学习“菜鸟教程”的rust教程: https://www.runoob.com/rust/rust-tutorial.html

编译运行环境直接在web页面[https://play.rust-lang.org/](https://play.rust-lang.org/" \t "https://www.runoob.com/rust/_blank)。

# 第一个 Rust 程序

fn main(){

print!("12345");

}

# 注释

与c/c++， java相同

## 单行注释//

## 多行注释/\*.......\*/

# println!的参数

print!-----打印，不在尾部添加回车

println!---打印，并且在尾部添加回车

fn main(){

print!("{},{}, {0} \r\n{2},{1}\t{{}}", 1,2,3);

}

## 最简单参数“{}”

## 下标参数{0}

## 转义符

格式字符串中通过 **{{** 和 **}}** 分别转义代表 { 和 }。但是其他常用转义字符与 C 语言里的转义字符一样，都是反斜杠开头的形式。

# 变量

常量：const a:u32 = 123;

定义"不可变变量"：let b = 123;

名义变量：let mut c = 34;

赋值：c = 456;

重绑定：let a =1;

下面的例子， 竟然不报错，奇怪

fn main(){

println!("{},{}, {0} \r\n{2},{1}\t{{}}", 1,2,3);

let mut a =10;

println!("{}",a);

a=2;

println!("{}",a);

let mut a =1;

println!("{}",a);

a = 20;

println!("{}",a);

}

# Rust 基本数据类型

## 整数型（Integer）

整数型简称整型，按照比特位长度和有无符号分为一下种类：

|  |  |  |
| --- | --- | --- |
| **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 |

1. 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
2. 转换

let x:i32 =123;

let z:i64 = x as i64;

## **浮点数型（Floating-Point）**

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。

Rust 不支持 **++** 和 **--**，因为这两个运算符出现在变量的前后会影响代码可读性，减弱了开发者对变量改变的意识能力

但是rust支持+=：

许多运算符号之后加上 = 号是自运算的意思，例如：

**sum += 1** 等同于 **sum = sum + 1**。

## **布尔型**

布尔型用 bool 表示，值只能为 true 或 false。

## **字符型**

字符型用 char 表示。

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。

Unicode 值的范围从 U+0000 到 U+D7FF 和 U+E000 到 U+10FFFF （包括两端）。 但是，"字符"这个概念并不存在于 Unicode 中，因此您对"字符"是什么的直觉可能与Rust中的字符概念不匹配。所以一般推荐使用字符串储存 UTF-8 文字（非英文字符尽可能地出现在字符串中）。

**注意：**由于中文文字编码有两种（GBK 和 UTF-8），所以编程中使用中文字符串有可能导致乱码的出现，这是因为源程序与命令行的文字编码不一致，所以在 Rust 中字符串和字符都必须使用 UTF-8 编码，否则编译器会报错。

# **复合类型**

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*

# 数组

## 用逗号隔开，表示多个元素

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*

## 用分号隔的2个表达式

## 开前一个表示“单个元素”，后一个表示“元素个数”

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*

# **函数**

fn <函数名> ( <参数> ) <函数体>

fn <函数名> ( <参数> )-->返回值类型 <函数体>

其中 Rust 函数名称的命名风格是小写字母以下划线分割：

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);  
}

1. 我们在源代码中的 main 函数之后定义了another\_function。 Rust不在乎您在何处定义函数，只需在某个地方定义它们即可
2. 参数定义： 参数名:类型
3. 返回值：-->

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。这样做的目的是为了让公开的函数能够形成可见的公报。

1. 函数体中，可以使用表达式（可以不使用返回值）

fn five() -> i32 {  
        5  
    }

# **条件语句**

## if

1. 条件表达式 number < 5 不需要用小括号包括（注意，不需要不是不允许）
2. Rust 中的 if 必须是有成对{}：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
3. 支持else if
4. If或else if中条件必须为boolean（不容许为整数）
5. 类似C的三元问好表达式

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

两个函数体表达式的类型必须一样！且必须有一个 else 及其后的表达式块

# **循环**

## while循环

Let mut c:i32 = 1000;

While(c > 1){

c -= 1;

