific version rather than do all the work of figuring out versions again. This lets you have a repeatable build automatically. In other words, we'll stay at 0.3.8 until we explicitly upgrade, and so will anyone who we share our code with, thanks to the lock file.

What about when we *do* want to use v0.3.9? Cargo has another command, update, which says 'ignore the lock, figure out all the latest versions that fit what we've specified. If that works, write those versions out to the lock file'. But, by default, Cargo will only look for versions larger than 0.3.0 and smaller than 0.4.0. If we want to move to 0.4.x, we'd have to update the Cargo.toml directly. When we do, the next time we cargo build, Cargo will update the index and re-evaluate our rand requirements.

There's a lot more to say about Cargo[14] and its ecosystem[15], but for now, that's all we need to know. Cargo makes it really easy to re-use libraries, and so Rustaceans tend to write smaller projects which are assembled out of a number of sub-packages.

Let's get on to actually *using* rand. Here's our next step:

```rust
extern crate rand;

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

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

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

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

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

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .ok()
```

---

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

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

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

The first thing we've done is change the first line. It now says extern crate rand. Because we declared rand in our [dependencies], we can use extern crate to let Rust know we'll be making use of it. This also does the equivalent of a use rand; as well, so we can make use of anything in the rand crate by prefixing it with rand::.

Next, we added another use line: use rand::Rng. We're going to use a method in a moment, and it requires that Rng be in scope to work. The basic idea is this: methods are defined on something called 'traits', and for the method to work, it needs the trait to be in scope. For more about the details, read the traits (section 5.20, page 208) section.

There are two other lines we added, in the middle:

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

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

We use the rand::thread_rng() function to get a copy of the random number generator, which is local to the particular thread (section 4.6, page 113) of execution we're in. Because we use rand::Rng'd above, it has a gen_range() method available. This method takes two arguments, and generates a number between them. It's inclusive on the lower bound, but exclusive on the upper bound, so we need 1 and 101 to get a number between one and a hundred.

The second line just prints out the secret number. This is useful while we're developing our program, so we can easily test it out. But we'll be deleting it for the final version. It's not much of a game if it prints out the answer when you start it up!

Try running our new program a few times:

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

Great! Next up: let's compare our guess to the secret guess.

3.1.4   Comparing guesses

Now that we've got user input, let's compare our guess to the random guess. Here's our next step, though it doesn't quite work yet:

```rust
extern crate rand;

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

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

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

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

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

    let mut guess = String::new();

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

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

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

A few new bits here. The first is another use. We bring a type called std::cmp::Ordering into scope. Then, five new lines at the bottom that use it:

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

The cmp() method can be called on anything that can be compared, and it takes a reference to the thing you want to compare it to. It returns the Ordering type we used earlier. We use a match (section 5.14, page 190) statement to determine exactly what kind of Ordering it is. Ordering is an enum (section 5.13, page 189), short for 'enumeration', which looks like this:

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

With this definition, anything of type Foo can be either a Foo::Bar or a Foo::Baz. We use the :: to indicate the namespace for a particular enum variant.

The Ordering[16] enum has three possible variants: Less, Equal, and Greater. The match statement takes a value of a type, and lets you create an 'arm' for each possible value. Since we have three types of Ordering, we have three arms:

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

If it's Less, we print Too small!, if it's Greater, Too big!, and if Equal, You win!. match is really useful, and is used often in Rust.

I did mention that this won't quite work yet, though. Let's try it:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
src/main.rs:28:21: 28:35 error: mismatched types:
 expected `&collections::string::String`,
   found `&_`
(expected struct `collections::string::String`,
   found integral variable) [E0308]
src/main.rs:28    match guess.cmp(&secret_number) {
                  ^~~~~~~~~~~~~~
error: aborting due to previous error
Could not compile `guessing_game`.
```

Whew! This is a big error. The core of it is that we have 'mismatched types'. Rust has a strong, static type system. However, it also has type inference. When we wrote let guess = String::new(), Rust was able to infer that guess should be a String, and so it doesn't make us write out the type. And with our secret_number, there are a number of types which can have a value between one and a

---

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

hundred: i32, a thirty-two-bit number, or u32, an unsigned thirty-two-bit number, or i64, a sixty-four-bit number. Or others. So far, that hasn't mattered, and so Rust defaults to an i32. However, here, Rust doesn't know how to compare the guess and the secret_number. They need to be the same type. Ultimately, we want to convert the String we read as input into a real number type, for comparison. We can do that with three more lines. Here's our new program:

```rust
extern crate rand;

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

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

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

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

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

    let mut guess = String::new();

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

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

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

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

The new three lines:

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

Wait a minute, I thought we already had a guess? We do, but Rust allows us to 'shadow' the previous guess with a new one. This is often used in this exact situation, where guess starts as a String, but we want to convert it to an u32. Shadowing lets us re-use the guess name, rather than forcing us to come up with two unique names like guess_str and guess, or something else.

We bind guess to an expression that looks like something we wrote earlier:

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

Followed by an ok().expect() invocation. Here, guess refers to the old guess, the one that was a String with our input in it. The trim() method on Strings will eliminate any white space at the beginning and end of our string. This is important, as we had to press the 'return' key to satisfy read_line(). This means that if we type 5 and hit return, guess looks like this: 5\n. The \n represents 'newline', the enter key. trim() gets rid of this, leaving our string with just the 5. The parse() method on strings[17] parses a string into some kind of number. Since it can parse a variety of numbers, we need to give Rust a hint as to the exact type of number we want. Hence, let guess: u32. The colon (:) after guess tells Rust we're going to annotate its type. u32 is an unsigned, thirty-two bit integer. Rust has a number of built-in number types[18], but we've chosen u32. It's a good default choice for a small positive number.

Just like read_line(), our call to parse() could cause an error. What if our string contained A%? There'd be no way to convert that to a number. As such, we'll do the same thing we did with read_line(): use the ok() and expect() methods to crash if there's an error.

Let's try our program out!

```
$ cargo run
  Compiling guessing_game v0.0.1 (file:///home/you/projects/guessing_game)
    Running `target/guessing_game`
Guess the number!
The secret number is: 58
Please input your guess.
  76
You guessed: 76
Too big!
```

Nice! You can see I even added spaces before my guess, and it still figured out that I guessed 76. Run the program a few times, and verify that guessing the number works, as well as guessing a number too small.

Now we've got most of the game working, but we can only make one guess. Let's change that by adding loops!

---

[17]http://doc.rust-lang.org/std/primitive.str.html#method.parse
[18]primitive-types.html#numeric-types

3.1.5   Looping

The loop keyword gives us an infinite loop. Let's add that in:

```rust
extern crate rand;

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

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

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

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

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

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

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

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

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

And try it out. But wait, didn't we just add an infinite loop? Yup. Remember our discussion about parse()? If we give a non-number answer, we'll return and quit. Observe:

```
$ cargo run
   Compiling guessing_game v0.0.1 (file:///home/you/projects/guessing_game)
     Running `target/guessing_game`
```

```
Guess the number!
The secret number is: 59
Please input your guess.
45
You guessed: 45
Too small!
Please input your guess.
60
You guessed: 60
Too big!
Please input your guess.
59
You guessed: 59
You win!
Please input your guess.
quit
thread '<main>' panicked at 'Please type a number!'
```

Ha! quit actually quits. As does any other non-number input. Well, this is suboptimal to say the least. First, let's actually quit when you win the game:

```rust
extern crate rand;

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

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

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

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

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

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

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

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

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

By adding the break line after the You win!, we'll exit the loop when we win. Exiting the loop also means exiting the program, since it's the last thing in main(). We have just one more tweak to make: when someone inputs a non-number, we don't want to quit, we just want to ignore it. We can do that like this:

```rust
extern crate rand;

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

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

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

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

  loop {
    println!("Please input your guess.");

    let mut guess = String::new();

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

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

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

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

These are the lines that changed:

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

This is how you generally move from 'crash on error' to 'actually handle the error', by switching from ok().expect() to a match statement. The Result returned by parse() is an enum just like Ordering, but in this case, each variant has some data associated with it: Ok is a success, and Err is a failure. Each contains more information: the successful parsed integer, or an error type. In this case, we match on Ok(num), which sets the inner value of the Ok to the name num, and then we just return it on the right-hand side. In the Err case, we don't care what kind of error it is, so we just use _ instead of a name. This ignores the error, and continue causes us to go to the next iteration of the loop.

Now we should be good! Let's try:

```
$ cargo run
   Compiling guessing_game v0.0.1 (file:///home/you/projects/guessing_game)
     Running `target/guessing_game`
Guess the number!
The secret number is: 61
Please input your guess.
10
You guessed: 10
Too small!
Please input your guess.
99
You guessed: 99
Too big!
Please input your guess.
```

```
foo
Please input your guess.
61
You guessed: 61
You win!
```

Awesome! With one tiny last tweak, we have finished the guessing game. Can you think of what it is? That's right, we don't want to print out the secret number. It was good for testing, but it kind of ruins the game. Here's our final source:

```rust
extern crate rand;

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

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

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

  loop {
    println!("Please input your guess.");

    let mut guess = String::new();

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

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

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

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

```
}
```

3.1.6 Complete!

At this point, you have successfully built the Guessing Game! Congratulations!

This first project showed you a lot: let, match, methods, associated functions, using external crates, and more. Our next project will show off even more.

3.2 Dining Philosophers 哲学家就餐问题

For our second project, let's look at a classic concurrency problem. It's called 'the dining philosophers'. It was originally conceived by Dijkstra in 1965, but we'll use the version from this paper[19] by Tony Hoare in 1985.

> In ancient times, a wealthy philanthropist endowed a College to accommodate five eminent philosophers. Each philosopher had a room in which he could engage in his professional activity of thinking; there was also a common dining room, furnished with a circular table, surrounded by five chairs, each labelled by the name of the philosopher who was to sit in it. They sat anticlockwise around the table. To the left of each philosopher there was laid a golden fork, and in the centre stood a large bowl of spaghetti, which was constantly replenished. A philosopher was expected to spend most of his time thinking; but when he felt hungry, he went to the dining room, sat down in his own chair, picked up his own fork on his left, and plunged it into the spaghetti. But such is the tangled nature of spaghetti that a second fork is required to carry it to the mouth. The philosopher therefore had also to pick up the fork on his right. When we was finished he would put down both his forks, get up from his chair, and continue thinking. Of course, a fork can be used by only one philosopher at a time. If the other philosopher wants it, he just has to wait until the fork is available again.

This classic problem shows off a few different elements of concurrency. The reason is that it's actually slightly tricky to implement: a simple implementation can deadlock. For example, let's consider a simple algorithm that would solve this problem:

1. A philosopher picks up the fork on their left.
2. They then pick up the fork on their right.
3. They eat.
4. They return the forks.

Now, let's imagine this sequence of events:

---

[19]http://www.usingcsp.com/cspbook.pdf

1. Philosopher 1 begins the algorithm, picking up the fork on their left.
2. Philosopher 2 begins the algorithm, picking up the fork on their left.
3. Philosopher 3 begins the algorithm, picking up the fork on their left.
4. Philosopher 4 begins the algorithm, picking up the fork on their left.
5. Philosopher 5 begins the algorithm, picking up the fork on their left.
6. ... ? All the forks are taken, but nobody can eat!

There are different ways to solve this problem. We'll get to our solution in the tutorial itself. For now, let's get started modelling the problem itself. We'll start with the philosophers:

```rust
struct Philosopher {
    name: String,
}

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

fn main() {
    let p1 = Philosopher::new("Baruch Spinoza");
    let p2 = Philosopher::new("Gilles Deleuze");
    let p3 = Philosopher::new("Karl Marx");
    let p4 = Philosopher::new("Friedrich Nietzsche");
    let p5 = Philosopher::new("Michel Foucault");
}
```

Here, we make a struct (section 5.12, page 186) to represent a philosopher. For now, a name is all we need. We choose the String (section 5.18, page 203) type for the name, rather than &str. Generally speaking, working with a type which owns its data is easier than working with one that uses references.

Let's continue:

```rust
impl Philosopher {
    fn new(name: &str) -> Philosopher {
        Philosopher {
            name: name.to_string(),
        }
    }
}
```

This impl block lets us define things on Philosopher structs. In this case, we define an 'associated function' called new. The first line looks like this:

```rust
fn new(name: &str) -> Philosopher {
```

We take one argument, a `name`, of type `&str`. This is a reference to another string. It returns an instance of our `Philosopher` struct.

```rust
Philosopher {
    name: name.to_string(),
}
```

This creates a new `Philosopher`, and sets its `name` to our `name` argument. Not just the argument itself, though, as we call `.to_string()` on it. This will create a copy of the string that our `&str` points to, and give us a new `String`, which is the type of the `name` field of `Philosopher`.

Why not accept a `String` directly? It's nicer to call. If we took a `String`, but our caller had a `&str`, they'd have to call this method themselves. The downside of this flexibility is that we *always* make a copy. For this small program, that's not particularly important, as we know we'll just be using short strings anyway.

One last thing you'll notice: we just define a `Philosopher`, and seemingly don't do anything with it. Rust is an 'expression based' language, which means that almost everything in Rust is an expression which returns a value. This is true of functions as well, the last expression is automatically returned. Since we create a new `Philosopher` as the last expression of this function, we end up returning it.

This name, `new()`, isn't anything special to Rust, but it is a convention for functions that create new instances of structs. Before we talk about why, let's look at `main()` again:

```rust
fn main() {
    let p1 = Philosopher::new("Baruch Spinoza");
    let p2 = Philosopher::new("Gilles Deleuze");
    let p3 = Philosopher::new("Karl Marx");
    let p4 = Philosopher::new("Friedrich Nietzsche");
    let p5 = Philosopher::new("Michel Foucault");
}
```

Here, we create five variable bindings with five new philosophers. These are my favorite five, but you can substitute anyone you want. If we *didn't* define that `new()` function, it would look like this:

```rust
fn main() {
    let p1 = Philosopher { name: "Baruch Spinoza".to_string() };
    let p2 = Philosopher { name: "Gilles Deleuze".to_string() };
    let p3 = Philosopher { name: "Karl Marx".to_string() };
    let p4 = Philosopher { name: "Friedrich Nietzche".to_string() };
    let p5 = Philosopher { name: "Michel Foucault".to_string() };
}
```

That's much noisier. Using `new` has other advantages too, but even in this simple case, it ends up being nicer to use.

Now that we've got the basics in place, there's a number of ways that we can tackle the broader problem here. I like to start from the end first: let's set up a way for each philosopher to finish eating. As a tiny step, let's make a method, and then loop through all the philosophers, calling it:

```rust
struct Philosopher {
  name: String,
}

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

  fn eat(&self) {
    println!("{} is done eating.", self.name);
  }
}

fn main() {
  let philosophers = vec![
    Philosopher::new("Baruch Spinoza"),
    Philosopher::new("Gilles Deleuze"),
    Philosopher::new("Karl Marx"),
    Philosopher::new("Friedrich Nietzsche"),
    Philosopher::new("Michel Foucault"),
  ];

  for p in &philosophers {
    p.eat();
  }
}
```

Let's look at main() first. Rather than have five individual variable bindings for our philosophers, we make a Vec<T> of them instead. Vec<T> is also called a 'vector', and it's a growable array type. We then use a for (section 5.6, page 164) loop to iterate through the vector, getting a reference to each philosopher in turn.

In the body of the loop, we call p.eat(), which is defined above:

```rust
fn eat(&self) {
  println!("{} is done eating.", self.name);
}
```

In Rust, methods take an explicit self parameter. That's why eat() is a method, but new is an associated function: new() has no self. For our first version of eat(), we just print out the name of the

philosopher, and mention they're done eating. Running this program should give you the following output:

```
Baruch Spinoza is done eating.
Gilles Deleuze is done eating.
Karl Marx is done eating.
Friedrich Nietzsche is done eating.
Michel Foucault is done eating.
```

Easy enough, they're all done! We haven't actually implemented the real problem yet, though, so we're not done yet!

Next, we want to make our philosophers not just finish eating, but actually eat. Here's the next version:

```rust
use std::thread;

struct Philosopher {
    name: String,
}

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

    fn eat(&self) {
        println!("{} is eating.", self.name);

        thread::sleep_ms(1000);

        println!("{} is done eating.", self.name);
    }
}

fn main() {
    let philosophers = vec![
        Philosopher::new("Baruch Spinoza"),
        Philosopher::new("Gilles Deleuze"),
        Philosopher::new("Karl Marx"),
        Philosopher::new("Friedrich Nietzsche"),
        Philosopher::new("Michel Foucault"),
    ];
```

```rust
    for p in &philosophers {
        p.eat();
    }
}
```

Just a few changes. Let's break it down.

```rust
use std::thread;
```

use brings names into scope. We're going to start using the thread module from the standard library, and so we need to use it.

```rust
    fn eat(&self) {
        println!("{} is eating.", self.name);

        thread::sleep_ms(1000);

        println!("{} is done eating.", self.name);
    }
```

We now print out two messages, with a sleep_ms() in the middle. This will simulate the time it takes a philosopher to eat.

If you run this program, You should see each philosopher eat in turn:

```
Baruch Spinoza is eating.
Baruch Spinoza is done eating.
Gilles Deleuze is eating.
Gilles Deleuze is done eating.
Karl Marx is eating.
Karl Marx is done eating.
Friedrich Nietzsche is eating.
Friedrich Nietzsche is done eating.
Michel Foucault is eating.
Michel Foucault is done eating.
```

Excellent! We're getting there. There's just one problem: we aren't actually operating in a concurrent fashion, which is a core part of the problem!

To make our philosophers eat concurrently, we need to make a small change. Here's the next iteration:

```rust
use std::thread;

struct Philosopher {
```

```rust
    name: String,
}

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

    fn eat(&self) {
        println!("{} is eating.", self.name);

        thread::sleep_ms(1000);

        println!("{} is done eating.", self.name);
    }
}

fn main() {
    let philosophers = vec![
        Philosopher::new("Baruch Spinoza"),
        Philosopher::new("Gilles Deleuze"),
        Philosopher::new("Karl Marx"),
        Philosopher::new("Friedrich Nietzsche"),
        Philosopher::new("Michel Foucault"),
    ];

    let handles: Vec<_> = philosophers.into_iter().map(|p| {
        thread::spawn(move || {
            p.eat();
        })
    }).collect();

    for h in handles {
        h.join().unwrap();
    }
}
```

All we've done is change the loop in main(), and added a second one! Here's the first change:

```rust
let handles: Vec<_> = philosophers.into_iter().map(|p| {
    thread::spawn(move || {
        p.eat();
    })
}).collect();
```

While this is only five lines, they're a dense five. Let's break it down.

```
let handles: Vec<_> =
```

We introduce a new binding, called `handles`. We've given it this name because we are going to make some new threads, and that will return some handles to those threads that let us control their operation. We need to explicitly annotate the type here, though, due to an issue we'll talk about later. The `_` is a type placeholder. We're saying "`handles` is a vector of something, but you can figure out what that something is, Rust."

```
philosophers.into_iter().map(|p| {
```

We take our list of philosophers and call `into_iter()` on it. This creates an iterator that takes ownership of each philosopher. We need to do this to pass them to our threads. We take that iterator and call `map` on it, which takes a closure as an argument and calls that closure on each element in turn.

```
  thread::spawn(move || {
    p.eat();
  })
```

Here's where the concurrency happens. The `thread::spawn` function takes a closure as an argument and executes that closure in a new thread. This closure needs an extra annotation, `move`, to indicate that the closure is going to take ownership of the values it's capturing. Primarily, the `p` variable of the `map` function.

Inside the thread, all we do is call `eat()` on `p`.

```
}).collect();
```

Finally, we take the result of all those `map` calls and collect them up. `collect()` will make them into a collection of some kind, which is why we needed to annotate the return type: we want a `Vec<T>`. The elements are the return values of the `thread::spawn` calls, which are handles to those threads. Whew!

```
for h in handles {
  h.join().unwrap();
}
```

At the end of `main()`, we loop through the handles and call `join()` on them, which blocks execution until the thread has completed execution. This ensures that the threads complete their work before the program exits.

If you run this program, you'll see that the philosophers eat out of order! We have multi-threading!

Gilles Deleuze is eating.
Gilles Deleuze is done eating.
Friedrich Nietzsche is eating.
Friedrich Nietzsche is done eating.
Michel Foucault is eating.
Baruch Spinoza is eating.
Baruch Spinoza is done eating.
Karl Marx is eating.
Karl Marx is done eating.
Michel Foucault is done eating.

But what about the forks? We haven't modeled them at all yet.

To do that, let's make a new struct:

```rust
use std::sync::Mutex;

struct Table {
    forks: Vec<Mutex<()>>,
}
```

This Table has an vector of Mutexes. A mutex is a way to control concurrency: only one thread can access the contents at once. This is exactly the property we need with our forks. We use an empty tuple, (), inside the mutex, since we're not actually going to use the value, just hold onto it.

Let's modify the program to use the Table:

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

struct Philosopher {
    name: String,
    left: usize,
    right: usize,
}

impl Philosopher {
    fn new(name: &str, left: usize, right: usize) -> Philosopher {
        Philosopher {
            name: name.to_string(),
            left: left,
            right: right,
        }
    }

    fn eat(&self, table: &Table) {
        let _left = table.forks[self.left].lock().unwrap();
```

```rust
    let _right = table.forks[self.right].lock().unwrap();

    println!("{} is eating.", self.name);

    thread::sleep_ms(1000);

    println!("{} is done eating.", self.name);
  }
}

struct Table {
  forks: Vec<Mutex<()>>,
}

fn main() {
  let table = Arc::new(Table { forks: vec![
    Mutex::new(()),
    Mutex::new(()),
    Mutex::new(()),
    Mutex::new(()),
    Mutex::new(()),
  ]});

  let philosophers = vec![
    Philosopher::new("Baruch Spinoza", 0, 1),
    Philosopher::new("Gilles Deleuze", 1, 2),
    Philosopher::new("Karl Marx", 2, 3),
    Philosopher::new("Friedrich Nietzsche", 3, 4),
    Philosopher::new("Michel Foucault", 0, 4),
  ];

  let handles: Vec<_> = philosophers.into_iter().map(|p| {
    let table = table.clone();

    thread::spawn(move || {
      p.eat(&table);
    })
  }).collect();

  for h in handles {
    h.join().unwrap();
  }
}
```

Lots of changes! However, with this iteration, we've got a working program. Let's go over the details:

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

We're going to use another structure from the std::sync package: Arc<T>. We'll talk more about it when we use it.

```
struct Philosopher {
    name: String,
    left: usize,
    right: usize,
}
```

We need to add two more fields to our Philosopher. Each philosopher is going to have two forks: the one on their left, and the one on their right. We'll use the usize type to indicate them, as it's the type that you index vectors with. These two values will be the indexes into the forks our Table has.

```
fn new(name: &str, left: usize, right: usize) -> Philosopher {
    Philosopher {
        name: name.to_string(),
        left: left,
        right: right,
    }
}
```

We now need to construct those left and right values, so we add them to new().

```
fn eat(&self, table: &Table) {
    let _left = table.forks[self.left].lock().unwrap();
    let _right = table.forks[self.right].lock().unwrap();

    println!("{} is eating.", self.name);

    thread::sleep_ms(1000);

    println!("{} is done eating.", self.name);
}
```

We have two new lines. We've also added an argument, table. We access the Table's list of forks, and then use self.left and self.right to access the fork at that particular index. That gives us access to the Mutex at that index, and we call lock() on it. If the mutex is currently being accessed by someone else, we'll block until it becomes available.

The call to lock() might fail, and if it does, we want to crash. In this case, the error that could happen is that the mutex is 'poisoned'[20], which is what happens when the thread panics while the lock is held. Since this shouldn't happen, we just use unwrap().

---

[20]http://doc.rust-lang.org/std/sync/struct.Mutex.html#poisoning

One other odd thing about these lines: we've named the results `_left` and `_right`. What's up with that underscore? Well, we aren't planning on *using* the value inside the lock. We just want to acquire it. As such, Rust will warn us that we never use the value. By using the underscore, we tell Rust that this is what we intended, and it won't throw a warning.

What about releasing the lock? Well, that will happen when `_left` and `_right` go out of scope, automatically.

```rust
let table = Arc::new(Table { forks: vec![
    Mutex::new(()),
    Mutex::new(()),
    Mutex::new(()),
    Mutex::new(()),
    Mutex::new(()),
]});
```

Next, in `main()`, we make a new `Table` and wrap it in an `Arc<T>`. 'arc' stands for 'atomic reference count', and we need that to share our `Table` across multiple threads. As we share it, the reference count will go up, and when each thread ends, it will go back down.

```rust
let philosophers = vec![
    Philosopher::new("Baruch Spinoza", 0, 1),
    Philosopher::new("Gilles Deleuze", 1, 2),
    Philosopher::new("Karl Marx", 2, 3),
    Philosopher::new("Friedrich Nietzsche", 3, 4),
    Philosopher::new("Michel Foucault", 0, 4),
];
```

We need to pass in our `left` and `right` values to the constructors for our `Philosopher`s. But there's one more detail here, and it's *very* important. If you look at the pattern, it's all consistent until the very end. Monsieur Foucault should have `4,0` as arguments, but instead, has `0,4`. This is what prevents deadlock, actually: one of our philosophers is left handed! This is one way to solve the problem, and in my opinion, it's the simplest.

```rust
let handles: Vec<_> = philosophers.into_iter().map(|p| {
    let table = table.clone();

    thread::spawn(move || {
        p.eat(&table);
    })
}).collect();
```

Finally, inside of our `map()`/`collect()` loop, we call `table.clone()`. The `clone()` method on `Arc<T>` is what bumps up the reference count, and when it goes out of scope, it decrements the count. You'll notice we can introduce a new binding to `table` here, and it will shadow the old one. This is often used so that you don't need to come up with two unique names.

With this, our program works! Only two philosophers can eat at any one time, and so you'll get some output like this:

Gilles Deleuze is eating.
Friedrich Nietzsche is eating.
Friedrich Nietzsche is done eating.
Gilles Deleuze is done eating.
Baruch Spinoza is eating.
Karl Marx is eating.
Baruch Spinoza is done eating.
Michel Foucault is eating.
Karl Marx is done eating.
Michel Foucault is done eating.

Congrats! You've implemented a classic concurrency problem in Rust.

3.3 Rust inside other languages Rust 与其他语言

For our third project, we're going to choose something that shows off one of Rust's greatest strengths: a lack of a substantial runtime.

As organizations grow, they increasingly rely on a multitude of programming languages. Different programming languages have different strengths and weaknesses, and a polyglot stack lets you use a particular language where its strengths make sense, and use a different language where it's weak.

A very common area where many programming languages are weak is in runtime performance of programs. Often, using a language that is slower, but offers greater programmer productivity is a worthwhile trade-off. To help mitigate this, they provide a way to write some of your system in C, and then call the C code as though it were written in the higher-level language. This is called a 'foreign function interface', often shortened to 'FFI'.

Rust has support for FFI in both directions: it can call into C code easily, but crucially, it can also be called *into* as easily as C. Combined with Rust's lack of a garbage collector and low runtime requirements, this makes Rust a great candidate to embed inside of other languages when you need some extra oomph.

There is a whole chapter devoted to FFI (section 4.8, page 128) and its specifics elsewhere in the book, but in this chapter, we'll examine this particular use-case of FFI, with three examples, in Ruby, Python, and JavaScript.

3.3.1 The problem

There are many different projects we could choose here, but we're going to pick an example where Rust has a clear advantage over many other languages: numeric computing and threading.

Many languages, for the sake of consistency, place numbers on the heap, rather than on the stack. Especially in languages that focus on object-oriented programming and use garbage collection, heap allocation is the default. Sometimes optimizations can stack allocate particular numbers, but rather than relying on an optimizer to do its job, we may want to ensure that we're always using primitive number types rather than some sort of object type.

Second, many languages have a 'global interpreter lock', which limits concurrency in many situations. This is done in the name of safety, which is a positive effect, but it limits the amount of work that can be done at the same time, which is a big negative.

To emphasize these two aspects, we're going to create a little project that uses these two aspects heavily. Since the focus of the example is the embedding of Rust into the languages, rather than the problem itself, we'll just use a toy example:

> Start ten threads. Inside each thread, count from one to five million. After All ten threads are finished, print out 'done!'.

I chose five million based on my particular computer. Here's an example of this code in Ruby:

```ruby
threads = []

10.times do
 threads << Thread.new do
  count = 0

  5_000_000.times do
   count += 1
  end
 end
end

threads.each {|t| t.join }
puts "done!"
```

Try running this example, and choose a number that runs for a few seconds. Depending on your computer's hardware, you may have to increase or decrease the number.

On my system, running this program takes 2.156 seconds. And, if I use some sort of process monitoring tool, like top, I can see that it only uses one core on my machine. That's the GIL kicking in.

While it's true that this is a synthetic program, one can imagine many problems that are similar to this in the real world. For our purposes, spinning up some busy threads represents some sort of parallel, expensive computation.

3.3.2 A Rust library

Let's re-write this problem in Rust. First, let's make a new project with Cargo:

```
$ cargo new embed
$ cd embed
```

This program is fairly easy to write in Rust:

```rust
use std::thread;

fn process() {
  let handles: Vec<_> = (0..10).map(|_| {
    thread::spawn(|| {
      let mut _x = 0;
      for _ in (0..5_000_001) {
        _x += 1
      }
    })
  }).collect();

  for h in handles {
    h.join().ok().expect("Could not join a thread!");
  }
}
```

Some of this should look familiar from previous examples. We spin up ten threads, collecting them into a handles vector. Inside of each thread, we loop five million times, and add one to _x each time. Why the underscore? Well, if we remove it and compile:

```
$ cargo build
   Compiling embed v0.1.0 (file:///home/steve/src/embed)
src/lib.rs:3:1: 16:2 warning: function is never used: `process`, #[warn(dead_code)] on
⬚ by default
src/lib.rs:3 fn process() {
src/lib.rs:4     let handles: Vec<_> = (0..10).map(|_| {
src/lib.rs:5         thread::spawn(|| {
src/lib.rs:6           let mut x = 0;
src/lib.rs:7           for _ in (0..5_000_001) {
src/lib.rs:8             x += 1
          ...
src/lib.rs:6:17: 6:22 warning: variable `x` is assigned to, but never used, #[warn(unus
⬚ ed_variables)] on by default
src/lib.rs:6           let mut x = 0;
                 ^~~~~
```

That first warning is because we are building a library. If we had a test for this function, the warning would go away. But for now, it's never called.

The second is related to x versus _x. Because we never actually *do* anything with x, we get a warning about it. In our case, that's perfectly okay, as we're just trying to waste CPU cycles. Prefixing x with the underscore removes the warning.

Finally, we join on each thread.

Right now, however, this is a Rust library, and it doesn't expose anything that's callable from C. If we tried to hook this up to another language right now, it wouldn't work. We only need to make two small changes to fix this, though. The first is modify the beginning of our code:

```
#[no_mangle]
pub extern fn process() {
```

We have to add a new attribute, no_mangle. When you create a Rust library, it changes the name of the function in the compiled output. The reasons for this are outside the scope of this tutorial, but in order for other languages to know how to call the function, we need to not do that. This attribute turns that behavior off.

The other change is the pub extern. The pub means that this function should be callable from outside of this module, and the extern says that it should be able to be called from C. That's it! Not a whole lot of change.

The second thing we need to do is to change a setting in our Cargo.toml. Add this at the bottom:

```
[lib]
name = "embed"
crate-type = ["dylib"]
```

This tells Rust that we want to compile our library into a standard dynamic library. By default, Rust compiles into an 'rlib', a Rust-specific format.

Let's build the project now:

```
$ cargo build --release
   Compiling embed v0.1.0 (file:///home/steve/src/embed)
```

We've chosen cargo build --release, which builds with optimizations on. We want this to be as fast as possible! You can find the output of the library in target/release:

```
$ ls target/release/
build deps examples libembed.so native
```

That libembed.so is our 'shared object' library. We can use this file just like any shared object library written in C! As an aside, this may be embed.dll or libembed.dylib, depending on the platform.

Now that we've got our Rust library built, let's use it from our Ruby.

3.3.3 Ruby

Open up a embed.rb file inside of our project, and do this:

```
require 'ffi'

module Hello
  extend FFI::Library
  ffi_lib 'target/release/libembed.so'
  attach_function :process, [], :void
end

Hello.process

puts "done!"
```

Before we can run this, we need to install the ffi gem:

```
$ gem install ffi # this may need sudo
Fetching: ffi-1.9.8.gem (100%)
Building native extensions.  This could take a while...
Successfully installed ffi-1.9.8
Parsing documentation for ffi-1.9.8
Installing ri documentation for ffi-1.9.8
Done installing documentation for ffi after 0 seconds
1 gem installed
```

And finally, we can try running it:

```
$ ruby embed.rb
done!
$
```

Whoah, that was fast! On my system, this took 0.086 seconds, rather than the two seconds the pure Ruby version took. Let's break down this Ruby code:

```
require 'ffi'
```

We first need to require the ffi gem. This lets us interface with our Rust library like a C library.

```
module Hello
  extend FFI::Library
  ffi_lib 'target/release/libembed.so'
```

The ffi gem's authors recommend using a module to scope the functions we'll import from the shared library. Inside, we extend the necessary FFI::Library module, and then call ffi_lib to load up our shared object library. We just pass it the path that our library is stored, which as we saw before, is target/release/libembed.so.

```
attach_function :process, [], :void
```

The attach_function method is provided by the FFI gem. It's what connects our process() function in Rust to a Ruby function of the same name. Since process() takes no arguments, the second parameter is an empty array, and since it returns nothing, we pass :void as the final argument.

```
Hello.process
```

This is the actual call into Rust. The combination of our module and the call to attach_function sets this all up. It looks like a Ruby function, but is actually Rust!

```
puts "done!"
```

Finally, as per our project's requirements, we print out done!.

That's it! As we've seen, bridging between the two languages is really easy, and buys us a lot of performance.

Next, let's try Python!

3.3.4 Python

Create an embed.py file in this directory, and put this in it:

```
from ctypes import cdll

lib = cdll.LoadLibrary("target/release/libembed.so")

lib.process()

print("done!")
```

Even easier! We use cdll from the ctypes module. A quick call to LoadLibrary later, and we can call process().

On my system, this takes 0.017 seconds. Speedy!

3.3.5 Node.js

Node isn't a language, but it's currently the dominant implementation of server-side JavaScript. In order to do FFI with Node, we first need to install the library:

```
$ npm install ffi
```

After that installs, we can use it:

```
var ffi = require('ffi');

var lib = ffi.Library('target/release/libembed', {
  'process': [ 'void', [] ]
});

lib.process();

console.log("done!");
```

It looks more like the Ruby example than the Python example. We use the ffi module to get access to ffi.Library(), which loads up our shared object. We need to annotate the return type and argument types of the function, which are 'void' for return, and an empty array to signify no arguments. From there, we just call it and print the result.

On my system, this takes a quick 0.092 seconds.

3.3.6 Conclusion

As you can see, the basics of doing this are *very* easy. Of course, there's a lot more that we could do here. Check out the FFI (section 4.8, page 128) chapter for more details.

4

Effective Rust 高效的 Rust

So you've learned how to write some Rust code. But there's a difference between writing *any* Rust code and writing *good* Rust code.

到现在为止，你已经学过如何写一些 rust 语言的代码了。但是，在普通的 rust 代码和好的 rust 代码之间是有些大不同的

This section consists of relatively independent tutorials which show you how to take your Rust to the next level. Common patterns and standard library features will be introduced. Read these sections in any order of your choosing.

本章节有相对独立的教程，能够向你展示如何让你的 rust 语言水平下一个更高阶段。通用模式和标准库特性将会被介绍到。你可以以任意顺序阅读本章节。

4.1 The Stack and the Heap 栈和堆

As a systems language, Rust operates at a low level. If you're coming from a high-level language, there are some aspects of systems programming that you may not be familiar with. The most important one is how memory works, with a stack and a heap. If you're familiar with how C-like languages use stack allocation, this chapter will be a refresher. If you're not, you'll learn about this more general concept, but with a Rust-y focus.

作为一门系统语言，rust 在系统中一个比较低的层次运行。如果你来自于高级语言，有几个系统编程方面的内容可能你不熟悉。最重要的事情是内存是如何通过栈和堆来进行工作的。如果你熟悉类似 c 语言的堆栈分配机制，本章将会重温一遍。如果你不熟悉，你将学到更多的通用知识，但是是站在 rust 语言的角度上

4.1.1   Memory management 内存管理

These two terms are about memory management. The stack and the heap are abstractions that help you determine when to allocate and deallocate memory.

有两种内存管理方式，栈和堆是抽象的，用于帮助你决定什么时候分配和释放内存。

Here's a high-level comparison:

这里有一个比较好的解释:

The stack is very fast, and is where memory is allocated in Rust by default. But the allocation is local to a function call, and is limited in size. The heap, on the other hand, is slower, and is explicitly allocated by your program. But it's effectively unlimited in size, and is globally accessible.

栈的管理是非常快的，尤其是在内存有 rust 语言默认进行内存分配的时候。但是这些分配是局部函数方法调用，且会收到占用空间大小的限制。另一个管理方式堆的管理相对比较就慢多了，并且由你的程序进行明确分配。优点是，它不受限于占用空间的大小，并且能够被全局访问。

4.1.2   The Stack 栈

Let's talk about this Rust program:

让我们看看下面的 rust 程序

```
fn main() {
  let x = 42;
}
```

This program has one variable binding, x. This memory needs to be allocated from somewhere. Rust 'stack allocates' by default, which means that basic values 'go on the stack'. What does that mean?

这个程序有一个变量 x. 内存需要从某一处开始分配。在 rust 中默认的 "栈分配" 有一个基本的叫法 '入栈'。那这意味着什么呢?

Well, when a function gets called, some memory gets allocated for all of its local variables and some other information. This is called a 'stack frame', and for the purpose of this tutorial, we're going to ignore the extra information and just consider the local variables we're allocating. So in this case, when main() is run, we'll allocate a single 32-bit integer for our stack frame.This is automatically handled for you, as you can see, we didn't have to write any special Rust code or anything.

嗯，当一个函数被调用，一些内存就会被分配给所有的局部变量和一些其他信息。这被称作 "栈帧"。介于本教程的意图，我们讲忽略掉一些额外的信息，只考虑所有的局部变量。因此，在这种情况下当 main() 运行的时候，我们将会为 "栈帧" 分配一个 32 位的整数。这个内存分配是自动处理的，正如你所见到的，我们不需要写任何特殊的 Rust 代码和做其他事情。

When the function is over, its stack frame gets deallocated. This happens automatically, we didn't have to do anything special here.

当一个函数执行结束，他的栈帧就会被回收，这也是自动发生的，我们不需要做任何特殊的操作。

That's all there is for this simple program. The key thing to understand here is that stack allocation is very, very fast. Since we know all the local variables we have ahead of time, we can grab the memory all at once. And since we'll throw them all away at the same time as well, we can get rid of it very fast too.

这就是这个简单的程序的全部了。我们需要弄清楚的关键事情就是栈管理操作是非常非常快的。一旦我们提前知道所有的变量，我们就能够一次抓取到所有的内存信息。同样，一旦我们在这个过程中较好的丢掉他们，我们也就能够快速的清除掉它。

The downside is that we can't keep values around if we need them for longer than a single function. We also haven't talked about what that name, 'stack' means. To do that, we need a slightly more complicated example:

缺点是我们不能够保持变量存在，当我们不只在一个函数中需要他们。我们同样还没有讨论 'stack' 到底是什么。为了说明这一个问题，我们需要一个更复杂些的例子。

```
fn foo() {
  let y = 5;
  let z = 100;
}

fn main() {
  let x = 42;

  foo();
}
```

This program has three variables total: two in foo(), one in main(). Just as before, when main() is called, a single integer is allocated for its stack frame. But before we can show what happens when foo() is called, we need to visualize what's going on with memory. Your operating system presents a view of memory to your program that's pretty simple: a huge list of addresses, from 0 to a large number, representing how much RAM your computer has. For example, if you have a gigabyte of RAM, your addresses go from 0 to 1,073,741,824. That number comes from 230, the number of bytes in a gigabyte.

这个程序总共有三个变量：2 个在 foo() 中，一个在 main() 中。就像签一个例子，当 main() 被执行时，一个整数被分配在栈帧上。但是，在我们展示当 foo() 被执行时发生什么前，我们需要想象一下在内存中发生了什么。你的操作系统呈现程序的内存视图是非常简单的：一个从 0 到一个很大的数字的很长的地址列表用来表示你的电脑中有多少内存空间。例如，如果你有一个 1G 的内存，你的地址列表将是从 0 到 1,073,741,824。这个数字是 230，1GB 所表示的字节数。

This memory is kind of like a giant array: addresses start at zero and go up to the final number. So here's a diagram of our first stack frame:

内存像是一个巨大的数组，地址从 0 开始走到最终数目。所以我们第一个栈帧的图表如下

Address 地址	Name 名称	Value 值
0	x	42

We've got x located at address 0, with the value 42. **我们在地址** 0 **处得到** x, **他的值是** 42.

When foo() is called, a new stack frame is allocated:

当 foo() **被执行时，一下新的栈帧被分配出来：**

Address 地址	Name 名称	Value 值
2	z	100
1	y	5
0	x	42

Because 0 was taken by the first frame, 1 and 2 are used for foo()'s stack frame. It grows upward, the more functions we call.

因为 0 **被第一帧占用，所以** 1 **和** 2 **被用于** foo() **的栈帧。我们调用的函数越多，他增长的越快。**

There's some important things we have to take note of here. The numbers 0, 1, and 2 are all solely for illustrative purposes, and bear no relationship to the actual numbers the computer will actually use. In particular, the series of addresses are in reality going to be separated by some number of bytes that separate each address, and that separation may even exceed the size of the value being stored.

这里有些特别重要的地方需要我们注意。数字 0,1 **和** 2 **只是用来表名我们的意图，和电脑中实际使用的真实数目没有任何关系。特别是在实际中，系列中的地址是被一些字节数分离开的，并且有些分离甚至超过了数据被分配的大小。**

After foo() is over, its frame is deallocated:

在 foo() **执行结束后，他的栈帧被释放了：**

Address	Name	Value
0	x	42

And then, after main(), even this last value goes away. Easy!

然后， main() **执行结束后，连最终的数据都被清空了。真简单**!

It's called a 'stack' because it works like a stack of dinner plates: the first plate you put down is the last plate to pick back up. Stacks are sometimes called 'last in, first out queues' for this reason, as the last value you put on the stack is the first one you retrieve from it.

它被称之为"栈"是因为它工作起来向是一摞餐盘

Let's try a three-deep example:

让我们尝试一下三层嵌套的例子：

```rust
fn bar() {
    let i = 6;
```

```
}

fn foo() {
  let a = 5;
  let b = 100;
  let c = 1;

  bar();
}

fn main() {
  let x = 42;

  foo();
}
```

Okay, first, we call main():

OK，首先我们执行 main():

Address	Name	Value
0	x	42

Next up, main() calls foo():

下一步，mian() 调用 foo():

Address	Name	Value
3	c	1
2	b	100
1	a	5
0	x	42

And then foo() calls bar():

然后，foo() 调用 bar():

Address	Name	Value
4	i	6
3	c	1
2	b	100
1	a	5
0	x	42

Whew! Our stack is growing tall. **噢，我们的栈长高了。**

After bar() is over, its frame is deallocated, leaving just foo() and main(): **当 bar() 调用结束，他的栈帧被释放，只留下 foo() 和 main()：**

Address	Name	Value
3	c	1
2	b	100
1	a	5
0	x	42

And then foo() ends, leaving just main():

然后 foo() 运行结束，只剩下 main()：

Address	Name	Value
0	x	42

And then we're done. Getting the hang of it? It's like piling up dishes: you add to the top, you take away from the top.

然后，运行结束。懂得它的窍门了吗？这就像我们堆放的零食：你放在顶部，然后从顶部拿走。

4.1.3   The Heap 堆

Now, this works pretty well, but not everything can work like this. Sometimes,you need to pass some memory between different functions, or keep it alive for longer than a single function's execution. For this, we can use the heap.

现在，它运行的很好，但是并不是所有的事情都能够像那样运行。有些时候，你需要在几个不同的函数中传递内存，或者保持它存在时间比函数执行时间还要长久。为达到这个目的，我们可以使用堆。

In Rust, you can allocate memory on the heap with the Box<T> type[1].

在 Rust 语言中，你能够以盒模型 (Box<T> type[2]) 的方式在堆上分配内存。

Here's an example:

下面是一个例子

```rust
fn main() {
    let x = Box::new(5);
    let y = 42;
}
```

---

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

Here's what happens in memory when main() is called:

当 main() 被执行时，内存发生如下变化:

Address	Name	Value
1	y	42
0	x	??????

We allocate space for two variables on the stack. y is 42, as it always has been, but what about x? Well, x is a Box<i32>, and boxes allocate memory on the heap. The actual value of the box is a structure which has a pointer to 'the heap'. When we start executing the function, and Box::new() is called,it allocates some memory for the heap, and puts 5 there. The memory now lookslike this:

我们在栈上分配了两个变量的空间。y 是 42, 它永远都是，但是 x 呢? 好吧，x 是一个盒模型 Box<i32>, 盒模型分配的内存是在堆上的。盒模型实际上是一个结构体，它有一个指向 "堆" 的指针。当我们开始执行函数的时候，Box::new() 被调用了，它在堆上分配了一些内存，并把 5 放在那些内存地址上。现在内存看起来像是这样:

Address	Name	Value
230		5
...	...	...
1	y	42
0	x	230

We have 230 in our hypothetical computer with 1GB of RAM. And since our stack grows from zero, the easiest place to allocate memory is from the other end. So our first value is at the highest place in memory. And the value of the struct at x has a raw pointer (section 5.36, page 271) to the place we've allocated on the heap, so the value of x is 230, the memory location we've asked for.

