概述

Rust 语言是一种高效、可靠的通用高级语言。其具有一下特点：

- 高性能 - Rust 速度惊人且内存利用率极高。
- 可靠性 - Rust 丰富的类型系统和所有权模型保证了内存安全和线程安全，让您在编译期就能够消除各种各样的错误。
- 生产力 - Rust 拥有出色的文档、友好的编译器和清晰的错误提示信息，还集成了一流的工具。

文档：https://kaisery.github.io/trpl-zh-cn/ch01-01-installation.html
语法：https://doc.rust-lang.org/reference/introduction.html

StartUp

环境搭建

Unix

1. 执行以下命令进行安装：

```
curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

Windows

1. 从 https://www.rust-lang.org/install.html 进行下载安装。

HelloWorld

执行以下命令创建项目，并进入项目文件夹：

```
cargo new hello_world
cd hello_world
```

hello_word 目录下面存在以下两个文件：

```
Cargo.toml
src/main.rs
```

Cargo.toml 文件内容：

```
[package]
name = "hello_world"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
edition = "2018"

[dependencies]
```

src/main.rs 文件内容：

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

执行以下命令进行代码编译：

```
cargo build
```

这个命令会创建一个可执行文件./target/debug/hello_world

文件布局

Cargo 使用文件放置约定，文件放置具有以下规则：

```
.
├── Cargo.lock
├── Cargo.toml
├── src/
│   ├── lib.rs
│   ├── main.rs
│   └── bin/
│       ├── named-executable.rs
│       ├── another-executable.rs
│       └── multi-file-executable/
│           ├── main.rs
```

```
│               └── some_module.rs
├── benches/
│   ├── large-input.rs
│   └── multi-file-bench/
│       ├── main.rs
│       └── bench_module.rs
├── examples/
│   ├── simple.rs
│   └── multi-file-example/
│       ├── main.rs
│       └── ex_module.rs
└── tests/
    ├── some-integration-tests.rs
    └── multi-file-test/
        ├── main.rs
        └── test_module.rs
```

- Cargo.toml 和 Cargo.lock 存储在包的根目录。
- 源代码位于 src 目录中。
- 默认库文件是 src/lib.rs。
- 默认的可执行文件是 src/main.rs。
- 其他可执行文件可以放在 src/bin/ 中。
- 基准测试文件在 benches 目录中。
- 示例文件在 examples 目录中。
- 集成测试文件在 tests 目录。

注释

在 Rust 中，注释必须以两道斜杠开始，并持续到本行的结尾。对于超过一行的注释，需要在每一行前都加上 //。

例子：

```
// So we’re doing something complicated here, long enough that we need
// multiple lines of comments to do it! Whew! Hopefully, this comment will
// explain what’s going on.
```

rust 支持的注释方式有如下一些：

- //：行注释
- //!：内部行文档注释
- ///：外部行文档注释
- /*...*/：块注释
- /*!...*/：内部块文档注释
- /**...*/：外部块文档注释

关键字

如下关键字目前有对应其描述的功能。

- as - 强制类型转换，消除特定包含项的 trait 的歧义，或者对 use 和 extern crate 语句中的项重命名
- break - 立刻退出循环
- const - 定义常量或不变裸指针（constant raw pointer）
- continue - 继续进入下一次循环迭代
- crate - 链接（link）一个外部 crate 或一个代表宏定义的 crate 的宏变量
- dyn - 动态分发 trait 对象
- else - 作为 if 和 if let 控制流结构的 fallback
- enum - 定义一个枚举
- extern - 链接一个外部 crate 、函数或变量
- false - 布尔字面值 false
- fn - 定义一个函数或 函数指针类型 (function pointer type)
- for - 遍历一个迭代器或实现一个 trait 或者指定一个更高级的生命周期
- if - 基于条件表达式的结果分支
- impl - 实现自有或 trait 功能
- in - for 循环语法的一部分
- let - 绑定一个变量
- loop - 无条件循环
- match - 模式匹配
- mod - 定义一个模块
- move - 使闭包获取其所捕获项的所有权
- mut - 表示引用、裸指针或模式绑定的可变性

- pub - 表示结构体字段、impl 块或模块的公有可见性
- ref - 通过引用绑定
- return - 从函数中返回
- Self - 实现 trait 的类型的类型别名
- self - 表示方法本身或当前模块
- static - 表示全局变量或在整个程序执行期间保持其生命周期
- struct - 定义一个结构体
- super - 表示当前模块的父模块
- trait - 定义一个 trait
- true - 布尔字面值 true
- type - 定义一个类型别名或关联类型
- unsafe - 表示不安全的代码、函数、trait 或实现
- use - 引入外部空间的符号
- where - 表示一个约束类型的从句
- while - 基于一个表达式的结果判断是否进行循环

如下关键字没有任何功能，不过由 Rust 保留以备将来的应用。
- abstract
- async
- await
- become
- box
- do
- final
- macro
- override
- priv
- try
- typeof
- unsized
- virtual
- yield

crate

crate 是一个 binary 或者 library。crate root 是一个源文件，Rust 编译器以它为起始点，并构成你的 crate 的根模块。

Cargo 遵循的一个约定：
- src/main.rs 就是一个与包同名的 binary crate 的 crate 根。
- 如果包目录中包含 src/lib.rs，则包带有与其同名的 library crate，且 src/lib.rs 是 crate 根。

通过将文件放在 src/bin 目录下，一个包可以拥有多个 binary crate：每个 src/bin 目录下的文件(或子目录)都会被编译成一个独立的二进制 crate。

package

包（package）是提供一系列功能的一个或者多个 crate。一个包会包含有一个 Cargo.toml 文件，阐述如何去构建这些 crate。

包中所包含的内容由几条规则来确立：
1. 一个包中至多只能包含一个 library crate；
2. 一个包中可以包含任意多个 binary crate；
3. 一个包中至少包含一个 crate，无论是 library crate 或 binary crate。

workspace

Cargo 提供了一个叫 workspace 的功能，它可以帮助我们管理多个相关的协同开发的 package。workspace 管理一系列共享同样的 Cargo.lock 和 target 输出目录的 package。

由于 workspace 只在根目录有一个 Cargo.lock，而不是在每一个 crate 目录都有 Cargo.lock。如果 Cargo 解析到在不同的 crate 中存在多个版本的相同依赖，则会将其都解析为同一版本并记录到唯一的 Cargo.lock 中。这确保了所有的 crate 都使用完全相同版本的依赖。

如果需要向 crates.io 发布 workspace 中的 crate， workspace 中的每一个 crate 都需要单独发布。cargo publish 命令并没有 --all 或者 -p 参数，所以必须进入每一个 crate 的目录并运行 cargo publish 来发布 workspace 中的每一个 crate。

下面是一个创建有一个 binary crate 和一个 library crate 的例子：

```
//1.首先创建工作空间目录
```

```
$mkdir add-example
$cd add-example

//2.在工作空间目录创建 Cargo.toml 文件。注意：工作空间目录的 Cargo.toml 只包含成员信息
[workspace]
members = [
    "adder",
    "add-one",
]

//3.在 add-example 目录运行 cargo new adder 创建 binary crate
$cargo new adder

//4.在 add-example 目录运行 cargo new add-one --lib 创建 library crate
$ cargo new add-one --lib

//5.创建完成后将会拥有如下结构
├──── Cargo.toml
├──── add-one
│    ├──── Cargo.toml
│    └──── src
│        └──── lib.rs
├──── adder
│    ├──── Cargo.toml
│    └──── src
│        └──── main.rs

//6.在 add-one/src/lib.rs 中添加如下代码
pub fn add_one(x: i32) -> i32 {
    x + 1
}

//7.在 adder/Cargo.toml 添加内部依赖
[dependencies]
add-one = { path = "../add-one" }

//8.在 adder/src/main.rs 中添加如下代码
use add_one;
fn main() {
    let num = 10;
    println!("Hello, world! {} plus one is {}!", num, add_one::add_one(num));
}

//9.在 add-example 目录中运行 cargo build 来构建工作空间
$ cargo build

//10.build 完成后目录结构如下
├──── Cargo.lock
├──── Cargo.toml
├──── add-one
│    ├──── Cargo.toml
│    └──── src
│        └──── lib.rs
├──── adder
│    ├──── Cargo.toml
│    └──── src
│        └──── main.rs
└──── target
```

module

模块（module）让我们可以将一个 crate 中的代码进行分组，以提高可读性与重用性。模块还可以控制项的私有性，即项是可以被外部代码使用的（public），还是作为一个内部实现的内容而不能被外部代码使用（private）。

模块定义

定义一个模块，是以 mod 关键字为起始，然后指定模块的名字，并且用花括号包围模块的主体。在模块内，我们还可以定义其他的模块，以及保存一些定义的其他项，比如结构体、枚举、常量、特性、或者函数。例如在 src/lib.rs 中定义如下内容：

```
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}
        fn seat_at_table() {}
    }

    mod serving {
        fn take_order() {}
        fn server_order() {}
        fn take_payment() {}
    }
}
```

模块路径

src/main.rs 和 src/lib.rs 叫做 crate 根，这两个文件的内容都分别在 crate 模块结构的根组成了一个名为 crate 的模块，该结构被称为模块树（module tree）。例如上例的模块树结构如下：

```
crate
 └── front_of_house
     ├── hosting
     │   ├── add_to_waitlist
     │   └── seat_at_table
     └── serving
         ├── take_order
         ├── serve_order
         └── take_payment
```

使用路径的方式，可以在模块树中找到一个项的位置，路径有两种形式：

- 绝对路径（absolute path）从 crate 根开始，以 crate 名或者字面值 crate 开头。
- 相对路径（relative path）从当前模块开始，以 self、super 或当前模块的标识符开头。

模块私有性

模块和模块直接的关系定义如下：

- 如果一个模块 A 被包含在模块 B 中，则模块 A 称为模块 B 的子模块。
- 如果一个模块 A 被包含在模块 B 中，则模块 B 称为模块 A 的父模块。

Rust 中默认所有项（函数、方法、结构体、枚举、模块和常量）都是私有的。默认可见性规则如下：

1. 父模块中的项不能使用子模块中的私有项。
2. 子模块中的项可以使用父模块中的私有项。

通过使用 pub 关键字来创建公共项，使子模块的内部部分暴露给上级模块。例如：

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // Absolute path
    crate::front_of_house::hosting::add_to_waitlist();
    // Relative path
    front_of_house::hosting::add_to_waitlist();
}
```

另外在一个结构体定义的前面使用了 pub ，这个结构体会变成公有的，但是这个结构体的字段仍然是私有的，可以在字段前添加 pub 使其变为公有字段。例如：

```
mod back_of_house {
    pub struct Breakfast {
        pub toast: String,
        seasonal_fruit: String,
    }
```

```
    impl Breakfast {
        pub fn summer(toast: &str) -> Breakfast {
            Breakfast {
                toast: String::from(toast),
                seasonal_fruit: String::from("peaches"),
            }
        }
    }
}
```

注意：对于公有枚举，其每个成员都是公有的，不必在枚举前添加额外的 pub 使用变为公有。

模块分割

可以通过将不同模块分割进不同的文件来更好的进行代码组织。每个文件就是一个模块，通过递进式的申明(父模块申明有哪些子模块)，可以构造一个完整的模块树。

进行模块分割有以下要点：

- 根模块是 crate，即 lib.rs 或 main.rs 所在的目录。
- 模块名称与文件名称一致。例如模块 front_of_house 的文件名称为 front_of_house.rs。
- 在父模块中使用<pub> mod <module_name>的方式申明他的子模块。子模块文件在与父模块同名的目录下。
- 根模块 crate 下的子模块在 lib.rs 或 main.rs 中申明，这些子模块文件和 lib.rs 或 main.rs 同目录。
- 根据私有性规则，同一目录下的不同模块中的项，只能访问对方的公有项。

例如在 src/main.rs 中定义如下内容：

```
mod front_of_house {
    mod hosting {
        pub fn add_to_waitlist() {
            println!("Hello, world!");
        }
    }

    pub mod serving {
        use crate::front_of_house::hosting;
        pub fn take_order() {
            hosting::add_to_waitlist()
        }
    }
}

fn main() {
    front_of_house::serving::take_order();
}
```

进行拆分后的文件目录如下：

```
├─── Cargo.toml
└─── src
    └─── main.rs
    └─── front_of_house.rs
    └─── front_of_house
        └─── hosting.rs
        └─── serving.rs
```

main.rs 文件内容如下：

```
mod front_of_house;

fn main() {
    front_of_house::serving::take_order();
}
```

front_of_house.rs 文件内容如下：

```
mod hosting;
pub mod serving;
```

front_of_house/hosting.rs 文件内容如下：

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

front_of_house/serving.rs 文件内容如下：

```
use crate::front_of_house::hosting;
pub fn take_order() {
```

```
    hosting::add_to_waitlist()
}
```

模块引入

可以使用 use 和模块绝对路径将其他模块的项引入当前作用域。例如：

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
}
```

通过在 crate 根增加 use crate::front_of_house::hosting，现在 hosting 在作用域中就是有效的名称了，如同 hosting 模块被定义于 crate 根一样。

也可以使用 use 和相对路径来将其他模块的项引入当前作用域。例如：

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
}
```

as 别称

使用 as 关键字可以提供新的名称。例如：

```
use std::fmt::Result;
use std::io::Result as IoResult;

fn function1() -> Result {
    // --snip--
}

fn function2() -> IoResult<()> {
    // --snip--
}
```

重导出名称

当使用 use 关键字将名称导入作用域时，在新作用域中可用的名称是私有的。如果为了让调用你编写的代码能够像在自己的作用域内引用这些类型，可以结合 pub 和 use 重导出名称。

例子：

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

pub use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
```

```
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
}
```

通过 pub use，外部代码现在可以通过新路径 hosting::add_to_waitlist 来调用 add_to_waitlist 函数。如果没有指定 pub use，eat_at_restaurant 函数可以在其作用域中调用 hosting::add_to_waitlist，但外部代码则不允许使用这个新路径。

使用外部包

使用外部包时，首先需要在 Cargo.toml 中添加外部包。例如使用 rand 外部包：

```
// Cargo.toml
[dependencies]
rand = "0.5.5"
```

接着，将外部包中的定义引入项目包的作用域。例如引入 rand::Rng：

```
use rand::Rng;

fn main() {
    let secret_number = rand::thread_rng().gen_range(1, 101);
}
```

注意标准库（std）对于你的包来说也是外部 crate。因为标准库随 Rust 语言一同分发，无需修改 Cargo.toml 来引入 std，而只需要直接引入即可。

嵌套路径引入

当需要引入很多定义于相同包或相同模块的项时，为每一项单独列出一行会占用源码很大的空间，这个时候可以使用嵌套路径将相同的项在一行中引入作用域。例如：

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

// 等价于

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

引入所有公有定义

通过 glob 运算符"*"可以将所有的公有定义引入作用域。例如：

