腾讯课堂 零声教育 C/C++Linux服务器开发/高级架构师【零声教育】-学习视频教程-腾讯课堂 (qq.com)

0 官方文档

官方网站：https://rustwiki.org/
Rust 官方文档：https://rustwiki.org/zh-CN/
rust标准库：https://rustwiki.org/zh-CN/std/
Rust语言中文社区： https://rustcc.cn/

重点内容：
熟悉rust数据类型、函数调用、模块调用、结构组织。

1 环境搭建

1.1 Linux环境

1、安装配置

执行

```
$ export RUSTUP_DIST_SERVER=https://mirrors.ustc.edu.cn/rust-static
$ export RUSTUP_UPDATE_ROOT=https://mirrors.ustc.edu.cn/rust-static/rustup
$ sudo curl https://sh.rustup.rs -sSf | sh
```

显示
info: downloading installer

Welcome to Rust!

This will download and install the official compiler for the Rust
.....省略
profile: default
modify PATH variable: yes

1. **Proceed with installation (default)**
2. Customize installation
3. Cancel installation

选择

```
1
```

info: profile set to 'default'
info: default host triple is aarch64-unknown-linux-gnu
info: syncing channel updates for 'stable-aarch64-unknown-linux-gnu'
info: latest update on 2021-02-11, rust version 1.50.0 (cb75ad5db 2021-02-10)
.....省略

stable-x86_64-unknown-linux-gnu installed - rustc 1.60.0 (7737e0b5c 2022-04-04)

Rust is installed now. Great!

To get started you may need to restart your current shell.
This would reload your PATH environment variable to include
Cargo's bin directory ($HOME/.cargo/bin).

To configure your current shell, run:
source $HOME/.cargo/env 目的是使环境变量生效

执行

```
lqf@ubuntu:~$ source $HOME/.cargo/env
lqf@ubuntu:~$ cat .cargo/env
```

```
#!/bin/sh

rustup shell setup

affix colons on either side of $PATH to simplify matching

case ":![](https://g.yuque.com/gr/latex?
%7BPATH%7D%3A%22%20in%0A%20%20%20%20*%3A%22#card=math&code=%7BPATH%7D%3A%22%20in
%0A%20%20%20%20%2A%3A%22&id=bBopM)HOME/.cargo/bin":*)
;;
*)

# Prepending path in case a system-installed rustc needs to be overridden

export PATH="![](https://g.yuque.com/gr/latex?
HOME%2F.cargo%2Fbin%3A#card=math&code=HOME%2F.cargo%2Fbin%3A&id=lYKmB)PATH"
;;
esac
```

2、编写HelloWorld

vim main.rs

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

3、编译运行

lqf@ubuntu:~/rust/learn/helloworld$ rustc main.rs
lqf@ubuntu:~/rust/learn/helloworld$ ls
main main.rs
lqf@ubuntu:~/rust/learn/helloworld$ ./main
Hello, world!
lqf@ubuntu:~/rust/learn/helloworld$

4、添加环境变量

在vim /etc/profile添加环境变量：

> export PATH="$HOME/.cargo/bin:$PATH"

效果如下

5 、vscode 连接虚拟机

https://blog.csdn.net/qq_40300094/article/details/114639608

1.2 cargo使用

1.2.1 cargo常用命令

cargo --version	打印版本
cargo new hello_cargo	创建项目
cargo build	构建项目 （比如进入hello_cargo后构建）
cargo run	运行项目
cargo test	测试
cargo check	检测代码语法

1.2.2 标准的项目目录结构

一个典型的 Package 目录结构如下：

```
.
├── 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
```

这也是 Cargo 推荐的目录结构，解释如下：

- Cargo.toml 和 Cargo.lock 保存在 package 根目录下
- 源代码放在 src 目录下
- 默认的 lib 包根是 src/lib.rs
- 默认的二进制包根是 src/main.rs
- 其它二进制包根放在 src/bin/ 目录下
- 基准测试 benchmark 放在 benches 目录下
- 示例代码放在 examples 目录下
- 集成测试代码放在 tests 目录下