在我们假定的电脑的 1G 内存上我们有 230 个地址。一旦我们的栈从 0 开始增长，最容易的方式是从另一端分配内存。所以，我们的第一个值时在内存地址最大的地方。x 结构体的值有一个原始指针 (section 5.36, page 271) 指向我们在堆上申请的地址，所以 x 结构体的值是我们申请的内存地址——230。

We haven't really talked too much about what it actually means to allocate and deallocate memory in these contexts. Getting into very deep detail is out of the scope of this tutorial, but what's important to point out here is that the heap isn't just a stack that grows from the opposite end. We'll have an example of this later in the book, but because the heap can be allocated and freed in any order, it can end up with 'holes'. Here's a diagram of the memory layout of a program which has been running for a while now:

我们还没有真正探讨实际意义上的上下文内存分配和释放问题。谈论更深层次的细节超出了本书的内容范围，但是需要指出的是，堆并不总是从另一端开始的栈。稍后，本书中会有一个关于这个的例子，但是由于堆可以以任何顺序分配和释放，他可能产生漏洞。这里有一个正在运行的程序的内存层的视图:

Address	Name	Value
230		5
(230) - 1		
(230) - 2		
(230) - 3		42
...	...	...
3	y	(230) - 3
2	y	42
1	y	42
0	x	230

In this case, we've allocated four things on the heap, but deallocated two of them. There's a gap between 230 and (230) - 3 which isn't currently being used. The specific details of how and why this happens depends on what kind of strategy you use to manage the heap. Different programs can use different 'memory allocators', which are libraries that manage this for you.Rust programs use jemalloc[3] for this purpose.

在这种情况下，我们分配了四个结构体在堆上，并且释放了他们中的两个。在 230 和 (230) - 3 之间有一个间隙，在当前的内存中没有被使用。为什么会产生这么特殊的情况取决于你用何种策略来管理堆。不同的程序使用不同 "内存分配器"——就是为你管理堆的库。Rust 程序使用 jemalloc[4] 来达到这一目的。

Anyway, back to our example. Since this memory is on the heap, it can stay alive longer than the function which allocates the box. In this case, however, it doesn't [ˆmoving] When the function is over, we need to free the stack frame for main(). Box<T>, though, has a trick up its sleeve: Drop (section 5.21, page 215). The implementation of Drop for Box deallocates the memory that was allocated when it was created. Great! So when x goes away, it first frees the memory allocated on the heap:

让我们回到我们的例子中来。一旦内存在堆上，它就能够比分配盒模型空间的函数存在时间更长。然而在这种情况下，在函数运行结束的时候，它不能够移除，我们需要为 main().Box<T> 释放栈帧，不过有一个诀窍来解决他的遗留：Drop (section 5.21, page 215)。当为 Box 执行 Drop 的操作时，释放了在它被创建时分配的内存。强大！所以，当 x 被注销时，首先释放的是在堆上分配的内存空间：

Address	Name	Value
1	y	42
0	x	??????

And then the stack frame goes away, freeing all of our memory.

然后栈帧被移除了，所有的内存都被释放了。

---

[3]http://www.canonware.com/jemalloc/

[4]http://www.canonware.com/jemalloc/

4.1.4 Arguments and borrowing 参数和引用

We've got some basic examples with the stack and the heap going, but what about function arguments and borrowing? Here's a small Rust program:

我们已经理解了栈和堆管理的基本例子，但是对函数的参数和引用了解到了什么? 这里有一个简单的 Rust 程序:

```rust
fn foo(i: &i32) {
  let z = 42;
}

fn main() {
  let x = 5;
  let y = &x;

  foo(y);
}
```

When we enter main(), memory looks like this:

当我们进入 main(), 内存视图看起来是这样子的:

Address	Name	Value
1	y	0
0	x	5

x is a plain old 5, and y is a reference to x. So its value is the memory location that x lives at, which in this case is 0.

x 被解释为 5，y 是 x 的一个引用。所以 y 的值是 x 存在时在内存中的地址——在本例子中是 0。

What about when we call foo(), passing y as an argument?

当我们调用了 foo() 后，将 y 作为参数传递进去后，又是一种什么情况?

Address	Name	Value
3	z	42
2	i	0
1	y	0
0	x	5

Stack frames aren't just for local bindings, they're for arguments too. So in this case, we need to have both i, our argument, and z, our local variable binding. i is a copy of the argument, y. Since y's value is 0, so is i's.

栈帧并不只是分配给局部变量的,它们也会为参数分配的。所以在本案例中,我们需要对 i——**参数,和** z——**本地局部变量分配内存栈帧**

This is one reason why borrowing a variable doesn't deallocate any memory: the value of a reference is just a pointer to a memory location. If we got rid of the underlying memory, things wouldn't work very well.

这就是为什么引用一个变量不会释放任何内存的一个原因:引用的值只是指向一个内存地址的指针。如果我们脱离了基础的内存,程序将不能够很好的运行。

4.1.5   A complex example 一个复杂的例子

Okay, let's go through this complex program step-by-step:

好了,让我们一步步进入这个复杂的程序

```rust
fn foo(x: &i32) {
  let y = 10;
  let z = &y;

  baz(z);
  bar(x, z);
}

fn bar(a: &i32, b: &i32) {
  let c = 5;
  let d = Box::new(5);
  let e = &d;

  baz(e);
}

fn baz(f: &i32) {
  let g = 100;
}

fn main() {
  let h = 3;
  let i = Box::new(20);
  let j = &h;

  foo(j);
}
```

First, we call main():

首先,我们调用 main()**:**

Address	Name	Value
230		20
...	...	...
2	j	0
1	i	230
0	h	3

We allocate memory for j, i, and h. i is on the heap, and so has a value pointing there.

我们为 j,i 和 h 分配了内存。i 被分配在堆上，所以有一个指针在在那里。

Next, at the end of main(), foo() gets called:

下一步，在 main() 的最后,foo() 被调用了：

Address	Name	Value
230		20
...	...	...
5	z	4
4	y	10
3	x	0
2	j	0
1	i	230
0	h	3

Space gets allocated for x, y, and z. The argument x has the same value as j, since that's what we passed it in. It's a pointer to the 0 address,since j points at h.

内存空间被分配给 x,y 和 z。自我们将 j 传递进来后，参数 x 有一个与 j 相同的值。它是指向编号为 0 的内存地址，因为 j 指向 h。

Next, foo() calls baz(), passing z:

然后，foo() 调用 baz()，传参 z：

Address	Name	Value
230		20
...	...	...
7	g	100
6	f	4
5	z	4
4	y	10
3	x	0
2	j	0
1	i	230
0	h	3

We've allocated memory for f and g. baz() is very short, so when it's over, we get rid of its stack frame:

我们为 f 和 g 分配了内存。函数 baz() 非常短，所以当他结束的时候，我们清除了它的栈帧。

Address	Name	Value
230	...	20
...	...	...
5	z	4
4	y	10
3	x	0
2	j	0
1	i	230
0	h	3

Next, foo() calls bar() with x and z:

然后是，foo() 调用了 bar()，使用 x 和 z 作为参数：

Address	Name	Value
230	...	20
(230) - 1	...	5
...	...	...
10	e	9
9	d	(230) - 1
8	c	5
7	b	4
6	a	0
5	z	4
4	y	10
3	x	0
2	j	0
1	i	230
0	h	3

We end up allocating another value on the heap, and so we have to subtract one from 230. It's easier to just write that than 1,073,741,823. In any case, we set up the variables as usual.

我们不能够在堆上分配另一个值，所以我们必须在内存地址 230-1 处分配。这么标记只是因为它比 1,073,741,823 好写。无论任何情况，我们像平常一样建立了变量。

At the end of bar(), it calls baz():

在函数 bar() 的最后，它调用了函数 baz()：

Address	Name	Value
230		20
(230) - 1		5
...	...	...
12	g	100
11	f	4
10	e	9
9	d	(230) - 1
8	c	5
7	b	4
6	a	0
5	z	4
4	y	10
3	x	0
2	j	0
1	i	230
0	h	3

With this, we're at our deepest point! Whew! Congrats for following along this far.

通过这个，我们到达了最深层！恭喜你跟随着走了这么远。

After baz() is over, we get rid of f and g:

baz() 结束后，我们清除了 f 和 g:

Address	Name	Value
230		20
(230) - 1		5
...	...	...
10	e	9
9	d	(230) - 1
8	c	5
7	b	4
6	a	0
5	z	4
4	y	10
3	x	0
2	j	0
1	i	230
0	h	3

Next, we return from bar(). d in this case is a Box<T>, so it also frees what it points to: (230) - 1.

然后，我们返回到函数 bar()。在本案例中 d 是一个盒模型 Box<T>, 所以它指向的内存地址 (230) - 1

同样被释放了。

Address	Name	Value
230		20
...	...	...
5	z	4
4	y	10
3	x	0
2	j	0
1	i	230
0	h	3

And after that, foo() returns:

之后，返回到了函数 foo()**：**

Address	Name	Value
230		20
...	...	...
2	j	0
1	i	230
0	h	3

And then, finally, main(), which cleans the rest up. When i is Dropped,it will clean up the last of the heap too.

最终，main() **开始清理释放。当** i **被** Drop **的时候，它同样将清理掉最后的堆。**

4.1.6   What do other languages do? 其他的语言是怎么做的?

Most languages with a garbage collector heap-allocate by default. This means that every value is boxed. There are a number of reasons why this is done, but they're out of scope for this tutorial. There are some possible optimizations that don't make it true 100% of the time, too. Rather than relying on the stack and Drop to clean up memory, the garbage collector deals with the heap instead.

大多数的语言默认有一个垃圾回收器。这意味着，每一个值都是一个盒模型。这样做有许多原因，他们超出了本教程的范围，故不在此赘述。同样，这些可能的优化并不总是保证 100% **正确。但是垃圾回收器使用堆管理而不是依赖栈和** "Drop" **来清理内存。**

4.1.7   Which to use? 到底该使用哪一种

So if the stack is faster and easier to manage, why do we need the heap? A big reason is that Stack-allocation alone means you only have LIFO semantics for reclaiming storage. Heap-allocation is

strictly more general, allowing storage to be taken from and returned to the pool in arbitrary order, but at a complexity cost.

那么尽管栈操作比较快，且更容易管理，我们为什么还需要堆？一个很大方面的原因是栈分配意味着你只有先进后出，后进先出的方式来声明存储空间。堆分配更加通用一些，允许内存空间以任意顺序方式分配和返回到内存池，但是这需要付出一个复杂度大小的代价。

Generally, you should prefer stack allocation, and so, Rust stack-allocates by default. The LIFO model of the stack is simpler, at a fundamental level. This has two big impacts: runtime efficiency and semantic impact.

通常，我们选择栈分配，同样 Rust 语言也是默认使用栈分配。在系统基础层面上，栈的先进后出的方式是比较简单的。这会产生两个方面的影响：执行效率和语义对撞。

Runtime Efficiency.

Managing the memory for the stack is trivial: The machine just increments or decrements a single value, the so-called "stack pointer".Managing memory for the heap is non-trivial: heap-allocated memory is freed at arbitrary points, and each block of heap-allocated memory can be of arbitrary size, the memory manager must generally work much harder to identify memory for reuse.

栈的内存管理是琐碎的：机器只能够递增或者递减一个值，这个被称作 "堆栈指针"。堆的内存管理是不琐碎的：堆分配内存可以被随意的释放，并且堆分配内存的每一块都可以是任意大小，内存管理理者必须费力运行，以保证内存确认可用的内存被重新使用

If you'd like to dive into this topic in greater detail, this paper[5] is a great introduction.

如果你想喜欢本话题的更深层次，this paper[6] 是一个非常不错的介绍。

Semantic impact

Stack-allocation impacts the Rust language itself, and thus the developer's mental model. The LIFO semantics is what drives how the Rust language handles automatic memory management. Even the deallocation of a uniquely-owned heap-allocated box can be driven by the stack-based LIFO semantics, as discussed throughout this chapter. The flexibility (i.e. expressiveness) of non LIFO-semantics means that in general the compiler cannot automatically infer at compile-time where memory should be freed; it has to rely on dynamic protocols,potentially from outside the language itself, to drive deallocation (reference counting, as used by Rc<T> and Arc<T>, is one example of this).

栈分配影响着 Rust 语言本身和开发者的心智模型。后进先出的哲学驱动着 Rust 语言如何自动进行内存管理。甚至于就像本章中探讨的，独有的堆分配也是被基于栈分配的后进先出的哲学来驱动的。非后进先出的灵活性（即表现）是指在编译器在编译过程中无法自动推断某些内存应该被释放时，它不得不依赖于来字语言本身之外的动态协议，来驱动释放操作——参考：Rc<T> 和 Arc<T> 的用法就是一个关于这方面的例子。

---

[5]http://www.cs.northwestern.edu/~pdinda/icsclass/doc/dsa.pdf
[6]http://www.cs.northwestern.edu/~pdinda/icsclass/doc/dsa.pdf

When taken to the extreme, the increased expressive power of heap allocation comes at the cost of either significant runtime support (e.g. in the form of a garbage collector) or significant programmer effort (in the form of explicit memory management calls that require verification not provided by the Rust compiler).

当到了极致的时候，随着不断增加的堆分配的表现力来自于每一个重要运行时支持的成本（比如垃圾回收器的形式）或者重要成员（在 Rust 编译器提供的方法之外的明确的内存管理形式上）所花费的精力。

4.2   Testing 测试

> Program testing can be a very effective way to show the presence of bugs, butit is hopelessly inadequate for showing their absence.Edsger W. Dijkstra, "The Humble Programmer" (1972)
>
> **"程序测试是表明存在故障的非常有效的方法，但对于证明没有故障，调试是很无能为力的。" 艾兹格・迪科斯彻《谦卑的程序员》**1972

Let's talk about how to test Rust code. What we will not be talking about is the right way to test Rust code. There are many schools of thought regarding the right and wrong way to write tests. All of these approaches use the same basic tools, and so we'll show you the syntax for using them.

那么，我们来讨论下如何调试 Rust 代码。我们不会讨论调试 Rust 代码的正确方法。有很多在写测试代码的正确和错误方式方面有想法的学校。他们都在使用最基本的工具，所以我们将告诉你使用它们的方法。

4.2.1   The **test** attribute **test** 属性

At its simplest, a test in Rust is a function that's annotated with the test attribute. Let's make a new project with Cargo called adder:

简单说来，Rust 语言中的测试代码是一个函数，它使用 test 属性作为注解。现在，让我们使用 Cargo new 新建一个名为 adder 的项目：

```
$ cargo new adder
$ cd adder
```

Cargo will automatically generate a simple test when you make a new project.Here's the contents of src/lib.rs:

当你新建一个项目的时候，Cargo 将自动生成一个简单的测试. 下面是 src/lib.rs 文件的内容：

```
#[test]
fn it_works() {
}
```

Note the #[test]. This attribute indicates that this is a test function. It currently has no body. That's good enough to pass! We can run the tests with cargo test:

大家注意下 #[test]，这个属性表明这是一个测试函数。现在，它还没有内容。但是已经足够我们进行测试了。我们可以使用 cargo test 命令来运行这些测试用例：

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a
```

running 1 test **运行一条测试**
test it_works ... ok **运行测试方法it_works成功**

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured
测试结果：成功。1通过；0失败；0略过；0达标

   Doc-tests adder **文档测试**

running 0 tests   **运行0条测试**

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
测试结果：成功。0通过；0失败；0略过；0达标

Cargo compiled and ran our tests. There are two sets of output here: one for the test we wrote, and another for documentation tests. We'll talk about those later. For now, see this line:

Cargo 编译并运行测试。这里有两个输出结果：一个是我们写的测试的，另一个是文档测试。我们稍后谈论他们。现在，看着一行代码：

test it_works ... ok

Note the it_works. This comes from the name of our function:
注意这个 it_works，他来自于我们定义的方法名

```
fn it_works() {
```

We also get a summary line:
我们还得到概要信息：

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

So why does our do-nothing test pass? Any test which doesn't panic! passes, and any test that does panic! fails. Let's make our test fail:

为什么我们什么都没做，然后测试就通过了呢？任何没有 panic! 的测试通过，任何有 pannic! 的测试失败。让我们来做一个失败情况的测试吧：

```rust
#[test]
fn it_works() {
  assert!(false);
}
```

assert! is a macro provided by Rust which takes one argument: if the argument is true, nothing happens. If the argument is false, it panic!s.Let's run our tests again:

assert! **是一个** Rust **内置的需要一个参数的宏：如果这个参数是** true，**那么什么都不会发生。如果参数是** false, **那么他就** panic!. **我们再运行一下测试：**

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test it_works ... FAILED

failures:

---- it_works stdout ----
      thread 'it_works' panicked at 'assertion failed: false', /home/steve/tmp/adder/
⬚ src/lib.rs:3



failures:
    it_works

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured

thread '<main>' panicked at 'Some tests failed', /home/steve/src/rust/src/libtest/lib.r
⬚ s:247
```

Rust indicates that our test failed:

Rust **命令行显示我们的测试失败了：**

```
test it_works ... FAILED
```

And that's reflected in the summary line:

并且反映在概要信息中：

```
test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured
```

We also get a non-zero status code:

我们还可以得到了一个非零状态码：

```
$ echo $?
101
```

This is useful if you want to integrate cargo test into other tooling.

如果你想集成 cargo test 到其他工具中，这是非常有用的。

We can invert our test's failure with another attribute: should_panic:

我们还可以通过使用另一个属性——should_panic——来将测试失败情况确认为成功情况：

```rust
#[test]
#[should_panic]
fn it_works() {
    assert!(false);
}
```

This test will now succeed if we panic! and fail if we complete. Let's try it:

如果我们 panic! 且执行结果为失败时，这个测试将会显示成功。让我们运行它：

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

Rust provides another macro, assert_eq!, that compares two arguments for equality:

Rust 提供了另一个宏——assert_eq!——它是用来比较两个参数是否相等：

```rust
#[test]
#[should_panic]
fn it_works() {
    assert_eq!("Hello", "world");
}
```

Does this test pass or fail? Because of the should_panic attribute, it passes:

现在测试是否成功呢？因为它拥有 should_panic **属性，所以他测试通过：**

```
$ cargo test
  Compiling adder v0.0.1 (file:///home/you/projects/adder)
    Running target/adder-91b3e234d4ed382a

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

should_panic tests can be fragile, as it's hard to guarantee that the test didn't fail for an unexpected reason. To help with this, an optional expected parameter can be added to the should_panic attribute. The test harness will make sure that the failure message contains the provided text. A safer version of the example above would be:

should_panic **测试是很不可靠的，因为它很难保证测试不会因为意外原因而运行失败。为了确保测试不会因为意外原因运行失败，一个可选的** expected **参数被加入到** should_panic **属性中。测试套件将确保失败信息中包含着提供的文本。上面例子的安全版本应当如下：**

```
#[test]
#[should_panic(expected = "assertion failed")]
fn it_works() {
    assert_eq!("Hello", "world");
}
```

That's all there is to the basics! Let's write one 'real' test:

基础知识就是这些了！让我们写一个真正的测试吧：

```rust
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[test]
fn it_works() {
    assert_eq!(4, add_two(2));
}
```

This is a very common use of assert_eq!: call some function with some known arguments and compare it to the expected output.

这是一个 assert_eq! **的普通用法：调用一个使用可知参数的函数，并将他的运行结果与预期输出进行比较。**

4.2.2 The **tests** module **tests 单元**

There is one way in which our existing example is not idiomatic: it's missing the tests module. The idiomatic way of writing our example looks like this:

在我们存在的例子中有一点是不符合通用习惯的：他没有 tests **单元。写例子的惯用方式应该像这样：**

```rust
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));
  }
}
```

There's a few changes here. The first is the introduction of a mod tests with a cfg attribute. The module allows us to group all of our tests together, and to also define helper functions if needed, that don't become a part of the rest of our crate. The cfg attribute only compiles our test code if we're currently trying to run the tests. This can save compile time, and also ensures that our tests are entirely left out of a normal build.

这里有少许的不同。第一个变化是在声明一个 mod tests **之前需要一个** cfg **属性。这样，这个模块允许我们将所有的测试放在一起，并且如果需要的话，还可以定义辅助函数——不会成为我们** crate **剩余的一部分。如果我们当前正在尝试运行测试，那么** cfg **属性只会编译我们的测试代码。这样就会节省编译时间，同样能够确定的是，我们的测试是完全被排除在普通构建之外的。**

The second change is the use declaration. Because we're in an inner module,we need to bring our test function into scope. This can be annoying if you have a large module, and so this is a common use of the glob feature.Let's change our src/lib.rs to make use of it:

第二个变化是使用 use **声明。因为测试代码是在一个内部模块，我们需要将测试方法引入到当前的作用域内。如果你有一个庞大的模块，这会非常麻烦的，所以有一个通用的** glob **方法。那么让我们修改一下我们的代码** src/lib.rs **来使用这个测试方法。**

```rust
pub fn add_two(a: i32) -> i32 {
  a + 2
}

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

  #[test]
  fn it_works() {
    assert_eq!(4, add_two(2));
  }
}
```

Note the different use line. Now we run our tests:

注意，这里有所不同的 use 语句，运行测试：

```
$ cargo test
   Updating registry `https://github.com/rust-lang/crates.io-index`
  Compiling adder v0.0.1 (file:///home/you/projects/adder)
   Running target/adder-91b3e234d4ed382a

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

It works!

它运行正常！

The current convention is to use the tests module to hold your "unit-style" tests. Anything that just tests one small bit of functionality makes sense to go here. But what about "integration-style" tests instead? For that, we have the tests directory.

当前的约定是使用 test 单元作为单元式测试。，通常情况，他只能够测试一个功能的一小部分。如果是集成测试呢。为达到这一目的，我们设定了 tests 目录。

4.2.3 The **tests** directory **tests** 目录

To write an integration test, let's make a tests directory, and put a tests/lib.rs file inside, with this as its contents:

要编写集成测试，我们先建立一个 tests 目录，并且放入一个 tests/lib.rs 文件，文件内容如下：

```rust
extern crate adder;

#[test]
fn it_works() {
  assert_eq!(4, adder::add_two(2));
}
```

This looks similar to our previous tests, but slightly different. We now have an extern crate adder at the top. This is because the tests in the tests directory are an entirely separate crate, and so we need to import our library.This is also why tests is a suitable place to write integration-style tests:they use the library like any other consumer of it would.

看上去与之前的测试代码没啥不同，但还是有一些区别。我们现在在文件开始使用了 extern crate adder。这是因为在 tests 目录中的测试代码是完全独立的包。所以我们需要导入我们的库文件。这样是为什么 tests 目录是一个适合些继承测试的地方：它们可以随意的使用库文件。

Let's run them:

运行测试：

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

     Running target/lib-c18e7d3494509e74

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

Now we have three sections: our previous test is also run, as well as our new one.

现在，我们有三个部分：我们以前测试也运行，我们新的测试内容。

That's all there is to the tests directory. The tests module isn't needed here, since the whole thing is focused on tests.

这就是 tests **目录的所有内容了。** tests **单元在这里是不需要的，因为这里所有的事情重点都在测试上。**

Let's finally check out that third section: documentation tests.

最后让我们进入到第三部分：文档测试

4.2.4   Documentation tests 文档测试

Nothing is better than documentation with examples. Nothing is worse than examples that don't actually work, because the code has changed since the documentation has been written. To this end, Rust supports automatically running examples in your documentation. Here's a fleshed-out src/lib.rs with examples:

没有什么比带例子的文档更好的。没有什么比不能如期运行的例子更差劲的了，因为文档一旦被写入，代码就被改变了。为此，Rust **语言支持自动运行你在文档中的例子。这里有一个非常充实的有着使用例子的文件** src/lib.rs:

```rust
//! The `adder` crate provides functions that add numbers to other numbers.
//!
//! # Examples
//!
//! ```
//! assert_eq!(4, adder::add_two(2));
//! ```

/// This function adds two to its argument.
///
/// # Examples
///
/// ```
/// use adder::add_two;
///
/// assert_eq!(4, add_two(2));
/// ```
pub fn add_two(a: i32) -> i32 {
  a + 2
}


#[cfg(test)]
mod tests {
```

```
  use super::*;

  #[test]
  fn it_works() {
    assert_eq!(4, add_two(2));
  }
}
```

Note the module-level documentation with //! and the function-level documentation with ///. Rust's documentation supports Markdown in comments, and so triple graves mark code blocks. It is conventional to include the # Examples section, exactly like that, with examples following.

注意使用了//! **的模块级文档和使用了**/// **的函数级文档。**Rust **的文件支持** MarkDown **语法注释，所以有三种标记代码块。包含** # Examples **是常规做法，并且紧跟随着的是实例。**

Let's run the tests again:

在此运行测试：

```
$ cargo test
  Compiling adder v0.0.1 (file:///home/steve/tmp/adder)
    Running target/adder-91b3e234d4ed382a

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

    Running target/lib-c18e7d3494509e74

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 2 tests
test add_two_0 ... ok
test _0 ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured
```

Now we have all three kinds of tests running! Note the names of the documentation tests: the _0 is generated for the module test, and add_two_0 for the function test. These will auto increment with names like add_two_1 as you add more examples.

现在我们掌握了全部的三种测试方式！注意文档测试的名称： _0 **是由模块测试生成的，** add_two_0 **是有函数测试生成的。随着你增加更多的例子，** _* **将像** add_two_1 **这样自动增加。**

4.3   Conditional Compilation 条件编译

Rust has a special attribute, #[cfg], which allows you to compile codebased on a flag passed to the compiler. It has two forms:

Rust **语言有一个特殊的属性——** #[cfg] **——它允许你基于一个标记参数来编译代码。它有两种形式：**

```
#[cfg(foo)]

#[cfg(bar = "baz")]
```

They also have some helpers:

他们有一些辅助项

```
#[cfg(any(unix, windows))]

#[cfg(all(unix, target_pointer_width = "32"))]

#[cfg(not(foo))]
```

These can nest arbitrarily:

他们可以任意嵌套：

```
#[cfg(any(not(unix), all(target_os="macos", target_arch = "powerpc")))]
```

As for how to enable or disable these switches, if you're using Cargo,they get set in the [features] section[7] of your Cargo.toml:

那么当你使用 Cargo **的时候，如何开启和禁用这些开关呢？他们被设定在你的** Cargo.toml **文件的** [features] section[8] **配置块中。**

```
[features]
# no features by default
default = []

# The "secure-password" feature depends on the bcrypt package.
secure-password = ["bcrypt"]
```

---

[7]http://doc.crates.io/manifest.html#the-{[}features{]}-section
[8]http://doc.crates.io/manifest.html#the-{[}features{]}-section

When you do this, Cargo passes along a flag to rustc:

当你做这么做时，Cargo **传递给** rustc **一个标记：**

--cfg feature="${feature_name}"

The sum of these cfg flags will determine which ones get activated, and therefore, which code gets compiled. Let's take this code:

这些 cfg **标记汇总起来将决定哪些代码能够被激活，因此那部分代码得以被编译。让我们看下面的代码：**

```
#[cfg(feature = "foo")]
mod foo {
}
```

If we compile it with cargo build --features "foo", it will send the --cfg feature="foo" flag to rustc, and the output will have the mod foo in it.If we compile it with a regular cargo build, no extra flags get passed on,and so, no foo module will exist.

如果我们使用 cargo build --features "foo" **来编译它，它将会给** rustc **添加一个**--cfg feature="foo" **的标记，并且在输出信息中有** mod foo。**如果我们使用普通的** cargo build **来编译它，那么就没有给** rustc **添加任何标记，因此也就没有** foo **存在。**

4.3.1   cfg_attr

You can also set another attribute based on a cfg variable with cfg_attr:

你同样可以使用 cfg_attr **给** cfg **变量设置额外的属性值。**

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

Will be the same as #[b] if a is set by cfg attribute, and nothing otherwise.

如果 a **被** cfg **属性设置，**#[b] **也将同样被设计，否则反之。**

4.3.2   cfg!

The cfg! syntax extension (section 6.1, page 279) lets you use these kinds of flags elsewhere in your code, too:

cfg! **语法扩展** (section 6.1, page 279) **也能让你在代码中其他地方使用这种标记：**

```
if cfg!(target_os = "macos") || cfg!(target_os = "ios") {
    println!("Think Different!");
}
```

These will be replaced by a true or false at compile-time, depending on the configuration settings.

根据不同的配置设置，在编译时，这些标记被 true **或者** false **替代。**

4.4   Documentation 文档

Documentation is an important part of any software project, and it's first-class in Rust. Let's talk about the tooling Rust gives you to document your project.

文档在任何软件项目中都是一个很重要的部分，在 Rust 语言中，他是第一等级的。

About rustdoc 关于 rustdoc

The Rust distribution includes a tool, rustdoc, that generates documentation.rustdoc is also used by Cargo through cargo doc.

Rust **分发版本内置了一个工具——**rustdoc**——用于生成文档。**rustdoc **也同样被** Cargo **命令** cargo doc **使用。**

Documentation can be generated in two ways: from source code, and from standalone Markdown files.

文档可以通过两种方式生成：从源代码，或者从单独的 markdown **语句**

Documenting source code

The primary way of documenting a Rust project is through annotating the source code. You can use documentation comments for this purpose:

文档化一个 Rust **语言项目的首要方式是通过注释源代码。你能够使用文档注释达到此目的：**

```
/// Constructs a new `Rc<T>`.
///
/// # Examples
///
/// ```
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// ```
pub fn new(value: T) -> Rc<T> {
   // implementation goes here
}
```

This code generates documentation that looks like this[9]. I've left the implementation out, with a regular comment in its place. That's the first thing to notice about this annotation: it uses ///, instead of //. The triple slash indicates a documentation comment.

此代码生成的文档看起来像这样[10]**。我们在代码执行中使用了一个普通的注释。首先要注意的事情是这个符号：使用的是///而不是//。三个反斜杠表示文档注释。**

---

[9]http://doc.rust-lang.org/nightly/std/rc/struct.Rc.html#method.new
[10]http://doc.rust-lang.org/nightly/std/rc/struct.Rc.html#method.new

Documentation comments are written in Markdown.

文档注释是使用 Markdown **格式来写。**

Rust keeps track of these comments, and uses them when generating documentation.This is important when documenting things like enums:

Rust **持续跟踪这些注释，并且在生成文档时使用他们。当文档化枚举类型数据时，这是很重要的。**

```
/// The `Option` type. See [the module level documentation](../) for more.
enum Option<T> {
    /// No value
    None,
    /// Some value `T`
    Some(T),
}
```

The above works, but this does not:

上面的代码正常运行，但是下面的却不行：

```
/// The `Option` type. See [the module level documentation](../) for more.
enum Option<T> {
    None, /// No value
    Some(T), /// Some value `T`
}
```

You'll get an error:

你会得到一处报错：

```
hello.rs:4:1: 4:2 error: expected ident, found `}`
hello.rs:4 }
           ^
```

This unfortunate error[11] is correct: documentation comments apply to the thing after them, and there's no thing after that last comment.

这个 unfortunate error **不期望的错误**[12]**是正确的：文档注释会请求在他们后面的内容，并且在最后一条注释后面没有内容。**

Writing documentation comments 如何写文档注释 Anyway, let's cover each part of this comment in detail:

无论任何情况，让我们详细的了解注释的没一部分：

---

[11]https://github.com/rust-lang/rust/issues/22547
[12]https://github.com/rust-lang/rust/issues/22547

```
/// Constructs a new `Rc<T>`.
# fn foo() {}
```

The first line of a documentation comment should be a short summary of its functionality. One sentence. Just the basics. High level.

文档注释的第一行应该是一个功能的简要说明。一句话。只包括基本的内容。高度概括。

```
///
/// Other details about constructing `Rc<T>`s, maybe describing complicated
/// semantics, maybe additional options, all kinds of stuff.
///
# fn foo() {}
```

Our original example had just a summary line, but if we had more things to say,we could have added more explanation in a new paragraph.

我们上面的例子只有一个摘要行，如果我们有很多东西要说，我们我们需要在一个新的段落中增加更多的说明。

Special sections 特殊段

```
/// # Examples
# fn foo() {}
```

Next, are special sections. These are indicated with a header, #. There are three kinds of headers that are commonly used. They aren't special syntax,just convention, for now.

下一部分是特殊段。他们使用 # 作为标记。通常有三种标题的用法。现在，这些内容并不是特殊语法，只是惯例。

```
/// # Panics
# fn foo() {}
```

Unrecoverable misuses of a function (i.e. programming errors) in Rust are usually indicated by panics, which kill the whole current thread at the very least. If your function has a non-trivial contract like this, that is detected/enforced by panics, documenting it is very important.

在 Rust 语言中一个函数的滥用（例如变成错误）通常被标记为 panics, 这至少会杀死整个线程。如果你的函数有一个重要的约定就像这样会被 panics 检测红着强制使用，文档化就会变得非常重要。

```
/// # Failures
# fn foo() {}
```

If your function or method returns a `Result<T, E>`, then describing the conditions under which it returns `Err(E)` is a nice thing to do. This is slightly less important than `Panics`, because failure is encoded into the type system, but it's still a good thing to do.

如果你的函数或者方法有一个返回值 Result<T, E>, 那么在它返回 Err(E) 前提下描述是一个非常好的事情。相比于 Panics, 这个稍微次要些，因为失败是被编码进类型系统的，但这样做仍然不失为一个好事情。

```
/// # Safety
# fn foo() {}
```

If your function is `unsafe`, you should explain which invariants the caller is responsible for upholding.

如果你的函数是 unsafe 的，你需要说明地哪一个不变量的调用者来负责维护。

```
/// # Examples
///
/// ```
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// ```
# fn foo() {}
```

Third, `Examples`. Include one or more examples of using your function or method, and your users will love you for it. These examples go inside of code block annotations, which we'll talk about in a moment, and can have more than one section:

第三个是 Examples 例子。包含一个或者多个你的函数或者方法的使用例子，你的用户将因为这个喜欢你。这些例子将进入到代码块注释中，我们一会会讨论这个，并且他可能有多个部分。

```
/// # Examples
///
/// Simple `&str` patterns:
///
/// ```
/// let v: Vec<&str> = "Mary had a little lamb".split(' ').collect();
/// assert_eq!(v, vec!["Mary", "had", "a", "little", "lamb"]);
/// ```
///
/// More complex patterns with a lambda:
///
/// ```
/// let v: Vec<&str> = "abc1def2ghi".split(|c: char| c.is_numeric()).collect();
/// assert_eq!(v, vec!["abc", "def", "ghi"]);
/// ```
# fn foo() {}
```

Let's discuss the details of these code blocks.

下面我们讨论代码块的详细内容。

Code block annotations 代码块注释   To write some Rust code in a comment, use the triple graves:

使用三个 "'" 可以在注释中写一些 Rust 代码。

```
/// ```
/// println!("Hello, world");
/// ```
# fn foo() {}
```

If you want something that's not Rust code, you can add an annotation:

如果你想使用其他语言代码，你可以增加一个注释:

```
/// ```c
/// printf("Hello, world\n");
/// ```
# fn foo() {}
```

This will highlight according to whatever language you're showing off.If you're just showing plain text, choose text.

这将根据你的语言来显示高亮，如果你只想显示一个纯文本，那么请使用 text。

It's important to choose the correct annotation here, because rustdoc uses it in an interesting way: It can be used to actually test your examples, so that they don't get out of date. If you have some C code but rustdoc thinks it's Rust because you left off the annotation, rustdoc will complain when trying to generate the documentation.

选择一个恰当的注释是非常重要的，因为 rustdoc 命令用一种有趣的方式来使用它：它被用来实际测试你的例子，所以它们不会过时。如果你试用了 c 语言代码，因为你没有留下注释，那么 rustdoc 将会默认他为 Rust 代码，那么当 rustdoc 尝试生成文档的时候，就会报错。

Documentation as tests 文档即测试

Let's discuss our sample example documentation:

来说一下我们简单例子的文档:

```
/// ```
/// println!("Hello, world");
/// ```
# fn foo() {}
```

You'll notice that you don't need a fn main() or anything here.rustdoc will automatically add a main() wrapper around your code, and in the right place.For example:

你会注意到，在这里，你不需要一个 fn main() **或者任何东西。** rustdoc **将自动增加一个** main() **包在你的代码中，并且在正确的位置。例如：**

```
/// ```
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// ```
# fn foo() {}
```

This will end up testing:

这会产生下面的测试：

```
fn main() {
    use std::rc::Rc;
    let five = Rc::new(5);
}
```

Here's the full algorithm rustdoc uses to postprocess examples:

下面是 rustdoc **使用的处理后的完整算法：**

1. Any leading #![foo] attributes are left intact as crate attributes. **任何前面带有** #![foo] **属性的是保持完整不变的** crate **属性**
2. Some common allow attributes are inserted, including unused_variables,unused_assignments,unused_mut,unused_attributes, and dead_code. Small examples often trigger these lines. **包括** unused_variables, unused_assignments, unused_mut,unused_attributes **和** dead_code **的一些允许的属性是可以插入的。小粒子长长出发这些代码。**
3. If the example does not contain extern crate, then extern crate <mycrate>; is inserted. **如果例子中不包括** extern crate, **那么** extern crete<mycrate>; **会被插入。**
4. Finally, if the example does not contain fn main, the remainder of the text is wrapped in fn main() { your_code }. **最后，如果你的例子没有包含** fn main, **文本剩余的部分将被包裹进** fn main() { your_code }**。**

Sometimes, this isn't enough, though. For example, all of these code samples with /// we've been talking about? The raw text:

然而有时候，这些是不够的。比如，我们说过所有使用///**的代码？原始文本是：**

```
/// Some documentation.
# fn foo() {}
```

looks different than the output: **看出他们的不同了嘛?**

```
/// Some documentation.
# fn foo() {}
```

Yes, that's right: you can add lines that start with #, and they will be hidden from the output, but will be used when compiling your code.You can use this to your advantage. In this case, documentation comments need to apply to some kind of function, so if I want to show you just a documentation comment, I need to add a little function definition below it. At the same time, it's just there to satisfy the compiler, so hiding it makes the example more clear. You can use this technique to explain longer examples in detail, while still preserving the testability of your documentation. For example, this code:

是的，这是正确的：你可以在开始位置使用 # **, 他们会在输出时隐藏掉，但在编译代码时会被用到。你可以使用这个作为你的优势。这种情况下，文档注释需要适用于某种功能，所以如果我要告诉你它只是一个文档注释，我需要在下面加点函数定义它。同时，它只是为了满足编译器，所以隐藏它会使例子更加清晰。你可以使用此方法在细节处注释你的例子，同时还能够保证文档的可测试性。例如下面的代码：**

```
let x = 5;
let y = 6;
println!("{}", x + y);
```

Here's an explanation, rendered:

这是一个给出的一个注释。

First, we set x to five:

首先我们设置 x **的值为** 5**：**

```
let x = 5;
# let y = 6;
# println!("{}", x + y);
```

Next, we set y to six:

然后我们设置 y **的值为** 6**：**

```
# let x = 5;
let y = 6;
# println!("{}", x + y);
```

Finally, we print the sum of x and y:

最终打印出 x **和** y **的和：**

```
# let x = 5;
# let y = 6;
println!("{}", x + y);
```

Here's the same explanation, in raw text:

这原始文本中类似的注释：

First, we set x to five:

```
let x = 5;
# let y = 6;
# println!("{}", x + y);
```

Next, we set y to six:

```
# let x = 5;
let y = 6;
# println!("{}", x + y);
```

Finally, we print the sum of x and y:

```
# let x = 5;
# let y = 6;
println!("{}", x + y);
```

By repeating all parts of the example, you can ensure that your example still compiles, while only showing the parts that are relevant to that part of your explanation.

通过重复例子的所有部分，你能够确定你的代码仍然能够通过编译，同时只显示相关解释说明的部分内容。

Documenting macros 文档化宏　　Here's an example of documenting a macro:

这是文档化宏的一个例子：

```
/// Panic with a given message unless an expression evaluates to true.
///
/// # Examples
///
/// ```
/// # #[macro_use] extern crate foo;
/// # fn main() {
/// panic_unless!(1 + 1 == 2, "Math is broken.");
/// # }
/// ```
```

```
///
/// ```should_panic
/// # #[macro_use] extern crate foo;
/// # fn main() {
/// panic_unless!(true == false, "I'm broken.");
/// # }
/// ```
#[macro_export]
macro_rules! panic_unless {
    ($condition:expr, $($rest:expr),+) => ({ if ! $condition { panic!($($rest),+); } });
}
# fn main() {}
```

You'll note three things: we need to add our own extern crate line, so that we can add the #[macro_use] attribute. Second, we'll need to add our own main() as well. Finally, a judicious use of # to comment out those two things, so they don't show up in the output.

你会注意到三点事情：因为我们需要手动增加我们自己的 extern crate **代码，所以我们能够使用** #[macro_use] **属性。第二点，我们同样需要增加自己的** main()。**最后，** # **用来注释以上两点内容，所以他们不会在输出中显示出来。**

Running documentation tests 运行文档测试   To run the tests, either

现在运行测试。

```
$ rustdoc --test path/to/my/crate/root.rs
# or
$ cargo test
```

That's right, cargo test tests embedded documentation too. However, cargo test will not test binary crates, only library ones. This is due to the way rustdoc works: it links against the library to be tested, but with a binary, there's nothing to link to.

非常好， cargo test **命令也测试了嵌入的文档。然而** cargo test **不会检测二进制文件，只会检测库中的那些。这取决于** rustdoc **的运行逻辑：它只测试连接到的库，但是二进制文件没有什么可以链接的。**

There are a few more annotations that are useful to help rustdoc do the right thing when testing your code:

有一些注释，在测试代码时，能够帮助 rustdoc **做这些应该的事情。**

```
/// ```ignore
/// fn foo() {
/// ```
# fn foo() {}
```

The ignore directive tells Rust to ignore your code. This is almost never what you want, as it's the most generic. Instead, consider annotating it with text if it's not code, or using #s to get a working example that only shows the part you care about.

ignore 指令是用来告诉 Rust 忽略掉你的代码。这在很多时候不是你想要的，因为他是最通用的。相反，如果他们不是代码你应该考虑使用欧冠 text 注释，或者使用 # 开始一个使用例子，用来只展示你关心的部分。

```
/// ```should_panic
/// assert!(false);
/// ```
# fn foo() {}
```

should_panic tells rustdoc that the code should compile correctly, but not actually pass as a test. should_panic 告诉 rustdoc 代码应该立即被编译，而不是作为一个测试通过。

```
/// ```no_run
/// loop {
///     println!("Hello, world");
/// }
/// ```
# fn foo() {}
```

The no_run attribute will compile your code, but not run it. This is important for examples such as "Here's how to start up a network service," which you would want to make sure compile,but might run in an infinite loop!

no_run 属性只会编译你的代码，而不是运行它。这是非常重要的，比如这样的例子 "Here's how to start up a network service, 如何开启一个网络服务" 是你确定要编译的，但是可能会进入死循环中。

Documenting modules 文档模块 Rust has another kind of doc comment, //!. This comment doesn't document the next item, but the enclosing item. In other words:

Rust 语言有另外一种文档注释，//! 这个注释不会文档化下一个部分，但是会封装项目。换句话说：

```
mod foo {
  //! This is documentation for the `foo` module.
  //!
  //! # Examples

  // ...
}
```

This is where you'll see //! used most often: for module documentation. If you have a module in foo.rs, you'll often open its code and see this:

这是//! 经常被使用到的地方：作为模块文档。如果你在 foo.rs 中有一个模块，只要你打开它，就能够看到像这样的代码：

```
//! A module for using `foo`s.
//!
//! The `foo` module contains a lot of useful functionality blah blah blah
```

Documentation comment style 文档注释的风格   Check out RFC 505[13] for full conventions around the style and format of documentation.

打开 RFC 505[14]**可以看到关于文档的格式和风格的全部惯例。**

Other documentation 其他文档

All of this behavior works in non-Rust source files too. Because comments are written in Markdown, they're often .md files.

这种行为也应当在非 Rust **语言文档中使用。因为文档是使用** Markdown **格式来书写的，他们大多是**.md **后缀的文件。**

When you write documentation in Markdown files, you don't need to prefix the documentation with comments. For example:

当你在 Markdown **文件中写文档时，你不需要给注释文档使用前缀。例如：**

```
/// # Examples
///
/// ```
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// ```
# fn foo() {}
```

is just

就是

```
# Examples

use std::rc::Rc;

let five = Rc::new(5);
```

---

[13]https://github.com/rust-lang/rfcs/blob/master/text/0505-api-comment-conventions.md
[14]https://github.com/rust-lang/rfcs/blob/master/text/0505-api-comment-conventions.md

when it's in a Markdown file，There is one wrinkle though: Markdown files need to have a title like this:

当它是一个 markdown **代码文件时。虽然有一个波浪符格式：** Markdown **文件需要有一个像这样的标题：**

% The title

This is the example documentation.

This % line needs to be the very first line of the file.

% **必须在文件第一行的开头**

doc attributes **doc 属性**

At a deeper level, documentation comments are sugar for documentation attributes:

更深层次，文档注释就是文档属性的糖：

```
/// this
# fn foo() {}

#[doc="this"]
# fn bar() {}
```

are the same, as are these:

他们是相同的，还有这个

```
//! this

#![doc="/// this"]
```

You won't often see this attribute used for writing documentation, but it can be useful when changing some options, or when writing a macro.

