rust学习笔记

时间：2024年4月21日13:10:45

作者：迪克聪

# 一、[Rust 生态系统](https://google.github.io/comprehensive-rust/zh-CN/cargo/rust-ecosystem.html" \l "rust-%E7%94%9F%E6%80%81%E7%B3%BB%E7%BB%9F)

Rust 生态系统由许多工具组成，主要包括：

rustc：Rust 编译器，可将 .rs 文件转换为二进制文件和其他中间格式。

cargo：Rust 依赖项管理器和构建工具。Cargo 知道如何下载托管在 [https://crates.io](https://crates.io/) 上的依赖项,并在构建项目时将它们传递给 rustc。Cargo 还附带一个内置的测试运行程序，用于执行单元测试。

rustup：Rust 工具链安装和更新工具。当 Rust 发布新版本时，此工具用于安装并更新 rustc 和 cargo。 此外，rustup 还可以下载标准库的文档。可以同时安装多个版本的 Rust，rustup 会根据需要让你在这些版本之间切换。

Rust 有一个快速发布时间表，每六周就会发布一次新版本。新版本保持与旧版本的向后兼容性，并添加新功能。

共有三个发布阶段：“稳定版（stable）”、“测试版（beta）”和“夜间版（nightly）”。

新功能会先在“夜间版”上测试，“测试版”会每六周转为“稳定版”

Rust 区分 版本（edition）：当前版本是 Rust 2021。之前的版本是 Rust 2015 和 Rust 2018。

这些版本支持对语言进行向后不兼容的更改。

为防止破坏代码，版本是可选的： 通过 Cargo.toml 文件为 crate 选择合适的版本。

## 1.1 cargo和rust的关系

[Rust之cargo简单熟悉 | DD'Notes (z-dd.online)](https://notes.z-dd.online/2023/05/09/Rust%E4%B9%8Bcargo%E7%AE%80%E5%8D%95%E7%86%9F%E6%82%89/)

cargo是什么？为什么需要这个cargo？

cargo是Rust的包管理器，Rust的包分为2种，一种是**二进制**可执行的包，一种是库的包，默认情况下就是第一种binary包

在Rust里，一个库或者可执行程序叫做一个crate，一般简单情况下，我们可以直接使用Rust的编译器rustc来编译这个crate（箱），但如果编译这个crate需要许多编译参数，包含许多依赖，或者需要其他构建工具，手动去操作就会很麻烦，而且也很难管理，这样包管理器cargo就应运而生了

