Rust 编程精讲

视频课地址：https://ke.qq.com/course/2992548?tuin=a71606

目录

Rust 编程精讲 1
Rust 简介 2
Rust 语言的特点： 2
Java, GO, Rust 等对比 3
企业使用 Rust 案例与 Rust 招聘 4
RUST 开发环境安装 6
Hello world-调试 8
Rust 基础语法-变量与基本类型 9
Rust 运算符 10
语句 11
函数 13
结构体 15
枚举 16
模块 19
模块拆分 20
集合 24
错误处理 27
文件 IO 30
各编程语言基础语法对比 30
泛型编程 30
Trait 编程 32
动态大小类型（dynamically sized types）DST 37
面向对象 38
所有权 41
生命周期 46
函数式编程 52
智能指针 56
并发编程 66
不安全 rust 71
网络编程 75
数据库编程 76
json 支持 77
redis 78
实战篇-web 服务器 80
实战篇-web 服务器（1）单线程版本 80

实战篇-web 服务器（2）多线程版本.....81
实战篇-web 服务器（3）线程池版本.....81

Rust 简介

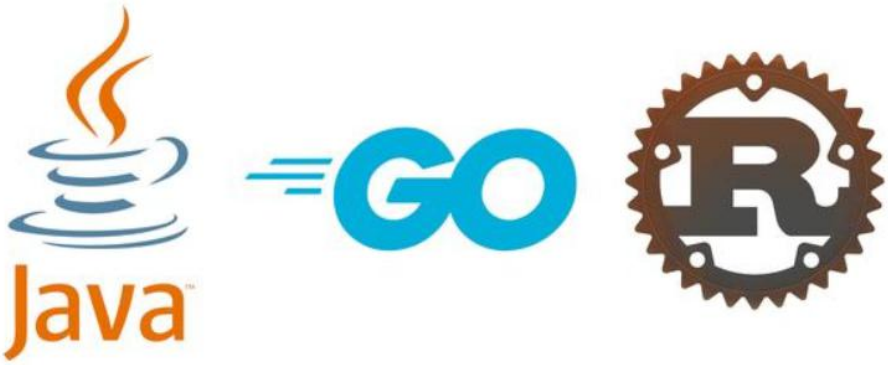
Mozilla 基金会：
Rust 语言由 Mozilla（网景公司员工 2003 年成立的基金会，主要产品包括 Mozilla Firefox 和 Mozilla Thunderbird） 开发，最早发布于 2014 年 9 月（GO2007，C1972，C++1983，JAVA1995）。Rust 的编译器是在 MIT License 和 Apache License 2.0 双重协议声明下的免费开源软件。截至目前(2020 年 1 月)最新的编译器版本是 1.41.0

Rust 语言的特点

高性能：Rust 速度惊人且内存利用率极高。由于极少的运行时（程序能正常运行所必需的基础代码，比如标准库不可或缺部分，解释器，GC 等。常见的有：C 的 glibc，msvcrt.dll 等被应用程序静态链接或者动态链接，.net CLR,Java JRE,Python Cpython）和没有垃圾回收，它能够胜任对性能要求特别高的服务，可以在嵌入式设备上运行，还能轻松和其他语言集成。
安全性： Rust 丰富的类型系统和所有权模型保证了内存安全和线程安全，在编译期就能够消除各种各样的错误，比如数组的溢出。
命令行工具，Web 服务，嵌入式，系统程序等
学习曲线：较高，所有权，生命周期等内容有一定的难度，但难度不会超过 C、C++。学过 C 语言，学习 RUST 更容易。
是 GO，C 和 C++有力竞争者
参考资源：
https://www.rust-lang.org/zh-CN
https://doc.rust-lang.org/std/index.html
《Rust 权威指南》 电子工业出版社 Steve&Carol
学习建议：至少已经学过一门编程语言，比如 C，C++，Java 等

Java, GO, Rust 等对比

Java 和 Go 都是支持垃圾收集(GC)的语言，具有 STW(停止世界) 的副作用
Java 比 Go 或 Rust 消耗更多的内存，相关测试里，Java 使用的内存高达 8000%
在相关的测试里，Go 应用程序使用的 CPU 比 Java 少 20％，同时处理比 java 版多出 38％的请求。另一方面，Rust 版本使用的 CPU 比 Go 减少了 57％，而处理的请求却增加了 13％
Go 是 Java 的直接竞争对手；而 Rust 是 Java，Go，C 和 C ++的重要竞争对手
Go 和 Rust 都非常适合 Web 服务，CLI（command-line interface，命令行程序），系统程序开发。但是，Rust 比 Go 具有根本优势。它不是垃圾回收语言，与 C 和 C++相比，它可以安全地编写代码。
Go 并不特别适合编写 OS 内核，而这是 Rust 的强项，并与 C/C ++竞争，因为它们是使用 OS 编写的长期存在和事实上的语言。Rust 与 C/C++竞争的另一种方式在嵌入式领域
GO 语法少，学习难度低；Rust 所有权，借用，生命周期提高了学习曲线
Linux 之父非常担忧没人继续维护内核："真的很难找到维护者！" Linux 内核大部分是由 C 语言编写的。为此，Hohndel 问道："C 语言是否会被 GO 及 Rust 语言取代，我们这些用 C 的人有没有可能在二十年后变得像现在的 COBOL 程序员一样？" Torvalds 的回应是，"C 语言目前仍是全球十大人气语言之一。但对语言的具体选择并不会对内核造成太大的影响。与驱动程序相似，内核团队正在研究多种语言接口，相信不久之后就能实现。总而言之，我们必将使用不同的模型编写 Linux 代码，而 C 绝对不会是其中唯一的模型选项。"



企业使用 Rust 案例与 Rust 招聘

AWS 使用 Rust。Facebook 已开始使用 Rust，苹果、谷歌、Dropbox 和 Cloudflare 也已开始使用它。

“C++就其核心而言不是一种安全的语言”，微软的云开发者代言人 Ryan Levick 在 AllThingsOpen 虚拟会议上表示，他解释了为何微软正逐渐从 C/C++改用 Rust 来构建其基础设施软件。微软还鼓励其他软件行业巨头考虑采取同样的做法。“我们使用的语言很古老，来自不同的年代，无法为我们提供防范种种漏洞的能力。C++不是一种内存安全的语言，没人会真的假装它是内存安全的语言。”来自微软的通用漏洞披露（CVE）中 70%是内存安全问题。他说：“几乎一直是这种情况，并没有减少的趋势。尽管我们付出了巨大的努力来解决这个问题，但这仍似乎司空见惯。”微软如此钟情 Rust 的主要原因是，它是一种内存安全语言，只有极少的运行时检查。Rust 擅长构建正确的程序。内存中不安全的部分也在 API 后面被封装起来。

