

# Rust 学习笔记

## Rust 学习笔记

1. 介绍
  - 1.1 学习资源
  - 1.2 下载和安装
2. 常见编程概念
  - 2.1 变量和可变性
  - 2.2 数据类型
  - 2.3 函数
  - 2.4 注释
  - 2.5 控制流
3. 认识所有权
  - 3.1 什么是所有权
  - 3.2 引用与借用
  - 3.3 Slice 类型
5. 枚举和模式匹配
  - 5.1 枚举的定义
  - 5.2 match 控制流结构
  - 5.3 if let 和 let else 简洁控制流
6. 使用包、Crate 和模块管理项目
  - 6.1 包和 Crate
  - 6.2 定义模块来控制作用域与私有性
  - 6.3 引用模块树中项的路径
  - 6.4 使用 use 关键字将路径引入作用域
  - 6.5 将模块拆分成多个文件
7. 常见集合
  - 7.1 使用 Vector 储存列表
  - 7.2 使用字符串储存 UTF-8 编码的文本
  - 7.3 使用 Hash Map 储存键值对
8. 错误处理
  - 8.1 用 panic! 处理不可恢复的错误
  - 8.2 用 Result 处理可恢复的错误
  - 8.3 要不要 panic!
9. 泛型、Trait 和生命周期
  - 9.1 泛型数据类型
  - 9.2 Trait: 定义共同行为
  - 9.3 生命周期确保引用有效
10. 编写自动化测试
  - 10.1 如何编写测试
  - 10.2 控制测试如何运行
  - 10.3 测试的组织结构
11. 函数式语言特性: 迭代器与闭包

## 1. 介绍

## 1.1 学习资源

- Rust 官方学习文档：
  1. 英文版: [The Rust Programming Language](#)
  2. 中文版: [Rust 程序设计语言](#)

## 1.2 下载和安装

可以按照以下步骤下载和安装 Rust

1. 官网提供的 Linux 系统的安装命令如下

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

2. 如果无法下载, 尝试下面方法

```
# 1. 下载脚本 (使用镜像加速)
$ curl -fSL https://mirrors.ustc.edu.cn/rust-static/rustup.sh -o
rustup.sh

# 2. 修改权限
$ chmod +x rustup.sh

# 3. 运行安装脚本
$ RUSTUP_DIST_SERVER=https://mirrors.ustc.edu.cn/rust-static \
> RUSTUP_UPDATE_ROOT=https://mirrors.ustc.edu.cn/rust-static/rustup \
> ./rustup.sh
```

然后激活环境变量

```
$ source $HOME/.cargo/env
```

## 2. 常见编程概念

### 2.1 变量和可变性

- 变量: 默认是不可改变的 (immutable), 但可以在变量名前添加 `mut` 来使其可变

```
let x = 5;
let mut y = 6;
```

- **常量** (constants): 是绑定到一个名称的不允许改变的值

1. 不允许对常量使用 `mut`
2. 声明常量使用 `const` 关键字而不是 `let`，并且**必须**注明值的类型
3. Rust 对常量的命名约定是在单词之间使用全大写加下划线

```
const THREE_HOURS_IN_SECONDS: u32 = 60 * 60 * 3;
```

- **遮蔽** (Shadowing): 指第一个变量被第二个变量遮蔽，此时任何使用该变量名的行为中都会视为是在使用第二个变量，直到第二个变量自己也被遮蔽或第二个变量的作用域结束。可以用相同变量名称来遮蔽一个变量，以及重复使用 `let` 关键字来多次遮蔽

1. 遮蔽再次使用 `let` 实际上创建了一个新变量，并且可以改变值的类型和复用这个名字
2. 而 `mut` 相比于遮蔽，不可以改变值的类型

```
// 遮蔽的用法
let x = 5;
let x = x + 1; // x = 6
{
    let x = x * 2; // x = 12
} // x = 6
```

```
// 遮蔽可以改变值的类型
let spaces = " ";
let spaces = spaces.len();
```

```
// ❌ 错误示例: mut 不可以改变值的类型
let mut spaces = " ";
spaces = spaces.len();
```

## 2.2 数据类型

- Rust 中，每一个值都有一个特定的**数据类型** (data type)，数据类型分为**标量** (scalar) 和**复合** (compound)
- Rust 是 **静态类型** (statically typed) 语言，在编译时就必须知道所有变量的类型。当多种类型均有可能时，必须增加**类型注解**

```
let guess: u32 = "42".parse().expect("Not a number!"); // 增加类型注解 :  
u32
```

- **标量类型**：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**

1. **整型**：没有小数部分的数字。 `isize` 和 `usize` 类型依赖运行程序的计算机架构：64 位架构上它们是 64 位的，32 位架构上它们是 32 位的

长度	有符号	无符号
8-bit	<code>i8</code>	<code>u8</code>
16-bit	<code>i16</code>	<code>u16</code>
32-bit	<code>i32</code>	<code>u32</code>
64-bit	<code>i64</code>	<code>u64</code>
128-bit	<code>i128</code>	<code>u128</code>
架构相关	<code>isize</code>	<code>usize</code>

2. **浮点型**

```
let x = 2.0; // f64 类型  
let y: f32 = 3.0; // f32 类型
```

3. **布尔类型**

4. **字符类型**

- **复合类型**：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）

1. **元组类型**：是一个将多个不同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

```
let tup: (i32, f64, u8) = (500, 6.4, 1);  
let (x, y, z) = tup; // 解构元组值  
let five_hundred = tup.0; // 用点号 (.) 后跟索引值来直接访问元组元素
```

2. **数组类型**：数组中的每个元素的类型必须相同，并且数组长度固定

```
let a = [1, 2, 3, 4, 5];
let a: [i32; 5] = [1, 2, 3, 4, 5]; // 包含数组元素类型和元素数量
let a = [3; 5]; // 指定初始值和元素个数, 与 let a = [3, 3, 3, 3, 3] 效果相同
let first = a[0]; // 访问数组元素
```

