# 1 引用crate（/kreɪt/）

引用功能模块的步骤：

1. 在项目的Cargo.toml中显示引入某个模块。
2. Rs文件中使用use关键字声明引入库的名字

Cargo build：编译项目，会自动下载引用的库及其依赖的库。

Cargo update：更新依赖库到最新的版本，注意不是跨大版本，如果要跨大版本，需要手动修改toml文件实现，使用该命令会忽略cargo.lock文件，update完成后更新到cargo.lock中。

Cargo.lock：该文件包含了最后一次编译项目时所有引用库及其依赖库的版本等，重新编译时直接引用，除非修改了toml的引用库版本或者使用了update语句手动做了更新。

## 1.1 引用随机数功能块

Crate，从此处看更像是一个功能模块或者说一个库，rust提供了多种多样的crate来实现不同的功能，比如这里要引入的随机数函数库rand，就提供了生成随机数的功能。

首先，在Cargo.toml文件中的[dependencies]节中添加引入语句：

rand = "0.8.3"

然后，在rs文件中添加引入语句：

use rand::Rng;

这样在rs源码中就可以使用rand结构下的方法来生成随机数了，调用如下：

rand::thread\_rng().gen\_range(start..=end)

注释：

1. 直接调用cargo build进行编译，会自动下载并编译rand库所依赖的其他库文件。
2. Thread\_rng是Rand结构体的方法，命名方式符合“方法\_名字”或“功能\_名字”的方式。
3. 注意gen\_range的参数是随机数范围，形式如“start..end”，为左闭右开区间，如果使用左闭右闭区间，形式如“start..=end”。
4. 调用功能库，你不仅要知道功能库的名字，还需要知道在功能库的功能特征，以及调用哪些功能函数，每个功能库的使用说明应该在库所带的文件中，使用命令“cargo doc --open”将会生成整个项目及其依赖库的说明文档，并显示在html中。

## 1.2 引入比较结构功能

比较结构封装在标准库中，标准库不需要再toml中标明引入，引用语句如下：

use std::cmp::Ordering;

Ordering是一个枚举对象，包含Less，Greater，Equal值，调用cmp函数返回这个枚举结果。

调用方法如下：

    match input\_number.cmp(&my\_rand\_number) {

        Ordering::Less => println!("Less"),

        Ordering::Equal => println!("Equal"),

        Ordering::Greater => println!("Less"),

    };

注释：

# 2 数据结构

Rust是一个强类型语言，每个值都必须有类型。

Rust默认数字类型是i32。

Rust默认变量是不可变得。

## 2.1 变量和常量的区别

这里为什么要讨论变量和常量的区别？因为rust默认变量是不可变的，常量也是不可变的，所以有必要说明他们的不同。

相同点，都是将值绑定到一个名称上。

不同点：

1. 变量允许使用mut改成可变的，常量一旦定义就是不可变的。
2. 变量使用let关键字定义，常量使用const关键字定义且必须注明值的类型。
3. 变量需要在作用域中定义，常量没有限值，可以在任何作用域定义。
4. 变量定义为在运行时再计算值得表达式，常量不可以，必须是常量表达式。

## 2.2 数据类型

Rust是一种静态类型语言，也就是编译时必须知道变量的类型。

每个值都由数据类型。

在声明一些变量时，如果表达式不能确定结果是什么类型的，需要指定返回值的类型：

    let guess: i32 = "32".parse().expect("not a number");

    println!("the value of guess = {}", guess);

当然也有在表达式中指定返回数据类型的方式，这里只是为了说明。

### 2.2.1 标量

标量表示单个值，rust中有四种标量，布尔，整形，浮点型，字符类型（不是字符串）。

### **2.2.2 整形**

| **Length** | **Signed** | **Unsigned** |
| --- | --- | --- |
| 8-bit | i8 | u8 |
| 16-bit | i16 | u16 |
| 32-bit | i32 | u32 |
| 64-bit | i64 | u64 |
| 128-bit | i128 | u128 |
| arch | isize | usize |

6中整形，其中arch根据计算机有所不同，64位架构的，就是64位，32位架构的就是32位。

值范围，有符号： -(2n - 1) to 2n - 1 - 1

无符号的：0 to 2n - 1

编写整数的方式：

| **Number literals** | **Example** |
| --- | --- |
| Decimal | 98\_222 |
| Hex | 0xff |
| Octal | 0o77 |
| Binary | 0b1111\_0000 |
| Byte (u8 only) | b'A' |

注意，当你写了一个值如52，默认是i32，如果想指定类型，可以将类型关键字加到值得后面，如52u8，表示单字节无符号整数。

当数字过长的时候，可以使用\_连接以便于阅读，rust可以自动识别。

需要注意的是，如果给变量赋的值超过了变量的范围，就会发生越界，如果debug模式对直接崩溃，release模式会自动计算到变量的值范围内，但是却不是我们想要的值，标准库中有处理这些溢出的方法。

### 2.2.3 浮点型

Rust有两种浮点，分别是f32和f64，代表单精度和双精度浮点数。默认情况下为f64。

### 2.2.4 布尔类型

包含两种布尔值类型，true和false，使用bool关键字指定。

布尔值主要使用在条件判断语句中，如if语句。

### 2.2.5 字符类型

Rust使用char指定字符，单引号表示。单个char占用4个字节，并使用unicode标量值来表示。

当把一个Unicode字符串写入文本文件或者以其他形式存储，字符串中的Unicode标量会被编码成Unicode定义的几种编码形式中的一种。每一种字符串编码形式都把字符串编码成更小块的编码单元。其中有：

