Rust 简明教程

Rust 简明教程系列文章链接：

- Rust 简明教程 (Nov 24, 2019)

Rust 是一门系统编程语言(Systems Programming Language)，兼顾安全(Safety)、性能(Speed)和并发(Concurrency)。Rust作为一门底层的系统编程语言，理论上，使用 C/C++ 的领域都可以使用Rust实现，例如对硬件需要精细控制的嵌入式编程、对性能要求极高的应用软件（数据库引擎、浏览器引擎，3D渲染引擎等）。相对于 C/C++ 的系统性缺陷（内存管理不当造成的安全漏洞），Rust通过所有权(Ownership)机制在编译期间确保内存安全，无需垃圾回收(Garbage Collection, GC)，也不需要手动释放内存。

1. Hello World

1.1 安装 Rust

- 在线安装
  - Windows：下载 rustup-init.exe，自动引导安装。
  - Linux：*curl –proto ‘=https’ –tlsv1.2 -sSf https://sh.rustup.rs | sh*
- 离线安装
  - 下载独立安装包
  - Windows: 下载 .msi 文件，双击安装即可。
  - Linux：下载 .tar.gz 文件，*tar -xvf xxx.tar.gz* 解压后，执行 install.sh 即可。
- 查看版本

```
$ rustc --version
rustc 1.39.0 (4560ea788 2019-11-04)
$ cargo --version
cargo 1.39.0 (1c6ec66d5 2019-09-30)
```

1.2 第一个Rust程序

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

- 保存为 hello_world.rs，rs 为 Rust 语言的后缀。
- 编译：*rustc hello_world.rs*。
- 执行：*./hello_world (Linux) ，hello_world.exe*(Windows)

尝试下 *println!* 更多的用法。

```
fn main() {
    println!("{}, {}!", "Hello", "world"); // Hello, world!
    println!("{0}, {1}!", "Hello", "world"); // Hello, world!
    println!("{greeting}, {name}!", greeting="Hello", name="world"); // Hello, world!

    let y = String::from("Hello, ") + "world!";
    println!("{}", y); // Hello, world!
}
```

以上代码将输出

```
Hello, world!
Hello, world!
Hello, world!
Hello, world!
```

1.3 使用 Cargo

为了方便之后的调试和学习，先介绍 Rust 内置的包管理和构建系统 `Cargo` ，crates.io 是 Rust 的社区仓库。

- 创建新项目：*cargo new*
- 编译：*cargo build*
- 运行：*cargo run*
- 更新项目依赖：*cargo update*
- 执行测试：*cargo test*
- 生成文档：*cargo doc*
- 静态检查：*cargo check*
- 新建二进制(Binary/Executable)项目

```
$ cargo new tutu --bin
$ cd tutu && tree
├── Cargo.toml
```

在 main.rs 中写入

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

在项目目录下执行 *cargo run*

```
$ cargo run
 tutu git:(master) ✗ cargo run
   Compiling tutu v0.1.0 (/xxx/demo/tutu)
    Finished dev [unoptimized + debuginfo] target(s) in 0.49s
     Running `target/debug/tutu`
Hello, Cargo!
```

- 新建 Library 项目

```
$ cargo new tutu --lib
$ cd tutu && tree
├── Cargo.toml
└── src
    └── lib.rs
```

- Cargo.toml 是工程的描述文件，包含 Cargo 所需的所有元信息。
- src 放置源代码。
- main.rs / lib.rs 是入口文件。

运行 *cargo run* 或 *cargo build*，可执行文件将生成在 target/debug/ 目录，运行 *cargo build –release*，可执行文件将生成在 target/release/ 。

2 基本概念

2.1 注释

```

/// 外部注释
mod test {
    // 行注释
    /* 块注释 */
```

```
8
9   mod test {
10      //! 包/模块级别的注释
11
12      // ...
13  }
```

> /// 用于 mod 块外部， //! 用于书写包/模块级别的注释
> 注释支持 markdown 语法，使用 *cargo doc* 生成 HTML 文档。

2.2 变量

- 局部变量

Rust 中变量默认是不可变的(immutable)，称为变量绑定(Variable bindings)，使用 `mut` 标志为可变(mutable)。

let 声明的变量是局部变量，声明时可以不初始化，使用前初始化即可。Rust是静态类型语言，编译时会检查类型，使用let声明变量时可以省略类型，编译时会推断一个合适的类型。

```
1   // 不可变
2   let c;
3   let a = true;
4   let b: bool = true;
5   let (x, y) = (1, 2);
6   c = 12345;
7
8   // 可变
9   let mut z = 5;
10  z = 6;
```

- 全局变量

rust 中可用 static 声明全局变量。用 static 声明的变量的生命周期是整个程序，从启动到退出，它占用的内存空间是固定的，不会在执行过程中回收。另外，static 声明语句，必须显式标明类型，不支持类型自动推导。全局变量在声明时必须初始化，且须是简单赋值，不能包括复杂的表达式、语句和函数调用。

```
1   // 静态变量(不可变)
2   static N: i32 = 5;
3
4   // 静态变量(可变)
5   static mut N: i32 = 5;
```

const 的生命周期也是整个程序，const 与 static 的最大区别在于，编译器开个一定会给 const 常量分配内存空间，在编译过程中，它很可能会被内联优化，类似于C语言的宏定义。

```
const N: i32 = 5;
```