Levick 表示，安全编程的这种能力不应该被轻视。实际上，它可以带来逾 10 倍的提升，因此值得投入心血。这主要是由于几乎所有的 C/C++代码都需要接受安全审计、查找有无不安全行为，而用 Rust 编写的需要接受检查的不安全代码只是大多数代码库的一小部分。

PingCAP （TiDB 开源分布式关系型数据库）用 Rust ，字节跳动也一直在招 Rust 的人，阿里巴巴前一段上线了一个用 Rust 写的时序数据库替代 InfluxDB，“系统消耗对比低好几倍以上，另外查询性能也有很大改善”。B 站和知乎的后端组件里也有用 Rust 实现的

Mozilla Firefox 浏览器，Gecko 浏览器；Tor 浏览器；

Discord（一款游戏聊天应用）；

GNOME Fractal（一款即时消息客户端）；

谷歌 Fuchsia 操作系统；

Redox OS；

Rocket Web Framework；

Servo（一款由 Mozilla 开发的网页浏览器排版引擎）；

Terminus DB（一款数据库软件）；

Libra（数字货币）；

OpenDNS

Dropbox

Cloudflare

字节跳动招聘

Rust/C++ 工程师

25k-50k /北京 / 经验3-5年 / 本科及以上 / 全职

后端开发

1天前 发布于拉勾网

职位诱惑：

六险一金，高薪期权，弹性工作，免费三餐

职位描述：

职位职责：
1、开发跨平台高性能native客户端核心组件；
2、负责沟通及创作工具的系统设计，实现，优化和演进；
3、研究分析主流 IM，优化实现方案，改进产品功能；
4、负责设计和优化 IM 协议、弱网通信、推送、存储、网络并发、并行计算、加密以及安全等；
5、保证工程质量和开发效率。

职位要求：
1、良好的系统编程能力，喜爱或有Rust经验以及扎实的C/C++功底更佳；
2、喜爱关注新技术，愿意尝试更优解决方案；
3、不设边界，愿意探索了解事物运转原理；
4、熟悉开源社区；
5、热衷自动化完成事情；
6、关注代码设计，有持续学习习惯。

工作地址

北京 - 海淀区 - 学清路东50米学清嘉创大厦B座1层

中国人民银行金融科技研究院区块链项目组招聘

加密资产研究员

40k-80k /北京 / 经验3-5年 / 本科及以上 / 全职

金融

2020-06-22 发布于拉勾网

职位诱惑:

发展空间大，福利好

职位描述:

•跟踪加密资产相关技术研究

职位要求:

•熟悉大数据分析技术、具备扎实的网络知识，3年以上相关工作经验;

•至少熟练使用Java/Python/Rust/Go中的一种语言;

•较高的英语水平;

•了解加密资产市场、或股票、证券等资产交易市场

能力突出者，薪资不设限！！！

工作地址

北京 - 丰台区 - 太平桥 - 恒泰中心A座20层

RUST 开发环境安装

1，安装 rust 编译依赖的 C 语言编译工具：Windows-VS2013+， Linux-gcc、clang，macOS-xcode
2，安装 rust 编译器：https://www.rust-lang.org/zh-CN/tools/install
3，设置 rustc 的 PATH 环境变量：%USERPROFILE%\.cargo\bin
C:\Users\zyr\.cargo\bin
4，测试：rustc -V
rustc 1.44.1 (c7087fe00 2020-06-17)
5，安装 IDE:VSCode https://code.visualstudio.com
6，安装插件：rls 和 native debug，C++

C:\Users\zyr\Desktop\rustup-init.exe

```
 C:\Users\zyr\.rustup

This can be modified with the RUSTUP_HOME environment variable.

This path will then be added to your PATH environment variable by
modifying the HKEY_CURRENT_USER/Environment/PATH registry key.

You can uninstall at any time with rustup self uninstall and
these changes will be reverted.

Current installation options:


   default host triple: x86_64-pc-windows-msvc
     default toolchain: stable
               profile: default
  modify PATH variable: yes

1) Proceed with installation (default)
2) Customize installation
3) Cancel installation
>


info: profile set to 'default'
info: default host triple is x86_64-pc-windows-msvc
info: syncing channel updates for 'stable-x86_64-pc-windows-msvc'
471.9 KiB / 471.9 KiB (100 %) 398.2 KiB/s in  2s ETA:  0s
info: latest update on 2020-06-18, rust version 1.44.1 (c7087fe00 2020-06-17)
info: downloading component 'cargo'
524.6 KiB /   3.3 MiB ( 16 %)    0 B/s in  1s ETA: Unknown
```

```
Rust is installed now. Great!

To get started you need Cargo's bin directory (%USERPROFILE%.cargo\bin) in your
PATH
environment variable. Future applications will automatically have the
correct environment, but you may need to restart your current shell.

Press the Enter key to continue.
```