在撰写文档时，这个属性时不常见的，但是当更改某些选项，后者写一个宏的时候，他就会非常有用。

Re-exports 转口   rustdoc will show the documentation for a public re-export in both places:

rustdoc **在任何位置都会给文档一个输出口：**

```
extern crate foo;

pub use foo::bar;
```

This will create documentation for bar both inside the documentation for the crate foo, as well as the documentation for your crate. It will use the same documentation in both places.

这不仅会为 bar **创建文档，同样会为** crate **类型的** foo **创建文档。在两个地方它会使用相同的文档。**

This behavior can be suppressed with no_inline:

这种行为可以通过使用 on_inline **来取缔：**

```
extern crate foo;

#[doc(no_inline)]
pub use foo::bar;
```

Controlling HTML 控制超文本文件   You can control a few aspects of the HTML that rustdoc generates through the #![doc] version of the attribute:

你可以控制 rustdoc **通过** #[doc] **版本属性生成的一部分** HTML **代码：**

```
#![doc(html_logo_url = "http://www.rust-lang.org/logos/rust-logo-128x128-blk-v2.png",
    html_favicon_url = "http://www.rust-lang.org/favicon.ico",
    html_root_url = "http://doc.rust-lang.org/")];
```

This sets a few different options, with a logo, favicon, and a root URL.

这里对 logo**，图标和跟地址进行了一些不同的设置。**

Generation options 生成选项

rustdoc also contains a few other options on the command line, for further customization:

rustdoc **同样内置了一些命令行选项，来应对定制：**

- --html-in-header FILE: includes the contents of FILE at the end of the <head>...</head> section. **在** <head>...</head> **标签的末尾引入一个文件的内容。**
- --html-before-content FILE: includes the contents of FILE directly after <body>, before the rendered content (including the search bar). **直接在** <body> **标签后面引入文件内容，在其他显示内容（包括搜索框）之前。**
- --html-after-content FILE: includes the contents of FILE after all the rendered content. **在所有显示内容之后，进入文件内容。**

Security note 安全事项

The Markdown in documentation comments is placed without processing into the final webpage. Be careful with literal HTML:

不需要生成最后的 web **页面，注释中的** markdown **部分就会被替换。请小心使用** HTML **标记：**

4.5 Iterators 迭代器

Let's talk about loops.

下面我们来说说循环。

Remember Rust's for loop? Here's an example:

还记得 Rust 语言的 for 循环吗？这里有个例子：

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

Now that you know more Rust, we can talk in detail about how this works.Ranges (the 0..10) are 'iterators'. An iterator is something that we can call the .next() method on repeatedly, and it gives us a sequence of things.

现在你了解到更多的 Rust 知识，我们可以讨论一下循环是怎么运行的了。0..10 **就是 iterators 迭代器。迭代器就是我们能够不断重复代用的**.next()**，并且它给了我们一个事物的序列。**

Like this:

就像这样：

```rust
let mut range = 0..10;

loop {
  match range.next() {
    Some(x) => {
      println!("{}", x);
    },
    None => { break }
  }
}
```

We make a mutable binding to the range, which is our iterator. We then loop,with an inner match. This match is used on the result of range.next(),which gives us a reference to the next value of the iterator. next returns an Option<i32>, in this case, which will be Some(i32) when we have a value and None once we run out. If we get Some(i32), we print it out, and if we get None, we break out of the loop.

我们设置了一个范围的变量，它就是我们的迭代器。使用一个内置的 match 方法进行循环。match **被用在 range.next() 的值上，他给我们迭代器下一个值的地址引用。**netx **返回一个 Option<i32>（译者：就是整型值），在本案例中，当我们有一个值时，他就是 Some(i32), 或者运行一次 None。如果我们的**

匹配值是 Some(u=i32)，我们就打印出来，如果我们匹配的值是 None，我们 break 打断这个循环，跳出来。

This code sample is basically the same as our for loop version. The for loop is just a handy way to write this loop/match/break construct.

这段代码基本与我们的 for 循环相同。该 for 循环只是用了一个方便的方式来写这个 loop 循环/match 匹配/break 跳出结构。

for loops aren't the only thing that uses iterators, however. Writing your own iterator involves implementing the Iterator trait. While doing that is outside of the scope of this guide, Rust provides a number of useful iterators to accomplish various tasks. Before we talk about those, we should talk about a Rust anti-pattern. And that's using ranges like this.

然而，for 循环并不是使用迭代器的唯一方式。编写你自己的的迭代器，涉及到了 Iterator 特性。然而这样做超出了本教程的范围，Rust 提供了许多有用的迭代器来完成歌中任务。在我们谈论这些之前，我们来讨论下 Rust 语言的反射模式。它们行这样使用序列。

Yes, we just talked about how ranges are cool. But ranges are also very primitive. For example, if you needed to iterate over the contents of a vector,you may be tempted to write this:

是的，我们只谈论下为什么序列是非常酷的。但是，他同样是很原始的。举个例子，如果遍历一个响亮的内容，你可能会这样写：

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

for i in 0..nums.len() {
  println!("{}", nums[i]);
}
```

This is strictly worse than using an actual iterator. You can iterate over vectors directly, so write this:

这比使用实际的迭代器都要糟糕。你可以直接遍历向量，所以应该这么写：

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

for num in &nums {
  println!("{}", num);
}
```

There are two reasons for this. First, this more directly expresses what we mean. We iterate through the entire vector, rather than iterating through indexes, and then indexing the vector. Second, this version is more efficient:the first version will have extra bounds checking because it used indexing,nums[i]. But since we yield a reference to each element of the vector in turn with the iterator, there's no bounds checking in the second example. This is very common with iterators: we can ignore unnecessary bounds checks, but still know that we're safe.

这样做有两个原因。首先，这更能直接表达处我们的意思。我们遍历整个向量，并且索引向量，而不是遍历索引。第二点原因就是这样做更加高效：第一版的代码需要额外的边界检查，因为他用的

是索引-nums[i]。但是由于我们使用迭代器时得到了向量循环的每一个元素的索引，所以在第二版的代码中不需要边界检查。这是迭代器的通用做法：我们忽略不必要的边界检查，却仍然保持边界安全。

There's another detail here that's not 100% clear because of how println! works. num is actually of type &i32. That is, it's a reference to an i32,not an i32 itself. println! handles the dereferencing for us, so we don't see it. This code works fine too:

这里有另一个细节，因为 println! 的执行方式，不是 100% 的清楚。num 是真实的 &i32 类型。这是说，他是一个 i32 的地址引用，而不是 i32 自身。println! 自动为我们处理了这个引用，所以，我们看不到它。也因为这样，代码能够很好的运行：

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

for num in &nums {
  println!("{}", *num);
}
```

Now we're explicitly dereferencing num. Why does &nums give us references?Firstly, because we explicitly asked it to with &. Secondly, if it gave us the data itself, we would have to be its owner, which would involve making a copy of the data and giving us the copy. With references, we're just borrowing a reference to the data, and so it's just passing a reference, without needing to do the move.

现在我们明确解除 num 的地址引用。为什么 &nums 给我们的是一个地址引用呢？首先，因为我们使用 & 来明确的请求它。第二如果他给我们他们身的数据，我们不得不程伟他的拥有者，这将涉及到制作数据副本，并给我们这个副本。使用地址引用，我们只需要引用数据的一个地址，并且他只是传递一个地址，不需要做其他事情。

So, now that we've established that ranges are often not what you want, let's talk about what you do want instead.

所以，现在我们已经确定了序列常常不是我们所需要的，让我们讨论下，你想要的是什么吧。

There are three broad classes of things that are relevant here: iterators,*iterator adapters*, and *consumers*. Here's some definitions:

一共有三大种类：迭代器，迭代器适配器和消费者。这是他们的定义：

- *iterators* give you a sequence of values. 迭代器给你有一些值的序列。
- *iterator adapters* operate on an iterator, producing a new iterator with a different output sequence. 迭代器适配器操作一个迭代器，生成一个新的不同序列的迭代器。
- *consumers* operate on an iterator, producing some final set of values. 消费者操作一个迭代器，生成最后的一组值。

Let's talk about consumers first, since you've already seen an iterator, ranges.

让我们首先讨论消费者，因为你已经见过一个迭代器——序列。

Consumers 消费者

A *consumer* operates on an iterator, returning some kind of value or values.The most common consumer is collect(). This code doesn't quite compile,but it shows the intention:

一个 *consumer* **操作一个迭代器时，返回某种值或者所有值。最通常使用的消费者是** collect(). **这段代码无法被编译，只是为了表明某种意图。**

```
let one_to_one_hundred = (1..101).collect();
```

As you can see, we call collect() on our iterator. collect() takes as many values as the iterator will give it, and returns a collection of the results. So why won't this compile? Rust can't determine what type of things you want to collect, and so you need to let it know.Here's the version that does compile:

曾如你所见，我们在迭代器上调用了 collect()。collect() **收集尽可能躲得迭代器的值，并返回结果集。那么为什么这个代码不能够被编译呢？** Rust **语言无法决定你想要收集何种类型的内容，所以你需要让它知道。这是那段代码的编译版本：**

```
let one_to_one_hundred = (1..101).collect::<Vec<i32>>();
```

If you remember, the ::<> syntax allows us to give a type hint,and so we tell it that we want a vector of integers. You don't always need to use the whole type, though. Using a _ will let you provide a partial hint:

如果你还记得， ::<> **语法允许我们给出一个类型提示，那么我们可以告诉它，我们需要一个整型的向量。然而你并不需要使用所有的类型。你可以通过使用_来给出部分提示：**

```
let one_to_one_hundred = (1..101).collect::<Vec<_>>();
```

This says "Collect into a Vec<T>, please, but infer what the T is for me."_ is sometimes called a "type placeholder" for this reason.

这是说 "收集成一个向量 Vec<T>, **拜托，不要为我推断** T。" **因为这个原因，_有时候被称作占位符。**

collect() is the most common consumer, but there are others too. find() is one:

collect() **是最常用的消费者，还有一些其他方法也是。** find() **也是其中一个：**

```
let greater_than_forty_two = (0..100)
                .find(|x| *x > 42);

match greater_than_forty_two {
  Some(_) => println!("We got some numbers!"),
  None => println!("No numbers found :("),
}
```

find takes a closure, and works on a reference to each element of an iterator. This closure returns true if the element is the element we're looking for, and false otherwise. Because we might not find a matching element, find returns an Option rather than the element itself.

find 使用一个闭包，运行是建立在迭代器每一个元素的地址引用基础上的。当元素时我们寻找的时，这段代码中的闭包会返回 true, 反之，返回 false。因为我们可能不会找到匹配的元素，find 会返回一个 Option 而不是元素本身。

Another important consumer is fold. Here's what it looks like:

另一个重要的消费者是 fold。这是它看起来的样子：

```
let sum = (1..4).fold(0, |sum, x| sum + x);
```

fold() is a consumer that looks like this:fold(base, |accumulator, element| ...). It takes two arguments: the first is an element called the *base.* The second is a closure that itself takes two arguments: the first is called the *accumulator*, and the second is an *element.* Upon each iteration, the closure is called, and the result is the value of the accumulator on the next iteration. On the first iteration, the base is the value of the accumulator.

fold() 是一个想这样子的消费者：fold(base, |accumulator,element| ...)。它有两个参数：一个是叫做 *base* 的元素，另一个是一个拥有两个参数的闭包（其中一个参数被叫做 *accumulator*，另一个是 *element*）。在每一次迭代中，闭包都被调用，结果是对下一个迭代累加器的值。在第一个迭代中，base 是累加器的值。

Okay, that's a bit confusing. Let's examine the values of all of these things in this iterator:

好吧，这里有点混乱。让我们来看看在迭代器中，所有这些内容的值：

base	accumulator	element	closure result
0	0	1	1
0	1	2	3
0	3	3	6

We called fold() with these arguments:

我们调用 fold() 使用这些参数：

```
.fold(0, |sum, x| sum + x);
```

So, 0 is our base, sum is our accumulator, and x is our element. On the first iteration, we set sum to 0, and x is the first element of nums,1. We then add sum and x, which gives us 0 + 1 = 1. On the second iteration, that value becomes our accumulator, sum, and the element is the second element of the array, 2. 1 + 2 = 3, and so that becomes the value of the accumulator for the last iteration. On that iteration,x is the last element, 3, and 3 + 3 = 6, which is our final result for our sum. 1 + 2 + 3 = 6, and that's the result we got.

所以，0 就是 base，num 是 accumulator,x 是 element。在第一次迭代时，我们设置 sum 的值为 0，x 是 nums 的第一个元素——1. 然后我们将 sum 和 x 相加，得到 0+1=1。在第二次迭代中，这个值变

成了 accumulator——sum **的值，** element **是数组的第二个元素**——2.1+2=3，**他有成为最后一个迭代时** accumulator **的值。在最后一次迭代时，** x **是最后一个元素**——3，3+3=6, **这也是最后的** sum **的值。** 1+2+3=6, **这就是我们得到最终结果。**

Whew. fold can be a bit strange the first few times you see it, but once it clicks, you can use it all over the place. Any time you have a list of things,and you want a single result, fold is appropriate.

fold **在你最初看他的几回时可能是有点奇怪的，但是一旦它被激活，你可以用在所有的地方。任何时候，你有一个清单，需要返回单一结果时，** fold **是最合适的。**

Consumers are important due to one additional property of iterators we haven't talked about yet: laziness. Let's talk some more about iterators, and you'll see why consumers matter.

消费者的重要性取决于我们还没有讨论的，迭代器的一个附加属性： laziness。**让我们多讨论些迭代器，你讲明白消费者关系。**

Iterators 迭代器

As we've said before, an iterator is something that we can call the .next() method on repeatedly, and it gives us a sequence of things.Because you need to call the method, this means that iterators can be *lazy* and not generate all of the values upfront. This code, for example, does not actually generate the numbers 1-100, instead creating a value that merely represents the sequence:

正如我们之前所说，迭代器就是一个能够重复调用.next() **方法的事物，它给我们一个事物的序列。因为我们需要调用这个方法，这意味着迭代器是** *lazy*，**而不是生成前期所有的值。例如，这段代码并不是直接生成** 1-100, **而是，创建一个仅仅表示一个序列的值，**

```
let nums = 1..100;
```

Since we didn't do anything with the range, it didn't generate the sequence.Let's add the consumer:

因为我们使用这个范围什么都不能做，他不能够生成序列。让我们加入一个消费者：

```
let nums = (1..100).collect::<Vec<i32>>();
```

Now, collect() will require that the range gives it some numbers, and so it will do the work of generating the sequence.

现在 collect() **将请求范围给它的一些数，所以，它能够做生成序列这个事情。**

Ranges are one of two basic iterators that you'll see. The other is iter().iter() can turn a vector into a simple iterator that gives you each element in turn:

Ranges **是你即将看到的两个基本迭代器之一。另一个是** iter()。iter **能够将向量转换成一个简单的迭代器，依次给你每一个元素：**

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

for num in nums.iter() {
  println!("{}", num);
}
```

These two basic iterators should serve you well. There are some more advanced iterators, including ones that are infinite.

这两个基本的迭代器将很好的服务你。还有一些高级迭代器，包括哪些无穷大的。

That's enough about iterators. Iterator adapters are the last concept we need to talk about with regards to iterators. Let's get to it!

迭代器的东西已经足够多了。迭代器适配器是我们需要讨论的最后一个概念，让我们开始吧！

Iterator adapters 迭代器适配器

Iterator adapters take an iterator and modify it somehow, producing a new iterator. The simplest one is called map:

Iterator adapters **获取一个迭代器，并在某些时候修改它，生成一个新的迭代器。最基本的一个叫做** map:

```
(1..100).map(|x| x + 1);
```

map is called upon another iterator, and produces a new iterator where each element reference has the closure it's been given as an argument called on it.So this would give us the numbers from 2-100. Well, almost! If you compile the example, you'll get a warning:

map **被另一个迭代器调用，并生成一个新的迭代器，在每一个元素引用拥有一个闭包座位参数被他调用的地方。所以，这将给我们从** 2-100 **的数字。好了，差不多了！如果你编译这个例子，你会收到一个敬告：**

```
warning: unused result which must be used: iterator adaptors are lazy and
    do nothing unless consumed, #[warn(unused_must_use)] on by default
(1..100).map(|x| x + 1);
 ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Laziness strikes again! That closure will never execute. This example doesn't print any numbers:

laziness **再次罢工了！闭包讲不会被执行。这个例子没有输出任何数字：**

```
(1..100).map(|x| println!("{}", x));
```

If you are trying to execute a closure on an iterator for its side effects,just use for instead.

如果你想自一个迭代器的副本上执行一个闭包，请使用 for。

There are tons of interesting iterator adapters. take(n) will return an iterator over the next n elements of the original iterator. Note that this has no side effect on the original iterator. Let's try it out with our infinite iterator from before:

有成千上万的迭代器适配器。 take(n) **将反悔一个覆盖了原来第** n **个元素迭代器的迭代器。注意，这对原来的迭代器无副作用。让我们尝试一下之前说过的无限迭代器：**

```
for i in (1..).step_by(5).take(5) {
  println!("{}", i);
}
```

This will print

这将会打印

```
1
6
11
16
21
```

filter() is an adapter that takes a closure as an argument. This closure returns true or false. The new iterator filter() produces only the elements that that closure returns true for:

filter() **是一个适配器，它使用一个闭包座位参数。这个闭包反悔** true **或者** false。**新的迭代器** filter() **只生成一个在闭包返回** true **时的元素：**

```
for i in (1..100).filter(|&x| x % 2 == 0) {
  println!("{}", i);
}
```

This will print all of the even numbers between one and a hundred.(Note that because filter doesn't consume the elements that are being iterated over, it is passed a reference to each element, and thus the filter predicate uses the &x pattern to extract the integer itself.)

这将打印 0 **到** 100 **的所有的每一个偶数。（注意，** filter **并不消耗正在迭代的元素，他只是传递一个美格元素的地址引用，并且** filter **使用** &x **方式来提取整数本身。）**

You can chain all three things together: start with an iterator, adapt it a few times, and then consume the result. Check it out:

你可以将三个事物关联在一起了，首先是迭代器，适应它一段时间，然后是产生一个结果。看看下面的代码

```
(1..1000)
  .filter(|&x| x % 2 == 0)
  .filter(|&x| x % 3 == 0)
  .take(5)
  .collect::<Vec<i32>>();
```

This will give you a vector containing 6, 12, 18, 24, and 30.

这将给我们一个包含有 6，12，18，24 **和** 30 **的向量。**

This is just a small taste of what iterators, iterator adapters, and consumers can help you with. There are a number of really useful iterators, and you can write your own as well. Iterators provide a safe, efficient way to manipulate all kinds of lists. They're a little unusual at first, but if you play with them, you'll get hooked. For a full list of the different iterators and consumers, check out the iterator module documentation[15].

这只是一个关于迭代器，迭代器适配器和消费者能够帮你什么忙的小尝试。有太多有用的迭代器，同样，你可以写你自己的迭代器。迭代器提供一个安全、高效的方式来操纵各种各样的清单列表。起先，它们是有点不寻常，当你多使用他们，你就会上瘾。对于一个完整的迭代器和消费者之间不同之处，可以点击链接 iterator module documentation 迭代器模块文档[16]查看。

4.6 Concurrency **并发**

Concurrency and parallelism are incredibly important topics in computer science, and are also a hot topic in industry today. Computers are gaining more and more cores, yet many programmers aren't prepared to fully utilize them.

并发和并行是计算机科学的非常重要课题，也是当今社会行业的一个热门话题。计算机正获得越来越多的内核，然而，大多数的程序员都没有做好充分利用他们的准备。

Rust's memory safety features also apply to its concurrency story too. Even concurrent Rust programs must be memory safe, having no data races. Rust's type system is up to the task, and gives you powerful ways to reason about concurrent code at compile time.

Rust 的内存安全特性同样也允许它适用于并发。甚至并发的 Rust 程序也必须是内存安全的，没有数据溢出。Rust 的类型体系能够胜任这一工作，并能够提供给你在编译时推断并发代码的强大的方式。

Before we talk about the concurrency features that come with Rust, it's important to understand something: Rust is low-level enough that all of this is provided by the standard library, not by the language. This means that if you don't like some aspect of the way Rust handles concurrency, you can implement an alternative way of doing things. mio[17] is a real-world example of this principle in action.

在我们谈论 Rust 语言的并发之前，理解一些概念是非常重要的：Rust 语言是非常底层的语言，所有的一切都是由标准库来提供的，而不是由语言本身。这意味着，如果你不喜欢 Rust 语言处理并发方式的某些部分时，你可以使用另一种方式来做同样的事情。mio[18] 是一个践行这一原则的真实的例子。

Background: `Send` and `Sync` 后台发送和同步

Concurrency is difficult to reason about. In Rust, we have a strong, static type system to help us reason about our code. As such, Rust gives us two traits to help us make sense of code that can possibly be concurrent.

---

[15]http://doc.rust-lang.org/std/iter/index.html
[16]http://doc.rust-lang.org/std/iter/index.html
[17]https://github.com/carllerche/mio
[18]https://github.com/carllerche/mio

并发是很难推理的。在 Rust **语言中，我们有一个强大的静态类型体系来帮助我们推断代码。正因此，**Rust **提供给我们两个显著特性帮助我们写出感觉上不可能并发的代码。**

Send 发送   The first trait we're going to talk about is Send[19]. When a type T implements Send, it indicates to the compiler that something of this type is able to have ownership transferred safely between threads.

我们要讲的第一个特性就是 send **发送**[20]**。当一个类型** T **实现了** Send **时，它表示着，编译器有权在线程间安全传递这个类型的内容。**

This is important to enforce certain restrictions. For example, if we have a channel connecting two threads, we would want to be able to send some data down the channel and to the other thread. Therefore, we'd ensure that Send was implemented for that type.

强制执行一定的限制是非常必要的。例如，我们有一个包含两个线程的信道，我们想要能够发送一些数据到其他的线程中。所以我们需要确定 Send **是那个类型的实现方式。**

In the opposite way, if we were wrapping a library with FFI that isn't threadsafe, we wouldn't want to implement Send, and so the compiler will help us enforce that it can't leave the current thread.

反过来，如果我们正在封装一个不是线程安全的，使用了外部函数接口的库，我们不需要实现 send**，所以编译器需要帮助我们限制它不能够离开当前的线程。**

Sync 同步   The second of these traits is called Sync[21].When a type T implements Sync, it indicates to the compiler that something of this type has no possibility of introducing memory unsafety when used from multiple threads concurrently.

第二个显著特性叫做 Sync **同步**[22]**。当一个类型** T **实现了** sync **时，他表示着，在使用多个并发线程时，编译器没有可能引入不安全内存类型的内容。**

For example, sharing immutable data with an atomic reference count is threadsafe. Rust provides a type like this, Arc<T>, and it implements Sync,so it is safe to share between threads.

例如，使用一个原子级引用计数共享不可变数据室线程安全的。Rust **语言提供了一个像这样的类型——**Arc<T>, **它实现了** Sync, **所以它是安全的在线程之间共享内容。**

These two traits allow you to use the type system to make strong guarantees about the properties of your code under concurrency. Before we demonstrate why, we need to learn how to create a concurrent Rust program in the first place!

这两个显著特性允许你使用类型体系，从而在并发时给你代码属性的一个强大的保障。。在我们解释原因之前，我们需要学习，如何在一开始便创建一个并发的 Rust **程序。**

---

[19]http://doc.rust-lang.org/std/marker/trait.Send.html
[20]http://doc.rust-lang.org/std/marker/trait.Send.html
[21]http://doc.rust-lang.org/std/marker/trait.Sync.html
[22]http://doc.rust-lang.org/std/marker/trait.Sync.html

Threads 线程

Rust's standard library provides a library for threads, which allow you to run Rust code in parallel. Here's a basic example of using std::thread:

Rust **标准库提供一个线程库，这个库允许你并行运行** Rust **代码。这是一个使用** std::thread **的基本的例子：**

```
use std::thread;

fn main() {
    thread::spawn(|| {
        println!("Hello from a thread!");
    });
}
```

The thread::spawn() method accepts a closure, which is executed in a new thread. It returns a handle to the thread, that can be used to wait for the child thread to finish and extract its result:

thread::spawn() **方法接受一个闭包，这个闭包被执行在一个新的线程上。它返回一个线程的句柄，这个句柄可以用来等待子线程结束并提取结果：**

```
use std::thread;

fn main() {
    let handle = thread::spawn(|| {
        "Hello from a thread!"
    });

    println!("{}", handle.join().unwrap());
}
```

Many languages have the ability to execute threads, but it's wildly unsafe.There are entire books about how to prevent errors that occur from shared mutable state. Rust helps out with its type system here as well, by preventing data races at compile time. Let's talk about how you actually share things between threads.

许多语言具有执行多线程的能力，然而，大部分都不安全。有很多正本都是关于如何在共享可变状态时避免产生错的的书籍。同样， Rust **用它的类型体系通过避免在编译时产生数据溢出来帮助我们。让我们讨论一下如何真正的在线程间共享内容。**

Safe Shared Mutable State 安全共享的可变状态

Due to Rust's type system, we have a concept that sounds like a lie: "safe shared mutable state." Many programmers agree that shared mutable state is very, very bad.

因为 Rust 的类型体系，我们有了一个概念，听起来这像是一个谎言："安全共享的可变状态"。很多程序员都认为共享可变状态时非常非常可怕的。

Someone once said this:

有些人曾经这样说过：

> Shared mutable state is the root of all evil. Most languages attempt to deal with this problem through the 'mutable' part, but Rust deals with it by solving the 'shared' part.
> **共享可变状态时所有罪恶的根源。大多数语言试图通过可变部分来处理这个问题，然而 Rust 是通过解决共享部分来处理它的。**

The same ownership system (section 5.8, page 167) that helps prevent using pointers incorrectly also helps rule out data races, one of the worst kinds of concurrency bugs.

同样，ownership system 所有权体系 (section 5.8, page 167) 也是帮助避免使用错误的指针，同样帮助排除数据溢出——最坏情况下的并发错误。

As an example, here is a Rust program that would have a data race in many languages. It will not compile:

这里有一个例子，是一个 Rust 程序，在很多语言它会产生数据溢出。它不能编译：

```
use std::thread;

fn main() {
    let mut data = vec![1u32, 2, 3];

    for i in 0..3 {
        thread::spawn(move || {
            data[i] += 1;
        });
    }

    thread::sleep_ms(50);
}
```

This gives us an error:

这个会给出一个错误信息：

```
8:17 error: capture of moved value: `data`
      data[i] += 1;
      ^~~~
```

In this case, we know that our code *should* be safe, but Rust isn't sure. And it's actually not safe: if we had a reference to data in each thread, and the thread takes ownership of the reference, we have

three owners! That's bad. We can fix this by using the Arc<T> type, which is an atomic reference counted pointer. The 'atomic' part means that it's safe to share across threads.

在这个案例中，我们知道我们的代码 应当是安全的，然而，Rust **无法确定。事实上，它并不安全：如果我们有一个** data **的地址因为在每一个线程中，线程获得了地址引用的所有权，我们就有了三个拥有者！这是很可怕的。我们可以通过使用一个原子引用计数指针的** Arc<T> **类型来修复这个问题。原子部分意味着，他是安全跨线程共享的。**

Arc<T> assumes one more property about its contents to ensure that it is safe to share across threads: it assumes its contents are Sync. But in our case, we want to be able to mutate the value. We need a type that can ensure only one person at a time can mutate what's inside. For that, we can use the Mutex<T> type. Here's the second version of our code. It still doesn't work,but for a different reason:

Arc<T> **假定其内容多了一部分，一次来确定，他是跨线程安全共享的：它假定它的内容是** Sync**。然而在我们的例子中，我们想要的是能够改变这个值。我们需要一个能够确定在同一时间只有一个人能够改变内部内容。为此我们可以使用** Mutex<t> **类型。这是第二版的代码。因为另一个原因，它仍然不能够运行：**

```
use std::thread;
use std::sync::Mutex;

fn main() {
    let mut data = Mutex::new(vec![1u32, 2, 3]);

    for i in 0..3 {
        let data = data.lock().unwrap();
        thread::spawn(move || {
            data[i] += 1;
        });
    }

    thread::sleep_ms(50);
}
```

Here's the error:

这是错误信息：

```
<anon>:9:9: 9:22 error: the trait `core::marker::Send` is not implemented for the type
 `std::sync::mutex::MutexGuard<'_, collections::vec::Vec<u32>>` [E0277]
<anon>:11        thread::spawn(move || {
         ^~~~~~~~~~~~~
<anon>:9:9: 9:22 note: `std::sync::mutex::MutexGuard<'_, collections::vec::Vec<u32>>` c
 annot be sent between threads safely
<anon>:11        thread::spawn(move || {
         ^~~~~~~~~~~~~
```

You see, Mutex[23] has a lock[24] method which has this signature:

看明白了 Meutex[25]**有一个** lock **锁方法**[26]**，该方法拥有这个签名：**

```
fn lock(&self) -> LockResult<MutexGuard<T>>
```

Because Send is not implemented for MutexGuard<T>, we can't transfer the guard across thread boundaries, which gives us our error.

因为 Send **不能够实现** MutexGuard<T>**，所以我们无法跨线程边界进行防护，这给我们错误信息。**

We can use Arc<T> to fix this. Here's the working version:

我们可以使用 Arc<T> **来修复它。这是可以正常运行的版本代码：**

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

fn main() {
  let data = Arc::new(Mutex::new(vec![1u32, 2, 3]));

  for i in 0..3 {
    let data = data.clone();
    thread::spawn(move || {
      let mut data = data.lock().unwrap();
      data[i] += 1;
    });
  }

  thread::sleep_ms(50);
}
```

We now call clone() on our Arc, which increases the internal count. This handle is then moved into the new thread. Let's examine the body of the thread more closely:

我们现在在 Arc **上调用** clone()**，它能够增加内部计数。这个句柄将移动新的线程。让我们更加密切地检查下线程内部信息：**

```
thread::spawn(move || {
  let mut data = data.lock().unwrap();
  data[i] += 1;
});
```

---

[23]http://doc.rust-lang.org/std/sync/struct.Mutex.html
[24]http://doc.rust-lang.org/std/sync/struct.Mutex.html#method.lock
[25]http://doc.rust-lang.org/std/sync/struct.Mutex.html
[26]http://doc.rust-lang.org/std/sync/struct.Mutex.html#method.lock

First, we call lock(), which acquires the mutex's lock. Because this may fail,it returns an Result<T, E>, and because this is just an example, we unwrap() it to get a reference to the data. Real code would have more robust error handling here. We're then free to mutate it, since we have the lock.

首先我们调用 lock()，它获取互斥的锁状态。因为这个可能会失败，所以他返回一个 Result<T,E> 的信息，并且由于这只是一个例子，我们打开它，获取内部数据的引用。在这个地方，真正的代码将有更多的错误处理。一旦我们拥有了这个锁，我们随意地改变它，

Lastly, while the threads are running, we wait on a short timer. But this is not ideal: we may have picked a reasonable amount of time to wait but it's more likely we'll either be waiting longer than necessary or not long enough, depending on just how much time the threads actually take to finish computing when the program runs.

最后，当线程运行时，我们需要等待一小段时间。然而，这并不够理想：我们可能选择了一个合理的得带时间，但是取决于在程序运行时，线程真正结束计算小号的时间多少，它可能比我们的等待时间还长，或者比如等待时间长。

A more precise alternative to the timer would be to use one of the mechanisms provided by the Rust standard library for synchronizing threads with each other. Let's talk about one of them: channels.

一个更精准的替代计时器是使用一个由 Rust 标准库提供的，用于同步每一个线程的机制。下面让我们来讨论下他们中的一个：信道。

Channels 信道

Here's a version of our code that uses channels for synchronization, rather than waiting for a specific time:

这是代码使用信道同步的一个版本，而不是等待一个特定时间：

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

fn main() {
    let data = Arc::new(Mutex::new(0u32));

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

    for _ in 0..10 {
        let (data, tx) = (data.clone(), tx.clone());

        thread::spawn(move || {
            let mut data = data.lock().unwrap();
            *data += 1;

            tx.send(());
```

```
  });
}

for _ in 0..10 {
  rx.recv();
}
}
```

We use the mpsc::channel() method to construct a new channel. We just send a simple () down the channel, and then wait for ten of them to come back.

我们使用 mpsc::channel() **方式来构造一个新的信道。我们只是** send **一个简单地** () **给信道，然后等待他们十个回来。**

While this channel is just sending a generic signal, we can send any data that is Send over the channel!

然而，这个信道只是发送了一个通用的信号，我们可以通过信道发送任意数据！

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

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

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

    thread::spawn(move || {
      let answer = 42u32;

      tx.send(answer);
    });
  }

  rx.recv().ok().expect("Could not receive answer");
}
```

A u32 is Send because we can make a copy. So we create a thread, ask it to calculate the answer, and then it send()s us the answer over the channel.

u32 **被** Send **因为我们制造了一个副本。因此，我们创建了一个新的线程，请求他计算答案，然后它通过信道将答案** send() **回来。**

Panics

A panic! will crash the currently executing thread. You can use Rust's threads as a simple isolation mechanism:

pannic! **会使当前正在执行的线程崩溃。你可以使用** Rust **的线程作为一个简单地隔离机制：**

use std::thread;

```
let result = thread::spawn(move || {
    panic!("oops!");
}).join();
```

assert!(result.is_err());

Our Thread gives us a Result back, which allows us to check if the thread has panicked or not.

Thread **线程返回来一个** Result **结果以便我们能够检查线程是否崩溃。**

4.7 Error Handling **错误处理**

> The best-laid plans of mice and men Often go awry

> "Tae a Moose", Robert Burns "**不管是人是鼠，即使最如意的安排设计，结局也往往会出其不意。**" **致老鼠，罗伯特·彭斯**

Sometimes, things just go wrong. It's important to have a plan for when the inevitable happens. Rust has rich support for handling errors that may (let's be honest: will) occur in your programs.

有些时候，程序就是出错。所以当事情不可避免的发生时，有一个计划是非常重要的。Rust **拥有丰富的错误处理机制，或许就会（说实话，应该将会）在你的程序中出现。**

There are two main kinds of errors that can occur in your programs: failures,and panics. Let's talk about the difference between the two, and then discuss how to handle each. Then, we'll discuss upgrading failures to panics.

在程序中发生的错误主要有两大主要错误：失败和 panics**。让我们先谈谈两者的区别，然后再讨论如何处理他们。最后我们将讨论将失败升级为** panics**。**

4.7.1 Failure vs. Panic **失败和** panic

Rust uses two terms to differentiate between two forms of error: failure, and panic. A *failure* is an error that can be recovered from in some way. A *panic* is an error that cannot be recovered from.

Rust **语言有两种术语来区分两种形式的错误：失败和** panic**。***failure* **是能够以某种方式恢复的错误。而** *panic* **则是不能够恢复的错误。**

What do we mean by "recover"? Well, in most cases, the possibility of an error is expected. For example, consider the parse function:

那么恢复对我们来说意味着什么？好吧，在大多数情况下，错误是能够被预见到的。例如，参考一下 parse **函数：**

```
"5".parse();
```

This method converts a string into another type. But because it's a string, you can't be sure that the conversion actually works. For example, what should this convert to?

这个方法是转换字符串到另一个类型。但是因为他是一个字符串，我们不能够确定转换能够实际发生。例如这个应该转换成什么？

```
"hello5world".parse();
```

This won't work. So we know that this function will only work properly for some inputs. It's expected behavior. We call this kind of error a *failure.*

这是无法运行的。尽管我们知道，这个方法只对某些输入正常解析。他是预期行为。我们成这种错误叫做 *failure*。

On the other hand, sometimes, there are errors that are unexpected, or which we cannot recover from. A classic example is an assert!:

另一方面，有时候，有些错误是无法预见的，或者是我们无法进行转换的。一个经典的例子是一个 assert!:

```
assert!(x == 5);
```

We use assert! to declare that something is true. If it's not true, something is very wrong. Wrong enough that we can't continue with things in the current state. Another example is using the unreachable!() macro:

我们使用 assert! **来声明某些事情是正确的。如果它不是正确的，事情就会非常糟糕。错误以致我们在当前状态下无法继续。另一个例子是使用** unreachable!() **宏：**

```
enum Event {
  NewRelease,
}

fn probability(_: &Event) -> f64 {
  // real implementation would be more complex, of course
  0.95
}

fn descriptive_probability(event: Event) -> &'static str {
  match probability(&event) {
    1.00 => "certain",
    0.00 => "impossible",
    0.00 ... 0.25 => "very unlikely",
    0.25 ... 0.50 => "unlikely",
    0.50 ... 0.75 => "likely",
```

```
      0.75 ... 1.00 => "very likely",
   }
}

fn main() {
   std::io::println(descriptive_probability(NewRelease));
}
```

This will give us an error:

这将如下显示错误：

error: non-exhaustive patterns: `_` not covered [E0004]

While we know that we've covered all possible cases, Rust can't tell. It doesn't know that probability is between 0.0 and 1.0. So we add another case:

尽管我们知道我们已经涵盖了所有可能的情况， Rust **不能。他不知道的概率在** 0.0 **到** 1.0。**所以我们又举出了另一个例子：**

```
use Event::NewRelease;

enum Event {
   NewRelease,
}

fn probability(_: &Event) -> f64 {
   // real implementation would be more complex, of course
   0.95
}

fn descriptive_probability(event: Event) -> &'static str {
   match probability(&event) {
      1.00 => "certain",
      0.00 => "impossible",
      0.00 ... 0.25 => "very unlikely",
      0.25 ... 0.50 => "unlikely",
      0.50 ... 0.75 => "likely",
      0.75 ... 1.00 => "very likely",
      _ => unreachable!()
   }
}

fn main() {
   println!("{}", descriptive_probability(NewRelease));
}
```

We shouldn't ever hit the _ case, so we use the unreachable!() macro to indicate this. unreachable!() gives a different kind of error than Result.Rust calls these sorts of errors *panics*.

我们永远无法命中_情况，所以我们使用了 unreachable!() 宏来表明这一点。unreachable!() 产生了一个不同的错误而不是 Result。Rust 语言中，这些类型的错误统称为 panics。

4.7.2   Handling errors with **Option** and **Result** 使用 **option** 选项和 **Result** 结果来处理错误信息

The simplest way to indicate that a function may fail is to use the Option<T> type. For example, the find method on strings attempts to find a pattern in a string, and returns an Option:

表示一个函数可能失败的最简单方式是使用 Option<T> 类型。举个栗子，基于字符串的 find 方法，试图在一个字符串中找到一个模式，并返回一个 Option:

```
let s = "foo";

assert_eq!(s.find('f'), Some(0));
assert_eq!(s.find('z'), None);
```

This is appropriate for the simplest of cases, but doesn't give us a lot of information in the failure case. What if we wanted to know *why* the function failed? For this, we can use the Result<T, E> type. It looks like this:

这个只能适用于最简单的情况，也无法给我提供在失败情况下更多的信息。如果我们想知道，为什么这个方法会失败？为此，我们可以使用 Result<T,E> 类型。它看起来是这样子的：

```
enum Result<T, E> {
  Ok(T),
  Err(E)
}
```

This lumen is provided by Rust itself, so you don't need to define it to use it in your code. The Ok(T) variant represents a success, and the Err(E) variant represents a failure. Returning a Result instead of an Option is recommended for all but the most trivial of situations.

枚举类型是由 Rust 本身提供的，所以我们在代码中，不需要定义它即可使用。Ok<T> 变量代表着成功，Err<T> 变量代表着失败。除了最简单的情况，我们推荐你使用返回一个 Return 而不是使用一个 Option。

Here's an example of using Result:

这里有一个使用 Result 的例子：

```
#[derive(Debug)]
enum Version { Version1, Version2 }
```

```
#[derive(Debug)]
enum ParseError { InvalidHeaderLength, InvalidVersion }

fn parse_version(header: &[u8]) -> Result<Version, ParseError> {
  if header.len() < 1 {
    return Err(ParseError::InvalidHeaderLength);
  }
  match header[0] {
    1 => Ok(Version::Version1),
    2 => Ok(Version::Version2),
    _ => Err(ParseError::InvalidVersion)
  }
}

let version = parse_version(&[1, 2, 3, 4]);
match version {
  Ok(v) => {
    println!("working with version: {:?}", v);
  }
  Err(e) => {
    println!("error parsing header: {:?}", e);
  }
}
```

This function makes use of an enum, ParseError, to enumerate the various errors that can occur.

这个方法使用了一个枚举类型的变量 ParsError，来列举可能发生的各种错误。

The Debug[27] trait is what lets us print the enum value using the {:?} format operation.

Debug[28]特性使用 {:?} 格式化操作打印枚举值。

4.7.3 Non-recoverable errors with **panic!** 无法恢复的错误 **panic**

In the case of an error that is unexpected and not recoverable, the panic! macro will induce a panic. This will crash the current thread, and give an error:

在这个错误的案例中，错误是不能够预见的，且无法恢复的，panic 宏将引发一个 panic。这将会导致当前线程崩溃，并给出如下错误：

```
panic!("boom");
```

gives

---

[27]http://doc.rust-lang.org/std/fmt/trait.Debug.html
[28]http://doc.rust-lang.org/std/fmt/trait.Debug.html

```
thread '<main>' panicked at 'boom', hello.rs:2
```

when you run it.

当你运行它时，会给出如上信息。

Because these kinds of situations are relatively rare, use panics sparingly.

由于这种情况那个是很少见的，所以应该尽量避免使用 panics。

4.7.4   Upgrading failures to panics 将失败升级为 panic

In certain circumstances, even though a function may fail, we may want to treat it as a panic instead. For example, io::stdin().read_line(&mut buffer) returns a Result<usize>, when there is an error reading the line. This allows us to handle and possibly recover from error.

在某些确定的情况下，即使一个函数可能会失败，我们可能要将它当作 panic **对待。例如，当有一行内容读取错误时，** io::stdin().read_line(&mut buffer) **返回一个** Result<usize>。**这允许我们处理，并且可能恢复错误。**

If we don't want to handle this error, and would rather just abort the program,we can use the unwrap() method:

如果我们不希望处理此错误，宁愿退出程序，我们可以使用 unwrap() **方法：**

```
io::stdin().read_line(&mut buffer).unwrap();
```

unwrap() will panic! if the Result is Err. This basically says "Give me the value, and if something goes wrong, just crash." This is less reliable than matching the error and attempting to recover, but is also significantly shorter. Sometimes, just crashing is appropriate.

如果 Result **是** Err **类型，** unwrap() **将引起** panic!。**基本上来就是** "给我那个值，如果出错，只管崩溃!" **比起匹配错误，并且试图恢复它，这是更不可靠的，然而这样做明显的简短。**

There's another way of doing this that's a bit nicer than unwrap():

还有另一种方式的做法要比 unwrap() **要好：**

```
let mut buffer = String::new();
let input = io::stdin().read_line(&mut buffer)
            .ok()
            .expect("Failed to read line");
```

ok() converts the Result into an Option, and expect() does the same thing as unwrap(), but takes a message. This message is passed along to the underlying panic!, providing a better error message if the code errors.

ok() **将** Result **转换成** Option，expect() **对** unwrap() **做同样的事情，但是会返回一个消息。这个消息被传递给底层的** panic!，**如果代码出错了，这个底层的** panic! **会提供了一个更好的错误消息**

4.7.5   Using **try!** 使用 **try!**

When writing code that calls many functions that return the Result type, the error handling can be tedious. The try! macro hides some of the boilerplate of propagating errors up the call stack.

当编写的代码调用了很多返回 Result **类型的方法时，错误处理变得枯燥乏味。** try! **宏隐藏了一些会传播错误信息到调用堆栈上的模板。**

It replaces this:

它取代了这一点

```rust
use std::fs::File;
use std::io;
use std::io::prelude::*;

struct Info {
    name: String,
    age: i32,
    rating: i32,
}

fn write_info(info: &Info) -> io::Result<()> {
    let mut file = File::create("my_best_friends.txt").unwrap();

    if let Err(e) = writeln!(&mut file, "name: {}", info.name) {
        return Err(e)
    }
    if let Err(e) = writeln!(&mut file, "age: {}", info.age) {
        return Err(e)
    }
    if let Err(e) = writeln!(&mut file, "rating: {}", info.rating) {
        return Err(e)
    }

    return Ok(());
}
```

With this:

使用这个

```rust
use std::fs::File;
use std::io;
use std::io::prelude::*;

struct Info {
    name: String,
```

```rust
    age: i32,
    rating: i32,
}

fn write_info(info: &Info) -> io::Result<()> {
    let mut file = try!(File::create("my_best_friends.txt"));

    try!(writeln!(&mut file, "name: {}", info.name));
    try!(writeln!(&mut file, "age: {}", info.age));
    try!(writeln!(&mut file, "rating: {}", info.rating));

    return Ok(());
}
```

Wrapping an expression in try! will result in the unwrapped success (Ok) value, unless the result is Err, in which case Err is returned early from the enclosing function.

在 try! 中包裹一个表达式，将可以得到一个成功的 (Ok) 值，除非结果是 Err, 在每一种情况下，Err 总是从封闭的函数中及早返回。

It's worth noting that you can only use try! from a function that returns a Result, which means that you cannot use try! inside of main(), because main() doesn't return anything.