2.3 函数

使用 `fn` 声明函数。

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

参数需要指定类型

```
fn print_sum(a: i8, b: i8) {
    println!("sum is: {}", a + b);
}
```

默认返回值为空 `()` ，如果有返回值，需要使用 `->` 指定返回类型。

```
fn plus_one(a: i32) -> i32 {
    a + 1
    // 等价于 return a + 1，可省略为 a + 1
}
```

可以利用元组(tuple)返回多个值

```
fn plus_one(a: i32) -> (i32, i32) {
    (a, &a + 1)
}

fn main() {
    let (add_num, result) = plus_one(10);
    println!("{} + 1 = {}", add_num, result); // 10 + 1 = 11
}
```

函数指针也可以作为变量使用

2.4 基本数据类型

- 布尔值(bool)
- 字符(char)
- 有符号整型(i8, i16, i32, i64, i128)
- 无符号整型(u8, u16, u32, u64, u128)
- 指针大小的有符号/无符号整型(isize/usize，取决于计算机架构，32bit 的系统上，isize 等价于i32)
- 浮点数(f32, f64)
- 数组(arrays)，由相同类型元素构成，长度固定。

```
let a = [1, 2, 3]; // a[0] = 1, a[1] = 2, a[2] = 3
let mut b = [1, 2, 3];

let c: [int; 3] = [1, 2, 3]; // [类型; 数组长度]

let d: ["my value"; 3]; //["my value", "my value", "my value"];

let e: [i32; 0] = []; // 空数组

println!("{:?}", a); //[1, 2, 3]
```

数组(arrays)的长度是可不变的，动态/可变长数组可以使用 `Vec` (非基本数据类型)。

- 元组(tuples)，由相同/不同类型元素构成，长度固定。

```
let a = (1, 1.5, true, 'a', "Hello, world!");
// a.0 = 1, a.1 = 1.5, a.2 = true, a.3 = 'a', a.4 = "Hello, world!"

let b: (i32, f64) = (1, 1.5);

let (c, d) = b; // c = 1, d = 1.5
let (e, _, _, _, f) = a; //e = 1, f = "Hello, world!", _ 作为占位符使用，表示忽略该位置
let g = (0,); // 只包含一个元素的元组

let h = (b, (2, 4), 5); //((1, 1.5), (2, 4), 5)

println!("{:?}", a); //(1, 1.5, true, 'a', "Hello, world!")
```

- 切片(slice)，指向一段内存的指针。

切片并没有拷贝原有的数组，只是指向原有数组的一个连续部分，行为同数组。访问切片指向的数组/数据结构，可以使用 & 操作符。

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

let b: &[i32] = &a; // 全部
let c = &a[0..4]; // [0, 4)
let d = &a[..]; // 全部

let e = &a[1..3]; // [2, 3]
let e = &a[1..]; // [2, 3, 4]
let e = &a[..3]; // [1, 2, 3]
```

- 字符串(str)

在 Rust 中， `str` 是不可变的静态分配的一个未知长度的UTF-8字节序列。 `&str` 是指向该字符串的切片。

```
let a = "Hello, world!"; //a: &'static str
let b: &str = "你好，世界！";
```

字符串切片 `&str` 指向的字符串是静态分配的，在 Rust 中，有另一个堆分配的，可变长的字符串类型 `String` (非基本数据类型)。通常由字符串切片 `&str` 通过 *to_string()* 或 *String::from()* 方法转换得到。

```
let s1 = "Hello, world!".to_string();
let s2 = String::from("Hello, world!");
```

- 函数(functions)

函数指针也是基本数据类型，可以赋值给其他的变量。

2.5 操作符

- 算数运算符

```
+ - * / %
let a = 5;
let b = a + 1; //6
let c = a - 1; //4
let d = a * 2; //10
let e = a / 2; // * 2 not 2.5
let f = a % 2; //1
```

- 比较运算符

```
== = != < > <= >=
let a = 1;
let b = 2;

let c = a == b; //false
let d = a != b; //true
let e = a < b; //true
let f = a > b; //false
let g = a <= a; //true
let h = a >= a; //true

let i = true > false; //true
let j = 'a' > 'A'; //true
```

- 逻辑运算符

```
 ! && ||
let a = true;
let b = false;

let c = !a; //false
let d = a && b; //false
let e = a || b; //true
```

- 位运算符

```
& | ^ << >>
let a = 1;
let b = 2;

let c = a & b; //0  (01 && 10 -> 00)
let d = a | b; //3  (01 || 10 -> 11)
let e = a ^ b; //3  (01 != 10 -> 11)
let f = a << b; //4  (左移 -> '01'+'00' -> 100)
let g = a >> a; //0  (右移 -> o1--> 0)
```

- 赋值运算符