2 基本类型、变量、函数、模块

2.1 变量和变量

- 定义：

```
let a = 5; //不可变变量
    a = "abc"; //错误：类型不对
    a = 4.56;  //错误：精度不对
    a = 456;  //错误：不可变变量
let mut b = 5; //可变变量
    b = 6 //正确
const a = "BASE_INFO"; //常量
```

- 说明：Rust 是强类型语言，但具有自动判断变量类型的能力
- 定义：重影

```
 fn main() {
    let x = 5;
    let x = x + 1;
    let x = x * 2;
    println!("The value of x is: {}", x); //输出：The value of x is: 12
}
```

- 说明：重影与可变变量的赋值不是一个概念，重影是指用同一个名字重新代表另一个变量实体，其类型、可变属性和值都可以变化。但可变变量赋值仅能发生值的变化。

2.2 数据类型

2.2.1 整数型

- 说明

位长度	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	isize	usize

isize 和 usize 两种整数类型是用来衡量数据大小的，它们的位长度取决于所运行的目标平台，如果是 32 位架构的处理器将使用 32 位位长度整型。

- 实例

```
let mut _price = 64;
```

溢出测试：

```
fn main() {
    // let myage:u8 = 256;      // 溢出测试
    // println!("my age :{}", myage);
    let myage = 255;  // 将值改为255、256编译
    let myage2:u8 = myage;
    println!("my age :{}", myage2);
    println!("i8最大值:{},最小值是:{}",i8::max_value() ,i8::min_value());
}
```

2.2.2 浮点类型

- 说明Rust 与其它语言一样支持 32 位浮点数（f32）和 64 位浮点数（f64）。默认情况下，64.0 将表示 64 位浮点数，因为现代计算机处理器对两种浮点数计算的速度几乎相同，但 64 位浮点数精度更高。
- 实例

```
let mut _price = 64.0;
```

2.2.3 字符类型

- 说明Rust的 char 类型大小为 4 个字节，代表 Unicode标量值，这意味着它可以支持中文，日文和韩文字符等非英文字符甚至表情符号和零宽度空格在 Rust 中都是有效的 char 值。
- 实例

```
let mut _is_ok = "this is a demo";
```

2.2.4 布尔类型

- 说明布尔型用 bool 表示，值只能为 true 或 false，占用1个字节，8bit。
- 实例

```
let mut _is_ok = true;
_is_ok = false;
```

2.2.5 复合类型

- 说明

元组用**一对 *( )*** 包括的一组数据，可以包含不同种类的数据；

- 实例

```
let tup: (i32, f64, u8) = (500, 6.4, 1);
// tup.0 等于 500
// tup.1 等于 6.4
// tup.2 等于 1
let (x, y, z) = tup;
// y 等于 6.4
```

- 说明

数组用**一对 *[ ]*** 包括的同类型数据。

- 实例

```
 let a = [1, 2, 3, 4, 5];
// a 是一个长度为 5 的整型数组

let b = ["January", "February", "March"];
// b 是一个长度为 3 的字符串数组

let c: [i32; 5] = [1, 2, 3, 4, 5];
// c 是一个长度为 5 的 i32 数组

let d = [3; 5];
// 等同于 let d = [3, 3, 3, 3, 3];

let first = a[0];
let second = a[1];
// 数组访问

a[0] = 123; // 错误：数组 a 不可变
let mut a = [1, 2, 3];
a[0] = 4; // 正确
```

2.3 函数

- 说明

基本格式：fn <函数名> (<参数>) <函数体>

- 实例

```
// 函数名称的命名风格是小写字母以下划线分割
fn main() {
    println!("Hello, world!");
    another_function();
}

fn another_function() {
    println!("Hello, runoob!");
}
//定义函数如果需要具备参数必须声明参数名称和类型：
fn main() {
    another_function(5, 6);
}

fn another_function(x: i32, y: i32) {
    println!("x 的值为 : {}", x);
    println!("y 的值为 : {}", y);
}

// {} 可以表示为代码块，在块中可以使用函数语句，最后一个步骤是表达式，此表达式的结果值是整个表达式块所代表的值。这种表达式块叫做函数体表达式。注意：x + 1 之后没有分号，否则它将变成一条语句！
fn main() {
    let x = 5;

    let y = {
        let x = 3;
        x + 1
    };

    println!("x 的值为 : {}", x);
    println!("y 的值为 : {}", y);
}
//1、函数声明返回值类型的方式：在参数声明之后用 -> 来声明函数返回值的类型（不是 : ）
//2、Rust 不支持自动返回值类型判断！如果没有明确声明函数返回值的类型，函数将被认为是"纯过程"，不允许产生返回值，return 后面不能有返回值表达式
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
```