        UTF-8编码：把一个字符串以8位的一个编码单元进行编码

        UTF-16编码：把一个字符串以16位的一个编码单元进行编码

        UTF-32编码：把一个字符串以32位的一个编码单元进行编码

Rust默认utf-8。

### 2.4.6 复合类型

符合类型就是将多个值放到一起表示的类型方式。

Rust定义了两种默认的符合类型，数组和元组。

#### 2.4.6.1 元组

元组可以包括多种类型的值，一旦声明，长度固定，大小不能再改变。

声明元组需要指定每个值得类型，否则会使用默认类型。

如：

let \_tup:(i32, u32, String) = (1,2,"kajlsdf".to\_string());

注意，字符串常量默认是&str类型，即字符串切片类型。

元组如何遍历：

方式1：匹配

    let \_tup:(i32, u32, String) = (1,2,"kajlsdf".to\_string());

    let (x,y,z) = \_tup;

    println!("the value of x = {}", x);

    println!("the value of y = {}", y);

    println!("the value of z = {}", z);

方式2：通过.直接访问元组元素

    let \_tup:(i32, u32, String) = (1,2,"kajlsdf".to\_string());

    println!("the value of x = {}", \_tup.0);

    println!("the value of y = {}", \_tup.1);

    println!("the value of z = {}", \_tup.2);

更多的元组方法可以查看文档。

没有任何元素的元组用()表示，表示单位值，单位元组，如果表达式没有任何结果返回，默认返回单位值。

#### 2.4.6.2 数组

数组包含多个值，值的类型必须相同。与其他编程语言不同的是，数组必须要包含一个长度，像元组一样，一旦声明，不允许改变大小。

如果想在栈上分配内存，而不是在堆上，数组和元组都很有用。与vector不同，vector可以改变大小，但是是在堆上。

总之，如果知道固定长度，那么用数组，不知道用向量就好了。

声明数组很有意思，如果需要指定数据数据类型，在声明时指定。

  let \_arr:[i32;6] = [1,2,3,4,5,6];

注意，分号表示结束，分号前是数据类型，后是数组长度，必须初始化，并且元素个数对应。

Rust中向来可以用值来告诉它绑定的变量是什么类型，数组也可以：

    let \_arr = [3;5];

注意分号是结束的意思，前面的3是初始值和值类型，后是元素个数。

数组元素如何遍历：

方式：使用索引，注意与元组的不同点。

    let \_arr = [3;5];

    println!("the value of \_arr\_0 = {}", \_arr[0]);

    println!("the value of \_arr\_1 = {}", \_arr[1]);

数据越界会发生什么：崩溃，除非有错误处理机制。

## 2.3 遮挡

rust中定义了这个新的特征，当在同一个作用域中声明同名变量的时候，前一个变量会被后来声明的变量所遮挡，也就是前一个变量不可用。

fn main() {

    let x = 5;

    let x = x + 1;

    {

        let x = x \* 2;

        println!("the value of x = {}", x);

    }

    println!("the value of x = {}", x);

}

运行结果：

the value of x = 12

the value of x = 6

理解这段代码，涉及到了一个新的特征所有权，let x=5,值5的所有权给了变量x，注意值5只能给一个变量所拥有，也就是说一个值同时只能被一个变量所拥有，这点和C/C++很不同，比如数据地址可以被多个指针变量所拥有；有趣的来了，let x = x + 1 这句话中，x+1这个+会获得5的所有权，计算结束将返回值的所有权交给本行的接收者x，此时值等于6；进入一个大括号，6的所有权会移动到打括号内x，此时第二行的x也没有值6的所有权了，大括号作用域结束时释放了传入的x的所有权，此时值6的所有权又给了第二行的x，所以此时这个x是可用的。

综上，rust计算是通过所有权来限值权限的，所有权是核心，理解所有权很重要，是开发rust程序的基础。

所有权和遮挡是两个不相关的概念。

let x = x + 1这句话，运行完毕后，值5的所有权是释放了的，此时第一句的x是有效的，只是被第二句的x给遮挡了，是无法使用的，我觉得这才是遮挡的真正涵义，同理大括号作用域有效时,let x = x\*2这句中的x会遮挡第二句的x，当作用域结束，其中的x被销毁后，被遮挡的第二句的x又可以被看到了，即又有效了。

遮挡可以不修改变量的不可变属性，通过遮挡机制对现有变量进行一些修改，但是变量仍然是不可变的；

与mut不同，mut修饰的变量可以随时修改，具有不可预知性，还有不同就是遮挡是可以改变变量的类型的，let会生成新的变量，只是变量名与之前的相同。

对编码提供了一定的便利性，不用想太多的变量名。

## 2.4 字符串相关

Rust提供了String结构来处理字符串结构

### 2.4.1 字符串转换为数字

如：

let input\_number:i32 = input\_number.trim().parse().expect("Please type a number");

trim()函数去除字符串前后空格，并去除“\n”或“\r\n”（Windows）。

parse()函数将字符串转换为响应的数值类型，类型由需要绑定的变量类型确定，又因为rust是强类型的，所以变量必须指明类型，如上变量指定为u32 了。

该函数很容易失败，如用户输入了%，那么转换会异常并返回结果对象result，expert函数会处理这个result输出日志，并导致程序崩溃，结束运行。

为了异常不崩溃，我们可以以另外一种方式来处理parse()的返回结果，该函数返回结果是Result枚举对象，该对象包含Ok和Err两个值，所以可以用匹配表达式来处理。

        let input\_number:i32 = match input\_number.trim().parse(){

            Ok(num)=>num,//如果解析成功，这里将成功后结果返回，也就是parse函数的返回结果这里手动进行了处理

            Err(\_)=>{

                println!("请输入数字！");

                continue;//这里跳转到下一次循环

            },

        };