## 2.3 函数

- 语句和表达式

1. 语句 (Statements) 是执行一些操作但不返回值的指令
2. 表达式 (Expressions) 计算并产生一个值

```
let y = {
    let x = 3;
    x + 1
}; // y = 4
```

- Rust 中通过输入 `fn` 后面跟着函数名和一对圆括号来定义函数

1. 可以定义为拥有 **参数** (parameters) 的函数
2. 函数可以向调用它的代码**返回值**

```
fn main() {
    let x = plus_one(5);
}
fn plus_one(x: i32) → i32 {
    x + 1 // 相当于 return x + 1;
}
```

## 2.4 注释

- Rust 中惯用的注释样式是以两个斜杠开始注释, 并持续到本行的结尾

1. 单行注释: 使用 `//` 开头
2. 多行注释: 使用 `/*` 开头, `*/` 结束
3. 文档注释: 使用 `///` 或 `//!` 开头

## 2.5 控制流

- `if` 表达式：允许根据条件执行不同的代码分支

1. 代码中的条件**必须**是 `bool` 值
2. 可以在 `let` 语句右侧使用 `if` 表达式
3. `if` 的每个分支的可能的返回值都**必须**是相同类型

```
// if、else if、else语句的基本用法
let num = 6;
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 3 or 2");
}
```

```
// ❌ 错误示例：代码中条件必须是 bool 值
let num = 3;
if num {
    println!("number was three");
}
```

```
// 在 let 语句右侧使用 if 表达式
let num = if true { 5 } else { 6 };
```

```
// ❌ 错误示例：if 的每个分支的可能的返回值都必须是相同类型
let num = if condition { 5 } else { "six" };
```

- `loop` 关键字：重复执行一段代码直到明确要求停止
1. 可以使用 `break` 关键字停止循环，`continue` 关键字跳过某次循环
  2. 可以从循环返回值
  3. 可以指定循环标签，将标签与 `break` 或 `continue` 一起使用

```
// 从循环返回值
let mut counter = 0;
let result = loop {
    counter += 1;
    if counter == 10 {
        break counter * 2; // 返回值 20
    }
};
```

```
// 使用循环标签打破外层循环
let mut count = 0;
'counting_up: loop {
    let mut remaining = 10;
    loop {
        if remaining == 9 {
            break;
        }
        if count == 2 {
            break 'counting_up;
        }
        remaining -= 1;
    }
    count += 1;
}
```

- **while** 条件循环：当条件为 **true**，执行循环。当条件不再为 **true**，调用 **break** 停止循环

```
let mut num = 3;
while num != 0 {
    num -= 1;
}
```

- **for** 循环遍历集合：对一个集合的每个元素执行一些代码

```
let a = [10, 20, 30, 40, 50];
for element in a {
    println!("the value is: {element}");
}
```

### 3. 认识所有权

## 3.1 什么是所有权

- 变量作用域：是一个项（item）在程序中有效的范围

```
{                                // s 在这里无效，它尚未声明
    let s = "hello";             // 从此处起，s 是有效的
    // 使用 s
}
```

- **所有权**（ownership）是 Rust 用于如何管理内存的一组规则

Rust 中的每一个值都有一个 **所有者**（owner）

值在任一时刻有且只有一个所有者

当所有者离开作用域，这个值将被丢弃

1. 移动堆上的数据后，所有权转移。但是可以使用 `clone` 方法深度复制堆中的数据

```
// 所有权转移
let s1 = String::from("hello");
let s2 = s1; // s1 的所有权转移给了 s2，之后 s1 不再有效
println!("{s1}, world!");
```

```
// 深度复制堆中数据
let s1 = String::from("hello");
let s2 = s1.clone();
println!("s1 = {s1}, s2 = {s2}");
```

2. 对于整型这样存储在栈上的数据，拷贝实际的值是快速的

```
let x = 5;
let y = x;
println!("x = {x}, y = {y}");
```

3. 堆上的数据作为参数传入函数后，所有权转移。但是函数的返回值可以转移所有权

```
// 向函数传递值会使所有权转移
fn main() {
    let s = String::from("hello"); // s 进入作用域
    takes_ownership(s);             // s 的值移动到函数里 ...
```

```

let x = 5;
makes_copy(x);

println!("{}", x);
} // 这里, x 先移出了作用域, 然后是 s。但因为 s 的值已被移走,
// 没有特殊之处

fn takes_ownership(some_string: String) { // some_string 进入作用域
    println!("{}", some_string);
} // 这里, some_string 移出作用域并调用 `drop` 方法。
// 占用的内存被释放

fn makes_copy(some_integer: i32) { // some_integer 进入作用域
    println!("{}", some_integer);
} // 这里, some_integer 移出作用域。没有特殊之处

```

```

// 通过返回值转移所有权
fn main() {
    let s1 = gives_ownership(); // gives_ownership 将它的返回值
    // 传递给 s1
    let s2 = String::from("hello"); // s2 进入作用域
    let s3 = takes_and_gives_back(s2); // s2 被传入
    // takes_and_gives_back,
    // 它的返回值又传递给 s3
} // 此处, s3 移出作用域并被丢弃。s2 被 move, 所以无事发生
// s1 移出作用域并被丢弃

fn gives_ownership() -> String { // gives_ownership 将会把返回值
    // 传入
    // 调用它的函数
    let some_string = String::from("yours"); // some_string 进入作用
    // 域
    some_string // 返回 some_string 并将其移至
    // 调用函数
}
// 该函数将传入字符串并返回该值

fn takes_and_gives_back(a_string: String) -> String {
    // a_string 进入作用域
    a_string // 返回 a_string 并移出给调用的函数
}

```

## 3.2 引用与借用

- **引用** (*reference*) 像一个指针, 因为它是一个地址, 可以由此访问储存于该地址的属于其他变量的数据。引用不获取所有权, 但能访问数据, 通过 `&` 符号创建