```
2
3   a += 5; //2 + 5 = 7
4   a -= 2; //7 - 2 = 5
5   a *= 5; //5 * 5 = 25
6   a /= 2; //25 / 2 = 12 not 12.5
7   a %= 5; //12 % 5 = 2
8
9   a &= 2; //10 && 10 -> 10 -> 2
10  a |= 5; //010 || 101 -> 111 -> 7
11  a ^= 2; //111 != 010 -> 101 -> 5
12  a <<= 1; //'101'+'0' -> 1010 -> 10
13  a >>= 2; //1010--> 10 -> 2
```

- 类型转换运算符: as

```
1   let a = 15;
2   let b = (a as f64) / 2.0; //7.5
```

- 借用(Borrowing)与解引用(Dereference)操作符

Rust 引入了所有权(Ownership)的概念，所以在引用(Reference)的基础上衍生了借用(Borrowing)的概念，所有权概念不在这里展开。

简单而言，引用是为已存在变量创建一个别名；获取引用作为函数参数称为借用；解引用是与引用相反的动作，目的是返回引用指向的变量本身。

```
1   // 引用/借用:  & &mut
2
3   fn main() {
4       let s1 = String::from("hello");
5       let len = calculate_length(&s1);
6       println!("The length of '{}' is {}.", s1, len); // The length of 'hello' is 5.
7   }
8
9   fn calculate_length(s: &String) -> usize { // 获取引用作为函数参数称为借用
10      s.len()
11  }
```

```
1   // 解引用:  *
2
3   fn main() {
```

```
    {
        let third  = v.get_mut(2).unwrap();
        *third += 50;
    }
    println!("v={:?}", v); // v=[1, 2, 53, 4, 5]
}
```

> References and Borrowing - 官方指南
> What is ownership - 官方指南

2.6 控制流(Control Flows)

- if - else if - else

```
let team_size = 7;
if team_size < 5 {
    println!("Small");
} else if team_size < 10 {
    println!("Medium");
} else {
    println!("Large");
}

// 条件块中有返回值时，类型需要一致，可替代C语言的三目运算符
let is_below_eighteen = if team_size < 18 { true } else { false };
```

- match

可替代C语言的 `switch case` 。

```
let tshirt_width = 20;
let tshirt_size = match tshirt_width {
    16 => "S", // check 16
    17 | 18 => "M", // check 17 and 18
    19 ... 21 => "L", // check from 19 to 21 (19,20,21)
    22 => "XL",
    _ => "Not Available",
};
println!("{}", tshirt_size); // L
```

_ 表示匹配剩下的任意情况。

```rust
let mut a = 1;
while a <= 10 {
    println!("Current value : {}", a);
    a += 1; // Rust不支持++/--自增自减语法
}
```

- loop

类似于C语言的 `while(1)`

```rust
let mut a = 0;
loop {
    if a == 0 {
        println!("Skip Value : {}", a);
        a += 1;
        continue;
    } else if a == 2 {
        println!("Break At : {}", a);
        break;
    }
    println!("Current Value : {}", a);
    a += 1;
}

// Skip Value : 0
// Current Value : 1
// Break At : 2
```

- for

```rust
for a in 0..10 { //(a = 0; a <10; a++)
    println!("Current value : {}", a);
}

'outer_for: for c1 in 1..6 { //set label outer_for
    'inner_for: for c2 in 1..6 {
        println!("Current Value : [{}][{}]", c1, c2);
        if c1 == 2 && c2 == 2 { break 'outer_for; } // 结束外层循环
    }
}

let group : [&str; 4] = ["Mark", "Larry", "Bill", "Steve"];
```

```
15  }
```

在 for 表达式中的 `break 'outer_for` ，loop 和 while 也有相同的使用方式。

3. 其他数据类型

3.1 结构体(struct)

和元组(tuple)一样，结构体(struct)支持组合不同的数据类型，但不同于元组，结构体需要给每一部分数据命名以标志其含义。因而结构体比元组更加灵活，不依赖顺序来指定或访问实例中的值。

- 定义结构体

```
1  struct User {
2      username: String,
3      email: String,
4      sign_in_count: u64,
5      active: bool,
6  }
```

- 创建实例

```
1  let user1 = User {
2      email: String::from("someone@example.com"),
3      username: String::from("someusername123"),
4      active: true,
5      sign_in_count: 1,
6  };
```

- 修改某个字段的值

```
1  let mut user1 = User {
2      email: String::from("someone@example.com"),
3      username: String::from("someusername123"),
4      active: true,
5      sign_in_count: 1,
6  };
7
8  user1.email = String::from("anotheremail@example.com");
```

- 变量与字段名同名的简写语法

```
    User {
        email,
        username,
        active: true,
        sign_in_count: 1,
    }
}
```

- 元组结构体(tuple structs)

元组结构体有着结构体名称提供的含义，但没有具体的字段名。在参数个数较少时，无字段名称，仅靠下标也有很强的语义时，为每个字段命名就显得多余了。例如：

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

let black = Color(0, 0, 0);
let origin = Point(3, 4);
```

VS

```
struct Point {
    x: i32
    y: i32
}
let origin = Point {
    x: 3
    y: 4
}
```

3.2 枚举(enum)

- 定义枚举

```
enum IpAddrKind {
    V4,
    V6,
}
```

- 使用枚举值

```
fn route(ip_type: IpAddrKind) { }
route(four);
route(IpAddrKind::V6);
```