注意分号结束以及格式，match的各个分支用“，”结束。

代码块的每行代码用“；”结束。

## 2.5 函数

Rust应用程序的函数入口点是main。

函数使用关键字fn声明。

Rust中使用蛇形命名法来书写函数或变量名字，蛇形命名法，即所有字符小写，单词用下划线连接。

rust不关心函数在何处定义，只关心它定义在了何处。（main前还是main后）

fn main() {

    println!("Hello, world!");

    other\_function();

}

fn other\_function() {

    println!("Hello, world1!");

}

### 2.5.1 函数参数

函数可以定义形参。

应当注意rust中定义变量或者值的类型的形式，都是在它们的后面，变量使用冒号加类型来指定，常量直接跟着表示类型的关键字。如let v1：i32 = 8; let v1 = 56u32;

同样需要定义形参的类型（形参必须指定类型）。据说这是深思熟虑的结果，因为编译器不知道调用函数的地方传递的是什么类型的参数，当不知道类型的时候，就需要在一个地方，通常是接收数据方，指定类型。

fn main() {

    println!("Hello, world!");

    other\_function(5);

}

fn other\_function(x:i32) {

    println!("the param is {}!", x);

}

定义多个形参：

fn main() {

    println!("Hello, world!");

    other\_function(5, 'a');

}

fn other\_function(x:i32, y:char) {

    println!("the param is {} {}!", x,y);

}

### 2.5.2 函数体包含语句和表达式

函数体，就是函数后面有大括号包含的代码块。

与其他函数不同，语句和表达式在rust中有较大的区别和语义。

语句是执行操作但是不返回值得指令。

表达式产出一个返回值。

例如let x = 5;这是一条语句。

函数的定义本身就是语句。

记住一个关键点，语句不返回值。

    let x = (let y = 100);

所以这样的语句是错误的。

表达式需要有返回值。

表达式可以是语句的一部分。

例如，let x=5;这条语句中，5就是一个表达式。

调用函数是一个表达式。没有指定返回值默认返回一个元组。

   let x = other\_function(5, 'a');

    println!("other\_function is {:?}!",x);

调用宏是一个表达式。

   let x = other\_function(5, 'a');

   let x =  println!("other\_function is {:?}!",x);

   println!("other\_function is {:?}!",x);

大括号可以是一个表达式。

    let x = {

        let y = 100;

        y + 1

    };

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

注意语句块的最后一句是表达式，不包含分号，如果有分号就没有返回值，那么语句块也就没有了返回值，就不是表达式而是语句了，会编译失败。

### 2.5.3 函数的返回值

函数可以向调用他们的代码返回值。

声明返回值类型的方式，是在声明函数的函数体之前使用关键字 ->指定返回值类型，并不指定返回值得名称。

函数的返回值与函数体最后的表达式是同意义的，具有相同的数据类型否则编译失败。

fn main() {

    other\_function();

}

fn other\_function() -> i32 {

    5;

}

如上，函数体没有表达式时默认返回元组，但是指定返回值类型是i32，编译失败。

fn main() {

    other\_function();

}

fn other\_function() -> i32 {

    "skdfasd"

}

如上，也编译失败，指定i32但是返回字符串。

# 3 流控制方式

rust中控制代码执行流的方式有两种，if和循环。

## 3.1 循环

Rust提供了多种循环的控制方式。

### 3.1.1 Loop

    loop {

    }

无限循环在控制台一般可以Ctrl+C发出结束信号来退出，也可以输入非法值导致程

序崩溃退出，这都是非正常的退出方式，还可以输入quit正常退出。

注意loop是一个表达式，如果最后一句没有分号结束，那么视为该表达式最后的输出。

循环可以有标签，这点与C/C++很不同，相应的break时也可以停止指定的循环。

    let mut x:u32 = 1;

    'outer\_loop: loop {

        println!("print {}", x);

        x += 1;

        if x == 30 {

            break 'outer\_loop;

        }

    }

注意标签的声明方式：’标签

Loop可以有返回值，不过它的返回值和if表达式的不同，需要以分号结束。

    let mut x:u32 = 1;

    let x = loop {

        x += 1;

        if x == 30 {

            break x;

        }

    };

### 3.1.2 if语句

If语句允许你根据条件进行分支处理。

与C和C++不同的是：

首先是if语句的书写格式，条件不需要小括号。

    let x = 5;

    if x < 5 {

        println!("the value of x is smaller than 5");

    }

    else if x == 5 {

        println!("the value of x is equal to 5");

    }

    else {

        println!("the value of x is lager than 5");

    }

其次是 条件表达式必须是布尔类型，否则编译会出错，而C/C++可以是一个数值。

适用于比较少的条件判断，否则如果有太多的分支，会造成代码混乱，rust其实提供了更强大的分支选择构造:match。

### 3.1.3 在let语句中使用if表达式

注意，if语句是个表达式。既然是表达式，那么它就应该有返回值。

    let num = if true {1} else {0};

    println!("the value of num is {:?}", num);

If表达式返回是元组，注意if表达式结果如果要绑定到某个变量，那么必须有else语句来处理其他情况，换句话说，就是所有情况都要处理到。

还有if每个分支的结果必须是相同的类型的，否则会编译失败。

以上都好理解。

### 3.1.4 循环

Rust中有三种循环，loop，while和for。

#### 3.1.4.1 while

基本同C/C++，只是需要注意书写格式。

#### 3.1.4.2 For

为了解决while循环无法处理的索引判断问题，提高代码安全性，提供了for循环。

    let arr = [1,2,3,4,5,6];

    for i in arr {

        println!("the value of i is {}", i);

    }

安全性是的for循环称为最常用的循环。

    for i in (2..7).rev() {

        println!("the value of i is {}", i);

    }

    for i in arr.iter().rev() {

        println!("the value of i is {}", i);