cargo主要做以下4个工作：

①引入两个包含包信息的元数据文件

②获取并构建包的依赖项

③用正确的参数调用rustc或其他构建工具来生成包

④引入规则约定，使使用Rust包变得更容易

值得一提的是，Cargo 本身就是一个功能强大且全面的工具。它能够实现许多高级功能，包括但不限于：

项目/软件包结构

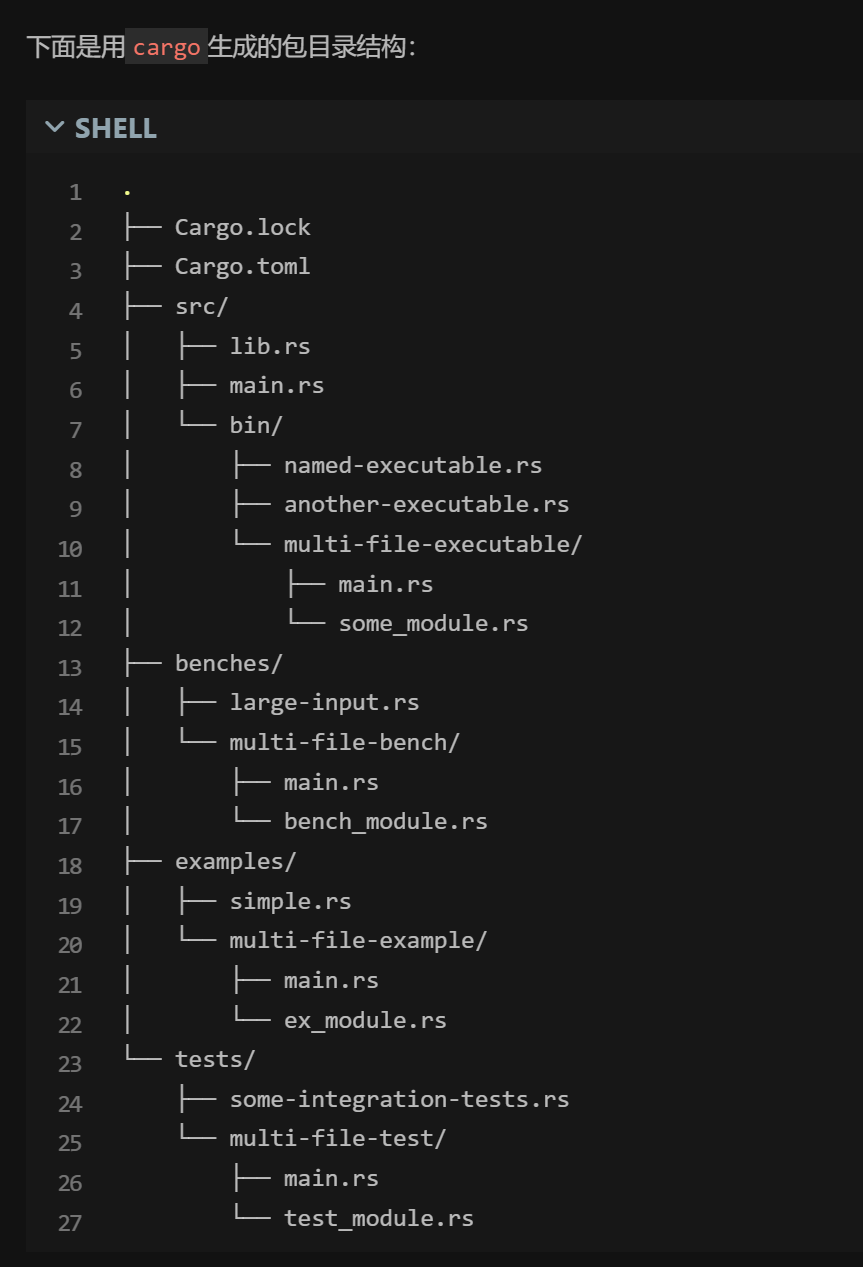
工作区

开发依赖和运行时依赖管理/缓存

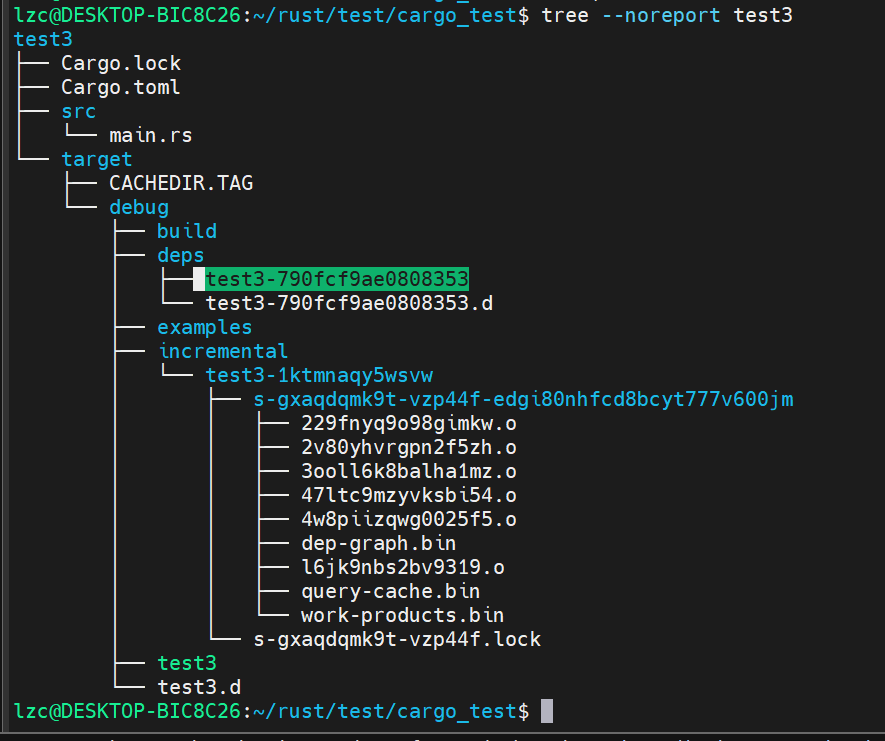
构建脚本

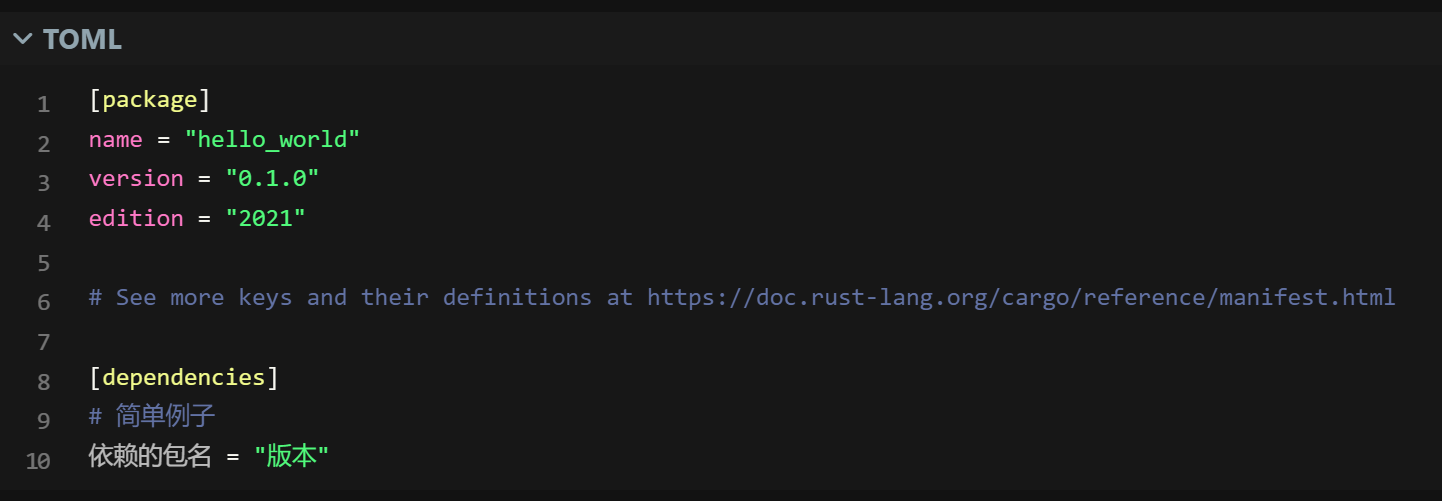
全局安装

它还可以使用子命令插件（例如 cargo clippy）进行扩展。

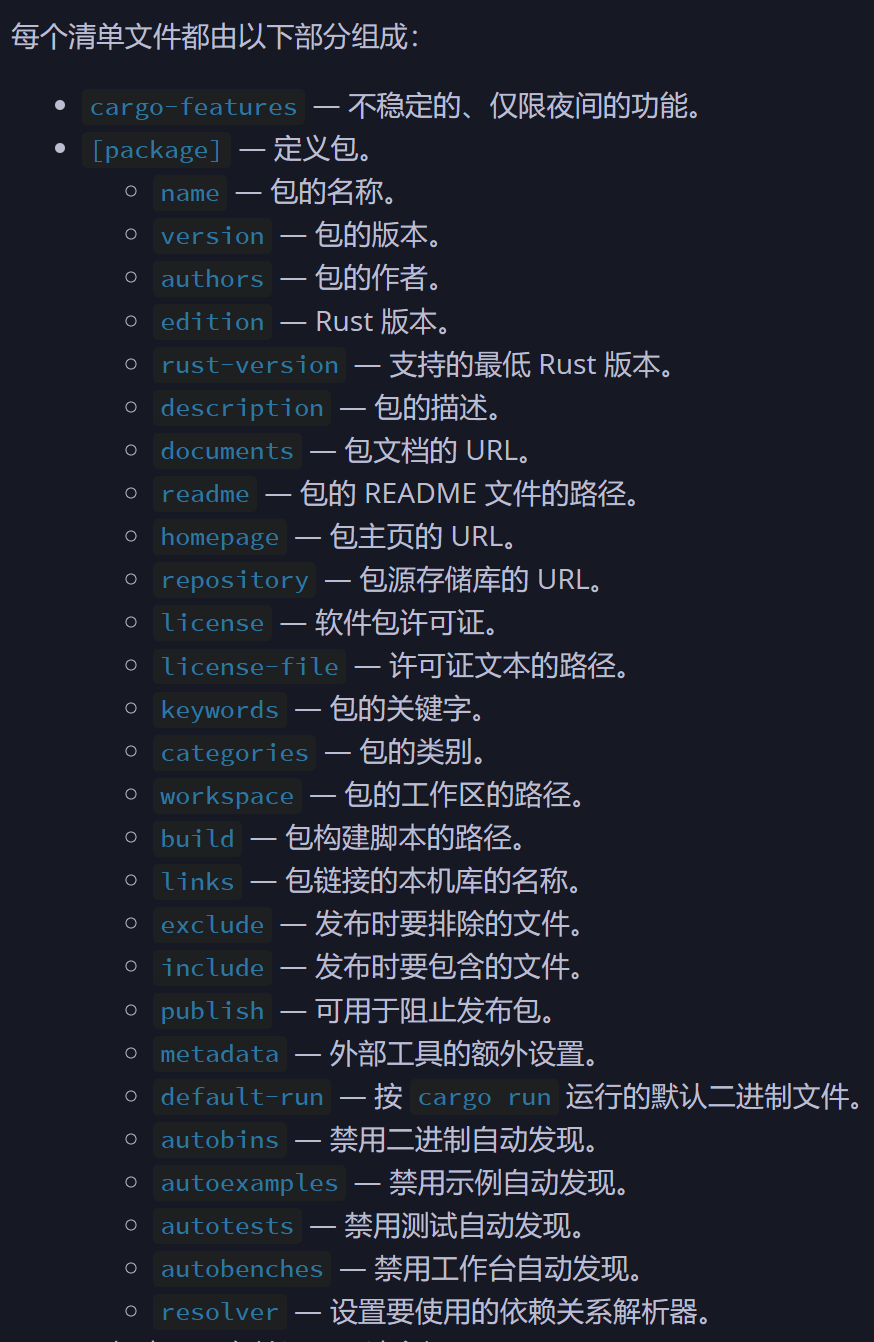


## 1.2 cargo.toml文件

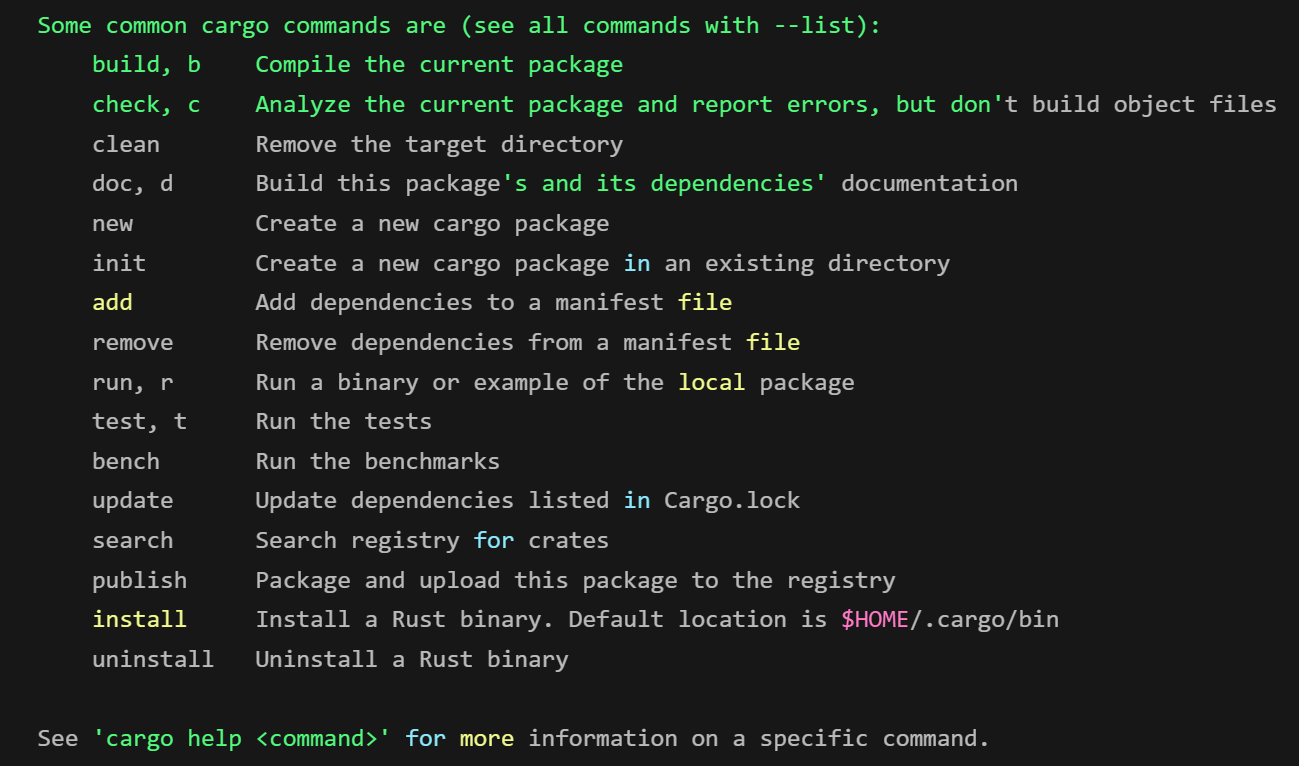




这个被叫做一个manifest， 它包含了Cargo编译这个包所需的所有元数据，manifest相关的格式及元素组成说明：每个包的文件 称为其清单。它写的采用TOML格式。它包含编译包所需的元数据。 [舱单格式 - 货物手册 (rust-lang.org)](https://doc.rust-lang.org/cargo/reference/manifest.html)



## 1.3 使用cargo



cargo --help 命令查看相关用法

1、使用 cargo new exercise 为代码新建一个 exercise/ 目录，

2、使用 cargo run 构建并运行二进制文件、

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

4、可以通过修改 Cargo.toml 为项目添加依赖项。当运行 cargo 命令时，系统会自动下载和编译缺失的依赖项。

cargo官方文档：[简介 - The Cargo Book (rust-lang.org)](https://doc.rust-lang.org/stable/cargo/)

# 二、rust基础学习

## 2.1 rust介绍

Rust 是一门静态编译语言，其功能定位与 C++ 相似

rustc 使用 LLVM 作为它的后端。

Rust 支持多种平台和架构:

x86、ARM、WebAssembly……

Linux、Mac、Windows……

Rust 被广泛用于各种设备中：

固件和引导程序，

智能显示器，

手机，

桌面，

服务器。

Rust 的优势

编译时内存安全：在编译时可防止所有类内存 bug

不存在未初始化的变量。

不存在“双重释放”。

不存在“释放后使用”。

不存在 NULL 指针。

不存在被遗忘的互斥锁。

不存在线程之间的数据竞争。

不存在迭代器失效。

没有未定义的运行时行为：每个 Rust 语句的行为都有明确定义

数组访问有边界检查。

整数溢出有明确定义（panic 或回绕）。

现代语言功能：具有与高级语言一样丰富且人性化的表达能力

枚举和模式匹配。

泛型。

无额外开销的外部函数接口（FFI）。

零成本抽象。

强大的编译器错误提示。

内置依赖管理器。

对测试的内置支持。

优秀的语言服务协议（Language Server Protocol）支持。

Rust 基本语法：变量、标量（scalar）和复合（compound）类型、枚举（Enum）、结构体（struct）、引用、函数和方法。

Types and type inference.