值得一提的是，你只能够对能够返回 Result 值的函数使用 try! 类型，这意味着，你不能够在 main() 方法中使用，因为 main() 不会返回任何内容。

try! makes use of From<Error>[29] to determine what to return in the error case.

try! 是使用 from<Error>[30]来决定在错误情况下返回什么内容。

4.8　FFI 对外函数接口

4.8.1　Introduction 说明

This guide will use the snappy[31] compression/decompression library as an introduction to writing bindings for foreign code. Rust is currently unable to call directly into a C++ library, but snappy includes a C interface (documented in snappy-c.h[32]).

本书将使用 snappy[33]压缩发压缩库作为例子来编写外部代码的变量绑定。目前 Rust 语言不能够直接使用 C++ 库，然而 snappy 含有一个 c 接口 (记录在 snappy-c.h[34]).

---

[29]http://doc.rust-lang.org/std/convert/trait.From.html
[30]http://doc.rust-lang.org/std/convert/trait.From.html
[31]https://github.com/google/snappy
[32]https://github.com/google/snappy/blob/master/snappy-c.h
[33]https://github.com/google/snappy
[34]https://github.com/google/snappy/blob/master/snappy-c.h

The following is a minimal example of calling a foreign function which will compile if snappy is installed:

下面就是一个简单的调用外部函数的例子，如果 snappy 已经被安装了，那么他将会被编译。

```
# #![feature(libc)]
extern crate libc;
use libc::size_t;

#[link(name = "snappy")]
extern {
    fn snappy_max_compressed_length(source_length: size_t) -> size_t;
}

fn main() {
    let x = unsafe { snappy_max_compressed_length(100) };
    println!("max compressed length of a 100 byte buffer: {}", x);
}
```

The extern block is a list of function signatures in a foreign library, in this case with the platform's C ABI. The #[link(...)] attribute is used to instruct the linker to link against the snappy library so the symbols are resolved.

extern 代码块是一个外部库的函数标记列表，在本案例中使用的是 C 平台 ABI。#[link(...)] 属性是用来声明一个可以映射到 snappy 库的连接器，解决了符号链接

Foreign functions are assumed to be unsafe so calls to them need to be wrapped with unsafe {} as a promise to the compiler that everything contained within truly is safe. C libraries often expose interfaces that aren't thread-safe, and almost any function that takes a pointer argument isn't valid for all possible inputs since the pointer could be dangling, and raw pointers fall outside of Rust's safe memory model.

外部函数被认为是不安全的，所以需要使用 unsafe{} 来封装他们，作为向编译器的一个保证，来确保包含在内部的任何内容真的是安全的。C 语言经常暴露一些非线程安全的接口，并且他们使用一个指针参数，因为指针可能为空指针，不能够验证所有可能的输入，从而导致原始指针落到 Rust 的安全内存模型之外的地方。

When declaring the argument types to a foreign function, the Rust compiler can not check if the declaration is correct, so specifying it correctly is part of keeping the binding correct at runtime.

当声明一个外部函数的参数类型时，Rust 编译器不能够检查是否正确，所以正确地指定类型是保证在运行时保持绑定正确的一部分。

The extern block can be extended to cover the entire snappy API:

extern 代码块能够被扩展来覆盖整个的 snappy API：

```
# #![feature(libc)]
extern crate libc;
use libc::{c_int, size_t};
```

```
#[link(name = "snappy")]
extern {
  fn snappy_compress(input: *const u8,
           input_length: size_t,
           compressed: *mut u8,
           compressed_length: *mut size_t) -> c_int;
  fn snappy_uncompress(compressed: *const u8,
            compressed_length: size_t,
            uncompressed: *mut u8,
            uncompressed_length: *mut size_t) -> c_int;
  fn snappy_max_compressed_length(source_length: size_t) -> size_t;
  fn snappy_uncompressed_length(compressed: *const u8,
               compressed_length: size_t,
               result: *mut size_t) -> c_int;
  fn snappy_validate_compressed_buffer(compressed: *const u8,
                compressed_length: size_t) -> c_int;
}
# fn main() {}
```

4.8.2 **Creating a safe interface 创建一个安全的接口**

The raw C API needs to be wrapped to provide memory safety and make use of higher-level concepts like vectors. A library can choose to expose only the safe, high-level interface and hide the unsafe internal details.

原始的 C API 需要提供安全内存和使用像向量这样的高级概念。一个库能够选择只暴露安全的、高级接口并隐藏内部不安全信息。

Wrapping the functions which expect buffers involves using the slice::raw module to manipulate Rust vectors as pointers to memory. Rust's vectors are guaranteed to be a contiguous block of memory. The length is number of elements currently contained, and the capacity is the total size in elements of the allocated memory. The length is less than or equal to the capacity.

封装具有缓冲区的方法包括使用 slice::raw **模块来作为内存指针来操纵 Rust 的向量。Rust 的向量是被保护为一个连续的内存块。长度就是当前内部元素的数量，容量是所有分配内存元素大小的总和。长度是远远小于或者等于容量的**

```
# #![feature(libc)]
# extern crate libc;
# use libc::{c_int, size_t};
# unsafe fn snappy_validate_compressed_buffer(_: *const u8, _: size_t) -> c_int { 0 }
# fn main() {}
pub fn validate_compressed_buffer(src: &[u8]) -> bool {
  unsafe {
    snappy_validate_compressed_buffer(src.as_ptr(), src.len() as size_t) == 0
  }
```

```
}
```

The validate_compressed_buffer wrapper above makes use of an unsafe block, but it makes the guarantee that calling it is safe for all inputs by leaving off unsafe from the function signature.

上面的 validate_compressed_buffer **封装使用了一个** unsafe **模块，然而，通过函数标记在离开** unsafe **封装时，它保证对于所有的输入操作，调用他是安全的。**

The snappy_compress and snappy_uncompress functions are more complex, since a buffer has to be allocated to hold the output too.

snappy_compress **和** snappy_uncompress **更加负责，因为一个缓冲区已经被分配用来保持输出。**

The snappy_max_compressed_length function can be used to allocate a vector with the maximum required capacity to hold the compressed output. The vector can then be passed to the snappy_compress function as an output parameter. An output parameter is also passed to retrieve the true length after compression for setting the length.

snappy_max_compressed_length **函数用来分配一个使用所需要的最大容量的向量，来保证压缩输出。这个向量随后可以被传递给** snappy_compress **作为一个输出参数。输出参数同样被传递用来检查为设定长度的压缩后的真实长度。**

```
# #![feature(libc)]
# extern crate libc;
# use libc::{size_t, c_int};
# unsafe fn snappy_compress(a: *const u8, b: size_t, c: *mut u8,
#                d: *mut size_t) -> c_int { 0 }
# unsafe fn snappy_max_compressed_length(a: size_t) -> size_t { a }
# fn main() {}
pub fn compress(src: &[u8]) -> Vec<u8> {
  unsafe {
    let srclen = src.len() as size_t;
    let psrc = src.as_ptr();

    let mut dstlen = snappy_max_compressed_length(srclen);
    let mut dst = Vec::with_capacity(dstlen as usize);
    let pdst = dst.as_mut_ptr();

    snappy_compress(psrc, srclen, pdst, &mut dstlen);
    dst.set_len(dstlen as usize);
    dst
  }
}
```

Decompression is similar, because snappy stores the uncompressed size as part of the compression format and snappy_uncompressed_length will retrieve the exact buffer size required.

解压缩是类似的，因为 snappy **将未压缩大小作为压缩格式的一部分来存储，** snappy_uncompressed_length **会校验所需要的精确地缓冲区大小。**

```
# #![feature(libc)]
# extern crate libc;
# use libc::{size_t, c_int};
# unsafe fn snappy_uncompress(compressed: *const u8,
#                compressed_length: size_t,
#                uncompressed: *mut u8,
#                uncompressed_length: *mut size_t) -> c_int { 0 }
# unsafe fn snappy_uncompressed_length(compressed: *const u8,
#                     compressed_length: size_t,
#                     result: *mut size_t) -> c_int { 0 }
# fn main() {}
pub fn uncompress(src: &[u8]) -> Option<Vec<u8>> {
  unsafe {
    let srclen = src.len() as size_t;
    let psrc = src.as_ptr();

    let mut dstlen: size_t = 0;
    snappy_uncompressed_length(psrc, srclen, &mut dstlen);

    let mut dst = Vec::with_capacity(dstlen as usize);
    let pdst = dst.as_mut_ptr();

    if snappy_uncompress(psrc, srclen, pdst, &mut dstlen) == 0 {
      dst.set_len(dstlen as usize);
      Some(dst)
    } else {
      None // SNAPPY_INVALID_INPUT
    }
  }
}
```

For reference, the examples used here are also available as a library on GitHub[35].

这里的例子同样参考适用于在 [github **上的库**][library on GitHub](https://github.com/thestinger/rust-snappy)**。**

4.8.3   Destructors 析构函数

Foreign libraries often hand off ownership of resources to the calling code.When this occurs, we must use Rust's destructors to provide safety and guarantee the release of these resources (especially in the case of panic).

外部库经常将资源的所有权交给调用带按摩。当这发生时，我们需要使用 Rust **语言的析构函数来保证安全和保障这些资源的额释放（尤其是在** panic **的情况下）。**

---

[35]https://github.com/thestinger/rust-snappy

For more about destructors, see the Drop trait[36].

想要了解更多析构函数相关的知识，可以参见 Drop trait 丢弃特性[37]。

4.8.4 Callbacks from C code to Rust functions 从 C 代码到 Rust 函数的回调方法

Some external libraries require the usage of callbacks to report back their current state or intermediate data to the caller.It is possible to pass functions defined in Rust to an external library.The requirement for this is that the callback function is marked as extern with the correct calling convention to make it callable from C code.

一些外部库需要使用回调方法来汇报它们的当前状态或者是调用的中间数据。在 Rust 中定义一个函数传递到外部库是可能的。要求是回调函数必须使用正确的调用约定标记为 extern，来确保他能够在 C 代码中调用到。

The callback function can then be sent through a registration call to the C library and afterwards be invoked from there.

回调函数能够被一个注册调用发送给 C 库，稍后从哪里被调用。

A basic example is:

基本栗子是这样子的:

Rust code:

这是 Rust 代码:

```
extern fn callback(a: i32) {
  println!("I'm called from C with value {0}", a);
}

#[link(name = "extlib")]
extern {
  fn register_callback(cb: extern fn(i32)) -> i32;
  fn trigger_callback();
}

fn main() {
  unsafe {
    register_callback(callback);
    trigger_callback(); // Triggers the callback
  }
}
```

---

[36]http://doc.rust-lang.org/std/ops/trait.Drop.html
[37]http://doc.rust-lang.org/std/ops/trait.Drop.html

C code:

这是 C 代码：

```
typedef void (*rust_callback)(int32_t);
rust_callback cb;

int32_t register_callback(rust_callback callback) {
  cb = callback;
  return 1;
}

void trigger_callback() {
 cb(7); // Will call callback(7) in Rust
}
```

In this example Rust's main() will call trigger_callback() in C,which would, in turn, call back to callback() in Rust.

在这个例子中， Rust **的** main() **将调用** C **中的** trigger_callback(), **反过来，他将会回调** Rust **中的** callback()

Targeting callbacks to Rust objects 定位 Rust 对象的回调

The former example showed how a global function can be called from C code.However it is often desired that the callback is targeted to a special Rust object. This could be the object that represents the wrapper for the respective C object.

前面一个例子说明了，一个全局的函数是如何被 C 代码调用的。然而，定位回调给一个 Rust 对象是经常被期望的。这可能是一个表示着封装了各个 C 对象的对象。

This can be achieved by passing an unsafe pointer to the object down to the C library. The C library can then include the pointer to the Rust object in the notification. This will allow the callback to unsafely access the referenced Rust object.

这个能够通过传递一个不安全的指向这个对象的指针到 C 库中。然后 C 库将这个 Rust 对象的指针包含进声明中。这将允许回调对引用的 Rust 对象进行不安全地存取。

Rust code:

Rust **代码：**

```
#[repr(C)]
struct RustObject {
  a: i32,
  // other members
}
```

```rust
extern "C" fn callback(target: *mut RustObject, a: i32) {
    println!("I'm called from C with value {0}", a);
    unsafe {
        // Update the value in RustObject with the value received from the callback
        (*target).a = a;
    }
}

#[link(name = "extlib")]
extern {
    fn register_callback(target: *mut RustObject,
                cb: extern fn(*mut RustObject, i32)) -> i32;
    fn trigger_callback();
}

fn main() {
    // Create the object that will be referenced in the callback
    let mut rust_object = Box::new(RustObject { a: 5 });

    unsafe {
        register_callback(&mut *rust_object, callback);
        trigger_callback();
    }
}
```

C code:

C **代码:**

```c
typedef void (*rust_callback)(void*, int32_t);
void* cb_target;
rust_callback cb;

int32_t register_callback(void* callback_target, rust_callback callback) {
    cb_target = callback_target;
    cb = callback;
    return 1;
}

void trigger_callback() {
  cb(cb_target, 7); // Will call callback(&rustObject, 7) in Rust
}
```

Asynchronous callbacks 异步回调

In the previously given examples the callbacks are invoked as a direct reaction to a function call to the external C library. The control over the current thread is switched from Rust to C to Rust for the execution of the callback, but in the end the callback is executed on the same thread that called the function which triggered the callback.

前面给出的例子中回调函数是作为一个调用外部 C 库的函数的直接反射来调用的。通过执行回调方法，当前线程的控制权从 Rust 转移到 C 中，然后又从 C 中转移到 Rust 中，然而直到最后，回调还是被执行在引发回调的函数所调用的同一个线程上。

Things get more complicated when the external library spawns its own threads and invokes callbacks from there. In these cases access to Rust data structures inside the callbacks is especially unsafe and proper synchronization mechanisms must be used. Besides classical synchronization mechanisms like mutexes, one possibility in Rust is to use channels (in std::comm) to forward data from the C thread that invoked the callback into a Rust thread.

当外部库派生出自己的线程，并从哪里调用回调时，事情变得更加复杂。在这种情况下，存取回调中 Rust 数据结构是特别不安全的，所以必须有一个适当的同步机制。除了经典的互斥变量同步机制，在 Rust 语言中有一个可能性是使用信道（在 std::comm 中）来转发数据到能够调用回到到一个 Rust 线程的 C 线程中。

If an asynchronous callback targets a special object in the Rust address space it is also absolutely necessary that no more callbacks are performed by the C library after the respective Rust object gets destroyed.This can be achieved by unregistering the callback in the object's destructor and designing the library in a way that guarantees that no callback will be performed after deregistration.

如果一个异步回调针对着 Rust 地址空间中的一个特殊对象，各自的 Rust 对象被销毁之后没有 C 库中不会执行任何回调时绝对必要的。这可以通过在对象析构函数中反注册回调并将库设计为一种能够确保在反注册后没有任何回调会被执行的方式来实现。

4.8.5   Linking

The link attribute on extern blocks provides the basic building block for instructing rustc how it will link to native libraries. There are two accepted forms of the link attribute today:

在 extern 代码块的 link 属性提供了一个基本的构建块来告诉 rustc 如何连接到一个本地库文件。有两种可以被接受的 link 属性：

- #[link(name = "foo")]
- #[link(name = "foo", kind = "bar")]

In both of these cases, foo is the name of the native library that we're linking to, and in the second case bar is the type of native library that the compiler is linking to. There are currently three known types of native libraries: **在这两种形式中，foo 是我们要连接到的本地库文件的名字，在第二种形式中，bar 是编译器正在连接的本地库文件的类型。一共有三种已知的本地库类型：**

- Dynamic **动态**- #[link(name = "readline")]

- Static 静态 - #[link(name = "my_build_dependency", kind = "static")]
- Frameworks 框架 - #[link(name = "CoreFoundation", kind = "framework")]

Note that frameworks are only available on OSX targets.

请注意，框架类型仅仅适用于 osx 系统。

The different kind values are meant to differentiate how the native library participates in linkage. From a linkage perspective, the rust compiler creates two flavors of artifacts: partial (rlib/staticlib) and final (dylib/binary).Native dynamic libraries and frameworks are propagated to the final artifact boundary, while static libraries are not propagated at all.

不同的 kind 类型值旨在区分本地库文件是何种方式参与到连接中的。通过链接方式来看，Rust 语言编译器创建了两种形式的东西：局部的（rlib/staticlib）和最终的 (dylib/binary)。本地动态库和框架被传播给最终编译文件，静态库文件则根本不会被传播。

A few examples of how this model can be used are:

下面是介绍如何使用这个模型的一些例子：

- A native build dependency. Sometimes some C/C++ glue is needed when writing some rust code, but distribution of the C/C++ code in a library format is just a burden. In this case, the code will be archived into libfoo.a and then the rust crate would declare a dependency via #[link(name = "foo", kind ="static")].
- **一个原生构建依赖。写一些 Rust 代码时，有时需要一些 C/C++，然而，在库文件中 C/C++ 的分发正式一个麻烦。在这种情况下，该代码被归档进 libfoo.a，然后后 Rust Crate 将通过 #[link(name="foo",kind="static")] 来声明一个依赖。**

Regardless of the flavor of output for the crate, the native static library will be included in the output, meaning that distribution of the native static library is not necessary.

无论为 crate 输出何种形式，本地静态库将被包含进输出中，这意味本地静态库的分发不是必须的。

- A normal dynamic dependency. Common system libraries (like readline) are available on a large number of systems, and often a static copy of these libraries cannot be found. When this dependency is included in a rust crate,partial targets (like rlibs) will not link to the library, but when the rlib is included in a final target (like a binary), the native library will be linked in.

一个正常的动态依赖。在大多数系统中，普通的系统库文件 (比如 readline) 是可用的，然而，这些苦文件的静态副本确实常常无法找到。当这种依赖被包含在 rust crate 中时，部分目标 (比如 rlibs) 将不会连接到这个库文件，然而，当 rlib 会被包含进最终的目标（比如二进制文件），本地库文件也将被包含进来。

On OSX, frameworks behave with the same semantics as a dynamic library.

在 OSX 系统中，框架作为动态库表现相同的语境。

4.8.6   Unsafe blocks **不安全代码块**

Some operations, like dereferencing unsafe pointers or calling functions that have been marked unsafe are only allowed inside unsafe blocks. Unsafe blocks isolate unsafety and are a promise to the compiler that the unsafety does not leak out of the block.

有些操作，比如取消不安全的指针引用或者已经被标记为不安全的调用函数只在不安全代码块中才被允许。不安全代码块隔离了不安全，并且向编译器承诺，不安全代码不会泄露。

Unsafe functions, on the other hand, advertise it to the world. An unsafe function is written like this:

另一方面，不安全的函数，宣扬他给所有的代码。一个不安全函数应该这样写：

```
unsafe fn kaboom(ptr: *const i32) -> i32 { *ptr }
```

This function can only be called from an unsafe block or another unsafe function.

这个函数只能够从一个 unsafe **不安全代码块或者另一个不安全函数中条用。**

4.8.7   Accessing foreign globals **全局外部访问**

Foreign APIs often export a global variable which could do something like track global state. In order to access these variables, you declare them in extern blocks with the static keyword:

外部 API **接口常常暴露一个全局变量，他可以做一些像跟踪全局状态类的事情。为了访问这些变来那个，你需要在** extern **代码块中使用** static **关键词来声明他们：**

```
# #![feature(libc)]
extern crate libc;

#[link(name = "readline")]
extern {
  static rl_readline_version: libc::c_int;
}

fn main() {
  println!("You have readline version {} installed.",
      rl_readline_version as i32);
}
```

Alternatively, you may need to alter global state provided by a foreign interface. To do this, statics can be declared with mut so we can mutate them.

或者，你可能需要同意一个外部接口来修改全局状态。为此，可以使用 mut **来声明这些静态变量，以便我们可以改变他们。**

```
# #![feature(libc)]
extern crate libc;

use std::ffi::CString;
use std::ptr;

#[link(name = "readline")]
extern {
    static mut rl_prompt: *const libc::c_char;
}

fn main() {
    let prompt = CString::new("[my-awesome-shell] $").unwrap();
    unsafe {
        rl_prompt = prompt.as_ptr();

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

        rl_prompt = ptr::null();
    }
}
```

Note that all interaction with a `static mut` is unsafe, both reading and writing. Dealing with global mutable state requires a great deal of care.

需要注意的是，所有使用 `static mut` 的接口，无论读写，都是不安全的。处理全局可变状态需要非常非常细心关照。

4.8.8 Foreign calling conventions 外部调用约定

Most foreign code exposes a C ABI, and Rust uses the platform's C calling convention by default when calling foreign functions. Some foreign functions, most notably the Windows API, use other calling conventions. Rust provides a way to tell the compiler which convention to use:

大多数外部代码暴露一个 C ABI，当调用外部函数时，Rust 默认使用 C 平台的调用协议。一些外部函数，特别是 Windows API，使用另外一些调用约定。Rust 语言提供了一种方式，来告诉编译器使用哪一种约定：

```
# #![feature(libc)]
extern crate libc;

#[cfg(all(target_os = "win32", target_arch = "x86"))]
#[link(name = "kernel32")]
#[allow(non_snake_case)]
extern "stdcall" {
    fn SetEnvironmentVariableA(n: *const u8, v: *const u8) -> libc::c_int;
```

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

This applies to the entire extern block. The list of supported ABI constraints are:

这适用于全部的 extern **代码块。支持** ABI **约定的列表如下：**

- stdcall
- aapcs
- cdecl
- fastcall
- Rust
- rust-intrinsic
- system
- C
- win64

Most of the abis in this list are self-explanatory, but the system abi may seem a little odd. This constraint selects whatever the appropriate ABI is for interoperating with the target's libraries. For example, on win32 with a x86 architecture, this means that the abi used would be stdcall. On x86_64, however, windows uses the C calling convention, so C would be used. This means that in our previous example, we could have used extern "system" { ... } to define a block for all windows systems, not just x86 ones.

大多数在次列表中的 ABI **是不言自明的，但是，**systemABI **可能显得有些奇怪。此约束选择任何适用的** SPI **来与目标库文件进行相互操作。比如，在使用** X86 **架构的** win32 **系统上，这意味着** ABI **使用的将是** stdcall。**然而，在** x86_64 **架构上，**windows **系统使用**C **调用约定，所有**C **将被使用。这意味着，在我们之前的例子中，我们可以使用** Extern "system" {...} **来为所有的** windows **系统来定义一个代码块，而不仅仅是** x86 **系统，**

4.8.9   Interoperability with foreign code

Rust guarantees that the layout of a struct is compatible with the platform's representation in C only if the #[repr(C)] attribute is applied to it. #[repr(C, packed)] can be used to lay out struct members without padding. #[repr(C)] can also be applied to an enum.

Rust **语言保证** struct **布局是在** C **中是适用于平台级别的表现，除非** #[repr(C)] **属性被用来适用它。** #[repr(C, packed)] **能够被用来布局结构成员而不是填充。**#[repr(C)] **同样可应用于一个枚举。**

Rust's owned boxes (Box<T>) use non-nullable pointers as handles which point to the contained object. However, they should not be manually created because they are managed by internal allocators. References can safely be assumed to be non-nullable pointers directly to the type. However, breaking the borrow checking or mutability rules is not guaranteed to be safe, so prefer using raw pointers (*) if that's needed because the compiler can't make as many assumptions about them.

Rust **自有的** boxes（ Box<T> ）**使用一个非空指针作为指向包含对象的句柄。然而，它们不应该被手动创建，因为他们有内部分配器进行管理。地址引用可以安全的假定为指向这种类型的非空指针。然而打断引用检测后者可变规则是不能够保证安全的，因此如果非必要请尽量使用原始指针（ * ），因为编译器不能够对他们进行尽可能多的推断。**

Vectors and strings share the same basic memory layout, and utilities are available in the vec and str modules for working with C APIs. However, strings are not terminated with \0. If you need a NUL-terminated string for interoperability with C, you should use the CString type in the std::ffi module.

向量和字符串共享同一个基本的内存布局，utilities **在使用** C API **时，适用于** Vec **和** str **模块。然而，字符串不会以**\0 **终止。如果你需要一个与** C **相互操作的没有结尾的字符串，你需要在** std::ffi **模块中使用** CSting **类型。**

The standard library includes type aliases and function definitions for the C standard library in the libc module, and Rust links against libc and libm by default.

4.8.10   The "nullable pointer optimization" 空指针的优化

Certain types are defined to not be null. This includes references (&T,&mut T), boxes (Box<T>), and function pointers (extern "abi" fn()).When interfacing with C, pointers that might be null are often used.As a special case, a generic enum that contains exactly two variants, one of which contains no data and the other containing a single field, is eligible for the "nullable pointer optimization". When such an enum is instantiated with one of the non-nullable types, it is represented as a single pointer, and the non-data variant is represented as the null pointer. So Option<extern "C" fn(c_int) -> c_int> is how one represents a nullable function pointer using the C ABI.

某些类型被定义为非 null。**这包括地址引用（** &T,&mut T ）**，**boxes（ Box<T> ）**，和函数指针（** extern "abi" fn() ）**。当使用** C **接口时，可能为空的指针式常常被使用的。作为一种特殊情况，一个通用的包括确定的两个变量体的** enum，**一个没有包含任何数据，另一个包含一个单独的字段，很可能就是一个"空指针优化"。当这样的枚举类型使用一个非空类型数据实例化之后，它被认为是一个单一指针，空数据变量被认为是一个空指针。所以** Option<extern "C" fn(c_int) -> c_int> **就是如何认定一个使用** C ABI **的空函数指针。**

4.8.11   Calling Rust code from C

You may wish to compile Rust code in a way so that it can be called from C. This is fairly easy, but requires a few things:

你可能希望用另一种方式编译 Rust **代码，那样，他就能够被** C **语言调用掉。这个是相当容易的，但是需要几个前提：**

```
#[no_mangle]
pub extern fn hello_rust() -> *const u8 {
    "Hello, world!\0".as_ptr()
}
# fn main() {}
```

The extern makes this function adhere to the C calling convention, as discussed above in "Foreign Calling Conventions[38]". The no_mangle attribute turns off Rust's name mangling, so that it is easier to link to.

extern **使得这个函数如上面讨论的外部调用约定**[39]**附加在** C **调用约定上。**no_mangle **属性关闭了** Rust **语言的名称重编，所以它可以很容易的被连接到。**

4.9   Borrow and AsRef 借用和引用

The Borrow[40] and AsRef[41] traits are very similar, but different. Here's a quick refresher on what these two traits mean.

Borrow **借用**[42]**和** AsRef **地址引用** '[43]**特性是非常相似的，但是却又有不同。这里是对于这两个特性代表着什么意思。**

4.9.1   Borrow 借用

The Borrow trait is used when you're writing a datastructure, and you want to use either an owned or borrowed type as synonymous for some purpose.

Borrow **借用特性用当你正在写一个数据结构，出于某种原因，你要么使用一个自有的类型要么使用一个借用类型作为代名词的时候，可以使用** Borrow **借用特性**

For example, HashMap[44] has a get method[45] which uses Borrow:

例如，HashMap[46]**有一个使用了** Borrow **借用特性的** get **方法**[47]**:**

```rust
fn get<Q: ?Sized>(&self, k: &Q) -> Option<&V>
  where K: Borrow<Q>,
      Q: Hash + Eq
```

This signature is pretty complicated. The K parameter is what we're interested in here. It refers to a parameter of the HashMap itself:

这个标记是相当复杂的。k **参数就是我们感兴趣的。它指向的是** HashMap **自身的一个参数:**

---

[38] ffi.html#foreign-calling-conventions
[39] ffi.html#foreign-calling-conventions
[40] http://doc.rust-lang.org/std/borrow/trait.Borrow.html
[41] http://doc.rust-lang.org/std/convert/trait.AsRef.html
[42] http://doc.rust-lang.org/std/borrow/trait.Borrow.html
[43] http://doc.rust-lang.org/std/convert/trait.AsRef.html
[44] http://doc.rust-lang.org/std/collections/struct.HashMap.html
[45] http://doc.rust-lang.org/std/collections/struct.HashMap.html#method.get
[46] http://doc.rust-lang.org/std/collections/struct.HashMap.html
[47] http://doc.rust-lang.org/std/collections/struct.HashMap.html#method.get

```
struct HashMap<K, V, S = RandomState> {
```

The K parameter is the type of *key* the HashMap uses. So, looking at the signature of get() again, we can use get() when the key implements Borrow<Q>. That way, we can make a HashMap which uses String keys,but use &strs when we're searching:

K **参数就是** HashMap **使用的** *Key* **的类型。所以再次看一下** get() **的标记，当** *key* **实现了** Borrow<Q> **后，我们可以使用** get() **方法。通过这种方式，我们可以做出一个** HashMap **，它使用的时** String **字符串键名，然而，当我们当我们查找时使用的时** &str:

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

let mut map = HashMap::new();
map.insert("Foo".to_string(), 42);

assert_eq!(map.get("Foo"), Some(&42));
```

This is because the standard library has impl Borrow<str> for String.

这是因为标准库拥有一个 impl Borrow<str> for String**。**

For most types, when you want to take an owned or borrowed type, a &T is enough. But one area where Borrow is effective is when there's more than one kind of borrowed value. Slices are an area where this is especially true: you can have both an &[T] or a &mut [T]. If we wanted to accept both of these types, Borrow is up for it:

对于大多数类型来说，当我们想要使用一个自由类型或者借用类型时，一个 &T **就足够了。然而有一个地方使用** Borrow **借用是高效的，那就是不只是一种借用值的时候。分片是一个尤其如此地区：你可以使用** &[T] **或者** &mut [T] **这两种。如果我们想要接受这两种类型，** Borrow **借用弥补了这一点：**

```
use std::borrow::Borrow;
use std::fmt::Display;

fn foo<T: Borrow<i32> + Display>(a: T) {
    println!("a is borrowed: {}", a);
}

let mut i = 5;

foo(&i);
foo(&mut i);
```

This will print out a is borrowed: 5 twice.

这将会打印 a is borrowed: 5 **两次。**

4.9.2   AsRef **地址引用**

The `AsRef` trait is a conversion trait. It's used for converting some value to a reference in generic code. Like this:

`AsRef` **特性就是转换特性。它用于转换一些值到通用代码中的参考。就像着这样：**

```
let s = "Hello".to_string();

fn foo<T: AsRef<str>>(s: T) {
  let slice = s.as_ref();
}
```

4.9.3   Which should I use? **我应该用哪一个？**

We can see how they're kind of the same: they both deal with owned and borrowed versions of some type. However, they're a bit different.

我们可以看到它们是多么的相似：它们都是处理一些类型的自有和借用版本。然而，它们有一点不同

Choose `Borrow` when you want to abstract over different kinds of borrowing, or when you're building a datastructure that treats owned and borrowed values in equivalent ways, such as hashing and comparison.

当你想要抽象化不同种类的借用，后者当你正在创建一个将自有和借用值以同等方式对待的数据结构时，请选择 `Borrow`**，比如哈希和比较。**

Choose `AsRef` when you want to convert something to a reference directly, and you're writing generic code.

当你正在写通用代码，想要直接转换某些内容为参考时请使用 `AsRef`**。**

4.10   Release Channels **发行通道**

The Rust project uses a concept called 'release channels' to manage releases. It's important to understand this process to choose which version of Rust your project should use.

Rust **项目使用了一个叫做 '发行通道' 的概念来管理发布。这对于理解你的项目应该应该选择哪一个版本的** Rust **来使用的过程很重要。**

4.10.1   Overview **概述**

There are three channels for Rust releases:

rust **发行版一共有三个通道：**

- Nightly **每日版本**
- Beta **测试版本**
- Stable **稳定版本**

New nightly releases are created once a day. Every six weeks, the latest nightly release is promoted to 'Beta'. At that point, it will only receive patches to fix serious errors. Six weeks later, the beta is promoted to 'Stable', and becomes the next release of 1.x.

每日版本是一天创建一次。每 6 个星期，最后一个每日版本被提升为测试版本。在这一个点上，它将只接受修复验证错误的补丁。6 周后，测试版本被晋升为正式版，并成为 1.x 的下一个版本。

This process happens in parallel. So every six weeks, on the same day, nightly goes to beta, beta goes to stable. When 1.x is released, at the same time, 1.(x + 1)-beta is released, and the nightly becomes the first version of 1.(x + 2)-nightly.

这个过程是平行发生的。所以每 6 周，在同一天，每日版本变成测试版，测试版本变成稳定版。当 1.x 被发布了，在同时 1.(x+1)-beta 也被发布，并且，每日版本变成了 1.(x+2)-nightly 的第一个版本。

4.10.2   Choosing a version 选择版本

Generally speaking, unless you have a specific reason, you should be using the stable release channel. These releases are intended for a general audience.

一般说来，除非你有一个特殊原因，你应该使用稳定版本通道。这些版本都是面向普通用户的。

However, depending on your interest in Rust, you may choose to use nightly instead. The basic tradeoff is this: in the nightly channel, you can use unstable, new Rust features. However, unstable features are subject to change, and so any new nightly release may break your code. If you use the stable release, you cannot use experimental features, but the next release of Rust will not cause significant issues through breaking changes.

然而，取决于你对 Rust 的兴趣，你可以攒泽使用每日版本。基本的权衡是这样的：在每日版本通道中，你可以使用不稳定的，新的 Rust 特性。但是，不稳定的特性可能会产生改变，因此任何新的每日版本都可以破坏你的代码。如果你使用了稳定版本，则不能够使用实验性特性，但是 Rust 的下一个版本将不会因为阻止变化产生明显的问题。

4.10.3   Helping the ecosystem through CI 通过 CI 来帮助生态链

What about beta? We encourage all Rust users who use the stable release channel to also test against the beta channel in their continuous integration systems. This will help alert the team in case there's an accidental regression.

为什么要测试？我们鼓励所有的使用稳定版本通道的 Rust 用户同样在他们的持续集成系统中尝试使用测试版本。

Additionally, testing against nightly can catch regressions even sooner, and so if you don't mind a third build, we'd appreciate testing against all channels.

此外，针对每日版本的测试能够更早的抓住回归，如果你不介意第三个版本，我们非常感谢你测试所有的通道。

5

Syntax and Semantics 语法和语义

This section breaks Rust down into small chunks, one for each concept.

这一章节将 Rust 语言分解成一小块一小块的，每一小块是一个概念。

If you'd like to learn Rust from the bottom up, reading this in order is a great way to do that.

如果你喜欢从基础开始学习 Rust，按照顺序阅读这一部分是完成这一目的的一个很好地方式

These sections also form a reference for each concept, so if you're reading another tutorial and find something confusing, you can find it explained somewhere in here.

这些章节同样是每一部分的一个索引，所以，如果你正在阅读另一个教程，感觉有些混乱，你可以在这里，找到某个地方的解释。

5.1 Variable Bindings 变量绑定

Virtually every non-'Hello World' Rust program uses *variable bindings.* They look like this:

几乎每一个非 "hello world" 的 Rust 程序都在使用了变量绑定。他们看起来是这样子的：

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

Putting fn main() { in each example is a bit tedious, so we'll leave that out in the future. If you're following along, make sure to edit your main() function, rather than leaving it off. Otherwise, you'll get an error.

在每一个例子中写入一个 fn main(){ 是枯燥无味的，所以我们将它留到以后。如果你跟随者教程，请务必编辑你的 main() 函数，而不是把它删掉，不然你会得到一个报错。

In many languages, this is called a *variable*, but Rust's variable bindings have a few tricks up their sleeves. For example the left-hand side of a let expression is a 'pattern[1]', not just a variable name. This means we can do things like:

在许多语言中，这被称之为变量，但是 Rust 的变量绑定有一些高招。例如，一个 let 表达式的左侧是一个 [pattern 模式][pattern]，而不是只有一个变量名。这意味着我们可以这样做：

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

After this expression is evaluated, x will be one, and y will be two.Patterns are really powerful, and have their own section[2] in the book. We don't need those features for now, so we'll just keep this in the back of our minds as we go forward.

这个表达式被求值后，x 将会是 1，y 将会是 2. 模式是非常强大的，并且在本书中有自己的章节[3]。现在我们不需要那些特性，我们将这些置于脑后，因为我们还要继续学习。

Rust is a statically typed language, which means that we specify our types up front, and they're checked at compile time. So why does our first example compile? Well, Rust has this thing called 'type inference'. If it can figure out what the type of something is, Rust doesn't require you to actually type it out.

Rust 是一门静态类型的语言，这意味着我们首先要声明我们的类型，并且在编译时会被检查。那么为什么我们的第一个例子能够编译通过呢？好吧，Rust 有个叫做 '类型接口' 的东西。如果 Rust 能够算出某些变量的类型是什么，它并不要求你真正的将它表示出来。

We can add the type if we want to, though. Types come after a colon (:):

然而，如果我们需要，我们可以增加类型。类型总是在一个冒号（:）后面：

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

If I asked you to read this out loud to the rest of the class, you'd say "x is a binding with the type i32 and the value five."

如果我要求你向全班其余同学大声的读出来，你应该说："x 是一个 i32 类型的变量，它的值是 5。"

In this case we chose to represent x as a 32-bit signed integer. Rust has many different primitive integer types. They begin with i for signed integers and u for unsigned integers. The possible integer sizes are 8, 16, 32, and 64 bits.

在这个案例中，我们选择声明 x 为一个 32 位的带符号的整数。Rust 有很多不同的原始整数类型。他们使用 i 标记带符号的整数，使用 u 标记不带符号的整数。能够被使用的整数字节是 8 位，16 位，32 位和 64 位。

In future examples, we may annotate the type in a comment. The examples will look like this:

在未来的案例中，我们可能在注释中声明一个类型。例子看起来是这样的：

---

[1]patterns.md
[2]patterns.md
[3]patterns.md

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

Note the similarities between this annotation and the syntax you use with `let`. Including these kinds of comments is not idiomatic Rust, but we'll occasionally include them to help you understand what the types that Rust infers are.

请注意这个声明和你使用 let **语法的相似性。包含这种注释不是地道的** Rust **代码编写方式，然而，我们会偶尔使用来帮助你理解** Rust **推断的类型是什么。**

By default, bindings are *immutable*. This code will not compile:

默认，变量绑定是不可变的。以下代码将无法编译：

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

It will give you this error:

它会给出错误：

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

If you want a binding to be mutable, you can use `mut`:

如果你想要一个可变的变量绑定，你可以使用 mut**：**

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

There is no single reason that bindings are immutable by default, but we can think about it through one of Rust's primary focuses: safety. If you forget to say `mut`, the compiler will catch it, and let you know that you have mutated something you may not have intended to mutate. If bindings were mutable by default, the compiler would not be able to tell you this. If you *did* intend mutation, then the solution is quite easy: add `mut`.

变量绑定默认为不可变的，有很多原因，但是我们可以考虑一下 Rust **的主要关注点之一：安全。如果你忘记标记** mut**，编译器将扑捉它，并且让你知道你已经改变了之前你没有预计到会改变的一个变量。如果变量绑定默认是可变的，编译器将不能够告诉你这个。如果你_希望_改变，解决方式很简单：增加** mut**。**

There are other good reasons to avoid mutable state when possible, but they're out of the scope of this guide. In general, you can often avoid explicit mutation, and so it is preferable in Rust. That said, sometimes, mutation is what you need, so it's not verboten.

在可能的情况下，还有其他一些好理由来避免可变状态，然而，他们超出了本教程的范围。通常情况下，你能够经常避免明确的可变类型，所以在 Rust 语言中，这种默认方式是最可取的。这就是说，有些时候，你需要可变类型，所以它们不被禁止。

Let's get back to bindings. Rust variable bindings have one more aspect that differs from other languages: bindings are required to be initialized with a value before you're allowed to use them.

让我们回到变量绑定上来。Rust 的变量绑定跟其他语言有不止一个的不同方面：在你被允许使用它们之前，变量绑定必须要有一个初始化值。

Let's try it out. Change your src/main.rs file to look like this:

让我们试一下。将你的 src/main.rs 文件更改成下面的样子：

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

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

You can use cargo build on the command line to build it. You'll get a warning, but it will still print "Hello, world!":

你可以在命令行中使用 cargo build 来构建它。你会得到一个警告，但是它仍然将打印出 "hello world！"：

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

Rust warns us that we never use the variable binding, but since we never use it, no harm, no foul. Things change if we try to actually use this x,however. Let's do that. Change your program to look like this:

Rust 警告我们我们没有使用变量绑定，因为我们没有使用它，没有害处，就没有犯规。然而，如果我们尝试真正的使用这个 x 变量，事情就会变化了。让我们那样做一下。改变你的程序如下：

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

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

And try to build it. You'll get an error:

然后试图构建它，你会得到一个错误：

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

Rust will not let us use a value that has not been initialized. Next, let's talk about this stuff we've added to println!.

Rust 将不会允许我们使用一个我们没有初始化的值。下一步，让我们讨论下我们加入到 println! 方法中的东西。

If you include two curly braces ({}, some call them moustaches...) in your string to print, Rust will interpret this as a request to interpolate some sort of value. *String interpolation* is a computer science term that means "stick in the middle of a string." We add a comma, and then x, to indicate that we want x to be the value we're interpolating. The comma is used to separate arguments we pass to functions and macros, if you're passing more than one.

如果你在你要打印的字符串中包含了两个花括号（ {}, 有些人称他们胡子），Rust 会推断他们为一个请求插入一些顺序的值的敌方。字符串插入是一个计算机科学术语，这表示 "插入到一个字符串的中间。" 我们增加一个逗号，然后是 x，来表名我们想要将 x 作为我们插入的值。如果你要传递不止一个参数的话，逗号被用于分割我们传递给函数和宏的参数。

When you just use the curly braces, Rust will attempt to display the value in a meaningful way by checking out its type. If you want to specify the format in a more detailed manner, there are a wide number of options available[4].For now, we'll just stick to the default: integers aren't very complicated to print.

当我们只需要使用花括号时，Rust 通过检查它的类型将试图以一种有意义的方式来显示这个值。如果我们想要指定更详细的格式，这里有一个很广泛的可选数字类型[5]。现在我们只使用默认：整数对打印来说并不是很复杂的。

5.2 Functions 函数

Every Rust program has at least one function, the main function:

每一个 Rust 程序至少都要有一个函数——main 函数：

---

[4]http://doc.rust-lang.org/std/fmt/index.html
[5]http://doc.rust-lang.org/std/fmt/index.html

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

This is the simplest possible function declaration. As we mentioned before,fn says 'this is a function', followed by the name, some parentheses because this function takes no arguments, and then some curly braces to indicate the body. Here's a function named foo:

这是最简单的函数声明。正如我们之前提到的，fn **表示 "这是一个函数"，紧跟着的是函数名，然后是一对括号，因为这个函数没有使用参数，然后是一对花括号来表示函数体本身。这里有一个叫做** foo **的函数：**

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

So, what about taking arguments? Here's a function that prints a number:

那么，有参数的话会怎样呢？这里有一个能够打印出一个数字的函数：

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

Here's a complete program that uses print_number:

这里有一个完成的程序使用 print_number **函数**;

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

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

As you can see, function arguments work very similar to let declarations:you add a type to the argument name, after a colon.

正如你看到的，函数参数非常想 let **声明：在冒号后面给参数名增加一个类型**

Here's a complete program that adds two numbers together and prints them:

这里有一个完整的例子，一次使用两个数字，并打印他们：

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

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

You separate arguments with a comma, both when you call the function, as well as when you declare it.

当你调用一个函数时，使用一个逗号来分割参数，就像你声明它时一样。

Unlike let, you *must* declare the types of function arguments. This does not work:

不像 let 声明，你 必须声明函数参数的类型。下面这样的程序不会运行：

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

You get this error:

你会得到错误：

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

This is a deliberate design decision. While full-program inference is possible,languages which have it, like Haskell, often suggest that documenting your types explicitly is a best-practice. We agree that forcing functions to declare types while allowing for inference inside of function bodies is a wonderful sweet spot between full inference and no inference.

这是一个深思熟虑的设计决定。虽然全程序推断是可能的，拥有这种方式的语言有 Haskell，经常建议明确的注释你的参数类型是一个最佳实践。我们同意在允许函数体内部推断的同时强制函数声明类型是全局推断和禁止推断之间的灵活点。

What about returning a value? Here's a function that adds one to an integer:

有返回值会怎样？ 这里有一个对整数 +1 函数：

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

Rust functions return exactly one value, and you declare the type after an 'arrow', which is a dash (-) followed by a greater-than sign (>). The last line of a function determines what it returns. You'll note the lack of a semicolon here. If we added it in:

Rust **函数的返回一个明确的值，你在一个 "箭头"（有一个破折号-紧跟着一个大于号 >）之后声明一个类型。函数的最后一行角定了它返回什么。你会注意到这里缺少一个分号。如果我们加入了它（分号）:**

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

We would get an error:

会报错

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

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

This reveals two interesting things about Rust: it is an expression-based language, and semicolons are different from semicolons in other 'curly brace and semicolon'-based languages. These two things are related.

这里揭示了 Rust 的两个有趣的内容：它是一个基于表达式的语言，并且分号不同于其他基于 "花括号和分号" 的语言。这两件事情是相关联的。

Expressions vs. Statements 表达式 VS. 声明

Rust is primarily an expression-based language. There are only two kinds of statements, and everything else is an expression.

Rust 是一个主要基于表达是的语言。只有两种声明类型，并且其他一切都是表达式。

So what's the difference? Expressions return a value, and statements do not.That's why we end up with 'not all control paths return a value' here: the statement x + 1; doesn't return a value. There are two kinds of statements in Rust: 'declaration statements' and 'expression statements'. Everything else is an expression. Let's talk about declaration statements first.

有什么区别呢？表达式返回一个值，声明不会。这就是为什么我们在这里使用 '不是全部控制路径返回值'：声明 x + 1; 不会返回值。在 Rust 中有两种声明方式：'宣布声明' 和 '表达式声明'。其他一切都是表达式。我们首先讨论下宣布声明；

In some languages, variable bindings can be written as expressions, not just statements. Like Ruby:

在某些语言中，变量绑定可以被写成表达式，而不只是一个声明。像是 Ruby 语言：

```ruby
x = y = 5
```

In Rust, however, using let to introduce a binding is *not* an expression. The following will produce a compile-time error:

然而在 Rust 中在一个变量绑定时使用 let 不是一个表达式。下面的代码将产生一个编译错误：

```
let x = (let y = 5); // expected identifier, found keyword `let`
```

The compiler is telling us here that it was expecting to see the beginning of an expression, and a let can only begin a statement, not an expression.

编译器在告诉我们这里被预期为一个表达式的开始，但是 let 只能够开始一个声明，不能够开始一个表达式。

Note that assigning to an already-bound variable (e.g. y = 5) is still an expression, although its value is not particularly useful. Unlike other languages where an assignment evaluates to the assigned value (e.g. 5 in the previous example), in Rust the value of an assignment is an empty tuple ():

需要注意的是，指定一个已经准备好的变量（例如 y = 5）仍然是一个表达式，尽管他的值不是特别有用。不像其他语言，一个赋值语句回来指定变量（例如前面一个例子中的 5），在 Rust 中，指定的值僵尸一个空的元组 ()：

```
let mut y = 5;

let x = (y = 6); // x has the value `()`, not `6`
```

The second kind of statement in Rust is the *expression statement*. Its purpose is to turn any expression into a statement. In practical terms, Rust's grammar expects statements to follow other statements. This means that you use semicolons to separate expressions from each other. This means that Rust looks a lot like most other languages that require you to use semicolons at the end of every line, and you will see semicolons at the end of almost every line of Rust code you see.

在 Rust 中的第二种声明是表达式声明。它的目的是将任意表达式转换成语句。就真实情况而言，Rus 的语法希望语句来跟随其他声明。这意味着，你使用分号来区分表达式。这也意味着 Rust 看起来像是其他大多数要求你在每一行的结束使用分号的语言，并且你将会在你见到的 Rust 代码的几乎每一行的末尾都有分号。

What is this exception that makes us say "almost"? You saw it already, in this code:

是什么让我们说"几乎"？你已经见过了，在这个代码中：

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

Our function claims to return an i32, but with a semicolon, it would return () instead. Rust realizes this probably isn't what we want, and suggests removing the semicolon in the error we saw before.

我们的函数声明返回一个 i32 整数，但是使用了分号后，他只能够反悔 ()。Rust 意识到这可能不是我们需要的，并会建议移除我们之前看到的错误中的分号。

Early returns 及早返回

But what about early returns? Rust does have a keyword for that, return:

什么是及早返回？Rust 有一个关键词 return：

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

  // we never run this code!
  x + 1
}
```

Using a `return` as the last line of a function works, but is considered poor style:

在函数的最后一行使用 return，是可以运行的，但是被认为是差的格式：

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

The previous definition without `return` may look a bit strange if you haven't worked in an expression-based language before, but it becomes intuitive over time.