    }

数组还有很多的使用方法。

# 4 所有权

所有权是rust中独有的特征，它使得rust可以在没有垃圾收集器的情况下保证内存安全，理解所有权对rust来讲非常重要。

所有权相关内容主要由：借用，切片以及rust如何在内存中存放数据。

## 4.1 什么是所有权

Rust的核心特征是所有权，尽管该特征非常容易解释，但是对该语言其他部分内容来讲有非常深刻的影响。

所有编程语言在运行时都必须管理计算机内容的使用方式。有的语言使用垃圾回收机制，不断搜寻需要回收的内存，有的语言需要程序员负责内存的分配与回收。Rust使用第三种方式，使用一组规则来管理内存，在编译时检查这组规则，这组规则就是所有权的分配规则。

所有权是个新概念，需要时间来适应，一旦掌握了技巧，就很容易开发安全程序。

使用rust需要了解堆和堆栈的概念。

所有存储在堆栈上的数据，必须知道大小，所有未知大小的数据必须存储在堆上。

推送到堆栈，比在堆上处理速度更快，因为栈永远在操作栈顶，对需要通过指针找到操作位置。

一旦理解了所有权，就不需要知道堆栈和堆的原理，但是知道这些有助有开发高效安全的程序。

### 4.1.1 所有权规则

1. 每个值都由其所有者。
2. 一次只能有一个所有者。
3. 当所有者超出范围时，该值被删除。

### 4.1.2 变量作用域

Rust作用域类似其他编程语言，不再赘述。

### 4.1.3 用字符串类型说明所有权

所有所有权为什么要使用String类型，因为栈上的固定大小，容易理解，在堆上的数据如String比较复杂，通过它来说明rust如何在堆上分配内存和回收内存。

Rust分配一个字符串在堆上，有不同的方法，这里用也是很常用的：

    let s = String::from("s: &str");

    println!("the value of s is {}", s);

### 4.1.4 内存分配

在堆上分配内存分为两步：

1. 在运行时从内存分配器上分配足够的内存。
2. 需要一种方式，在分配的内存不再需要时进行回收。

然而，不同的编程语言，第二部分是不同的，有的有垃圾回收机制，有的则需要手动管理，rust采用的是一种当变量超出作用域那么它所所有的内存就会被回收。

    {

        let s = String::from("hello"); // 此句之后变量s有效

        // 操作

    }                                  // 作用域结束, s不再有效，内存被回收

Rust在作用域结束时，自动调用内存释放函数 drop。

C++中有类似的机制：资源获取即初始化。Resource Acquisition Is Initialization (RAII)

### 4.1.4 变量与数据的交互方式之移动

先看固定大小数据的移动：

    let x = 5;

    let y = x;

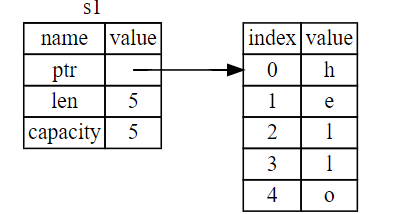
值5的所有权绑定到变量x，然后复制x中的值并将其绑定到变量y。

再看复制数据，如String这种不固定大小的数据的移动：

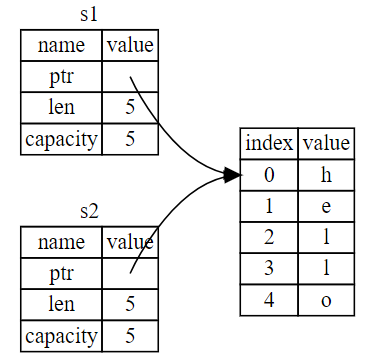
    let s1 = String::from("test");

    let s2 = s1;

如果还是按照固定大小数据的理解方式，s2应该有了同样的一份字符串值，实际上不是这样的：



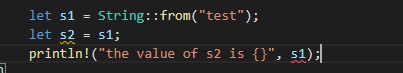
字符串内存模型



拷贝后字符串内存模型

这样，实际的方式是，s1浅拷贝并将值绑定到s2。

这样就引发了一个问题，s1，s2中共有两个指针值相同一块内存，存在潜在的多种释放危险。



Rust为了解决这个问题，规定s1的值绑定到s2后，s1将不再有效，只有有效的变量离开作用域才会释放内存，就不会出现同一块内存被多种释放的问题。

S1绑定到s2发生的是类似其他语言中的浅拷贝概念，但是rust中还附带使s1失效的操作，所以rust称之为移动。

Rust有个原则，就是永远不会自动深拷贝数据。

### 4.1.6 变量与数据的交互方式之克隆

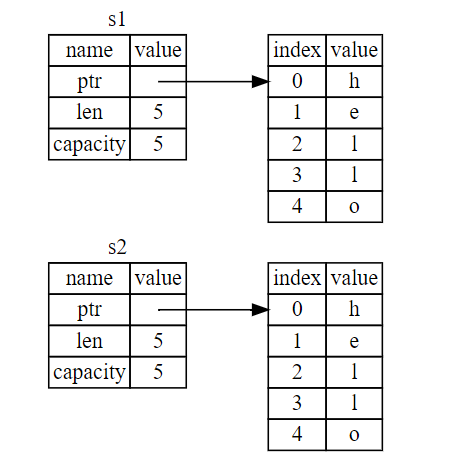
如果趋势需要深拷贝，rust提供了clone方法。

    let s1 = String::from("test");

    let s2 = s1.clone();

    println!("the value of s2 is {}", s1);

    println!("the value of s2 is {}", s2);