1. 引用分为**不可变引用**和**可变引用**，在任意给定时间，要么只能有一个可变引用，要么只能有多个不可变引用
2. 引用必须总是有效的（Rust 编译器确保引用永远不会变成悬垂引用）

```
// 不可变引用
fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1); // 创建一个指向值 s1 的不可变引用
    println!("The length of '{s1}' is {len}.");
}

fn calculate_length(s: &String) -> usize { // s 是 String 的引用
    s.len()
} // 这里，s 离开了作用域。但因为它并不拥有引用值的所有权，所以什么也不会发生
```

```
// 可变引用
fn main() {
    let mut s = String::from("hello");
    change(&mut s); // 创建一个指向值 s 的可变引用
}

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

```
// ✗ 在同一作用域中，对同一数据只能有一个可变引用
let mut s = String::from("hello");
let r1 = &mut s;
let r2 = &mut s; // 会导致数据竞争(竞态条件)
println!("{}", r1, r2);
```

```
// ✗ 在同一作用域中，对同一数据不能同时存在可变引用和不可变引用
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!("{}", r1, r2, r3);
```

## 3.3 Slice 类型

- 切片（*slice*）允许引用集合中一段连续的元素序列，而不用引用整个集合

1. slice 是一种引用，所以它不拥有所有权

## 5. 枚举和模式匹配

### 5.1 枚举的定义

- **枚举** (*enumerations*), 也被称作 *enums*, 允许通过列举可能的**变体** (*variants*) 来定义一个类型

1. 可以**枚举**出所有可能的值
2. 每个变体可以处理不同类型和数量的数据
3. 可以使用 `impl` 来为枚举定义方法

```
// 枚举的基本用法
enum Message {
    Quit, // 没有关联任何数据
    Move { x: i32, y: i32 }, // 类似结构体包含命名字段
    Write(String), // 包含单独一个 String
    ChangeColor(i32, i32, i32), // 包含三个 i32
}

fn process_message(msg: Message) {
    match msg {
        Message::Quit => println!("Quit message received"),
        Message::Move { x, y } => println!("Move to coordinates: ({}, {})", x, y),
        Message::Write(text) => println!("Write message: {}", text),
        Message::ChangeColor(r, g, b) => println!("Change color to: ({}, {}, {})", r, g, b),
    }
}

fn main() {
    let messages = vec![
        Message::Quit,
        Message::Move { x: 10, y: 20 },
        Message::Write(String::from("hello")),
        Message::ChangeColor(255, 0, 0),
    ];
    for message in messages {
        process_message(message);
    }
}
```

```
}
```

```
// 使用 impl 来为枚举定义方法
impl Message {
    fn call(&self) {
        // 在这里定义方法体
    }
}

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

- `Option<T>` 是**标准库**定义的一个枚举，编码了一个值要么有值要么没值的场景
  1. Rust 并没有很多其他语言中有的空值功能。**空值**（*Null*）是一个值，代表没有值
  2. `Option<T>` 枚举被包含在了 `prelude` 之中，无需将其显式引入作用域，可以直接使用
  3. 在对 `Option<T>` 进行运算之前必须将其转换为 `T`（使用 `unwrap()` 或 `match` 模式匹配）
  4. 只要一个值不是 `Option<T>` 类型，就**可以安全**的认定它的值不为空

```
// Rust 标准库中 Option<T> 的定义
enum Option<T> {
    None,
    Some(T),
}
```

## 5.2 match 控制流结构

- `match` 作为一个控制流运算符，可以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
  1. 模式可以由**字面值**、**变量**、**通配符**和许多其他内容构成
  2. 如果想要在分支中运行多行代码，可以使用大括号，而分支后的逗号是可选的
  3. 每个分支相关联的代码是一个表达式，而表达式的结果值将作为整个 `match` 表达式的返回值
  4. `match` 的分支**必须覆盖所有的可能性**，不然就会报错（可以用 `other` 或 `_` 占位符匹配其他所有值）

```
enum Coin {
    Penny,
    Nickel,
    Dime,
```

```

        Quarter,
    }
}

fn value_in_cents(coin: Coin) → u8 {
    match coin {
        Coin::Penny ⇒ {
            println!("Lucky penny!");
            1
        }, // 这里的逗号是可选地，可以写也可以不写
        Coin::Nickel ⇒ 5,
        Coin::Dime ⇒ 10,
        Coin::Quarter ⇒ 25,
    }
}

```

```

// ❌ 错误示例: match 分支必须覆盖所有的可能性，这里没有覆盖 None
fn plus_one(x: Option<i32>) → Option<i32> {
    match x {
        Some(i) ⇒ Some(i + 1),
    }
}

```

```

// 使用 other 或 _ 占位符匹配其他所有值
let roll = 9;
match roll {
    3 ⇒ println!("You rolled a 3!"),
    7 ⇒ println!("You rolled a 7!"),
    other ⇒ (), // () 表示不做任何操作
    // 也可以改成 _ ⇒ ()
}

```

## 5.3 if let 和 let else 简洁控制流

- `if let` 语法用于处理只匹配一个模式的值而忽略其他模式的情况
- 1. `if let` 可以编写更少的代码，更少的缩进和更少的样板代码。但是也会失去 `match` 强制要求的穷尽性检查来确保没有忘记处理某些情况
- 2. 可以在 `if let` 中包含一个 `else`。 `else` 块中的代码与 `match` 表达式中的 `_` 分支块中的代码相同

```
let config_max = Some(3u8);
if let Some(max) = config_max {
    println!("The maximum is configured to be {max}");
} else {
    println!("The maximum is not configured.");
}
```