控制流结构：循环、条件语句等。

用户定义的类型：结构体和枚举。

模式匹配：解构枚举、结构体和数组（array）。

基本语法相关：

函数以 fn 开头。

像C和C++一样，块由花括号分隔。

main 函数是程序的入口点。

Rust有卫生宏 (hygienic macros)，println! 就是一个例子。

Rust 字符串是 UTF-8 编码的，可以包含任何 Unicode 字符。

注释符号： //（单行注释），/\* 123213 \*/（多行或代码块注释）

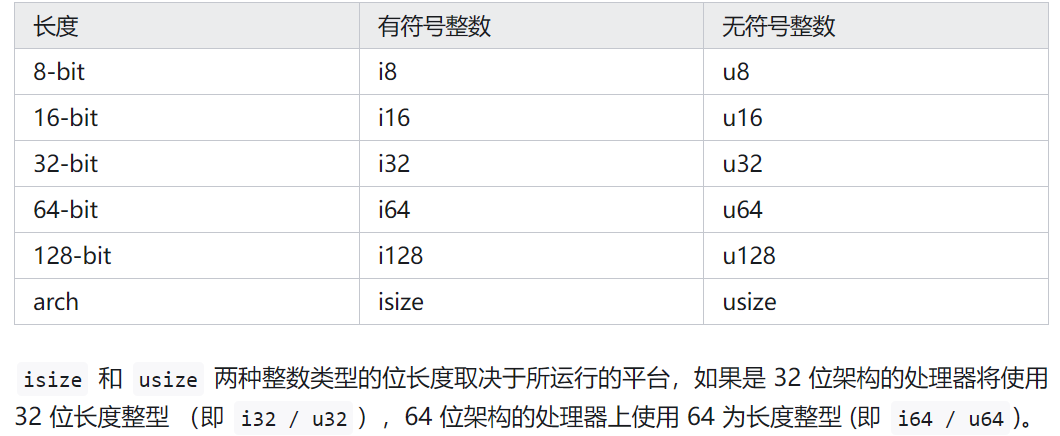
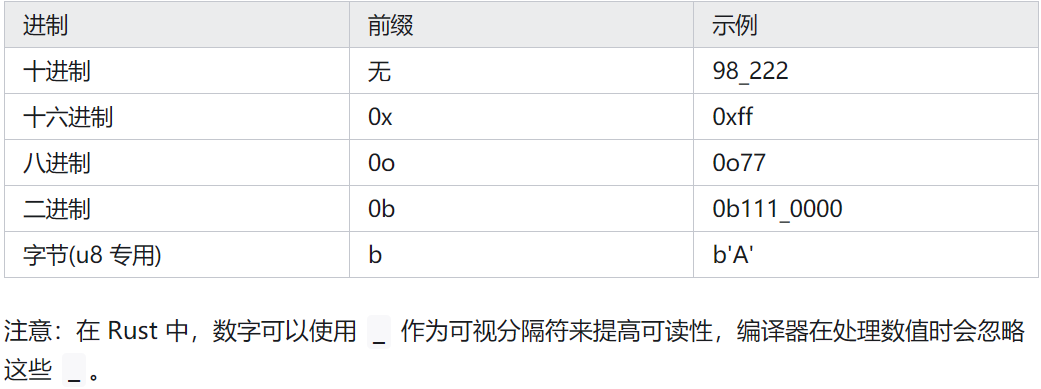
## 2.2 数据类型

标量类型（Scalar Types）：整型、浮点型、布尔类型和字符类型。

### 2.2.1 整型：

**let** x: **i32** **=** **-**123; *// 有符号整数*

**let** y: **u32** **=** 456; *// 无符号整数*

### 2.2.2 浮点型：

**let** x: **f32** **=** 3.14; *// f32 单精度浮点数*

**let** y: **f64** **=** 2.71828; *// f64 双精度浮点数*

**let** z **=** 2.71828; *// 默认是 f64 双精度浮点数*

### 2.2.3 布尔类型：

**let** t: **bool** **=** **true**;

**let** f: **bool** **=** **false**;

**let** and\_result **=** a **&&** b; *// 返回 false*

**let** or\_result **=** a **||** b; *// 返回 true*

**let** not\_result **=** **!**a; *// 返回 false*

### 2.2.4 字符、字符串类型:

字符类型（char）用于表示单个 Unicode 标量值，这意味着它可以表示比 ASCII 更广泛的字符集。在 Rust 中，字符（char）是通过单引号（'）来表示的，而字符串（String 或 &str）是通过双引号（"）来表示的。

**let** c: **char** **=** 'z';

**let** z: **char** **=** 'ℤ'; *// Unicode值*

**let** heart\_eyed\_cat **=** ' ';

特殊字符：可以使用转义序列表示，如换行符 '\n'，制表符 '\t'，单引号 '\''，反斜杠 '\\' 等。

Unicode 字符：使用 \u{} 转义和大括号内的十六进制数值来表示，如 '好' 可以用 '\u{597D}' 表示。

字符串：

**String** - a modifiable, owned string.

&str 为只读字符串。字符串字面量属于此类型。

String 是一种用户定义的类型，具有构造函数 (::new()) 和类似 s.push\_str(..) 的方法。

&str 中的 & 表示这是一个引用。我们稍后将介绍引用，现在只需将 &str 视为一个单位，表示 “只读字符”。

let greeting: &str = "Greetings";

let planet: &str = "🪐";

let mut sentence = String::new();

sentence.push\_str(greeting);

sentence.push\_str(", ");

sentence.push\_str(planet);

println!("final sentence: {}", sentence);

println!("{:?}", &sentence[0..5]);

### 2.2.5 复合类型(compound Types)

**元组**（Tuple）：

元组是可以包含多个不同类型值的一种集合。元组的长度固定，一旦声明，它的长度不会改变。

**let** tup: (**i32**, **f64**, char) **=** (500, 6.4, 'y');

tup 是一个元组，包含了一个 i32 类型的整数、一个 f64 类型的浮点数，以及一个 char 类型的字符。

元组解构：

**let** tup **=** (500, 6.4, 1);

**let** (x, y, z) **=** tup;

println**!**("The value of y is: {}", y); *// 输出 6.4*

用索引访问元组中的元素。

**let** five\_hundred **=** tup.0;

**let** six\_point\_four **=** tup.1;

**let** y **=** tup.2;

**数组**（Array）

数组是有多个**相同类型**值组成的集合。与元组一样，数组的长度也是固定的，

**let** a **=** [1, 2, 3, 4, 5];

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

**let** a **=** [3; 5]; *// 等同于 let a = [3, 3, 3, 3, 3];*

**let** first **=** a[0]; *// 访问第一个元素*

**let** second **=** a[1]; *// 访问第二个元素*

### 2.2.6 自定义类型（Custom Types）:

结构体(struct)：

**struct** **User** {

username: String,

email: String,

sign\_in\_count: **u64**,

active: **bool**,

}

创建结构体

let user1 **=** User {

email: String::from("admin@example.com"),

username: String::from("ziyang"),

active: **true**,

sign\_in\_count: 1,

};

可以通过派生特性 #[derive(Debug)] 来允许结构体实例在使用 println! 宏时使用 {:?} 或 {:#?} 格式化输出。

### 2.2.7 枚举（Enum）:

**enum** **IpAddrKind** {

V4,

V6,

}

枚举也可以关联数据。

**enum** **IpAddr** {

V4(String),

V6(String),

}

每个枚举变体关联的数据都可以有不同类型。

**enum** **Message** {

Quit,

Move { x: **i32**, y: **i32** },

Write(String),

ChangeColor(**i32**, **i32**, **i32**),

}

**使用方法：**

**let** four **=** IpAddrKind::V4;

**let** six **=** IpAddrKind::V6;

### 2.2.8 函数返回值特殊类型:Never “!”

! 类型主要与 Rust 中的 match 表达式一起用于保证所有可能情况都已处理。如果 match 的一个分支结束于一个永不返回的函数，Rust 知道不需要返回值。这就是所谓的"穷尽性检查"

错误处理:

**fn** **forever**() -> **!** {

**loop** {

*// 无限循环，永不返回*

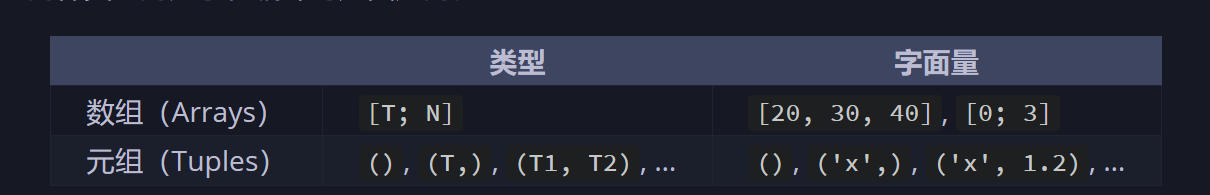
}

}

forever 函数有一个 ! 返回类型，表示此函数将永远不会返回一个值。

经常与 panic! 宏一起使用，它会使当前线程崩溃，并可以带有一个错误消息。由于 panic! 永远不会返回，它的返回类型是 !。

### 2.2.9 元组和数组



数组：

fn main() {

let mut a: [i8; 10] = [42; 10];

a[5] = 0;

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

}

添加 #，比如 {a:#?}，会输出“美观打印（pretty printing）” 格式，这种格式可能会更加易读。

元组：

fn main() {

let t: (i8, bool) = (7, true);

println!("t.0: {}", t.0);

println!("t.1: {}", t.1);

}

和数组一样，元组也具有固定的长度。

元组将不同类型的值组成一个复合类型。

元组中的字段可以通过英文句号加上值的下标进行访问比如：t.0, t.1。

for 语句支持对数组进行迭代（但不支持元组）。

fn main() {

let primes = [2, 3, 5, 7, 11, 13, 17, 19];

for prime in primes {

for i in 2..prime {

assert\_ne!(prime % i, 0);

}

}

}

模式匹配：

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

#[rustfmt::skip]

fn main() {

let input = 'x';

match input {

'q' => println!("Quitting"),

'a' | 's' | 'w' | 'd' => println!("Moving around"),

'0'..='9' => println!("Number input"),

key if key.is\_lowercase() => println!("Lowercase: {key}"),

\_ => println!("Something else"),

}

}

解构：

解构是一种从数据结构中提取数据的方法，具体方法是编写与数据结构匹配的模式，将变量绑定到数据结构的子组件。

元组（Tuples）

fn main() {

describe\_point((1, 0));