- 枚举成员关联数据

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

let home = IpAddr::V4(127, 0, 0, 1);

let loopback = IpAddr::V6(String::from("::1"));
```

更复杂的例子

```
enum Message {
    Quit, //  不关联数据
    Move { x: i32, y: i32 }, // 匿名结构体
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

- match 控制流

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


 fn value_in_cents(coin: Coin) -> u32 {
     match coin {
         Coin::Penny => {
             println!("Lucky penny!");
             1
         },
         Coin::Nickel => 5,
```

```
18        }
19    }
```

- Option

Option是标准库中定义的一个非常重要的枚举类型。Option 类型应用广泛因为它编码了一个非常普遍的场景，即一个值要么有值要么没值。对Rust而言，变量在使用前必须要赋予一个有效的值，所以不存在空值(Null)，因此在使用任意类型的变量时，编译器确保它总是有一个有效的值，可以自信使用而无需做任何检查。如果一个值可能为空，需要显示地使用 `Option<T>` 来表示。

Option 的定义如下：

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

`Option<T>` 包含2个枚举项：

1) None，表明失败或没有值
2) Some(value)，元组结构体，封装了一个 T 类型的值 value

得益于 `Option` ，Rust 不允许一个可能存在空值的值，像一个正常的有效值那样工作，在编译时就能够检查出来。Rust显得更加安全，不用担心出现其他语言运行时才会出现的空指针异常的bug。例如：

```
let x: i8 = 5; // Rust 没有空值(Null)，因此 i8只能被赋予一个有效值。
let y: Option<i8> = Some(5); // y 可能为空，需要显示地表示为枚举类型 Option
let sum = x + y;
```

尝试将不可能出现无效值的 `x:i8` 与可能出现无效值的 `y: Option<i8>` 相加时，编译器会报错：

```
error[E0277]: the trait bound `i8: std::ops::Add<std::option::Option<i8>>` is
not satisfied
 -->
  |
5 |     let sum = x + y;
  |                 ^ no implementation for `i8 + std::option::Option<i8>`
  |
```

总结一下，如果一个值可能为空，必须使用枚举类型 `Option<T>` ，否则必须赋予有效值。而为了使用 `Option<T>` ，需要编写处理每个成员的代码，当*T*为有效值时，才能够从 *Some(T)* 中取出 *T* 的值来使用，如

例如，当y为有效值时，返回x和y的和；为空值时，返回x。

```
fn plus(x: i8, y: Option<i8>) -> i8 {
    match y {
        None => x,
        Some(i) => x + i,
    }
}

fn main() {
    let y1: Option<i8> = Some(5);
    let y2: Option<i8> = None;

    let z1 = plus(10, y1);
    let z2 = plus(10, y2);

    println!("z1={}, z2={}", z1, z2); // z1=15, z2=10
}
```

- if let 控制流

match 还有一种简单场景，可以简写为 `if let` 。如下，y有值时，打印和，y无值时，啥事也不做。

```
fn plus(x: i8, y: Option<i8>) {
    match y {
        Some(i) => { println!("x + y = {}", x + i) },
        None => {},
    }
}

fn main() {
    let y1: Option<i8> = Some(5);
    let y2: Option<i8> = None;

    plus(10, y1); // x + y = 15
    plus(10, y2);
}
```

简写为 `if let` ，则是

```
fn plus(x: i8, y: Option<i8>) {
    if let Some(i) = y {
```

```
}
```

如果只使用 `if` 呢？

```
fn plus(x: i8, y: Option<i8>) {
    if y.is_some() {
        let i = y.unwrap(); // 获得 Some 中的 T 值。
        println!("x + y = {}", x + i);
    }
}
```

`if let` 语句也可以包含 `else`。

```
fn plus(x: i8, y: Option<i8>) {
    if let Some(i) = y {
        println!("x + y = {}", x + i);
    } else {
        println!("y is None");
    }
}

// 等价于

fn plus(x: i8, y: Option<i8>) {
    match y {
        Some(i) => { println!("x + y = {}", x + i) },
        None => { println!("y is None") },
    }
}
```

3.3 实现方法和接口(impl & traits)

- 实现方法(impl)

```
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
```

```
}

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

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

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

- 关联函数(associated functions)

关联函数不以 `self` 作为参数，关联函数之所以成为函数而不是方法，是因为关联函数并不作用于一个结构体的实例。我们之前创建字符串类型时，使用过的 *String::from* 就是关联函数。关联函数经常被用作返回一个结构体新实例的构造函数。例如我们可以提供一个关联函数，它接受一个维度参数并且同时作为宽和高，这样可以更轻松的创建一个正方形 Rectangle 而不必指定两次同样的值：

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

let rect2 = Rectangle::square(10);
```

- 实现接口(traits)

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

impl Summary for Rectangle {
    fn summarize(&self) -> String {
        format!("{{width={}, height={}}}", self.width, self.height)
    }
```

```rust
// 接口也支持继承
trait Person {
    fn full_name(&self) -> String;
}

trait Employee : Person { //Employee inherit from person trait
    fn job_title(&self) -> String;
}

trait Expat {
    fn salary(&self) -> f32
}

trait ExpatEmployee : Employee + Expat { // 多继承，同时继承 Employee 和 Expat
    fn additional_tax(&self) -> f64;
}
```