}

## 不支持do-while

## 不支持类似c语言的三元组for循环

C语言，for三元组： 初始化，判断，修改变量

## for循环

1. 遍历数组中的每个元素

let a = [10, 20, 30, 40, 50];  
    for i in a.iter() {  
        println!("值为 : {}", i);  
    }

1. 遍历整数区间

for i in 0..5 {

}

其中 0..5就是一个整数区间

## loop循环

loop {  
        if xxxxx {  
            break;  
        }  
     }

## 几种循环方式的比较

与while比， loop将结束循环的判断放在了结构体中，而while将是否循环的条件放在结构体外的专门一块代码中

For循环， 是自己专门代码控制， 遍历的范围， 记录循环的变化，以及判断是否继续循环。它只支持在某个区间做遍历

# **所有权**

## 所有权3原则

每个值都有一个所有者（变量）

每个值最多只能有一个所有者（变量）

当所有者不在有效范围后，值将被删除释放

## 变量的范围

从声明变量开始有效直到变量“所在域”（对应的大括号结束而）结束

## 基本类型，元祖， 以及“类型，元祖”的数组，可以同时被多个变量引用因为他们不是堆内存；对于堆内存， 必须只有一个owner

正确的代码

let var = 123;

let x = var;

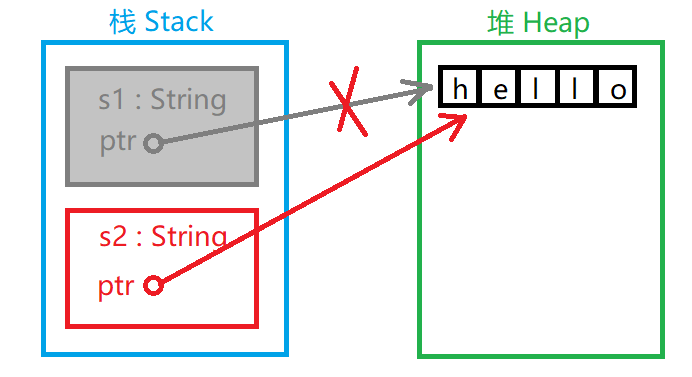
println!("{},{}", var , x);

错误的代码

let s1 = String::from("hello");

let s2 = s1;

println!("{},{}", s1 , s2);

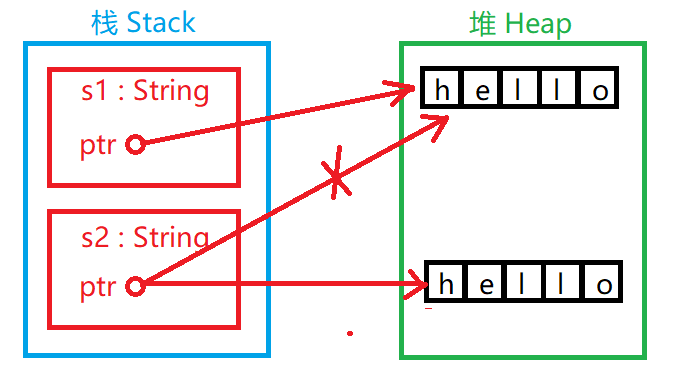


## 克隆

修改为如下就可以：

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

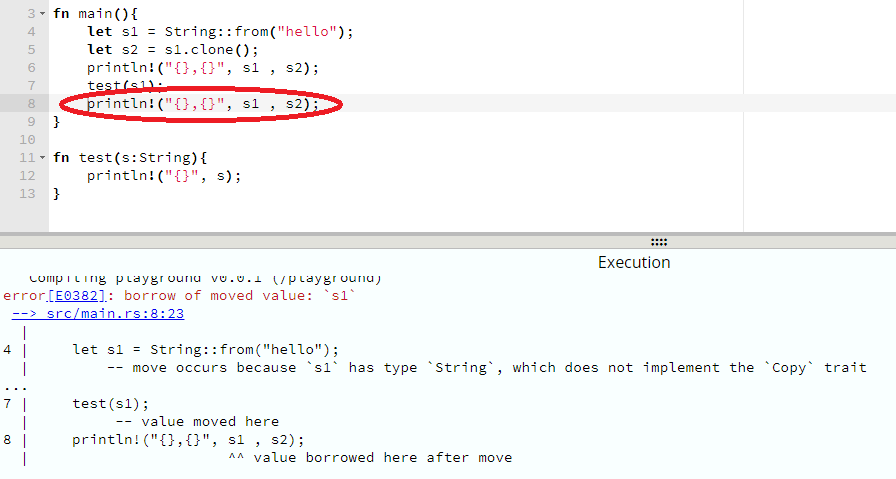
此时有2个对象， S1,S2各自指向自己的对象。 不过s2是从是s1克隆来的， s2的内存地址与s1不同， 但是s2地址的内容与s1地址的内容是相同的