}

fn describe\_point(point: (i32, i32)) {

match point {

(0, \_) => println!("on Y axis"),

(\_, 0) => println!("on X axis"),

(x, \_) if x < 0 => println!("left of Y axis"),

(\_, y) if y < 0 => println!("below X axis"),

\_ => println!("first quadrant"),

}

}

数组（Arrays）

#[rustfmt::skip]

fn main() {

let triple = [0, -2, 3];

println!("Tell me about {triple:?}");

match triple {

[0, y, z] => println!("First is 0, y = {y}, and z = {z}"),

[1, ..] => println!("First is 1 and the rest were ignored"),

\_ => println!("All elements were ignored"),

}

}

指出 .. 是如何扩展以适应不同数量的元素的。

展示使用模式 [.., b] 和 [a@..,b] 来匹配切片的尾部。

### 2.2.10 引用

#### 1、共享引用

引用提供了一种可以访问另一个值但无需对该值负责的方式，也被称为 “借用”。共享引用处于只读状态，且引用的数据无法更改。

fn main() {

let a = 'A';

let b = 'B';

let mut r: &char = &a;

println!("r: {}", \*r);

r = &b;

println!("r: {}", \*r);

}

对类型 T 的共享引用表示为 &T。可以使用 & 运算符创建引用值。\* 运算符会 “解引用”某个引用，并得到该引用值。

Rust 会静态禁止悬垂引用：

fn x\_axis(x: i32) -> &(i32, i32) {

let point = (x, 0);

return &point;//报错

}

#### 2、独占引用

独占引用（也称为可变引用）允许更改其所引用的值。它们的类型为 &mut T。

fn main() {

let mut point = (1, 2);

let x\_coord = &mut point.0;

\*x\_coord = 20;

println!("point: {point:?}");

}

结果：point: (20, 2)

“独占模式”表示只有此引用可用于访问该值。在独占引用存在期间，不允许同时存在其他引用（无论是共享引用还是独占引用），并且无法访问引用的值。

### 2.2.11 类型别名（重定义类型）

类似于C语言中的typedef。

enum CarryableConcreteItem {

Left,

Right,

}

type Item = CarryableConcreteItem;

// Aliases are more useful with long, complex types:

use std::cell::RefCell;

use std::sync::{Arc, RwLock};

type PlayerInventory = RwLock<Vec<Arc<RefCell<Item>>>>;

## 2.3 调试打印输出println!()

rust的打印输出是通过println!()宏来实现的，不是函数是一个宏定义。

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

println!("helloworld") //会自动换行

print!("helloworld\n"); //需要自行加入换行符"\n"

“{ }” 是一个占位符，所以我们传 递 guess 作为一个参数。如果我们有多个{} ，我们应该传递多个参数。

也可以这么用：

let print\_test = 99;

println!("print\_test = {print\_test}");

fn main() {

println!("{}", 1); // 默认用法,打印Display

println!("{:o}", 9); // 八进制

println!("{:x}", 255); // 十六进制 小写

println!("{:X}", 255); // 十六进制 大写

println!("{:p}", &0); // 指针

println!("{:b}", 15); // 二进制

println!("{:e}", 10000f32); // 科学计数(小写)

println!("{:E}", 10000f32); // 科学计数(大写)

println!("{:?}", "test"); // 打印Debug

println!("{:#?}", ("test1", "test2")); // 带换行和缩进的Debug打印

println!("{a} {b} {b}", a = "x", b = "y"); // 命名参数

}

## 2.4 变量的说明

### 2.4.1 变量的可变性

不可变变量 (Immutable)

可变变量 (Mutable)

let x: i32 = 10;//rust默认定义的变量是可读不可写的。

let mut x:i32 = 10;//增加木头关键字后变量x可读可写。

### 2.4.2 变量遮蔽（Shadowing）

在相同作用域中，使用相同的名称声明一个新变量。这个新声明的变量“遮蔽”了先前声明的同名变量。当新变量被引入时，它会使得与之同名的先前变量不再可见，直到新变量离开作用域前，无法访问被遮蔽的变量。

遮蔽的好处是可以重新使用更有意义或适用于当前上下文的变量名，遮蔽通常用于转换值的类型或给变量一个新值，而这样做并不影响原来变量的不可变性。

**let** x **=** 5;

**let** x **=** x **+** 1; *// 遮蔽前一个 x 变量*

**let** x **=** "现在 x 是一个字符串"; *// 再次遮蔽，改变了值的类型*

**fn** **main**() {

**let** x **=** 5;

**let** x **=** x **+** 1;

{

*// 在使用花括号创建的内部作用域内，变量 `x` 被遮蔽*

**let** x **=** x **\*** 2;

println**!**("x 在当前作用域的值是: {x}"); *// 12*

}

*// 当作用域结束后，内部 shadowing 的作用域也结束了，`x` 又返回到 `6`*

println**!**("x 在当前作用域的值是: {x}"); *// 6*

}

### 2.4.3 常量（Constants）

常量使用 const 关键字声明，并且它们总是不可变的。

常量必须在声明时就赋值，而且它们的类型必须显式声明。

常量名应该全部大写，否则会报警告。

**const** MAX\_POINT: **u32** **=** 100\_000;

**const** CONST\_VAL **=** 1000; *// error: missing type for `const` item*

常量在编译时被赋值，且必须是一个恒定的表达式，不能是运行时的结果。

**const** CONST\_VAL: **i32** **=** get\_val(); // error

**const** CONST\_VAL: **i32**;

CONST\_VAL **=** 1000; // error

常量无法进行变量遮蔽，相同常量名重复定义时会报错。

const CONST\_VAL: i32 = 1000;

const CONST\_VAL: &str = "str";// error

const 关键字可以用于全局作用域中声明常量，也可以在局部作用域中使用。

## 2.5函数

### 2.5.1 函数的格式

main 函数是项目的入口函数，每个可执行的 Rust 程序都必须包含一个 main 函数。使用fn关键字定义，后跟函数名称和一对圆括号。在圆括号中，可以定义零个或多个参数，每个参数后面跟着一个冒号(:)和它的类型。大括号 {} 包围着函数的主体内容。如果函数需要返回值，必须声明返回类型，通过在参数列表后添加一个箭头(->)和返回类型来完成。

fn <方法名>(<参数名1>: <类型>, <参数名2>: <类型2>) -> <返回值类型> {

// 函数体，如果有返回值，则使用return关键字，或者省略它，

// 最后的表达式作为返回值（无分号）

}

**fn** **print\_sum**(a: **i32**, b: **i32**) {

println**!**("和是: {}", a **+** b);

}

**fn** **main**() {

print\_sum(5, 6);

}

### 2.5.2 函数的返回值

返回值的类型声明紧跟在参数列表之后，并以 -> 开头，返回值通过在函数的最后一个表达式上省略分号来隐式返回或者通过关键字“return”返回。

最后一个表达式没有分号才作为返回值，如果有分号将作为一个表达式。

**fn** **add**(a: **i32**, b: **i32**) -> **i32** {

a **+** b *// 注意这里没有分号，表示这是返回的值*

}

**fn** **add**(a: **i32**, b: **i32**) -> **i32** {

**if** a **==** 0 **||** b **==** 0 {

**return** 0 *// 通过 return 提前返回*

}

a **+** b //如果执行到这里则是隐式返回

}

### 2.5.3 函数的私有性

在 Rust 中，**默认**函数是私有的，只能在定义它们的模块内部访问。只有 pub 关键字声明是公共的，当函数使用pub关键字后，才能使函数在其他模块中可见。

*// 默认是私有的，只能在模块内访问*

**fn** **add**(a: **i32**, b: **i32**) -> **i32** {

a **+** b

}

*// 使用 pub 关键字声明公共的*

**pub** **fn** **add\_two\_num**(a: **i32**, b: **i32**) -> **i32** {

a **+** b

}

### 2.5.4 函数的可变参数操作

Rust 不支持直接的可变参数函数(Variadic Functions)，但可以通过接受一个切片或者元组来间接实现。

数组接收可变参数示例

**fn** **sum**(slice: **&**[**i32**]) -> **i32** {

slice.iter().fold(0, **|**acc, **&**x**|** acc **+** x)

}

**fn** **main**() {

**let** numbers **=** vec**!**[1, 2, 3, 4, 5];

**let** result **=** sum(**&**numbers);

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

}

在这个例子中，sum函数接受一个i32类型的切片作为参数，并使用迭代器的fold方法来计算总和。然后就可以通过一个向量调用这个函数，在这种情况下是numbers向量。将向量的借用&numbers传递给sum函数时，它将被自动借用为一个切片，这样函数就可以接受不同数量的参数。

元组接收可变参数示例

**fn** **sum\_tuple**(args: (**i32**, **i32**, **i32**)) -> **i32** {

**let** (a, b, c) **=** args;

a **+** b **+** c

}

**fn** **main**() {

**let** result **=** sum\_tuple((1, 2, 3));

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

}

在这个例子中，sum\_tuple函数接受一个包含三个i32元素的元组。元组在参数列表中是固定大小的，但是可以通过创建不同大小的元组类型来模拟可变参数。但这种方法不如使用切片那样灵活，如果需要不同数量的参数，通常推荐使用切片。

### 2.5.5 函数的模式匹配

以在函数定义中直接解构复合类型。例如，倘若你传递一个元组或结构体给函数，可以直接在参数列表中解构它们，从而得到它们的部分或全部内容。这种特性在处理具有多个部分的数据类型时非常有用，我们无须先创建一个变量，然后再使用模式匹配来解构它，因为我们可以直接在函数的参数中完成这个步骤。

**fn** **print\_coordinates**(**&**(x, y): **&**(**i32**, **i32**)) {

println**!**("Current location: ({}, {})", x, y);