```
use std::collections::*;
```

使用 glob 运算符时请多加小心！Glob 会使得我们难以推导作用域中有什么名称和它们是在何处定义的。

glob 运算符经常用于测试模块 tests 中，有时也用于 prelude 模式。

所有权

Rust 的值可以在堆上，也可以在栈上。栈上的值可以通过指向该值的"引用"或者该值"本身"进行访问，堆上的值只能通过指向该值的"智能指针"进行访问。注意：

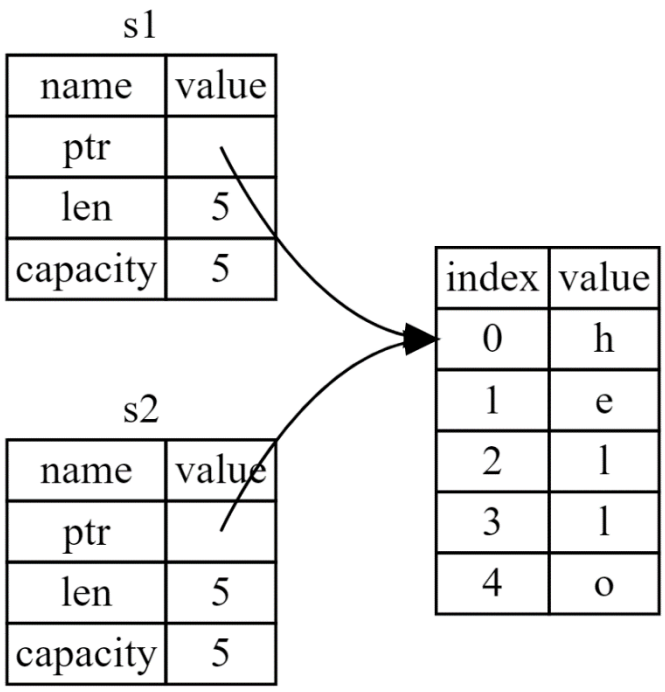
- 值的引用类似于 C++指针值，但是引用只可以引用栈上的值。
- 智能指针表现的像引用一样，智能指针的内部可以存在指向堆上的值，例如 Box、Rc、Week 等存在指向堆上值。
- 引用和智能指针本身都是栈上的值。

Rust 的所有权针对是栈上的值，具有以下规则：

1. 这些值都有一个被称为其所有者（owner）的变量。
2. 一个值在任意时刻有且只有一个所有者。
3. 当所有者（变量）离开作用域，这个值将被丢弃，如果该值的变量类型实现了 Drop trait，那么会调用其 drop 函数进行堆内存释放操作。

所有权转移

Rust 的一个变量 A 的值复制给另外一个变量 B 时，会进行一次浅拷贝，将栈上的值完全复制一份新值，新值的所有者为变量 B。如果此时变量 A 和变量 B 的类型是一个实现了 Drop trait 的类型，如果安装其他语言的逻辑，在变量 A 和变量 B 分别离开作用域时，将会导致两次的 drop，而导致堆内存的两次释放。例如：



因此 Rust 在进行变量复制时采用以下规则：

1. 默认情况下 B=A 将使用变量移动的方式，即 B=A 将导致变量 A 失效，在该语句之后无法再使用变量 A。
2. 如果变量类型添加 Copy trait 注解，那么 B=A 将不会导致变量 A 失效，Copy 注解指示值完全在栈上。以下基础类型是默认 Copy 类型：
   - 整数类型，比如 u32。
   - 布尔类型，true 和 false。
   - 浮点数类型，比如 f64。
   - 字符类型，char。
   - 元组，当且仅当其包含的类型也都是 Copy 的时候。比如，(i32, i32) 是 Copy 的，但 (i32, String) 就不是。

注意：

1. 只有在变量离开作用域的时候才会进行 drop，所有权转移的时候不会进行任何操作。
2. 引用离开作用域时不会导致其引用的值被 drop。因为引用其实就是 Copy 类型值。
3. 对于函数的入参以及返回值，遵循变量复制的规则。
4. 如果变量的类型实现了 Drop trait，那么其就无法添加 Copy trait 注解，他们是互斥的。

引用/借用

默认情况下，如果一个变量既非引用，也非 Copy 类型，那么其作为参数传递给函数后将无法使用。通常我们使用引用来解决这类问题，并且引用可以避免栈上的浅拷贝。

"&"符号作用与类型前面，代表这是一个引用类型(类似指针)，"&"符号作用与表达式前面，代表这是一个借用(类似取地址)，这允许你使用值但不获取其所有权。例子：

```
fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1);
    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

注意：引用类型的变量离开作用域时不会导致其引用的值被 drop。

解引用运算符

常规引用是一个指针类型，一种理解指针的方式是将其看成指向储存在其他某处值的箭头。在 Rust 中可以使用解引用运算符"*"来跟踪所引用的数据。

例如：

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

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

可变引用

默认情况下的引用是不可变引用，可以添加 mut 变为可变引用。可变引用指向的是可变变量。例如：

```
fn main() {
    let mut s = String::from("hello");
    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

可变引用具有以下限制：

1. 引用的作用域从声明的地方开始一直持续到最后一次使用为止。
2. 在特定作用域中的特定数据只能有一个可变引用。
3. 在特定作用域中的特定数据不能同时包含可变引用和不可变引用。

只能有一个可变引用例子：

```
let mut s = String::from("hello");

// 无效代码
// let r1 = &mut s;
// let r2 = &mut s;

println!("{}, {}", r1, r2);
```

不能同时包含可变引用和不可变引用例子：

```
let mut s = String::from("hello");

let r1 = &s; // 没问题
let r2 = &s; // 没问题
let r3 = &mut s; // 大问题

println!("{}, {}, and {}", r1, r2, r3);
```

引用作用域例子：

```
let mut s = String::from("hello");

let r1 = &s; // 没问题
let r2 = &s; // 没问题
println!("{} and {}", r1, r2);
// 此位置之后 r1 和 r2 不再使用

let r3 = &mut s; // 没问题
println!("{}", r3);
```

生命周期

生命周期的主要目标是避免悬垂引用，避免引用的数据已经提前离开作用域，这会导致程序引用了非预期引用的数据。例如：

```
{
    let r;
    {
        let x = 5;
        r = &x;
    }
    println!("r: {}", r);
}
// 尝试使用离开作用域的值的引用
```

Rust 编译器有一个"借用检查器"，它比较作用域来确保所有的借用都是有效的。借用检查器会检查引用的生命周期是否比被引用的数据短。借用检查器需要配合"生命周期注解语法"来检查借用的有效性。

生命周期注解语法：生命周期参数名称必须以撇号（'）开头，其名称通常全是小写，类似于泛型其名称非常短。'a 是大多数人默认使用的名称。生命周期参数注解位于引用的 & 之后，并有一个空格来将引用类型与生命周期注解分隔开。例如：

```
&i32        // 引用
&'a i32     // 带有显式生命周期的引用
```

```
&'a mut i32 // 带有显式生命周期的可变引用
```

函数定义中的生命周期注解

当从函数返回一个引用，返回值的生命周期参数需要与一个参数的生命周期参数相匹配。如果返回的引用没有指向任何一个参数，那么唯一的可能就是它指向一个函数内部创建的值，它将会是一个悬垂引用，因为它将会在函数结束时离开作用域。

就像泛型类型参数，泛型生命周期参数需要声明在函数名和参数列表间的尖括号中，并且泛型生命周期参数不存在于函数体中的任何代码中。

例如：

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

生命周期参数是一个约束，如果函数的引用返回值和某个入参变量不相干，那么不要为其添加生命周期参数。例如：

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

结构体定义中的生命周期注解

当结构体内部包含引用时，需要添加生命周期注解。

类似于泛型参数类型，必须在结构体名称后面的尖括号中声明泛型生命周期参数，以便在结构体定义中使用生命周期参数。

例子：

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}
```

方法定义中的生命周期注解

定义方法时，结构体字段的生命周期必须总是在 impl 关键字之后声明并在结构体名称之后被使用。和定义函数一样，方法的泛型生命周期参数需要声明在函数名和参数列表间的尖括号中，并且泛型生命周期参数不存在于函数体中的任何代码中。

例子：

```
impl<'a> ImportantExcerpt<'a> {
    fn announce_and_return_part<'b>(&self, announcement: &'b str) -> &'a str {
        println!("Attention please: {}", announcement);
        self.part
    }
}
```

静态生命周期注解

'static 是生命周期注解，表明其生命周期能够存活于整个程序期间。所有的字符串字面值都拥有 'static 生命周期。

例子：

```
let s: &'static str = "I have a static lifetime.";
```

省略的生命周期注解

编译器采用三条规则来判断引用何时不需要明确的注解，其中函数或方法的参数的生命周期被称为"输入生命周期"（input lifetimes），而返回值的生命周期被称为"输出生命周期"（output lifetimes）。如果编译器检查完这三条规则后仍然存在没有计算出生命周期的引用，编译器将会停止并生成错误。这些规则适用于 fn 定义，以及 impl 块：

1. 每一个是引用的参数都有它自己的生命周期参数。例如：fn foo<'a, 'b>(x: &'a i32, y: &'b i32)
2. 如果只有一个输入生命周期参数，那么它被赋予所有输出生命周期参数。例如：fn foo<'a>(x: &'a i32) -> &'a i32。
3. 如果方法有多个输入生命周期参数并且其中一个参数是 &self 或 &mut self，那么所有输出生命周期参数被赋予 self 的生命周期。

复杂的生命周期注解

下面是一个在同一函数中指定泛型类型参数、trait bounds 和生命周期的语法：

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

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

智能指针

智能指针是一类数据结构，他们的表现类似指针，但是也拥有额外的元数据和功能。普通引用和智能指针的一个额外的区别是引用是一类只借用数据的指针；相反，在大部分情况下，智能指针"拥有"他们指向的数据，当智能指针离开作用域时，可以选择是否销毁其指向的数据。

智能指针通常使用结构体实现。智能指针区别于常规结构体的显著特性在于其实现了 Deref 和 Drop trait。

- Deref trait 允许智能指针结构体实例表现的像引用一样，这样就可以编写既用于引用、又用于智能指针的代码。
- Drop trait 允许我们自定义当智能指针离开作用域时运行的代码。

标准库中最常用的智能指针有以下一些：

- Box<T>，用于在堆上分配值
- Rc<T>，一个引用计数类型，其数据可以有多个所有者
- Weak<T>，一个引用计数类型，其数据可以有多个所有者，和 Rc 的区别在于不保证引用数据仍然存在
- RefCell<T>，一个在运行时而不是在编译时执行借用规则的类型。

Deref Trait

实现 Deref trait 允许我们重载解引用运算符"*"。例如 Box<T>实现了 Deref trait，那么可以被如下使用：

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

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

当类型 T 实现了 Deref trait 时，当对类型&T 的值 v 进行*v 进行操作时，等效于*v.deref()。

下面是一个实现 Deref Trait 的示例：

```
use std::ops::Deref;

struct MyBox<T>(T);

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
    }
}

impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &T {
        &self.0
    }
}

fn main() {
    let x = 5;
    let y = MyBox::new(x);
```

```
    assert_eq!(5, x);
    assert_eq!(5, *y); // *y 实际执行为*(y.deref())
}
```

DerefMut trait

Rust 提供了 DerefMut trait 用于重载可变引用的 * 运算符。当类型 T 实现了 DerefMut trait 时，当对类型&mut T 的值 v 进行*v 进行操作时，等效于*v.deref_mut()

例子：

```
use std::ops::{Deref, DerefMut};

struct DerefMutExample<T> {
    value: T
}

impl<T> Deref for DerefMutExample<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        &self.value
    }
}

impl<T> DerefMut for DerefMutExample<T> {
    fn deref_mut(&mut self) -> &mut Self::Target {
        &mut self.value
    }
}

let mut x = DerefMutExample { value: 'a' };
*x = 'b';
assert_eq!('b', *x);
```

Drop Trait

Drop Trait，其允许我们在值要离开作用域时执行一些代码。可以为任何类型提供 Drop trait 的实现，同时所指定的代码被用于释放类似于文件或网络连接的资源。

例子：

```
struct CustomSmartPointer {
    data: String,
}

impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        println!("Dropping CustomSmartPointer with data `{}`!", self.data);
    }
}

fn main() {
    let c = CustomSmartPointer { data: String::from("my stuff") };
    let d = CustomSmartPointer { data: String::from("other stuff") };
    println!("CustomSmartPointers created.");
}
```

当我们希望在作用域结束之前就强制释放变量的话，我们应该使用的是由标准库提供的 std::mem::drop，而不是直接调用 drop 方法。Rust 不允许我们显式调用 drop 因为 Rust 仍然会在 main 的结尾对值自动调用 drop，这会导致一个 double free 错误，因为 Rust 会尝试清理相同的值两次。

std::mem::drop 位于 prelude，因此我们可以使用如下例子：

```
fn main() {
    let c = CustomSmartPointer { data: String::from("some data") };
    println!("CustomSmartPointer created.");
    drop(c);
    println!("CustomSmartPointer dropped before the end of main.");
```

```
}
```

解引用强制多态

解引用强制多态是 Rust 在函数或方法传参上的一种便利，其将实现了 Deref/DerefMut 的类型的引用转换为通过 Deref/DerefMut 所能够转换的类型的引用。这时会有一系列的 deref 方法被调用。

Rust 在发现类型和 trait 实现满足三种情况时会进行解引用强制多态：

- 当 T: Deref<Target=U> 时从 &T 到 &U。
- 当 T: DerefMut<Target=U> 时从 &mut T 到 &mut U。
- 当 T: Deref<Target=U> 时从 &mut T 到 &U。

例子：

```
fn main() {
    let m = MyBox::new(String::from("Rust"));
    hello(&m);
}
```

等价于：

```
fn main() {
    let m = MyBox::new(String::from("Rust"));
    hello(&(*m)[..]);
}
```

字面量

整数

Rust 中的整数字面值有如下一些：

整数字面值	例子
Decimal（十进制）	98_222
Hex（十六进制）	0xff
Octal（八进制）	0o77
Binary（二进制）	0b1111_0000
Byte（单字节字符）(仅限于 u8)	b'A'

Rust 的默认整数字面量在 32 位系统和 64 位系统上都是 i32 类型，除非添加了类型注解，或者使用类型后缀。

注意：除了 Byte 以外的所有整数字面值允许使用类型后缀，例如 57u8，同时也允许使用"_"做为分隔符以方便读数，例如 1_000

浮点数

Rust 的浮点数字面值有如下一些：3.5、27、-113.75、0.0078125、0、-1、1e-8。

Rust 的默认浮点数字面量在 32 位系统和 64 位系统上都是 f64 类型，除非添加了类型注解，或者使用类型后缀。

注意：浮点型字面值允许使用类型后缀，例如 1.0f32 或 1.0_f32 或 1e-8f32 或 1e-8_f32。

布尔值

Rust 中的布尔值字面量有两个可能的值：true 和 false。

字符

在 Rust 中，拼音字母（Accented letters），中文、日文、韩文等字符，emoji（绘文字）以及零长度的空白字符都是有效的 char 值。Unicode 标量值包含从 U+0000 到 U+D7FF 和 U+E000 到 U+10FFFF 在内的值。

Rust 中的字符有如下指定的方式：

```
let u = '\u{0065}';
let c = 'z';
let z = 'ℤ';
let heart_eyed_cat = '😻';
```

字符串

在 Rust 中，字符串是 str 类型，通常以&str 的方式出现。Rust 中的字符串都是有效的 utf-8 字符。

Rust 中的字符串有如下指定的方式：

```
let s = "Löwe 老虎 Léopard";
let s = "Hello, world!";
let s = " Mary   had\ta little  \n\t lamb";
let s: &'static str= "Hello, world!";
let s = r"\n"; // 原始字符串字面值
```

数据类型

在 Rust 中，每一个值都属于某一个数据类型（data type）。根据值及其使用方式，编译器通常可以推断出我们想要用的类型，但是当多种类型均有可能时，此时必须增加类型注解。例如：

```
let guess: u32 = "42".parse().expect("Not a number!");
```