编写

建立 rustproject 文件夹

cargo new hello

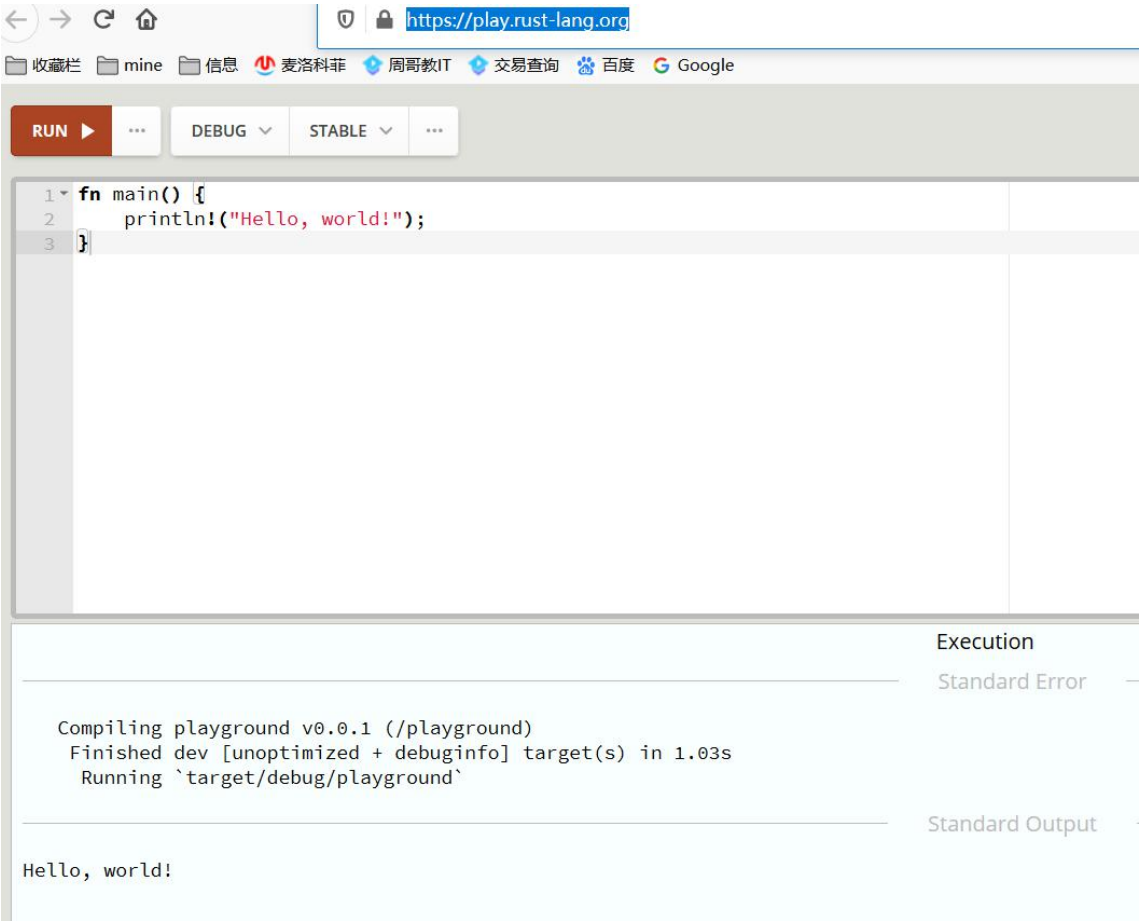
cd .\hello

cargo build

cargo run

注释://, /* xxxxx */

https://play.rust-lang.org/



Hello world-调试

调试
安装插件 C/C++ (Windows)或 CodeLLDB (OS X / Linux)
Click Debug -> Add Configuration
If you're on Windows then select C++ (Windows) Mac or Linux then select LLDB: Custom Launch
create and open launch.json
"program": "${workspaceFolder}/target/debug/${workspaceFolderBasename}.exe"
File -> Preferences -> Settings:break
下断点
F5 开始调试
Cargo 使用
内置构建系统和包管理器
cargo --version
cargo new
Cargo.toml
[dependencies]

mobc-redis = "0.5.0"
mobc = "0.5.7"
cargo build [--release]
cargo run
cargo check

Rust 基础语法-变量与基本类型

变量定义与可变性：
let x = 5;
let mut x = 5;

隐藏：
let x=5;
let x=x+1;
let x=x*2;
let spaces = " ";
let spaces=spaces.len();

变量类型：（整型（i8-i64, u8-u64, isize,usize)，浮点型(f32, f64)，布尔型（bool, 1 字节），字符型（4 字节））
let x:i32=100;//123_456, 0xff, 0o67, 0b111_000, b'A'(u8)
let x=100;
let value:u32="42".parse.expect("Not a number");
let x=1.414;//f64
let x:f32=1.732; //f32
let t=true;
let f:bool=false;
let c='H';
let hello:&str="hello world";
let hello="hello world"

常量：
const MAX_VALUE:u32 = 100_000;
全局变量：
static HELLO_WORLD:&str="hello world";
static mut COUNTER:u32=100;
数据类型（2）
元组类型:不同类型的值组合在一起
let tup:(i32, f64, u8)=(500, 6.4,1);
let tup=(500, 6.4,1);

let (x,y,z)=tup;
tup.0, tup.1,tup.2
数组类型：相同类型的值组合在一起
let a=[1,2,3,4,5];
let fruits=[“apple”，“pear”，“grape”，“banana”];
let a:[i32;5]=[1,2,3,4,5];
let a=[3;5]//[3,3,3,3,3]
a[0], a[1], a[10];
类型别名 type
type Kilometers = i32;
let x:i32=5;
let y:Kilometers = 5;
println!(“x+y={}”, x+y);

Rust 运算符

算术运算符

+	expr + expr	算术加法	Add
+=	var += expr	算术加法与赋值	AddAssign
-	expr - expr	算术减法	Sub
-=	var -= expr	算术减法与赋值	SubAssign
*	expr * expr	算术乘法	Mul
*=	var *= expr	算术乘法与赋值	MulAssign
/	expr / expr	算术除法	Div
/=	var /= expr	算术除法与赋值	DivAssign
%	expr % expr	算术取模	Rem
%=	var %= expr	算术取模与赋值	RemAssign

(没有++，--）

比较运算符

<	expr < expr	小于比较	PartialOrd
<=	expr <= expr	小于等于比较	PartialOrd
==	expr == expr	等于比较	PartialEq
>	expr > expr	大于比较	PartialOrd
>=	expr >= expr	大于等于比较	PartialOrd
!=	var != expr	不等比较	PartialEq

位运算符

!	!expr	按位非	Not
&	expr & expr	按位与	BitAnd
&=	var &= expr	按位与与及赋值	
<<	expr << expr	左移	Shl

<<= var <<= expr 左移与赋值 ShlAssign
>> expr >> expr 右移 Shr
>>= var >>= expr 右移与赋值 ShrAssign
^ expr ^ expr 按位异或 BitXor
^= var ^= expr 按位异或与赋值
| expr | expr 按位或 BitOr
逻辑运算符
|= var |= expr 按位或与赋值 BitOrAssign
&& expr && expr 逻辑与
! !expr 逻辑非 Not
|| expr || expr 逻辑或
其它运算符
= var = expr, ident = type 赋值 / 等值
& &expr, &mut expr 借用
* *expr 解引用
- - expr 算术取负 Neg
-> fn(...) -> type, |...| -> type 函数与闭包，返回类型
. expr.ident 成员访问
等等