## 函数参数的所有权

let s = String::from("hello");

takes\_ownership(s);//从这里开始，s已经被移动了。此后s不再有效



## 函数返回值的所有权

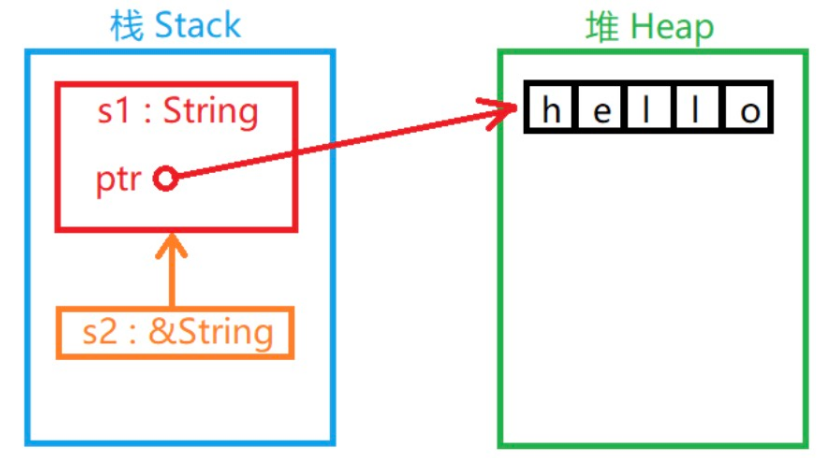
被当做返回值的变量， 会将其所有的对象的所有权移到调用函数的地方， 不会在函数结束时被删除

## 引用与借用

### 什么叫“引用”

实质上"引用"是变量的间接访问方式。

let s1 = String::from("hello");  
    let s2 = &s1;//这个就叫引用，s2持有的是s1



let s1 = String::from("hello");  
    let s2 = &s1;  
    let s3 = s1;//s1被s2借用， 所以其所有权不容许被转移  
    println!("{}", s2);//因为s1的所有权被转移，s1无效。 所以引用s1的时也变得无效

### 引用不能获得所有权， 不能修改引用变量的值

### 除非明确让渡修改权

let s2 = &mut s1;

让渡使用权后， s1不能被修改， 不能被读取

fn main(){

let mut s1 = String::from("hello");

{

let s2 = &mut s1;

//s1.push\_str("h1"); //s1已经被&mut引用（让渡所有权的引用），所以不能修改s1

s2.push\_str("h2");

//println!("{}", s1);//s1已经被&mut引用（让渡所有权的引用），所以不能读取s1

println!("{}" ,s2);

}

s1.push\_str("h4"); //s2已经超出作用域，s2已经被删除，所以s1的被引用已经结束。所以s又没有被出让所有权。s1重新可以被修改， 被读取

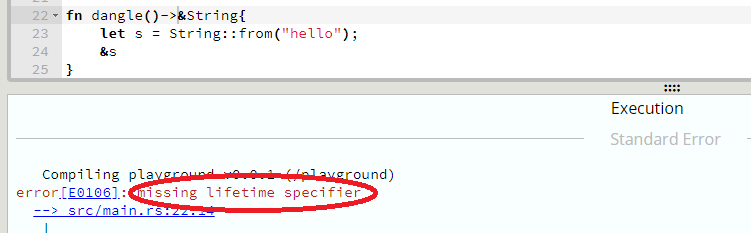
println!("{}", s1);

}

### 禁止“悬垂引用”

典型的是：返回值不能是函数被变量的引用

fn dangle() -> &String {  
    let s = String::from("hello");  
    &s  
}



# **完**