### 4.1.7 栈数据之拷贝

只有在栈上的数据才会实现copy函数。

栈上的数据固定大小，不再区分深浅拷贝等。

注意实现了 copy的类是实现不了drop的，同理实现了drop的也实现不了copy，会存在编译时错误，从而避免这中间产生的错误。

那么谁可以实现copy？

一般来讲，标量可以实现copy，任何需要堆上分配内存的变量都不可以实现copy。

特殊的，如元组，如果它的元素全是标量，则可以实现copy。

### 4.1.8 所有权和函数

函数需要传参和返回数据，这时候涉及到移动和拷贝概念。

用一个例子来说明移动和拷贝的区别：

fn main() {

    let s = String::from("hello world");//堆上分配内存

    let x:u32 = 5;//栈上分配内存

    make\_move(s);//s的值移动到形参s上

    make\_copy(x);//x的值拷贝到形参x上

    //println!("s is {}", s);//由于调用make\_move时s的值得所有权移动到了形参上，不会自动移动回来，所以这里编译失败了。

    println!("x is {}",x);//由于x是拷贝，没有所有权的移动，所以x还是有效的

}

fn make\_move(s: String){

    println!("the string s is {}", s);

}//形参的作用域结束，s指向的堆内存释放。

fn make\_copy(x:u32){

    println!("the x is {}", x);

}

### 4.1.9 返回值和作用域

fn main() {

    let s1 = gives\_ownership();//函数内创建一个字符串，并将所有权传出

    println!("the string s1 is {}", s1);

    let s2 = String::from("other string on heap!");

    println!("the string s2 is {}", s2);

    let s3 = gives\_and\_back\_ownership(s2);//函数外部创建的字符串，移动到函数内部，操作完成后将所有权交回

    //println!("the string s2 is {}", s2);//由于所有权已经移动到了s3，s2无效

    println!("the string s3 is {}", s3);//s3是有效的

}

fn gives\_ownership()->String{

    let s = String::from("a string on heap!");

    s

}//形参s的作用域结束，但是它的值所有权被移动到了返回参数，所以字符串不会释放。

fn gives\_and\_back\_ownership(s: String)->String{

    s

}//形参s的作用域结束，但是它的值所有权被移动到了返回参数，所以字符串不会释放。

以上代码说明的所有权的移动过程。

这也带来了一问题，如果我还想用s2怎么办，所有权已经被移走，有没有办法只是让函数使用它而并不剥夺其所有权？

方式1：就像上面的，将所有权传递出来。

方式2：rust支持的另外一个特征，引用。

## 4.2 引用和借用

### 4.2.1 引用

引用符号：&。它允许你使用某个变量，而不占有它的所有权。

fn main() {

    let s1 = String::from("a string on heap!");

    let s1\_len = make\_ref(&s1);

    println!("the  s1 is: {}", s1);

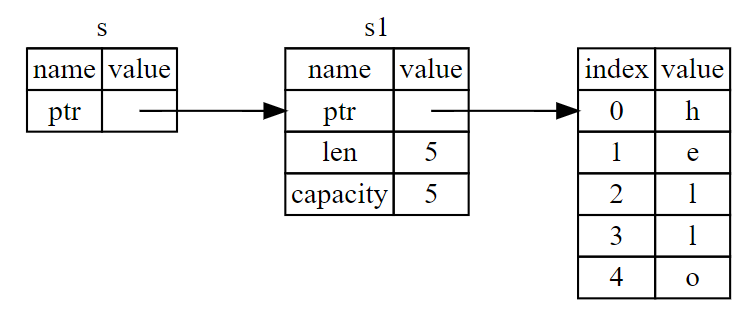
    println!("the len of s1 is: {}", s1\_len);

}

fn make\_ref(s:&String)->usize {

    s.len()

}



与引用相反的解引用，用符号\*表示，这里深入讨论。

变量拥有某个值得所有权，当变量超出它的作用域时，就是释放变量所占用的内存。

如果我们引用的变量是不可变引用，那么不能改变引用它指向的值。

### 4.2.2 可变引用

可变引用使用关键字mut实现。

fn main() {

    let mut s1 = String::from("a string on heap!");

    let s1\_len = make\_ref(&mut s1);//可变引用必须指明，告诉读者函数内部要改这个变量

    println!("the  s1 is: {}", s1);

    println!("the len of s1 is: {}", s1\_len);

}

fn make\_ref(s:&mut String)->usize {

    s.push\_str("xxxx");

    s.len()

}

可变引用是有条件的，同一时间只能有一个对特定片段的可变引用。

当有不可变引用时，不可有可变引用，防止资源冲突（如果可变引用在不可变引用的作用域外，则可以使用）。

   let mut s = String::from("a string on heap!");

    let s1 = &s;

    let s2 = &s;

    println!("the len of s1 is: {}{}", s1, s2);

    let s3 = &mut s;//可变引用在s1，s2被使用之中会编译出错的（非词法声明周期）

    println!("the len of s1 is: {}", s3);

### 4.2.3 悬挂引用

其他语言中有悬挂指针的概念，rust中编译器确保数据在对数据的引用超出范围之前不会超出自己的范围。

出现悬挂引用会触发编译器报错。

fn main() {

    let s = make\_haning();

}

fn make\_haning() -> &String {

    let s = String::from("aaaa");

    &s

}

### 4.2.4 引用规则

1. 任何时刻，你可以有一个可变引用，或可以有多个不可变引用。
2. 引用必须始终有效。

## 4.3 切片引用

切片是一种引用类型，切片的目的是使用集合中的某一段数据，而不是使用整个集合。

这里写个例子：

查找字符串中某个字符，如果存在则返回字符的索引，如果不存在则返回字符串长度。

分析：字符串时对char类型的封装，相当于字符数组，因此可以用索引来表示字符串中的字符位置。

fn main() {

    let s = String::from("hello world");

    println!("s is {}", s);

    let pos = get\_char\_pos(&s);

    println!("pos is {}", pos);

}

fn get\_char\_pos(s:&String) -> usize {

    let s\_bytes = s.as\_bytes();//这里使用字节设计自来表示字符串

    for (i,&c) in s\_bytes.iter().enumerate() {//这里使用了引用来表示某个字节的值因为enumerate返回的是元素的引用。

        if c == b' ' {

            return i;