3.3 泛型(Generics)

当我们实现一个函数或者数据结构时，往往希望参数可以支持不同的类型，泛型可以解决这个问题。声明参数类型时，换成大写的字母，例如字母 T，同时使用 `<T>` 告诉编译器 *T* 是泛型。

- 函数中使用泛型

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

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

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];
```

```
22      println!("The largest char is {}", result);
23  }
```

- 结构体使用泛型

```
1   struct Point<T> {
2       x: T,
3       y: T,
4   }
5
6   fn main() {
7       let integer = Point { x: 5, y: 10 };
8       let float = Point { x: 1.0, y: 4.0 };
9   }
```

- 枚举使用泛型

```
1   enum Option<T> {
2       Some(T),
3       None,
4   }
5
6   enum Result<T, E> {
7       Ok(T),
8       Err(E),
9   }
```

Result 枚举有两个泛型类型，T 和 E。Result 有两个成员：Ok，它存放一个类型 T 的值，而 Err 则存放一个类型 E 的值。这个定义使得 Result 枚举能很方便的表达任何可能成功（返回 T 类型的值）也可能失败（返回 E 类型的值）的操作。回忆一下示例 9-3 中打开一个文件的场景：当文件被成功打开 T 被放入了 *std::fs::File* 类型而当打开文件出现问题时 E 被放入了 *std::io::Error* 类型。

- 方法中使用泛型

```
1   struct Point<T> {
2       x: T,
3       y: T,
4   }
5
6   impl<T> Point<T> {
7       fn x(&self) -> &T {
```

```
10  }
11
12  fn main() {
13      let p = Point { x: 5, y: 10 };
14
15      println!("p.x = {}", p.x());
16  }
```

3.4 常见集合 Vec

- 新建

```
1  let v: Vec<i32> = Vec::new();  // 空集合
2  // let v = vec![1, 2, 3];  // 含初始值的集合，vec!是为方便初始化Vec提供的宏。
3
4  println!("第三个元素 {}", &v[2]); // 3
5  println!("第100个元素 {}", &v[100]); // panic error
6
7  assert_eq!(v.get(2), Some(&3));
8  assert_eq!(v.get(100), None);
```

v.get(2) 和 &v[2] 都能获取到 Vec 的值，区别在于 &v[2] 返回的是该元素的引用，引用一个不存在的位置，会引发错误。v.get(2) 返回的是枚举类型 `Option<&T>` 。v.get(2) 返回的是 Some(&3)，v.get(100) 返回的是 *None*。

- 更新

```
1  let v: Vec<i32> = Vec::new();
2  v.push(5);
3  v.push(6);
4  v.push(7);
5  v.push(8);
6  v.pop() //删除最后一个元素
```

- 遍历

```
1  let v = vec![100, 32, 57];
2  for i in &v {
3      println!("{}", i);
4  }
5
```

```
8       *i += 50;
9   }
```

- if let 控制流

如果我们想修改 Vec 中第2个元素的值呢？可以这么写：

```
1   fn main() {
2       let mut v = vec![1, 2, 3, 4, 5];
3       {
4           let third  = v.get_mut(2).unwrap();
5           *third += 50;
6       }
7       println!("v={:?}", v); // v=[1, 2, 53, 4, 5]
8   }
```

因为 v.get_mut() 的返回值是 Option<T> 枚举类型，那么可以使用 if let 来简化代码。

```
1   fn main() {
2       let mut v = vec![1, 2, 3, 4, 5];
3       if let Some(third) = v.get_mut(2) {
4           *third += 50;
5       }
6       println!("v={:?}", v); // v=[1, 2, 53, 4, 5]
7   }
```

- while let 控制流

if let 可以用于单个元素的场景， while let 就适用于遍历的场景了。

```
1   let mut stack = vec![1, 2, 3, 4, 5];
2   while let Some(top) = stack.pop() {
3       println!("{}", top); // 依次打印 5 4 3 2 1
4   }
```

> 更多用法参考：Vec - 官方文档

3.5 常见集合 String

Rust 的核心语言中只有一种字符串类型： str ，字符串切片，它通常以被借用的形式出现， &str 。这里提到的字符串，是字节的集合，外加一些常用方法实现。因为是集合，增持增删改，长度也可变。

```
let mut s1 = String::new();
let s2 = "initial contents".to_string();
let s3 = String::new();
```

- 更新

```
let mut s = String::from("foo");
s.push_str("bar"); // 附加字符串
s.push('!') // 附加单字符
assert_eq!(s.remove(0), 'f'); // 删除某个位置的字符

let s1 = String::from("Hello, ");
let s2 = String::from("world!");
let s3 = s1 + &s2;
```

- format

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

let s = format!("{}-{}-{}", s1, s2, s3);
println!("{}", s); // tic-tac-toe
```

- 索引

```
let v = String::from("hello");
assert_eq!(Some('h'), v.chars().nth(0));
```

- 遍历

```
let v = String::from("hello");
for c in v.chars() {
    println!("{}", c);
}
```