- `let...else`

## 6. 使用包、Crate 和模块管理项目

- 一个包 (*package*) 可以包含多个二进制 `crate` 项和一个可选的库 `crate`

- **包** (*Packages*): Cargo 的一个功能, 它允许构建、测试和分享 `crate`
- **Crates**: 一个模块的树形结构, 它形成了库或可执行文件项目
- **模块** (*Modules*) 和 **use**: 允许控制作用域和路径的私有性
- **路径** (*path*): 一个为例如结构体、函数或模块等项命名的方式

### 6.1 包和 Crate

- **包** (*package*) 是提供一系列功能的一个或者多个 `crate` 的捆绑 (Cargo 实际上就是一个包)
  1. 包会包含一个 *Cargo.toml* 文件, 阐述如何去构建这些 `crate`
  2. 包中至多可以包含一个库 `crate`, 可以包含任意多个二进制 `crate`, 但必须至少包含一个 `crate`
- `crate` 有二进制 `crate` 和库 `crate` 两种形式

```
# 创建二进制 crate, 在 my_binary_crate/src/main.rs 中编写代码
cargo new my_binary_crate

# 创建库 crate, 在 my_library_crate/src/lib.rs 中编写代码
cargo new my_library_crate --lib
```

## 6.2 定义模块来控制作用域与私有性

### 模块小抄 (Cheat Sheet)

- **从 crate 根节点开始**：当编译一个 crate, 编译器首先在 crate 根文件（通常，对于一个库 crate 而言是 `src/lib.rs`，对于一个二进制 crate 而言是 `src/main.rs`）中寻找需要被编译的代码
  - **声明模块**：在 crate 根文件中，可以声明一个新模块；比如，用 `mod garden`；声明了一个叫做 `garden` 的模块。编译器会在下列路径中寻找模块代码：
    - 内联，用大括号替换 `mod garden` 后跟的分号
    - 在文件 `src/garden.rs`
    - 在文件 `src/garden/mod.rs`
  - **声明子模块**：在除了 crate 根节点以外的任何文件中，你可以定义子模块。比如，你可能在 `src/garden.rs` 中声明 `mod vegetables`；。编译器会在以父模块命名的目录中寻找子模块代码：
    - 内联，直接在 `mod vegetables` 后方不是一个分号而是一个大括号
    - 在文件 `src/garden/vegetables.rs`
    - 在文件 `src/garden/vegetables/mod.rs`
  - **模块中的代码路径**：若一个模块是 crate 的一部分，则可以在隐私规则允许的前提下，从同一个 crate 内的任意地方，通过代码路径引用该模块的代码。比如，一个 `garden vegetables` 模块下的 `Asparagus` 类型可以通过 `crate::garden::vegetables::Asparagus` 访问
  - **私有 vs 公用**：一个模块里的代码默认对其父模块私有。为了使一个模块公用，应当在声明时使用 `pub mod` 替代 `mod`。为了使一个公用模块内部的成员公用，应当在声明前使用 `pub`
  - **use 关键字**：在一个作用域内，`use` 关键字创建了一个项的快捷方式，用来减少长路径的重复。在任何可以引用 `crate::garden::vegetables::Asparagus` 的作用域，可以通过 `use crate::garden::vegetables::Asparagus`；创建一个快捷方式，然后你就可以在作用域中只写 `Asparagus` 来使用该类型
- 使用 `mod` 关键字定义**模块**，可以将一个 crate 中的代码进行分组，以提高可读性与重用性
    1. 模块中的代码默认是私有的，可以利用模块控制项的**私有性** (`privacy`)

## 6.3 引用模块树中项的路径

- Rust 中路径有两种形式，绝对路径和相对路径。两种路径都后跟一个或多个由双冒号（`::`）分割的标识符
  1. **绝对路径**（*absolute path*）是以 crate 根（root）开头的完整路径；对于外部 crate 的代码，是以 crate 名开头的绝对路径，对于当前 crate 的代码，则以字面值 `crate` 开头
  2. **相对路径**（*relative path*）从当前模块开始，以 `self`、`super` 或当前模块中的某个标识符开头
    - 使用 `super` 可以从父模块开始构建相对路径，而不是从当前模块或者 crate 根开始（类似以 `..` 语法开始一个文件系统路径）
- `pub` 关键字用于控制模块的可见性
  1. 使模块公有并不使其内容也是公有的
  2. 对于结构体，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
  3. 对于枚举，枚举定义的前面使用了 `pub`，则它的所有变体都将变为公有

## 6.4 使用 use 关键字将路径引入作用域

- `use` 关键字可以创建一个**捷径**，在作用域中的任何地方使用这个更短的名字
  1. `use` 只能创建 `use` 所在的特定作用域内的捷径
  2. `use` 的习惯用法：函数优先引入父模块以明确来源、结构体 / 枚举直接引入完整路径、同名项需要通过父模块区分
  3. 可以使用嵌套路径来清理大量的 `use` 列表
  4. `pub` 与 `use` 组合使用的方法被称为**重导出**，可以让作用域之外的代码能够像在当前作用域中一样使用该名称

```
// 使用 use 创建的捷径
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();
}
```

```
// ❌ 错误示例：子模块无法直接访问父模块中 use 引入的路径
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting;

mod customer {
    pub fn eat_at_restaurant() {
        hosting::add_to_waitlist(); // customer 的子模块不同于 use 语句的
        作用域
    }
}
```

```
// 将 hosting 引入到 customer 模块中
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

mod customer {
    use crate::front_of_house::hosting;
    pub fn eat_at_restaurant() {
        hosting::add_to_waitlist();
    }
}
```

```
// 未经嵌套的 use 路径
use std::cmp::Ordering;
use std::io;
use std::io::Write;
```

```
// 嵌套后的 use 路径
use std::{cmp::Ordering, io::{self, Write}};
```

- **as** 关键字可以指定一个新的本地名称或者**别名**

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

- 使用外部包

- glob 运算符将一个路径下**所有**公有项引入作用域，在指定路径后跟 `*` glob 运算符
  1. 需要小心使用，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

```
// 使用 * glob 运算符
use std::collections::*;
```