        }

    }

    s.len()

}

这个例子中存在的问题：

Pos这个值在s有效之前是有效的，但是s如果无效呢？那么它就是无效的了，虽然它还保存原来的值，究其原因就是pos和s是不相关的两个变量。

通过代码逻辑来确保这种事情不发生是很烧脑的，rust提供了一种类型来解决这种数据分离现象：字符串切片。

### 4.3.1 字符串切片

    let s1 = &s[0..3];

    let s2 = &s[3..s.len()];

    println!("s1 is \"{}\", and s2 is \"{}\"", s1, s2);

字符串切片用的符号形式为：&str。

fn get\_char\_slice(s:&String) -> &str {

    let s\_bytes = s.as\_bytes();//这里使用字节数组来表示字符串

    for (i,&c) in s\_bytes.iter().enumerate() {//这里使用了引用来表示某个字节的值

        if c == b' ' {

            return &s[..i];

        }

    }

    &s[..]

}

使用切片后，编译器会确保对字符串的引用有效，所以不允许调用clear()函数清空字符串，如果调用了会报编译错误。

回想引用的规则，如果我们有个不可变得引用，那么可变引用将会被拒绝，调用clear函数需要清空字符串，所以它需要获得一个可变引用，编译器会阻止这种操作。

### 4.3.2 字符串字面量是切片类型

    let s = "hello world";

这里的s是一个切片类型，即&str，是一个特殊的指向二进制区域的切片，这也是字符串不可变的原因，&str是一个不可变引用。

### 4.3.3 字符串切片作为参数

定义函数的时候使用字符串切片，而不是使用对string的引用，是的我们的api在不损失功能的情况下更加的灵活和有用。

fn get\_char\_slice\_by\_slice(s:&str) -> &str {

    let s\_bytes = s.as\_bytes();//这里使用字节数组来表示字符串

    for (i,&c) in s\_bytes.iter().enumerate() {//这里使用了引用来表示某个字节的值

        if c == b' ' {

            return &s[..i];

        }

    }

    &s[..]

}

## 4.4 其他切片类型

对任何集合都可以使用切片，只是与字符串切片特有的&str有所不同。

    let arr = [1,2,3,4,5,6,7,8,9,10,11,12,13];

    let arr\_s1 = &arr[..6];

    println!("the arr is {:?}",arr);

    println!("the arr\_s1 is {:?}",arr\_s1);

所有权，借用（引用）和切片的概念，确保了你编写内存安全的程序。

# 5 结构体

结构体是一种自定义数据类型，将多种类型的数据集合到一起。

## 5.1 定义和实例化结构体

结构体类似于元组，都可以存储不同类型的数据。

不同之处是，结构体可以给各个数据块命名，这样就可以不依赖元素顺序来访问各个数据块。

关键词struct定义结构体，rust中结构体的成员称为字段，字段定义：”名字：类型”。

Struct创建之后需要使用，使用时需要指定每个字段的值来实例化。

struct User {

    name: String,

    addr: String,

    age: u8,

}

fn main() {

    let st = User {

        name: String::from("张三"),

        addr: String::from("北京"),

        age: 22,

    };

    println!("Name {}, Addr {}, Age {}", st.name, st.addr, st.age);

}

字段名字使用蛇形命名，否则会警告。

使用时需要实例化，每个成员都是赋值，否则报错。

打印时，不能使用{:?}格式，因为没有实现内置debug的某个trait。

字符串放到堆上时使用内置函数，不能硬编码。

如果要修改成员怎么办，将结构体实例声明为可变类型，let mut st...。

整个实例必须是可变的，不允许只将某个字段设置为可变。

使用函数初始化实例时候，函数参与字段名字一致，可以简写初始化代码。

fn get\_user\_instance(name: String, addr: String, age: u8) -> User {

    User {

        name,

        addr,

        age,

    }

}

使用结构更新语法，从其他实例更新实例。

普通做法，将一个实例的字段，赋值给另一个实例的字段。

    let st1 = User {

        name: String::from("王五"),

        addr: st.addr,

        age: st.age,

    };

升级做法：更改字段按需更改其余字段使用..批量赋值，且不用管字段顺序。

    let st1 = User {

        name: String::from("王五五"),

        ..st1

    };

注意：具有移动特性的字段，赋值之后之前的实例该字段不可用，因为移动到了新的实例。在最后例中原来str1的name和addr字段不可用了，因为移动给了新的st1，但是标量类型（实现的是copy）是可用的。

如何使用元组来实现结构体。

为什么要用元组结构体？当不需要字段名的时候。

元组结构体的字段具有元组结构体名称提供的附加含义。

定义方式，struct关键字表示，名称后面使用元组形式表示。

    struct Color(u8,u8,u8);

    struct Point(u8,u8);

    let c1 = Color(0,0,0);

    let p1 = Point(1,1);

元组结构可以想访问元组那样访问。

没有任何字段的单元结构。

当你需要在某些类型上实现一个trait，但又没有任何数据想要存储在类型本身中时候。

    struct Student;

结构体数据的所有权。

## 5.2 结构体应用实例

略

注意：如果我们直接打印结构，或者使用：？打印结构的内容，会提示没有实现display或debug的trait，rust内置了这种trait，但是自定义的结构想使用的话，需要在定义结构时，显式的集成这些特性。

使用debug格式打印自定义结构，还可以使用dbg!宏，打印是宏获得结构实例的所有权，打印完成返回所有权。

fn main() {

    let widgth = 10;

    let heigth = 10;

    println!("rectangle is {}", area(widgth, heigth));