标量类型

标量（scalar）类型代表一个单独的值。Rust 有四种基本的标量类型：整型、浮点型、布尔类型和字符类型。

整型

Rust 中的整型有如下一些：

长度	有符号	无符号
8-bit	i8	u8
16-bit	i16	u16
32-bit	i32	u32
64-bit	i64	u64
128-bit	i128	u128
arch(64 位架构 64-bit；32 位架构 32-bit)	isize	usize

注意：当出现整型溢出问题时，如果在 debug 模式编译时，Rust 检查整型溢出问题并使程序 panic，在 release 构建中，rust 会进行一种被称为二进制补码 wrapping，例如 256 赋值为 u8 时，变为 0。

浮点型

Rust 的浮点数类型是 f32 和 f64，分别占 32 位和 64 位。f32 是单精度浮点数，f64 是双精度浮点数。

此外，f32 和 f64 可以表示以下几个特殊值：

- -0：语义上等于 0，-0 == 0 为 true。
- INFINITY(∞)与 NEG_INFINITY(-∞)：可以由类似 1.0/0.0 的计算得到。
- NAN(Not a Number)：此值可以由类似(-1.0).sqrt()计算得到。NAN 与任何浮点数不等，包括自己，并且即不大于也不小于任何浮点数。

布尔型

Rust 中的布尔类型使用 bool 表示。

如果将布尔类型值转换为整数类型，则 true 为 1，false 为 0。例如：

```
assert_eq!(true as i32, 1);
assert_eq!(false as i32, 0);
```

字符类型

Rust 的 char 类型是语言中最原生的字母类型。char 由单引号指定，不同于字符串使用双引号。

Rust 的 char 类型的大小为四个字节(four bytes)，并代表了一个 Unicode 标量值（Unicode Scalar Value），这意味着它可以比 ASCII 表示更多内容。Unicode 标量值包含从 U+0000 到 U+D7FF 和 U+E000 到 U+10FFFF 在内的值。

注意：所有字符都是有效的 u32 类型，并且进行如下转换：

```
let c = '💯';
let i = c as u32;
assert_eq!(128175, i);
```

复合类型

复合类型（Compound types）可以将多个值组合成一个类型。Rust 有两个原生的复合类型：元组（tuple）和数组（array）。

数组类型

数组也可以包含多个值，数组中的每个元素的类型必须相同。数组类型声明如下：

```
let a: [i32; 5] = [1, 2, 3, 4, 5];
```

其中 i32 是每个元素的类型。数字 5 表明该数组包含五个元素。

Rust 中的数组是固定长度的，一旦声明，它们的长度不能增长或缩小。如果需要一个允许增长和缩小长度的类似数组的集合类型，那么可以使用标准库提供的 vector 类型。

数组中的值位于中括号内的逗号分隔的列表中，声明时类型注解可以省略。例如：

```
fn main() {
    let a = [1, 2, 3, 4, 5];
}
```

可以使用类似字符串字面量的方式创建数组：

```
let s = b"1234"; // s 类型为[u8;4]
let s = br"1234\n"; // 原始字符串内容的 u8 数组，s 类型为[u8;5]
```

如果要为每个元素创建包含相同值的数组，可以指定初始值，后跟分号，然后在方括号中指定数组的长度。例如：

```
let a = [3; 5];
// 等价 let a = [3, 3, 3, 3, 3]
```

可以使用索引来访问数组的元素。例如：

```
fn main() {
    let a = [1, 2, 3, 4, 5];
    let first = a[0];
    let second = a[1];
}
```

当使用无效索引访问时，会产生一个运行时错误。

元组类型

元组是一个将多个其他类型的值组合进一个复合类型的主要方式。元组长度固定：一旦声明，其长度不会增大或缩小。元组中的每一个位置都有一个类型，这些不同值的类型不必相同。元组类型声明如下：

```
let tup: (i32, f64, u8) = (500, 6.4, 1);
```

使用包含在圆括号中的逗号分隔的值列表来创建一个元组，声明时类型注解可以省略。例如：

```
fn main() {
    let tup = (500, 6.4, 1);
}
```

可以使用点号（.）后跟值的索引来直接访问它们。元组的第一个索引值是 0。例如：

```
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);
    let five_hundred = x.0;
    let six_point_four = x.1;
    let one = x.2;
}
```

对于单一元素的元组，应该使用(expr,)的语法，因为(expr)表示为一个表达式。例如：

```
fn main() {
    let x: (i32,) = (500,);
    let five_hundred = x.0;
}
```

解构元组

可以使用模式匹配（pattern matching）来解构（destructure）元组值。例如：

```
fn main() {
    let tup = (500, 6.4, 1);
    let (x, y, z) = tup;
```

```
    println!("The value of y is: {}", y);
}
```

忽略剩余值

在解构元组时，通常需要指定和元组数量相同的个数的变量个数，但是也可以通过..忽略剩余值，而只解构关心位置的值。例如：

```
fn main() {
    let numbers = (2, 4, 8, 16, 32);

    match numbers {
        (first, .., last) => {
            println!("Some numbers: {}, {}", first, last);
        },
    }
}
```

元组入参

可以在函数参数中匹配元组，将传递给函数的元组拆分为值。例如：

```
fn print_coordinates(&(x, y): &(i32, i32)) {
    println!("Current location: ({}, {})", x, y);
}

fn main() {
    let point = (3, 5);
    print_coordinates(&point);
}
```

元组返回值

元组可以作为函数返回值，用以返回多个值。例如：

```
fn area(a: u32) -> (u32, u32) {
   (a, a)
}
```

结构体类型

结构体和元组类似。结构体的每一部分可以是不同类型。但不同于元组，结构体需要命名各部分数据以便能清楚的表明其值的意义。

使用 struct 关键字定义结构体。在大括号中，定义每一部分数据的名字和类型，称为字段（field）。例如：

```
struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}
```

实例化

通过为每个字段指定具体值来创建这个结构体的实例。创建一个实例需要以结构体的名字开头，接着在大括号中使用 key: value 键-值对的形式提供字段。例如：

```
let user1 = User {
    email: String::from("someone@example.com"),
    username: String::from("someusername123"),
    active: true,
    sign_in_count: 1,
};
```

注意：要为所有字段都提供具体值，rust 没有默认值。

字段初始化简写

在函数内实例化结构体时，如果函数的参数名与字段名都完全相同，我们可以使用字段初始化简写语法。例如：

```
fn build_user(email: String, username: String) -> User {
    User {
        email,
        username,
        active: true,
        sign_in_count: 1,
```

```
    }
}
```

结构体更新语法

使用旧实例的大部分值但改变其部分值来创建一个新的结构体实例通常是很有帮助的。这可以通过结构体更新语法（struct update syntax）实现。.. 语法指定了剩余未显式设置值的字段应有与给定实例对应字段相同的值。例如：

```
let user2 = User {
    email: String::from("another@example.com"),
    username: String::from("anotherusername567"),
    ..user1
};
```

字段访问

可以使用点号从结构体中获取某个特定的值。如果结构体的实例是 mut 的，我们可以使用点号为对应的字段赋值。例如：

```
let mut user1 = User {
    email: String::from("someone@example.com"),
    username: String::from("someusername123"),
    active: true,
    sign_in_count: 1,
};

user1.email = String::from("anotheremail@example.com");
```

注意：整个实例必须是可变的，Rust 并不允许只将某个字段标记为可变。

解构结构体

可以通过带有模式的 let 语句将结构体进行分解。例如：

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p = Point { x: 0, y: 7 };

    let Point { x: a, y: b } = p;
    assert_eq!(0, a);
    assert_eq!(7, b);
}
```

如果解构的变量名等同于字段名称，那么只需列出结构体字段的名称，则模式创建的变量会有相同的名称。例如：

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p = Point { x: 0, y: 7 };

    let Point { x, y } = p;
    assert_eq!(0, x);
    assert_eq!(7, y);
}
```

另外可以解构结构体和元组的复杂类型。例如：

```
let ((feet, inches), Point {x, y}) = ((3, 10), Point { x: 3, y: -10 });
```

忽略剩余值

在解构结构体时，通常需要指定结构体的每个字段值，但是也可以通过..忽略不关心的字段值。例如：

```
struct Point {
    x: i32,
    y: i32,
    z: i32,
}
```

```
let origin = Point { x: 0, y: 0, z: 0 };

match origin {
    Point { x, .. } => println!("x is {}", x),
}
```

元组结构体

可以定义与元组类似的结构体，称为元组结构体（tuple structs）。元组结构体有着结构体名称提供的含义，但没有具体的字段名，只有字段的类型。例如：

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

let black = Color(0, 0, 0);
let origin = Point(0, 0, 0);
```

元组结构体实例类似于元组：可以将其解构为单独的部分，也可以使用.后跟索引来访问单独的值，等等。

类单元结构体

可以定义一个没有任何字段的结构体！它们被称为类单元结构体（unit-like structs）因为它们类似于()，即 unit 类型。例如：

```
struct Color;
```

类单元结构体常常在你想要在某个类型上实现 trait 但不需要在类型中存储数据的时候发挥作用。

枚举类型

枚举允许你通过列举可能的成员（variants） 来定义一个类型。

使用 enum 关键字定义枚举。在大括号中，定义每一个枚举值。例如：

```
enum IpAddrKind {
    V4,
    V6,
}

let four = IpAddrKind::V4;
let six = IpAddrKind::V6;
```

定义后，IpAddrKind 就是一个可以在代码中使用的自定义数据类型了。枚举的成员位于其标识符的命名空间中，并使用两个冒号分开。

注意：枚举类型也可以定义方法，其方式同结构体。

关联数据

可以将数据直接放进枚举成员，使得数据和枚举相关联。例如：

```
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}

let home = IpAddr::V4(127, 0, 0, 1);
let loopback = IpAddr::V6(String::from("::1"));
```

直接将数据附加到枚举的每个成员上。

枚举成员中内嵌了多种多样的类型，例如：

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

这个枚举有四个含有不同类型的成员：

- Quit 没有关联任何数据。
- Move 包含一个匿名结构体。
- Write 包含单独一个 String。
- ChangeColor 包含三个 i32。

关联数据的值提取可以参看解构枚举。

Option<T>

Rust 没有空值，不过拥有一个可以编码存在或不存在概念的枚举。这个枚举是 Option<T>，Option<T> 枚举被包含在了 prelude 之中。其定义如下：

```
enum Option<T> {
    Some(T),
    None,
}

let some_number = Some(5);
let some_string = Some("a string");
let absent_number: Option<i32> = None;
```

Option<T>有多种方法提取其中的 T 值，具体看这里。

Slice 类型

slice 类型允许你引用集合中一段连续的元素序列，而不用引用整个集合。内置的 slice 类型有字符串 slice(str)和数组 slice([T])。

可以使用一个由中括号中的 [starting_index..ending_index] 指定的 range 创建一个 slice，其中 starting_index 是 slice 的第一个位置，ending_index 则是 slice 最后一个位置的后一个值。

如果想要从第一个索引 0 开始，可以不写 starting_index，如[..ending_index]。
如果包含最后一个索引值，也可以舍弃 ending_index，如[starting_index..]。
如果想要引用整个集合，可以舍弃 starting_index 和 ending_index，如[..]。

注意：slice 的使用方式和其他语言的 slice 不同。

例子：

```
let s: String = String::from("hello world");
let world: &str = &s[6..11];

let a: [i32; 5] = [1, 2, 3, 4, 5];
let slice: &[i32] = &a[1..3];
```

字符串 slice

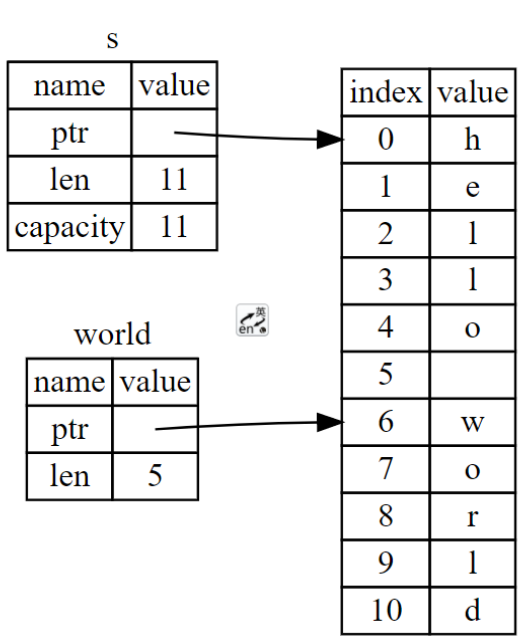
字符串 slice 是 String 中一部分值的引用，其数据类型为 str，以&str 借用的方式出现(因为其是动态大小类型)。

slice 的数据结构存储了 slice 的开始位置和长度。例如：

```
let s = String::from("hello world");
let world = &s[6..11];
```

数据结构如下所示：

s



字符串 slice 的使用方式查看这里。另外字符串 slice 不支持索引的访问方式，因为 unsize 类型没有实现 std::slice::SliceIndex <str>。

注意：字符串 slice range 的索引必须位于有效的 UTF-8 字符边界内，如果尝试从一个多字节字符的中间位置创建字符串 slice，则程序将会因错误而退出。

数组 slice

数组 slice 是数组中一部分值的引用，其数据类型为[T]，通常以&[T]借用的方式出现(因为其是动态大小类型)。

数组 slice 跟字符串 slice 的工作方式一样，通过存储第一个集合元素的引用和一个集合总长度。

数组 slice 的使用方式查看这里。数组 slice 支持索引的访问方式，例如：

```
    let a: [i32; 5] = [1, 2, 3, 4, 5];
    let slice: &[i32] = &a[1..3];
    let i: i32 = slice[1]
```

函数指针类型

通过函数指针允许我们使用函数作为另一个函数的参数。函数的类型是 fn，指定参数为函数指针的语法类似于闭包。例子：

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

fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}

fn main() {
    let answer = do_twice(add_one, 5);

    println!("The answer is: {}", answer);
}
```

函数指针实现了所有三个闭包 trait（Fn、FnMut 和 FnOnce），所以总是可以在调用期望闭包的函数时传递函数指针作为参数。

字符串 str

Rust 的核心语言中只有一种字符串类型：str，字符串 slice，它以被借用的形式出现(因为其是动态大小类型)，&str。详见字符串 slice。

String 类型是由标准库提供的，而没有写进核心语言部分，它是可增长的、可变的、有所有权的、UTF-8 编码的字符串类型。

字符串类型 String 的使用方式看这里。

新建字符串

可以使用 new 函数创建一个空字符串，例如：

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

可以用任何实现了 Display trait 的类型的 to_string 方法，字符串字面值也实现了它。例如：

```
let s = "initial contents".to_string();
```

可以使用 String::from 函数来从字符串字面值创建 String。例如：

```
let s = String::from("initial contents");
```

更新字符串

有以下几种方法可以更新字符串：

- 使用 push_str 方法向 String 附加字符串 slice：

  ```
  let mut s = String::from("foo");
  s.push_str("bar");
  ```

- 使用 push 方法将字母加入 String：

  ```
  let mut s = String::from("lo");
  s.push('l');
  ```

- 使用 + 运算符拼接字符串：

  ```
  let s1 = String::from("tic");
  let s2 = String::from("tac");
  let s3 = String::from("toe");
  let s = s1 + "-" + &s2 + "-" + &s3; // 注意 s1 被移动了，不能继续使用
  ```