3 程序逻辑控制

条件语句：if
循环：while，for，loop

3.1 条件语句

- 说明

```
// 条件为bool类型， Rust 中的 if 不存在单语句不用加 {} 的规则，不允许使用一个语句代替一个块
if 条件 {

} else {

}
```

- 实例

```
 if true {
    printIn!("haha")
} else {
    printIn!("hehe")
}
```

- 说明

  在 Rust 中我们可以使用 if-else 结构实现类似于三元条件运算表达式 (A ? B : C) 的效果
  Rust是一个以表达式为主的语言
  let a=1；这是语句
  if (a == b) 这里面a==b 就是表达式
  表达式：可以包括定义某值，或判断某物，最终会有一个“值”的体现

- 实例

```
let number = if a > 0 { 1 } else { -1 };
```

3.2 循环

- **说明**

  1、while循环格式如下：

```
  while 条件 {

  }
```

2、for循环格式如下：

```
  //  for 循环使用三元语句控制循环，但是 Rust 中没有这种用法
  let a = [10, 20, 30, 40, 50];
  for i in a.iter() {

  }
```

3、 loop循环

```
//  for 循环使用三元语句控制循环，但是 Rust 中没有这种用法
loop {
     break;
}
```

- 实例

```
let s = ['R', 'U', 'N', 'O', 'O', 'B'];
let mut i = 0;
let location = loop {
  let ch = s[i];
  if ch == 'O' {
      break i;
  }
  i += 1;
};
println!(" \'O\' 的索引为 {}", location); //'O' 的索引为 3

i = 0;
while i < s.len() {
  let ch = s[i];
  if ch == 'O' {
      break;
  }
  i += 1;
}
println!(" \'O\' 的索引为 {}", i);//'O' 的索引为 3

for n in 0..s.len() {
  let ch = s[n];
  if ch == 'O' {
     i = n;
     break;
  }
}
println!(" \'O\' 的索引为 {}", i);//'O' 的索引为 3
```

4 所有权

更高级的所有权讲解在后续课程的生命周期里面讲解。

- 说明

  变量范围
  所有权对大多数开发者而言是一个新颖的概念，它是 Rust 语言为高效使用内存而设计的语法机制。所有权概念是为了让 Rust 在编译阶段更有效地分析内存资源的有用性以实现内存管理而诞生的概念
  1、Rust 中的每个值都有一个变量，称为其所有者。
  2、一次只能有一个所有者。
  3、当所有者不在程序运行范围时，该值将被删除。

- 变量范围

  Rust 之所以没有明示释放的步骤是因为在变量范围结束的时候，Rust 编译器自动添加了调用释放资源函数的步骤。

```
{
    // 在声明以前，变量 s 无效
    let s = "Darren";
    // 这里是变量 s 的可用范围
}
// 变量范围已经结束，变量 s 无效
```

4.1 变量与数据交互的方式

4.1.1 移动

基本数据类型的移动

- 说明

  仅在栈中的数据的"移动"方式是直接复制，这不会花费更长的时间或更多的存储空间。"基本数据"类型有这些：所有整数类型，例如 i32 、 u32 、 i64 等。布尔类型 bool，值为 true 或 false 。所有浮点类型，f32 和 f64。字符类型 char。仅包含以上类型数据的元组（Tuples）。

- 实例

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

String 对象的数据移动

- 说明

  两个 String 对象在栈中，每个 String 对象都有一个指针指向堆中的 "hello" 字符串。在给 s2 赋值时，只有栈中的数据被复制了，堆中的字符串依然还是原来的字符串。

- 实例

```
{
    let s1 = String::from("hello");
    let s2 = s1;
    println!("{}, world!", s1); // 错误! s1 已经失效: [E0382] borrow of moved
value: `s1`.
}
```