    //let tuple:(i32,i32) = (10,10);

    let tuple = (10,11);

    println!("rectangle is {}", area1(tuple));

    println!("rectangle is {}", tuple.0 \* tuple.1);

    let tuple = (String::from("张三"),11);

    {

        let tuple1 = tuple;

        println!("user is {:?}",tuple1);

        println!("user is {}:{}",tuple1.0,tuple1.1);

    }

    //println!("user is {}:{}",tuple.0,tuple.1);//tuple已经移动到了tuple1，所以这里不可用

    let tuple = (String::from("张三"),11);

    area3(tuple);//这里由于元组元素不是标量，所以执行的不是copy，而是move，如果不想拿走tuple的所有权，可以用切片引用或叫借用

    //println!("user is {}:{}",tuple.0,tuple.1);//由于上一句代码是对元组是move，所以这里元组已经不可再调用

    let st1 = Area {

        widgth: 11,

        heigth: 11,

    };

    println!("rectangle is {}",area2(&st1));

    println!("rectangle is {}",st1.widgth \* st1.heigth);

    println!("rectangle is {}",st1.area());

    println!("rectangle is {:?}",st1);

    let r1 = Area::square(13);

    println!("rectangle is {:?}",r1);

    dbg!(st1);

}

fn area(widgth: i32, heigth: i32) -> i32 {

    widgth \* heigth

}

fn area1(tuple: (i32,i32)) -> i32 {

    tuple.0 \* tuple.1

}

#[derive(Debug)]

struct Area {

    widgth: i32,

    heigth: i32,

}

impl Area {

    //定义方法

    fn area(&self)->i32 {

        self.widgth \* self.heigth

    }

    //定义不需要实例自身的方法,实例是点不出来的，只能用::调用

    fn square(size: i32) -> Area {

        Area{

            widgth: size,

            heigth: size}

    }

}

fn area2(rectangle: &Area) -> i32 {

    rectangle.widgth \* rectangle.heigth

}

fn area3(tuple: (String,i32)){

    println!("user is {}:{}",tuple.0,tuple.1);

}

# 6 枚举与模式匹配

枚举类型：使代码表达变得更简洁合适，并不是不可替代。

## 6.1 定义枚举

举例：任何IP地址都有4和6两个版本，但是不能同时是这两个IP，可以用枚举来表达。

//定义一个枚举,具有两个枚举值V4和V6

enum IpAddrKind {

    V4,

    V6,

}

## 6.2 枚举值

//定义一个枚举,具有两个枚举值V4和V6

#[derive(Debug)]

enum IpAddrKind {

    V4,

    V6,

}

//借助结构体和枚举，表达IPV4和6

#[derive(Debug)]

struct IpAddr{

    kind: IpAddrKind,

    address: String,

}

fn main() {

    let four = IpAddrKind::V4;//枚举类型，值为V4

    let six = IpAddrKind::V6;//枚举类型，值为V6

    println!("{:?}",four);

    println!("{:?}",six);

    let ip1 = IpAddr {

        kind: IpAddrKind::V4,

        address: String::from("127.0.0.1"),

    };

    let ip2 = IpAddr {

        kind: IpAddrKind::V6,

        address: String::from("::1"),

    };

    println!("{:?}",ip1);

    println!("{:?}",ip2);

}

使用结构和枚举的方式表达ipv4和ipv6。

其实还有更为简便的方法：枚举值也可以定义类型和值，类似于C/C++的枚举，也有类型和值。

    enum IpAddrNew{

        V4(String),

        V6(String),

    }

注意，枚举枚举类型的方式与定义结构体的不同之处。

#[derive(Debug)]

enum IpAddrNew{

        V4(String),

        V6(String),

    }

    let ip1 = IpAddrNew::V4(String::from("127.0.0.1"));

    let ip2 = IpAddrNew::V6(String::from("::1"));

    println!("{:?}",ip1);

    println!("{:?}",ip2);

我们也可以获得IP1和IP2，这里可以看到，枚举的字段名其实就是构造枚举实例的函数，上例中，V4传入一个字符串，返回一个枚举实例。

使用枚举而不使用结构的一个原因就是枚举封装更紧密，使用能够更灵活。

#[derive(Debug)]

enum IpAddTest2 {

        Ip1(u8,u8,u8,u8),

        Ip2(String),

    }

    let ip1 = IpAddTest2::Ip1(1,2,2,2);

    let ip2 = IpAddTest2::Ip2(String::from("127.0.0.1"));

    println!("{:?}",ip1);

    println!("{:?}",ip2);

枚举可以像结构体那样，定义方法。

fn main() {

enum Message {

Quit,

Move { x: i32, y: i32 },

Write(String),

ChangeColor(i32, i32, i32),

}

impl Message {

fn call(&self) {

// method body would be defined here

}

}

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

m.call();

}

## 6.3 Option枚举和Null值

Option枚举是一个标准库定义的枚举。

包含null值， 可以帮助检查你是否处理所有应该处理的情况，防止常见的bug。

Rust中没有定义null，但是option提供两个选择，null和非null。

空值是普遍存在的，如果你使用空值代表了你认为的非空值，那么就会出现bug。

但是null所表达的概念，是一个非常有用的概念，表示当前值无效。

Rust表示问题不在概念本身，而在实现方式，rust定义了Option枚举来使用null，并且规避了使用null造成的bug。

    enum Option<T>

    {

        None,

        Some(T),

    }

注意：Option枚举非常重要，默认已经引用到了核心库中，不需要写Option而只要使用None和Some(T)就可以获得整个枚举实例。

枚举和结构体内的字段，定义的字段名字使用驼峰命名法，函数名或变量名使用蛇形命名法。

枚举是一种类型，不能直接使用Some（T）中的T值，可以通过匹配控制流来使用。

## 6.4 匹配控制流操作符

Match是一个很强大的控制流操作符，它允许你将一个值与匹配的模型作比较，并执行相应的代码。

Match的能力来自于各种模式的表达能力，以及能够让编译器确认所有可能的情况都已经处理。

### 6.4.1 绑定到值得模式

enum Manny {

    Yuan(u32),

    Jiao(u32),

    Fen(u32),

}

fn match\_in\_manny(manny: Manny) -> u32{

    match manny {

        Manny::Yuan(manny) => {

            println!("Yuan arm excute!");