}

**fn** **main**() {

**let** point **=** (3, 5);

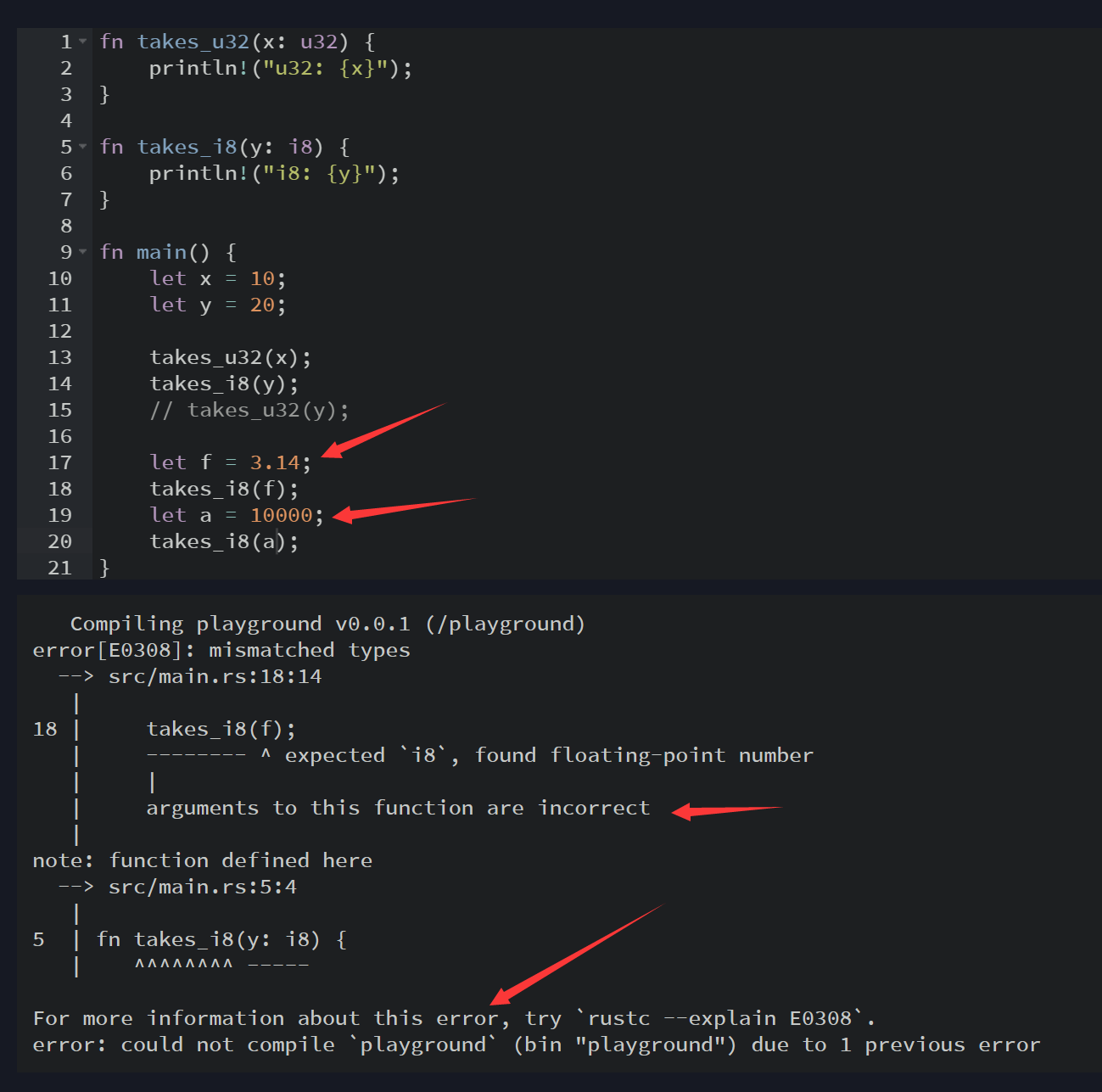
print\_coordinates(**&**point);

}

在这个例子中，print\_coordinates 函数接受一个引用到一个元组 (i32, i32) 的参数。参数 &(x, y) 是一个模式，它解构传入的元组，并允许我们在函数体内直接使用 x 和 y。当我们调用 print\_coordinates(&point) 时，point 元组被解构，元组中的值被分别绑定到 x 和 y。

2.5.6 函数传参的类型推导

Rust 会根据变量的使用来确定其类型：



## 2.6 流控制

条件控制、循环控制、break 和 continue、代码块和作用域、函数、宏。

### 2.6.1 条件控制

Rust 的许多语法与 C相似，行内注释以 // 起始，块注释使用 /\* ... \*/ 来界定。if 和 while 等关键词作用与以上语言一致。变量赋值使用 =，值之间比较使用 ==。

if表达式：

fn main() {

let x = 10;

if x < 20 {

println!("small");

} else if x < 100 {

println!("biggish");

} else {

println!("huge");

}

}

fn main() {

let x = 10;

let size = if x < 20 { "small" } else { "large" };

println!("number size: {}", size);

}

### 2.6.2 循环控制

#### 2.6.2.1 loop

loop 关键字，在Rust中，使用loop表示一个无限死循环。其中，我们可以使用continue和break控制执行流程。continue；语句表示本次循环内，后面的语句不再执行，直接进入下一轮循环。break；语句表示跳出循环，不再继续。示例如下：

//loop死循环

//continue；语句表示本次循环内，后面的语句不再执行，直接进入下一轮循环。

//break: 表示跳出循环不再继续。

fn loop\_test1(){

println!("\*\*\*\*\* loop\_test1 start \*\*\*\*\*");

let mut count = 0u32;

loop{

count +=1;

if count == 5 {

println!("###count=={},continue!",count);

continue;

}

println!("print count={}",count);

sleep(Duration::from\_millis(100)); //睡眠 100 ms

if count>10{

break; //如果没有break则一直死循环。

}

}

println!("\*\*\*\*\* loop\_test1 end \*\*\*\*\*\n");

}

break语句和continue语句还可以在多重循环中选择跳出到哪一层的循环。

//loop循环 + 跳出标签所指的那层循环。

fn loop\_test2()

{

println!("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* loop\_test2 start \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

let mut count1 = 0u32;

let mut count2 = 0u32;

'loop1:loop{

count1 +=1;

'loop2:loop{

count2+=1;

if count2 == 2 {

println!("###count2=={},continue 'loop2!",count2);

continue 'loop2;

}

if count2 >= 4 {

count2 = 0;

break ;

}

if count1 == 5{

println!("###count1=={},break: out of 'loop1!",count1);

break 'loop1;

}

println!("print count1={} -- count2={}",count1,count2);

}

}

println!("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* loop\_test2 end \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

}

loop结构也可以作为表达式的一部分。在loop内部break的后面可以跟一个表达式，这个表达式就是最终的loop表达式的值。如果一个loop永远不返回，那么它的类型就是“发散类型”。示例如下：

//loop结构作为表达式的一部分

fn loop\_test3(){

println!("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* loop\_test3 start \*\*\*\*\*");

let mut count = 0u32;

println!("print count={}",count);

count = loop{

break 99;

};

println!("print count={}",count);

#[warn(unreachable\_code)]

{

//count = loop{};//代码运行到这里会进入死循环。后面代码不再执行。

//println!("print count={}",count);

}

println!("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* loop\_test3 end \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

}

#### 2.6.2.2 while

while 关键字，while语句是带条件判断的循环语句。其语法是while关键字后跟条件判断语句，最后是结果语句块。如果条件满足，则持续循环执行结果语句块。示例如下：

//while语句是带条件判断的循环语句,同样可有continue和break来控制循环流程。。

fn while\_test1(){

println!("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* while\_test1 start \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

let mut while\_count = 0;

while while\_count < 10 {

while\_count+=1;

println!("print while\_count={}",while\_count);

if while\_count == 3 {

println!("###continue: while while\_count= {}",while\_count);

continue;

}

if while\_count == 5 {

println!("###break: out of while while\_count = {}",while\_count);

break;

}

}

println!("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* while\_test1 end \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

}

while实现死/无限循环：

fn while\_test2(){

println!("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* while\_test2 start \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

let mut while\_count = 0;

#[warn(while\_true)]{

while true {

while\_count+=1;

println!("print while\_count={}",while\_count);

sleep(Duration::from\_millis(500)); //睡眠 500 ms

// if while\_count == 10 {

// break; //如果没有break则一直循环。

// }

}

}

println!("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* while\_test2 end \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

}会警告提示用关键字“loop”：

warning: denote infinite loops with `loop { ... }`

--> src/main.rs:108:9

|

108 | while true {

| ^^^^^^^^^^ help: use `loop`

**注意**：loop和while true的死/无限循环区别：

//loop和while true无限循环区别。

fn loop\_while\_infinitecycle\_test3(){

println!("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* loop\_while\_infinitecycle\_test3 start \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

let x;

**loop** { x = 1; break; }

println!("{}", x);//没问题。

let x;

while true { x = 1; break; }

println!("{}", x);//报错，提示x不确定被赋初值，却被使用。

println!("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* loop\_while\_infinitecycle\_test3 end \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

}

原因在于对编译器来说loop是一定会进入循环体去执行，但是while是需要先判断条件为真才进入循环体执行。

#### 2.6.2.2 for

rust中的for 关键字类似其他语言中的for-each循环，但不是C/C++的三段式for循环语句。for循环的主要用处是利用迭代器对包含同样类型的多个元素的容器执行遍历，如数组、链表、HashMap、HashSet等。在Rust中，我们可以轻松地定制自己的容器和迭代器，因此也很容易使for循环也支持自定义类型。for循环内部也可以使用continue和break控制执行流程。举例如下：

//for 循环。continue和break控制执行流程

fn for\_test1(){

println!("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* for\_test1 start \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

let mut for\_count: i32 ;

for n in 1..10{

for\_count = n;

if for\_count == 3 {

println!("###continue: for for\_count= {}",for\_count);

continue;

}

if for\_count == 6 {

println!("###break: out of for for\_count = {}",for\_count);

break;

}

println!("print for\_count={}",for\_count);

}

println!("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* for\_test1 end \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

}

//for 循环。for + match 的用法

fn for\_test2(){

println!("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* for\_test2 start \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

/\* 数值匹配 \*/

for n in 0..=20{

if n == 16 {

println!("###continue: for n= {}",n);

continue;

}

match n {

0 => println!("match 0 n={}",n),

0|2|4|6|8 => println!("match n is even number:{}",n),

1|3|5|7|9 => println!("match n is odd number:{}",n),

10..=15 => println!("match n is '10'<=n<='19' : {}",n),

\_ => println!("useless number {}",n),

}

}

/\* 字符串匹配 \*/

let for\_name = vec!["s1","s2","s3"];

for name in for\_name.iter(){

match name{

&"s1" => println!("match \"s1\" name ={}",name),

&"s2" => println!("match \"s2\" name ={}",name),

\_ => println!("current name={}, isn't = \"s1\" or \"s2\"!",name),

}

}

println!("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* for\_test2 end \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