  + 运算符使用了 add 函数，这个函数签名看起来像这样，因此所有权进行了转移：

  ```
  fn add(self, s: &str) -> String {
  ```

- 可以使用 format! 宏拼接字符串：

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

let s = format!("{}-{}-{}", s1, s2, s3);
```

format 宏不会获取任何参数的所有权。

索引字符串

在 Rust 中，不支持索引字符串，包括 String 类型或者时 str 类型。因为字符串索引应该返回的类型是不明确的：字节值、字符、字形簇或者字符串 slice。

Vec

vector 可以认为是可变长度的数组，允许我们在一个单独的数据结构中储存多于一个的值，它在内存中彼此相邻地排列所有的值。vector 只能储存相同类型的值。

Vector 的使用方式看这里。

新建 vector

调用 Vec::new 函数可以创建一个新的 vector。例如：

```
let v: Vec<i32> = Vec::new();
```

注意：这里必须增加一个类型注解。

另外可以使用 vec!宏创建一个新的 vector。例如：

```
let v = vec![1, 2, 3];
```

注意：使用 vec!宏不需要类型注解，因为 Rust 可以根据初始值的类型推断出 v 的类型。

更新 vector

可以使用 push 方法向 vector 新增元素。例如：

```
let mut v = Vec::new();

v.push(5);
v.push(6);
v.push(7);
v.push(8);
```

读取 vector 元素

有两种方法引用 vector 中储存的值：

- 使用索引语法访问。该方式返回一个引用。例如：

```
let v = vec![1, 2, 3, 4, 5];
let third: &i32 = &v[2];
```

- 使用 get 方法访问。该方式返回一个 Option<&T>。例如：

```
let v = vec![1, 2, 3, 4, 5];
match v.get(2) {
    Some(third) => println!("The third element is {}", third),
    None => println!("There is no third element."),
}
```

两种访问方式的区别不仅在于返回值类型不同，当使用索引语法引用一个不存在的元素时 Rust 会造成 panic，而 get 方法仅是返回 None。

遍历 vector 元素

vector 实现了 IntoIterator trait，因此可以直接使用，来进行遍历 vector 元素。例子：

```
let v = vec![100, 32, 57];
for i in &v {
    println!("{}", i);
}
```

对于要改变数组元素，则可以添加 mut。例子：

```
let mut v = vec![100, 32, 57];
```

```
for i in &mut v {
    *i += 50;
}
```

HashMap

HashMap<K, V> 类型储存了一个键类型 K 对应一个值类型 V 的映射。

HashMap 默认使用一种“密码学安全的”哈希函数，它可以抵抗拒绝服务攻击。然而这并不是可用的最快的算法，不过为了更高的安全性值得付出一些性能的代价。如果性能监测显示此哈希函数非常慢，以致于你无法接受，你可以指定一个不同的 hasher 来切换为其它函数。这里有一些已经实现的 hasher。

Map 的使用方式看这里。

新建 map

可以使用 new 创建一个空的 HashMap。例子：

```
use std::collections::HashMap;
let mut scores = HashMap::new();;
```

可以使用一个元组的 vector 的 collect 方法创建一个带有初始值的 HashMap，其中每个元组包含一个键值对。例子：

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

let teams  = vec![String::from("Blue"), String::from("Yellow")];
let initial_scores = vec![10, 50];

let scores: HashMap<_, _> = teams.iter().zip(initial_scores.iter()).collect();
```

更新 map

可以使用 insert 增加元素。例子：

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

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);
```

注意：使用 insert 方法时，如果键已经存在，则会覆盖原来的值。

也可以使用 entry 获取键的条目状况，然后调用其 or_insert 方法选择性插入，并返回其值。例如：

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

let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);

scores.entry(String::from("Yellow")).or_insert(50);
scores.entry(String::from("Blue")).or_insert(50);

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

注意：or_insert 方法返回的时指定键的值的可变引用，可以直接修改该键的值。例如：

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

let text = "hello world wonderful world";

let mut map = HashMap::new();

for word in text.split_whitespace() {
    let count = map.entry(word).or_insert(0);
    *count += 1;
}

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

读取 map 元素

可以通过 get 方法并提供对应的键来从哈希 map 中获取值。例如：

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

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

let team_name = String::from("Blue");
let score = scores.get(&team_name);
```

注意：get 方法返回的是 Option<V>。

遍历 map 元素

HashMap 实现了 IntoIterator trait，因此可以直接使用，来进行遍历其中元素。例子：

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

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

for (key, value) in &scores {
    println!("{}: {}", key, value);
}
```

对于要改变 hash 的键的值，则可以添加 mut。例子：

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

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

for (key, value) in &mut scores {
    *value = value*value;
    println!("{}: {}", key, value);
}
```

Box

Box 是最简单的智能指针，类型是 Box<T>，它允许你将一个值放在堆上而不是栈上。Box 通常被应用在以下场景：

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

注意：Box 具有单一读的特点，无法修改指针指向的数据值，无法进行共享读。

创建 Box

使用 Box::new 关联函数可以创建一个 box。例如：

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

递归类型

Rust 需要在编译时知道类型占用多少空间，因此如果计算中出现类型递归，那么将无法得到最终存储类型的值所需的大小。有两种方式可以打破这种递归：1、使用类似 box 的智能指针；2、使用引用。

下面是使用 box 打破类型递归的示例：

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}
```

```
use crate::List::{Cons, Nil};

fn main() {
    let list = Cons(1,
        Box::new(Cons(2,
            Box::new(Cons(3,
                Box::new(Nil))))));
}
```

Rc

使用 box 创建的智能指针，只允许堆上的数据被一个人拥有，在 box 离开作用域时，其内部指向的堆上的数据即被释放。

为了允许堆上的数据允许被多个人拥有，rust 提供了 rc 类型，在 rc 离开作用域时，仅会减少拥有者计数，如果拥有者为 0 时，将释放堆上的数据。通过这种方式，允许多个 rc 共享内部的数据。

注意：Rc 只能用于单线程场景。Rc 具有共享读的特点，但是无法修改指针指向的数据值。

创建 Rc

通过 Rc::new 关联函数，可以创建 rc 智能指针。例如：

```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
}
```

共享 Rc

通过 Rc::clone 关联函数，可以创建一个共享智能指针的共享副本。这将会增加指针内部的引用计数。例如：

```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
}
```

RefCell

RefCell 允许在有不可变引用时也可以改变内部数据，并且其和 Box 一样，具有数据的唯一的所有权。

RefCell、引用、Box 具有以下不同：

- 对于引用和 Box，借用规则的不可变性作用于编译时。对于 RefCell，借用规则的不可变性作用于运行时。
- 对于引用，如果违反这些规则，会得到一个编译错误。对于 RefCell<T>，如果违反这些规则程序会 panic 并退出。

注意：RefCell 只能用于单线程场景。RefCell 具有单一读写的特点，允许修改指针指向的值，但是无法共享读。但是可以结合 Rc 和 RefCell 来支持共享读、单一写的情况。

创建 RefCell

通过 RefCell::new 关联函数，可以创建 RefCell。例如：

```
#[derive(Debug)]
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::rc::Rc;
use std::cell::RefCell;

fn main() {
    let value = Rc::new(RefCell::new(5));

    let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));

    let b = Cons(Rc::new(RefCell::new(6)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(10)), Rc::clone(&a));
}
```

记录借用

对于普通变量，创建不可变借用和可变借用时，分别使用 & 和 &mut 语法。对于 RefCell<T> 来说，则是 borrow 和 borrow_mut 方法创建不可变借用和可变借用。

borrow 方法返回 Ref<T> 类型的智能指针，borrow_mut 方法返回 RefMut<T> 类型的智能指针，这两个类型都实现了 Deref，可以当作常规引用对待。

RefCell<T> 记录当前有多少个活动的 Ref<T> 和 RefMut<T> 智能指针。每次调用 borrow，RefCell<T> 将活动的不可变借用计数加一。当 Ref<T> 值离开作用域时，不可变借用计数减一。RefCell<T> 在任何时候只允许有多个不可变借用或一个可变借用。

例子：

```
pub trait Messenger {
    fn send(&self, msg: &str);
}

struct MockMessenger {
    sent_messages: RefCell<Vec<String>>,
}

impl MockMessenger {
    fn new() -> MockMessenger {
        MockMessenger { sent_messages: RefCell::new(vec![]) }
    }
}

impl Messenger for MockMessenger {
    fn send(&self, message: &str) {
        let mut borrow = self.sent_messages.borrow_mut();
        borrow.push(String::from(message));
    }
}
```

Rc<RefCell>

通过使用 Rc<RefCell<T>>的方式，可以得到一个允许修改指针指向的值并且可以共享读的智能指针。

使用共享可变的智能指针，需要特别注意，在上一个 RefMut 离开作用域前，不能再进行 borrow_mut，或者 borrow。

例子：

```
#[derive(Debug)]
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
```

```
}

use crate::List::{Cons, Nil};
use std::rc::Rc;
use std::cell::RefCell;

fn main() {
    let value = Rc::new(RefCell::new(5));

    let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));

    let b = Cons(Rc::new(RefCell::new(6)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(10)), Rc::clone(&a));

    *value.borrow_mut() += 10;

    println!("a after = {:?}", a);
    println!("b after = {:?}", b);
    println!("c after = {:?}", c);
}
```

Weak

在使用智能指针时，很容易出现循环引用，而 rust 编译器认为循环引用导致的内存泄露是内存完全的。

使用 Weak 可以打破循环引用。当调用 Rc::clone 会增加 Rc<T> 实例的 strong_count，在 strong_count 为 0 时， Rc<T> 的实例会被清理，当调用 Rc::downgrade 并传递 Rc<T> 实例的引用，可以创建 Rc<T> 实例的弱引用 Weak<T>。不同于调用 Rc::clone 时将 Rc<T> 实例的 strong_count 加 1，调用 Rc::downgrade 会将 weak_count 加 1。即使 weak_count 不为 0，strong_count 为 0 时，Rc<T> 的实例会被清理。

如下是一个循环引用例子：

```
#[derive(Debug)]
enum List {
    Cons(i32, RefCell<Rc<List>>),
    Nil,
}

impl List {
    fn tail(&self) -> Option<&RefCell<Rc<List>>> {
        match self {
            Cons(_, item) => Some(item),
            Nil => None,
        }
    }
}

fn main() {
    let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));

    println!("a initial rc count = {}", Rc::strong_count(&a));
    println!("a next item = {:?}", a.tail());

    let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a)))); // b 指向 a

    println!("a rc count after b creation = {}", Rc::strong_count(&a));
    println!("b initial rc count = {}", Rc::strong_count(&b));
    println!("b next item = {:?}", b.tail());

    if let Some(link) = a.tail() {
        *link.borrow_mut() = Rc::clone(&b); // 将 a 指向 b，造成循环引用
    }

    println!("b rc count after changing a = {}", Rc::strong_count(&b));
    println!("a rc count after changing a = {}", Rc::strong_count(&a));

    // Uncomment the next line to see that we have a cycle;
```

```
    // it will overflow the stack
    // println!("a next item = {:?}", a.tail());
}
```

创建 Weak

通过 Weak::new 关联函数，可以创建一个未引用的 Weak 引用，如果调用 upgrade 方法，将会返回 None。例子：

```
use std::rc::Weak;

let empty: Weak<i64> = Weak::new();
assert!(empty.upgrade().is_none());
```

通过调用 Rc::downgrade 关联函数，并传递 Rc<T> 实例的引用可以创建一个弱引用。例子：

```
#[derive(Debug)]
struct Node {
    value: i32,
    parent: RefCell<Weak<Node>>,
    children: RefCell<Vec<Rc<Node>>>,
}

fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());

    let branch = Rc::new(Node {
        value: 5,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });

    *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
}
```

使用 Weak

由于即使 weak_count 不为 0，strong_count 为 0 时，Rc<T> 的实例也会被清理，因此 Weak<T>中的引用数据不一定存在。为此可以调用 Weak<T> 实例的 upgrade 方法，这会返回 Option<Rc<T>>。如果 Rc<T> 值还未被丢弃，则结果是 Some；如果 Rc<T> 已被丢弃，则结果是 None。

例子：

```
#[derive(Debug)]
struct Node {
    value: i32,
    parent: RefCell<Weak<Node>>,
    children: RefCell<Vec<Rc<Node>>>,
}

fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());

    let branch = Rc::new(Node {
        value: 5,
        parent: RefCell::new(Weak::new()),
```

```
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });

    *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
}
```

类型别名

通过使用 type 关键字来给予现有类型另一个名字。例如：

```
type Kilometers = i32;

let x: i32 = 5;
let y: Kilometers = 5;

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

类型别名的主要用途时用于提供一个可读性的名称，以及简化很复杂的泛型类型。例如：

```
type Thunk = Box<dyn Fn() + Send + 'static>;

let f: Thunk = Box::new(|| println!("hi"));

fn takes_long_type(f: Thunk) {
    // --snip--
}

fn returns_long_type() -> Thunk {
    // --snip--
}
```

强制类型转换

通过使用 as 关键字可以进行强制类型转换。例如：

```
assert_eq!(true as i32, 1);
assert_eq!(false as i32, 0);
```

动态大小类型

Rust 在编译时，通常需要知道类型的大小，而动态大小类型是只有在运行时才知道大小的类型。str 是动态大小类型。对于&T 是一个储存了 T 所在的内存位置的单个值，而&str 则是两个值：str 的地址和其长度。&str 在编译时的大小是 usize 长度的两倍。

Rust 中的 Sized trait 决定一个类型的大小是否在编译时可知。这个 trait 自动为编译器在编译时就知道大小的类型实现。

Rust 隐式的为每一个泛型函数增加了 Sized bound。对于如下泛型函数定义：

```
fn generic<T>(t: T) {
    // --snip--
}
```

实际处理为：

```
fn generic<T: Sized>(t: T) {
    // --snip--
}
```

泛型函数默认只能用于在编译时已知大小的类型。然而可以使用如下特殊语法来放宽这个限制：

```
fn generic<T: ?Sized>(t: &T) {
    // --snip--
}
```

?Sized trait bound 可以读作 “T 可能是也可能不是 Sized 的”， 这个语法只能用于 Sized trait。

注意：对于动态大小类型，只能使用引用，因为其大小是编译时未知的。

泛型

泛型是具体类型或其他属性的抽象替代。可以使用泛型为像函数、结构体、枚举和方法的项创建定义。

函数泛型

当使用泛型定义函数时，在函数签名中指定参数和返回值的类型的地方改用泛型来表示。例如：

```
fn largest<T>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

函数泛形通过 fn::<type1,...>的方式可以显式指定类型：

```
let max = largest::<u8>(list)
```

结构体泛型

可以用 <> 语法来定义结构体，它包含一个或多个泛型参数类型字段，必须在结构体名称后面的尖括号中声明泛型参数的名称，接着在结构体定义中可以指定具体数据类型的位置使用泛型类型。例如：

```
struct Point<T, U> {
    x: T,
    y: U,
}

fn main() {
    let both_integer = Point { x: 5, y: 10 };
    let both_float = Point { x: 1.0, y: 4.0 };
    let integer_and_float = Point { x: 5, y: 4.0 };
}
```

结构体通过 struct_name<type1,...>的方式可以显式的指定类型：

```
let both_integer = Point<u8,u8> { x: 5, y: 10 };
```

枚举泛型

枚举可以在成员中存放泛型数据类型。其语法类似结构体。例如：

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

枚举通过 enum_name<type1,...>的方式可以显式的指定类型：