-

4.1.2 克隆

- 说明

  但如果需要将数据单纯的复制一份以供他用，可以使用数据的第二种交互方式——克隆。

- 实例

```
 {
    let s1 = String::from("hello");
    let s2 = s1.clone();
    println!("s1 = {}, s2 = {}", s1, s2); //s1 = hello, s2 = hello
}
```

4.2 函数的所有权机制

- 说明

  如果将变量当作参数传入函数，那么它和移动的效果是一样的。

- 实例

```rust
fn main() {
    let s = String::from("hello");
    // s 被声明有效

    takes_ownership(s);
    // s 的值被当作参数传入函数
    // 所以可以当作 s 已经被移动，从这里开始已经无效

    let x = 5;
    // x 被声明有效

    makes_copy(x);
    // x 的值被当作参数传入函数
    // 但 x 是基本类型，依然有效
    // 在这里依然可以使用 x 却不能使用 s

} // 函数结束，x 无效，然后是 s. 但 s 已被移动，所以不用被释放


fn takes_ownership(some_string: String) {
    // 一个 String 参数 some_string 传入，有效
    println!("{}", some_string);
} // 函数结束，参数 some_string 在这里释放

fn makes_copy(some_integer: i32) {
    // 一个 i32 参数 some_integer 传入，有效
    println!("{}", some_integer);
} // 函数结束，参数 some_integer 是基本类型，无需释放
```

4.3 函数返回值的所有权机制

- 说明被当作函数返回值的变量所有权将会被移动出函数并返回到调用函数的地方，而不会直接被无效释放。
- 实例

```rust
fn main() {
  let s1 = gives_ownership();
  // gives_ownership 移动它的返回值到 s1

  let s2 = String::from("hello");
  // s2 被声明有效

  let s3 = takes_and_gives_back(s2);
  // s2 被当作参数移动，s3 获得返回值所有权

  println!("s1 = {}, s2 = {}, s3 = {},",s1, s2, s3); //s2 :borrow of moved
value: `s2`
} // s3 无效被释放，s2 被移动，s1 无效被释放.


fn gives_ownership() -> String {
  let some_string = String::from("hello");
  // some_string 被声明有效

  return some_string;
  // some_string 被当作返回值移动出函数
```

```
}

fn takes_and_gives_back(a_string: String) -> String {
  // a_string 被声明有效

  a_string  // a_string 被当作返回值移出函数
}
```

4.4 引用与租借

- 说明

  引用不会获得值的所有权。**引用只能租借（Borrow）值的所有权。**引用本身也是一个类型并具有一个值，这个值记录的是别的值所在的位置，但引用不具有所指值的所有权；当一个变量的值被引用时，变量本身不会被认定无效。因为"引用"并没有在栈中复制变量的值；

- 实例

```
// 当一个变量的值被引用时，变量本身不会被认定无效。因为"引用"并没有在栈中复制变量的值；
{
    let s1 = String::from("hello");
    let s2 = &s1;
    println!("s1 is {}, s2 is {}", s1, s2);
}

// 函数参数传递的道理一样
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()
}

// 这段程序不正确：因为 s2 租借的 s1 已经将所有权移动到 s3，所以 s2 将无法继续租借使用 s1 的所有权。如果需要使用 s2 使用该值，必须重新租借：
{
    let s1 = String::from("hello");
    let s2 = &s1;
    let s3 = s1;
    println!("{}", s2);
}

//这段程序是正确的
{
    fn main() {
    let s1 = String::from("hello");
    let mut s2 = &s1;
    let s3 = s1;
    s2 = &s3; // 重新从 s3 租借所有权
    println!("{}", s2);
}
```

- 说明

  可变引用与不可变引用相比除了权限不同以外，可变引用不允许多重引用，但不可变引用可以

- 实例

```
//不正确
{
    let mut s = String::from("hello");

    let r1 = &mut s;
    let r2 = &mut s;

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

}

// 正确
 {
    let  s = String::from("hello");

    let r1 = &s;
    let r2 = &s;

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

}
```