## 6.5 将模块拆分成多个文件

- Rust 允许将一个包拆分为多个 crate，并将一个 crate 拆分为若干模块，从而可以在一个模块中引用另一个模块中定义的项

## 7. 常见集合

### 7.1 使用 Vector 储存列表

- `Vec<T>` 类型也被称为 vector，允许在一个单独的数据结构中储存多个值，它在内存中彼此相邻地排列所有的值
  1. vector 只能储存相同类型的值
  2. vector 是用泛型实现的
- vector 的**方法函数**
  1. 新建 vector

```
// 新建 vector, 可以调用 Vec::new 函数
let v: Vec<i32> = Vec::new(); // 新建一个空 vector
let v = vec![1, 2, 3]; // 使用 vec! 宏来新建一个包含初值的 vector
```

#### 2. 更新 vector

```
// 更新 vector, 可以使用 push 方法
let mut v = Vec::new();
v.push(5);
v.push(6);
```

#### 3. 读取 vector 的元素

```
// 通过索引方法, 获得索引位置元素的引用
let v = vec![1, 2, 3, 4, 5];
let third: &i32 = &v[2];
println!("The third element is {third}");

// 通过 get 方法, 获得用于 match 的 Option<&T>
let third: Option<&i32> = v.get(2);
match third {
    Some(third) => println!("The third element is {third}"),
    None => println!("There is no third element."),
}
```

```
// ❌ 错误示例: 获取数组长度外的索引
let v = vec![1, 2, 3, 4, 5];
let does_not_exist = &v[100]; // 会造成 panic
let does_not_exist = v.get(100); // 会返回 None
```

```
// ❌ 错误示例: 在拥有 vector 中项的引用的同时向其增加一个元素
let mut v = vec![1, 2, 3, 4, 5];
let first = &v[0];
v.push(6); // 可能会要求分配新内存, 并将老的元素拷贝到新的空间中, 导致第一个元素的引用指向被释放的内存
println!("The first element is: {first}");
```

#### 4. 遍历 vector 的元素

```
// 遍历 vector 中元素的不可变引用
let v = vec![100, 32, 57];
for i in &v { // 如果使用 for i in v 的话, v 的所有权会在 for 循环开始时转移
    println!("{i}");
}
```

```
// 遍历 vector 中元素的可变引用
let mut v = vec![100, 32, 57];
for i in &mut v {
    *i += 50;
}
```

#### 5. 使用枚举来存储多种类型

```
enum SpreadsheetCell {
    Int(i32),
    Float(f64),
    Text(String),
}

let row = vec![
    SpreadsheetCell::Int(3),
    SpreadsheetCell::Text(String::from("blue")),
    SpreadsheetCell::Float(10.12),
];
```

## 7.2 使用字符串储存 UTF-8 编码的文本

- `String` 类型是一种可增长、可变、可拥有、UTF-8 编码的字符串类型
- 字符串的方法函数

### 1. 新建字符串

```
// 新建一个空的 String
let mut s = String::new();
```

```
// 使用 to_string 方法从字符串字面值创建 String
let s = "initial contents".to_string();
```

```
// 使用 String::from 函数从字符串字面值创建 String
let s = String::from("initial contents");
```

### 2. 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 将一个字符加入 String 值中
let mut s = String::from("lo");
s.push('l');
```

```
// 使用 + 运算符将两个 String 值合并到一个新的 String 值中
let s1 = String::from("Hello, ");
let s2 = String::from("world!");
let s3 = s1 + &s2; // 注意 s1 被移动了, 不能继续使用
```

```
// 使用 format! 宏返回 String 值
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");
let s = format!("{s1}-{s2}-{s3}");
```

```
// 使用 replace 将所有目标子串替换为新子串
let s = String::from("I like dogs");
let s1 = s.replace("dogs", "cats"); // 此时 s 为 String::from("I like
cats")
```

### 3. 字符串索引

```
// 无法通过索引的方式去访问字符串中的某个字符, 但是可以使用切片的方式
&s1[start..end], 且需要保证 start 和 end 必须准确落在字符的边界处
let s1 = String::from("hi,中国");
let h = &s1[0..1]; // `h` 字符在 UTF-8 格式中只需要 1 个字节来表示
assert_eq!(h, "h");
let h1 = &s1[3..6]; // `中` 字符在 UTF-8 格式中需要 3 个字节来表示
assert_eq!(h1, "中");
```

### 4. 遍历字符串的方法

```
// 调用 chars 方法将其分开并返回两个 char 类型的值
for c in "3д".chars() { // 也可以用 for c in String::from("3д").chars()
    println!("{}", c);
}
// 打印如下内容:
// 3
// д
```

```
// 调用 bytes 方法返回每一个原始字节
for b in "3д".bytes() {
    println!("{b}");
}

// 打印如下内容:
// 208
// 151
// 208
// 180
```

## 7.3 使用 Hash Map 储存键值对

- `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
- 哈希 map 的方法函数

### 1. 新建哈希 map

```
// 新建一个哈希 map 并插入一些键值对
use std::collections::HashMap; // HashMap 没有被 prelude 自动引用, 所以需要手动导入

let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);
```

## 8. 错误处理

- Rust 将错误分为两大类：**可恢复的**（*recoverable*）和 **不可恢复的**（*unrecoverable*）错误

### 8.1 用 panic! 处理不可恢复的错误

- `panic!` 宏用于处理不可恢复的错误
- 1. 在实践中有两种方法造成 panic：执行会造成代码 panic 的操作（如访问超过数组结尾的内容）或者显式调用 `panic!`

```
// panic 的方式 1: 显式调用 panic!
fn main() {
    panic!("crash and burn");
}
```

```
// panic 的方式 2: 执行会造成代码 panic 的操作
fn main() {
    let v = vec![1, 2, 3];
    v[99]; // vector 越界访问
}
```

## 8.2 用 Result 处理可恢复的错误

- `Result<T, E>` 是标准库定义的一个枚举，编码了操作要么成功要么失败的场景