语句

（1）if

```
if
let x = 10;
if x<5 {        //if x{
    println!("x<5");
}else if x%3==0 {
    println!("x%3==0");
} else {
     println!(“x>5 and x%3!=0”};
}
let if
let condition = true;
let value = if condition {
     8
}else {
     5 // "five", err
};
loop
let mut counter = 0;
```

```
let result = loop {
        counter += 1;
        if counter == 10 {
            break counter * 2;
        }
};
while
let mut number = 3;
while number != 0 {
        println!("{}!", number);
        number = number - 1;
    }
for
let a = [10, 20, 30, 40, 50];
for element in a.iter() {
        println!("the value is: {}", element);
    }
match
let x = 5;
match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    4 => println!("four"),
    5 => println!("five"),
    _ => println!("something else"),
}

let x = 5;
let number = match x {
    1 => "one",
    2 => "two",
    3 => "three",
    4 => "four",
    5 => "five",
    _ => "something else",
};
let number = 13;
println!("Tell me about {}", number);
match number {
    1 => println!("One!"),
```

```
    2 | 3 | 5 | 7 | 11 => println!("This is a prime"),
    13...19 => println!("A teen"),
    _ => println!("Ain't special"),
}

let n    = 4;
match n    {
   n if n > 2 => println!("> 2"),
   n if n > 3 => println!("> 3"),
   _ => println!("some else"),
};
```

函数

```
fn funcname(paramname:type, ...)->type{
    if ...{
        return value;
    }
    value
}

fn myadd(x:i32, y:i32)->i32 {
    x+y
}
fn factorial(i: u64) -> u64 {
    match i {
        0 => 1,
        n => n * factorial(n-1)
    }
}

fn factorial(i: u64) -> u64 {
        let mut acc = 1;
         for num in 2..=i {
            acc *= num;
         }
        acc
}
```

函数指针做参数

```
fn add_one(x:i32)->i32 {
    x+1
```

```
}

fn myadd(f:fn(i32)->i32, arg:i32)->i32 {
    f(arg)+f(arg)
}
```

() :表示 1 个特殊的元组——没有元素的元组，称为 unit，它表示一个函数没有任何信息需要返回，类似于 void

```
fn main() {
    //statements
  }
 fn main() -> () {
    //statements
  }
```

rust1.6 开始，main 可以返回 Result

```
use std::error::Error;
fn main() -> Result<(), Box<dyn Error>>{
}
```

!：发散函数不返回，它使用感叹号!作为返回类型表示

```
fn never_return() -> ! {
  panic!("This function will never return");
}
```

多值返回

```
fn pow_2_3(x: i32) -> (i32, i32) {
  (x*x, x*x*x)
}
```

命令行参数

```
use std::env;
let args: Vec<String> = env::args().collect();
args.len();
let query = &args[1];
let filename = &args[2];
```

函数（3）:从不返回的 never type !

在函数从不返回的时候充当返回值

```
fn bar() -> ! {//空类型或者 never 类型，从不返回值的函数（发散函数）充当返回类型
    // --snip--
}

let guess: u32 = match guess.trim().parse() {
    Ok(num) => num,
    Err(_) => continue, //continue 不返回
};
```

loop{}
panic!
!类型可以强制转换为其它类型

结构体

```
#[derive(Debug)]//添加注解来派生 debug trait
struct User {
	email: String,
	username: String,
	active: bool,
	sign_in_count: u64,
}//没有分号
let user1 = User {
	email: String::from("someone@example.com"),
	username: String::from("someusername123"),
	active: true,
	sign_in_count: 1,
};
let user2 = User {
	email: String::from("another@example.com"),
	username: String::from("anotherusername567"),
	active: user1.active,
	sign_in_count: user1.sign_in_count,
	//或者..user1
};
println!(“user:{:?}”, user1};//结构体没有实现默认的 display 实现，无法确定使用什么格式化内容。
:?告知 println!使用 debug 的格式化输出，一种格式化 trait，友好形式打印结构体
fn build_user(email: String, username: String) -> User {
	User {
		email: email,//email,
		username: username,//username,
		active: true,
		sign_in_count: 1,
	}
}
```

结构体中也可以存储指向其他数据的引用，不过涉及到生命周期问题，保证实例中引用数据的有效期不能短于实例本身。email:String VS email:&str

元组结构体：

```
struct Color(i32, i32, i32);//无成员名称，匿名
struct Point(i32, i32, i32);

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

可以定义没有任何字段的空结构体，为类型实现 trait

元组和结构体的区别。元组的成员没有名字

```
struct Rectangle {
    width: u32,
    height: u32,
}
impl Rectangle {
  fn area(&self) -> u32 {//方法，self 引用，指代调用该方法的结构体实例
        self.width * self.height
    }
  fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
    fn square(size: u32) -> Rectangle {//内联函数，Rectangle::square(3);
        Rectangle { width: size, height: size }
    }
}
```

和 C++的定义对比

枚举

```
enum IpAddrKind {
    V4,//枚举变体
    V6,
}
fn value_in_ipkind(ipkind: IpAddrKind ) -> u32 {
    match ipkind{
        IpAddrKind ::V4 => 4,
        IpAddrKind ::V6 => 6,
     }
}

struct IpAddr {
    kind: IpAddrKind,
```

```
    address: String,
}
let home = IpAddr {
    kind: IpAddrKind::V4,
    address: String::from("127.0.0.1"),
};
let loopback = IpAddr {
    kind: IpAddrKind::V6,
    address: String::from("::1"),
};

enum IpAddr {
    V4(String),//数据嵌入枚举变体，和 C 语言不同
    V6(String),
}
let home = IpAddr::V4(String::from("127.0.0.1"));
let loopback = IpAddr::V6(String::from("::1"));

enum Message {
    Quit,
    Move { x: i32, y: i32 },//嵌入匿名结构体
    Write(String),//嵌入字符串
    ChangeColor(i32, i32, i32),//嵌入三个整数
}
```

和结构体区别：结构体需要定义三种不同类型的结构体，枚举类型是单独的一个类型，便于设计一个函数统一处理。

```
impl Message {
    fn call(&self) {
        ...
    }
}
let m = Message::Write(String::from(“helloworld”));
m.call();
```

枚举（2）Match 与枚举结合

绑定值模式：

```
enum IpAddr {
    V4(String),//数据嵌入枚举变体
    V6(String),
}
let home = IpAddr::V4(String::from("127.0.0.1"));
```

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

fn value_in_ipkind(ipkind: IpAddr ) -> u32 {
    match ipkind{
        IpAddr ::V4(str) =>{
            println!("V4: {}", str);
            4
        },
        IpAddr ::V6(str) => {
            println!("V6: {}", str);
            6
        },
    }
}
```

match 与枚举结合。使用 match 来匹配枚举值，并将其中的值绑定到某个变量上，接着根据这个值执行相应的代码。社区用户最爱

枚举（2）:标准库中的 Option

Option 与空值处理

没有 NULL（发明人东尼·霍尔(Tony Hoare),图灵奖得主把 Null 引用称为十亿美元的错误）

```
enum Option <T>{//标识一个值无效或缺失
    Some(T),//T 是泛型，可以包含任何数据
    None,
}
let some_number = Some(5);//省略 Option::
let some_string = Some("a string");//值存在，被 Some 持有
let absent_number: Option<i32> = None;//不存在有效值

let x: i8 = 5;
let y: Option<i8> = Some(5);//x, y 是不同类型

let sum = x + y;//编译失败
```