```
let result: Result<u8, String> = Ok(0)
```

方法泛型

泛型方法可以分为以下三种：

- 为泛型结构体或者枚举实现的方法。
- 为普通结构体或者枚举实现的泛型方法。
- 为泛型结构体或者枚举实现的泛型方法。

为泛型结构体/枚举实现的方法

为泛型结构体或者枚举实现方法时，必须在 impl 后面声明 T，这样 Rust 就知道结构体中的尖括号中的类型是泛型而不是具体类型。例如：

```
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
```

```
}
```

为普通结构体/枚举实现的泛型方法

为普通结构体或者枚举实现的泛型方法类似于泛型函数。例如：

```
struct Point {
    x: f32,
    y: f32,
}

impl Point {
    fn add<T>(&self, add: T) {
        self.x+=add
        self.y+=add
    }
}
```

泛型方法的显式调用方式和泛型函数是一样的。

为泛型结构体/枚举实现的泛型方法

结构体定义中的泛型类型参数并不总是与结构体方法签名中使用的泛型是同一类型。例如：

```
struct Point<T, U> {
    x: T,
    y: U,
}

impl<T, U> Point<T, U> {
    fn mixup<V, W>(self, other: Point<V, W>) -> Point<T, W> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}
```

泛型方法的显式调用方式和泛型函数是一样的。

为特定泛型结构体/枚举实现方法

另外可以为特定类型的泛型结构体或者枚举实现方法。例如：

```
struct Point<T> {
    x: T,
    y: T,
}

impl Point<f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}
```

此例表示只有 Point<f32>类型才有 distance_from_origin 方法。

配合 trait bound，可以为实现了特定 trait 的类型实现方法。例如：

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

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

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

```
        }
    }
}

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

trait

通过 trait 以一种抽象的方式定义共享的行为。然后通过 trait bounds 指定泛型拥有哪些特定行为。

定义 trait

一个类型的行为由其可供调用的方法构成。如果可以对不同类型调用相同的方法的话，这些类型就可以共享相同的行为了。trait 定义是一种将方法签名组合起来的方法，目的是定义一个实现某些目的所必需的行为的集合。类似于其他语言中的常被称为接口的功能。

用 trait 关键字来声明一个 trait，后面是 trait 的名字，然后在大括号中声明描述实现这个 trait 的类型所需要的行为的方法签名，在方法签名后跟分号，而不是在大括号中提供其实现。例如：

```
pub trait Summary {
    fn summarize(&self) -> String;
}
```

trait 体中可以有多个方法：一行一个方法签名且都以分号结尾。

实现 trait

在 impl 关键字之后，提供需要实现 trait 的名称，接着是 for 和需要实现 trait 的类型的名称。可以为任意类型实现 trait，而不仅限于结构体和枚举类型。

例如为结构体 NewsArticle 实现 Summary trait：

```
pub trait Summary {
    fn summarize(&self) -> String;
}

pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}, by {} ({})", self.headline, self.author, self.location)
    }
}
```

注意：实现 trait 时，只有当 trait 或者要实现 trait 的类型位于 crate 的本地作用域时，才能为该类型实现 trait。没有这条规则的话，两个 crate 可以分别对相同类型实现相同的 trait，而 Rust 将无从得知应该使用哪一个实现。这条规则确保了其他人编写的代码不会破坏你代码。

泛型类型实现 trait

为泛型类型实现 trait 和普通的实现 trait 类似。例如：

```
pub trait Summary {
    fn summarize(&self) -> String;
}

pub struct NewsArticle<T> {
    pub headline: T,
```

```
    pub location: T,
    pub author: T,
    pub content: T,
}

impl<T> Summary for NewsArticle<T> {
    fn summarize(&self) -> String {
        format!("{}, by {} ({})", self.headline, self.author, self.location)
    }
}
```

特定泛型类型实现 trait

使用 trait bound 可以为实现了指定 trait 的泛型类型实现 trait 的方法。

例如为实现了 Display trait 的泛型类型 T，实现 ToString trait 中的方法：

```
impl<T: Display> ToString for T {
    // --snip--
}
```

默认实现

可以为 trait 中的某些或全部方法提供默认的行为，这样就不用在每个类型的每个实现中都定义自己的行为。例如：

```
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more...)")
    }
}

pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle {
}
```

默认实现允许调用相同 trait 中的其他方法，哪怕这些方法没有默认实现。例如：

```
pub trait Summary {
    fn summarize_author(&self) -> String;

    fn summarize(&self) -> String {
        format!("(Read more from {}...)", self.summarize_author())
    }
}

pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summary for Tweet {
    fn summarize_author(&self) -> String {
        format!("@{}", self.username)
    }
}
```

trait 参数

可以使用 impl Trait 语法来使得函数接受多种不同类型的参数。例如：

```
pub trait Summary {
    fn summarize(&self) -> String;
```

```
}

pub fn notify(item: impl Summary) {
    println!("Breaking news! {}", item.summarize());
}
```

如果需要同时实现两个不同的 trait，可以通过"+"语法实现。例如：

```
pub fn notify(item: impl Summary + Display) {
 ...
}
```

注意：impl Trait 语法的参数的本质为编译器在编译时支持不同类型参数。而 trait 对象为运行时支持不同类型参数。

trait 返回值

可以使用 impl Trait 语法，来返回实现了某个 trait 的类型。例如：

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

和 trait 参数一样如果需要同时实现两个不同的 trait，可以通过"+"语法实现。

虽然返回的是实现了某个 trait 的类型，但是只能返回单一类型。impl Trait 语法的参数的本质为编译器在编译时支持不同类型参数。例如：

```
fn returns_summarizable(switch: bool) -> impl Summary {
    if switch {
        NewsArticle {
            headline: String::from("Penguins win the Stanley Cup Championship!"),
            location: String::from("Pittsburgh, PA, USA"),
            author: String::from("Iceburgh"),
            content: String::from("The Pittsburgh Penguins once again are the best
            hockey team in the NHL."),
        }
    } else {
        Tweet {
            username: String::from("horse_ebooks"),
            content: String::from("of course, as you probably already know, people"),
            reply: false,
            retweet: false,
        }
    }
}
```

如果需要返回不同的类型，可以使用 trait 对象方式。

trait bound

impl Trait 语法适用于直观的例子，它不过是一个较长形式的语法糖，这被称为 trait bound。impl Trait 很方便，适用于短小的例子，trait bound 则适用于更复杂的场景。trait bound 例子：

```
pub fn notify<T: Summary>(item: T) {
    println!("Breaking news! {}", item.summarize());
}
```

同样，可以通过 "+" 指定多个 trait bound。例如：

```
pub fn notify<T: Summary + Display>(item: T) {
}
```

当存在过多的 trait bound 时，每个泛型有其自己的 trait bound，所以有多个泛型参数的函数在名称和参数列表之间会有很长的 trait bound 信息，这使得函数签名难以阅读。因此可以使用 where 从句中指定 trait bound 的语法。例如：

```
fn some_function<T: Display + Clone, U: Clone + Debug>(t: T, u: U) -> i32 {
}

// 等价于
```

```
fn some_function<T, U>(t: T, u: U) -> i32
    where T: Display + Clone,
          U: Clone + Debug {
}
```

对于具有泛型参数的 trait 进行泛型参数 bound 时，需要指明 trait 种的泛型参数：

```
trait Shape<T> {
    fn draw(&self, T);
    fn name() -> &'static str;
}

fn draw_twice<T: Shape<Surface>>(surface: Surface, sh: T) {
    sh.draw(surface);
    sh.draw(surface);
}
```

对于具有关联类型的 trait 进行泛型参数 bound 时，需要使用如下方式指明关联类型：

```
pub trait FnOnce<Args> {
    type Output;
    extern "rust-call" fn call_once(self, args: Args) -> Self::Output;
}

fn f<F : FnOnce() -> String> (g: F) {
    println!("{}", g());
}
```

"FnOnce() -> String"类似函数签名，入参为泛型参数，返回值为 trait 关联类型。

trait 对象

泛型类型参数一次只能替代一个具体类型，而 trait 对象则允许在运行时替代多种具体类型。trait 对象指向一个实现了我们指定 trait 的类型的实例，以及一个用于在运行时查找该类型的 trait 方法的表。可以通过指定某种指针来创建 trait 对象，例如&引用或 Box<T>智能指针。

注意：只有对象安全的 trait 才可以组成 trait 对象。如果一个 trait 中所有的方法有如下属性时，则该 trait 是对象安全的：

- 返回值类型不为 Self。
- 方法没有任何泛型类型参数。
- dyn trait 对象是动态大小类型，因此必须使用引用&dyn trait 或者智能指针 Box<dyn trait>的方式。

例子。在使用泛型时，对于不同类型 T 得生成不同的 Screen 实例，这和预想的不同的类型 T 只生成一个 Screen 实例有区别：

```
pub trait Draw {
    fn draw(&self);
}

pub struct Button {
    pub width: u32,
    pub height: u32,
    pub label: String,
}

impl Draw for Button {
    fn draw(&self) {
        // 实际绘制按钮的代码
    }
}

struct SelectBox {
    width: u32,
    height: u32,
    options: Vec<String>,
}

impl Draw for SelectBox {
    fn draw(&self) {
        // code to actually draw a select box
    }
}
```

```
pub struct Screen<T: Draw> {
    pub components: Vec<T>,
}

impl<T> Screen<T>
    where T: Draw {
    pub fn run(&self) {
        for component in self.components.iter() {
            component.draw();
        }
    }
}

fn main() {
    let screen = Screen {
        components: vec![
            SelectBox {
                width: 75,
                height: 10,
                options: vec![
                    String::from("Yes"),
                    String::from("Maybe"),
                    String::from("No")
                ],
            },
        ],
    };

    screen.run();

    let screen = Screen {
        components: vec![
            Button {
                width: 50,
                height: 10,
                label: String::from("OK"),
            },
        ],
    };

    screen.run();

}
```

可以通过 dyn 关键字，后跟随 trait 名称表示这是一个 trait 对象。例如：

```
pub trait Draw {
    fn draw(&self);
}

pub struct Button {
    pub width: u32,
    pub height: u32,
    pub label: String,
}

impl Draw for Button {
    fn draw(&self) {
        // 实际绘制按钮的代码
    }
}

struct SelectBox {
    width: u32,
    height: u32,
    options: Vec<String>,
```

```
}

impl Draw for SelectBox {
    fn draw(&self) {
        // code to actually draw a select box
    }
}

pub struct Screen {
    pub components: Vec<Box<dyn Draw>>,
}

impl Screen {
    pub fn run(&self) {
        for component in self.components.iter() {
            component.draw();
        }
    }
}

fn main() {
    let screen = Screen {
        components: vec![
            Box::new(SelectBox {
                width: 75,
                height: 10,
                options: vec![
                    String::from("Yes"),
                    String::from("Maybe"),
                    String::from("No")
                ],
            }),
            Box::new(Button {
                width: 50,
                height: 10,
                label: String::from("OK"),
            }),
        ],
    };

    screen.run();
}
```

trait 依赖

如果一个 trait 需要使用另一个 trait 的功能，在这种情况下，需要被依赖 trait 也被实现。这个被依赖的 trait 就是我们实现的 trait 的父 trait。

例子：

```
use std::fmt;

trait OutlinePrint: fmt::Display {
    fn outline_print(&self) {
        let output = self.to_string();
        let len = output.len();
        println!("{}", "*".repeat(len + 4));
        println!("*{}*", " ".repeat(len + 2));
        println!("* {} *", output);
        println!("*{}*", " ".repeat(len + 2));
        println!("{}", "*".repeat(len + 4));
    }
}
```

同名冲突

Rust 既不能避免一个 trait 与另一个 trait 拥有相同名称的方法，也不能阻止为同一类型同时实现这两个 trait。不过，当调用这些同名

方法时，需要告诉 Rust 我们希望使用哪一个。当同一作用域的两个类型实现了同一 trait，Rust 就不能计算出我们期望的是哪一个类型，但是可以使用完全限定语法来指定使用哪一个方法。

例子：

```
trait Pilot {
    fn fly(&self);
}

trait Wizard {
    fn fly(&self);
}

struct Human;

impl Pilot for Human {
    fn fly(&self) {
        println!("This is your captain speaking.");
    }
}

impl Wizard for Human {
    fn fly(&self) {
        println!("Up!");
    }
}

impl Human {
    fn fly(&self) {
        println!("*waving arms furiously*");
    }
}

fn main() {
    let person = Human;
    Pilot::fly(&person);
    Wizard::fly(&person);
    person.fly();
}
```

关联类型

关联类型是一个将类型占位符与 trait 相关联的方式，这样 trait 的方法签名中就可以使用这些占位符类型。例如：

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
}
```

如果关联类型存在 bound，则使用 where 的 bound 方式，例如：

```
pub trait Iterator {
    type Item: Sized;

    fn next(&mut self) -> Option<Self::Item>;
}
```

关联类型和泛型的显著区别在于，一个类型可以实现多个使用的泛型的 trait，但是只能实现一个关联类型的 trait。例如：

```
pub trait Iterator<T> {
    fn next(&mut self) -> Option<T>;
}
```

可以实现为 impl Iterator<String> for Counter，或任何其他类型，这样就可以有多个 Counter 的 Iterator 的实现。

具有关联类型的 trait 实现方式如下：

```
impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
```

```
        ...
    }
}
```

Self 类型

在 trait 中可以使用 Self 指代实现了 trait 类型的类型。例如：

```
pub trait Draw {
    fn draw(&self) Self;
}
```

默认泛型类型参数

当在 trait 中使用泛型类型参数时，可以为泛型指定一个默认的具体类型。当实现未指定泛型类型时，则泛型使用默认类型。

默认参数类型主要用于如下两个方面：

- 扩展类型而不破坏现有代码。
- 在大部分用户都不需要的特定情况进行自定义。

例子：

```
trait Add<T=Self> {
    type Output;

    fn add(self, rhs: T) -> Self::Output;
}

#[derive(Debug, PartialEq)]
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() {
    assert_eq!(Point { x: 1, y: 0 } + Point { x: 2, y: 3 },
               Point { x: 3, y: 3 });
}
```

变量

在 Rust 中，使用 let 关键字来创建变量。例如：

```
let foo = 1;
//新建了一个叫做 foo 的变量并把它绑定到值 1。
```

可以使用类型注解来指定变量的数据类型。例如：

```
let guess: u32 = 1;
```

注意：rust 中所有定义的变量如果未使用，会产生警告，但是可以在变量名前添加"_"来消除警告。例如：

```
fn main() {
    let _x = 5;
    let y = 10; // 会产生警告
}
```

常量

在 Rust 中，使用 const 关键字来创建常量。例如：

```
const MAX_POINTS: u32 = 100_000;
```

注意：

- 常量声明必须注明值的类型。
- 常量只能被设置为常量表达式。
- 常量可以在全局作用域中声明。

Rust 常量的命名规范是使用下划线分隔的大写字母单词，并且可以在数字字面值中插入下划线来提升可读性。

全局变量

全局变量在 Rust 中被称为 静态（static）变量。使用 static 关键字进行申明，并且必须标注值的类型。例如：

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

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

注意：

- 静态变量只能储存拥有 'static 生命周期的引用。
- 访问不可变静态变量是安全的。
- 访问和修改可变静态变量都是不安全的，需要使用 unsafe 代码块。

Rust 全局变量的命名规范是使用下划线分隔的大写字母单词，并且可以在数字字面值中插入下划线来提升可读性。

可变性

Rust 中的变量默认是不可改变的，即是无法修改该变量的值。例如：