如果你之前没有使用过一门基于表达式的语言的话，前面没有使用 return 的定义语句看起可能有点怪，但是随着时间的迁移，它将变得直观。

Diverging functions 发散函数

Rust has some special syntax for 'diverging functions', which are functions that do not return:

Rust 有一些特殊语法用于 '发散函数'，它是一个没有返回值的函数：

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

`panic!` is a macro, similar to `println!()` that we've already seen. Unlike `println!()`, `panic!()` causes the current thread of execution to crash with the given message.

panic! 是一个宏，跟我们已经见过的 println!() 相似。不像 println!(),panic!() 使用给定的信息致使执行的当前线程崩溃。

Because this function will cause a crash, it will never return, and so it has the type '!', which is read 'diverges'. A diverging function can be used as any type:

因为这个函数会致使一个崩溃，它永远没有返回值，所以他使用被称为 '发散' 的! 类型。一个发散函数可以被用于任意类型：

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

5.3 Primitive Types 原始数据类型

The Rust language has a number of types that are considered 'primitive'. This means that they're built-in to the language. Rust is structured in such a way that the standard library also provides a number of useful types built on top of these ones, as well, but these are the most primitive.

Rust **语言有很多被认为是原始数据类型的语言。**

5.3.1 Booleans

Rust has a built in boolean type, named bool. It has two values, true and false:

```rust
let x = true;

let y: bool = false;
```

A common use of booleans is in if conditionals (section 5.5, page 163).

You can find more documentation for bools in the standard library documentation[6].

5.3.2 char

The char type represents a single Unicode scalar value. You can create chars with a single tick: (')

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

Unlike some other languages, this means that Rust's char is not a single byte, but four.

You can find more documentation for chars in the standard library documentation[7].

5.3.3 Numeric types

Rust has a variety of numeric types in a few categories: signed and unsigned, fixed and variable, floating-point and integer.

These types consist of two parts: the category, and the size. For example, u16 is an unsigned type with sixteen bits of size. More bits lets you have bigger numbers.

If a number literal has nothing to cause its type to be inferred, it defaults:

---

[6]http://doc.rust-lang.org/std/primitive.bool.html
[7]http://doc.rust-lang.org/std/primitive.char.html

```
let x = 42; // x has type i32

let y = 1.0; // y has type f64
```

Here's a list of the different numeric types, with links to their documentation in the standard library:

- i8[8]
- i16[9]
- i32[10]
- i64[11]
- u8[12]
- u16[13]
- u32[14]
- u64[15]
- isize[16]
- usize[17]
- f32[18]
- f64[19]

Let's go over them by category:

Signed and Unsigned

Integer types come in two varieties: signed and unsigned. To understand the difference, let's consider a number with four bits of size. A signed, four-bit number would let you store numbers from -8 to +7. Signed numbers use "two's complement representation". An unsigned four bit number, since it does not need to store negatives, can store values from 0 to +15.

Unsigned types use a u for their category, and signed types use i. The i is for 'integer'. So u8 is an eight-bit unsigned number, and i8 is an eight-bit signed number.

---

[8]http://doc.rust-lang.org/std/primitive.i8.html
[9]http://doc.rust-lang.org/std/primitive.i16.html
[10]http://doc.rust-lang.org/std/primitive.i32.html
[11]http://doc.rust-lang.org/std/primitive.i64.html
[12]http://doc.rust-lang.org/std/primitive.u8.html
[13]http://doc.rust-lang.org/std/primitive.u16.html
[14]http://doc.rust-lang.org/std/primitive.u32.html
[15]http://doc.rust-lang.org/std/primitive.u64.html
[16]http://doc.rust-lang.org/std/primitive.isize.html
[17]http://doc.rust-lang.org/std/primitive.usize.html
[18]http://doc.rust-lang.org/std/primitive.f32.html
[19]http://doc.rust-lang.org/std/primitive.f64.html

Fixed size types

Fixed size types have a specific number of bits in their representation. Valid bit sizes are 8, 16, 32, and 64. So, u32 is an unsigned, 32-bit integer, and i64 is a signed, 64-bit integer.

Variable sized types

Rust also provides types whose size depends on the size of a pointer of the underlying machine. These types have 'size' as the category, and come in signed and unsigned varieties. This makes for two types: isize and usize.

Floating-point types

Rust also has two floating point types: f32 and f64. These correspond to IEEE-754 single and double precision numbers.

5.3.4 Arrays

Like many programming languages, Rust has list types to represent a sequence of things. The most basic is the *array*, a fixed-size list of elements of the same type. By default, arrays are immutable.

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

Arrays have type [T; N]. We'll talk about this T notation in the generics section (section 5.19, page 205). The N is a compile-time constant, for the length of the array.

There's a shorthand for initializing each element of an array to the same value. In this example, each element of a will be initialized to 0:

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

You can get the number of elements in an array a with a.len():

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

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

You can access a particular element of an array with *subscript notation*:

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

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

Subscripts start at zero, like in most programming languages, so the first name is names[0] and the second name is names[1]. The above example prints The second name is: Brian. If you try to use a subscript that is not in the array, you will get an error: array access is bounds-checked at run-time. Such errant access is the source of many bugs in other systems programming languages.

You can find more documentation for arrays in the standard library documentation[20].

5.3.5   Slices

A 'slice' is a reference to (or "view" into) another data structure. They are useful for allowing safe, efficient access to a portion of an array without copying. For example, you might want to reference just one line of a file read into memory. By nature, a slice is not created directly, but from an existing variable. Slices have a length, can be mutable or not, and in many ways behave like arrays:

```rust
let a = [0, 1, 2, 3, 4];
let middle = &a[1..4]; // A slice of a: just the elements 1, 2, and 3
let complete = &a[..]; // A slice containing all of the elements in a
```

Slices have type &[T]. We'll talk about that T when we cover generics (section 5.19, page 205).

You can find more documentation for slices in the standard library documentation[21].

5.3.6   str

Rust's str type is the most primitive string type. As an unsized type (section 5.32, page 251), it's not very useful by itself, but becomes useful when placed behind a reference, like &str (section 5.18, page 203). As such, we'll just leave it at that.

You can find more documentation for str in the standard library documentation[22].

5.3.7   Tuples

A tuple is an ordered list of fixed size. Like this:

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

The parentheses and commas form this two-length tuple. Here's the same code, but with the type annotated:

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

---

[20]http://doc.rust-lang.org/std/primitive.array.html
[21]http://doc.rust-lang.org/std/primitive.slice.html
[22]http://doc.rust-lang.org/std/primitive.str.html

As you can see, the type of a tuple looks just like the tuple, but with each position having a type name rather than the value. Careful readers will also note that tuples are heterogeneous: we have an i32 and a &str in this tuple. In systems programming languages, strings are a bit more complex than in other languages. For now, just read &str as a *string slice*, and we'll learn more soon.

You can assign one tuple into another, if they have the same contained types and arity[23]. Tuples have the same arity when they have the same length.

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

x = y;
```

You can access the fields in a tuple through a *destructuring let*. Here's an example:

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

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

Remember before (section 5.1, page 147) when I said the left-hand side of a let statement was more powerful than just assigning a binding? Here we are. We can put a pattern on the left-hand side of the let, and if it matches up to the right-hand side, we can assign multiple bindings at once. In this case, let "destructures" or "breaks up" the tuple, and assigns the bits to three bindings.

This pattern is very powerful, and we'll see it repeated more later.

You can disambiguate a single-element tuple from a value in parentheses with a comma:

```
(0,); // single-element tuple
(0); // zero in parentheses
```

Tuple Indexing

You can also access fields of a tuple with indexing syntax:

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

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

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

Like array indexing, it starts at zero, but unlike array indexing, it uses a ., rather than []s.

You can find more documentation for tuples in the standard library documentation[24].

---

[23]glossary.html#arity

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

5.3.8   Functions

Functions also have a type! They look like this:

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

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

In this case, x is a 'function pointer' to a function that takes an i32 and returns an i32.

5.4   Comments 注释

Now that we have some functions, it's a good idea to learn about comments.Comments are notes that you leave to other programmers to help explain things about your code. The compiler mostly ignores them.

我们有了几个函数，了解注释是一个好主意。注释是标注在你将代码给其他程序员后，帮助他们理解你的代码的东西。编译器大都忽略他们。

Rust has two kinds of comments that you should care about: *line comments* and *doc comments*.

Rust 有两种你需要关注的注释类型，行注释和块注释。

```
// Line comments are anything after '//' and extend to the end of the line.

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

// If you have a long explanation for something, you can put line comments next
// to each other. Put a space between the // and your comment so that it's
// more readable.
```

The other kind of comment is a doc comment. Doc comments use /// instead of //, and support Markdown notation inside:

另一种注释类是是块注释。块注释使用///替代//，并且支持 Markdown 符号

```
/// Adds one to the number given.
///
/// # Examples
///
/// ```
/// let five = 5;
///
/// assert_eq!(6, add_one(5));
/// ```
```

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

When writing doc comments, providing some examples of usage is very, very helpful. You'll notice we've used a new macro here: assert_eq!. This compares two values, and panic!s if they're not equal to each other. It's very helpful in documentation. There's another macro, assert!, which panic!s if the value passed to it is false.

当些块注释时，提供一些应用示例是非常非常有帮助的。你会注意到我们在这里使用了一个新的宏： assert_eq!**。它对比两个值，并且在他们相互见不相等时产生** panic!**。在文档中它是非常有用的。这里还有另一个宏，** assert!**，如果传递给它的值是** false**，就会** panic!**。**

You can use the rustdoc[25] tool to generate HTML documentation from these doc comments, and also to run the code examples as tests!

你可以使用 rustdoc[26]**工具从这些块注释来生成** html **文档注释，同样可以运行的代码例子作为测试！**

5.5　if if 条件语句

Rust's take on if is not particularly complex, but it's much more like the if you'll find in a dynamically typed language than in a more traditional systems language. So let's talk about it, to make sure you grasp the nuances.

if is a specific form of a more general concept, the 'branch'. The name comes from a branch in a tree: a decision point, where depending on a choice, multiple paths can be taken.

In the case of if, there is one choice that leads down two paths:

```
let x = 5;

if x == 5 {
  println!("x is five!");
}
```

If we changed the value of x to something else, this line would not print. More specifically, if the expression after the if evaluates to true, then the block is executed. If it's false, then it is not.

If you want something to happen in the false case, use an else:

```
let x = 5;

if x == 5 {
  println!("x is five!");
```

---

[25]documentation.md
[26]documentation.md

```rust
} else {
  println!("x is not five :(");
}
```

If there is more than one case, use an else if:

```rust
let x = 5;

if x == 5 {
  println!("x is five!");
} else if x == 6 {
  println!("x is six!");
} else {
  println!("x is not five or six :(");
}
```

This is all pretty standard. However, you can also do this:

```rust
let x = 5;

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

Which we can (and probably should) write like this:

```rust
let x = 5;

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

This works because if is an expression. The value of the expression is the value of the last expression in whichever branch was chosen. An if without an else always results in () as the value.

5.6   for loops for 循环语句

The for loop is used to loop a particular number of times. Rust's for loops work a bit differently than in other systems languages, however. Rust's for loop doesn't look like this "C-style" for loop:

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

Instead, it looks like this:

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

In slightly more abstract terms,

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

The expression is an iterator (section 4.5, page 105). The iterator gives back a series of elements. Each element is one iteration of the loop. That value is then bound to the name var, which is valid for the loop body. Once the body is over, the next value is fetched from the iterator, and we loop another time. When there are no more values, the for loop is over.

In our example, 0..10 is an expression that takes a start and an end position, and gives an iterator over those values. The upper bound is exclusive, though, so our loop will print 0 through 9, not 10.

Rust does not have the "C-style" for loop on purpose. Manually controlling each element of the loop is complicated and error prone, even for experienced C developers.

5.7 while loops while 循环语句

Rust also has a while loop. It looks like this:

```
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 loops are the correct choice when you're not sure how many times you need to loop.

If you need an infinite loop, you may be tempted to write this:

```rust
while true {
```

However, Rust has a dedicated keyword, loop, to handle this case:

```rust
loop {
```

Rust's control-flow analysis treats this construct differently than a while true, since we know that it will always loop. In general, the more information we can give to the compiler, the better it can do with safety and code generation, so you should always prefer loop when you plan to loop infinitely.

Ending iteration early

Let's take a look at that while loop we had earlier:

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

while !done {
  x += x - 3;

  println!("{}", x);

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

We had to keep a dedicated mut boolean variable binding, done, to know when we should exit out of the loop. Rust has two keywords to help us with modifying iteration: break and continue.

In this case, we can write the loop in a better way with break:

```rust
let mut x = 5;

loop {
  x += x - 3;

  println!("{}", x);

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

We now loop forever with loop and use break to break out early.

continue is similar, but instead of ending the loop, goes to the next iteration. This will only print the odd numbers:

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

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

Both `continue` and `break` are valid in both `while` loops and `for` loops (section 5.6, page 164).

5.8 Ownership 权限体系

This guide is one of three presenting Rust's ownership system. This is one of Rust's most unique and compelling features, with which Rust developers should become quite acquainted. Ownership is how Rust achieves its largest goal, memory safety. There are a few distinct concepts, each with its own chapter:

- ownership, which you're reading now
- borrowing (section 5.9, page 170), and their associated feature 'references'
- lifetimes (section 5.10, page 177), an advanced concept of borrowing

These three chapters are related, and in order. You'll need all three to fully understand the ownership system.

5.8.1 Meta

Before we get to the details, two important notes about the ownership system.

Rust has a focus on safety and speed. It accomplishes these goals through many 'zero-cost abstractions', which means that in Rust, abstractions cost as little as possible in order to make them work. The ownership system is a prime example of a zero-cost abstraction. All of the analysis we'll talk about in this guide is *done at compile time*. You do not pay any run-time cost for any of these features.

However, this system does have a certain cost: learning curve. Many new users to Rust experience something we like to call 'fighting with the borrow checker', where the Rust compiler refuses to compile a program that the author thinks is valid. This often happens because the programmer's mental model of how ownership should work doesn't match the actual rules that Rust implements. You probably will experience similar things at first. There is good news, however: more experienced Rust developers report that once they work with the rules of the ownership system for a period of time, they fight the borrow checker less and less.

With that in mind, let's learn about ownership.

5.8.2   Ownership

Variable bindings (section 5.1, page 147) have a property in Rust: they 'have ownership' of what they're bound to. This means that when a binding goes out of scope, the resource that they're bound to are freed. For example:

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

When v comes into scope, a new Vec<T>[27] is created. In this case, the vector also allocates space on the heap (section 4.1, page 65), for the three elements. When v goes out of scope at the end of foo(), Rust will clean up everything related to the vector, even the heap-allocated memory. This happens deterministically, at the end of the scope.

5.8.3   Move semantics

There's some more subtlety here, though: Rust ensures that there is *exactly one* binding to any given resource. For example, if we have a vector, we can assign it to another binding:

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

let v2 = v;
```

But, if we try to use v afterwards, we get an error:

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

let v2 = v;

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

It looks like this:

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

A similar thing happens if we define a function which takes ownership, and try to use something after we've passed it as an argument:

---

[27]http://doc.rust-lang.org/std/vec/struct.Vec.html

```
fn take(v: Vec<i32>) {
  // what happens here isn't important.
}

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

take(v);

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

Same error: 'use of moved value'. When we transfer ownership to something else, we say that we've 'moved' the thing we refer to. You don't need some sort of special annotation here, it's the default thing that Rust does.

The details

The reason that we cannot use a binding after we've moved it is subtle, but important. When we write code like this:

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

let v2 = v;
```

The first line allocates memory for the vector object, v, and for the data it contains. The vector object is stored on the stack (section 4.1, page 65) and contains a pointer to the content ([1, 2, 3]) stored on the heap (section 4.1, page 65). When we move v to v2, it creates a copy of that pointer, for v2. Which means that there would be two pointers to the content of the vector on the heap. It would violate Rust's safety guarantees by introducing a data race. Therefore, Rust forbids using v after we've done the move.

It's also important to note that optimizations may remove the actual copy of the bytes on the stack, depending on circumstances. So it may not be as inefficient as it initially seems.

Copy types

We've established that when ownership is transferred to another binding, you cannot use the original binding. However, there's a trait (section 5.20, page 208) that changes this behavior, and it's called Copy. We haven't discussed traits yet, but for now, you can think of them as an annotation to a particular type that adds extra behavior. For example:

```
let v = 1;

let v2 = v;

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

In this case, v is an i32, which implements the Copy trait. This means that, just like a move, when we assign v to v2, a copy of the data is made. But, unlike a move, we can still use v afterward. This is because an i32 has no pointers to data somewhere else, copying it is a full copy.

We will discuss how to make your own types Copy in the traits (section 5.20, page 208) section.

5.8.4   More than ownership

Of course, if we had to hand ownership back with every function we wrote:

```rust
fn foo(v: Vec<i32>) -> Vec<i32> {
  // do stuff with v

  // hand back ownership
  v
}
```

This would get very tedious. It gets worse the more things we want to take ownership of:

```rust
fn foo(v1: Vec<i32>, v2: Vec<i32>) -> (Vec<i32>, Vec<i32>, i32) {
  // do stuff with v1 and v2

  // hand back ownership, and the result of our function
  (v1, v2, 42)
}

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

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

Ugh! The return type, return line, and calling the function gets way more complicated.

Luckily, Rust offers a feature, borrowing, which helps us solve this problem. It's the topic of the next section!

5.9   References and Borrowing 地址引用和借用

This guide is one of three presenting Rust's ownership system. This is one of Rust's most unique and compelling features, with which Rust developers should become quite acquainted. Ownership is how Rust achieves its largest goal, memory safety. There are a few distinct concepts, each with its own chapter:

- ownership (section 5.8, page 167), the key concept

- borrowing, which you're reading now
- lifetimes (section 5.10, page 177), an advanced concept of borrowing

These three chapters are related, and in order. You'll need all three to fully understand the ownership system.

5.9.1   Meta

Before we get to the details, two important notes about the ownership system.

Rust has a focus on safety and speed. It accomplishes these goals through many 'zero-cost abstractions', which means that in Rust, abstractions cost as little as possible in order to make them work. The ownership system is a prime example of a zero cost abstraction. All of the analysis we'll talk about in this guide is *done at compile time*. You do not pay any run-time cost for any of these features.

However, this system does have a certain cost: learning curve. Many new users to Rust experience something we like to call 'fighting with the borrow checker', where the Rust compiler refuses to compile a program that the author thinks is valid. This often happens because the programmer's mental model of how ownership should work doesn't match the actual rules that Rust implements. You probably will experience similar things at first. There is good news, however: more experienced Rust developers report that once they work with the rules of the ownership system for a period of time, they fight the borrow checker less and less.

With that in mind, let's learn about borrowing.

5.9.2   Borrowing

At the end of the ownership (section 5.8, page 167) section, we had a nasty function that looked like this:

```rust
fn foo(v1: Vec<i32>, v2: Vec<i32>) -> (Vec<i32>, Vec<i32>, i32) {
    // do stuff with v1 and v2

    // hand back ownership, and the result of our function
    (v1, v2, 42)
}

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

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

This is not idiomatic Rust, however, as it doesn't take advantage of borrowing. Here's the first step:

```
fn foo(v1: &Vec<i32>, v2: &Vec<i32>) -> i32 {
  // do stuff with v1 and v2

  // return the answer
  42
}

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

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

// we can use v1 and v2 here!
```

Instead of taking Vec<i32>s as our arguments, we take a reference: &Vec<i32>. And instead of passing v1 and v2 directly, we pass &v1 and &v2. We call the &T type a 'reference', and rather than owning the resource, it borrows ownership. A binding that borrows something does not deallocate the resource when it goes out of scope. This means that after the call to foo(), we can use our original bindings again.

References are immutable, just like bindings. This means that inside of foo(), the vectors can't be changed at all:

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

let v = vec![];

foo(&v);
```

errors with:

```
error: cannot borrow immutable borrowed content `*v` as mutable
v.push(5);
^
```

Pushing a value mutates the vector, and so we aren't allowed to do it.

5.9.3 &mut references

There's a second kind of reference: &mut T. A 'mutable reference' allows you to mutate the resource you're borrowing. For example:

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

This will print 6. We make y a mutable reference to x, then add one to the thing y points at. You'll notice that x had to be marked mut as well, if it wasn't, we couldn't take a mutable borrow to an immutable value.

Otherwise, &mut references are just like references. There *is* a large difference between the two, and how they interact, though. You can tell something is fishy in the above example, because we need that extra scope, with the { and }. If we remove them, we get an error:

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

}
^
```

As it turns out, there are rules.

5.9.4 The Rules

Here's the rules about borrowing in Rust:

First, any borrow must last for a smaller scope than the owner. Second, you may have one or the other of these two kinds of borrows, but not both at the same time:

- 0 to N references (&T) to a resource.
- exactly one mutable reference (&mut T)

You may notice that this is very similar, though not exactly the same as, to the definition of a data race:

> There is a 'data race' when two or more pointers access the same memory location at the same time, where at least one of them is writing, and the operations are not synchronized.

With references, you may have as many as you'd like, since none of them are writing. If you are writing, you need two or more pointers to the same memory, and you can only have one &mut at a time. This is how Rust prevents data races at compile time: we'll get errors if we break the rules.

With this in mind, let's consider our example again.

Thinking in scopes

Here's the code:

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

*y += 1;

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

This code gives us this error:

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

This is because we've violated the rules: we have a &mut T pointing to x, and so we aren't allowed to create any &Ts. One or the other. The note hints at how to think about this problem:

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

}
^
```

In other words, the mutable borow is held through the rest of our example. What we want is for the mutable borrow to end *before* we try to call println! and make an immutable borrow. In Rust, borrowing is tied to the scope that the borrow is valid for. And our scopes look like this:

```rust
let mut x = 5;

let y = &mut x;   // -+ &mut borrow of x starts here
          // |
*y += 1;       // |
          // |
println!("{}", x); // -+ - try to borrow x here
          // -+ &mut borrow of x ends here
```

The scopes conflict: we can't make an &x while y is in scope.

So when we add the curly braces:

```
let mut x = 5;

{
  let y = &mut x; // -+ &mut borrow starts here
  *y += 1;     // |
}           // -+ ... and ends here

println!("{}", x); // <- try to borrow x here
```

There's no problem. Our mutable borrow goes out of scope before we create an immutable one. But scope is the key to seeing how long a borrow lasts for.

Issues borrowing prevents

Why have these restrictive rules? Well, as we noted, these rules prevent data races. What kinds of issues do data races cause? Here's a few.

Iterator invalidation   One example is 'iterator invalidation', which happens when you try to mutate a collection that you're iterating over. Rust's borrow checker prevents this from happening:

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

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

This prints out one through three. As we iterate through the vectors, we're only given references to the elements. And v is itself borrowed as immutable, which means we can't change it while we're iterating:

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

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

Here's the error:

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

We can't modify v because it's borrowed by the loop.

use after free   References must live as long as the resource they refer to. Rust will check the scopes of your references to ensure that this is true.

If Rust didn't check that this property, we could accidentally use a reference which was invalid. For example:

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

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

We get this error:

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

note: ...but borrowed value is only valid for the block suffix following
statement 0 at 4:18
```

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

In other words, y is only valid for the scope where x exists. As soon as x goes away, it becomes invalid to refer to it. As such, the error says that the borrow 'doesn't live long enough' because it's not valid for the right amount of time.

The same problem occurs when the reference is declared *before* the variable it refers to:

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

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

We get this error:

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

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

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

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

5.10   Lifetimes 生命周期

This guide is one of three presenting Rust's ownership system. This is one of Rust's most unique and compelling features, with which Rust developers should become quite acquainted. Ownership is how Rust achieves its largest goal, memory safety. There are a few distinct concepts, each with its own chapter:

- ownership (section 5.8, page 167), the key concept
- borrowing (section 5.9, page 170), and their associated feature 'references'
- lifetimes, which you're reading now

These three chapters are related, and in order. You'll need all three to fully understand the ownership system.

5.10.1   Meta

Before we get to the details, two important notes about the ownership system.

Rust has a focus on safety and speed. It accomplishes these goals through many 'zero-cost abstractions', which means that in Rust, abstractions cost as little as possible in order to make them work. The ownership system is a prime example of a zero-cost abstraction. All of the analysis we'll talk about in this guide is *done at compile time*. You do not pay any run-time cost for any of these features.

However, this system does have a certain cost: learning curve. Many new users to Rust experience something we like to call 'fighting with the borrow checker', where the Rust compiler refuses to compile a program that the author thinks is valid. This often happens because the programmer's mental model of how ownership should work doesn't match the actual rules that Rust implements. You probably will experience similar things at first. There is good news, however: more experienced Rust developers report that once they work with the rules of the ownership system for a period of time, they fight the borrow checker less and less.

With that in mind, let's learn about lifetimes.

5.10.2   Lifetimes

Lending out a reference to a resource that someone else owns can be complicated. For example, imagine this set of operations:

- I acquire a handle to some kind of resource.
- I lend you a reference to the resource.
- I decide I'm done with the resource, and deallocate it, while you still have your reference.
- You decide to use the resource.

Uh oh! Your reference is pointing to an invalid resource. This is called a dangling pointer or 'use after free', when the resource is memory.

To fix this, we have to make sure that step four never happens after step three. The ownership system in Rust does this through a concept called lifetimes, which describe the scope that a reference is valid for.

When we have a function that takes a reference by argument, we can be implicit or explicit about the lifetime of the reference:

```rust
// implicit
fn foo(x: &i32) {
}

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

The 'a reads 'the lifetime a'. Technically, every reference has some lifetime associated with it, but the compiler lets you elide them in common cases. Before we get to that, though, let's break the explicit example down:

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

This part declares our lifetimes. This says that bar has one lifetime, 'a. If we had two reference parameters, it would look like this:

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

Then in our parameter list, we use the lifetimes we've named:

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

If we wanted an &mut reference, we'd do this:

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

If you compare &mut i32 to &'a mut i32, they're the same, it's just that the lifetime 'a has snuck in between the & and the mut i32. We read &mut i32 as 'a mutable reference to an i32' and &'a mut i32 as 'a mutable reference to an i32 with the lifetime 'a'.

You'll also need explicit lifetimes when working with struct (section 5.12, page 186)s:

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

fn main() {
  let y = &5; // this is the same as `let _y = 5; let y = &_y;`
  let f = Foo { x: y };

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

As you can see, structs can also have lifetimes. In a similar way to functions,

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

declares a lifetime, and

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

uses it. So why do we need a lifetime here? We need to ensure that any reference to a `Foo` cannot outlive the reference to an `i32` it contains.

Thinking in scopes

A way to think about lifetimes is to visualize the scope that a reference is valid for. For example:

```rust
fn main() {
    let y = &5;    // -+ y goes into scope
                   // |
    // stuff       // |
                   // |
}                  // -+ y goes out of scope
```

Adding in our `Foo`:

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

fn main() {
    let y = &5;         // -+ y goes into scope
    let f = Foo { x: y }; // -+ f goes into scope
    // stuff            // |
                        // |
}                       // -+ f and y go out of scope
```

Our `f` lives within the scope of `y`, so everything works. What if it didn't? This code won't work:

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

fn main() {
    let x;             // -+ x goes into scope
                       // |
    {                  // |
```

```
    let y = &5;        // ---+ y goes into scope
    let f = Foo { x: y }; // ---+ f goes into scope
    x = &f.x;          // || error here
  }                    // ---+ f and y go out of scope
                       // |
  println!("{}", x);   // |
}                      // -+ x goes out of scope
```

Whew! As you can see here, the scopes of f and y are smaller than the scope of x. But when we do x = &f.x, we make x a reference to something that's about to go out of scope.

Named lifetimes are a way of giving these scopes a name. Giving something a name is the first step towards being able to talk about it.

'static

The lifetime named 'static' is a special lifetime. It signals that something has the lifetime of the entire program. Most Rust programmers first come across 'static when dealing with strings:

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

String literals have the type &'static str because the reference is always alive: they are baked into the data segment of the final binary. Another example are globals:

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

This adds an i32 to the data segment of the binary, and x is a reference to it.

Lifetime Elision

Rust supports powerful local type inference in function bodies, but it's forbidden in item signatures to allow reasoning about the types just based in the item signature alone. However, for ergonomic reasons a very restricted secondary inference algorithm called "lifetime elision" applies in function signatures. It infers only based on the signature components themselves and not based on the body of the function, only infers lifetime parameters, and does this with only three easily memorizable and unambiguous rules. This makes lifetime elision a shorthand for writing an item signature, while not hiding away the actual types involved as full local inference would if applied to it.

When talking about lifetime elision, we use the term *input lifetime* and *output lifetime*. An *input lifetime* is a lifetime associated with a parameter of a function, and an *output lifetime* is a lifetime associated with the return value of a function. For example, this function has an input lifetime:

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

This one has an output lifetime:

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

This one has a lifetime in both positions:

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

Here are the three rules:

- Each elided lifetime in a function's arguments becomes a distinct lifetime parameter.

- If there is exactly one input lifetime, elided or not, that lifetime is assigned to all elided lifetimes in the return values of that function.

- If there are multiple input lifetimes, but one of them is &self or &mut self, the lifetime of self is assigned to all elided output lifetimes.

Otherwise, it is an error to elide an output lifetime.

Examples   Here are some examples of functions with elided lifetimes. We've paired each example of an elided lifetime with its expanded form.

```rust
fn print(s: &str); // elided
fn print<'a>(s: &'a str); // expanded

fn debug(lvl: u32, s: &str); // elided
fn debug<'a>(lvl: u32, s: &'a str); // expanded

// In the preceding example, `lvl` doesn't need a lifetime because it's not a
// reference (`&`). Only things relating to references (such as a `struct`
// which contains a reference) need lifetimes.

fn substr(s: &str, until: u32) -> &str; // elided
fn substr<'a>(s: &'a str, until: u32) -> &'a str; // expanded

fn get_str() -> &str; // ILLEGAL, no inputs

fn frob(s: &str, t: &str) -> &str; // ILLEGAL, two inputs
fn frob<'a, 'b>(s: &'a str, t: &'b str) -> &str; // Expanded: Output lifetime is unclea
 r
```

```rust
fn get_mut(&mut self) -> &mut T; // elided
fn get_mut<'a>(&'a mut self) -> &'a mut T; // expanded

fn args<T:ToCStr>(&mut self, args: &[T]) -> &mut Command // elided
fn args<'a, 'b, T:ToCStr>(&'a mut self, args: &'b [T]) -> &'a mut Command // expanded

fn new(buf: &mut [u8]) -> BufWriter; // elided
fn new<'a>(buf: &'a mut [u8]) -> BufWriter<'a> // expanded
```

5.11   Mutability 可变性

Mutability, the ability to change something, works a bit differently in Rust than in other languages. The first aspect of mutability is its non-default status:

```rust
let x = 5;
x = 6; // error!
```

We can introduce mutability with the `mut` keyword:

```rust
let mut x = 5;
```

```rust
x = 6; // no problem!
```

This is a mutable variable binding (section 5.1, page 147). When a binding is mutable, it means you're allowed to change what the binding points to. So in the above example, it's not so much that the value at x is changing, but that the binding changed from one `i32` to another.

If you want to change what the binding points to, you'll need a mutable reference (section 5.9, page 170):

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

y is an immutable binding to a mutable reference, which means that you can't bind y to something else (`y = &mut z`), but you can mutate the thing that's bound to y (`*y = 5`). A subtle distinction.

Of course, if you need both:

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

Now y can be bound to another value, and the value it's referencing can be changed.

It's important to note that `mut` is part of a pattern (section 5.15, page 192), so you can do things like this:

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

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

5.11.1 Interior vs. Exterior Mutability

However, when we say something is 'immutable' in Rust, that doesn't mean that it's not able to be changed: We mean something has 'exterior mutability'. Consider, for example, Arc<T>[28]:

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

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

When we call clone(), the Arc<T> needs to update the reference count. Yet we've not used any muts here, x is an immutable binding, and we didn't take &mut 5 or anything. So what gives?

To understand this, we have to go back to the core of Rust's guiding philosophy, memory safety, and the mechanism by which Rust guarantees it, the ownership (section 5.8, page 167) system, and more specifically, borrowing[29]:

> You may have one or the other of these two kinds of borrows, but not both at the same time:
>
> - one or more references (&T) to a resource.
> - exactly one mutable reference (&mut T)

So, that's the real definition of 'immutability': is this safe to have two pointers to? In Arc<T>'s case, yes: the mutation is entirely contained inside the structure itself. It's not user facing. For this reason, it hands out &T with clone(). If it handed out &mut Ts, though, that would be a problem.

Other types, like the ones in the std::cell[30] module, have the opposite: interior mutability. For example:

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

let x = RefCell::new(42);

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

RefCell hands out &mut references to what's inside of it with the borrow_mut() method. Isn't that dangerous? What if we do:

---

[28]http://doc.rust-lang.org/std/sync/struct.Arc.html

[29]borrowing.html#The-Rules

[30]http://doc.rust-lang.org/std/cell/index.html

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

let x = RefCell::new(42);

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

This will in fact panic, at runtime. This is what RefCell does: it enforces Rust's borrowing rules at runtime, and panic!s if they're violated. This allows us to get around another aspect of Rust's mutability rules. Let's talk about it first.

Field-level mutability

Mutability is a property of either a borrow (&mut) or a binding (let mut). This means that, for example, you cannot have a struct (section 5.12, page 186) with some fields mutable and some immutable:

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

The mutability of a struct is in its binding:

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

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

a.x = 10;

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

b.x = 10; // error: cannot assign to immutable field `b.x`
```

However, by using Cell<T>, you can emulate field-level mutability:

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

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

```
}
```

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

```
point.y.set(7);
```

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

This will print y: Cell { value: 7 }. We've successfully updated y.

5.12   Structs 结构体

Structs are a way of creating more complex data types. For example, if we were doing calculations involving coordinates in 2D space, we would need both an x and a y value:

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

A struct lets us combine these two into a single, unified datatype:

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

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

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

There's a lot going on here, so let's break it down. We declare a struct with the struct keyword, and then with a name. By convention, structs begin with a capital letter and are camel cased: PointInSpace, not Point_In_Space.

We can create an instance of our struct via let, as usual, but we use a key: value style syntax to set each field. The order doesn't need to be the same as in the original declaration.

Finally, because fields have names, we can access the field through dot notation: origin.x.

The values in structs are immutable by default, like other bindings in Rust. Use mut to make them mutable:

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

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

    point.x = 5;

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

This will print The point is at (5, 0).

Rust does not support field mutability at the language level, so you cannot write something like this:

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

Mutability is a property of the binding, not of the structure itself. If you're used to field-level mutability, this may seem strange at first, but it significantly simplifies things. It even lets you make things mutable for a short time only:

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

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

    point.x = 5;

    let point = point; // this new binding can't change now

    point.y = 6; // this causes an error
}
```

5.12.1 Update syntax

A struct can include .. to indicate that you want to use a copy of some other struct for some of the values. For example:

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

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

This gives point a new y, but keeps the old x and z values. It doesn't have to be the same struct either, you can use this syntax when making new ones, and it will copy the values you don't specify:

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

5.12.2   Tuple structs

Rust has another data type that's like a hybrid between a tuple[31] and a struct, called a 'tuple struct'. Tuple structs have a name, but their fields don't:

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

These two will not be equal, even if they have the same values:

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

It is almost always better to use a struct than a tuple struct. We would write Color and Point like this instead:

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

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

---

[31]primitive-types.html#tuples

Now, we have actual names, rather than positions. Good names are important, and with a struct, we have actual names.

There *is* one case when a tuple struct is very useful, though, and that's a tuple struct with only one element. We call this the 'newtype' pattern, because it allows you to create a new type, distinct from that of its contained value and expressing its own semantic meaning:

```rust
struct Inches(i32);

let length = Inches(10);

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

As you can see here, you can extract the inner integer type through a destructuring let, just as with regular tuples. In this case, the let Inches(integer_length) assigns 10 to integer_length.

5.12.3 Unit-like structs

You can define a struct with no members at all:

```rust
struct Electron;
```

Such a struct is called 'unit-like' because it resembles the empty tuple, (), sometimes called 'unit'. Like a tuple struct, it defines a new type.

This is rarely useful on its own (although sometimes it can serve as a marker type), but in combination with other features, it can become useful. For instance, a library may ask you to create a structure that implements a certain trait (section 5.20, page 208) to handle events. If you don't have any data you need to store in the structure, you can just create a unit-like struct.

5.13 Enums 枚举

An enum in Rust is a type that represents data that could be one of several possible variants:

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

Each variant can optionally have data associated with it. The syntax for defining variants resembles the syntaxes used to define structs: you can have variants with no data (like unit-like structs),

variants with named data, and variants with unnamed data (like tuple structs). Unlike separate struct definitions, however, an enum is a single type. A value of the enum can match any of the variants. For this reason, an enum is sometimes called a 'sum type': the set of possible values of the enum is the sum of the sets of possible values for each variant.

We use the :: syntax to use the name of each variant: they're scoped by the name of the enum itself. This allows both of these to work:

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

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

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

Both variants are named Move, but since they're scoped to the name of the enum, they can both be used without conflict.

A value of an enum type contains information about which variant it is, in addition to any data associated with that variant. This is sometimes referred to as a 'tagged union', since the data includes a 'tag' indicating what type it is. The compiler uses this information to enforce that you're accessing the data in the enum safely. For instance, you can't simply try to destructure a value as if it were one of the possible variants:

```rust
fn process_color_change(msg: Message) {
    let Message::ChangeColor(r, g, b) = msg; // compile-time error
}
```

Not supporting these operations may seem rather limiting, but it's a limitation which we can overcome. There are two ways: by implementing equality ourselves, or by pattern matching variants with match (section 5.14, page 190) expressions, which you'll learn in the next section. We don't know enough about Rust to implement equality yet, but we'll find out in the traits (section 5.20, page 208) section.

5.14   Match 比较匹配

Often, a simple if (section 5.5, page 163)/else isn't enough, because you have more than two possible options. Also, conditions can get quite complex. Rust has a keyword, match, that allows you to replace complicated if/else groupings with something more powerful. Check it out:

```rust
let x = 5;

match x {
```

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

match takes an expression and then branches based on its value. Each 'arm' of the branch is of the form val => expression. When the value matches, that arm's expression will be evaluated. It's called match because of the term 'pattern matching', which match is an implementation of. There's an entire section on patterns (section 5.15, page 192) that covers all the patterns that are possible here.

So what's the big advantage? Well, there are a few. First of all, match enforces 'exhaustiveness checking'. Do you see that last arm, the one with the underscore (_)? If we remove that arm, Rust will give us an error:

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

In other words, Rust is trying to tell us we forgot a value. Because x is an integer, Rust knows that it can have a number of different values – for example, 6. Without the _, however, there is no arm that could match, and so Rust refuses to compile the code. _ acts like a 'catch-all arm'. If none of the other arms match, the arm with _ will, and since we have this catch-all arm, we now have an arm for every possible value of x, and so our program will compile successfully.

match is also an expression, which means we can use it on the right-hand side of a let binding or directly where an expression is used:

```
let x = 5;

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

Sometimes it's a nice way of converting something from one type to another.

5.14.1 Matching on enums

Another important use of the match keyword is to process the possible variants of an enum:

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

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

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

Again, the Rust compiler checks exhaustiveness, so it demands that you have a match arm for every variant of the enum. If you leave one off, it will give you a compile-time error unless you use _.

Unlike the previous uses of match, you can't use the normal if statement to do this. You can use the if let (section 5.22, page 216) statement, which can be seen as an abbreviated form of match.

5.15   Patterns 模式

Patterns are quite common in Rust. We use them in variable bindings (section 5.1, page 147), match statements (section 5.14, page 190), and other places, too. Let's go on a whirlwind tour of all of the things patterns can do!

A quick refresher: you can match against literals directly, and _ acts as an 'any' case:

```rust
let x = 1;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

This prints one.

5.15.1 Multiple patterns

You can match multiple patterns with |:

```rust
let x = 1;

match x {
    1 | 2 => println!("one or two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

This prints one or two.

5.15.2 Ranges

You can match a range of values with ...:

```rust
let x = 1;

match x {
    1 ... 5 => println!("one through five"),
    _ => println!("anything"),
}
```

This prints one through five.

Ranges are mostly used with integers and chars:

```rust
let x = '';

match x {
    'a' ... 'j' => println!("early letter"),
    'k' ... 'z' => println!("late letter"),
    _ => println!("something else"),
}
```

This prints something else.

5.15.3 Bindings

You can bind values to names with @:

```rust
let x = 1;

match x {
    e @ 1 ... 5 => println!("got a range element {}", e),
    _ => println!("anything"),
}
```

This prints `got a range element 1`. This is useful when you want to do a complicated match of part of a data structure:

```rust
#[derive(Debug)]
struct Person {
    name: Option<String>,
}

let name = "Steve".to_string();
let mut x: Option<Person> = Some(Person { name: Some(name) });
match x {
    Some(Person { name: ref a @ Some(_), .. }) => println!("{:?}", a),
    _ => {}
}
```

This prints `Some("Steve")`: We've bound the inner `name` to `a`.

If you use @ with |, you need to make sure the name is bound in each part of the pattern:

```rust
let x = 5;

match x {
    e @ 1 ... 5 | e @ 8 ... 10 => println!("got a range element {}", e),
    _ => println!("anything"),
}
```

5.15.4 Ignoring variants

If you're matching on an enum which has variants, you can use .. to ignore the value and type in the variant:

```rust
enum OptionalInt {
    Value(i32),
    Missing,
}

let x = OptionalInt::Value(5);
```

```
match x {
    OptionalInt::Value(..) => println!("Got an int!"),
    OptionalInt::Missing => println!("No such luck."),
}
```

This prints Got an int!.

5.15.5 Guards

You can introduce 'match guards' with if:

```
enum OptionalInt {
    Value(i32),
    Missing,
}

let x = OptionalInt::Value(5);

match x {
    OptionalInt::Value(i) if i > 5 => println!("Got an int bigger than five!"),
    OptionalInt::Value(..) => println!("Got an int!"),
    OptionalInt::Missing => println!("No such luck."),
}
```

This prints Got an int!.

5.15.6 ref and ref mut

If you want to get a reference (section 5.9, page 170), use the ref keyword:

```
let x = 5;

match x {
    ref r => println!("Got a reference to {}", r),
}
```

This prints Got a reference to 5.