Option<i8>，需要考虑值不存在的情况，编译器迫使我们处理这种情况。Option<T>要使用 T，必须转换为 T

持有一个可能为空的值，需要显示放入 Option<T>中。只要一个值的类型不是 Option<T>的，就可以安全假设这个值不是非空的

C 语言中的指针

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        None => None,
        Some(i) => Some(i + 1),//i 绑定了 Some 所包含的值
```

```
    }
}

//match 是穷尽匹配，强制处理 None 情况


let five = Some(5);
let six = plus_one(five);
let none = plus_one(None);
if let

let some_u8_value = Some(0u8);
match some_u8_value {
    Some(3) => println!("three"),
    _ => (),
}


//if 和 let，来处理只匹配一个模式的值而忽略其他模式的情况
let some_u8_value = Some(0u8);
if let Some(3) = some_u8_value {
    println!("three");
} else {
    println!("other");
}
```

模块

（1）二进制包
模块系统：
包(package)-->单元包(crate，二进制单元包和库单元包)-->模块（子模块）；
工作空间（workspace，巨大项目，多个包相互关联的项目）
一个包中必须带有至少一个（库或者二进制）crate，可以带有零个或一个库 crate 和任意多个二进制 crate。如果包同时包含 src/main.rs 和 src/lib.rs，那么它带有两个 crate：一个库和一个二进制项目，同名。如果只有其中之一，则包将只有一个库或者二进制 crate。
cargo new xxx
绝对路径(crate::xxx::yyy::zzz)与相对路径
super
包可以带有多个二进制 crate，需将其文件置于 src/bin 目录；每个文件将是一个单独的二进制 crate。
src/lib.rs or main.rs:

mod calc；
pub use crate::calc::adder;
adder::xxx

模块拆分

src/calc.rs:

```
pub mod adder {
    pub fn add(x:i32, y:i32)->i32{
    }
}
```

or
src/calc.rs:
pub mod adder;

src/calc/adder.rs

```
pub fn add(x:i32, y:i32)->i32{
}
```

main.rs 与 lib.rs
cargo new --lib xxx
mod（3）：pub 模块，函数，结构体，枚举等

```
mod calc {
    pub mod adder {
        pub fn add(x:i32, y:i32)->i32 {
            // 函数体
            x+y
        }
    }
    pub struct User {
        pub name:String,
        id:u32
    }
    impl User {
        pub fn build_user(String name)->User {
            User {
                name,
                id:1,
            }
```

```
        }
    }
    pub enum IpAddrKind{
        V4,
        V6,
    }
}
fn main() {

    // 绝对路径
    crate::calc::adder::add(3,4);
    // 相对路径
    calc::adder::add(3,4);

    let mut user = crate::calc::User::build_user(String::from(“tom”));
    user.name=String::from(”jack”);
    //user.id = 10;
    let ipv4=crate::calc::IpAddrKind::V4;

}
```

结构体默认是私有
枚举默认是共有
mod（4）：use
use：将模块，函数，结构体，枚举类型等名称引入当前作用域
crate::calc::adder::add_one
use crate::calc::adder;
adder::add_one

use 导入标注库：标准库（std）对于你的包来说也是外部 crate。因为标准库随 Rust 语言一同分发，无需修改 Cargo.toml 来引入 std，不过需要通过 use

use std::collections::HashMap;//结构体，枚举类型等使用完整路径

use std::fmt::Result;
use std::io::Result as IoResult;

use std::cmp::Ordering;
use std::io;
use std::{cmp::Ordering, io};//嵌套清理众多 use

```
use std::io;
use std::io::Write;
use std::io::{self, Write};

use std::collections::*;//所有公共条目都导入当前作用域
```

pub use：use 引入的名称是以私有的方式在新的作用域生效。加 pub 可以让外部代码访问到这些名称。即重导出（re-exporting)

```
mod sound {
    pub mod instrument {
        pub fn clarinet() {//黑管
            // 函数体
        }
    }
}
mod performance_group {
    pub use crate::sound::instrument;//重导出（re-exporting）

    pub fn clarinet_trio() {//三人组
        instrument::clarinet();
        instrument::clarinet();
        instrument::clarinet();
    }
}

fn main() {
    performance_group::clarinet_trio();
    performance_group::instrument::clarinet();
}
```

mod（5）：使用外部包

crates.io

Cargo.toml：

```
[dependencies]
rand = "0.3.14"
```

.rs 文件：

```
use rand::Rng;
```

运行：cargo build

Blocking waiting for file lock on package cache

C:\Users\zyr\.cargo\.package-cache

```
 Fetch [=====>                                                    ]  11.73%
```

换源：

cd ~/.cargo #C:\Users\zyr\.cargo

新创建 config 文件:

```
[source.crates-io]
registry = "https://github.com/rust-lang/crates.io-index"
replace-with = 'ustc'
[source.ustc]
registry = "git://mirrors.ustc.edu.cn/crates.io-index"
```

如果不支持 git，改为：http://mirrors.ustc.edu.cn/crates.io-index

第一次编译后生成 Cargo.lock 文件：确保 build 是可重现的，任何人在任何时候重新编译我们的代码都会生成相同的产物。rand 0.3.15 0.3.14。Cargo 遍历依赖库找到具体版本号，并将版本号写入 lock 文件。再次 build 的时候，优先使用 lock 文件中的版本。

修改 Cargo.toml，然后运行：cargo update 更新 lock 文件

模块（6）:workspace

workspace：管理多个相关的协同开发的包

```
$ mkdir add
$ cd add
```

创建: Cargo.toml

```
[workspace]
members = [
    "adder",
    "add_one",
]
cargo new adder
cargo build

cargo new add_one --lib
add_one/src/lib.rs
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