}

//for + while 循环混用。

fn for\_test3(){

println!("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* for\_test3 start \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

'outer: for x in 1..5 {

let mut i = 0;

while i < x {

i += 1;

if i == 3 {

println!("###break: 'outer. for: x={}, while:i={}",x,i);

break 'outer;

}

println!("for: x={}, while:i={}",x,i);

}

}

println!("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* for\_test3 end \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

}

//for+iter(), iter()函数获取数组中所有元素的值。

fn my\_fn()->u32{

let arr:[i32;4]=[10,20,30,40];

println!("array is {:?}",arr);

println!("array size is :{}",arr.len());

for val in arr.iter(){

println!("value is :{}",val);

}

return 0;

}

//for 引用传递

fn my\_fn(arr:&mut [i32;3])

{

for i in 0..3 {

arr[i]=0;

}

}

fn main() {

let mut arr:[i32;3]=[1,2,3];

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

**my\_fn**(&mut arr);

println!("arr after uptade: {:?}", arr);

}

注意：

for **i** in 0...10{}中的i是直接使用，是不用单独声明的，例如：

let mut i:u32 = 0;//警告未使用的变量：warning: unused variable: `i`

for i in 1..10{ //这个for循环的i不是上面声明的变量i.

println!("i={}",i);

}

#### 2.6.2.3 基于rust循环控制实现的相关算法

##### 1. 计算考拉兹序列的长度。

（若n为偶数）: n→n/2.（若n为奇数): n→n\*3+1.

从13开始，可以选代生成如下的序列：13→40→20→10→5→16→8→4→2→1，代码示例如下：

fn collatz\_length(mut n: i32) -> u32 {

let mut len = 1;

while n > 1 {

println!("len={}:{}",len,n);

n = if n % 2 == 0 { n / 2 } else { 3 \* n + 1 };

len += 1;

}

println!("len={}:{}",len,n);

return len;

}

##### 2. 求斐波那契数列

斐波那契函数的变化规律：F1=1,F2=1,Fn=Fn-1+Fn-2(n>=3)。

fn fib(mut n: u32) -> u32 {

let mut fn1:u32 = 1;

let mut fn2:u32 = 1;

let mut fn\_value:u32 = 0;

while n > 2{

println!("fn\_value={},fn1={},fn2={}",fn\_value,fn1,fn2);

/\* fn\_value = fn-1 + fn-2 \*/

fn\_value=fn1+fn2;

fn2=fn1;

fn1=fn\_value;

n-=1;

}

println!("fn1={},fn2={}",fn1,fn2);

return fn\_value;

}

##### 3. 猴子吃桃问题

一只猴子第一天摘了n个桃，当即吃了一半又多吃了一个，第二天又吃了一半再多吃一个，直到第十天早上想吃时只剩下1个，问第一天摘了多少桃？

fn peach() -> u32 {

let mut peach\_num:u32 = 1;//in day 10(no eat), only remained 1 peach.

for i in 1..10{

println!("day={},peach\_num={}",i,peach\_num);

peach\_num=2\*(peach\_num+1);

}

println!("day={},peach\_num={}",10,peach\_num);

return peach\_num;

}

##### 4. 实现冒泡排序

#[allow(unused\_variables)]// 使用#[allow(unused\_variables)]属性来禁用特定函数中的未使用变量警告。

fn my\_fn( mem:&mut Vec<i32>)

{

for i in 0..mem.len() {

for x in 0..mem.len()-1{

if mem[x] > mem[x+1]{

mem.swap(x,x+1);

}

}

}

}

fn main() {

let mut mem1 =vec![1,6,5,3,9,4,8,2,7,0];

println!("mem1: {:?}", mem1);

my\_fn(&mut mem1);

println!("mem1 after update: {:?}", mem1);

}

### 2.6.3 代码块和作用域

#### 2.6.3.1 代码块

rust的块用花括号“{ }”包含起来，最后一行没有分号；的语句是这个块的返回值。

fn main() {

let z = 13;

let x = {

let y = 10;

println!("y: {y}");

z - y

};

println!("x: {x}");

}

#### 2.6.3.2 作用域和遮蔽性（Shadowing）

fn main() {

let a = 10;

println!("before: {a}");

{

let a = "hello";

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

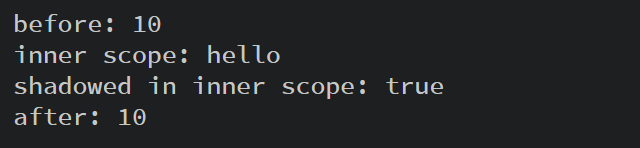
let a = true;

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

}

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

}



## 2.7 宏定义

在传统语言中，宏通常只是进行文本的替换，而在rust中可以可以做到更多，相应的也就更加复杂难学，Rust宏与其他语言宏的区别主要有以下几点：

编译时执行：Rust宏是在编译时执行的，而很多高级语言的宏通常是在运行时执行的。

强类型支持：Rust宏在执行时能够进行类型检查，因此可以保证生成的代码是类型安全的。

元编程能力：Rust宏可以进行元编程，即生成代码的代码，可以在编译时进行各种计算和类型检查，从而使生成的代码更加灵活和高效。

代码生成器：Rust宏可以作为代码生成器，为Rust程序员提供自定义的代码生成能力，以提高代码的复用性和性能。

模式匹配：Rust宏可以使用类似于模式匹配的语法，可以方便地处理各种复杂的代码结构。

经典的宏是println!宏用于向程序的标准输出打印一行字符串。

println!("Hello, world!");

注意宏的末尾有一个感叹号 !，这是 Rust 用于区分函数和宏的方法。宏在编译过程中会扩展为 Rust 代码，并且可以接受可变数量的参数。Rust 标准库包含各种有用的宏。

println!(format, ..) prints a line to standard output, applying formatting described in std::fmt.

format!(format, ..) 的用法与 println! 类似，但它以字符串形式返回结果。

dbg!(expression) 会记录表达式的值并返回该值。

todo!() 用于标记尚未实现的代码段。如果执行该代码段，则会触发 panic。

unreachable!() 用于标记无法访问的代码段。如果执行该代码段，则会触发 panic。

宏分类

声明式宏 macro\_rules!

派生宏 proc\_macro\_derive

属性宏 proc\_macro\_attribute

声明式宏：

macro\_rules**!** macro\_name {

// 匹配模式和模板替换

(pattern1) **=>** { */\* code1 \*/* };

(pattern2) **=>** { */\* code2 \*/* };

// 更多模式和模板替换

}

其中 macro\_name 是宏的名字， pattern是匹配的模式， code是需要替换的代码; 我们常用的 println! 、vec! 就都是声明式宏；

在 Rust 中，使用 **macro\_rules!** 关键字来定义声明式宏。

macro\_rules! my\_macro {

// 模式匹配和展开

($arg:expr) => {

// 生成的代码

// 使用 $arg 来代替匹配到的表达式

};

}

例子：

*// 宏的定义*  
macro\_rules! greet {

    // 模式匹配

    ($name:expr) => {

        // 宏的展开

        println!("Hello, {}!", $name);

    };

}

fn main() {

    // 调用宏

    greet!("World");

}

例子：

// 宏的定义

macro\_rules! vec {

// 基本情况，空的情况

() => {

Vec::new()

};

// 递归情况，带有元素的情况

($($element:expr),+ $(,)?) => {

{

let mut temp\_vec = Vec::new();

$(

temp\_vec.push($element);

)+

temp\_vec

}

};

}

fn main() {

// 调用宏

let my\_vec = vec![1, 2, 3];

println!("{:?}", my\_vec); // 输出: [1, 2, 3]

let empty\_vec = vec![];

println!("{:?}", empty\_vec); // 输出: []

}

例子：

#[macro\_export]

macro\_rules! hashmap {

($ ($key: expr => $val: expr),\* ) => {

{

let mut map = std::collections::HashMap::new();

$( map.insert($key, $val); )\*

map

}

};

}

​

/// 测试代码

#[test]

fn test\_hashmap() {

let map = hashmap!(1 => "one", 2=> "two", 3=> "three" );

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

}

#[macro\_export] 表示该宏导出可供其他包使用

macro\_rules! hashmap 表示该宏的名字是 hashmap

( $ ($key: expr => $val: expr),\* ) 这一段是匹配参数，属于最难理解的部分，$ ( xxx ),\* 格式表明对参数匹配时可以对xxx部分进行0次或若干次匹配(这里\*的作用类似于正则表示式的 , 当然也可以换成 ? "0或1次 或者 + "至少1次" ) ，xxx参数用逗号进行分隔且最后一次匹配不能有逗号；$key: expr => $val: expr 是参数的格式，expr代表匹配的类型，可以是任何有效的rust表达式；

$( map.insert($key, $val); )\* 进行代码的执行，执行次数与上面参数匹配的次数相同，$key $val就是每次匹配到参数的值

过程宏（Procedural Macros）

过程宏是一种更为灵活和强大的宏，允许在编译时通过自定义代码生成过程来操作抽象语法树（AST）。过程宏在功能上更接近于函数，但是它们在编写和使用上更加复杂。

过程宏的类型：

• 派生宏（Derive Macros）：用于自动实现trait（比如Copy、Debug）的宏。

• 属性宏（Attribute Macros）：用于在声明上附加额外的元数据，如#[derive(Debug)]。

过程宏的实现通常需要使用 proc\_macro 库提供的功能，例如 TokenStream 和 TokenTree，以便更直接地操纵源代码。

## 2.8 模式匹配

### 2.8.1 结构体

struct Foo {

x: (u32, u32),

y: u32,

}

#[rustfmt::skip]

fn main() {

let foo = Foo { x: (1, 2), y: 3 };

match foo {

Foo { x: (1, b), y } => println!("x.0 = 1, b = {b}, y = {y}"),

Foo { y: 2, x: i } => println!("y = 2, x = {i:?}"),

Foo { y, .. } => println!("y = {y}, other fields were ignored"),

}

}

结果：x.0 = 1, b = 2, y = 3

### 2.8.2 枚举

enum Result {

Ok(i32),

Err(String),

}

fn divide\_in\_two(n: i32) -> Result {

if n % 2 == 0 {

Result::Ok(n / 2)