```
// Rust 标准库中 Result<T, E> 的定义
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

```
// match 匹配不同的错误类型
use std::fs::File;
use std::io::ErrorKind;
fn main() {
    let greeting_file_result = File::open("hello.txt"); // File::open
    的返回值是 Result<T, E>
    let greeting_file = match greeting_file_result {
        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:?}"),
            },
            _ => {
                panic!("Problem opening the file: {error:?}");
            }
        },
    };
}
```

- `unwrap` 和 `expect` 是 `Option<T>` 和 `Result<T, E>` 类型提供的方法，用于提取内部的有效值，也是失败时 panic 的快捷方式
  1. 对 `Option<T>`：若为 `Some(T)`，返回内部的 `T`；若为 `None`，直接 panic
  2. 对 `Result<T, E>`：若为 `Ok(T)`，返回内部的 `T`；若为 `Err(E)`，直接 panic 并打印错误信息（`unwrap` 为默认错误信息，`expect` 为自定义错误信息）

```
// unwrap 方法和 expect 方法是失败时 panic 的快捷方式
use std::fs::File;
fn main() {
    // 使用 unwrap 方法
    let greeting_file = File::open("hello.txt").unwrap();
    // 使用 expect 方法
    let greeting_file = File::open("hello.txt")
        .expect("hello.txt should be included in this project");
}
```

- **传播错误**：函数的实现中调用了可能会失败的操作时，除了在这个函数中处理错误外，还可以选择让调用者知道这个错误并决定该如何处理
  1. `?` 运算符是处理 `Result<T, E>` 和 `Option<T>` 类型的语法糖，可以简化错误的传播逻辑，避免重复编写 `match` 表达式处理分支。且 `?` 只能用在返回值为 `Result<T, E>` 或 `Option<T>` 的函数中

```
// 传播错误示例：手动实现
use std::fs::File;
use std::io::{self, Read};
fn read_username_from_file() -> Result<String, io::Error> {
    let username_file_result = File::open("hello.txt"); //
    username_file_result 类型是 Result<File, io::Error>
    let mut username_file = match username_file_result { //
    username_file 类型是 File
        Ok(file) => file,
        Err(e) => return Err(e),
    };
    let mut username = String::new();
    // 函数签名解释: fn read_to_string(&mut self, buf: &mut String) ->
    Result<usize, io::Error>
    match username_file.read_to_string(&mut username) {
        Ok(_) => Ok(username),
        Err(e) => Err(e),
    }
}
```

```
// 传播错误示例：语法糖简化
use std::fs::File;
use std::io::{self, Read};
fn read_username_from_file() → Result<String, io::Error> {
    let mut username = String::new();
    File::open("hello.txt").read_to_string(&mut username)?;
    Ok(username)
}
```

## 8.3 要不要 panic!

# 9. 泛型、Trait 和生命周期

## 9.1 泛型数据类型

- **泛型**：作为高效处理重复概念的工具，是具体类型或其他属性的抽象替代
  1. 泛型允许我们使用一个可以代表多种类型的占位符来替换特定类型，以此来减少代码冗余
  2. 泛型并不会使程序比具体类型运行得慢，Rust 通过在编译时进行泛型代码的**单态化**（*monomorphization*）来保证效率

```
// 结构体定义中的泛型
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 };
}
```

```
// 枚举定义中的泛型
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

```
// 方法定义中的泛型
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> { // 必须在 impl 后面声明 T, Rust 才知道 Point 的尖括号中的
                    // 类型是泛型而不是具体类型
    fn x(&self) -> &T {
        &self.x
    }
}

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

## 9.2 Trait: 定义共同行为

- **trait** 是一种将方法签名组合起来的方法，用于定义共同行为（类似于其他语言中的[接口](#)（*interfaces*）的功能）
  1. 可以包含**仅有签名的方法**，也可以包含**带有默认实现的方法**
  2. 若 trait 中包含无默认实现的方法签名，实现该 trait 的类型必须提供这些方法的具体实现
  3. 可以有多个方法，无默认实现的方法签名以 `;` 结尾，有默认实现的方法需写具体逻辑（无需 `;`）

```
pub trait Summary {
    // 无默认实现的方法签名（必须提供具体实现）
    fn summarize(&self) -> String;

    // 有默认实现的方法（可选重写）
    fn summarize_author(&self) -> String {
        String::from("Unknown author")
    }
}
```

- 在类型上实现 trait

1. 与“在类型上实现方法”类似，但是语法为 `impl TraitName for TypeName`

```
// lib.rs
pub struct NewsArticle {
    pub headline: String,
```

```

    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle { // 类似于为类型实现方法(impl NewsArticle)
    fn summarize(&self) → String {
        format!("{}", by {} ({}), self.headline, self.author,
self.location)
    }
}

pub struct SocialPost {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub repost: bool,
}

impl Summary for SocialPost {
    fn summarize(&self) → String {
        format!("{}", self.username, self.content)
    }
}

```

```

// main.rs
use my_test::{SocialPost, Summary}; // my_test 是项目的 crate 名称, 定义在
Cargo.toml 的 name = "my_test" 中
fn main() {
    let post = SocialPost {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        repost: false,
    };
    println!("1 new post: {}", post.summarize());
}

```

- trait 作为参数

## 9.3 生命周期确保引用有效

- **生命周期**是描述**引用有效作用域**的概念，主要目标是避免**悬垂引用**
  1. Rust 中每个引用都有自己的生命周期
  2. 多数情况下，生命周期是隐式的、可被推断的
  3. 当引用的生命周期可能以不同的方式互相关联时，需要手动显式标注生命周期
  4. Rust 编译器有借用检查器 (*borrow checker*)，通过比较作用域来确保所有的借用都是有效的

```
// ❌ 错误示例：悬垂引用
fn main() {
    let r;                // -----+-- 'a
    {                    //          |
        let x = 5;        // -+-- 'b |
        r = &x;           //  |    |
    }                    // -+    |  内部花括号结束，x 被销毁（内存释
    放），此时 r 指向的内存已无效
    println!("r: {r}");   //          |
}                        // -----+
```