4.5 垂悬引用

- 说明

  如果放在有指针概念的编程语言里它就指的是那种没有实际指向一个真正能访问的数据的指针（注意，不一定是空指针，还有可能是已经释放的资源）

- 实例

```
fn main() {
    let reference_to_nothing = dangle(); // dangle()的返回的引用所对应的值是被释放了，
所以不允许
}

fn dangle() -> &String {
        // 在声明以前，变量 s 无效
    let s = String::from("hello"); //
    // 这里是变量 s 的可用范围
    &s
}// 变量范围已经结束，变量 s 无效
```

5 结构体

5.1 Struct 结构体

- 说明

  而结构体用于规范常用的数据结构。结构体的每个成员叫做"字段"。

- 定义struct 结构体类名 { 字段名 : 字段类型, ... }

```
struct Site {
    domain: String,
    name: String,
    nation: String,
    found: u32
}
```

- 规则

  struct 语句仅用来定义，不能声明实例，结尾不需要 ; 符号，而且每个字段定义之后用 , 分隔。

- 实例

结构体类名 { 字段名 : 字段值, ... }

```
let runoob = Site {
    domain: String::from("www.runoob.com"),
    name: String::from("RUNOOB"),
    nation: String::from("China"),
    found: 2013
};

let site = Site {
    domain: String::from("www.runoob.com"),
    name: String::from("RUNOOB"),
    ..runoob //..runoob 后面不可以有逗号。这种语法不允许一成不变的复制另一个结构体实例，意思就是说至少重新设定一个字段的值才能引用其他实例的值。
};
```

5.2 元组结构体

- 说明

  为了处理那些需要定义类型（经常使用）又不想太复杂的简单数据 struct Color(u8, u8, u8); 元组结构体对象的使用方式和元组一样，通过 . 和下标来进行访问：

- 实例

```
#[derive(Debug)]
{
    struct Color(u8, u8, u8);
    struct Point(f64, f64);

    let black = Color(0, 0, 0);
    let origin = Point(0.0, 0.0);

    println!("black = ({}, {}, {})", black.0, black.1, black.2);
    println!("origin = ({}, {})", origin.0, origin.1);

    println!("black is {:?}", black); //引入库：[derive(Debug)]，可以用 {:?} 占位符输出一整个结构体
    println!("origin is {:#?}", black); //引入库：[derive(Debug)]，属性较多的话可以使用另一个占位符 {:#?}

}
```

5.3 结构体方法

- 说明

  方法（Method）和函数（Function）类似，只不过它是用来操作结构体实例的。

- 实例

```
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!("rect1's area is {}", rect1.area());
}
```

-

5.4 结构体关联函数

- 说明

  之所以"结构体方法"不叫"结构体函数"是因为"函数"这个名字留给了这种函数：它在 impl 块中却没有 &self 参数。这种函数不依赖实例，但是使用它需要声明是在哪个 impl 块中的。

- 实例

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

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

fn main() {
    let rect = Rectangle::create(30, 50);
    println!("{:?}", rect);
}
```

-

5.5 单元结构体

- 说明

  结构体可以只作为一种象征而无需任何成员：

- 实例

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

6 枚举

rust的枚举相比以前学的语言更为灵活，但新的特性也不好理解。

6.1 基本语法

- 说明

enum 枚举名 { 枚举项 }

- 实例

```
enum Book {
    Papery, Electronic
}
let book = Book::Papery;
println!("{:?}", book); //Papery
```

- 说明

enum 枚举名 { 枚举项（类型） }

- 实例

```
enum Book {
    Papery(u32),
    Electronic(String),
}
let book = Book::Papery(1001);
println!("{:?}", book); //Papery(1001)
```

- 说明

enum Book { Papery { index: u32 }, Electronic { url: String }, }

- 实例

```
enum Book {
    Papery { index: u32 },
    Electronic { url: String },
}
let book = Book::Papery{index: 1001};//Papery { index: 1001 }
```

6.2 match 语法

- 说明

  通过 match 语句来实现分支结构（许多语言中的 switch ）

- 定义

  match 枚举类实例 { 分类1 => 返回值表达式，分类2 => 返回值表达式, ... }

- 实例