在Rust内部，String 是一个 `Vec<u8>` 的封装，但是有些字符可能会占用超过2个字符，所以 `String` 不支持直接索引，如果需要索引需要使用 *chars()* 转换后再使用。

3.6 常见集合 HashMap

- 新建

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

let mut scores = HashMap::new();

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

这里使用了 use 引入了 HashMap 结构体。

- 访问

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

let mut scores = HashMap::new();

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

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

- 更新

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

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10); // 10
scores.insert(String::from("Blue"), 25); // 25
// Blue 存在则不更新，不存在则更新，因此scores['Blue'] 仍为 25
scores.entry(String::from("Blue")).or_insert(50);
```

> 参考 HashMap - 官方文档

4 错误处理

4.1 不可恢复错误 panic!

Rust 有 panic!宏。当执行这个宏时，程序会打印出一个错误信息，展开并清理栈数据，然后接着退出。出现这种场景，一般是出现了一些不知如何处理的场景。

- 直接调用

```
fn main() {
    panic!("crash and burn");
}
```

执行 *cargo run* 将打印出

```
$ cargo run
Compiling tutu v0.1.0 (/xxx/demo/tutu)
    Finished dev [unoptimized + debuginfo] target(s) in 0.28s
     Running `target/debug/tutu`
thread 'main' panicked at 'crash and burn', src/main.rs:2:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace.
```

最后2行包含了 `panic!` 导致的报错信息，第1行是源码中 `panic!` 出现的位置 src/main.rs:2:5

- 代码bug引起的错误

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

    v[99]; // 越界
}
```

和之前一样，*cargo run* 的报错信息只有2行，缺少函数的调用栈，为了便于定位问题，可以设置 *RUST_BACKTRACE* 环境变量来获得更多的调用栈信息，Rust 中称之为 `backtrace` 。通过 `backtrace` ，可以看到执行到目前位置所有被调用的函数的列表。

例如执行 *RUST_BACKTRACE=1 cargo run*，这种方式的好处在于，环境变量只作用于当前命令。

```
$ RUST_BACKTRACE=1 cargo run
Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/tutu`
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 99', /rus
stack backtrace:
```

```
    ...
    17: <alloc::vec::Vec<T> as core::ops::index::Index<I>>::index
                at /rustc/xxx/src/liballoc/vec.rs:1796
    18: tutu::main
                at src/main.rs:4
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
```

第一行的报错信息，说明了错误的原因，长度越界。紧接着打印出了函数调用栈，src/main.rs:4 -> liballoc/vec.rs:1796 -> ...

在windows下，可以执行 *set RUST_BACKTRACE=1 && cargo run*。

- release

当出现 panic 时，程序默认会开始 展开（unwinding），这意味着 Rust 会回溯栈并清理它遇到的每一个函数的数据，不过这个回溯并清理的过程有很多工作。另一种选择是直接 终止（abort），这会不清理数据就退出程序。那么程序所使用的内存需要由操作系统来清理。

release模式下，希望程序越小越好，可以在 `Cargo.toml` 中设置 panic 为 abort。

```
[profile.release]
panic = "abort"
```

4.2 可恢复错误 Result

- 处理 Result

有些错误，希望能够捕获并且做相应的处理，Rust 提供了 `Result` 机制来处理可恢复错误，类似于其他语言中的 `try catch` 。

这是 `Result<T, E>` 的定义

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

有些函数会返回 `Result` ，那怎么知道一个函数的返回对象是不是 `Result` 呢？很简单！

```
fn main(){
    let f: u32 = File::create("hello.txt");
}
```

```
= note: expected type `u32`
            found type `std::result::Result<std::fs::File, std::io::Error>`
```

从报错信息可以看出，*File::create* 返回的是一个 `Result<fs::File. io::Error>` 对象，如果没有异常，我们可以从 `Result::Ok<T>` 获取到文件句柄。

下面是一个完整的示例，创建 hello.txt 文件，并尝试写入 “Hello, world!” 。

```rust
use std::fs::File;
use std::io::prelude::*;

fn main() {
    let f = File::create("hello.txt");

    let mut file = match f {
        Ok(file) => file,
        Err(error) => {
            panic!("Problem create the file: {:?}", error)
        },
    };
    match file.write_all(b"Hello, world!") {
        Ok(()) => {},
        Err(error) => {
            panic!("Failed to write: {:?}", error)
        }
    };
}
```

如果执行成功，可以看到在工程根目录下，多出了 hello.txt 文件。

- unwrap 和 expect

`Result` 的处理有时太过于繁琐，Rust 提供了一种简洁的处理方式 *unwrap* 。即，如果成功，直接返回 `Result::Ok<T>` 中的值，如果失败，则直接调用 `!panic` ，程序结束。

```rust
let f = File::open("hello.txt").unwrap(); // 若成功，f则被赋值为文件句柄，失败则结束。
```

expect 是更人性化的处理方式，允许在调用 `!panic` 时，返回自定义的提示信息，对调试很有帮助。

```rust
let f = File::open("hello.txt").expect("Failed to open hello.txt");
```