            manny

        }

        Manny::Jiao(manny) => {

            println!("Jiao arm excute!");

            manny

        }

        Manny::Fen(manny) => {

            println!("Fen arm excute!");

            manny

        }

    }

}

这是从枚举中提取值得方式。

### 6.4.2 Option<T> 匹配

fn plus\_one(x: Option<u32>)->u32 {

    match x{

        Some(i)=>{

            i

        }

        None => 0,

    }

}

    let xxxx = Some(5);

    let y = plus\_one(xxxx);

    println!("{:?}", y);

    println!("{:?}", xxxx);//这里为何不是移动后借用？？？

### 6.4.3 完全匹配

Rust的match检测，会提示检测所有的枚举项。

### 6.4.4. 完全匹配和下划线占位符

如果枚举有很多项怎么办，我们只希望处理其中的几项，这里有两个关键字other和\_，其中other表示虽然处理所有的，但是值还是需要使用的，如果其他项的值都不需要使用，可以直接使用\_处理。

    let dice\_roll = 9;

    match dice\_roll {

        3 => add\_fancy\_hat(),

        7 => remove\_fancy\_hat(),

        //other => move\_fancy\_hat(other),//other值还使用

        //\_ => (),//其他值都不管了

        \_ => reroll(),//其他值都调用reroll()

    }

## 6.5 简化匹配控制流

If let 语法，允许配置一个值，而忽略其他值。

   let num = Some(100);

    if let Some(x) = num {

        println!("value is {}",x);//只处理有值得情况，None值不管

    };//注意rust中有curly brakets就可以忽略分号

    //如果不匹配将不会做任何执行。

使用场景就是只匹配某个值，如果匹配失败也不管的状态。

如果要处理多个值，还是老老实实用match语法。

但是，既然有if就会有else，可以用if esle这种方式处理其他情况。

    let num = Some(123);

    if let Some(x) = num {

        println!("value is {}",x);//只处理有值得情况，None值不管

    }

    else {

        println!("value is mismatch");

    }

# 7 工程文件管理

当工程变得越来越大，工程文件管理显得尤为重要，可以帮助用户找到哪里定义特性，哪里可以修改特性。

到现在，我们编写的代码都是在一个文件中的一个模块中。随着代码量的增长，可以将代码分为多个模块或多个文件来组织。

一个包可以包含多个二进制包，也可以包含一个库包，随着包的增长，可以将部分提取到单独的crate中，称为外部的依赖项。

这也是代码重用的重要方式。

* **Packages:** A Cargo feature that lets you build, test, and share crates
* **Crates:** A tree of modules that produces a library or executable
* **Modules** and **use:** Let you control the organization, scope, and privacy of paths
* **Paths:** A way of naming an item, such as a struct, function, or module

# 7.1 功能包和功能箱

Crate是二进制文件或者库。

Crate的根是一个源文件，rust编译器从它开始编译（类似C++程序的main入口函数），package是一个或者多个crate，一个包包含一个cargo.toml，包含如何编译crate的内容。

一个package只能包含一个二进制库（指的是它生成的），但是可以包含多个二进制文件，最少应该包含一个crate（不管是二进制文件还是库）。

当我们执行cargo new xxx命令是，rust生成一个名为xxx的package，包含一个cargo.toml文件，内容中没有提到main.rs文件，因为rust约定了，在src/main.rs中的就是crate的根文件。

同样的，cargo同样知道src/lib.rs文件的位置，也就是库的跟文件。

Cargo自动将main.rs或者lib.rs传递给rustc编译器进行编译。

默认生成xxx中只包含一个main.rs文件，表示它是只包含一个名为xxx的crate二进制文件，如果package中的src下包含一个main.rs和一个lib.rs，表示它包含两个crate：一个二进制和一个库文件，都是和package同名的文件。一个package可以在src/bin下拥有多个二进制crate，只需要将文件放到里面就可以了，每个文件都是一个独立的crate。

Crate将相关的功能封装到一个范围内，其他项目通过引用就可以使用相关的功能，比如前面使用道德rand crate，提供了产生随机数的功能。

Rust的特性本地实现机制，不会让rustc混淆是本地定义的功能还是从其他crate中引用的功能。

## 7.2 定义模块来控制功能的私有和公有

涉及关键字，use，mod，pub，as等。

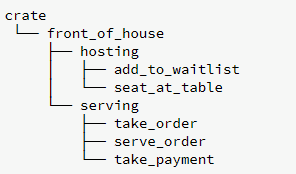
模块可以是一个功能箱中的功能分组，便于阅读和重用。

模块还可以控制功能项的私有和公有。

例如我们编写一个餐馆功能箱：

餐馆包含两个模块，一个是前台，一个是后厨。

Src/main.rs和src/lib.rs之所以别称为功能箱的跟文件，是因为这两文件都能生成一个名为crate的模块，该模块位于功能箱模块树的根部，如下：



该树显示了功能箱的模块树是如何嵌套的，其中crate不是我们定义的，而是默认生成的根模块。

## 7.3 使用路径来引用模块树中的模块

如果我们想引用一个函数，需要知道引用路径，一个package有两种方式来实现：

绝对路径：从功能箱的根crate模块开始一直引用到需要的模块。

相对路径：使用super或者self从当前模块开始引用到所需的模块。

两种方式都可以通过双冒号：：连接的多个路径进行引用。