修改 adder/Cargo.toml

```
[dependencies]
add_one = { path = "../add-one" }
```

修改 adder/src/main.rs

```
use add_one;
```

add_one::add_one(num)
cargo build
cargo run -p adder
工作空间只在根目录有一个 Cargo.lock
如果在 adder/Cargo.toml 和 add-one/Cargo.toml 中都增加 rand crate，则 Cargo 会将其都解析为同一版本并记录到唯一的 Cargo.lock 中
发布到 crates.io
代码，文档，测试都做好了
登录并创建 Crates.io 账号-GitHub 账号
crates.io/me/ 的账户设置页面并获取 API token
cargo login apitoken
crate Cargo.toml 需要一个唯一的名称和协议等
[package]
name = "xxx_game"
license = "MIT"

cargo publish 发布是 永久性的
修改 Cargo.toml 中的版本号，重新发布
撤回一个 crate
cargo yank --vers 1.0.1
cargo yank --vers 1.0.1 --undo

集合

（1）:vector 动态数组
let mut v: Vec<i32> = Vec::new();
let v = vec![1, 2, 3];
v.push(5);
访问：
v[0], &v[0], for

溢出：
let third: &i32 = &v[2]; //100,panic
v.get(2)//100, none //Option<&T>

//获取了 vector 的第一个元素的不可变引用并尝试在 vector 末尾增加一个元素的时候
let first = &v[0];//第一个元素的不可变引用
v.push(6);//err

```
for i in &v {//&mut v
    println!("{}", i);
```

```
}
enum Cell {
    Int(i32),
    Float(f64),
    Text(String),
}

let row = vec![
    Cell::Int(3),
    Cell::Text(String::from("blue")),
    Cell::Float(10.12),
];
```

//编写程序时不能确切无遗地知道运行时会储存进 vector 的所有类型，枚举技术就行不通了。相反，你可以使用 trait 对象

集合（2）:字符串

Rust 的核心语言中只有一种字符串类型：str，字符串 slice，它通常以被借用的形式出现，&str。它们是一些储存在别处的 UTF-8 编码字符串数据的引用

String

```
let mut s = String::new();
s.push_str("bar");
s.push('l');

let data = "initial contents";
let s = data.to_string();
// 该方法也可直接用于字符串字面值：
let s = "initial contents".to_string();

let s = String::from("initial contents");

let mut s1 = String::from("foo");
let s2 = "bar";
s1.push_str(s2);//引用，不用索引权
println!("s2 is {}", s2);

let s1 = String::from("Hello, ");
let s2 = String::from("world!");
let s3 = s1 + &s2; // 注意 s1 被移动了，不能继续使用
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");
```

```
let s = format!("{}-{}-{}", s1, s2, s3);
```

字符串 slice

```
let s="3 中国";
let ch=&s[0];//err，utf8，字符串中字节索引，并不总对应一个有效的 unicode 值

let hello = "З д р а в с т в у й т е";
let s = &hello[0..4];//s 会是一个 &str，它包含字符串的头四个字节
获取 &hello[0..1] 在运行时会 panic

for c in "helloworld".chars() {
    println!("{}", c);
}

for b in "helloworld".bytes() {
    println!("{}", b);
}
```

集合（3）：hashmap

哈希 map（hash map）。Key-value 集合，插入查找 O（1），hashMap,map,dict。HashMap<K, V> 类型储存了一个键类型 K 对应一个值类型 V 的映射。它通过一个 哈希函数（hashing function）来实现映射，决定如何将键和值放入内存中

```
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);
for (key, value) in &scores {
    println!("{}: {}", key, value);
}
```

更新

```
scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Blue"), 25);
//只在键没有对应值时插入
scores.entry(String::from("Red")).or_insert(50);
```

or_insert 方法事实上会返回这个键的值的一个可变引用（&mut V）。这里我们将这个可变引用储存在 count 变量中，所以为了赋值必须首先使用星号（*）解引用 count

```
//如果是第一次看到某个单词，就插入值 0
let count = word_map.entry(word).or_insert(0);
```

```
*count += 1;
```

像 i32 这样的实现了 Copy trait 的类型，其值可以拷贝进哈希 map。对于像 String 这样拥有所有权的值，其值将被移动而哈希 map 会成为这些值的所有者

```
let field_name = String::from("Favorite color");
let field_value = String::from("Blue");
let mut map = HashMap::new();
map.insert(field_name, field_value);
// 这里 field_name 和 field_value 不再有效，
// 尝试使用它们看看会出现什么编译错误！
```

另一个构建哈希 map 的方法是使用一个元组的 vector 的 collect 方法

```
let teams    = vec![String::from("Blue"), String::from("Yellow")];
let initial_scores = vec![10, 50];

let scores: HashMap<_, _> = teams.iter().zip(initial_scores.iter()).collect();
```

错误处理

(1):panic

错误的处理，打开文件失败，调用函数失败等

Rust 将错误组合成两个主要类别：可恢复错误（recoverable）和 不可恢复错误（unrecoverable）

Rust 并没有异常。可恢复错误有 Result<T, E> 值，以及 panic!。其中 panic!在遇到不可恢复错误时停止程序执行

```
fn main() {
    let v = vec![1, 2, 3];
    v[99];
}

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

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。

unwinding

当出现 panic 时，程序默认会开始 展开（unwinding），这意味着 Rust 会回溯栈（沿着调用栈相反方向）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，而且需要在二进制中存储许多额外信息。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。如果你需要项目的最终二进

制文件越小越好，panic 时通过在 Cargo.toml 的 [profile] 部分增加 panic = 'abort'，可以由展开切换为终止

[profile.release]

panic = 'abort'

backtrace

backtrace 是一个执行到目前位置所有被调用的函数的列表

backtrace 从头开始读直到发现你编写的文件。这是问题的发源地。这一行往上是你的代码调用的代码；往下则是调用你的代码的代码。这些行可能包含核心 Rust 代码，标准库代码或用到的 crate 代码

RUST_BACKTRACE=1 cargo run

$Env:RUST_BACKTRACE=1

错误处理(2):Result<T,E>

Result 枚举类型：

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

use std::fs::File;
fn main() {
	let f = File::open("hello.txt");
	let f = match f {
		Ok(file) => file,
		Err(error) => {
			panic!("problem: {:?}", error),
		},
	};
}
```

unwrap 与 expect：

//如果 Result 值是成员 Ok，unwrap 会返回 Ok 中的值。如果 Result 是成员 Err，unwrap 会为我们调用 panic!