- **生命周期标注语法**

1. 生命周期的标注不会改变引用的生命周期长度，只是用于描述多个引用的生命周期期间的关系
2. 单个生命周期标注本身没有意义

```
// 生命周期参数名以 ' 开头，通常全小写且非常短（很多人使用 'a），标注在引用的 & 符号
后，使用空格将标注和引用类型分开
&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
```

- **函数签名中的生命周期标注**

1. 指定了泛型生命周期参数后，函数可以接收带有任何生命周期的引用
2. 从函数返回引用时，返回类型的生命周期参数需要与其中一个参数的生命周期匹配

```
// 泛型生命周期参数声明在函数名和参数列表之间的 < > 里
fn longest<'a>(x: &'a str, y: &'a str) → &'a str {
    if x.len() > y.len() { x } else { y }
}
```

```
// 这里不需要为参数 y 指定生命周期参数，因为 y 的生命周期与参数 x 和返回值的生命周期没有任何关系

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

```

- 结构体定义中的生命周期标注

1. struct 中的引用类型，需要在每个引用上添加生命周期标注

```
struct ImportantExcerpt<'a> { // 必须在结构体名称后面的 < > 中声明泛型生命周期参数
    part: &'a str, // 这个注解意味着 ImportantExcerpt 的实例不能比其 part 字段中的引用存在的更久
}

fn main() {
    let novel = String::from("Call me Ishmael. Some years ago...");
    let first_sentence = novel.split('.').next().unwrap();
    let i = ImportantExcerpt {
        part: first_sentence,
    };
}

```

- 生命周期省略

## 10. 编写自动化测试

### 10.1 如何编写测试

- Rust 提供了专门用来编写测试的功能：`test` 属性、`assert!` / `assert_eq!` / `assert_ne!` 等断言类宏和 `should_panic` 属性
- 1. `cargo test` 命令会运行项目中所有的测试
- 2. `#[cfg(test)]` 是 Rust 的条件编译属性，只有运行 `cargo test` 时，被它标记的代码才会被编译和包含到最终产物中
- 3. `#[test]` 是 Rust 测试框架的测试用例标记属性，作用是将普通函数标记为测试用例，运行测试时自动执行，且只能标记函数。发生 panic 表示测试失败

```
$ cargo test # 运行项目中所有的测试

```

- **测试函数体**通常执行如下三种操作：

1. 设置任何所需的数据或状态
2. 运行需要测试的代码
3. 断言其结果是我们所期望的

```
// src/lib.rs
pub fn add(left: u64, right: u64) → u64 {
    left + right
}

#[cfg(test)] // Rust 的条件编译标记，表示只有在运行测试时，这个模块才会被编译和
              执行
mod tests {
    use super::*; // 导入父模块的所有公共符号(这里是为了访问 add 函数)
    // 成功的测试用例
    #[test]
    fn it_works() {
        let result = add(2, 2);
        assert_eq!(result, 4); // Rust 内置的断言宏，用于验证两个值是否相等
    }
    // 失败的测试用例
    // #[test]
    // fn another() {
    //     panic!("Make this test fail");
    // }
}
```

- 使用**断言类宏**来检查结果

1. **assert!** 宏由标准库提供，需要向 **assert!** 宏提供一个求值为布尔值的参数。如果值是 **true**，**assert!** 什么也不做（测试通过）；如果值为 **false**，**assert!** 会调用 **panic!** 宏（导致测试失败）
2. **assert\_eq!** 宏由标准库提供，需要向其提供两个待比较的值作为参数。它会对这两个值执行相等性判断：如果值相等，**assert\_eq!** 什么也不做（测试通过）；如果值不相等，**assert\_eq!** 会调用 **panic!** 宏（导致测试失败），并自动打印两个值的具体内容
3. **assert\_ne!** 宏由标准库提供，需要向其提供两个待比较的值作为参数。它会对这两个值执行不等性判断：如果值不相等，**assert\_ne!** 什么也不做（测试通过）；如果值相等，**assert\_ne!** 会调用 **panic!** 宏（导致测试失败），并自动打印两个值的具体内容
4. 可以向 **assert!**、**assert\_eq!** 和 **assert\_ne!** 宏传递一个可选的失败信息参数，可以在测试失败时将自定义失败信息一同打印出来

```
// assert! 宏在测试用例中的使用
#[cfg(test)]
mod tests {
    #[test]
    fn test_sample_function_behavior() {
        let condition = true;
        assert!(condition);
    }
}
```

```
// ❌ 错误示例: assert! 触发 panic
#[cfg(test)]
mod tests {
    #[test]
    fn test_sample_function_behavior() {
        let condition = false;
        assert!(condition); // 会触发 panic, 导致测试失败
    }
}
```

```
// assert_eq! 宏在测试用例中的使用
pub fn add_two(a: usize) → usize {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn test_sample_function_behavior() {
        let result = add_two(2);
        assert_eq!(result, 4);
    }
}
```

```
// assert_ne! 宏在测试用例中的使用
pub fn add_two(a: usize) → usize {
    a + 3
}

#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn test_sample_function_behavior() {
        let result = add_two(2);
        assert_ne!(result, 4);
    }
}
```

```
// ❌ 错误示例: assert! 触发 panic, 并打印自定义的失败信息
pub fn greeting(name: &str) → String {
    String::from("Hello!")
}

#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn greeting_contains_name() {
        let result = greeting("Carol");
        assert!(
            result.contains("Carol"),
            "Greeting did not contain name, value was `{result}`"
        );
    }
}
```