} else {

Result::Err(format!("cannot divide {n} into two equal parts"))

}

}

fn main() {

let n = 100;

match divide\_in\_two(n) {

Result::Ok(half) => println!("{n} divided in two is {half}"),

Result::Err(msg) => println!("sorry, an error happened: {msg}"),

}

}

结果：100 divided in two is 50

### 2.8.3 模式匹配的条件控制

#### 1、if let 表达式

根据let值是否与表达式模式相匹配来执行不同的代码：

#![allow(unused)]

fn main() {

let dish = ("Ham", "Eggs");

// 此主体代码将被跳过，因为该模式被反驳

if let ("Bacon", b) = dish {

println!("Bacon is served with {}", b);

} else {

// 这个块将被执行。

println!("No bacon will be served");

}

// 此主体代码将被执行

if let ("Ham", b) = dish {

println!("Ham is served with {}", b);

}

if let \_ = 5 {

println!("不可反驳型的模式总是会匹配成功的");

}

}

结果：

No bacon will be served

Ham is served with Eggs

不可反驳型的模式总是会匹配成功的

if let表达式等价于match表达式，例如：

if let PATS = EXPR {

/\* body \*/

} else {

/\*else \*/

}

match EXPR {

PATS => { /\* body \*/ },

\_ => { /\* else \*/ }, // 如果没有 else块，这相当于 `()`

}

可以使用操作符 | 指定多个模式。 这与匹配(match)表达式中的 | 具有相同的语义：

#![allow(unused)]

fn main() {

enum E {

X(u8),

Y(u8),

Z(u8),

}

let v = E::Y(12);

if let E::X(n) | E::Y(n) = v {

assert\_eq!(n, 12);

}

}

// if let 表达式 else; 向 let 表达式 else；转换。

let a = 10;

if let 10 = a{

println!("a={a}");

}else{

println!("a!=10");

};

//变成：

let a = 10 else{

println!("a!=10");

return ; //用let ... else 需要在else后面返回。

};

println!("a={a}");

}

#### 2、while let 表达式

指出只要let值与 表达式 模式匹配，while let 循环就会一直进行下去。

struct Foo {

x: (u32, u32),

y: u32,

}

#[rustfmt::skip]

fn main() {

let foo = Foo { x: (2, 2), y: 3 };

while let 2 = foo.x.0 {

println!("x.1={},y={}",foo.x.1,foo.y);

std::thread::sleep(time::Duration::from\_millis(500));

}

}

## 2.9 方法和特征

### 2.9.1 方法

将函数与新类型相关联。您可以使用“impl”块来执行此操作：通过关键字impl来个结构体添加相关的处理方法。

#[derive(Debug)]

struct Race {

name: String,

laps: Vec<i32>,

}

impl Race {

// No receiver, a static method

fn new(name: &str) -> Self {

Self { name: String::from(name), laps: Vec::new() }

}

// Exclusive borrowed read-write access to self

fn add\_lap(&mut self, lap: i32) {

self.laps.push(lap);

}

// Shared and read-only borrowed access to self

fn print\_laps(&self) {

println!("Recorded {} laps for {}:", self.laps.len(), self.name);

for (idx, lap) in self.laps.iter().enumerate() {

println!("Lap {idx}: {lap} sec");

}

}

// Exclusive ownership of self

fn finish(self) {

let total: i32 = self.laps.iter().sum();

println!("Race {} is finished, total lap time: {}", self.name, total);

}

}

fn main() {

let mut race = Race::new("Monaco Grand Prix");

race.add\_lap(70);

race.add\_lap(68);

race.print\_laps();

race.add\_lap(71);

race.print\_laps();

race.finish();

// race.add\_lap(42);

}

结果：

Recorded 2 laps for Monaco Grand Prix:

Lap 0: 70 sec

Lap 1: 68 sec

Recorded 3 laps for Monaco Grand Prix:

Lap 0: 70 sec

Lap 1: 68 sec

Lap 2: 71 sec

Race Monaco Grand Prix is finished, total lap time: 209

### 2.9.1 特征

依据特征对类型进行抽象化处理。特征与接口类似

struct Dog {

name: String,

age: i8,

}

struct Cat {

lives: i8,

}

trait Pet {

fn talk(&self) -> String;

fn greet(&self) {

println!("Oh you're a cutie! What's your name? {}", self.talk());

}

}

impl Pet for Dog {

fn talk(&self) -> String {

format!("Woof, my name is {}!", self.name)

}

}

impl Pet for Cat {

fn talk(&self) -> String {

String::from("Miau!")

}

}

派生特征：

系统可以自动为您的自定义类型实现支持的 trait，如下所示：

fn main() {

let captain\_floof = Cat { lives: 9 };

let fido = Dog { name: String::from("Fido"), age: 5 };

captain\_floof.greet();

fido.greet();

}

#[derive(Debug, Clone, Default)]

struct Player {

name: String,

strength: u8,

hit\_points: u8,

}

fn main() {

let p1 = Player::default(); // Default trait adds `default` constructor.

let mut p2 = p1.clone(); // Clone trait adds `clone` method.

p2.name = String::from("EldurScrollz");

p2.strength = 100;

// Debug trait adds support for printing with `{:?}`.

println!("{:?} vs. {:?}", p1, p2);

}

## 2.10 泛型

### 2.10.1泛型函数

Rust 支持泛型，它允许您对使用或存储的类型进行算法或数据结构（例如排序或二叉树）的抽象化。

/// Pick `even` or `odd` depending on the value of `n`.

fn pick<T>(n: i32, even: T, odd: T) -> T {

if n % 2 == 0 {

even

} else {

odd

}

}

fn main() {

println!("picked a number: {:?}", pick(97, 222, 333));

println!("picked a tuple: {:?}", pick(28, ("dog", 1), ("cat", 2)));

}

结果：

picked a number: 333

picked a tuple: ("dog", 1)

Rust 会根据参数类型和返回值推理出 T 的类型。

这与 C++ 模板类似，但 Rust 会立即编译部分通用函数，因此该函数必须对所有符合约束条件的类型都有效。例如，请尝试修改 pick 函数，如果 n == 0，则返回 even + odd。即使仅使用带有整数的“pick”实例化，Rust 仍会将其视为无效。C++ 可让您做到这一点。

### 2.10.2 泛型类型

可以使用泛型对具体字段类型进行抽象化处理

#[derive(Debug)]

struct Point<T> {

x: T,

y: T,

}

impl<T> Point<T> {

fn coords(&self) -> (&T, &T) {

(&self.x, &self.y)

}

// fn set\_x(&mut self, x: T)

}

fn main() {

let integer = Point { x: 5, y: 10 };

let float = Point { x: 1.0, y: 4.0 };

println!("{integer:?} and {float:?}");

println!("coords: {:?}", integer.coords());

}

结果：

Point { x: 5, y: 10 } and Point { x: 1.0, y: 4.0 }

coords: (5, 10)

\*问：\*为什么 T 在 impl<T> Point<T> {} 中指定了两次？这不是多余的吗？

这是因为它是泛型类型的泛型实现部分。它们是独立的泛型内容。

这意味着这些方法是针对所有 T 定义的。

It is possible to write impl Point<u32> { .. }.

Point 依然是一个泛型，并且您可以使用 Point<f64>，但此块中的方法将仅适用于 Point<u32>。

### 2.10.3 特征边界和impl trait

使用泛型时，您通常会想要利用类型来实现某些特性， 这样才能调用此特征的方法。您可以使用 T: Trait 或 impl Trait 执行此操作：

fn duplicate<T: Clone>(a: T) -> (T, T) {

(a.clone(), a.clone())

}

// struct NotClonable;

fn main() {

let foo = String::from("foo");

let pair = duplicate(foo);

println!("{pair:?}");

}

结果：("foo", "foo")

与特征边界类似，impl Trait 语法可以在函数形参 和返回值中使用：

// Syntactic sugar for:

// fn add\_42\_millions<T: Into<i32>>(x: T) -> i32 {

fn add\_42\_millions(x: impl Into<i32>) -> i32 {

x.into() + 42\_000\_000

}

fn pair\_of(x: u32) -> impl std::fmt::Debug {

(x + 1, x - 1)

}

fn main() {

let many = add\_42\_millions(42\_i8);

println!("{many}");

let many\_more = add\_42\_millions(10\_000\_000);

println!("{many\_more}");

let debuggable = pair\_of(27);

println!("debuggable: {debuggable:?}");

}

结果：

42000042

52000000

debuggable: (28, 26)

对形参来说，impl Trait 就像是具有特征边界的匿名泛型形参。

对返回值类型来说，它则意味着返回值类型就是实现该特征的某具体类型， 无需为该类型命名。如果您不想在公共 API 中公开该具体类型，便可 使用此方法。

# 三、rust进阶学习

## 3.1 标准库

ust语言标准库是让Rust语言开发的软件具备可移植性的基础，它是一组最小的、经过实战检验的共享抽象，适用于更广泛的Rust生态系统。它提供了核心类型，如Vec和Option、类库定义的语言原语操作、标准宏、I/O和多线程等。Rust标准库，它有助于建立 Rust 库和程序使用的一组通用类型。R这样，两个库可以顺利地协同工作，因为它们都使用相同的String类型。所有Rust crate都可以使用std。因此，标准库可以在 use 语句中通过路径 std 访问标准库，如 std:::env。

事实上，Rust 包含标准库的多个层：core 、alloc 和 std。

core包括最基本的类型和函数，这些类型和函数不依赖于 libc、分配器，甚至不依赖于操作系统的存在。

alloc 包括需要全局堆分配器的类型，例如Vec 、Box 和 Arc。

嵌入式 Rust 应用通常只使用core ，偶尔会使用alloc 。

文档：

所有已发布的 Rust 库 crate 都会在 docs.rs 使用 rustdoc 工具自动记录。如需从项内（例如在模块内）为项编写文档，请使用：//! 或 /\*! .. \*/，这称为“内部文档注释”

//! This module contains functionality relating to divisibility of integers.