Here, the r inside the match has the type &i32. In other words, the ref keyword *creates* a reference, for use in the pattern. If you need a mutable reference, ref mut will work in the same way:

```
let mut x = 5;

match x {
    ref mut mr => println!("Got a mutable reference to {}", mr),
}
```

5.15.7 Destructuring

If you have a compound data type, like a struct (section 5.12, page 186), you can destructure it inside of a pattern:

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

let origin = Point { x: 0, y: 0 };

match origin {
    Point { x: x, y: y } => println!("({},{})", x, y),
}
```

If we only care about some of the values, we don't have to give them all names:

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

let origin = Point { x: 0, y: 0 };

match origin {
    Point { x: x, .. } => println!("x is {}", x),
}
```

This prints x is 0.

You can do this kind of match on any member, not just the first:

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

```rust
let origin = Point { x: 0, y: 0 };

match origin {
    Point { y: y, .. } => println!("y is {}", y),
}
```

This prints y is 0.

This 'destructuring' behavior works on any compound data type, like tuples[32] or enums (section 5.13, page 189).

5.15.8   Mix and Match

Whew! That's a lot of different ways to match things, and they can all be mixed and matched, depending on what you're doing:

```rust
match x {
    Foo { x: Some(ref name), y: None } => ...
}
```

Patterns are very powerful. Make good use of them.

5.16   Method Syntax 方法

Functions are great, but if you want to call a bunch of them on some data, it can be awkward. Consider this code:

```rust
baz(bar(foo)));
```

We would read this left-to right, and so we see 'baz bar foo'. But this isn't the order that the functions would get called in, that's inside-out: 'foo bar baz'. Wouldn't it be nice if we could do this instead?

```rust
foo.bar().baz();
```

Luckily, as you may have guessed with the leading question, you can! Rust provides the ability to use this 'method call syntax' via the impl keyword.

---

[32]primitive-types.html#tuples

5.16.1   Method calls

Here's how it works:

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

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

fn main() {
  let c = Circle { x: 0.0, y: 0.0, radius: 2.0 };
  println!("{}", c.area());
}
```

This will print `12.566371`.

We've made a struct that represents a circle. We then write an `impl` block, and inside it, define a method, `area`.

Methods take a special first parameter, of which there are three variants: `self`, `&self`, and `&mut self`. You can think of this first parameter as being the `foo` in `foo.bar()`. The three variants correspond to the three kinds of things `foo` could be: `self` if it's just a value on the stack, `&self` if it's a reference, and `&mut self` if it's a mutable reference. Because we took the `&self` parameter to `area`, we can use it just like any other parameter. Because we know it's a `Circle`, we can access the `radius` just like we would with any other struct.

We should default to using `&self`, as you should prefer borrowing over taking ownership, as well as taking immutable references over mutable ones. Here's an example of all three variants:

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

impl Circle {
  fn reference(&self) {
    println!("taking self by reference!");
  }
```

```rust
    fn mutable_reference(&mut self) {
        println!("taking self by mutable reference!");
    }

    fn takes_ownership(self) {
        println!("taking ownership of self!");
    }
}
```

5.16.2   Chaining method calls

So, now we know how to call a method, such as foo.bar(). But what about our original example,
foo.bar().baz()? This is called 'method chaining', and we can do it by returning self.

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

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

    fn grow(&self, increment: f64) -> Circle {
        Circle { x: self.x, y: self.y, radius: self.radius + increment }
    }
}

fn main() {
    let c = Circle { x: 0.0, y: 0.0, radius: 2.0 };
    println!("{}", c.area());

    let d = c.grow(2.0).area();
    println!("{}", d);
}
```

Check the return type:

```rust
# struct Circle;
# impl Circle {
fn grow(&self) -> Circle {
# Circle } }
```

We just say we're returning a `Circle`. With this method, we can grow a new circle to any arbitrary size.

5.16.3   Associated functions

You can also define associated functions that do not take a `self` parameter. Here's a pattern that's very common in Rust code:

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

impl Circle {
  fn new(x: f64, y: f64, radius: f64) -> Circle {
    Circle {
      x: x,
      y: y,
      radius: radius,
    }
  }
}

fn main() {
  let c = Circle::new(0.0, 0.0, 2.0);
}
```

This 'associated function' builds a new `Circle` for us. Note that associated functions are called with the `Struct::function()` syntax, rather than the `ref.method()` syntax. Some other langauges call associated functions 'static methods'.

5.16.4   Builder Pattern

Let's say that we want our users to be able to create Circles, but we will allow them to only set the properties they care about. Otherwise, the x and y attributes will be `0.0`, and the `radius` will be `1.0`. Rust doesn't have method overloading, named arguments, or variable arguments. We employ the builder pattern instead. It looks like this:

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

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

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

impl CircleBuilder {
  fn new() -> CircleBuilder {
    CircleBuilder { x: 0.0, y: 0.0, radius: 1.0, }
  }

  fn x(&mut self, coordinate: f64) -> &mut CircleBuilder {
    self.x = coordinate;
    self
  }

  fn y(&mut self, coordinate: f64) -> &mut CircleBuilder {
    self.y = coordinate;
    self
  }

  fn radius(&mut self, radius: f64) -> &mut CircleBuilder {
    self.radius = radius;
    self
  }

  fn finalize(&self) -> Circle {
    Circle { x: self.x, y: self.y, radius: self.radius }
  }
}

fn main() {
  let c = CircleBuilder::new()
        .x(1.0)
        .y(2.0)
        .radius(2.0)
        .finalize();

  println!("area: {}", c.area());
```

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

What we've done here is make another struct, CircleBuilder. We've defined our builder methods on it. We've also defined our area() method on Circle. We also made one more method on CircleBuilder: finalize(). This method creates our final Circle from the builder. Now, we've used the type system to enforce our concerns: we can use the methods on CircleBuilder to constrain making Circles in any way we choose.

5.17   Vectors 向量

A 'vector' is a dynamic or 'growable' array, implemented as the standard library type Vec<T>[33]. The T means that we can have vectors of any type (see the chapter on generics (section 5.19, page 205) for more). Vectors always allocate their data on the heap. You can create them with the vec! macro:

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

(Notice that unlike the println! macro we've used in the past, we use square brackets [] with vec! macro. Rust allows you to use either in either situation, this is just convention.)

There's an alternate form of vec! for repeating an initial value:

```
let v = vec![0; 10]; // ten zeroes
```

Accessing elements

To get the value at a particular index in the vector, we use []s:

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

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

The indices count from 0, so the third element is v[2].

Iterating

Once you have a vector, you can iterate through its elements with for. There are three versions:

---

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

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

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

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

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

Vectors have many more useful methods, which you can read about in their API documentation[34].

5.18   Strings **字符串**

Strings are an important concept for any programmer to master. Rust's string handling system is a bit different from other languages, due to its systems focus. Any time you have a data structure of variable size, things can get tricky, and strings are a re-sizable data structure. That being said, Rust's strings also work differently than in some other systems languages, such as C.

Let's dig into the details. A 'string' is a sequence of Unicode scalar values encoded as a stream of UTF-8 bytes. All strings are guaranteed to be a valid encoding of UTF-8 sequences. Additionally, unlike some systems languages, strings are not null-terminated and can contain null bytes.

Rust has two main types of strings: &str and String. Let's talk about &str first. These are called 'string slices'. String literals are of the type &'static str:

```rust
let string = "Hello there."; // string: &'static str
```

This string is statically allocated, meaning that it's saved inside our compiled program, and exists for the entire duration it runs. The string binding is a reference to this statically allocated string. String slices have a fixed size, and cannot be mutated.

A String, on the other hand, is a heap-allocated string. This string is growable, and is also guaranteed to be UTF-8. Strings are commonly created by converting from a string slice using the to_string method.

```rust
let mut s = "Hello".to_string(); // mut s: String
println!("{}", s);
```

---

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

```
s.push_str(", world.");
println!("{}", s);
```

Strings will coerce into &str with an &:

```
fn takes_slice(slice: &str) {
    println!("Got: {}", slice);
}

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

Viewing a String as a &str is cheap, but converting the &str to a String involves allocating memory. No reason to do that unless you have to!

Indexing

Because strings are valid UTF-8, strings do not support indexing:

```
let s = "hello";

println!("The first letter of s is {}", s[0]); // ERROR!!!
```

Usually, access to a vector with [] is very fast. But, because each character in a UTF-8 encoded string can be multiple bytes, you have to walk over the string to find the n   letter of a string. This is a significantly more expensive operation, and we don't want to be misleading. Furthermore, 'letter' isn't something defined in Unicode, exactly. We can choose to look at a string as individual bytes, or as codepoints:

```
let hachiko = "忠犬ハチ公";

for b in hachiko.as_bytes() {
    print!("{}, ", b);
}

println!("");

for c in hachiko.chars() {
    print!("{}, ", c);
}

println!("");
```

This prints:

229, 191, 160, 231, 138, 172, 227, 131, 143, 227, 131, 129, 229, 133, 172,
忠,**犬**,**ハ**,**チ**,**公**,

As you can see, there are more bytes than chars.

You can get something similar to an index like this:

```rust
let dog = hachiko.chars().nth(1); // kinda like hachiko[1]
```

This emphasizes that we have to go through the whole list of chars.

Concatenation

If you have a String, you can concatenate a &str to the end of it:

```rust
let hello = "Hello ".to_string();
let world = "world!";

let hello_world = hello + world;
```

But if you have two Strings, you need an &:

```rust
let hello = "Hello ".to_string();
let world = "world!".to_string();

let hello_world = hello + &world;
```

This is because &String can automatically coerece to a &str. This is a feature called 'Deref coercions (section 5.34, page 254)'.

5.19   Generics 泛型

Sometimes, when writing a function or data type, we may want it to work for multiple types of arguments. Luckily, Rust has a feature that gives us a better way: generics. Generics are called 'parametric polymorphism' in type theory, which means that they are types or functions that have multiple forms ('poly' is multiple, 'morph' is form) over a given parameter ('parametric').

Anyway, enough with type theory, let's check out some generic code. Rust's standard library provides a type, Option<T>, that's generic:

```
enum Option<T> {
  Some(T),
  None,
}
```

The <T> part, which you've seen a few times before, indicates that this is a generic data type. Inside the declaration of our enum, wherever we see a T, we substitute that type for the same type used in the generic. Here's an example of using Option<T>, with some extra type annotations:

```
let x: Option<i32> = Some(5);
```

In the type declaration, we say Option<i32>. Note how similar this looks to Option<T>. So, in this particular Option, T has the value of i32. On the right-hand side of the binding, we do make a Some(T), where T is 5. Since that's an i32, the two sides match, and Rust is happy. If they didn't match, we'd get an error:

```
let x: Option<f64> = Some(5);
// error: mismatched types: expected `core::option::Option<f64>`,
// found `core::option::Option<_>` (expected f64 but found integral variable)
```

That doesn't mean we can't make Option<T>s that hold an f64! They just have to match up:

```
let x: Option<i32> = Some(5);
let y: Option<f64> = Some(5.0f64);
```

This is just fine. One definition, multiple uses.

Generics don't have to only be generic over one type. Consider another type from Rust's standard library that's similar, Result<T, E>:

```
enum Result<T, E> {
  Ok(T),
  Err(E),
}
```

This type is generic over *two* types: T and E. By the way, the capital letters can be any letter you'd like. We could define Result<T, E> as:

```
enum Result<A, Z> {
  Ok(A),
  Err(Z),
}
```

if we wanted to. Convention says that the first generic parameter should be T, for 'type', and that we use E for 'error'. Rust doesn't care, however.

The Result<T, E> type is intended to be used to return the result of a computation, and to have the ability to return an error if it didn't work out.

Generic functions

We can write functions that take generic types with a similar syntax:

```rust
fn takes_anything<T>(x: T) {
    // do something with x
}
```

The syntax has two parts: the <T> says "this function is generic over one type, T", and the x: T says "x has the type T."

Multiple arguments can have the same generic type:

```rust
fn takes_two_of_the_same_things<T>(x: T, y: T) {
    // ...
}
```

We could write a version that takes multiple types:

```rust
fn takes_two_things<T, U>(x: T, y: U) {
    // ...
}
```

Generic functions are most useful with 'trait bounds', which we'll cover in the section on traits (section 5.20, page 208).

Generic structs

You can store a generic type in a struct as well:

```rust
struct Point<T> {
    x: T,
    y: T,
}

let int_origin = Point { x: 0, y: 0 };
let float_origin = Point { x: 0.0, y: 0.0 };
```

Similarly to functions, the <T> is where we declare the generic parameters, and we then use x: T in the type declaration, too.

5.20   Traits 特征

Do you remember the impl keyword, used to call a function with method syntax (section 5.16, page 197)?

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

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

Traits are similar, except that we define a trait with just the method signature, then implement the trait for that struct. Like this:

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

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

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

As you can see, the trait block looks very similar to the impl block, but we don't define a body, just a type signature. When we impl a trait, we use impl Trait for Item, rather than just impl Item.

We can use traits to constrain our generics. Consider this function, which does not compile, and gives us a similar error:

```rust
fn print_area<T>(shape: T) {
  println!("This shape has an area of {}", shape.area());
}
```

Rust complains:

error: type `T` does not implement any method in scope named `area`

Because T can be any type, we can't be sure that it implements the area method. But we can add a 'trait constraint' to our generic T, ensuring that it does:

```rust
fn print_area<T: HasArea>(shape: T) {
    println!("This shape has an area of {}", shape.area());
}
```

The syntax <T: HasArea> means any type that implements the HasArea trait. Because traits define function type signatures, we can be sure that any type which implements HasArea will have an .area() method.

Here's an extended example of how this works:

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

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

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

struct Square {
    x: f64,
    y: f64,
    side: f64,
}

impl HasArea for Square {
    fn area(&self) -> f64 {
        self.side * self.side
    }
}

fn print_area<T: HasArea>(shape: T) {
    println!("This shape has an area of {}", shape.area());
```

```rust
}

fn main() {
  let c = Circle {
    x: 0.0f64,
    y: 0.0f64,
    radius: 1.0f64,
  };

  let s = Square {
    x: 0.0f64,
    y: 0.0f64,
    side: 1.0f64,
  };

  print_area(c);
  print_area(s);
}
```

This program outputs:

```
This shape has an area of 3.141593
This shape has an area of 1
```

As you can see, print_area is now generic, but also ensures that we have passed in the correct types. If we pass in an incorrect type:

```rust
print_area(5);
```

We get a compile-time error:

```
error: failed to find an implementation of trait main::HasArea for int
```

So far, we've only added trait implementations to structs, but you can implement a trait for any type. So technically, we *could* implement HasArea for i32:

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

impl HasArea for i32 {
  fn area(&self) -> f64 {
    println!("this is silly");
```

```
    *self as f64
  }
}

5.area();
```

It is considered poor style to implement methods on such primitive types, even though it is possible.

This may seem like the Wild West, but there are two other restrictions around implementing traits that prevent this from getting out of hand. The first is that if the trait isn't defined in your scope, it doesn't apply. Here's an example: the standard library provides a Write[35] trait which adds extra functionality to Files, for doing file I/O. By default, a File won't have its methods:

```
let mut f = std::fs::File::open("foo.txt").ok().expect("Couldn't open foo.txt");
let result = f.write("whatever".as_bytes());
```

Here's the error:

```
error: type `std::fs::File` does not implement any method in scope named `write`

let result = f.write(b"whatever");
        ^~~~~~~~~~~~~~~~~~
```

We need to use the Write trait first:

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

let mut f = std::fs::File::open("foo.txt").ok().expect("Couldn't open foo.txt");
let result = f.write("whatever".as_bytes());
```

This will compile without error.

This means that even if someone does something bad like add methods to int, it won't affect you, unless you use that trait.

There's one more restriction on implementing traits. Either the trait or the type you're writing the impl for must be defined by you. So, we could implement the HasArea type for i32, because HasArea is in our code. But if we tried to implement Float, a trait provided by Rust, for i32, we could not, because neither the trait nor the type are in our code.

One last thing about traits: generic functions with a trait bound use 'monomorphization' (mono: one, morph: form), so they are statically dispatched. What's that mean? Check out the chapter on trait objects (section 5.23, page 218) for more details.

---

[35]http://doc.rust-lang.org/std/io/trait.Write.html

5.20.1   Multiple trait bounds

You've seen that you can bound a generic type parameter with a trait:

```
fn foo<T: Clone>(x: T) {
    x.clone();
}
```

If you need more than one bound, you can use +:

```
use std::fmt::Debug;

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

T now needs to be both Clone as well as Debug.

5.20.2   Where clause

Writing functions with only a few generic types and a small number of trait bounds isn't too bad, but as the number increases, the syntax gets increasingly awkward:

```
use std::fmt::Debug;

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

The name of the function is on the far left, and the parameter list is on the far right. The bounds are getting in the way.

Rust has a solution, and it's called a 'where clause':

```
use std::fmt::Debug;

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

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

fn main() {
  foo("Hello", "world");
  bar("Hello", "workd");
}
```

foo() uses the syntax we showed earlier, and bar() uses a where clause. All you need to do is leave off the bounds when defining your type parameters, and then add where after the parameter list. For longer lists, whitespace can be added:

```
use std::fmt::Debug;

fn bar<T, K>(x: T, y: K)
  where T: Clone,
      K: Clone + Debug {

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

This flexibility can add clarity in complex situations.

where is also more powerful than the simpler syntax. For example:

```
trait ConvertTo<Output> {
  fn convert(&self) -> Output;
}

impl ConvertTo<i64> for i32 {
  fn convert(&self) -> i64 { *self as i64 }
}

// can be called with T == i32
fn normal<T: ConvertTo<i64>>(x: &T) -> i64 {
  x.convert()
}

// can be called with T == i64
fn inverse<T>() -> T
```

```
    // this is using ConvertTo as if it were "ConvertFrom<i32>"
    where i32: ConvertTo<T> {
 1i32.convert()
}
```

This shows off the additional feature of `where` clauses: they allow bounds where the left-hand side is an arbitrary type (`i32` in this case), not just a plain type parameter (like `T`).

Default methods

There's one last feature of traits we should cover: default methods. It's easiest just to show an example:

```rust
trait Foo {
    fn bar(&self);

    fn baz(&self) { println!("We called baz."); }
}
```

Implementors of the `Foo` trait need to implement `bar()`, but they don't need to implement `baz()`. They'll get this default behavior. They can override the default if they so choose:

```rust
struct UseDefault;

impl Foo for UseDefault {
    fn bar(&self) { println!("We called bar."); }
}

struct OverrideDefault;

impl Foo for OverrideDefault {
    fn bar(&self) { println!("We called bar."); }

    fn baz(&self) { println!("Override baz!"); }
}

let default = UseDefault;
default.baz(); // prints "We called baz."

let over = OverrideDefault;
over.baz(); // prints "Override baz!"
```

5.20.3   Inheritance

Sometimes, implementing a trait requires implementing another trait:

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

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

Implementors of `FooBar` must also implement `Foo`, like this:

```rust
struct Baz;

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

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

If we forget to implement `Foo`, Rust will tell us:

```
error: the trait `main::Foo` is not implemented for the type `main::Baz` [E0277]
```

5.21   Drop 丢弃

Now that we've discussed traits, let's talk about a particular trait provided by the Rust standard library, `Drop`[36]. The `Drop` trait provides a way to run some code when a value goes out of scope. For example:

```rust
struct HasDrop;

impl Drop for HasDrop {
  fn drop(&mut self) {
    println!("Dropping!");
  }
}

fn main() {
  let x = HasDrop;

  // do stuff
```

---

[36]http://doc.rust-lang.org/std/ops/trait.Drop.html

```
} // x goes out of scope here
```

When x goes out of scope at the end of main(), the code for Drop will run. Drop has one method, which is also called drop(). It takes a mutable reference to self.

That's it! The mechanics of Drop are very simple, but there are some subtleties. For example, values are dropped in the opposite order they are declared. Here's another example:

```rust
struct Firework {
    strength: i32,
}

impl Drop for Firework {
    fn drop(&mut self) {
        println!("BOOM times {}!!!", self.strength);
    }
}

fn main() {
    let firecracker = Firework { strength: 1 };
    let tnt = Firework { strength: 100 };
}
```

This will output:

```
BOOM times 100!!!
BOOM times 1!!!
```

The TNT goes off before the firecracker does, because it was declared afterwards. Last in, first out.

So what is Drop good for? Generally, Drop is used to clean up any resources associated with a struct. For example, the Arc<T> type[37] is a reference-counted type. When Drop is called, it will decrement the reference count, and if the total number of references is zero, will clean up the underlying value.

5.22 if let if-let 语句

if let allows you to combine if and let together to reduce the overhead of certain kinds of pattern matches.

For example, let's say we have some sort of Option<T>. We want to call a function on it if it's Some<T>, but do nothing if it's None. That looks like this:

---

[37]http://doc.rust-lang.org/std/sync/struct.Arc.html

```
match option {
    Some(x) => { foo(x) },
    None => {},
}
```

We don't have to use match here, for example, we could use if:

```
if option.is_some() {
    let x = option.unwrap();
    foo(x);
}
```

Neither of these options is particularly appealing. We can use if let to do the same thing in a nicer way:

```
if let Some(x) = option {
    foo(x);
}
```

If a pattern (section 5.15, page 192) matches successfully, it binds any appropriate parts of the value to the identifiers in the pattern, then evaluates the expression. If the pattern doesn't match, nothing happens.

If you'd rather to do something else when the pattern does not match, you can use else:

```
if let Some(x) = option {
    foo(x);
} else {
    bar();
}
```

while let

In a similar fashion, while let can be used when you want to conditionally loop as long as a value matches a certain pattern. It turns code like this:

```
loop {
    match option {
        Some(x) => println!("{}", x),
        _ => break,
    }
}
```

Into code like this:

```
while let Some(x) = option {
  println!("{}", x);
}
```

5.23   Trait Objects 特质对象

When code involves polymorphism, there needs to be a mechanism to determine which specific version is actually run. This is called 'dispatch'. There are two major forms of dispatch: static dispatch and dynamic dispatch. While Rust favors static dispatch, it also supports dynamic dispatch through a mechanism called 'trait objects'.

Background

For the rest of this chapter, we'll need a trait and some implementations. Let's make a simple one, Foo. It has one method that is expected to return a String.

```
trait Foo {
  fn method(&self) -> String;
}
```

We'll also implement this trait for u8 and String:

```
impl Foo for u8 {
  fn method(&self) -> String { format!("u8: {}", *self) }
}

impl Foo for String {
  fn method(&self) -> String { format!("string: {}", *self) }
}
```

Static dispatch

We can use this trait to perform static dispatch with trait bounds:

```
fn do_something<T: Foo>(x: T) {
  x.method();
}

fn main() {
  let x = 5u8;
  let y = "Hello".to_string();
```

```
  do_something(x);
  do_something(y);
}
```

Rust uses 'monomorphization' to perform static dispatch here. This means that Rust will create a special version of do_something() for both u8 and String, and then replace the call sites with calls to these specialized functions. In other words, Rust generates something like this:

```
fn do_something_u8(x: u8) {
  x.method();
}

fn do_something_string(x: String) {
  x.method();
}

fn main() {
  let x = 5u8;
  let y = "Hello".to_string();

  do_something_u8(x);
  do_something_string(y);
}
```

This has a great upside: static dispatch allows function calls to be inlined because the callee is known at compile time, and inlining is the key to good optimization. Static dispatch is fast, but it comes at a tradeoff: 'code bloat', due to many copies of the same function existing in the binary, one for each type.

Furthermore, compilers aren't perfect and may "optimize" code to become slower. For example, functions inlined too eagerly will bloat the instruction cache (cache rules everything around us). This is part of the reason that #[inline] and #[inline(always)] should be used carefully, and one reason why using a dynamic dispatch is sometimes more efficient.

However, the common case is that it is more efficient to use static dispatch, and one can always have a thin statically-dispatched wrapper function that does a dynamic dispatch, but not vice versa, meaning static calls are more flexible. The standard library tries to be statically dispatched where possible for this reason.

Dynamic dispatch

Rust provides dynamic dispatch through a feature called 'trait objects'. Trait objects, like &Foo or Box<Foo>, are normal values that store a value of *any* type that implements the given trait, where the precise type can only be known at runtime.

A trait object can be obtained from a pointer to a concrete type that implements the trait by *casting* it (e.g. &x as &Foo) or *coercing* it (e.g. using &x as an argument to a function that takes &Foo).

These trait object coercions and casts also work for pointers like &mut T to &mut Foo and Box<T> to Box<Foo>, but that's all at the moment. Coercions and casts are identical.

This operation can be seen as 'erasing' the compiler's knowledge about the specific type of the pointer, and hence trait objects are sometimes referred to as 'type erasure'.

Coming back to the example above, we can use the same trait to perform dynamic dispatch with trait objects by casting:

```rust
fn do_something(x: &Foo) {
  x.method();
}

fn main() {
  let x = 5u8;
  do_something(&x as &Foo);
}
```

or by coercing:

```rust
fn do_something(x: &Foo) {
  x.method();
}

fn main() {
  let x = "Hello".to_string();
  do_something(&x);
}
```

A function that takes a trait object is not specialized to each of the types that implements Foo: only one copy is generated, often (but not always) resulting in less code bloat. However, this comes at the cost of requiring slower virtual function calls, and effectively inhibiting any chance of inlining and related optimizations from occurring.

Why pointers?   Rust does not put things behind a pointer by default, unlike many managed languages, so types can have different sizes. Knowing the size of the value at compile time is important for things like passing it as an argument to a function, moving it about on the stack and allocating (and deallocating) space on the heap to store it.

For Foo, we would need to have a value that could be at least either a String (24 bytes) or a u8 (1 byte), as well as any other type for which dependent crates may implement Foo (any number of bytes at all). There's no way to guarantee that this last point can work if the values are stored without a pointer, because those other types can be arbitrarily large.

Putting the value behind a pointer means the size of the value is not relevant when we are tossing a trait object around, only the size of the pointer itself.

Representation   The methods of the trait can be called on a trait object via a special record of function pointers traditionally called a 'vtable' (created and managed by the compiler).

Trait objects are both simple and complicated: their core representation and layout is quite straight-forward, but there are some curly error messages and surprising behaviors to discover.

Let's start simple, with the runtime representation of a trait object. The std::raw module contains structs with layouts that are the same as the complicated built-in types, including trait objects[38]:

```
pub struct TraitObject {
    pub data: *mut (),
    pub vtable: *mut (),
}
```

That is, a trait object like &Foo consists of a 'data' pointer and a 'vtable' pointer.

The data pointer addresses the data (of some unknown type T) that the trait object is storing, and the vtable pointer points to the vtable ('virtual method table') corresponding to the implementation of Foo for T.

A vtable is essentially a struct of function pointers, pointing to the concrete piece of machine code for each method in the implementation. A method call like trait_object.method() will retrieve the correct pointer out of the vtable and then do a dynamic call of it. For example:

```
struct FooVtable {
    destructor: fn(*mut ()),
    size: usize,
    align: usize,
    method: fn(*const ()) -> String,
}

// u8:

fn call_method_on_u8(x: *const ()) -> String {
    // the compiler guarantees that this function is only called
    // with `x` pointing to a u8
    let byte: &u8 = unsafe { &*(x as *const u8) };

    byte.method()
}

static Foo_for_u8_vtable: FooVtable = FooVtable {
    destructor: /* compiler magic */,
    size: 1,
    align: 1,
```

---

[38]http://doc.rust-lang.org/std/raw/struct.TraitObject.html

```
  // cast to a function pointer
  method: call_method_on_u8 as fn(*const ()) -> String,
};


// String:

fn call_method_on_String(x: *const ()) -> String {
  // the compiler guarantees that this function is only called
  // with `x` pointing to a String
  let string: &String = unsafe { &*(x as *const String) };

  string.method()
}

static Foo_for_String_vtable: FooVtable = FooVtable {
  destructor: /* compiler magic */,
  // values for a 64-bit computer, halve them for 32-bit ones
  size: 24,
  align: 8,

  method: call_method_on_String as fn(*const ()) -> String,
};
```

The destructor field in each vtable points to a function that will clean up any resources of the vtable's type, for u8 it is trivial, but for String it will free the memory. This is necessary for owning trait objects like Box<Foo>, which need to clean-up both the Box allocation as well as the internal type when they go out of scope. The size and align fields store the size of the erased type, and its alignment requirements; these are essentially unused at the moment since the information is embedded in the destructor, but will be used in the future, as trait objects are progressively made more flexible.

Suppose we've got some values that implement Foo, then the explicit form of construction and use of Foo trait objects might look a bit like (ignoring the type mismatches: they're all just pointers anyway):

```
let a: String = "foo".to_string();
let x: u8 = 1;

// let b: &Foo = &a;
let b = TraitObject {
  // store the data
  data: &a,
  // store the methods
  vtable: &Foo_for_String_vtable
};
```

```
// let y: &Foo = x;
let y = TraitObject {
  // store the data
  data: &x,
  // store the methods
  vtable: &Foo_for_u8_vtable
};

// b.method();
(b.vtable.method)(b.data);

// y.method();
(y.vtable.method)(y.data);
```

If b or y were owning trait objects (Box<Foo>), there would be a (b.vtable.destructor)(b.data) (respectively y) call when they went out of scope.

5.24 Closures 闭包

Rust not only has named functions, but anonymous functions as well. Anonymous functions that have an associated environment are called 'closures', because they close over an environment. Rust has a really great implementation of them, as we'll see.

5.24.1 Syntax

Closures look like this:

```
let plus_one = |x: i32| x + 1;

assert_eq!(2, plus_one(1));
```

We create a binding, plus_one, and assign it to a closure. The closure's arguments go between the pipes (|), and the body is an expression, in this case, x + 1. Remember that {} is an expression, so we can have multi-line closures too:

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

  result += 1;
  result += 1;

  result
```

```
};

assert_eq!(4, plus_two(2));
```

You'll notice a few things about closures that are a bit different than regular functions defined with fn. The first of which is that we did not need to annotate the types of arguments the closure takes or the values it returns. We can:

```
let plus_one = |x: i32| -> i32 { x + 1 };

assert_eq!(2, plus_one(1));
```

But we don't have to. Why is this? Basically, it was chosen for ergonomic reasons. While specifying the full type for named functions is helpful with things like documentation and type inference, the types of closures are rarely documented since they're anonymous, and they don't cause the kinds of error-at-a-distance that inferring named function types can.

The second is that the syntax is similar, but a bit different. I've added spaces here to make them look a little closer:

```
fn  plus_one_v1  (x: i32 ) -> i32 { x + 1 }
let plus_one_v2 = |x: i32 | -> i32 { x + 1 };
let plus_one_v3 = |x: i32 |        x + 1  ;
```

Small differences, but they're similar in ways.

5.24.2   Closures and their environment

Closures are called such because they 'close over their environment'. It looks like this:

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

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

This closure, plus_num, refers to a let binding in its scope: num. More specifically, it borrows the binding. If we do something that would conflict with that binding, we get an error. Like this one:

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

let y = &mut num;
```

Which errors with:

```
error: cannot borrow `num` as mutable because it is also borrowed as immutable
   let y = &mut num;
           ^~~
note: previous borrow of `num` occurs here due to use in closure; the immutable
 borrow prevents subsequent moves or mutable borrows of `num` until the borrow
 ends
   let plus_num = |x| x + num;
                 ^~~~~~~~~~~
note: previous borrow ends here
fn main() {
   let mut num = 5;
   let plus_num = |x| x + num;

   let y = &mut num;
}
^
```

A verbose yet helpful error message! As it says, we can't take a mutable borrow on `num` because the closure is already borrowing it. If we let the closure go out of scope, we can:

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

} // plus_num goes out of scope, borrow of num ends

let y = &mut num;
```

If your closure requires it, however, Rust will take ownership and move the environment instead:

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

let takes_nums = || nums;

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

This gives us:

```
note: `nums` moved into closure environment here because it has type
  `[closure(()) -> collections::vec::Vec<i32>]`, which is non-copyable
let takes_nums = || nums;
                    ^~~~~~~
```

Vec<T> has ownership over its contents, and therefore, when we refer to it in our closure, we have to take ownership of `nums`. It's the same as if we'd passed `nums` to a function that took ownership of it.

move closures

We can force our closure to take ownership of its environment with the move keyword:

```rust
let num = 5;

let owns_num = move |x: i32| x + num;
```

Now, even though the keyword is move, the variables follow normal move semantics. In this case, 5 implements Copy, and so owns_num takes ownership of a copy of num. So what's the difference?

```rust
let mut num = 5;

{
  let mut add_num = |x: i32| num += x;

  add_num(5);
}

assert_eq!(10, num);
```

So in this case, our closure took a mutable reference to num, and then when we called add_num, it mutated the underlying value, as we'd expect. We also needed to declare add_num as mut too, because we're mutating its environment.

If we change to a move closure, it's different:

```rust
let mut num = 5;

{
  let mut add_num = move |x: i32| num += x;

  add_num(5);
}

assert_eq!(5, num);
```

We only get 5. Rather than taking a mutable borrow out on our num, we took ownership of a copy.

Another way to think about move closures: they give a closure its own stack frame. Without move, a closure may be tied to the stack frame that created it, while a move closure is self-contained. This means that you cannot generally return a non-move closure from a function, for example.

But before we talk about taking and returning closures, we should talk some more about the way that closures are implemented. As a systems language, Rust gives you tons of control over what your code does, and closures are no different.

5.24.3 Closure implementation

Rust's implementation of closures is a bit different than other languages. They are effectively syntax sugar for traits. You'll want to make sure to have read the traits chapter (section 5.20, page 208) before this one, as well as the chapter on trait objects (section 5.23, page 218).

Got all that? Good.

The key to understanding how closures work under the hood is something a bit strange: Using () to call a function, like foo(), is an overloadable operator. From this, everything else clicks into place. In Rust, we use the trait system to overload operators. Calling functions is no different. We have three separate traits to overload with:

```rust
pub trait Fn<Args> : FnMut<Args> {
    extern "rust-call" fn call(&self, args: Args) -> Self::Output;
}

pub trait FnMut<Args> : FnOnce<Args> {
    extern "rust-call" fn call_mut(&mut self, args: Args) -> Self::Output;
}

pub trait FnOnce<Args> {
    type Output;

    extern "rust-call" fn call_once(self, args: Args) -> Self::Output;
}
```

You'll notice a few differences between these traits, but a big one is self: Fn takes &self, FnMut takes &mut self, and FnOnce takes self. This covers all three kinds of self via the usual method call syntax. But we've split them up into three traits, rather than having a single one. This gives us a large amount of control over what kind of closures we can take.

The ||{} syntax for closures is sugar for these three traits. Rust will generate a struct for the environment, impl the appropriate trait, and then use it.

5.24.4 Taking closures as arguments

Now that we know that closures are traits, we already know how to accept and return closures: just like any other trait!

This also means that we can choose static vs dynamic dispatch as well. First, let's write a function which takes something callable, calls it, and returns the result:

```rust
fn call_with_one<F>(some_closure: F) -> i32
    where F : Fn(i32) -> i32 {

    some_closure(1)
```

```
}

let answer = call_with_one(|x| x + 2);

assert_eq!(3, answer);
```

We pass our closure, |x| x + 2, to call_with_one. It just does what it suggests: it calls the closure, giving it 1 as an argument.

Let's examine the signature of call_with_one in more depth:

```
fn call_with_one<F>(some_closure: F) -> i32
```

We take one parameter, and it has the type F. We also return a i32. This part isn't interesting. The next part is:

```
    where F : Fn(i32) -> i32 {
```

Because Fn is a trait, we can bound our generic with it. In this case, our closure takes a i32 as an argument and returns an i32, and so the generic bound we use is Fn(i32) -> i32.

There's one other key point here: because we're bounding a generic with a trait, this will get monomorphized, and therefore, we'll be doing static dispatch into the closure. That's pretty neat. In many languages, closures are inherently heap allocated, and will always involve dynamic dispatch. In Rust, we can stack allocate our closure environment, and statically dispatch the call. This happens quite often with iterators and their adapters, which often take closures as arguments.

Of course, if we want dynamic dispatch, we can get that too. A trait object handles this case, as usual:

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

let answer = call_with_one(&|x| x + 2);

assert_eq!(3, answer);
```

Now we take a trait object, a &Fn. And we have to make a reference to our closure when we pass it to call_with_one, so we use &||.

5.24.5 Returning closures

It's very common for functional-style code to return closures in various situations. If you try to return a closure, you may run into an error. At first, it may seem strange, but we'll figure it out. Here's how you'd probably try to return a closure from a function:

```rust
fn factory() -> (Fn(i32) -> Vec<i32>) {
    let vec = vec![1, 2, 3];

    |n| vec.push(n)
}

let f = factory();

let answer = f(4);
assert_eq!(vec![1, 2, 3, 4], answer);
```

This gives us these long, related errors:

```
error: the trait `core::marker::Sized` is not implemented for the type
`core::ops::Fn(i32) -> collections::vec::Vec<i32>` [E0277]
f = factory();
^
note: `core::ops::Fn(i32) -> collections::vec::Vec<i32>` does not have a
constant size known at compile-time
f = factory();
^
error: the trait `core::marker::Sized` is not implemented for the type
`core::ops::Fn(i32) -> collections::vec::Vec<i32>` [E0277]
factory() -> (Fn(i32) -> Vec<i32>) {
        ^~~~~~~~~~~~~~~~~~~~~
note: `core::ops::Fn(i32) -> collections::vec::Vec<i32>` does not have a constant size
▨ known at compile-time
fa ctory() -> (Fn(i32) -> Vec<i32>) {
        ^~~~~~~~~~~~~~~~~~~~~
```

In order to return something from a function, Rust needs to know what size the return type is. But since `Fn` is a trait, it could be various things of various sizes: many different types can implement `Fn`. An easy way to give something a size is to take a reference to it, as references have a known size. So we'd write this:

```rust
fn factory() -> &(Fn(i32) -> Vec<i32>) {
    let vec = vec![1, 2, 3];

    |n| vec.push(n)
}

let f = factory();

let answer = f(4);
assert_eq!(vec![1, 2, 3, 4], answer);
```

But we get another error:

```
error: missing lifetime specifier [E0106]
fn factory() -> &(Fn(i32) -> i32) {
        ^~~~~~~~~~~~~~~~
```

Right. Because we have a reference, we need to give it a lifetime. But our `factory()` function takes no arguments, so elision doesn't kick in here. What lifetime can we choose? `'static`:

```rust
fn factory() -> &'static (Fn(i32) -> i32) {
    let num = 5;

    |x| x + num
}

let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

But we get another error:

```
error: mismatched types:
 expected `&'static core::ops::Fn(i32) -> i32`,
   found `[closure <anon>:7:9: 7:20]`
(expected &-ptr,
   found closure) [E0308]
     |x| x + num
     ^~~~~~~~~~~
```

This error is letting us know that we don't have a `&'static Fn(i32) -> i32`, we have a `[closure <anon>:7:9: 7:20]`. Wait, what?

Because each closure generates its own environment `struct` and implementation of `Fn` and friends, these types are anonymous. They exist just solely for this closure. So Rust shows them as `closure <anon>`, rather than some autogenerated name.

But why doesn't our closure implement `&'static Fn`? Well, as we discussed before, closures borrow their environment. And in this case, our environment is based on a stack-allocated 5, the `num` variable binding. So the borrow has a lifetime of the stack frame. So if we returned this closure, the function call would be over, the stack frame would go away, and our closure is capturing an environment of garbage memory!

So what to do? This *almost* works:

```rust
fn factory() -> Box<Fn(i32) -> i32> {
    let num = 5;

    Box::new(|x| x + num)
}
let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

We use a trait object, by `Boxing` up the `Fn`. There's just one last problem:

```
error: `num` does not live long enough
Box::new(|x| x + num)
         ^~~~~~~~~~~
```

We still have a reference to the parent stack frame. With one last fix, we can make this work:

```rust
fn factory() -> Box<Fn(i32) -> i32> {
    let num = 5;

    Box::new(move |x| x + num)
}
let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

By making the inner closure a `move Fn`, we create a new stack frame for our closure. By `Boxing` it up, we've given it a known size, and allowing it to escape our stack frame.

5.25 Universal Function Call Syntax 通用函数调用语法

Sometimes, functions can have the same names. Consider this code:

```rust
trait Foo {
    fn f(&self);
}

trait Bar {
    fn f(&self);
}
```

```rust
struct Baz;

impl Foo for Baz {
  fn f(&self) { println!("Baz's impl of Foo"); }
}

impl Bar for Baz {
  fn f(&self) { println!("Baz's impl of Bar"); }
}

let b = Baz;
```

If we were to try to call b.f(), we'd get an error:

```
error: multiple applicable methods in scope [E0034]
b.f();
 ^~~
note: candidate #1 is defined in an impl of the trait `main::Foo` for the type
`main::Baz`
  fn f(&self) { println!("Baz's impl of Foo"); }
  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
note: candidate #2 is defined in an impl of the trait `main::Bar` for the type
`main::Baz`
  fn f(&self) { println!("Baz's impl of Bar"); }
  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

We need a way to disambiguate which method we need. This feature is called 'universal function call syntax', and it looks like this:

```rust
Foo::f(&b);
Bar::f(&b);
```

Let's break it down.

```rust
Foo::
Bar::
```

These halves of the invocation are the types of the two traits: Foo and Bar. This is what ends up actually doing the disambiguation between the two: Rust calls the one from the trait name you use.

```rust
f(&b)
```

When we call a method like b.f() using method syntax (section 5.16, page 197), Rust will automatically borrow b if f() takes &self. In this case, Rust will not, and so we need to pass an explicit &b.

5.25.1 Angle-bracket Form

The form of UFCS we just talked about:

```
Trait::method(args);
```

Is a short-hand. There's an expanded form of this that's needed in some situations:

```
<Type as Trait>::method(args);
```

The <>:: syntax is a means of providing a type hint. The type goes inside the <>s. In this case, the type is Type as Trait, indicating that we want Trait's version of method to be called here. The as Trait part is optional if it's not ambiguous. Same with the angle brackets, hence the shorter form.

Here's an example of using the longer form.