```
enum Book {
    Papery(u32),
    Electronic {url: String},
}
let book = Book::Papery(1001);

match book {
    Book::Papery(i) => {
        println!("{}", i);
    },
    Book::Electronic { url } => {
        println!("{}", url);
    }
}

// 例外情况用下划线 _ 表示：
let t = "abc";
match t {
    "abc" => println!("Yes"),
    _ => {},
}
```

6.3 Option 枚举类

- 说明

  Rust 在语言层面彻底不允许空值 null 的存在，但无奈null 可以高效地解决少量的问题，所以 Rust 引入了 Option 枚举类：

- 定义

  enum Option{ Some(T), None, }

- 实例

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

let opt: Option<&str> = Option::None;
match opt {
    Option::Some(something) => {
        println!("{}", something);
    },
    Option::None => {
        println!("opt is nothing");
    }
}
```

7 组织管理

7.1 基本概览

Rust 中有三个重要的组织概念：箱、包、模块。
箱（Crate）：“箱”是二进制程序文件或者库文件，存在于“包”中，“箱”是树状结构的，它的树根是 编译器开始运行时编译的源文件所编译的程序。
包（Package）：工程的实质就是一个“包”，包必须由一个 Cargo.toml 文件来管理，该文件描述了包的基本信息以及依赖项。
模块（Module）：Rust 中的组织单位是模块（Module）,Rust 文件的内容都是一个“难以发现”的模块。如下

7.2 模块

模块（Module）
Rust 中的组织单位是模块（Module）,Rust 文件的内容都是一个“难以发现”的模块。如下

```
mod nation {
    mod government {
        fn govern() {}
    }
    mod congress {
        fn legislate() {}
    }
    mod court {
        fn judicial() {}
    }
}
```

说明
换成树状结构：
nation
├── government
│ └── govern
├── congress
│ └── legislate
└── court
└── judicial
Rust 中的路径分隔符是 ::
绝对路径：从 crate 关键字开始描述 如：crate::nation::government::govern();
相对路径：如：nation::government::govern();
Rust 文件的内容都是一个“难以发现”的模块。

```
// second_module.rs
pub fn message() -> String {
    String::from("This is the 2nd module.")
}
```

```
 // main.rs
mod second_module;

fn main() {
    println!("This is the main module.");
    println!("{}", second_module::message());
}
```

7.3 访问权限

- 说明

公共（**pub**lic）和私有（private）。默认情况下，如果不加修饰符，模块中的成员访问权将是私有的。

- 实例

```
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"),
            }
        }
    }
}

let mut meal = back_of_house::Breakfast::summer("Rye");
```

7.4 use 关键字

- 说明

  use 关键字能够将模块标识符引入当前作用域。
  as 关键字为标识符添加别名。

- 实例

```
//因为 use 关键字把 govern 标识符导入到了当前的模块下，可以直接使用,可以使用 as 关键字为标识符添加别名。
mod nation {
    pub mod government {
        pub fn govern() {}
    }
    pub fn govern() {}
}

use crate::nation::government::govern;
use crate::nation::govern as nation_govern;

fn main() {
    nation_govern();
    govern();
}

// use 关键字可以与 pub 关键字配合使用：
mod nation {
    pub mod government {
        pub fn govern() {}
    }
    pub use government::govern;
}
```

```rust
fn main() {
    nation::govern();
}

//引用标准库
use std::f64::consts::PI;

fn main() {
    println!("{}", (PI / 2.0).sin());
}
```

7.5 调用同级目录文件函数

ex7_5_1

```
lqf@ubuntu:/mnt/hgfs/0voice/vip/rust/1-src/ex7_5_1$ tree
.
├── Cargo.lock
├── Cargo.toml
└── src
    ├── lib.rs
    └── main.rs
```

lib.rs

```rust
// lib.rs
// 文件名默认就是一个模块
// pub mod lib {
//     pub fn Hello_rutst() {
//         let my_name  = "rust";
//         let  myage = 10; // 2012 年 1 月发布
//         println!("name:{}, age:{}", my_name, myage);
//     }
// }


pub fn hello_rutst() {
    let my_name  = "rust";
    let  myage = 10; // 2012 年 1 月发布
    println!("name:{}, age:{}", my_name, myage);
}


pub mod helper {
    pub fn helper_rutst() {
        let my_name  = "rust";
        let  myage = 10; // 2012 年 1 月发布
        println!("helper name:{}, age:{}", my_name, myage);
    }
}
```

main.rs

```rust
// main.rs
mod lib; // 默认加载lib.rs或者  lib/mod.rs

fn main() {
    lib::hello_rutst();
    lib::helper::helper_rutst();
}
```