Option：

了解 Option<T> 的一些用法。它可以存储“T”类型的值，或者不存储任何值。例如，‘String::find’ 会返回 Option<usize>。

fn main() {

let name = "Löwe 老虎 Léopard Gepardi";

let mut position: Option<usize> = name.find('é');

println!("find returned {position:?}");

assert\_eq!(position.unwrap(), 14);

position = name.find('Z');

println!("find returned {position:?}");

assert\_eq!(position.expect("Character not found"), 0);

}

Option被广泛使用，而不仅仅是在标准库中。

unwrap 会返回 Option 或 panic 中的值。expect 方法与此类似，但其使用错误消息。

出现 None 时您或许会恐慌，但不能 “无意中”忘记检查是否为 None 的情况。

在草拟阶段的编程中，频繁使用 unwrap/expect 进行处理十分常见，但在正式版代码时，通常以更为妥当的方式处理 None 的情况。

小众优化意味着 Option<T> 在内存中的大小通常与 T 相同。

Result：

Result 与 Option 相似，但表示操作成功或失败，且每个操作的类型不同。这类似于表达式练习中定义的 Res，但是一个泛型：Result<T, E>，其中 T 用于 Ok 变体，E 出现在 Err 变体中。

use std::fs::File;

use std::io::Read;

fn main() {

let file: Result<File, std::io::Error> = File::open("diary.txt");

match file {

Ok(mut file) => {

let mut contents = String::new();

if let Ok(bytes) = file.read\_to\_string(&mut contents) {

println!("Dear diary: {contents} ({bytes} bytes)");

} else {

println!("Could not read file content");

}

}

Err(err) => {

println!("The diary could not be opened: {err}");

}

}

}

与 Option 方法相同，成功值位于 Result 方法内部， 开发者必须显示提取成功值。因此，建议进行错误检查。在绝不应出现错误的情况下， 可以调用 unwrap() 或 expect() 方法，这也是一种开发者意向信号。

Result文档是推荐阅读的。不是在课程中，但值得一提。它包含许多有助于函数式编程的便捷方法和函数。

Result 是用于实现错误处理的标准类型。

String：

String 是标准堆分配的可扩容 UTF-8 字符串缓冲区：

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

fn main() {

let mut s1 = String::new();

s1.push\_str("Hello");

println!("s1: len = {}, capacity = {}", s1.len(), s1.capacity());

let mut s2 = String::with\_capacity(s1.len() + 1);

s2.push\_str(&s1);

s2.push('!');

println!("s2: len = {}, capacity = {}", s2.len(), s2.capacity());

let s3 = String::from("🇨🇭");

println!("s3: len = {}, number of chars = {}", s3.len(), s3.chars().count());

}

结果:

s1: len = 5, capacity = 8

s2: len = 6, capacity = 6

s3: len = 8, number of chars = 2

“String::new”会返回一个新的空字符串，如果您知道自己想要推送到字符串的数据量，请使用“String::with\_capacity”。

“String::len”会返回“String”的大小（以字节为单位，可能不同于以字符为单位的长度）。

“String::chars”会针对实际字符返回一个迭代器。请注意，由于字素簇，“char”可能与人们所认为的“字符”有所不同。

当人们提到字符串时，可能是指“&str”或“String”。

当某个类型实现“Deref<Target = T>”时，编译器会让您以公开透明方式从“T”调用方法。

我们尚未讨论过 Deref trait，所以本部分目前主要介绍文档中边栏的结构。

“String”会实现“Deref<Target = str>”，后者可公开透明地授予其访问“str”方法的权限。

Write and compare let s3 = s1.deref(); and let s3 = &\*s1;.

“String”是作为字节矢量的封装容器实现的，矢量上支持的许多操作在“String”上也受支持，但有一些额外保证。

比较将“String”编入索引的不同方式：

使用“s3.chars().nth(i).unwrap()”转换为字符，其中“i”代表是否出界。

通过使用“s3[0..4]”转换为子字符串，其中该 Slice 在或不在字符边界上。

可以使用 to\_string 方法将许多类型转换为字符串。对于实现的所有类型，都会自动实现此特征，因此任何可以格式化的内容也可以转换为字符串。Display

Vec:

Vec 是标准的可调整大小堆分配缓冲区：

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

fn main() {

let mut v1 = Vec::new();

v1.push(42);

println!("v1: len = {}, capacity = {}", v1.len(), v1.capacity());

let mut v2 = Vec::with\_capacity(v1.len() + 1);

v2.extend(v1.iter());

v2.push(9999);

println!("v2: len = {}, capacity = {}", v2.len(), v2.capacity());

// Canonical macro to initialize a vector with elements.

let mut v3 = vec![0, 0, 1, 2, 3, 4];

// Retain only the even elements.

v3.retain(|x| x % 2 == 0);

println!("{v3:?}");

// Remove consecutive duplicates.

v3.dedup();

println!("{v3:?}");

}

结果：

v1: len = 1, capacity = 4

v2: len = 2, capacity = 2

[0, 0, 2, 4]

[0, 2, 4]

Vec是一种集合类型，以及 String和 HashMap。它包含的数据存储在堆中。这意味着在编译时不需要知道数据量。它可以在运行时增长或缩小。

请注意 Vec<T> 也是泛型类型，但不必显式指定T。与 Rust 类型推理一样，T在第一次push调用期间建立了 。

“vec！[…]”是用来代替“Vec：：new（）”的规范化宏，它支持向矢量添加初始元素。如需将矢量编入索引，您可以使用“[’ ‘]”方法，但如果超出边界，矢量将会 panic。 此外，使用“get”将返回“Option”。 “pop”函数会移除最后一个元素。现在只需要知道一个 Vec类型值也可以访问所有记录的切片方法。

HashMap：

标准的哈希映射，内含针对 HashDoS 攻击的保护措施：

use std::collections::HashMap;

fn main() {

let mut page\_counts = HashMap::new();

page\_counts.insert("Adventures of Huckleberry Finn".to\_string(), 207);

page\_counts.insert("Grimms' Fairy Tales".to\_string(), 751);

page\_counts.insert("Pride and Prejudice".to\_string(), 303);

if !page\_counts.contains\_key("Les Misérables") {

println!(

"We know about {} books, but not Les Misérables.",

page\_counts.len()

);

}

for book in ["Pride and Prejudice", "Alice's Adventure in Wonderland"] {

match page\_counts.get(book) {

Some(count) => println!("{book}: {count} pages"),

None => println!("{book} is unknown."),

}

}

// Use the .entry() method to insert a value if nothing is found.

for book in ["Pride and Prejudice", "Alice's Adventure in Wonderland"] {

let page\_count: &mut i32 = page\_counts.entry(book.to\_string()).or\_insert(0);

\*page\_count += 1;

}

println!("{page\_counts:#?}");

}

结果输出：

We know about 3 books, but not Les Misérables.

Pride and Prejudice: 303 pages

Alice's Adventure in Wonderland is unknown.

{

"Alice's Adventure in Wonderland": 1,

"Pride and Prejudice": 304,

"Adventures of Huckleberry Finn": 207,

"Grimms' Fairy Tales": 751,

}

与“vec!”不同，不存在标准的“hashmap!”宏。不过，从 Rust 1.56 开始，HashMap 实现了“From<[(K, V); N]>”，让我们能够轻松地从字面量数组初始化哈希映射，HashMap 也可以基于任何可生成键-值元组的“Iterator”进行构建。

use std::collections::HashMap;

use std::hash::Hash;

/// Counter counts the number of times each value of type T has been seen.

struct Counter<T: Eq + Hash> {

values: HashMap<T, u64>,

}

impl<T: Eq + Hash> Counter<T> {

/// Create a new Counter.

fn new() -> Self {

Counter { values: HashMap::new() }

}

/// Count an occurrence of the given value.

fn count(&mut self, value: T) {

\*self.values.entry(value).or\_default() += 1;

}

/// Return the number of times the given value has been seen.

fn times\_seen(&self, value: T) -> u64 {

self.values.get(&value).copied().unwrap\_or\_default()

}

}

fn main() {

let mut ctr = Counter::new();

ctr.count(13);

ctr.count(14);

ctr.count(16);

ctr.count(14);

ctr.count(14);

ctr.count(11);

for i in 10..20 {

println!("saw {} values equal to {}", ctr.times\_seen(i), i);

}

let mut strctr = Counter::new();

strctr.count("apple");

strctr.count("orange");

strctr.count("apple");

println!("got {} apples", strctr.times\_seen("apple"));

}

结果：

saw 0 values equal to 10

saw 1 values equal to 11

saw 0 values equal to 12

saw 1 values equal to 13

saw 3 values equal to 14

saw 0 values equal to 15

saw 1 values equal to 16

saw 0 values equal to 17

saw 0 values equal to 18

saw 0 values equal to 19

got 2 apples

## 3.2 标准库特征



## 2.x 迭代器（Iterator）

### 2.x.1 迭代器的概念

[Rust中的迭代器的使用：map转换、filter过滤、fold聚合、chain链接 - Pomelo\_刘金 - 博客园 (cnblogs.com)](https://www.cnblogs.com/liujin-now/p/17311921.html)

Rust中的迭代器是一种强大的工具，它提供了一种灵活、通用的方法来遍历序列。

fn main() {

let a = vec![0,1,2,3,4,5];

for n in a.iter(){ //遍历数组a，这个iter()功能是宏vec!提供的。

println!("n = {}",n);

}

}

### 2.x.2 迭代器的用法

[Rust Iterator.fold用法及代码示例 - 纯净天空 (vimsky.com)](https://vimsky.com/examples/usage/rust-core-iter-trait.Iterator.fold-rt.html)

# rust教程链接：

rust官网在线编译运行网址：[Rust Playground (rust-lang.org)](https://play.rust-lang.org/?version=stable&mode=debug&edition=2021)

[Rust 教程 | 菜鸟教程 (runoob.com)](https://www.runoob.com/rust/rust-tutorial.html)

谷歌的rust教程：[Rust 生态系统 - Comprehensive Rust 🦀 (google.github.io)](https://google.github.io/comprehensive-rust/zh-CN/cargo/rust-ecosystem.html)

看到元组数组：[元组和数组 - Comprehensive Rust 🦀 (google.github.io)](https://google.github.io/comprehensive-rust/zh-CN/tuples-and-arrays.html)

进度：17.1