```
fn main() {
    let x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
```

可以通过在变量名之前加 mut 来使其可变。例如：

```
fn main() {
    let mut x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
```

不可变变量与常量的区别：

- 不允许对常量使用 mut。常量不光默认不能变，它总是不能变。
- 声明常量使用 const 关键字而不是 let，并且必须注明值的类型。
- 常量可以在任何作用域中声明，包括全局作用域。
- 常量只能被设置为常量表达式，而不能是函数调用的结果，或任何其他只能在运行时计算出的值。

隐藏变量

在 Rust 中，可以用相同变量名称重复使用 let 关键字来隐藏一个变量，例如：

```
fn main() {
    let x = 5;
    let x = x + 1;
    let x = x * 2;
    println!("The value of x is: {}", x);

    let spaces = "   ";
    let spaces = spaces.len();
    println!("The value of spaces is: {}", spaces);
}
```

注意：当再次使用 let 时，实际上创建了一个新变量，我们可以改变值的类型，但复用这个名字。

忽略值

在 rust 中，可以使用"_"作为变量名，表示忽略值。

忽略变量入参值：

```
fn foo(_: i32, y: i32) {
    println!("This code only uses the y parameter: {}", y);
}

fn main() {
    foo(3, 4);
}
```

忽略元组的部分值：

```
let (x, y, _) = (1, 2, 3);
```

忽略枚举匹配值：

```
let mut setting_value = Some(5);
let new_setting_value = Some(10);

match (setting_value, new_setting_value) {
    (Some(_), Some(_)) => {
        println!("Can't overwrite an existing customized value");
    }
    _ => {
        setting_value = new_setting_value;
    }
}

println!("setting is {:?}", setting_value);
```

模式匹配

在 rust 中用到了各种各样的模式匹配：

- 匹配字面值：match 匹配中的匹配字面值。
- 匹配命名变量：match 匹配中的匹配命名变量。
- 匹配值的范围：match 匹配中的匹配值的范围。
- 多个模式匹配：match 匹配中的匹配多个模式。
- 解构元组：变量赋值中的解构元组。
- 解构结构体：match 匹配中的解构结构体和变量赋值中的解构结构体。
- 解构枚举：match 匹配中的解构枚举。
- 解构嵌套结构体枚举：match 匹配中的解构嵌套结构体枚举。
- 忽略整个值：变量赋值中的忽略值，元组解构中的忽略值，枚举匹配中的忽略值。
- 忽略剩余部分变量值：元组解构中的忽略剩余部分，和结构体解构中的忽略剩余部分。

运算符

Rust 中的运算符有如下一些，其中某些运算符可以通过实现 trait 进行重载。

运算符	示例	解释	是否可重载
!	!expr	按位非或逻辑非	Not
!=	var != expr	不等比较	PartialEq
%	expr % expr	算术取模	Rem
%=	var %= expr	算术取模与赋值	RemAssign
&	expr & expr	按位与	BitAnd
&=	var &= expr	按位与及赋值	BitAndAssign
*	expr * expr	算术乘法	Mul
*=	var *= expr	算术乘法与赋值	MulAssign
+	expr + expr	算术加法	Add
+=	var += expr	算术加法与赋值	AddAssign
-	- expr	算术取负	Neg
-	expr - expr	算术减法	Sub

运算符	示例	解释	是否可重载
-=	var -= expr	算术减法与赋值	SubAssign
/	expr / expr	算术除法	Div
/=	var /= expr	算术除法与赋值	DivAssign
<<	expr << expr	左移	Shl
<<=	var <<= expr	左移与赋值	ShlAssign
<	expr < expr	小于比较	PartialOrd
<=	expr <= expr	小于等于比较	PartialOrd
==	expr == expr	等于比较	PartialEq
>	expr > expr	大于比较	PartialOrd
>=	expr >= expr	大于等于比较	PartialOrd
>>	expr >> expr	右移	Shr
>>=	var >>= expr	右移与赋值	ShrAssign
^	expr ^ expr	按位异或	BitXor
^=	var ^= expr	按位异或与赋值	BitXorAssign
|	expr | expr	按位或	BitOr
|=	var |= expr	按位或与赋值	BitOrAssign
||	expr || expr	逻辑或	
&&	expr && expr	逻辑与	

流程控制

if 表达式

if 表达式允许根据条件执行不同的代码分支。所有的 if 表达式都以 if 关键字开头，其后跟一个条件，在条件为真时希望执行的代码块位于紧跟条件之后的大括号中，也可以包含一个可选的 else 表达式来提供一个在条件为假时应当执行的代码块。条件必须是 bool 值，如果条件不是 bool 值，我们将得到一个错误。

另外也可以将 else if 表达式与 if 和 else 组合来实现多重条件。

例子：

```
fn main() {
    let number = 6;

    if number % 4 == 0 {
        println!("number is divisible by 4");
    } else if number % 3 == 0 {
        println!("number is divisible by 3");
    } else if number % 2 == 0 {
        println!("number is divisible by 2");
    } else {
        println!("number is not divisible by 4, 3, or 2");
    }
}
```

因为 if 是一个表达式，我们可以在 let 语句的右侧使用它，但是需要注意 if 的每个分支的可能的返回值都必须是相同类型。例子：

```
fn main() {
    let condition = true;
    let number = if condition {
        5
    } else {
        6
    };

    println!("The value of number is: {}", number);
}
```

if let 表达式

if let 语法让我们以一种不那么冗长的方式结合 if 和 let，来处理只匹配一个模式的值而忽略其他模式的情况。if let 表达式的缺点在

于其穷尽性没有为编译器所检查，而 match 表达式则检查了。

例如：

```
let mut count = 0;
if let Coin::Quarter(state) = coin {
    println!("State quarter from {:?}!", state);
} else {
    count += 1;
}

// 等价于

let mut count = 0;
match coin {
    Coin::Quarter(state) => println!("State quarter from {:?}!", state),
    _ => count += 1,
}
```

另外也可以组合并匹配 if let、else if 和 else if let 表达式来获得更多的灵活性。例如：

```
fn main() {
    let favorite_color: Option<&str> = None;
    let is_tuesday = false;
    let age: Result<u8, _> = "34".parse();

    if let Some(color) = favorite_color {
        println!("Using your favorite color, {}, as the background", color);
    } else if is_tuesday {
        println!("Tuesday is green day!");
    } else if let Ok(age) = age {
        if age > 30 {
            println!("Using purple as the background color");
        } else {
            println!("Using orange as the background color");
        }
    } else {
        println!("Using blue as the background color");
    }
}
```

loop 循环

loop 关键字告诉 Rust 一遍又一遍地执行一段代码直到你明确要求停止。可以使用 break 关键字来告诉程序何时停止循环，也可以使用 continue 关键字告诉程序跳过剩下代码从头执行代码块。

例子：

```
fn main() {
    loop {
        println!("again!");
    }
}
```

由于 loop 循环是一个代码块，因此可以使用 break 表达式的方式，为 loop 循环的返回值。例如：

```
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };

    println!("The result is {}", result);
}
```

while 循环

while 条件循环，当条件为真，执行循环，当条件不再为真，则退出循环。例如：

```
fn main() {
    let mut number = 3;

    while number != 0 {
        println!("{}!", number);
        number = number - 1;
    }

    println!("LIFTOFF!!!");
}
```

在 while 循环中可以使用 break 提前退出循环，但是不支持 break 表达式的方式。例如：

```
fn main() {
    let mut number = 3;

    while number != 0 {
        println!("{}!", number);
        number = number - 1;
        if number == 1 {
          break;
        }
    }

    println!("LIFTOFF!!!");
}
```

在 while 循环中可以使用 continue 跳过剩余代码，从头开始执行。例如：

```
fn main() {
    let mut number = 3;

    while number != 0 {
        if number == 2 {
            number = number -1;
            continue;
        }
        println!("{}!", number);
        number = number - 1;
    }

    println!("LIFTOFF!!!");
}
```

while let 循环

while let 允许只要模式匹配就一直进行 while 循环。例如：

```
let mut stack = Vec::new();

stack.push(1);
stack.push(2);
stack.push(3);

while let Some(top) = stack.pop() {
    println!("{}", top);
}
```

for 循环

for 循环用于遍历一个集合，对一个集合的每个元素执行一些代码。例如：

```
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a.iter() {
        println!("the value is: {}", element);
```

```
    }
}
```

for 循环有以下注意点：

- for x in X 中的 X 必须实现了 std::iter::IntoIterator trait。
- for 循环开始的第一个步就是调用 X 的 into_iter 方法获取实现了 std::iter::Iterator 的迭代器，然后不断调用 next 获取元素。
- 所有的迭代器都实现了 std::iter::IntoIterator trait，他们会返回自己本身。

for 循环可以使用 break 提前退出循环，但是不支持 break 表达式的方式。例如：

```
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a.iter() {
        println!("the value is: {}", element);
        if *element == 30 {
            break;
        }
    }
}
```

for 循环可以使用 continue 跳过剩余代码，从头开始执行。例如：

```
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a.iter() {
        if *element == 30 {
            continue;
        }
        println!("the value is: {}", element);
    }
}
```

循环标签

循环表达式可以具有标签，标签应该在循环表达式关键字之前，并且以"'"开头，例如：" 'foo: loop { break 'foo; }, 'bar: while false {}, 'humbug: for _ in 0..0 {}"。

当循环表达式具有标签时，break 或者 continue 可以选择结束哪层的循环或者从哪层的循环开始执行。例如：

```
'outer: loop {
    while true {
        break 'outer;
    }
}
```

对于 loop 循环，可以使用 break 'label EXPR 语法跳出指定 loop，并返回表达式的值。例如：

```
fn main() {
    let (mut a, mut b) = (1, 1);
    let result = 'outloop: loop {
        'inloop: while true {
            if b > 10 {
                break 'outloop b;
            }
            let c = a + b;
            a = b;
            b = c;
        }
    };
    assert_eq!(result, 13);
}
```

match 匹配

match 的极为强大的控制流运算符，它允许我们将一个值与一系列的模式相比较，并根据相匹配的模式执行相应代码。模式由如下一些内容组合而成：

- 字面值
- 解构的数组、枚举、结构体或者元组

- 变量
- 通配符
- 占位符

match 表达式由 match 关键字、用于匹配的值和一个或多个分支构成，这些分支包含一个模式和在值匹配分支的模式时运行的表达式：

```
match VALUE {
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
}
```

注意：
- match 表达式必须是穷尽（exhaustive）的，意为 match 表达式所有可能的值都必须被考虑到。
- 特定的模式 _ 可以匹配所有情况，不过它从不绑定任何变量。
- 所有 EXPRESSION 表达式的返回值应该相同。

因为 match 是一个表达式，我们可以在 let 语句的右侧使用它，但是需要注意 match 的每个分支的可能的返回值(允许不返回)都必须是相同类型。例子：

```
let x = 1;

let i = match x {
    1 => 2,
    2 => 3,
    _ => x,
};
println!("{}", i)
```

匹配模式

匹配字面值

match 可以直接匹配字面值模式。例如：

```
let x = 1;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

匹配命名变量

由于 match 会开始一个新作用域，match 表达式中作为模式的一部分声明的变量会覆盖 match 结构之外的同名变量，与所有变量一样。例子：

```
fn main() {
    let x = Some(5);
    let y = 10;

    match x {
        Some(50) => println!("Got 50"),
        Some(y) => println!("Matched, y = {:?}", y),
        _ => println!("Default case, x = {:?}", x),
    }

    println!("at the end: x = {:?}, y = {:?}", x, y);
}
```

匹配值的范围

..= 语法允许你匹配一个闭区间范围内的值。范围只允许用于数字或 char 值，因为编译器会在编译时检查范围不为空。例子：

```
let x = 'c';

match x {
    'a'..='j' => println!("early ASCII letter"),
    'k'..='z' => println!("late ASCII letter"),
    _ => println!("something else"),
}
```

匹配多个模式

在 match 表达式中，可以使用 | 语法匹配多个模式，它代表 或（or）的意思。例子：

```
let x = 1;

match x {
    1 | 2 => println!("one or two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

解构结构体

可以为所有字段创建变量，也可以使用字面值作为结构体模式的一部分进行解构。例子：

```
fn main() {
    let p = Point { x: 0, y: 7 };

    match p {
        Point { x, y: 0 } => println!("On the x axis at {}", x),
        Point { x: 0, y } => println!("On the y axis at {}", y),
        Point { x, y } => println!("On neither axis: ({}, {})", x, y),
    }
}
```

解构枚举

可以绑定匹配的模式的部分值，从枚举成员中提取值。例如：

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}

fn main() {
    let msg = Message::ChangeColor(0, 160, 255);

    match msg {
        Message::Quit => {
            println!("The Quit variant has no data to destructure.")
        }
        Message::Move { x, y } => {
            println!(
                "Move in the x direction {} and in the y direction {}",
                x,
                y
            );
        }
        Message::Write(text) => println!("Text message: {}", text),
        Message::ChangeColor(r, g, b) => {
            println!(
                "Change the color to red {}, green {}, and blue {}",
                r,
                g,
                b
            )
        }
    }
}
```

解构嵌套的结构体和枚举

match 直接解构嵌套的结构体和枚举。例如：

```

enum Message {
    Quit,
    Move { x: i32, y: i32 },
```

```
    Write(String),
    ChangeColor(Color),
}

fn main() {
    let msg = Message::ChangeColor(Color::Hsv(0, 160, 255));

    match msg {
        Message::ChangeColor(Color::Rgb(r, g, b)) => {
            println!(
                "Change the color to red {}, green {}, and blue {}",
                r,
                g,
                b
            )
        }
        Message::ChangeColor(Color::Hsv(h, s, v)) => {
            println!(
                "Change the color to hue {}, saturation {}, and value {}",
                h,
                s,
                v
            )
        }
        _ => ()
    }
}
```

匹配守卫

匹配守卫是一个指定于 match 分支模式之后的额外 if 条件，它也必须被满足才能选择此分支。

例子：

```
let num = Some(4);

match num {
    Some(x) if x < 5 => println!("less than five: {}", x),
    Some(x) => println!("{}", x),
    None => (),
}
```

只有 x 有值，并且值小于 5 才能匹配第一条分支。

匹配守卫可以和其他匹配模式配合使用。例子：

```
let x = 4;
let y = false;

match x {
    4 | 5 | 6 if y => println!("yes"),
    _ => println!("no"),
}
```

注意：优先进行模式匹配，再进行匹配守卫。因此此例类似"(4 | 5 | 6) if y => ..."，

模式绑定

at 运算符允许我们在创建一个存放值的变量的同时测试其值是否匹配模式。

例子：

```
enum Message {
    Hello { id: i32 },
}

let msg = Message::Hello { id: 5 };

match msg {
    Message::Hello { id: id_variable @ 3..=7 } => {
        println!("Found an id in range: {}", id_variable)
```

```
    },
    Message::Hello { id: 10..=12 } => {
        println!("Found an id in another range")
    },
    Message::Hello { id } => {
        println!("Found some other id: {}", id)
    },
}
```

此例中分支一将绑定变量 id_variable，并测试其值是否满足范围匹配。

函数

函数声明

Rust 中的函数定义以 fn 开始并在函数名后跟一对圆括号。大括号告诉编译器哪里是函数体的开始和结尾。

函数也可以被定义为拥有参数（parameters），参数是特殊变量，是函数签名的一部分。在函数签名中，必须声明每个参数的类型。当一个函数有多个参数时，使用逗号分隔。