7.6 调用其目录文件函数

```
lqf@ubuntu:/mnt/hgfs/0voice/vip/rust/1-src/ex7_6_1$ tree
.
├── Cargo.lock
├── Cargo.toml
└── src
    ├── lib1
    │   ├── config.rs
    │   └── mod.rs
    ├── lib2
    │   └── mod.rs
    ├── lib22.rs
    └── main.rs
```

lib1/config.rs

```rust
pub fn show_version() {
    println!("version: 1.0");
}
```

lib1/mod.rs

```rust
pub mod config;
// 文件名默认就是一个模块
// pub mod lib {
//     pub fn Hello_rutst() {
//         let my_name  = "rust";
//         let  myage = 10; // 2012 年 1 月发布
//         println!("name:{}, age:{}", my_name, myage);
//     }
// }


pub fn hello_rutst() {
    let my_name  = "rust";
    let  myage = 10; // 2012 年 1 月发布
    println!("lib1 name:{}, age:{}", my_name, myage);
}


pub mod helper {
    pub fn helper_rutst() {
        let my_name  = "rust";
        let  myage = 10; // 2012 年 1 月发布
        println!("lib1 helper name:{}, age:{}", my_name, myage);
```

```
    }
}
```

lib2/mod.rs

```
pub fn hello_rutst() {
    let my_name  = "rust";
    let  myage = 10; // 2012 年 1 月发布
    println!("lib2 name:{}, age:{}", my_name, myage);
}


pub mod helper {
    pub fn helper_rutst() {
        let my_name  = "rust";
        let  myage = 10; // 2012 年 1 月发布
        println!("lib2 helper name:{}, age:{}", my_name, myage);
    }
}
```

main.rs

```
// main.rs
// mod lib; // 默认加载lib.rs或者  lib/mod.rs
mod lib1;
mod lib2;
fn main() {
    lib1::config::show_version();
    lib1::hello_rutst();
    lib1::helper::helper_rutst();
    lib2::hello_rutst();
    lib2::helper::helper_rutst();
}
```

8 练习用户实体类、调用实体类 20分钟

封装一个结构体、并对外提供调用方法。

user包含以下字段：
id i32 // 用户id
name String // 用户名
age u8 // 年龄
tags [&'static str; 5] // 数组标签
通过new方法，用来初始化实体。

```
.
├── Cargo.lock
├── Cargo.toml
└── src
    ├── main.rs
    └── models
        ├── mod.rs
        └── user_model.rs
```

models/mod.rs

```
pub mod user_model;
```

models/user_model.rs

```
#[derive(Debug)] //这个`derive` 属性会自动创建所需的实现，使限定的`struct` 能使用
`fmt::Debug` 打印。
pub struct UserModel { // 结构体驼峰
    pub id: i32,
    pub name: String,
    pub age: u8,
    pub tags: [&'static str; 5]
}

// 函数 蛇形
pub fn new_user_model()-> UserModel {
    UserModel{
        id: 0,
        name: String::new(),
        age: 0,
        tags: ["";5]
    }
}
// 结构体方法
impl UserModel {
    pub fn set_name(&mut self, name:String)   {
        self.name = name
    }
    pub fn set_age(&mut self, age:u8)   {
        self.age = age
    }
    pub fn get_age(&self) -> u8  {
        self.age
    }
}
```

main.rs

```
mod models;
fn main() {
    let mut user = models::user_model::new_user_model();
    println!("1 {:?}", user);
    user.id = 20;
    user.name = String::from("Ovoice");
    user.tags[0] = "go";
    user.tags[1] = "rust";
    println!("2 {:?}", user);

    user.set_name(String::from("king"));
    user.set_age(18);
    println!("2 {:?}, age:{}", user, user.get_age());
}
```

补充

println打印

```rust
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
}
```