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

//expect 与 unwrap 的使用方式一样：返回文件句柄或调用 panic! 宏。expect 用来调用 panic! 的错误信息将会作为参数传递给 expect ，而不像 unwrap 那样使用默认的 panic! 信息

```
fn main() {
	let f = File::open("hello.txt").expect("Failed to open hello.txt");
}
```

传播错误：

```
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),
    }
}
```

？快捷传播：

```
fn read_username_from_file() -> Result<String, io::Error> {
    let mut f = File::open("hello.txt")?;//? 会提早返回整个函数并将一些 Err 值传播给调用者，? 只能被用于返回值类型为 Result 的函数
    let mut s = String::new();
    f.read_to_string(&mut s)?;
  //File::open("hello.txt")?.read_to_string(&mut s)?;
    Ok(s)
}
```

在不返回 Result 的函数中，当调用其他返回 Result 的函数时，需要使用 match 或 Result 的方法之一来处理，而不能用 ? 将潜在的错误传播给代码调用方

```
fn main() {
    let f = File::open("hello.txt")?;//err
}
fn main() -> Result<(), Box<dyn Error>> {
    let f = File::open("hello.txt")?;
    Ok(())
}
```

Box<dyn Error> 为 "trait 对象",可以理解 Box<dyn Error> 为 "任何类型的错误"。

panic! or result?

如果代码 panic，就没有恢复的可能。如果在任何错误场景都调用 panic!，不管是否有可能恢复，这样就是你代替调用者决定了这是不可恢复的。选择返回 Result 值的话，就将选择权交给了调用者，而不是代替他们做出决定。调用者可能会选择以符合他们场景的方式尝试恢复。返回 Result 是定义可能会失败的函数的一个好的默认选择。

示例、代码原型和测试都非常适合 panic

拥有健壮的错误处理代码的同时也会使得例子不那么明确;unwrap 和 expect 方法在原型设计时非常方便。当我们准备好让程序更加健壮时，它们会在代码中留下清晰的标记
当我们比编译器知道更多的情况,有一些其他的逻辑来确保 Result 会是 Ok 值时，调用 unwrap 也是合适的

```
use std::net::IpAddr;
let home: IpAddr = "127.0.0.1".parse().unwrap();
```

在当有可能会导致有害状态的情况下建议使用 panic。有害状态是指当一些假设、保证、协议或不可变性被打破的状态，例如无效的值、自相矛盾的值或者被传递了不存在的值。
如果别人调用你的代码并传递了一个没有意义的值，最好的情况也许就是 panic。并警告使用你的库的人他的代码中有 bug 以便他能在开发时就修复它。类似的，panic! 通常适合调用不能够控制的外部代码时，这时无法修复其返回的无效状态。
函数通常都遵循 契约（contracts）：他们的行为只有在输入满足特定条件时才能得到保证。当违反契约时 panic 是有道理的，因为这通常代表调用方的 bug，而且这也不是那种你希望调用方必须处理的错误。函数的契约，尤其是当违反它会造成 panic 的契约，应该在函数的 API 文档中得到解释。
然而当错误预期会出现时，返回 Result 仍要比调用 panic! 更为合适。这样的例子包括解析器接收到错误数据，或者 HTTP 请求返回一个表明触发了限流的状态。

文件 IO

创建
读
写
删除
重命名

各编程语言基础语法对比

https://www.mallocfree.com/data/bcyycmp-mallocfree.com.pdf

泛型编程

（1）：函数泛型
泛型是具体类型或其他属性的抽象替代

```
fn largest(list: &[i32]) -> i32 {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
```

```
                largest = item;
            }
        }
        largest
}
  let number_list = vec![34, 50, 25, 100, 65];
  let result = largest(&number_list);

fn largest<T>(list: &[T]) -> T {//T:PartialOrd+Copy
        let mut largest = list[0];
        for &item in list.iter() {
            if item > largest {
                largest = item;
            }
        }
        largest
}
        let number_list = vec![34, 50, 25, 100, 65];

        let result = largest(&number_list);

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

        let result = largest(&char_list);
struct Point<T> {
        x: T,
        y: T,
}

fn main() {
        let integer = Point { x: 5, y: 10 };
        let float = Point { x: 1.0, y: 4.0 };
}

struct Point<T, U> {
        x: T,
        y: U,
}

fn main() {
        let both_integer = Point { x: 5, y: 10 };
```

```
    let both_float = Point { x: 1.0, y: 4.0 };
    let integer_and_float = Point { x: 5, y: 4.0 };
}
```

泛型编程（3）：枚举泛型

```
enum Option<T> {
    Some(T),
    None,
}
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

泛型编程（4）：方法泛型

泛型编程（5）:单态化与性能分析

泛型性能：编译阶段单态化

Rust 实现了泛型，使得使用泛型类型参数的代码相比使用具体类型并没有任何速度上的损失。

Rust 通过在编译时进行泛型代码的 单态化（monomorphization）来保证效率。单态化是一个通过填充编译时使用的具体类型，将通用代码转换为特定代码的过程。

使用泛型来编写不重复的代码，而 Rust 将会为每一个实例编译其特定类型的代码。这意味着在使用泛型时没有运行时开销；当代码运行，它的执行效率就跟好像手写每个具体定义的重复代码一样

Trait 编程

（1）

trait 类似于其他语言中的接口（interfaces）的功能。一个类型的行为由其可供调用的方法构成。如果可以对不同类型调用相同的方法的话，这些类型就可以共享相同的行为了。trait 是定义一个实现某些目的所必需的行为的集合。

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

pub struct News {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}
```

```
impl Summary for News {
    fn summarize(&self) -> String {
        format!("{}, by {} ({})", self.headline, self.author, self.location)
    }
}
pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}

let tweet = Tweet {
    username: String::from("Mallocfree"),
    content: String::from("of course, as you probably already know, people"),
    reply: false,
};

println!("1 new tweet: {}", tweet.summarize());
```

提供默认实现

```
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more...)")
    }
}


let article = News {
    headline: String::from("Penguins win the Stanley Cup Championship!"),
    location: String::from("Pittsburgh, PA, USA"),
    author: String::from("Iceburgh"),
    content: String::from("The Pittsburgh Penguins once again are the best
    hockey team in the NHL."),
};

println!("New article available! {}", article.summarize());
```

Display, Copy, PartialOrd 等。

Trait 编程（2）

trait 作为参数

```
pub fn notify(item: impl Summary) {
	println!("Breaking news! {}", item.summarize());
}

trait bound(约束)
pub fn notify<T: Summary>(item: T) {
	println!("Breaking news! {}", item.summarize());
}

pub fn notify(item: impl Summary + Display) {}
pub fn notify<T: Summary + Display>(item: T) {}
fn some_function<T: Display + Clone, U: Clone + Debug>(t: T, u: U) -> i32 {}
fn some_function<T, U>(t: T, u: U) -> i32
	where T: Display + Clone,
		U: Clone + Debug{ }
```