- 使用 `should_panic` 检查 panic

1. `should_panic` 是 Rust 测试中用于标记测试函数的属性，用于验证某个操作**是否会触发 panic**：如果测试函数执行时确实发生了 panic，测试通过；如果未发生 panic，测试失败
2. 可以给 `should_panic` 属性增加一个可选的 `expected` 参数，测试工具会确保错误信息中包含其提供的文本

```
pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) → Guess {
```

```

        if value < 1 {
            panic!("Guess value must be greater than or equal to 1,
got {value}.");
        } else if value > 100 {
            panic!("Guess value must be less than or equal to 100, got
{value}.");
        }
        Guess { value }
    }
}
#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    #[should_panic(expected = "less than or equal to 100")] //
    // expected 参数提供的值是 Guess::new 函数 panic 信息的子串
    fn greater_than_100() {
        Guess::new(200);
    }
}

```

```

// ❌ 错误示例: panic 信息中未包含期望信息
pub struct Guess {
    value: i32,
}
impl Guess {
    pub fn new(value: i32) → Guess { // 将 if value < 1 和 else if
value > 100 的代码块互换
        if value < 1 {
            panic!("Guess value must be less than or equal to 100, got
{value}.");
        } else if value > 100 {
            panic!("Guess value must be greater than or equal to 1,
got {value}.");
        }
        Guess { value }
    }
}
#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    #[should_panic(expected = "less than or equal to 100")]
    fn greater_than_100() {
        Guess::new(200);
    }
}

```

```
}  
}
```

- 在测试中使用 `Result<T, E>`
  - 在 Rust 测试中，除了通过 `panic!`（或断言宏触发 panic）表示测试失败外，还可以让测试函数返回 `Result<T, E>` 类型：返回 `Ok(_)` 表示测试通过，返回 `Err(_)` 表示测试失败
  - `#[should_panic]` 用于验证测试函数会 panic，而返回 `Result<T, E>` 的测试失败时是返回 `Err` 而非 panic，因此两者不能同时使用。若需断言某个操作返回 `Err`，应通过 `assert!(value.is_err())` 等断言宏，而非 `#[should_panic]`

```
pub fn add(left: u64, right: u64) → u64 {  
    left + right  
}  
#[cfg(test)]  
mod tests {  
    use super::*;  
    #[test]  
    fn it_works() → Result<(), String> {  
        let result = add(2, 2);  
        if result == 4 {  
            Ok(())  
        } else {  
            Err(String::from("two plus two does not equal four"))  
        }  
    }  
}
```

## 10.2 控制测试如何运行

- `cargo test` 产生的二进制文件的默认行为是并发运行所有的测试。不过可以指定命令行参数来改变 `cargo test` 的默认行为
  - 参数传递规则：在 `--` 前是传递给 `cargo test` 的参数（`cargo test --help` 查看）；在 `--` 后是传递给测试二进制文件的参数（`cargo test -- --help` 查看）
  - 默认情况下，当测试通过时，Rust 的测试库会捕获打印到标准输出（`println!`）的所有内容，只有测试失败时才会显示出来
  - 可以使用 `ignore` 属性来标记耗时的测试并排除

```
# 控制线程的数量
$ cargo test -- --test-threads=1 # 控制线程的数量为 1
# 显示函数输出
$ cargo test -- --show-output # 显示成功测试的输出
# 通过名称运行测试
$ cargo test one_hundred # 运行单个测试: 只运行 one_hundred 这一个测试
$ cargo test add # 过滤运行多个测试: 任何名称包含 add 字符串的测试都会被运行
# 除非特别指定否则忽略某些测试
$ cargo test -- --ignored # 只运行被忽略的测试
$ cargo test -- --include-ignored # 不管是否忽略都要运行全部测试
```

```
#[cfg(test)]
mod tests {
    #[test]
    #[ignore] // 标记的函数不会被测试运行
    fn expensive_test() {
        // code that takes an hour to run
    }
}
```

## 10.3 测试的组织结构

- Rust 社区倾向于根据测试的两个主要分类来考虑问题: **单元测试** (*unit tests*) 与 **集成测试** (*integration tests*)
- **单元测试**是在与其他部分隔离的环境中测试每一个单元的代码, 以便于快速验证某个单元的代码功能是否符合预期
  1. 单元测试与它们要测试的代码共同存放在位于 *src* 目录下相同的文件中。规范是在每个文件中创建包含测试函数的 `tests` 模块, 并使用 `cfg(test)` 标注模块
  2. Rust 的私有性规则允许测试私有函数

```
// 单元测试示例
pub fn add(left: u64, right: u64) → u64 {
    left + right
}
#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn it_works() {
        let result = add(2, 2);
        assert_eq!(result, 4);
    }
}
```

```
// rust 允许测试私有函数
pub fn add_two(a: usize) → usize {
    internal_adder(a, 2)
}
fn internal_adder(left: usize, right: usize) → usize {
    left + right
}
#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn internal() {
        let result = internal_adder(2, 2); // 测试没有标记为 pub 的私有函数
        internal_adder
        assert_eq!(result, 4);
    }
}
```

- **集成测试**的目的是测试库的多个部分能否一起正常工作。一些单独能正确运行的代码单元集成在一起也可能会出现问题
  1. 为了编写集成测试，需要在项目根目录创建一个 *tests* 目录，与 *src* 同级
  2. 不需要将任何代码标注为 `#[cfg(test)]`

```
addor
├─ Cargo.lock
├─ Cargo.toml
├─ src
│   └─ lib.rs
└─ tests
    └─ integration_test.rs
```

```
// 集成测试示例
use addor::add_two; // addor 为 crate 名称, add_two 为 lib.rs 的函数
#[test]
fn it_adds_two() {
    let result = add_two(2);
    assert_eq!(result, 4);
}
```

## 11. 函数式语言特性：迭代器与闭包