函数体由一系列的语句和一个可选的结尾表达式构成。语句（Statements）是执行一些操作但不返回值的指令，以分号(;)结尾。表达式（Expressions）计算并产生一个值，表达式有以下多种表现：数学运算、函数调用、宏调用、代码块。

函数可以向调用它的代码返回值，rust 不对返回值命名，但要在箭头（->）后声明它的类型。函数的返回值等同于函数体最后一个表达式的值。也可以使用 return 关键字和指定值，从函数中提前返回。注意：如果函数的最后是一条语句，那么将没有返回值，等价于返回空元组。

使用函数名后跟圆括号来调用我们定义过的任意函数。被调用的函数不需要在被调用之前定义，只要定义了就行。

Rust 代码中的函数和变量名使用 snake case 规范风格。

main 函数是 binary crate 的入口函数。

例子：

```
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {}", x);
}

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

永不返回

Rust 有一个叫做"!"的特殊类型。这个特殊类型用于声明函数从不返回。例如：

```
fn bar() -> ! {
    // --snip--
}
```

从不返回的函数被称为发散函数，以下值的类型也是"!"

- continue，例如：

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

- panic，例如：

```
impl<T> Option<T> {
    pub fn unwrap(self) -> T {
        match self {
            Some(val) => val,
            None => panic!("called `Option::unwrap()` on a `None` value"),
        }
    }
}
```

- loop，例如：

```
print!("forever ");
```

```
    loop {
        print!("and ever ");
}
```

函数指针

可以通过函数指针允许我们使用函数作为另一个函数的参数。fn 被称为函数指针类型，例如：

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

fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}

fn main() {
    let answer = do_twice(add_one, 5);

    println!("The answer is: {}", answer);
}
```

函数指针实现了所有三个闭包 trait（Fn、FnMut 和 FnOnce），所以总是可以在调用期望闭包的函数时传递函数指针作为参数。

方法语法

方法与函数类似：它们使用 fn 关键字和名称声明，可以拥有参数和返回值，同时包含在某处调用该方法时会执行的代码。

定义方法

方法与函数是不同的，因为他们在"结构体"或者"枚举"或"trait 对象"的上下文中被定义，第一个参数总是 self，代表调用该方法的实例。方法的第一个参数可以有以下几种：

- self：获取实例的所有权，这种方法比较少见。
- mut self：获取可变实例的所有权，这种方法更少见。
- &self：借用实例，并读取其中数据。
- &mut self：借用实例，并读写其中数据。

结构体方法

使用 impl 关键字，后跟结构体类型名称，代表结构体的上下文。例如：

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

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

fn main() {
    let rect1 = Rectangle { width: 30, height: 50 };

    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
```

枚举方法

使用 impl 关键字，后跟枚举类型名称，代表枚举的上下文。例如：

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
```

```
    Write(String),
    ChangeColor(i32, i32, i32),
}

impl Message {
    fn call(&self) {
        // 在这里定义方法体
    }
}

let m = Message::Write(String::from("hello"));
m.call();
```

trait 实现

见实现 trait。

trait 默认实现

见 trait 默认实现。

调用方法

可以使用 object.something()的方式调用方法，当使用 object.something()调用方法时，Rust 有一个叫自动引用和解引用的功能，Rust 会自动为 object 添加&、&mut 或*以便使 object 与方法签名匹配。

例如：

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

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

fn main() {
    let rect1 = Rectangle { width: 30, height: 50 };

    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area() // 此处会自动转换为(&rect1).area();
    );
}
```

多个方法

如果拥有多个方法，那么可以在一个 impl 块中，也可以在多个 impl 块中。例如：

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

    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}

// 等价于

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

```
    }
}

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

关联函数

在 impl 块中定义的不以 self 作为参数的函数被称为关联函数，它们与结构体相关联。它们仍是函数而不是方法，因为它们并不作用于一个结构体的实例。可以使用结构体名和::语法来调用关联函数。

例子：

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn square(size: u32) -> Rectangle {
        Rectangle { width: size, height: size }
    }
}

fn main() {
    let rect1 = Rectangle::square(3);

    println!(
        "The area of the rectangle is {:?} square pixels.",
        rect1
    );
}
```

闭包

闭包是可以保存进变量或作为参数传递给其他函数的匿名函数。可以在一个地方创建闭包，然后在不同的上下文中执行闭包运算，闭包允许捕获调用者作用域中的值。

定义闭包

闭包的定义以一对竖线（|）开始，在竖线中指定闭包的参数，例如|num|，如果有多于一个参数，可以使用逗号分隔，例如|param1, param2|。参数之后是存放闭包体的大括号，如果闭包体只有一行则大括号是可以省略的。例子：

```
let expensive_closure = |num| {
    println!("calculating slowly...");
    thread::sleep(Duration::from_secs(2));
    num
};
```

类型推断

闭包不要求像 fn 函数那样在参数和返回值上注明类型。编译器能能够所处的上下文可靠的推断参数和返回值的类型，类似于它是如何能够推断大部分变量的类型一样。下面是一个显式注明类型的例子：

```
let expensive_closure = |num: u32| -> u32 {
    println!("calculating slowly...");
    thread::sleep(Duration::from_secs(2));
    num
};
```

对于未注明类型的闭包，闭包定义会为每个参数和返回值推断一个具体类型，如果尝试调用一个被推断为两个不同类型的闭包，编译器就会给出错误。例如：

```
let example_closure = |x| x;

let s = example_closure(String::from("hello"));
```

```
let n = example_closure(5);
```

闭包类型

每一个闭包实例有其自己独有的匿名类型，因此即便两个闭包有着相同的签名，他们的类型仍然可以被认为是不同。有时候，我们需要在函数参数、结构体、枚举中使用闭包变量，因此需要指定闭包的类型，但是由于每个闭包的类型都不同，因此 rust 提供了 Fn、FnMut 或 FnOnce 三个 trait，所有的闭包类型实现了其中的一个或者多个 trait，所有的函数实现了这三个 trait。

闭包可以通过三种方式捕获其环境，闭包周围的作用域被称为其环境。他们直接对应函数的三种获取参数的方式：

- Fn：不可变借用。Fn 用于从其环境获取不可变的借用值。
- FnMut：可变借用。FnMut 用于获取可变的借用值，因此其可以改变其环境。
- FnOnce：获取所有权。FnOnce 将消费从周围作用域捕获的变量，获取所有权并在定义闭包时将其移动进闭包。

由于所有闭包都可以被调用至少一次，所以所有闭包都实现了 FnOnce。没有移动被捕获变量的所有权到闭包内的闭包也实现了 FnMut。不需要对被捕获的变量进行可变访问的闭包则也实现了 Fn。

如果希望强制闭包获取其使用的环境值的所有权，可以在参数列表前使用 move 关键字。例如：

```
fn main() {
    let x = vec![1, 2, 3];
    let equal_to_x = move |z| z == x;
    println!("can't use x here: {:?}", x);
    let y = vec![1, 2, 3];
    assert!(equal_to_x(y));
}
```

注意：move 闭包仍然可能实现 Fn 或 FnMut，即使它们通过获取所有权捕获变量。这是因为闭包类型实现的特征取决于闭包如何处理捕获的值，而不是它如何捕获它们。

存放闭包的例子：

```
impl<T> Cacher<T>
    where T: Fn(u32) -> u32
{
    fn new(calculation: T) -> Cacher<T> {
        Cacher {
            calculation,
            value: None,
        }
    }

    fn value(&mut self, arg: u32) -> u32 {
        match self.value {
            Some(v) => v,
            None => {
                let v = (self.calculation)(arg);
                self.value = Some(v);
                v
            },
        }
    }
}

fn generate_workout(intensity: u32, random_number: u32) {
    let mut expensive_result = Cacher::new(|num| {
        println!("calculating slowly...");
        thread::sleep(Duration::from_secs(2));
        num
    });

    if intensity < 25 {
        println!(
            "Today, do {} pushups!",
            expensive_result.value(intensity)
        );
        println!(
            "Next, do {} situps!",
            expensive_result.value(intensity)
        );
    } else {
```

```
        if random_number == 3 {
            println!("Take a break today! Remember to stay hydrated!");
        } else {
            println!(
                "Today, run for {} minutes!",
                expensive_result.value(intensity)
            );
        }
    }
}
```

返回闭包

如果需要返回一个闭包，那么必须使用 Box<dyn Fn(i32) -> i32>的方式返回。例如：

```
fn returns_closure() -> Box<dyn Fn(i32) -> i32> {
    let y = 2;
    Box::new(move |x| x + y)
}
```

非捕获闭包

非捕获闭包是不从其环境中捕获任何内容的闭包。 它们可以被强制为具有匹配签名的函数指针（例如 fn()）。对于捕获闭包，则只能通过 Fn/FnMut/FnOnce 这三个 trait 来引用(见返回闭包)。

例如：

```
let add = |x, y| x + y;

let mut x = add(5,7);

type Binop = fn(i32, i32) -> i32;
let bo: Binop = add;
x = bo(5,7);
```

迭代器

迭代器模式允许对一个序列的项进行某些处理。其中迭代器负责遍历序列中的每一项和决定序列何时结束的逻辑。

创建和使用迭代器示例：

```
let v1 = vec![1, 2, 3];

let v1_iter = v1.iter();

for val in v1_iter {
    println!("Got: {}", val);
}
```

迭代器 Iterator

迭代器是实现了 Iterator trait 的类型的实例。其定义如下：

```
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
    // 此处省略了方法的默认实现
}
```

其中，next 每次被调用返回迭代器中的一个项，封装在 Some 中，当迭代器结束时，它返回 None。

有以下几种途径得到迭代器：

1. 通过在 vector 上调用 iter、into_iter 或 iter_mut 来创建一个迭代器。
2. 用标准库中其他的集合类型创建迭代器，比如哈希 map。
3. 可以实现 Iterator trait 来创建任何我们希望的迭代器。

注意：

1. 迭代器是惰性的（lazy），这意味着仅创建一个迭代器不会有任何其他效果，只有调用了迭代器的 next 方法才能消费其中的项。
2. 要调用迭代器的 next 方法，那么迭代器必须是可变的。

自定义 Iterator

自定义迭代器的唯一要求提供 next 方法。

例子：

```
struct Counter {
    count: u32,
}

impl Counter {
    fn new() -> Counter {
        Counter { count: 0 }
    }
}

impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        self.count += 1;

        if self.count < 6 {
            Some(self.count)
        } else {
            None
        }
    }
}
```

消费迭代器

Iterator trait 中定义了一类方法，这些方法会调用 next 方法，他们被称为消费适配器。

例如 sum 方法获取迭代器的所有权并反复调用 next 来遍历迭代器，它将每一个项加总到一个总和并在迭代完成时返回总和：

```
#[test]
fn iterator_sum() {
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();

    let total: i32 = v1_iter.sum();

    assert_eq!(total, 6);
}
```

迭代器适配器

Iterator trait 中定义了另一类方法，这些方法会产生新类型的迭代器，他们被称为迭代器适配器。

例如 map 方法使用闭包来调用每个元素以生成新的迭代器：

```
let v1: Vec<i32> = vec![1, 2, 3];

let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();

assert_eq!(v2, vec![2, 3, 4]);
```

其中 collect 方法消费新迭代器并创建一个 vector

for 循环

可以使用 for 循环来消费迭代器。具体见 for 循环。

并发编程

线程

Rust 标准库只提供了 1:1 线程模型实现。

创建线程

在 rust 中，调用 thread::spawn 函数并传递一个闭包来创建一个新的线程。下面是一个例子：

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

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

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

等待线程结束

thread::spawn 函数会返回一个 JoinHandle 以确保该线程能够运行至结束，通过调用 handle 的 join 会阻塞当前线程直到 handle 所代表的线程结束。

例子：

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

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

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

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

线程 move 闭包

由于新线程和旧线程是并行运行的，因此如果新线程启动时的闭包中存在引用旧线程中的值，运行时可能该值已经被清理，因此新线程闭包中是不能存在旧线程中值的引用的。为了避免编译器自动推断为引用类型，应当为闭包强制添加 move 关键字。例子：

```
use std::thread;

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

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

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

传递消息

Rust 中一个实现消息传递并发的主要工具是通道，通道有两部分组成，一个发送者（transmitter）和一个接收者（receiver）。

创建通道

可以使用 mpsc::channel 函数创建一个新的通道；mpsc 是多个生产者，单个消费者（multiple producer, single consumer）的缩写。

mpsc::channel 函数返回一个元组：第一个元素是发送端，而第二个元素是接收端。下面是一个例子：

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

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

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

发送消息

通道的发送端有一个 send 方法用来发送消息。send 方法返回一个 Result<T, E> 类型，所以如果接收端已经由于离开作用域导致被清理了，将没有发送值的目标，所以发送操作会返回错误。

例子：

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

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

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

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

mpsc 支持多个发送者，mpsc::Sender::clone 可以 clone 出多个发送者。例如：

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

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

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

thread::spawn(move || {
    let vals = vec![
        String::from("more"),
        String::from("messages"),
        String::from("for"),
        String::from("you"),
    ];

    for val in vals {
        tx.send(val).unwrap();
```

```
        thread::sleep(Duration::from_secs(1));
    }
});

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

接收消息

通道的接收端有四个有用的方法：recv、try_recv、recv_timeout、recv_deadline。

- recv 方法会阻塞主线程执行直到从通道中接收一个值。
- try_recv 方法不会阻塞，相反它立刻返回一个 Result<T, E>：Ok 值包含可用的信息，而 Err 值代表此时没有任何消息。
- recv_timeout 方法会阻塞直到从通道中接收一个值或者超时时间到达。
- recv_deadline 方法会阻塞直到从通道中接收一个值或者到达指定时间。

例子：

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

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

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

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

另外通道的接收端实现了 IntoIterator trait，因此可以使用 for 来循环遍历消息，如果没有消息则等待。例子：

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

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

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

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

注意：mpsc 仅允许一个接收者。

共享状态

互斥器

互斥器任意时刻，其只允许一个线程访问某些数据。Rust 中的互斥器是 Mutex<T>，使用关联函数 new 来创建一个 Mutex<T>，使用其 lock 方法获取锁，以访问互斥器中的数据。这个调用会阻塞当前线程，直到我们拥有锁为止。

lock 方法调用返回一个叫做 MutexGuard 的智能指针。这个智能指针实现了 Deref 来指向其内部数据；其也提供了一个 Drop 实现当 MutexGuard 离开作用域时自动释放锁，

单线程下使用 Mutex 的例子：

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

fn main() {
    let m = Mutex::new(5);

    {
        let mut num = m.lock().unwrap();
        *num = 6;
    }

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

原子引用计数

在单线程下使用 Mutex 没有任何意义，在多线程的情况下，必须使用类似共享智能指针 Rc，来使得线程间共享使用互斥器，但是 Rc 本身是线程非安全的，rust 提供了 Arc<T>，允许在线程间共享引用数据。相较于 Rc，Arc 具有一定的性能开销。Arc 的 API 和 Rc 是一样的。