我们可以实现类似于 *File::open* 这样的函数，让调用者能够自主绝对如何处理成功/失败的场景。

```
use std::io;
use std::io::Read;
use std::fs::File;

fn read_username_from_file() -> Result<String, io::Error> {
    let f = File::open("hello.txt");

    let mut f = match f {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut s = String::new();

    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}
```

上面的函数如果成功，则返回 hello.txt 的文本字符串，失败，则返回 `io::Error` 。

- 更简单的实现方式

```
use std::io;
use std::io::Read;
use std::fs::File;

fn read_username_from_file() -> Result<String, io::Error> {
    let mut f = File::open("hello.txt")?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    Ok(s)
}
```

这种写法使用了 `?` 运算符向调用者返回错误。

作用是：如果 `Result` 的值是 Ok，则该表达式返回 Ok 中的值且程序继续执行。如果值是 Error ，则将 Error 的值作为整个函数的返回值，好像使用了 `return` 关键字一样。这样写，逻辑更为清晰。

5 包、crate和模块

5.1 包和 crate

```
.
├── Cargo.lock
├── Cargo.toml
├── benches
│   └── large-input.rs
├── examples
│   └── simple.rs
├── src
│   ├── bin
│   │   └── another_executable.rs
│   ├── lib.rs
│   └── main.rs
└── tests
    └── some-integration-tests.rs
```

一个 Cargo 项目即一个包(Package)，一个包至少包含一个crate；可以包含零个或多个二进制crate（binary crate），但只能包含一个库crate(library crate)。 src/main.rs 是与包名同名的二进制 crate 的根，其他的二进制 crate 的根放置在 src/bin 目录下； src/lib.rs 是与包名同名的库 crate 的根。

5.2 模块

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

- 声明模块

Rust中使用mod来声明模块，模块允许嵌套，可以使用模块名作为路径使用，例如：

```rust
// src/main.rs

mod math {
    mod basic {
        fn plus(x: i32, y: i32) -> i32 { x + y}
        fn mul(x: i32, y: i32) -> i32 { x * y}
    }
}

fn main() {
```

```
13  }
```

- 引入作用域

使用 `use` 可以将路径引入作用域。

我们在 src/lib.rs 中声明一个模块，在 src/main.rs 中调用。

```
1   // src/lib.rs
2
3   pub mod greeting {
4       pub fn hello(name: &str) { println!("Hello, {}", &name) } // pub 才能外部可见
5   }
```

```
1   // src/main.rs
2
3   use tutu;
4
5   fn main() {
6       tutu::greeting::hello("Jack"); // Hello, Jack
7   }
```

路径的长度可以自由定义，也可以写成

```
1   // src/main.rs
2
3   use tutu::greeting;
4
5   fn main() {
6       greeting::hello("Jack");
7   }
```

src/main.rs 和 src/lib.rs 属于不同的 crate ，所以引入作用域时，需要带上包名 *tutu* 。

- 分隔模块

在 src/lib.rs 中可以使用 `mod` 声明多个模块，但有时为了可读性，习惯将每个模块写在独立的文件中。

新建 src/greeting.rs，写入

```
1   // src/greeting.rs
2
```

在 src/lib.rs 可以这样使用，

```
// src/lib.rs
pub mod greeting;

pub fn func() {
    greeting::hello("Tom");
}
```

关键点就在于 `mod greeting;` 这一行， `mod greeting` 后面没有具体实现，而是紧跟分号，则声明 greeting 的模块内容位于 src/greeting.rs 中。

其他crate，例如 src/main.rs 中的使用方式没有任何变化。

```
// src/main.rs

use tutu::greeting;

fn main() {
    greeting::hello("Jack");
}
```

6 测试

6.1 单元测试(unit tests)

```
fn plus(x: i32, y: i32) -> i32 {
    x + y
}

fn main() {
    let x = 10;
    let y = 20;
    println!("{} + {} = {}", x, y, plus(x, y))
}

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

```
$ cargo test
running 1 test
test it_works ... ok

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

更规范的写法是在每个源文件中，创建包含测试函数的 tests 模块，测试用例写在 tests 模块中，并使用 `cfg(test)` 标注模块。

```
fn plus(x: i32, y: i32) -> i32 {
    x + y
}

fn main() {
    println!("2 + 3 = {}", plus(2, 3));
}

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

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

因为内部测试的代码和源码在同一文件，因而需要使用 `#[cfg(test)]` 注解告诉 Rust 只在执行 cargo test 时才编译和运行测试代码，因此，可以在构建库时节省时间，并减少编译文件产生的文件大小。

单元测试的好处在于，可以测试私有函数。如果在独立的目录，例如 tests 下做测试，则只允许测试公开接口，即使用 `pub` 修饰后的模块和函数。

6.2 集成测试(integration tests)

集成测试用例和源码位于不同的目录，因而源码中的模块和函数，对于集成测试来说完全是外部的，因此，只能调用一部分库暴露的公共API。Cargo 约定集成测试的代码位于项目根路径下的 tests 目录中，Cargo 会将 tests 中的每一个文件当做一个 `crate` 来编译。

只有库 crate 才会向其他 crate 暴露可供调用的函数，因此在 src/main.rs 中定义的函数不能够通过 `extern crate` 的方式导入，所以也不能够被集成测试。

我们将 *plus* 函数移动到 src/lib.rs 中，新建 tests/test_lib.rs，最终的目录结构如下