```rust
trait Foo {
    fn clone(&self);
}

#[derive(Clone)]
struct Bar;

impl Foo for Bar {
    fn clone(&self) {
        println!("Making a clone of Bar");

        <Bar as Clone>::clone(self);
    }
}
```

This will call the Clone trait's clone() method, rather than Foo's.

5.26 Crates and Modules 箱型和模型

When a project starts getting large, it's considered good software engineering practice to split it up into a bunch of smaller pieces, and then fit them together. It's also important to have a well-defined interface, so that some of your functionality is private, and some is public. To facilitate these kinds of things, Rust has a module system.
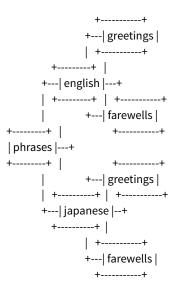
5.26.1 Basic terminology: Crates and Modules

Rust has two distinct terms that relate to the module system: 'crate' and 'module'. A crate is synonymous with a 'library' or 'package' in other languages. Hence "Cargo" as the name of Rust's

package management tool: you ship your crates to others with Cargo. Crates can produce an executable or a library, depending on the project.

Each crate has an implicit *root module* that contains the code for that crate. You can then define a tree of sub-modules under that root module. Modules allow you to partition your code within the crate itself.

As an example, let's make a *phrases* crate, which will give us various phrases in different languages. To keep things simple, we'll stick to 'greetings' and 'farewells' as two kinds of phrases, and use English and Japanese (日本語) as two languages for those phrases to be in. We'll use this module layout:

```
                    +----------+
              +---| greetings |
              |   +----------+
     +---------+  |
     +---| english |---+
     |   +---------+  |   +----------+
     |           +---| farewells |
+---------+  |           +----------+
| phrases |---+
+---------+  |           +----------+
     |           +---| greetings |
     |   +----------+  |   +----------+
     +---| japanese |--+
       +---------+  |
              |   +----------+
              +---| farewells |
                  +----------+
```

In this example, phrases is the name of our crate. All of the rest are modules. You can see that they form a tree, branching out from the crate *root*, which is the root of the tree: phrases itself.

Now that we have a plan, let's define these modules in code. To start, generate a new crate with Cargo:

```
$ cargo new phrases
$ cd phrases
```

If you remember, this generates a simple project for us:

```
$ tree .
.
⬚—— Cargo.toml
⬚—— src
   ⬚—— lib.rs

1 directory, 2 files
```

src/lib.rs is our crate root, corresponding to the phrases in our diagram above.

5.26.2 Defining Modules

To define each of our modules, we use the mod keyword. Let's make our src/lib.rs look like this:

```
mod english {
  mod greetings {
  }

  mod farewells {
  }
}

mod japanese {
  mod greetings {
  }

  mod farewells {
  }
}
```

After the mod keyword, you give the name of the module. Module names follow the conventions for other Rust identifiers: lower_snake_case. The contents of each module are within curly braces ({}).

Within a given mod, you can declare sub-mods. We can refer to sub-modules with double-colon (::) notation: our four nested modules are english::greetings, english::farewells, japanese::greetings, and japanese::farewells. Because these sub-modules are namespaced under their parent module, the names don't conflict: english::greetings and japanese::greetings are distinct, even though their names are both greetings.

Because this crate does not have a main() function, and is called lib.rs, Cargo will build this crate as a library:

```
$ cargo build
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
$ ls target/debug
build  deps  examples  libphrases-a7448e02a0468eaa.rlib  native
```

libphrase-hash.rlib is the compiled crate. Before we see how to use this crate from another crate, let's break it up into multiple files.

5.26.3 Multiple file crates

If each crate were just one file, these files would get very large. It's often easier to split up crates into multiple files, and Rust supports this in two ways.

Instead of declaring a module like this:

```rust
mod english {
    // contents of our module go here
}
```

We can instead declare our module like this:

```rust
mod english;
```

If we do that, Rust will expect to find either a english.rs file, or a english/mod.rs file with the contents of our module.

Note that in these files, you don't need to re-declare the module: that's already been done with the initial mod declaration.

Using these two techniques, we can break up our crate into two directories and seven files:

```
$ tree .
.
├── Cargo.lock
├── Cargo.toml
├── src
│   ├── english
│   │   ├── farewells.rs
│   │   ├── greetings.rs
│   │   └── mod.rs
│   ├── japanese
│   │   ├── farewells.rs
│   │   ├── greetings.rs
│   │   └── mod.rs
│   └── lib.rs
├── target
    └── debug
        ├── build
        ├── deps
        ├── examples
        ├── libphrases-a7448e02a0468eaa.rlib
        └── native
```

src/lib.rs is our crate root, and looks like this:

```rust
mod english;
mod japanese;
```

These two declarations tell Rust to look for either src/english.rs and src/japanese.rs, or src/english/mod.rs and src/japanese/mod.rs, depending on our preference. In this case, because our modules have sub-modules, we've chosen the second. Both src/english/mod.rs and src/japanese/mod.rs look like this:

```
mod greetings;
mod farewells;
```

Again, these declarations tell Rust to look for either src/english/greetings.rs and src/japanese/greetings.rs or src/english/farewells/mod.rs and src/japanese/farewells/mod.rs. Because these sub-modules don't have their own sub-modules, we've chosen to make them src/english/greetings.rs and src/japanese/farewells.rs. Whew!

The contents of src/english/greetings.rs and src/japanese/farewells.rs are both empty at the moment. Let's add some functions.

Put this in src/english/greetings.rs:

```
fn hello() -> String {
    "Hello!".to_string()
}
```

Put this in src/english/farewells.rs:

```
fn goodbye() -> String {
    "Goodbye.".to_string()
}
```

Put this in src/japanese/greetings.rs:

```
fn hello() -> String {
    "こんにちは".to_string()
}
```

Of course, you can copy and paste this from this web page, or just type something else. It's not important that you actually put 'konnichiwa' to learn about the module system.

Put this in src/japanese/farewells.rs:

```
fn goodbye() -> String {
    "さようなら".to_string()
}
```

(This is 'Sayōnara', if you're curious.)

Now that we have some functionality in our crate, let's try to use it from another crate.

5.26.4   Importing External Crates

We have a library crate. Let's make an executable crate that imports and uses our library.

Make a src/main.rs and put this in it (it won't quite compile yet):

```rust
extern crate phrases;

fn main() {
    println!("Hello in English: {}", phrases::english::greetings::hello());
    println!("Goodbye in English: {}", phrases::english::farewells::goodbye());

    println!("Hello in Japanese: {}", phrases::japanese::greetings::hello());
    println!("Goodbye in Japanese: {}", phrases::japanese::farewells::goodbye());
}
```

The extern crate declaration tells Rust that we need to compile and link to the phrases crate. We can then use phrases' modules in this one. As we mentioned earlier, you can use double colons to refer to sub-modules and the functions inside of them.

Also, Cargo assumes that src/main.rs is the crate root of a binary crate, rather than a library crate. Our package now has two crates: src/lib.rs and src/main.rs. This pattern is quite common for executable crates: most functionality is in a library crate, and the executable crate uses that library. This way, other programs can also use the library crate, and it's also a nice separation of concerns.

This doesn't quite work yet, though. We get four errors that look similar to this:

```
$ cargo build
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
src/main.rs:4:38: 4:72 error: function `hello` is private
src/main.rs:4     println!("Hello in English: {}", phrases::english::greetings::hello()
⊠ );
                            ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
note: in expansion of format_args!
<std macros>:2:25: 2:58 note: expansion site
<std macros>:1:1: 2:62 note: in expansion of print!
<std macros>:3:1: 3:54 note: expansion site
<std macros>:1:1: 3:58 note: in expansion of println!
phrases/src/main.rs:4:5: 4:76 note: expansion site
```

By default, everything is private in Rust. Let's talk about this in some more depth.

5.26.5   Exporting a Public Interface

Rust allows you to precisely control which aspects of your interface are public, and so private is the default. To make things public, you use the pub keyword. Let's focus on the english module first, so let's reduce our src/main.rs to just this:

```rust
extern crate phrases;

fn main() {
    println!("Hello in English: {}", phrases::english::greetings::hello());
    println!("Goodbye in English: {}", phrases::english::farewells::goodbye());
}
```

In our src/lib.rs, let's add pub to the english module declaration:

```rust
pub mod english;
mod japanese;
```

And in our src/english/mod.rs, let's make both pub:

```rust
pub mod greetings;
pub mod farewells;
```

In our src/english/greetings.rs, let's add pub to our fn declaration:

```rust
pub fn hello() -> String {
    "Hello!".to_string()
}
```

And also in src/english/farewells.rs:

```rust
pub fn goodbye() -> String {
    "Goodbye.".to_string()
}
```

Now, our crate compiles, albeit with warnings about not using the japanese functions:

```
$ cargo run
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
src/japanese/greetings.rs:1:1: 3:2 warning: function is never used: `hello`, #[warn(dea
⊠ d_code)] on by default
src/japanese/greetings.rs:1 fn hello() -> String {
src/japanese/greetings.rs:2     "こんにちは".to_string()
src/japanese/greetings.rs:3 }
src/japanese/farewells.rs:1:1: 3:2 warning: function is never used: `goodbye`, #[warn(d
⊠ ead_code)] on by default
src/japanese/farewells.rs:1 fn goodbye() -> String {
src/japanese/farewells.rs:2     "さようなら".to_string()
src/japanese/farewells.rs:3 }
     Running `target/debug/phrases`
Hello in English: Hello!
Goodbye in English: Goodbye.
```

Now that our functions are public, we can use them. Great! However, typing out phrases::english::greetings::hello() is very long and repetitive. Rust has another keyword for importing names into the current scope, so that you can refer to them with shorter names. Let's talk about use.

5.26.6 Importing Modules with **use**

Rust has a use keyword, which allows us to import names into our local scope. Let's change our src/main.rs to look like this:

```
extern crate phrases;

use phrases::english::greetings;
use phrases::english::farewells;

fn main() {
  println!("Hello in English: {}", greetings::hello());
  println!("Goodbye in English: {}", farewells::goodbye());
}
```

The two use lines import each module into the local scope, so we can refer to the functions by a much shorter name. By convention, when importing functions, it's considered best practice to import the module, rather than the function directly. In other words, you *can* do this:

```
extern crate phrases;

use phrases::english::greetings::hello;
use phrases::english::farewells::goodbye;

fn main() {
  println!("Hello in English: {}", hello());
  println!("Goodbye in English: {}", goodbye());
}
```

But it is not idiomatic. This is significantly more likely to introduce a naming conflict. In our short program, it's not a big deal, but as it grows, it becomes a problem. If we have conflicting names, Rust will give a compilation error. For example, if we made the japanese functions public, and tried to do this:

```
extern crate phrases;

use phrases::english::greetings::hello;
use phrases::japanese::greetings::hello;
```

```
fn main() {
  println!("Hello in English: {}", hello());
  println!("Hello in Japanese: {}", hello());
}
```

Rust will give us a compile-time error:

```
  Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
src/main.rs:4:5: 4:40 error: a value named `hello` has already been imported in this mo
⊠ dule [E0252]
src/main.rs:4 use phrases::japanese::greetings::hello;
          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
error: aborting due to previous error
Could not compile `phrases`.
```

If we're importing multiple names from the same module, we don't have to type it out twice. Instead of this:

```
use phrases::english::greetings;
use phrases::english::farewells;
```

We can use this shortcut:

```
use phrases::english::{greetings, farewells};
```

Re-exporting with `pub use`

You don't just use `use` to shorten identifiers. You can also use it inside of your crate to re-export a function inside another module. This allows you to present an external interface that may not directly map to your internal code organization.

Let's look at an example. Modify your src/main.rs to read like this:

```
extern crate phrases;

use phrases::english::{greetings,farewells};
use phrases::japanese;

fn main() {
  println!("Hello in English: {}", greetings::hello());
  println!("Goodbye in English: {}", farewells::goodbye());

  println!("Hello in Japanese: {}", japanese::hello());
  println!("Goodbye in Japanese: {}", japanese::goodbye());
}
```

Then, modify your src/lib.rs to make the japanese mod public:

```
pub mod english;
pub mod japanese;
```

Next, make the two functions public, first in src/japanese/greetings.rs:

```
pub fn hello() -> String {
  "こんにちは".to_string()
}
```

And then in src/japanese/farewells.rs:

```
pub fn goodbye() -> String {
  "さようなら".to_string()
}
```

Finally, modify your src/japanese/mod.rs to read like this:

```
pub use self::greetings::hello;
pub use self::farewells::goodbye;

mod greetings;
mod farewells;
```

The pub use declaration brings the function into scope at this part of our module hierarchy. Because we've pub used this inside of our japanese module, we now have a phrases::japanese::hello() function and a phrases::japanese::goodbye() function, even though the code for them lives in phrases::japanese::greetings::hello() and phrases::japanese::farewells::goodbye(). Our internal organization doesn't define our external interface.

Here we have a pub use for each function we want to bring into the japanese scope. We could alternatively use the wildcard syntax to include everything from greetings into the current scope: pub use self::greetings::*.

What about the self? Well, by default, use declarations are absolute paths, starting from your crate root. self makes that path relative to your current place in the hierarchy instead. There's one more special form of use: you can use super:: to reach one level up the tree from your current location. Some people like to think of self as . and super as .., from many shells' display for the current directory and the parent directory.

Outside of use, paths are relative: foo::bar() refers to a function inside of foo relative to where we are. If that's prefixed with ::, as in ::foo::bar(), it refers to a different foo, an absolute path from your crate root.

Also, note that we pub used before we declared our mods. Rust requires that use declarations go first.

This will build and run:

```
$ cargo run
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
    Running `target/debug/phrases`
Hello in English: Hello!
Goodbye in English: Goodbye.
Hello in Japanese: こんにちは
Goodbye in Japanese: さようなら
```

5.27   const and static 常量定义与静态型

Rust has a way of defining constants with the const keyword:

```
const N: i32 = 5;
```

Unlike let (section 5.1, page 147) bindings, you must annotate the type of a const.

Constants live for the entire lifetime of a program. More specifically, constants in Rust have no fixed address in memory. This is because they're effectively inlined to each place that they're used. References to the same constant are not necessarily guaranteed to refer to the same memory address for this reason.

5.27.1   static

Rust provides a 'global variable' sort of facility in static items. They're similar to constants, but static items aren't inlined upon use. This means that there is only one instance for each value, and it's at a fixed location in memory.

Here's an example:

```
static N: i32 = 5;
```

Unlike let (section 5.1, page 147) bindings, you must annotate the type of a static.

Statics live for the entire lifetime of a program, and therefore any reference stored in a constant has a 'static lifetime (section 5.10, page 177):

```
static NAME: &'static str = "Steve";
```

Mutability

You can introduce mutability with the mut keyword:

```
static mut N: i32 = 5;
```

Because this is mutable, one thread could be updating N while another is reading it, causing memory unsafety. As such both accessing and mutating a static mut is unsafe (section 5.37, page 273), and so must be done in an unsafe block:

```
unsafe {
  N += 1;

  println!("N: {}", N);
}
```

Furthermore, any type stored in a static must be Sync.

5.27.2   Initializing

Both const and static have requirements for giving them a value. They may only be given a value that's a constant expression. In other words, you cannot use the result of a function call or anything similarly complex or at runtime.

5.27.3   Which construct should I use?

Almost always, if you can choose between the two, choose const. It's pretty rare that you actually want a memory location associated with your constant, and using a const allows for optimizations like constant propagation not only in your crate but downstream crates.

A const can be thought of as a #define in C: it has metadata overhead but it has no runtime overhead. "Should I use a #define or a static in C," is largely the same question as whether you should use a const or a static in Rust.

5.28   Attributes 属性

Declarations can be annotated with 'attributes' in Rust. They look like this:

```
#[test]
```

or like this:

```
#![test]
```

The difference between the two is the !, which changes what the attribute applies to:

```
#[foo]
struct Foo;

mod bar {
  #![bar]
}
```

The #[foo] attribute applies to the next item, which is the struct declaration. The #![bar] attribute applies to the item enclosing it, which is the mod declaration. Otherwise, they're the same. Both change the meaning of the item they're attached to somehow.

For example, consider a function like this:

```
#[test]
fn check() {
  assert_eq!(2, 1 + 1);
}
```

It is marked with #[test]. This means it's special: when you run tests (section 4.2, page 80), this function will execute. When you compile as usual, it won't even be included. This function is now a test function.

Attributes may also have additional data:

```
#[inline(always)]
fn super_fast_fn() {
```

Or even keys and values:

```
#[cfg(target_os = "macos")]
mod macos_only {
```

Rust attributes are used for a number of different things. There is a full list of attributes in the reference[39]. Currently, you are not allowed to create your own attributes, the Rust compiler defines them.

5.29   **type** aliases **类型**

The type keyword lets you declare an alias of another type:

---

[39]http://doc.rust-lang.org/reference.html#attributes

```rust
type Name = String;
```

You can then use this type as if it were a real type:

```rust
type Name = String;

let x: Name = "Hello".to_string();
```

Note, however, that this is an *alias*, not a new type entirely. In other words, because Rust is strongly typed, you'd expect a comparison between two different types to fail:

```rust
let x: i32 = 5;
let y: i64 = 5;

if x == y {
  // ...
}
```

this gives

```
error: mismatched types:
 expected `i32`,
   found `i64`
(expected i32,
   found i64) [E0308]
   if x == y {
        ^
```

But, if we had an alias:

```rust
type Num = i32;

let x: i32 = 5;
let y: Num = 5;

if x == y {
  // ...
}
```

This compiles without error. Values of a Num type are the same as a value of type i32, in every way.

You can also use type aliases with generics:

```
use std::result;

enum ConcreteError {
  Foo,
  Bar,
}

type Result<T> = result::Result<T, ConcreteError>;
```

This creates a specialized version of the Result type, which always has a ConcreteError for the E part of Result<T, E>. This is commonly used in the standard library to create custom errors for each subsection. For example, io::Result[40].

5.30 Casting between types 类型转换

Rust, with its focus on safety, provides two different ways of casting different types between each other. The first, as, is for safe casts. In contrast, transmute allows for arbitrary casting, and is one of the most dangerous features of Rust!

5.30.1 **as**

The as keyword does basic casting:

```
let x: i32 = 5;

let y = x as i64;
```

It only allows certain kinds of casting, however:

```
let a = [0u8, 0u8, 0u8, 0u8];

let b = a as u32; // four eights makes 32
```

This errors with:

```
error: non-scalar cast: `[u8; 4]` as `u32`
let b = a as u32; // four eights makes 32
    ^~~~~~~~
```

It's a 'non-scalar cast' because we have multiple values here: the four elements of the array. These kinds of casts are very dangerous, because they make assumptions about the way that multiple underlying structures are implemented. For this, we need something more dangerous.

---

[40] http://doc.rust-lang.org/std/io/type.Result.html

5.30.2 **transmute**

The transmute function is provided by a compiler intrinsic (section 6.4, page 290), and what it does is very simple, but very scary. It tells Rust to treat a value of one type as though it were another type. It does this regardless of the typechecking system, and just completely trusts you.

In our previous example, we know that an array of four u8s represents a u32 properly, and so we want to do the cast. Using transmute instead of as, Rust lets us:

```rust
use std::mem;

unsafe {
  let a = [0u8, 0u8, 0u8, 0u8];

  let b = mem::transmute::<[u8; 4], u32>(a);
}
```

We have to wrap the operation in an unsafe block for this to compile successfully. Technically, only the mem::transmute call itself needs to be in the block, but it's nice in this case to enclose everything related, so you know where to look. In this case, the details about a are also important, and so they're in the block. You'll see code in either style, sometimes the context is too far away, and wrapping all of the code in unsafe isn't a great idea.

While transmute does very little checking, it will at least make sure that the types are the same size. This errors:

```rust
use std::mem;

unsafe {
  let a = [0u8, 0u8, 0u8, 0u8];

  let b = mem::transmute::<[u8; 4], u64>(a);
}
```

with:

```
error: transmute called on types with different sizes: [u8; 4] (32 bits) to u64
(64 bits)
```

Other than that, you're on your own!

5.31 Associated Types 事务

Associated types are a powerful part of Rust's type system. They're related to the idea of a 'type family', in other words, grouping multiple types together. That description is a bit abstract, so

let's dive right into an example. If you want to write a `Graph` trait, you have two types to be generic over: the node type and the edge type. So you might write a trait, `Graph<N, E>`, that looks like this:

```
trait Graph<N, E> {
  fn has_edge(&self, &N, &N) -> bool;
  fn edges(&self, &N) -> Vec<E>;
  // etc
}
```

While this sort of works, it ends up being awkward. For example, any function that wants to take a `Graph` as a parameter now *also* needs to be generic over the `Node` and `Edge` types too:

```
fn distance<N, E, G: Graph<N, E>>(graph: &G, start: &N, end: &N) -> u32 { ... }
```

Our distance calculation works regardless of our `Edge` type, so the `E` stuff in this signature is just a distraction.

What we really want to say is that a certain `Edge` and `Node` type come together to form each kind of `Graph`. We can do that with associated types:

```
trait Graph {
  type N;
  type E;

  fn has_edge(&self, &Self::N, &Self::N) -> bool;
  fn edges(&self, &Self::N) -> Vec<Self::E>;
  // etc
}
```

Now, our clients can be abstract over a given `Graph`:

```
fn distance<G: Graph>(graph: &G, start: &G::N, end: &G::N) -> uint { ... }
```

No need to deal with the `Edge` type here!

Let's go over all this in more detail.

Defining associated types

Let's build that `Graph` trait. Here's the definition:

```
trait Graph {
  type N;
  type E;
```

```
  fn has_edge(&self, &Self::N, &Self::N) -> bool;
  fn edges(&self, &Self::N) -> Vec<Self::E>;
}
```

Simple enough. Associated types use the `type` keyword, and go inside the body of the trait, with the functions.

These `type` declarations can have all the same thing as functions do. For example, if we wanted our `N` type to implement `Display`, so we can print the nodes out, we could do this:

```
use std::fmt;

trait Graph {
  type N: fmt::Display;
  type E;

  fn has_edge(&self, &Self::N, &Self::N) -> bool;
  fn edges(&self, &Self::N) -> Vec<Self::E>;
}
```

Implementing associated types

Just like any trait, traits that use associated types use the `impl` keyword to provide implementations. Here's a simple implementation of Graph:

```
struct Node;

struct Edge;

struct MyGraph;

impl Graph for MyGraph {
  type N = Node;
  type E = Edge;

  fn has_edge(&self, n1: &Node, n2: &Node) -> bool {
    true
  }

  fn edges(&self, n: &Node) -> Vec<Edge> {
    Vec::new()
  }
}
```

This silly implementation always returns `true` and an empty `Vec<Edge>`, but it gives you an idea of how to implement this kind of thing. We first need three `struct`s, one for the graph, one for the

node, and one for the edge. If it made more sense to use a different type, that would work as well, we're just going to use structs for all three here.

Next is the impl line, which is just like implementing any other trait.

From here, we use = to define our associated types. The name the trait uses goes on the left of the =, and the concrete type we're implementing this for goes on the right. Finally, we use the concrete types in our function declarations.

Trait objects with associated types

There's one more bit of syntax we should talk about: trait objects. If you try to create a trait object from an associated type, like this:

```
let graph = MyGraph;
let obj = Box::new(graph) as Box<Graph>;
```

You'll get two errors:

```
error: the value of the associated type `E` (from the trait `main::Graph`) must
be specified [E0191]
let obj = Box::new(graph) as Box<Graph>;
          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
24:44 error: the value of the associated type `N` (from the trait
`main::Graph`) must be specified [E0191]
let obj = Box::new(graph) as Box<Graph>;
          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

We can't create a trait object like this, because we don't know the associated types. Instead, we can write this:

```
let graph = MyGraph;
let obj = Box::new(graph) as Box<Graph<N=Node, E=Edge>>;
```

The N=Node syntax allows us to provide a concrete type, Node, for the N type parameter. Same with E=Edge. If we didn't provide this constraint, we couldn't be sure which impl to match this trait object to.

5.32   Unsized Types 动态大小类型

Most types have a particular size, in bytes, that is knowable at compile time. For example, an i32 is thirty-two bits big, or four bytes. However, there are some types which are useful to express, but do not have a defined size. These are called 'unsized' or 'dynamically sized' types. One example

is [T]. This type represents a certain number of T in sequence. But we don't know how many there are, so the size is not known.

Rust understands a few of these types, but they have some restrictions. There are three:

1. We can only manipulate an instance of an unsized type via a pointer. An &[T] works just fine, but a [T] does not.
2. Variables and arguments cannot have dynamically sized types.
3. Only the last field in a struct may have a dynamically sized type; the other fields must not. Enum variants must not have dynamically sized types as data.

So why bother? Well, because [T] can only be used behind a pointer, if we didn't have language support for unsized types, it would be impossible to write this:

```
impl Foo for str {
```

or

```
impl<T> Foo for [T] {
```

Instead, you would have to write:

```
impl Foo for &str {
```

Meaning, this implementation would only work for references (section 5.9, page 170), and not other types of pointers. With the impl for str, all pointers, including (at some point, there are some bugs to fix first) user-defined custom smart pointers, can use this impl.

5.32.1   ?Sized

If you want to write a function that accepts a dynamically sized type, you can use the special bound, ?Sized:

```
struct Foo<T: ?Sized> {
  f: T,
}
```

This ?, read as "T may be Sized", means that this bound is special: it lets us match more kinds, not less. It's almost like every T implicitly has T: Sized, and the ? undoes this default.

5.33   Operators and Overloading 运行和重载

Rust allows for a limited form of operator overloading. There are certain operators that are able to be overloaded. To support a particular operator between types, there's a specific trait that you can implement, which then overloads the operator.

For example, the + operator can be overloaded with the Add trait:

```rust
use std::ops::Add;

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

impl Add for Point {
  type Output = Point;

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

fn main() {
  let p1 = Point { x: 1, y: 0 };
  let p2 = Point { x: 2, y: 3 };

  let p3 = p1 + p2;

  println!("{:?}", p3);
}
```

In main, we can use + on our two Points, since we've implemented Add<Output=Point> for Point.

There are a number of operators that can be overloaded this way, and all of their associated traits live in the std::ops[41] module. Check out its documentation for the full list.

Implementing these traits follows a pattern. Let's look at Add[42] in more detail:

```rust
pub trait Add<RHS = Self> {
  type Output;

  fn add(self, rhs: RHS) -> Self::Output;
}
```

---

[41] http://doc.rust-lang.org/std/ops/index.html
[42] http://doc.rust-lang.org/std/ops/trait.Add.html

There's three types in total involved here: the type you impl Add for, RHS, which defaults to Self, and Output. For an expression let z = x + y, x is the Self type, y is the RHS, and z is the Self::Output type.

```rust
impl Add<i32> for Point {
    type Output = f64;

    fn add(self, rhs: i32) -> f64 {
        // add an i32 to a Point and get an f64
    }
}
```

will let you do this:

```rust
let p: Point = // ...
let x: f64 = p + 2i32;
```

5.34 Deref coercions 强制 Deref

The standard library provides a special trait, Deref[43]. It's normally used to overload *, the deference operator:

```rust
use std::ops::Deref;

struct DerefExample<T> {
    value: T,
}

impl<T> Deref for DerefExample<T> {
    type Target = T;

    fn deref(&self) -> &T {
        &self.value
    }
}

fn main() {
    let x = DerefExample { value: 'a' };
    assert_eq!('a', *x);
}
```

This is useful for writing custom pointer types. However, there's a language feature related to Deref: 'deref coercions'. Here's the rule: If you have a type U, and it implements Deref<Target=T>, values of &U will automatically coerce to a &T. Here's an example:

---

[43]http://doc.rust-lang.org/std/ops/trait.Deref.html

```rust
fn foo(s: &str) {
    // borrow a string for a second
}

// String implements Deref<Target=str>
let owned = "Hello".to_string();

// therefore, this works:
foo(&owned);
```

Using an ampersand in front of a value takes a reference to it. So `owned` is a `String`, `&owned` is an `&String`, and since `impl Deref<Target=str> for String`, `&String` will deref to `&str`, which `foo()` takes.

That's it. This rule is one of the only places in which Rust does an automatic conversion for you, but it adds a lot of flexibility. For example, the `Rc<T>` type implements `Deref<Target=T>`, so this works:

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

fn foo(s: &str) {
    // borrow a string for a second
}

// String implements Deref<Target=str>
let owned = "Hello".to_string();
let counted = Rc::new(owned);

// therefore, this works:
foo(&counted);
```

All we've done is wrap our `String` in an `Rc<T>`. But we can now pass the `Rc<String>` around anywhere we'd have a `String`. The signature of `foo` didn't change, but works just as well with either type. This example has two conversions: `Rc<String>` to `String` and then `String` to `&str`. Rust will do this as many times as possible until the types match.

Another very common implementation provided by the standard library is:

```rust
fn foo(s: &[i32]) {
    // borrow a slice for a second
}

// Vec<T> implements Deref<Target=[T]>
let owned = vec![1, 2, 3];

foo(&owned);
```

Vectors can `Deref` to a slice.

Deref and method calls

Deref will also kick in when calling a method. In other words, these are the same two things in Rust:

```rust
struct Foo;

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

let f = Foo;

f.foo();
```

Even though f isn't a reference, and foo takes &self, this works. That's because these things are the same:

```rust
f.foo();
(&f).foo();
(&&f).foo();
(&&&&&&&&f).foo();
```

A value of type &&&&&&&&&&&&&&&Foo can still have methods defined on Foo called, because the compiler will insert as many * operations as necessary to get it right. And since it's inserting *s, that uses Deref.

5.35   Macros 宏

By now you've learned about many of the tools Rust provides for abstracting and reusing code. These units of code reuse have a rich semantic structure. For example, functions have a type signature, type parameters have trait bounds, and overloaded functions must belong to a particular trait.

This structure means that Rust's core abstractions have powerful compile-time correctness checking. But this comes at the price of reduced flexibility. If you visually identify a pattern of repeated code, you may find it's difficult or cumbersome to express that pattern as a generic function, a trait, or anything else within Rust's semantics.

Macros allow us to abstract at a syntactic level. A macro invocation is shorthand for an "expanded" syntactic form. This expansion happens early in compilation, before any static checking. As a result, macros can capture many patterns of code reuse that Rust's core abstractions cannot.

The drawback is that macro-based code can be harder to understand, because fewer of the built-in rules apply. Like an ordinary function, a well-behaved macro can be used without understanding its implementation. However, it can be difficult to design a well-behaved macro! Additionally, compiler

errors in macro code are harder to interpret, because they describe problems in the expanded code, not the source-level form that developers use.

These drawbacks make macros something of a "feature of last resort". That's not to say that macros are bad; they are part of Rust because sometimes they're needed for truly concise, well-abstracted code. Just keep this tradeoff in mind.

5.35.1   Defining a macro

You may have seen the vec! macro, used to initialize a vector (section 5.17, page 202) with any number of elements.

```rust
let x: Vec<u32> = vec![1, 2, 3];
```

This can't be an ordinary function, because it takes any number of arguments. But we can imagine it as syntactic shorthand for

```rust
let x: Vec<u32> = {
  let mut temp_vec = Vec::new();
  temp_vec.push(1);
  temp_vec.push(2);
  temp_vec.push(3);
  temp_vec
};
```

We can implement this shorthand, using a macro: [44]

```rust
macro_rules! vec {
  ( $( $x:expr ),* ) => {
    {
      let mut temp_vec = Vec::new();
      $(
        temp_vec.push($x);
      )*
      temp_vec
    }
  };
}
```

Whoa, that's a lot of new syntax! Let's break it down.

```rust
macro_rules! vec { ... }
```

---

[44]The actual definition of vec! in libcollections differs from the one presented here, for reasons of efficiency and reusability.

This says we're defining a macro named vec, much as fn vec would define a function named vec. In prose, we informally write a macro's name with an exclamation point, e.g. vec!. The exclamation point is part of the invocation syntax and serves to distinguish a macro from an ordinary function.

Matching

The macro is defined through a series of rules, which are pattern-matching cases. Above, we had

```
( $( $x:expr ),* ) => { ... };
```

This is like a match expression arm, but the matching happens on Rust syntax trees, at compile time. The semicolon is optional on the last (here, only) case. The "pattern" on the left-hand side of => is known as a 'matcher'. These have their own little grammar[45] within the language.

The matcher $x:expr will match any Rust expression, binding that syntax tree to the 'metavariable' $x. The identifier expr is a 'fragment specifier'; the full possibilities are enumerated later in this chapter. Surrounding the matcher with $(...),* will match zero or more expressions, separated by commas.

Aside from the special matcher syntax, any Rust tokens that appear in a matcher must match exactly. For example,

```
macro_rules! foo {
    (x => $e:expr) => (println!("mode X: {}", $e));
    (y => $e:expr) => (println!("mode Y: {}", $e));
}

fn main() {
    foo!(y => 3);
}
```

will print

```
mode Y: 3
```

With

```
foo!(z => 3);
```

we get the compiler error

```
error: no rules expected the token `z`
```

---

[45]http://doc.rust-lang.org/reference.html#macros

Expansion

The right-hand side of a macro rule is ordinary Rust syntax, for the most part. But we can splice in bits of syntax captured by the matcher. From the original example:

```
$(
  temp_vec.push($x);
)*
```

Each matched expression $x will produce a single push statement in the macro expansion. The repetition in the expansion proceeds in "lockstep" with repetition in the matcher (more on this in a moment).

Because $x was already declared as matching an expression, we don't repeat :expr on the right-hand side. Also, we don't include a separating comma as part of the repetition operator. Instead, we have a terminating semicolon within the repeated block.

Another detail: the vec! macro has *two* pairs of braces on the right-hand side. They are often combined like so:

```
macro_rules! foo {
  () => {{
    ...
  }}
}
```

The outer braces are part of the syntax of macro_rules!. In fact, you can use () or [] instead. They simply delimit the right-hand side as a whole.

The inner braces are part of the expanded syntax. Remember, the vec! macro is used in an expression context. To write an expression with multiple statements, including let-bindings, we use a block. If your macro expands to a single expression, you don't need this extra layer of braces.

Note that we never *declared* that the macro produces an expression. In fact, this is not determined until we use the macro as an expression. With care, you can write a macro whose expansion works in several contexts. For example, shorthand for a data type could be valid as either an expression or a pattern.

Repetition

The repetition operator follows two principal rules:

1. $(...)* walks through one "layer" of repetitions, for all of the $names it contains, in lockstep, and
2. each $name must be under at least as many $(...)*s as it was matched against. If it is under more, it'll be duplicated, as appropriate.

This baroque macro illustrates the duplication of variables from outer repetition levels.

```rust
macro_rules! o_O {
  (
    $(
      $x:expr; [ $( $y:expr ),* ]
    );*
  ) => {
    &[ $($($( $x + $y ),*),*),* ]
  }
}

fn main() {
  let a: &[i32]
    = o_O!(10; [1, 2, 3];
        20; [4, 5, 6]);

  assert_eq!(a, [11, 12, 13, 24, 25, 26]);
}
```

That's most of the matcher syntax. These examples use $(...)*, which is a "zero or more" match. Alternatively you can write $(...)+ for a "one or more" match. Both forms optionally include a separator, which can be any token except + or *.

This system is based on "Macro-by-Example[46]" (PDF link).

5.35.2   Hygiene

Some languages implement macros using simple text substitution, which leads to various problems. For example, this C program prints 13 instead of the expected 25.

```c
#define FIVE_TIMES(x) 5 * x

int main() {
  printf("%d\n", FIVE_TIMES(2 + 3));
  return 0;
}
```

After expansion we have 5 * 2 + 3, and multiplication has greater precedence than addition. If you've used C macros a lot, you probably know the standard idioms for avoiding this problem, as well as five or six others. In Rust, we don't have to worry about it.

---

[46]http://www.cs.indiana.edu/ftp/techreports/TR206.pdf

```
macro_rules! five_times {
    ($x:expr) => (5 * $x);
}

fn main() {
    assert_eq!(25, five_times!(2 + 3));
}
```

The metavariable $x is parsed as a single expression node, and keeps its place in the syntax tree even after substitution.

Another common problem in macro systems is 'variable capture'. Here's a C macro, using a GNU C extension[47] to emulate Rust's expression blocks.

```
#define LOG(msg) ({ \
    int state = get_log_state(); \
    if (state > 0) { \
        printf("log(%d): %s\n", state, msg); \
    } \
})
```

Here's a simple use case that goes terribly wrong:

```
const char *state = "reticulating splines";
LOG(state)
```

This expands to

```
const char *state = "reticulating splines";
int state = get_log_state();
if (state > 0) {
    printf("log(%d): %s\n", state, state);
}
```

The second variable named state shadows the first one. This is a problem because the print statement should refer to both of them.

The equivalent Rust macro has the desired behavior.

```
macro_rules! log {
    ($msg:expr) => {{
        let state: i32 = get_log_state();
        if state > 0 {
```

---

[47]https://gcc.gnu.org/onlinedocs/gcc/Statement-Exprs.html

```
        println!("log({}): {}", state, $msg);
      }
  }};
}

fn main() {
  let state: &str = "reticulating splines";
  log!(state);
}
```

This works because Rust has a hygienic macro system[48]. Each macro expansion happens in a distinct 'syntax context', and each variable is tagged with the syntax context where it was introduced. It's as though the variable state inside main is painted a different "color" from the variable state inside the macro, and therefore they don't conflict.

This also restricts the ability of macros to introduce new bindings at the invocation site. Code such as the following will not work:

```
macro_rules! foo {
  () => (let x = 3);
}

fn main() {
  foo!();
  println!("{}", x);
}
```

Instead you need to pass the variable name into the invocation, so it's tagged with the right syntax context.

```
macro_rules! foo {
  ($v:ident) => (let $v = 3);
}

fn main() {
  foo!(x);
  println!("{}", x);
}
```

This holds for let bindings and loop labels, but not for items[49]. So the following code does compile:

---

[48]http://en.wikipedia.org/wiki/Hygienic_macro
[49]http://doc.rust-lang.org/reference.html#items

```
macro_rules! foo {
  () => (fn x() { });
}

fn main() {
  foo!();
  x();
}
```

5.35.3 Recursive macros

A macro's expansion can include more macro invocations, including invocations of the very same macro being expanded. These recursive macros are useful for processing tree-structured input, as illustrated by this (simplistic) HTML shorthand:

```
macro_rules! write_html {
  ($w:expr, ) => (());

  ($w:expr, $e:tt) => (write!($w, "{}", $e));

  ($w:expr, $tag:ident [ $($inner:tt)* ] $($rest:tt)*) => {{
    write!($w, "<{}>", stringify!($tag));
    write_html!($w, $($inner)*);
    write!($w, "</{}>", stringify!($tag));
    write_html!($w, $($rest)*);
  }};
}

fn main() {
  use std::fmt::Write;
  let mut out = String::new();

  write_html!(&mut out,
    html[
      head[title["Macros guide"]]
      body[h1["Macros are the best!"]]
    ]);

  assert_eq!(out,
    "<html><head><title>Macros guide</title></head>\
     <body><h1>Macros are the best!</h1></body></html>");
}
```

5.35.4   Debugging macro code

To see the results of expanding macros, run `rustc --pretty expanded`. The output represents a whole crate, so you can also feed it back in to `rustc`, which will sometimes produce better error messages than the original compilation. Note that the `--pretty expanded` output may have a different meaning if multiple variables of the same name (but different syntax contexts) are in play in the same scope. In this case `--pretty expanded,hygiene` will tell you about the syntax contexts.

`rustc` provides two syntax extensions that help with macro debugging. For now, they are unstable and require feature gates.

- `log_syntax!(...)` will print its arguments to standard output, at compile time, and "expand" to nothing.

- `trace_macros!(true)` will enable a compiler message every time a macro is expanded. Use `trace_macros!(false)` later in expansion to turn it off.

5.35.5   Syntactic requirements

Even when Rust code contains un-expanded macros, it can be parsed as a full syntax tree[50]. This property can be very useful for editors and other tools that process code. It also has a few consequences for the design of Rust's macro system.

One consequence is that Rust must determine, when it parses a macro invocation, whether the macro stands in for

- zero or more items,
- zero or more methods,
- an expression,
- a statement, or
- a pattern.

A macro invocation within a block could stand for some items, or for an expression / statement. Rust uses a simple rule to resolve this ambiguity. A macro invocation that stands for items must be either

- delimited by curly braces, e.g. `foo! { ... }`, or
- terminated by a semicolon, e.g. `foo!(...);`

Another consequence of pre-expansion parsing is that the macro invocation must consist of valid Rust tokens. Furthermore, parentheses, brackets, and braces must be balanced within a macro invocation. For example, `foo!([)` is forbidden. This allows Rust to know where the macro invocation ends.

More formally, the macro invocation body must be a sequence of 'token trees'. A token tree is defined recursively as either

---

[50]glossary.html#abstract-syntax-tree

- a sequence of token trees surrounded by matching (), [], or {}, or
- any other single token.

Within a matcher, each metavariable has a 'fragment specifier', identifying which syntactic form it matches.

- ident: an identifier. Examples: x; foo.
- path: a qualified name. Example: T::SpecialA.
- expr: an expression. Examples: 2 + 2; if true then { 1 } else { 2 }; f(42).
- ty: a type. Examples: i32; Vec<(char, String)>; &T.
- pat: a pattern. Examples: Some(t); (17, 'a'); _.
- stmt: a single statement. Example: let x = 3.
- block: a brace-delimited sequence of statements. Example: { log(error, "hi"); return 12; }.
- item: an item[51]. Examples: fn foo() { }; struct Bar;.
- meta: a "meta item", as found in attributes. Example: cfg(target_os = "windows").
- tt: a single token tree.

There are additional rules regarding the next token after a metavariable:

- expr variables must be followed by one of: => , ;
- ty and path variables must be followed by one of: => , : => as
- pat variables must be followed by one of: => , =
- Other variables may be followed by any token.

These rules provide some flexibility for Rust's syntax to evolve without breaking existing macros.

The macro system does not deal with parse ambiguity at all. For example, the grammar $($t:ty)* $e:expr will always fail to parse, because the parser would be forced to choose between parsing $t and parsing $e. Changing the invocation syntax to put a distinctive token in front can solve the problem. In this case, you can write $(T $t:ty)* E $e:exp.

5.35.6 Scoping and macro import/export

Macros are expanded at an early stage in compilation, before name resolution. One downside is that scoping works differently for macros, compared to other constructs in the language.

Definition and expansion of macros both happen in a single depth-first, lexical-order traversal of a crate's source. So a macro defined at module scope is visible to any subsequent code in the same module, which includes the body of any subsequent child mod items.

A macro defined within the body of a single fn, or anywhere else not at module scope, is visible only within that item.

---

[51]http://doc.rust-lang.org/reference.html#items

If a module has the macro_use attribute, its macros are also visible in its parent module after the child's mod item. If the parent also has macro_use then the macros will be visible in the grandparent after the parent's mod item, and so forth.

The macro_use attribute can also appear on extern crate. In this context it controls which macros are loaded from the external crate, e.g.

```
#[macro_use(foo, bar)]
extern crate baz;
```

If the attribute is given simply as #[macro_use], all macros are loaded. If there is no #[macro_use] attribute then no macros are loaded. Only macros defined with the #[macro_export] attribute may be loaded.

To load a crate's macros without linking it into the output, use #[no_link] as well.

An example:

```
macro_rules! m1 { () => (()) }

// visible here: m1

mod foo {
  // visible here: m1

  #[macro_export]
  macro_rules! m2 { () => (()) }

  // visible here: m1, m2
}

// visible here: m1

macro_rules! m3 { () => (()) }

// visible here: m1, m3

#[macro_use]
mod bar {
  // visible here: m1, m3

  macro_rules! m4 { () => (()) }

  // visible here: m1, m3, m4
}

// visible here: m1, m3, m4
```

When this library is loaded with #[macro_use] extern crate, only m2 will be imported.

The Rust Reference has a listing of macro-related attributes[52].

5.35.7 The variable **$crate**

A further difficulty occurs when a macro is used in multiple crates. Say that mylib defines

```
pub fn increment(x: u32) -> u32 {
  x + 1
}

#[macro_export]
macro_rules! inc_a {
  ($x:expr) => ( ::increment($x) )
}

#[macro_export]
macro_rules! inc_b {
  ($x:expr) => ( ::mylib::increment($x) )
}
```

inc_a only works within mylib, while inc_b only works outside the library. Furthermore, inc_b will break if the user imports mylib under another name.

Rust does not (yet) have a hygiene system for crate references, but it does provide a simple workaround for this problem. Within a macro imported from a crate named foo, the special macro variable $crate will expand to ::foo. By contrast, when a macro is defined and then used in the same crate, $crate will expand to nothing. This means we can write

```
#[macro_export]
macro_rules! inc {
  ($x:expr) => ( $crate::increment($x) )
}
```

to define a single macro that works both inside and outside our library. The function name will expand to either ::increment or ::mylib::increment.

To keep this system simple and correct, #[macro_use] extern crate ... may only appear at the root of your crate, not inside mod. This ensures that $crate is a single identifier.

5.35.8 The deep end

The introductory chapter mentioned recursive macros, but it did not give the full story. Recursive macros are useful for another reason: Each recursive invocation gives you another opportunity to pattern-match the macro's arguments.

---

[52]http://doc.rust-lang.org/reference.html#macro-related-attributes

As an extreme example, it is possible, though hardly advisable, to implement the Bitwise Cyclic Tag[53] automaton within Rust's macro system.

```rust
macro_rules! bct {
    // cmd 0:  d ... => ...
    (0, $($ps:tt),* ; $_d:tt)
        => (bct!($($ps),*, 0 ; ));
    (0, $($ps:tt),* ; $_d:tt, $($ds:tt),*)
        => (bct!($($ps),*, 0 ; $($ds),*));

    // cmd 1p:  1 ... => 1 ... p
    (1, $p:tt, $($ps:tt),* ; 1)
        => (bct!($($ps),*, 1, $p ; 1, $p));
    (1, $p:tt, $($ps:tt),* ; 1, $($ds:tt),*)
        => (bct!($($ps),*, 1, $p ; 1, $($ds),*, $p));

    // cmd 1p:  0 ... => 0 ...
    (1, $p:tt, $($ps:tt),* ; $($ds:tt),*)
        => (bct!($($ps),*, 1, $p ; $($ds),*));

    // halt on empty data string
    ( $($ps:tt),* ; )
        => (());
}
```

Exercise: use macros to reduce duplication in the above definition of the bct! macro.

5.35.9   Common macros

Here are some common macros you'll see in Rust code.

panic!

This macro causes the current thread to panic. You can give it a message to panic with:

```rust
panic!("oh no!");
```

vec!

The vec! macro is used throughout the book, so you've probably seen it already. It creates Vec<T>s with ease:

---

[53]http://esolangs.org/wiki/Bitwise_Cyclic_Tag

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

It also lets you make vectors with repeating values. For example, a hundred zeroes:

```rust
let v = vec![0; 100];
```

assert! and assert_eq!

These two macros are used in tests. assert! takes a boolean, and assert_eq! takes two values and compares them. Truth passes, success panic!s. Like this:

```rust
// A-ok!

assert!(true);
assert_eq!(5, 3 + 2);

// nope :(

assert!(5 < 3);
assert_eq!(5, 3);
```

try!

try! is used for error handling. It takes something that can return a Result<T, E>, and gives T if it's a Ok<T>, and returns with the Err(E) if it's that. Like this:

```rust
use std::fs::File;

fn foo() -> std::io::Result<()> {
  let f = try!(File::create("foo.txt"));

  Ok(())
}
```

This is cleaner than doing this:

```rust
use std::fs::File;

fn foo() -> std::io::Result<()> {
  let f = File::create("foo.txt");

  let f = match f {
    Ok(t) => t,
```

```
    Err(e) => return Err(e),
  };

  Ok(())
}
```

unreachable!

This macro is used when you think some code should never execute:

```
if false {
  unreachable!();
}
```

Sometimes, the compiler may make you have a different branch that you know will never, ever run. In these cases, use this macro, so that if you end up wrong, you'll get a panic! about it.