因此使用 Arc<Mutex<T>>可以在线程间共享互斥器。例子：

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

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

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

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

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

可扩展并发

有两个并发概念是内嵌于语言中的：std::marker 中的 Sync trait 和 Send trait。

- Send trait 表明类型的所有权可以在线程间传递。几乎所有的 Rust 类型都是 Send 的，不过有一些例外，包括 Rc<T>。任何完全由 Send 的类型组成的类型也会自动被标记为 Send。
- Sync trait 表明一个实现了 Sync 的类型可以安全的在多个线程中拥有其值的引用。换一种方式来说，对于任意类型 T，如果 &T（T 的引用）是 Send 的话 T 就是 Sync 的，这意味着其引用就可以安全的发送到另一个线程。任何完全由 Sync 的类型组成的类型也会自动被标记为 Sync。

通常并不需要手动实现 Send 和 Sync trait，因为由 Send 和 Sync 的类型组成的类型，自动就是 Send 和 Sync 的。因为他们是标记 trait，甚至都不需要实现任何方法。他们只是用来加强并发相关的不可变性的。

async/await

async/.await 是 Rust 内置语法，用于让异步函数编写得像同步代码。async 将代码块转化成 实现了 Future 特质的状态机。使用同步方法调用阻塞函数会阻塞整个线程，但阻塞 Future 只会 让出（yield）线程控制权，让其他 Future 继续执行。

使用 async/await 时，必须打开以下特性：

```
[dev-dependencies]
futures-preview = { version = "=0.3.0-alpha.16", features = ["async-await", "nightly"] }
```

async

async 主要有两种用法：async 函数和 async 代码块。这几种用法都会返回一个 Future 对象。例如：

```
// `foo()` 返回一个实现了 `Future<Output = u8>` 的类型.
// `foo().await` 将返回类型为 `u8` 的值.
async fn foo() -> u8 { 5 }

fn bar() -> impl Future<Output = u8> {
    // 这个 `async` 区域返回一个实现了 `Future<Output = u8>` 的类型.
    async {
        let x: u8 = foo().await;
        x + 5
    }
}
```

await

async 体以及其他 future 类型是惰性的：除非它们运行起来，否则它们 什么都做。运行 Future 最常见的方法是.await 它。当.await 在 Future 上调用时，它会尝试把 future 跑到完成状态。

如果 Future 被阻塞了，它会让出当前线程的控制权。能取得进展时， 执行器就会捡起这个 Future 并继续执行，让.await 求解。

生命周期

和传统函数不同，async fn 会获取引用以及其他拥有非'static 生命周期的参数，并返回被这些参数的生命周期约束的 Future：

```
// 这是一个 `async` 函数:
async fn foo(x: &u8) -> u8 { *x }

// 相当于这个普通函数:
fn foo_expanded<'a>(x: &'a u8) -> impl Future<Output = u8> + 'a {
    async move { *x }
}
```

这意味着执行.await 的时候，被引用的数据必须仍然有效，因此通常 future 不能被发送到其他任务或者线程执行。

async move

async 块和闭包允许使用 move 关键字，这和普通的闭包一样。一个 async move 块会获取 所指向变量的所有权，允许它的生命周期超过当前作用域(outlive)，但是放弃了与其他代码共享这些 变量的能力：

```
/// `async` 区域:
///
/// 多个 `async` 区域可以访问相同的本地变量，
/// 只要它们在变量的作用域内执行.
async fn blocks() {
    let my_string = "foo".to_string();

    let future_one = async {
        // ...
        println!("{}", my_string);
    };

    let future_two = async {
        // ...
        println!("{}", my_string);
    };

    // 运行两个 `future`，输出两次 "foo":
    let ((), ()) = futures::join!(future_one, future_two);
}

/// `async move` 区域:
///
/// 只有一个 `async move` 区域可以访问同一个被捕获的变量,
/// 因为被捕获的变量已经移动到 `async move` 生成的 `future` 中:
fn move_block() -> impl Future<Output = ()> {
    let my_string = "foo".to_string();
    async move {
```

```
        // ...
        println!("{}", my_string);
    }
}
```

注意事项

在使用多线程的 Future 执行器时，一个 Future 可能在线程间移动，所以任何在 async 体中使用的变量必须能够穿过线程，因为任何.await 都有可能导致线程切换。这意味着使用 Rc，&RefCell 或者其他没有实现 Send 特质的类型是不安全的，包括那些指向 没有 Sync 特质类型的引用。(注意：使用这些类型是允许的，只要他们不是在调用.await 的作用域内。)

类似的，横跨.await 持有一个非 future 感知的锁这种做法是很不好的，因为它能导致整个线程池 锁上：一个任务可能获得了锁，.await 然后让出到执行器，允许其他任务尝试获取所并导致死锁。 为了避免这种情况，使用 futures::lock 里的 Mutex 类型比起 std::sync 更好。

使用 unsafe

Rust 在编译时会强制执行的内存安全保证。但是可以通过 unsafe 关键字来切换到不安全 Rust，接着可以开启一个新的存放不安全代码的块。这里有五类可以在不安全 Rust 中进行而不能用于安全 Rust 的操作，它们称之为"不安全的超级力量"：

- 解引用裸指针
- 调用不安全的函数或方法
- 访问或修改可变静态变量
- 实现不安全 trait
- 访问 union 的字段

注意：unsafe 并不会关闭借用检查器或禁用任何其他 Rust 安全检查：如果在不安全代码中使用引用，它仍会被检查。unsafe 关键字只是提供了那五个不会被编译器检查内存安全的功能。

解引用裸指针

不安全的 Rust 有两个被称为裸指针的类型：*const T 和 *mut T，分别是不可变裸指针，和可变裸指针。

裸指针与引用和智能指针的区别在于：

- 允许忽略借用规则，可以同时拥有不可变和可变的指针，或多个指向相同位置的可变指针
- 不保证指向有效的内存
- 允许为空
- 不能实现任何自动清理功能

例子：

```
let mut num = 5;

let r1 = &num as *const i32; // 创建裸指针
let r2 = &mut num as *mut i32; // 创建裸指针

unsafe {
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", *r2);
}
```

注意：裸指针创建不需要在 unsafe 代码块中。

调用不安全的函数或方法

不安全函数和方法有一个额外的 unsafe 关键字。关键字 unsafe 表示该函数具有调用时需要满足的要求，而 Rust 不会保证满足这些要求，通常传入裸引用指针参数的函数具有 unsafe 关键字。例如：

```
unsafe fn dangerous() {}

unsafe {
    dangerous();
}
```

注意：在不安全函数中进行另一个不安全操作时无需新增额外的 unsafe 块。

使用 extern 函数调用外部代码时，也需要放入 unsafe 代码块。extern 块中声明的函数在 Rust 代码中总是不安全的。例如：

```
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
```

```
    unsafe {
        println!("Absolute value of -3 according to C: {}", abs(-3));
    }
}
```

访问或修改可变静态变量

静态变量可以是可变的，但是访问和修改可变静态变量都是不安全的。需要使用 unsafe 代码块。例如：

```
static mut COUNTER: u32 = 0;

fn add_to_count(inc: u32) {
    unsafe {
        COUNTER += inc;
    }
}

fn main() {
    add_to_count(3);

    unsafe {
        println!("COUNTER: {}", COUNTER);
    }
}
```

实现不安全 trait

当至少有一个方法中包含编译器不能验证的不变量时 trait 是不安全的。可以在 trait 之前增加 unsafe 关键字将 trait 声明为 unsafe，同时 trait 的实现也必须标记为 unsafe。

例子：

```
unsafe trait Foo {
    // methods go here
}

unsafe impl Foo for i32 {
    // method implementations go here
}
```

访问 union 的字段

union 和 struct 类似，但是在一个实例中同时只能使用一个声明的字段。联合体主要用于和 C 代码中的联合体交互。访问联合体的字段是不安全的，因为 Rust 无法保证当前存储在联合体实例中数据的类型。可以查看参考文档了解有关联合体的更多信息。

宏

在 rust 中，宏有以下两种方式：

- 声明宏：使用 macro_rules!的。
- 过程宏：过程宏分为以下三种类型：
  - 指令宏：在结构体和枚举上指定通过 derive 属性添加的代码。
  - 类属性宏：定义可用于任意项的自定义属性，并且创建新的属性。
  - 类函数宏：看起来像函数不过作用于作为参数传递的 token。

内置声明宏：https://doc.rust-lang.org/std/all.html#Macros
内置属性宏：https://doc.rust-lang.org/reference/attributes.html#built-in-attributes-index

声明宏

Rust 最常用的宏形式是声明宏，声明宏允许我们编写一些类似 match 表达式的代码。其根据 match 模式将匹配的代码替换为定义的代码。

类似 vec! 宏的定义如下：

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
```

```
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

- #[macro_export] 注解说明宏外部是可用的。
- macro_rules! 和宏名称开始宏定义，且所定义的宏并 不带 感叹号。名字后跟大括号表示宏定义体。
- 然后是单边模式 ($($x:expr),*) ，后跟 => 以及和模式相关的代码块。
- $()捕获了符合括号内模式的值以用于替换后的代码。$x:expr 表示其匹配 Rust 的任意表达式，并将该表达式记作 $x。
- $()之后的逗号说明一个可有可无的逗号分隔符可以出现在 $() 所匹配的代码之后。*说明该模式匹配零个或更多个之前的模式。
- 对于每个（在 => 前面）匹配模式中的 $() 的部分，生成零个或更多个（在 => 后面）位于 $()* 内的 temp_vec.push() ，生成的个数取决于该模式被匹配的次数。

宏展开类似如下代码：

```
let mut temp_vec = Vec::new();
temp_vec.push(1);
temp_vec.push(2);
temp_vec.push(3);
temp_vec
```

过程宏

过程宏接收 Rust 代码作为输入，在这些代码上进行操作，然后产生另一些代码作为输出，并不会改变原有的代码。

指令宏

指令宏只能在结构体和枚举上指定通过 derive 属性添加的代码。

例子：

```
extern crate proc_macro;

use crate::proc_macro::TokenStream;
use quote::quote;
use syn;

#[proc_macro_derive(HelloMacro)]
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {
    // 将 Rust 代码解析为语法树以便进行操作
    let ast = syn::parse(input).unwrap();

    // 构建 trait 实现
    impl_hello_macro(&ast)
}
```

- proc_macro_derive 指定这是一个指令宏，括号内为使用时指定的名字。
- input 为结构体或者枚举的 token 流。
- 输出为新的添加代码。

使用例子：

```
use hello_macro::HelloMacro;
use hello_macro_derive::HelloMacro;

#[derive(HelloMacro)]
struct Pancakes;

fn main() {
    Pancakes::hello_macro();
}
```

类属性宏

类属性宏与指令宏相似，不同于为 derive 属性生成代码，它们允许你创建新的属性。它们也更为灵活；derive 只能用于结构体和枚举；属性还可以用于其它的项，比如函数。

类属性宏定义的函数签名看起来像这样：

```
#[proc_macro_attribute]
```

```
pub fn route(attr: TokenStream, item: TokenStream) -> TokenStream {
```

attr 代表类属性本身，item 代表所标记的项。

使用例子：

```
#[route(GET, "/")]
fn index() {
```

attr 为 GET, "/"，item 为函数内容。

类函数宏

类函数宏定义看起来像函数调用的宏。类似于 macro_rules!，它们比函数更灵活；例如，可以接受未知数量的参数。

类函数宏签名类似如下：

```
#[proc_macro]
pub fn sql(input: TokenStream) -> TokenStream {
```

使用例子：

```
let sql = sql!(SELECT * FROM posts WHERE id=1);
```

错误处理

Rust 将错误组合成两个主要类别：可恢复错误（recoverable）和 不可恢复错误（unrecoverable）。可恢复错误通常代表向用户报告错误和重试操作是合理的情况，比如未找到文件。不可恢复错误通常是 bug 的同义词，比如尝试访问超过数组结尾的位置。

Rust 并没有异常，但是，有可恢复错误 Result<T, E>，和不可恢复(遇到错误时停止程序执行)错误 panic!。

不可恢复错误

Rust 有 panic!宏。当执行这个宏时，程序会打印出一个错误信息，展开并清理栈数据，然后接着退出。例如：

```
fn main() {
    panic!("crash and burn");
}

// 程序运行输出以下内容：
$ cargo run
   Compiling panic v0.1.0 (file:///projects/panic)
    Finished dev [unoptimized + debuginfo] target(s) in 0.25s
     Running `target/debug/panic`
thread 'main' panicked at 'crash and burn', src/main.rs:2:5
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

设置 RUST_BACKTRACE 环境变量非 0，打印完整的调用栈，但是这仅在 debug 模式下生效。

当出现 panic 时，程序默认会开始展开，这意味着 Rust 会回溯栈并清理它遇到的每一个函数的数据，不过这个回溯并清理的过程有很多工作。另一种选择是直接终止，这会不清理数据就退出程序。通过在 Cargo.toml 的[profile]部分增加 panic = 'abort'，可以由展开切换为终止。也可以在[profile.release] 部分增加 panic = 'abort'，表示只有在 release 的时候才切换为终止。

可恢复错误

大部分错误并没有严重到需要程序完全停止执行。有时，一个函数会因为一个容易理解并做出反应的原因失败。通常他们会返回一个Result<T, E>类型的结果，指示是否成功。

Result 枚举定义如下：

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

其中 T 代表成功时返回的 Ok 成员中的数据的类型，而 E 代表失败时返回的 Err 成员中的错误的类型。

Result 的使用方法看这里。

简单的处理错误的例子：

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");
```

```
    let f = match f {
        Ok(file) => file,
        Err(error) => {
            panic!("Problem opening the file: {:?}", error)
        },
    };
}
```

匹配不同的错误的例子：

```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("hello.txt") {
                Ok(fc) => fc,
                Err(e) => panic!("Problem creating the file: {:?}", e),
            },
            other_error => panic!("Problem opening the file: {:?}", other_error),
        },
    };
}
```

unwrap

unwrap 方法的作用是在 Result 值是成员 Ok，会返回 Ok 中的值。如果 Result 是成员 Err，会为我们调用 panic!。

例如：

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt").unwrap();
}
```

unwrap_or_else

unwrap_or_else 方法类似 unwrap，在 Result 值是成员 Ok，会返回 Ok 中的值。如果 Result 是成员 Err，会调用传入的闭包函数!。

例如：

```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let f = File::open("hello.txt").unwrap_or_else(|error| {
        if error.kind() == ErrorKind::NotFound {
            File::create("hello.txt").unwrap_or_else(|error| {
                panic!("Problem creating the file: {:?}", error);
            })
        } else {
            panic!("Problem opening the file: {:?}", error);
        }
    });
}
```

expect

expect 方法类似 unwrap，在 Result 值是成员 Ok，会返回 Ok 中的值。如果 Result 是成员 Err，为我们调用 panic!，不同的在于不适用默认的错误信息。

例如：

```
use std::fs::File;
```

```
fn main() {
    let f = File::open("hello.txt").expect("Failed to open hello.txt");
}
```

传播错误

通常我们会写样的代码，来传播错误。例如：

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

fn read_username_from_file() -> Result<String, io::Error> {
    let f = File::open("hello.txt");
    let mut f = match f {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut s = String::new();
    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}
```

如果函数调用返回 Err，则将错误信息返回。

在 rust 中可以使用"?"运算符进行简写错误传播。例如：

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

fn read_username_from_file() -> Result<String, io::Error> {
    let mut f = File::open("hello.txt")?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    Ok(s)
}
```

"?"运算符可以很好的支持链式调用。例如：

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

fn read_username_from_file() -> Result<String, io::Error> {
    let mut s = String::new();
    File::open("hello.txt")?.read_to_string(&mut s)?;
    Ok(s)
}
```