```
├── Cargo.toml
├── src
│   └── lib.rs
│   └── main.rs
├── tests
    └── test_lib.rs
```

```
// src/lib.rs
pub fn plus(x: i32, y: i32) -> i32 { // plus 必须是公共API，才能被集成测试。
    x + y
}
```

```
// src/main.rs

use tutu;

fn main() {
    println!("2 + 3 = {}", tutu::plus(2, 3));
}
```

```
// tests/test_lib.rs

use tutu;

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

运行 *cargo test* , 将输出

```
running 1 test
test it_adds_two ... ok

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

参考

- Rust 官方指南
- Rust 官方文档
- Cargo 官方文档

专题: Rust 简明教程

本文发表于 2019-11-24，最后修改于 2020-07-20。

本站永久域名「 **geektutu.com** 」，也可搜索「 **极客兔兔** 」找到我。

期待关注我的 「 **知乎专栏** 」 和 「 **微博** 」 ，查看最近的文章和动态。

上一篇 « Go语言动手写Web框架 - Gee第六天 模板(HTML Template)

下一篇 » WSL, Git, Mircosoft Terminal 等常用工具配置

推荐阅读

动手写分布式缓存 - GeeCache第六天 防止缓存击穿

发表于2020-02-16， 阅读约18分钟

动手写分布式缓存 - GeeCache第二天 单机并发缓存

发表于2020-02-12， 阅读约34分钟

博客折腾记(五) - 友链这件事，没那么简单

发表于2019-07-03， 阅读约13分钟

#关于我 (8) #百宝箱 (2) #Cheat Sheet (1) #Go (34) #友链 (1) #Pandas (3) #机器学习 (9) #Rust (1)

#TensorFlow (9) #mnist (5) #Python (9) #强化学习 (3) #OpenAI gym (4) #DQN (1) #Q-Learning (1)

#CNN (1) #TensorFlow 2 (10) #官方文档 (10) #推广 (1)

niqingyang 发表于 7 个月前

这是开始搞 Rust 了？和 golang 比较感觉怎么样？

geektutu 发表于 7 个月前

@niqingyang 这个语言最近比较火，跟着官方教程走了一遍，有个初步的理解。

和 golang 相比的话，两者面向的领域有交集，不过 Rust 可以做得更底层，比如开发一个嵌入式操作系统，golang 就做不到了。Rust 在编译期确保安全，所有权机制是Rust实现安全特性的核心，不过写起来有点反人类。

使用 golang 的场景应该不会用 Rust，比如云计算场景。不过 Rust 取代 C 的领域，比如开发操作系统，内核模块啥的，golang 也做不了。各有优势吧，Rust 的安全特性确实是替代 C 的一个好的切入点。

niqingyang 发表于 7 个月前

> @niqingyang 这个语言最近比较火，跟着官方教程走了一遍，有个初步的理解。
>
> 和 golang 相比的话，两者面向的领域有交集，不过 Rust 可以做得更底层，比如开发一个嵌入式操作系统，golang 就做不到了。Rust 在编译期确保安全，所有权机制是Rust实现安全特性的核心，不过写起来有点反人类。
>
> 使用 golang 的场景应该不会用 Rust，比如云计算场景。不过 Rust 取代 C 的领域，比如开发操作系统，内核模块啥的，golang 也做不了。各有优势吧，Rust 的安全特性确实是替代 C 的一个好的切入点。

又一个反人类语言 ... 还好不是来替换 java、golang 的，否则重度选择困难症患者表示好难啊！

weoking 发表于 4 个月前

> fn summarize(&self) -> String {
> format!("width={}, height={}", self.width, self.height)
> }
> 这里的双引号里面应该要少一对大括号，文中的写法编译器会报错

感觉Rust的语法跟go和python都有相似之处，学起来有种很微妙的感觉，不过代码风格还是挺优美，决定学完后这学期的嵌入式作业就用Rust写

@weoking 感谢指出问题，如果希望输出结果是：`{width=30, height=50}` ，那么需要用 `{{` 表示 `{` ，`}}` 表示 `}` ，修正如下：

```
fn summarize(&self) -> String {
    format!("{{width={}, height={}}}", self.width, self.height)
}
```

weihaipy 发表于 3 个月前

rust很好,只是有的语法确实怪异

Go Mock (gomock)简明教程
1 评论 ● 16天前
ericuni —— 可以实现, gmock 那种第一次调用返回一个值, 第二次调用返回另外一个值吗? ```go

Go语言动手写Web框架 - Gee第一天 http.Handler
6 评论 ● 13天前
maogou —— 博主写的真的是太赞了👍 建议博主把 ```go func (engine *Engine)

Go语言动手写Web框架 - Gee第三天 路由 Router
8 评论 ● 1月前
haochen233 —— Trie树有点懵了，那两个mathch函数和interst、search函数注释能多

博客折腾记(七) - Gitalk Plus
10 评论 ● 2月前
ayuayue —— 我的博客使用了gitalk后每次刷新页面都会产生一个issues,正常是这样吗,还是只有评论的时候才会生成