```
let x: Option<i32> = None;

match x {
  Some(_) => unreachable!(),
  None => println!("I know x is None!"),
}
```

unimplemented!

The unimplemented! macro can be used when you're trying to get your functions to typecheck, and don't want to worry about writing out the body of the function. One example of this situation is implementing a trait with multiple required methods, where you want to tackle one at a time. Define the others as unimplemented! until you're ready to write them.

5.35.10   Procedural macros

If Rust's macro system can't do what you need, you may want to write a compiler plugin (section 6.1, page 279) instead. Compared to macro_rules! macros, this is significantly more work, the interfaces are much less stable, and bugs can be much harder to track down. In exchange you get the flexibility of running arbitrary Rust code within the compiler. Syntax extension plugins are sometimes called 'procedural macros' for this reason.

5.36 Raw Pointers 原始指针

Rust has a number of different smart pointer types in its standard library, but there are two types that are extra-special. Much of Rust's safety comes from compile-time checks, but raw pointers don't have such guarantees, and are unsafe (section 5.37, page 273) to use.

`*const T` and `*mut T` are called 'raw pointers' in Rust. Sometimes, when writing certain kinds of libraries, you'll need to get around Rust's safety guarantees for some reason. In this case, you can use raw pointers to implement your library, while exposing a safe interface for your users. For example, `*` pointers are allowed to alias, allowing them to be used to write shared-ownership types, and even thread-safe shared memory types (the `Rc<T>` and `Arc<T>` types are both implemented entirely in Rust).

Here are some things to remember about raw pointers that are different than other pointer types. They:

- are not guaranteed to point to valid memory and are not even guaranteed to be non-null (unlike both `Box` and `&`);
- do not have any automatic clean-up, unlike `Box`, and so require manual resource management;
- are plain-old-data, that is, they don't move ownership, again unlike `Box`, hence the Rust compiler cannot protect against bugs like use-after-free;
- lack any form of lifetimes, unlike `&`, and so the compiler cannot reason about dangling pointers; and
- have no guarantees about aliasing or mutability other than mutation not being allowed directly through a `*const T`.

5.36.1 Basics

Creating a raw pointer is perfectly safe:

```rust
let x = 5;
let raw = &x as *const i32;

let mut y = 10;
let raw_mut = &mut y as *mut i32;
```

However, dereferencing one is not. This won't work:

```rust
let x = 5;
let raw = &x as *const i32;

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

It gives this error:

```
error: dereference of unsafe pointer requires unsafe function or block [E0133]
   println!("raw points at{}", *raw);
                   ^~~~
```

When you dereference a raw pointer, you're taking responsibility that it's not pointing somewhere that would be incorrect. As such, you need unsafe:

```rust
let x = 5;
let raw = &x as *const i32;

let points_at = unsafe { *raw };

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

For more operations on raw pointers, see their API documentation[54].

5.36.2   FFI

Raw pointers are useful for FFI: Rust's *const T and *mut T are similar to C's const T* and T*, respect-fully. For more about this use, consult the FFI chapter (section 4.8, page 128).

5.36.3   References and raw pointers

At runtime, a raw pointer * and a reference pointing to the same piece of data have an identical representation. In fact, an &T reference will implicitly coerce to an *const T raw pointer in safe code and similarly for the mut variants (both coercions can be performed explicitly with, respectively, value as *const T and value as *mut T).

Going the opposite direction, from *const to a reference &, is not safe. A &T is always valid, and so, at a minimum, the raw pointer *const T has to point to a valid instance of type T. Furthermore, the resulting pointer must satisfy the aliasing and mutability laws of references. The compiler assumes these properties are true for any references, no matter how they are created, and so any conversion from raw pointers is asserting that they hold. The programmer *must* guarantee this.

The recommended method for the conversion is

```rust
let i: u32 = 1;

// explicit cast
let p_imm: *const u32 = &i as *const u32;
let mut m: u32 = 2;

// implicit coercion
```

---

[54]http://doc.rust-lang.org/std/primitive.pointer.html

```
let p_mut: *mut u32 = &mut m;

unsafe {
  let ref_imm: &u32 = &*p_imm;
  let ref_mut: &mut u32 = &mut *p_mut;
}
```

The &*x dereferencing style is preferred to using a transmute. The latter is far more powerful than necessary, and the more restricted operation is harder to use incorrectly; for example, it requires that x is a pointer (unlike transmute).

5.37   unsafe 不安全块

Rust's main draw is its powerful static guarantees about behavior. But safety checks are conservative by nature: there are some programs that are actually safe, but the compiler is not able to verify this is true. To write these kinds of programs, we need to tell the compiler to relax its restrictions a bit. For this, Rust has a keyword, unsafe. Code using unsafe has less restrictions than normal code does.

Let's go over the syntax, and then we'll talk semantics. unsafe is used in two contexts. The first one is to mark a function as unsafe:

```
unsafe fn danger_will_robinson() {
  // scary stuff
}
```

All functions called from FFI (section 4.8, page 128) must be marked as unsafe, for example. The second use of unsafe is an unsafe block:

```
unsafe {
  // scary stuff
}
```

It's important to be able to explicitly delineate code that may have bugs that cause big problems. If a Rust program segfaults, you can be sure it's somewhere in the sections marked unsafe.

5.37.1   What does 'safe' mean?

Safe, in the context of Rust, means "doesn't do anything unsafe." Easy!

Okay, let's try again: what is not safe to do? Here's a list:

- Data races

- Dereferencing a null/dangling raw pointer
- Reads of undef[55] (uninitialized) memory
- Breaking the pointer aliasing rules[56] with raw pointers.
- `&mut T` and `&T` follow LLVM's scoped noalias[57] model, except if the `&T` contains an `UnsafeCell<U>`. Unsafe code must not violate these aliasing guarantees.
- Mutating an immutable value/reference without `UnsafeCell<U>`
- Invoking undefined behavior via compiler intrinsics:
- Indexing outside of the bounds of an object with `std::ptr::offset` (`offset` intrinsic), with the exception of one byte past the end which is permitted.
- Using `std::ptr::copy_nonoverlapping_memory` (`memcpy32`/`memcpy64` intrinsics) on overlapping buffers
- Invalid values in primitive types, even in private fields/locals:
- Null/dangling references or boxes
- A value other than `false` (0) or `true` (1) in a `bool`
- A discriminant in an `enum` not included in its type definition
- A value in a `char` which is a surrogate or above `char::MAX`
- Non-UTF-8 byte sequences in a `str`
- Unwinding into Rust from foreign code or unwinding from Rust into foreign code.

Whew! That's a bunch of stuff. It's also important to notice all kinds of behaviors that are certainly bad, but are expressly *not* unsafe:

- Deadlocks
- Reading data from private fields
- Leaks due to reference count cycles
- Exiting without calling destructors
- Sending signals
- Accessing/modifying the file system
- Integer overflow

Rust cannot prevent all kinds of software problems. Buggy code can and will be written in Rust. These things aren't great, but they don't qualify as `unsafe` specifically.

5.37.2   Unsafe Superpowers

In both unsafe functions and unsafe blocks, Rust will let you do three things that you normally can not do. Just three. Here they are:

1. Access or update a static mutable variable[58].

---

[55] http://llvm.org/docs/LangRef.html#undefined-values
[56] http://llvm.org/docs/LangRef.html#pointer-aliasing-rules
[57] http://llvm.org/docs/LangRef.html#noalias
[58] const-and-static.html#static

2. Dereference a raw pointer.
3. Call unsafe functions. This is the most powerful ability.

That's it. It's important that unsafe does not, for example, 'turn off the borrow checker'. Adding unsafe to some random Rust code doesn't change its semantics, it won't just start accepting anything.

But it will let you write things that *do* break some of the rules. Let's go over these three abilities in order.

Access or update a `static mut`

Rust has a feature called 'static mut' which allows for mutable global state. Doing so can cause a data race, and as such is inherently not safe. For more details, see the static[59] section of the book.

Dereference a raw pointer

Raw pointers let you do arbitrary pointer arithmetic, and can cause a number of different memory safety and security issues. In some senses, the ability to dereference an arbitrary pointer is one of the most dangerous things you can do. For more on raw pointers, see their section of the book (section 5.36, page 271).

Call unsafe functions

This last ability works with both aspects of unsafe: you can only call functions marked unsafe from inside an unsafe block.

This ability is powerful and varied. Rust exposes some compiler intrinsics (section 6.4, page 290) as unsafe functions, and some unsafe functions bypass safety checks, trading safety for speed.

I'll repeat again: even though you *can* do arbitrary things in unsafe blocks and functions doesn't mean you should. The compiler will act as though you're upholding its invariants, so be careful!

---

[59]const-and-static.html#static

6

Nightly Rust 每日 Rust 版本

Rust provides three distribution channels for Rust: nightly, beta, and stable. Unstable features are only available on nightly Rust. For more details on this process, see 'Stability as a deliverable[1]'.

To install nightly Rust, you can use rustup.sh:

```
$ curl -s https://static.rust-lang.org/rustup.sh | sh -s -- --channel=nightly
```

If you're concerned about the potential insecurity[2] of using curl|sh, please keep reading and see our disclaimer below. And feel free to use a two-step version of the installation and examine our installation script:

```
$ curl -f -L https://static.rust-lang.org/rustup.sh -O
$ sh rustup.sh --channel=nightly
```

If you're on Windows, please download either the 32-bit installer[3] or the 64-bit installer[4] and run it.

Uninstalling

If you decide you don't want Rust anymore, we'll be a bit sad, but that's okay. Not every programming language is great for everyone. Just run the uninstall script:

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

---

[1]http://blog.rust-lang.org/2014/10/30/Stability.html
[2]http://curlpipesh.tumblr.com
[3]https://static.rust-lang.org/dist/rust-1.0.0-beta-i686-pc-windows-gnu.msi
[4]https://static.rust-lang.org/dist/rust-1.0.0-beta-x86__64-pc-windows-gnu.msi

If you used the Windows installer, just re-run the .msi and it will give you an uninstall option.

Some people, and somewhat rightfully so, get very upset when we tell you to curl|sh. Basically, when you do this, you are trusting that the good people who maintain Rust aren't going to hack your computer and do bad things. That's a good instinct! If you're one of those people, please check out the documentation on building Rust from Source[5], or the official binary downloads[6].

Oh, we should also mention the officially supported platforms:

- Windows (7, 8, Server 2008 R2)
- Linux (2.6.18 or later, various distributions), x86 and x86-64
- OSX 10.7 (Lion) or greater, x86 and x86-64

We extensively test Rust on these platforms, and a few others, too, like Android. But these are the ones most likely to work, as they have the most testing.

Finally, a comment about Windows. Rust considers Windows to be a first-class platform upon release, but if we're honest, the Windows experience isn't as integrated as the Linux/OS X experience is. We're working on it! If anything does not work, it is a bug. Please let us know if that happens. Each and every commit is tested against Windows just like any other platform.

If you've got Rust installed, you can open up a shell, and type this:

$ rustc --version

You should see the version number, commit hash, commit date and build date:

rustc 1.0.0-nightly (f11f3e7ba 2015-01-04) (built 2015-01-06)

If you did, Rust has been installed successfully! Congrats!

This installer also installs a copy of the documentation locally, so you can read it offline. On UNIX systems, /usr/local/share/doc/rust is the location. On Windows, it's in a share/doc directory, inside wherever you installed Rust to.

If not, there are a number of places where you can get help. The easiest is the #rust IRC channel on irc.mozilla.org[7], which you can access through Mibbit[8]. Click that link, and you'll be chatting with other Rustaceans (a silly nickname we call ourselves), and we can help you out. Other great resources include the user's forum[9], and Stack Overflow[10].

---

[5]https://github.com/rust-lang/rust#building-from-source
[6]http://www.rust-lang.org/install.html
[7]irc://irc.mozilla.org/#rust
[8]http://chat.mibbit.com/?server=irc.mozilla.org&channel=%23rust
[9]http://users.rust-lang.org/
[10]http://stackoverflow.com/questions/tagged/rust

6.1 Compiler Plugins 编译插件

6.1.1 Introduction

rustc can load compiler plugins, which are user-provided libraries that extend the compiler's behavior with new syntax extensions, lint checks, etc.

A plugin is a dynamic library crate with a designated *registrar* function that registers extensions with rustc. Other crates can load these extensions using the crate attribute #![plugin(...)]. See the rustc::plugin[11] documentation for more about the mechanics of defining and loading a plugin.

If present, arguments passed as #![plugin(foo(... args ...))] are not interpreted by rustc itself. They are provided to the plugin through the Registry's args method[12].

In the vast majority of cases, a plugin should *only* be used through #![plugin] and not through an extern crate item. Linking a plugin would pull in all of libsyntax and librustc as dependencies of your crate. This is generally unwanted unless you are building another plugin. The plugin_as_library lint checks these guidelines.

The usual practice is to put compiler plugins in their own crate, separate from any macro_rules! macros or ordinary Rust code meant to be used by consumers of a library.

6.1.2 Syntax extensions

Plugins can extend Rust's syntax in various ways. One kind of syntax extension is the procedural macro. These are invoked the same way as ordinary macros (section 5.35, page 256), but the expansion is performed by arbitrary Rust code that manipulates syntax trees[13] at compile time.

Let's write a plugin roman_numerals.rs[14] that implements Roman numeral integer literals.

```
#![crate_type="dylib"]
#![feature(plugin_registrar, rustc_private)]

extern crate syntax;
extern crate rustc;

use syntax::codemap::Span;
use syntax::parse::token;
use syntax::ast::{TokenTree, TtToken};
use syntax::ext::base::{ExtCtxt, MacResult, DummyResult, MacEager};
use syntax::ext::build::AstBuilder;  // trait for expr_usize
use rustc::plugin::Registry;
```

---

[11]http://doc.rust-lang.org/rustc/plugin/index.html
[12]http://doc.rust-lang.org/rustc/plugin/registry/struct.Registry.html#method.args
[13]http://doc.rust-lang.org/syntax/ast/index.html
[14]https://github.com/rust-lang/rust/tree/master/src/test/auxiliary/roman_numerals.rs

```
fn expand_rn(cx: &mut ExtCtxt, sp: Span, args: &[TokenTree])
    -> Box<MacResult + 'static> {

  static NUMERALS: &'static [(&'static str, u32)] = &[
    ("M", 1000), ("CM", 900), ("D", 500), ("CD", 400),
    ("C",  100), ("XC",  90), ("L",  50), ("XL",  40),
    ("X",   10), ("IX",   9), ("V",   5), ("IV",   4),
    ("I",    1)];

  let text = match args {
    [TtToken(_, token::Ident(s, _))] => token::get_ident(s).to_string(),
    _ => {
      cx.span_err(sp, "argument should be a single identifier");
      return DummyResult::any(sp);
    }
  };

  let mut text = &*text;
  let mut total = 0;
  while !text.is_empty() {
    match NUMERALS.iter().find(|&&(rn, _)| text.starts_with(rn)) {
      Some(&(rn, val)) => {
        total += val;
        text = &text[rn.len()..];
      }
      None => {
        cx.span_err(sp, "invalid Roman numeral");
        return DummyResult::any(sp);
      }
    }
  }

  MacEager::expr(cx.expr_u32(sp, total))
}

#[plugin_registrar]
pub fn plugin_registrar(reg: &mut Registry) {
  reg.register_macro("rn", expand_rn);
}
```

Then we can use rn!() like any other macro:

```
#![feature(plugin)]
#![plugin(roman_numerals)]

fn main() {
```

```
  assert_eq!(rn!(MMXV), 2015);
}
```

The advantages over a simple fn(&str) -> u32 are:

- The (arbitrarily complex) conversion is done at compile time.
- Input validation is also performed at compile time.
- It can be extended to allow use in patterns, which effectively gives a way to define new literal syntax for any data type.

In addition to procedural macros, you can define new derive[15]-like attributes and other kinds of extensions. See Registry::register_syntax_extension[16] and the SyntaxExtension enum[17]. For a more involved macro example, see regex_macros[18].

Tips and tricks

Some of the macro debugging tips[19] are applicable.

You can use syntax::parse[20] to turn token trees into higher-level syntax elements like expressions:

```
fn expand_foo(cx: &mut ExtCtxt, sp: Span, args: &[TokenTree])
    -> Box<MacResult+'static> {

  let mut parser = cx.new_parser_from_tts(args);

  let expr: P<Expr> = parser.parse_expr();
```

Looking through libsyntax parser code[21] will give you a feel for how the parsing infrastructure works.

Keep the Spans[22] of everything you parse, for better error reporting. You can wrap Spanned[23] around your custom data structures.

Calling ExtCtxt::span_fatal[24] will immediately abort compilation. It's better to instead call ExtCtxt::span_err[25] and return DummyResult[26], so that the compiler can continue and find further errors.

---

[15]http://doc.rust-lang.org/reference.html#derive
[16]http://doc.rust-lang.org/rustc/plugin/registry/struct.Registry.html#method.register_syntax_extension
[17]http://doc.rust-lang.org/syntax/ext/base/enum.SyntaxExtension.html
[18]https://github.com/rust-lang/regex/blob/master/regex_macros/src/lib.rs
[19]macros.html#debugging-macro-code
[20]http://doc.rust-lang.org/syntax/parse/index.html
[21]https://github.com/rust-lang/rust/blob/master/src/libsyntax/parse/parser.rs
[22]http://doc.rust-lang.org/syntax/codemap/struct.Span.html
[23]http://doc.rust-lang.org/syntax/codemap/struct.Spanned.html
[24]http://doc.rust-lang.org/syntax/ext/base/struct.ExtCtxt.html#method.span_fatal
[25]http://doc.rust-lang.org/syntax/ext/base/struct.ExtCtxt.html#method.span_err
[26]http://doc.rust-lang.org/syntax/ext/base/struct.DummyResult.html

To print syntax fragments for debugging, you can use span_note[27] together with syntax::print::pprust::*_-to_string[28].

The example above produced an integer literal using AstBuilder::expr_usize[29]. As an alternative to the AstBuilder trait, libsyntax provides a set of quasiquote macros[30]. They are undocumented and very rough around the edges. However, the implementation may be a good starting point for an improved quasiquote as an ordinary plugin library.

6.1.3   Lint plugins

Plugins can extend Rust's lint infrastructure[31] with additional checks for code style, safety, etc. You can see src/test/auxiliary/lint_plugin_test.rs[32] for a full example, the core of which is reproduced here:

```
declare_lint!(TEST_LINT, Warn,
       "Warn about items named 'lintme'");

struct Pass;

impl LintPass for Pass {
  fn get_lints(&self) -> LintArray {
    lint_array!(TEST_LINT)
  }

  fn check_item(&mut self, cx: &Context, it: &ast::Item) {
    let name = token::get_ident(it.ident);
    if name.get() == "lintme" {
      cx.span_lint(TEST_LINT, it.span, "item is named 'lintme'");
    }
  }
}

#[plugin_registrar]
pub fn plugin_registrar(reg: &mut Registry) {
  reg.register_lint_pass(box Pass as LintPassObject);
}
```

Then code like

```
#![plugin(lint_plugin_test)]
```

---

[27] http://doc.rust-lang.org/syntax/ext/base/struct.ExtCtxt.html#method.span_note

[28] http://doc.rust-lang.org/syntax/print/pprust/index.html#functions

[29] http://doc.rust-lang.org/syntax/ext/build/trait.AstBuilder.html#tymethod.expr_usize

[30] http://doc.rust-lang.org/syntax/ext/quote/index.html

[31] http://doc.rust-lang.org/reference.html#lint-check-attributes

[32] https://github.com/rust-lang/rust/blob/master/src/test/auxiliary/lint_plugin_test.rs

```
fn lintme() { }
```

will produce a compiler warning:

```
foo.rs:4:1: 4:16 warning: item is named 'lintme', #[warn(test_lint)] on by default
foo.rs:4 fn lintme() { }
         ^~~~~~~~~~~~~~~
```

The components of a lint plugin are:

- one or more declare_lint! invocations, which define static Lint[33] structs;

- a struct holding any state needed by the lint pass (here, none);

- a LintPass[34] implementation defining how to check each syntax element. A single LintPass may call span_lint for several different Lints, but should register them all through the get_lints method.

Lint passes are syntax traversals, but they run at a late stage of compilation where type information is available. rustc's built-in lints[35] mostly use the same infrastructure as lint plugins, and provide examples of how to access type information.

Lints defined by plugins are controlled by the usual attributes and compiler flags[36], e.g. #[allow(test_lint)] or -A test-lint. These identifiers are derived from the first argument to declare_lint!, with appropriate case and punctuation conversion.

You can run rustc -W help foo.rs to see a list of lints known to rustc, including those provided by plugins loaded by foo.rs.

6.2 Inline Assembly 内联汇编

For extremely low-level manipulations and performance reasons, one might wish to control the CPU directly. Rust supports using inline assembly to do this via the asm! macro. The syntax roughly matches that of GCC & Clang:

```
asm!(assembly template
 : output operands
 : input operands
 : clobbers
 : options
 );
```

---

[33]http://doc.rust-lang.org/rustc/lint/struct.Lint.html
[34]http://doc.rust-lang.org/rustc/lint/trait.LintPass.html
[35]https://github.com/rust-lang/rust/blob/master/src/librustc/lint/builtin.rs
[36]http://doc.rust-lang.org/reference.html#lint-check-attributes

Any use of asm is feature gated (requires #![feature(asm)] on the crate to allow) and of course requires an unsafe block.

> **Note**: the examples here are given in x86/x86-64 assembly, but all platforms are supported.

Assembly template

The assembly template is the only required parameter and must be a literal string (i.e. "")

```
#![feature(asm)]

#[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
fn foo() {
  unsafe {
    asm!("NOP");
  }
}

// other platforms
#[cfg(not(any(target_arch = "x86", target_arch = "x86_64")))]
fn foo() { /* ... */ }

fn main() {
  // ...
  foo();
  // ...
}
```

(The feature(asm) and #[cfg]s are omitted from now on.)

Output operands, input operands, clobbers and options are all optional but you must add the right number of : if you skip them:

```
# #![feature(asm)]
# #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
# fn main() { unsafe {
asm!("xor %eax, %eax"
  :
  :
  : "{eax}"
  );
# }}
```

Whitespace also doesn't matter:

```
# #![feature(asm)]
# #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
# fn main() { unsafe {
asm!("xor %eax, %eax" ::: "{eax}");
# } }
```

Operands

Input and output operands follow the same format: : "constraints1"(expr1), "constraints2"(expr2), ...".
Output operand expressions must be mutable lvalues, or not yet assigned:

```
# #![feature(asm)]
# #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
fn add(a: i32, b: i32) -> i32 {
    let c: i32;
    unsafe {
        asm!("add $2, $0"
            : "=r"(c)
            : "0"(a), "r"(b)
            );
    }
    c
}
# #[cfg(not(any(target_arch = "x86", target_arch = "x86_64")))]
# fn add(a: i32, b: i32) -> i32 { a + b }

fn main() {
    assert_eq!(add(3, 14159), 14162)
}
```

If you would like to use real operands in this position, however, you are required to put curly braces
{} around the register that you want, and you are required to put the specific size of the operand.
This is useful for very low level programming, where which register you use is important:

```
# #![feature(asm)]
# #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
# unsafe fn read_byte_in(port: u16) -> u8 {
let result: u8;
asm!("in %dx, %al" : "={al}"(result) : "{dx}"(port));
result
# }
```

Clobbers

Some instructions modify registers which might otherwise have held different values so we use the
clobbers list to indicate to the compiler not to assume any values loaded into those registers will

stay valid.

```
# #![feature(asm)]
# #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
# fn main() { unsafe {
// Put the value 0x200 in eax
asm!("mov $$0x200, %eax" : /* no outputs */ : /* no inputs */ : "{eax}");
# } }
```

Input and output registers need not be listed since that information is already communicated by the given constraints. Otherwise, any other registers used either implicitly or explicitly should be listed.

If the assembly changes the condition code register cc should be specified as one of the clobbers. Similarly, if the assembly modifies memory, memory should also be specified.

Options

The last section, options is specific to Rust. The format is comma separated literal strings (i.e. :"foo", "bar", "baz"). It's used to specify some extra info about the inline assembly:

Current valid options are:

1. *volatile* - specifying this is analogous to __asm__ __volatile__ (...) in gcc/clang.
2. *alignstack* - certain instructions expect the stack to be aligned a certain way (i.e. SSE) and specifying this indicates to the compiler to insert its usual stack alignment code
3. *intel* - use intel syntax instead of the default AT&T.

```
# #![feature(asm)]
# #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
# fn main() {
let result: i32;
unsafe {
  asm!("mov eax, 2" : "={eax}"(result) : : : "intel")
}
println!("eax is currently {}", result);
# }
```

6.3   No stdlib 没有 std 库文件

By default, std is linked to every Rust crate. In some contexts, this is undesirable, and can be avoided with the #![no_std] attribute attached to the crate.

```
// a minimal library
#![crate_type="lib"]
#![feature(no_std)]
#![no_std]
# // fn main() {} tricked you, rustdoc!
```

Obviously there's more to life than just libraries: one can use #[no_std] with an executable, controlling the entry point is possible in two ways: the #[start] attribute, or overriding the default shim for the C main function with your own.

The function marked #[start] is passed the command line parameters in the same format as C:

```
#![feature(lang_items, start, no_std, libc)]
#![no_std]

// Pull in the system libc library for what crt0.o likely requires
extern crate libc;

// Entry point for this program
#[start]
fn start(_argc: isize, _argv: *const *const u8) -> isize {
    0
}

// These functions and traits are used by the compiler, but not
// for a bare-bones hello world. These are normally
// provided by libstd.
#[lang = "stack_exhausted"] extern fn stack_exhausted() {}
#[lang = "eh_personality"] extern fn eh_personality() {}
#[lang = "panic_fmt"] fn panic_fmt() -> ! { loop {} }
# // fn main() {} tricked you, rustdoc!
```

To override the compiler-inserted main shim, one has to disable it with #![no_main] and then create the appropriate symbol with the correct ABI and the correct name, which requires overriding the compiler's name mangling too:

```
#![feature(no_std)]
#![no_std]
#![no_main]
#![feature(lang_items, start)]

extern crate libc;

#[no_mangle] // ensure that this symbol is called `main` in the output
pub extern fn main(argc: i32, argv: *const *const u8) -> i32 {
    0
```

```
}
```

```
#[lang = "stack_exhausted"] extern fn stack_exhausted() {}
#[lang = "eh_personality"] extern fn eh_personality() {}
#[lang = "panic_fmt"] fn panic_fmt() -> ! { loop {} }
# // fn main() {} tricked you, rustdoc!
```

The compiler currently makes a few assumptions about symbols which are available in the executable to call. Normally these functions are provided by the standard library, but without it you must define your own.

The first of these three functions, stack_exhausted, is invoked whenever stack overflow is detected. This function has a number of restrictions about how it can be called and what it must do, but if the stack limit register is not being maintained then a thread always has an "infinite stack" and this function shouldn't get triggered.

The second of these three functions, eh_personality, is used by the failure mechanisms of the compiler. This is often mapped to GCC's personality function (see the libstd implementation[37] for more information), but crates which do not trigger a panic can be assured that this function is never called. The final function, panic_fmt, is also used by the failure mechanisms of the compiler.

Using libcore

> **Note**: the core library's structure is unstable, and it is recommended to use the standard
> library instead wherever possible.

With the above techniques, we've got a bare-metal executable running some Rust code. There is a good deal of functionality provided by the standard library, however, that is necessary to be productive in Rust. If the standard library is not sufficient, then libcore[38] is designed to be used instead.

The core library has very few dependencies and is much more portable than the standard library itself. Additionally, the core library has most of the necessary functionality for writing idiomatic and effective Rust code.

As an example, here is a program that will calculate the dot product of two vectors provided from C, using idiomatic Rust practices.

```
#![feature(lang_items, start, no_std, core, libc)]
#![no_std]

# extern crate libc;
extern crate core;
```

---

[37]http://doc.rust-lang.org/std/rt/unwind/index.html
[38]http://doc.rust-lang.org/core/index.html

```
use core::prelude::*;

use core::mem;

#[no_mangle]
pub extern fn dot_product(a: *const u32, a_len: u32,
              b: *const u32, b_len: u32) -> u32 {
  use core::raw::Slice;

  // Convert the provided arrays into Rust slices.
  // The core::raw module guarantees that the Slice
  // structure has the same memory layout as a &[T]
  // slice.
  //
  // This is an unsafe operation because the compiler
  // cannot tell the pointers are valid.
  let (a_slice, b_slice): (&[u32], &[u32]) = unsafe {
    mem::transmute((
      Slice { data: a, len: a_len as usize },
      Slice { data: b, len: b_len as usize },
    ))
  };

  // Iterate over the slices, collecting the result
  let mut ret = 0;
  for (i, j) in a_slice.iter().zip(b_slice.iter()) {
    ret += (*i) * (*j);
  }
  return ret;
}

#[lang = "panic_fmt"]
extern fn panic_fmt(args: &core::fmt::Arguments,
          file: &str,
          line: u32) -> ! {
  loop {}
}

#[lang = "stack_exhausted"] extern fn stack_exhausted() {}
#[lang = "eh_personality"] extern fn eh_personality() {}
# #[start] fn start(argc: isize, argv: *const *const u8) -> isize { 0 }
# fn main() {}
```

Note that there is one extra lang item here which differs from the examples above, panic_fmt. This must be defined by consumers of libcore because the core library declares panics, but it does not define it. The panic_fmt lang item is this crate's definition of panic, and it must be guaranteed to

never return.

As can be seen in this example, the core library is intended to provide the power of Rust in all circumstances, regardless of platform requirements. Further libraries, such as liballoc, add functionality to libcore which make other platform-specific assumptions, but continue to be more portable than the standard library itself.

6.4   Intrinsics 内部函数

> **Note**: intrinsics will forever have an unstable interface, it is recommended to use the stable interfaces of libcore rather than intrinsics directly.

These are imported as if they were FFI functions, with the special rust-intrinsic ABI. For example, if one was in a freestanding context, but wished to be able to transmute between types, and perform efficient pointer arithmetic, one would import those functions via a declaration like

```
# #![feature(intrinsics)]
# fn main() {}

extern "rust-intrinsic" {
  fn transmute<T, U>(x: T) -> U;

  fn offset<T>(dst: *const T, offset: isize) -> *const T;
}
```

As with any other FFI functions, these are always unsafe to call.

6.5   Lang items 组件

> **Note**: lang items are often provided by crates in the Rust distribution, and lang items themselves have an unstable interface. It is recommended to use officially distributed crates instead of defining your own lang items.

The rustc compiler has certain pluggable operations, that is, functionality that isn't hard-coded into the language, but is implemented in libraries, with a special marker to tell the compiler it exists. The marker is the attribute #[lang = "..."] and there are various different values of ..., i.e. various different 'lang items'.

For example, Box pointers require two lang items, one for allocation and one for deallocation. A freestanding program that uses the Box sugar for dynamic allocations via malloc and free:

```
#![feature(lang_items, box_syntax, start, no_std, libc)]
#![no_std]
```

```
extern crate libc;

extern {
  fn abort() -> !;
}

#[lang = "owned_box"]
pub struct Box<T>(*mut T);

#[lang = "exchange_malloc"]
unsafe fn allocate(size: usize, _align: usize) -> *mut u8 {
  let p = libc::malloc(size as libc::size_t) as *mut u8;

  // malloc failed
  if p as usize == 0 {
    abort();
  }

  p
}
#[lang = "exchange_free"]
unsafe fn deallocate(ptr: *mut u8, _size: usize, _align: usize) {
  libc::free(ptr as *mut libc::c_void)
}

#[start]
fn main(argc: isize, argv: *const *const u8) -> isize {
  let x = box 1;

  0
}

#[lang = "stack_exhausted"] extern fn stack_exhausted() {}
#[lang = "eh_personality"] extern fn eh_personality() {}
#[lang = "panic_fmt"] fn panic_fmt() -> ! { loop {} }
```

Note the use of abort: the exchange_malloc lang item is assumed to return a valid pointer, and so needs to do the check internally.

Other features provided by lang items include:

- overloadable operators via traits: the traits corresponding to the ==, <, dereferencing (*) and + (etc.) operators are all marked with lang items; those specific four are eq, ord, deref, and add respectively.
- stack unwinding and general failure; the eh_personality, fail and fail_bounds_checks lang items.

- the traits in std::marker used to indicate types of various kinds; lang items send, sync and copy.
- the marker types and variance indicators found in std::marker; lang items covariant_type, contravariant_lifetime, etc.

Lang items are loaded lazily by the compiler; e.g. if one never uses Box then there is no need to define functions for exchange_malloc and exchange_free. rustc will emit an error when an item is needed but not found in the current crate or any that it depends on.

6.6   Link args 连接属性

There is one other way to tell rustc how to customize linking, and that is via the link_args attribute. This attribute is applied to extern blocks and specifies raw flags which need to get passed to the linker when producing an artifact. An example usage would be:

```
#![feature(link_args)]

#[link_args = "-foo -bar -baz"]
extern {}
# fn main() {}
```

Note that this feature is currently hidden behind the feature(link_args) gate because this is not a sanctioned way of performing linking. Right now rustc shells out to the system linker, so it makes sense to provide extra command line arguments, but this will not always be the case. In the future rustc may use LLVM directly to link native libraries in which case link_args will have no meaning.

It is highly recommended to *not* use this attribute, and rather use the more formal #[link(...)] attribute on extern blocks instead.

6.7   Benchmark Tests 基准测试

Rust supports benchmark tests, which can test the performance of your code. Let's make our src/lib.rs look like this (comments elided):

```
#![feature(test)]

extern crate test;

pub fn add_two(a: i32) -> i32 {
  a + 2
}

#[cfg(test)]
mod tests {
```

```rust
use super::*;
use test::Bencher;

#[test]
fn it_works() {
    assert_eq!(4, add_two(2));
}

#[bench]
fn bench_add_two(b: &mut Bencher) {
    b.iter(|| add_two(2));
}
}
```

Note the `test` feature gate, which enables this unstable feature.

We've imported the `test` crate, which contains our benchmarking support. We have a new function as well, with the `bench` attribute. Unlike regular tests, which take no arguments, benchmark tests take a `&mut Bencher`. This `Bencher` provides an `iter` method, which takes a closure. This closure contains the code we'd like to benchmark.

We can run benchmark tests with `cargo bench`:

```
$ cargo bench
   Compiling adder v0.0.1 (file:///home/steve/tmp/adder)
     Running target/release/adder-91b3e234d4ed382a

running 2 tests
test tests::it_works ... ignored
test tests::bench_add_two ... bench:        1 ns/iter (+/- 0)

test result: ok. 0 passed; 0 failed; 1 ignored; 1 measured
```

Our non-benchmark test was ignored. You may have noticed that `cargo bench` takes a bit longer than `cargo test`. This is because Rust runs our benchmark a number of times, and then takes the average. Because we're doing so little work in this example, we have a `1 ns/iter (+/- 0)`, but this would show the variance if there was one.

Advice on writing benchmarks:

- Move setup code outside the `iter` loop; only put the part you want to measure inside
- Make the code do "the same thing" on each iteration; do not accumulate or change state
- Make the outer function idempotent too; the benchmark runner is likely to run it many times
- Make the inner `iter` loop short and fast so benchmark runs are fast and the calibrator can adjust the run-length at fine resolution
- Make the code in the `iter` loop do something simple, to assist in pinpointing performance improvements (or regressions)

Gotcha: optimizations

There's another tricky part to writing benchmarks: benchmarks compiled with optimizations activated can be dramatically changed by the optimizer so that the benchmark is no longer benchmarking what one expects. For example, the compiler might recognize that some calculation has no external effects and remove it entirely.

```
#![feature(test)]

extern crate test;
use test::Bencher;

#[bench]
fn bench_xor_1000_ints(b: &mut Bencher) {
    b.iter(|| {
        (0..1000).fold(0, |old, new| old ^ new);
    });
}
```

gives the following results

```
running 1 test
test bench_xor_1000_ints ... bench:        0 ns/iter (+/- 0)

test result: ok. 0 passed; 0 failed; 0 ignored; 1 measured
```

The benchmarking runner offers two ways to avoid this. Either, the closure that the iter method receives can return an arbitrary value which forces the optimizer to consider the result used and ensures it cannot remove the computation entirely. This could be done for the example above by adjusting the b.iter call to

```
b.iter(|| {
    // note lack of `;` (could also use an explicit `return`).
    (0..1000).fold(0, |old, new| old ^ new)
});
```

Or, the other option is to call the generic test::black_box function, which is an opaque "black box" to the optimizer and so forces it to consider any argument as used.

```
#![feature(test)]

extern crate test;

b.iter(|| {
    let n = test::black_box(1000);
```

```
   (0..n).fold(0, |a, b| a ^ b)
})
```

Neither of these read or modify the value, and are very cheap for small values. Larger values can be passed indirectly to reduce overhead (e.g. `black_box(&huge_struct)`).

Performing either of the above changes gives the following benchmarking results

```
running 1 test
test bench_xor_1000_ints ... bench:      131 ns/iter (+/- 3)

test result: ok. 0 passed; 0 failed; 0 ignored; 1 measured
```

However, the optimizer can still modify a testcase in an undesirable manner even when using either of the above.

6.8 Box Syntax and Patterns Box 语法和模式

Currently the only stable way to create a `Box` is via the `Box::new` method. Also it is not possible in stable Rust to destructure a `Box` in a match pattern. The unstable `box` keyword can be used to both create and destructure a `Box`. An example usage would be:

```rust
#![feature(box_syntax, box_patterns)]

fn main() {
  let b = Some(box 5);
  match b {
    Some(box n) if n < 0 => {
      println!("Box contains negative number {}", n);
    },
    Some(box n) if n >= 0 => {
      println!("Box contains non-negative number {}", n);
    },
    None => {
      println!("No box");
    },
    _ => unreachable!()
  }
}
```

Note that these features are currently hidden behind the `box_syntax` (box creation) and `box_patterns` (destructuring and pattern matching) gates because the syntax may still change in the future.

6.8.1   Returning Pointers

In many languages with pointers, you'd return a pointer from a function so as to avoid copying a large data structure. For example:

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

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

fn main() {
  let x = Box::new(BigStruct {
    one: 1,
    two: 2,
    one_hundred: 100,
  });

  let y = foo(x);
}
```

The idea is that by passing around a box, you're only copying a pointer, rather than the hundred ints that make up the BigStruct.

This is an antipattern in Rust. Instead, write this:

```rust
#![feature(box_syntax)]

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

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

fn main() {
  let x = Box::new(BigStruct {
```

```
    one: 1,
    two: 2,
    one_hundred: 100,
  });

  let y: Box<BigStruct> = box foo(x);
}
```

This gives you flexibility without sacrificing performance.

You may think that this gives us terrible performance: return a value and then immediately box it up ?! Isn't this pattern the worst of both worlds? Rust is smarter than that. There is no copy in this code. main allocates enough room for the box, passes a pointer to that memory into foo as x, and then foo writes the value straight into the Box<T>.

This is important enough that it bears repeating: pointers are not for optimizing returning values from your code. Allow the caller to choose how they want to use your output.

6.9   Slice Patterns 分片模式

If you want to match against a slice or array, you can use & with the slice_patterns feature:

```
#![feature(slice_patterns)]

fn main() {
  let v = vec!["match_this", "1"];

  match &v[..] {
    ["match_this", second] => println!("The second element is {}", second),
    _ => {},
  }
}
```

The advanced_slice_patterns gate lets you use .. to indicate any number of elements inside a pattern matching a slice. This wildcard can only be used once for a given array. If there's an identifier before the .., the result of the slice will be bound to that name. For example:

```
#![feature(advanced_slice_patterns, slice_patterns)]

fn is_symmetric(list: &[u32]) -> bool {
  match list {
    [] | [_] => true,
    [x, inside.., y] if x == y => is_symmetric(inside),
    _ => false
```

```
  }
}

fn main() {
  let sym = &[0, 1, 4, 2, 4, 1, 0];
  assert!(is_symmetric(sym));

  let not_sym = &[0, 1, 7, 2, 4, 1, 0];
  assert!(!is_symmetric(not_sym));
}
```

6.10   Associated Constants 关联常量

With the `associated_consts` feature, you can define constants like this:

```
#![feature(associated_consts)]

trait Foo {
  const ID: i32;
}

impl Foo for i32 {
  const ID: i32 = 1;
}

fn main() {
  assert_eq!(1, i32::ID);
}
```

Any implementor of `Foo` will have to define `ID`. Without the definition:

```
#![feature(associated_consts)]

trait Foo {
  const ID: i32;
}

impl Foo for i32 {
}
```

gives

```
error: not all trait items implemented, missing: `ID` [E0046]
```

```
impl Foo for i32 {
}
```

A default value can be implemented as well:

```
#![feature(associated_consts)]

trait Foo {
    const ID: i32 = 1;
}

impl Foo for i32 {
}

impl Foo for i64 {
    const ID: i32 = 5;
}

fn main() {
    assert_eq!(1, i32::ID);
    assert_eq!(5, i64::ID);
}
```

As you can see, when implementing Foo, you can leave it unimplemented, as with i32. It will then use the default value. But, as in i64, we can also add our own definition.

Associated constants don't have to be associated with a trait. An impl block for a struct works fine too:

```
#![feature(associated_consts)]

struct Foo;

impl Foo {
    pub const FOO: u32 = 3;
}
```

7

Glossary 专业术语

Not every Rustacean has a background in systems programming, nor in computer science, so we've added explanations of terms that might be unfamiliar.

Arity   Arity refers to the number of arguments a function or operation takes.

```
let x = (2, 3);
let y = (4, 6);
let z = (8, 2, 6);
```

In the example above x and y have arity 2. z has arity 3.

Abstract Syntax Tree   When a compiler is compiling your program, it does a number of different things. One of the things that it does is turn the text of your program into an 'abstract syntax tree', or 'AST'. This tree is a representation of the structure of your program. For example, 2 + 3 can be turned into a tree:

```
 +
/ \
2  3
```

And 2 + (3 * 4) would look like this:

```
 +
/ \
2  *
  / \
  3  4
```

8

Academic Research 学术研究

An incomplete list of papers that have had some influence in Rust.

Recommended for inspiration and a better understanding of Rust's background.

Type system

- Region based memory management in Cyclone[1]
- Safe manual memory management in Cyclone[2]
- Typeclasses: making ad-hoc polymorphism less ad hoc[3]
- Macros that work together[4]
- Traits: composable units of behavior[5]
- Alias burying[6] - We tried something similar and abandoned it.
- External uniqueness is unique enough[7]
- Uniqueness and Reference Immutability for Safe Parallelism[8]
- Region Based Memory Management[9]

Concurrency

[1] http://209.68.42.137/ucsd-pages/Courses/cse227.w03/handouts/cyclone-regions.pdf
[2] http://www.cs.umd.edu/projects/PL/cyclone/scp.pdf
[3] http://www.ps.uni-sb.de/courses/typen-ws99/class.ps.gz
[4] https://www.cs.utah.edu/plt/publications/jfp12-draft-fcdf.pdf
[5] http://scg.unibe.ch/archive/papers/Scha03aTraits.pdf
[6] http://www.cs.uwm.edu/faculty/boyland/papers/unique-preprint.ps
[7] http://www.computingscience.nl/research/techreps/repo/CS-2002/2002-048.pdf
[8] https://research.microsoft.com/pubs/170528/msr-tr-2012-79.pdf
[9] http://www.cs.ucla.edu/~palsberg/tba/papers/tofte-talpin-iandc97.pdf

- Singularity: rethinking the software stack[10]
- Language support for fast and reliable message passing in singularity OS[11]
- Scheduling multithreaded computations by work stealing[12]
- Thread scheduling for multiprogramming multiprocessors[13]
- The data locality of work stealing[14]
- Dynamic circular work stealing deque[15] - The Chase/Lev deque
- Work-first and help-first scheduling policies for async-finish task parallelism[16] - More general than fully-strict work stealing
- A Java fork/join calamity[17] - critique of Java's fork/join library, particularly its application of work stealing to non-strict computation
- Scheduling techniques for concurrent systems[18]
- Contention aware scheduling[19]
- Balanced work stealing for time-sharing multicores[20]
- Three layer cake[21]
- Non-blocking steal-half work queues[22]
- Reagents: expressing and composing fine-grained concurrency[23]
- Algorithms for scalable synchronization of shared-memory multiprocessors[24]

Others

- Crash-only software[25]
- Composing High-Performance Memory Allocators[26]
- Reconsidering Custom Memory Allocation[27]

Papers *about* Rust

- GPU programming in Rust[28]

---

[10] https://research.microsoft.com/pubs/69431/osr2007_rethinkingsoftwarestack.pdf

[11] https://research.microsoft.com/pubs/67482/singsharp.pdf

[12] http://supertech.csail.mit.edu/papers/steal.pdf

[13] http://www.eecis.udel.edu/%7Ecavazos/cisc879-spring2008/papers/arora98thread.pdf

[14] http://www.aladdin.cs.cmu.edu/papers/pdfs/y2000/locality_spaa00.pdf

[15] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.170.1097&rep=rep1&type=pdf

[16] http://www.cs.rice.edu/%7Eyguo/pubs/PID824943.pdf

[17] http://www.coopsoft.com/ar/CalamityArticle.html

[18] http://www.ece.rutgers.edu/%7Eparashar/Classes/ece572-papers/05/ps-ousterhout.pdf

[19] http://www.blagodurov.net/files/a8-blagodurov.pdf

[20] http://www.cse.ohio-state.edu/hpcs/WWW/HTML/publications/papers/TR-12-1.pdf

[21] http://www.upcrc.illinois.edu/workshops/paraplop10/papers/paraplop10_submission_8.pdf

[22] http://www.cs.bgu.ac.il/%7Ehendlerd/papers/p280-hendler.pdf

[23] http://www.mpi-sws.org/~turon/reagents.pdf

[24] https://www.cs.rochester.edu/u/scott/papers/1991_TOCS_synch.pdf

[25] https://www.usenix.org/legacy/events/hotos03/tech/full_papers/candea/candea.pdf

[26] http://people.cs.umass.edu/~emery/pubs/berger-pldi2001.pdf

[27] http://people.cs.umass.edu/~emery/pubs/berger-oopsla2002.pdf

[28] http://www.cs.indiana.edu/~eholk/papers/hips2013.pdf

- Parallel closures: a new twist on an old idea[29] - not exactly about rust, but by nmatsakis

---