返回 trait

```
fn returns_summarizable() -> impl Summary {//只适用于返回单一类型的情况
	Tweet {
		username: String::from("mallocfree"),
		content: String::from("of course, as you probably already know, people"),
		reply: false,
	}
}
```

使用 trait bounds 来修复 largest 函数

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

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

Trait 编程（3）

使用 trait bound 有条件地实现方法

```
use std::fmt::Display;
struct Point {
	x: T,
	y: T,
}
impl<T>	Point <T> {
	fn new(x: T, y: T) -> Self {
		Self {
			x,
			y,
		}
	}
}
impl<T: Display + PartialOrd> Point <T> {
	fn cmp_display(&self) {
		if self.x >= self.y {
			println!("The largest member is x = {}", self.x);
		} else {
			println!("The largest member is y = {}", self.y);
		}
	}
}
```

Trait 编程（4）

完全限定语法与消歧义：调用相同名称的方法

```
trait Pilot {
	fn fly(&self);
}
trait Wizard {
	fn fly(&self);
}
struct Human;
impl Pilot for Human {
	fn fly(&self) {
		println!("This is your captain speaking.");
	}
}
impl Wizard for Human {
	fn fly(&self) {
		println!("Up!");
	}
}
```

```
impl Human {
	fn fly(&self) {//方法
		println!("*waving arms furiously*");
	}
}
fn main() {
	let person = Human;
	Pilot::fly(&person);
	Wizard::fly(&person);
	person.fly();
}
trait Animal {
	fn baby_name() -> String;
}
struct Dog;

impl Dog {
	fn baby_name() -> String {//关联函数
		String::from("Spot")
	}
}
impl Animal for Dog {
	fn baby_name() -> String {
		String::from("puppy")
	}
}

fn main() {
	println!("A baby dog is called a {}", Dog::baby_name());
	//Animal::baby_name();<Dog as Animal>::baby_name()

}
```

<Dog as Animal>::baby_name();//使用完全限定语法来指定我们希望调用的是 Dog 上 Animal trait 实现中的 baby_name 函数

newtype 模式为类型实现 trait

孤儿规则（orphan rule）:只要 trait 或类型对于当前 crate 是本地的话就可以在此类型上实现该 trait（都不是本地的，那就不行）。

绕开这个限制的方法是使用 newtype 模式（newtype pattern），它涉及到在一个元组结构体（用没有命名字段的元组结构体来创建不同的类型）中创建一个新类型。这个元组结构体带有一个字段作为希望实现 trait 的类型的简单封装。接着这个封装类型对于 crate 是本地的，这样就可以在这个封装上实现 trait

```
use std::fmt;

struct Wrapper(Vec<String>);

impl fmt::Display for Wrapper {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "[{}]", self.0.join(", "))
    }
}

fn main() {
    let w = Wrapper(vec![String::from("hello"), String::from("world")]);
    println!("w = {}", w);
}
```

动态大小类型（dynamically sized types）DST

考虑一下这些代码，有的不能正常工作：

```
let s1: str = "Hello world";
let s2:&str="hello world";
let s3="hello mallocfree.com";
```

str 是一个 DST；直到运行时我们都不知道字符串有多长。因为直到运行时都不能知道大其小，也就意味着不能创建 str 类型的变量，也不能获取 str 类型的参数。

Rust 需要知道应该为特定类型的值分配多少内存，同时所有同一类型的值必须使用相同数量的内存。s2 和 s3 的类型是 &str 而不是 str。slice 数据结储存了开始位置和 slice 的长度。所以虽然 &T 是一个储存了 T 所在的内存位置的单个值，&str 则是 两个 值：str 的地址和其长度。我们总是知道 &str 的大小，而无论其引用的字符串是多长。有一些额外的元信息来储存动态信息的大小：必须将动态大小类型的值置于某种指针之后。

为了处理 DST，Rust 有一个特定的 trait 来决定一个类型的大小是否在编译时可知：这就是 Sized trait。这个 trait 自动为编译器在编译时就知道大小的类型实现。另外，Rust 隐式的为每一个泛型函数增加了 Sized bound。也就是说，对于如下泛型函数定义：

```
fn generic(t: T) {
    // --snip--
}
```

实际上被当作如下处理：

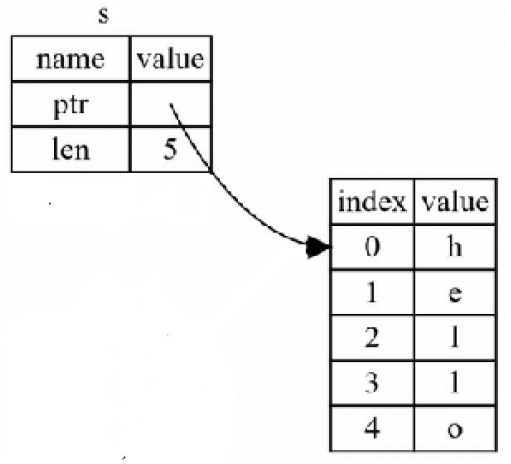
```
fn generic<T: Sized>(t: T) {
    // --snip--
}
```

泛型函数默认只能用于在编译时已知大小的类型。然而可以使用如下特殊语法来放宽这个限制：

```
fn generic<T: ?Sized>(t: &T) {
    // --snip--
}
```

?Sized trait bound 与 Sized 相对；也就是说，它可以读作 “T 可能是也可能不是 Sized 的”。这个语法只能用于 Sized ，而不能用于其他 trait。

t 参数的类型从 T 变为了 &T：因为其类型可能不是 Sized 的，所以需要将其置于某种指针之后。



面向对象

（1）

对象包含数据和行为：结构体和枚举

封装实现细节：结构体 pub 公开，默认私有

继承：继承在很多语言中失宠，因为其时常带有共享多于所需的代码的风险。子类不应总是共享其父类的多种特征，但是继承却始终如此。如此会使程序设计更为不灵活，并引入无意义的子类方法调用，或由于方法实际并不适用于子类而造成错误的可能性。trait 替代

多态

面向对象（2）:多态

vector 只能存储同种类型元素，定义枚举来储存整型，浮点型和文本成员的替代方案

多态：使用 trait 对象存储不同类型值，动态派发

定义一个 Draw trait；两个实现 Draw trait 的控件结构体：Button，SelectBox；一个屏幕结构体：Screen

```
pub trait Draw {
    fn draw(&self);
}
pub struct Screen {
    //Box 是智能指针，里面存放所有实现了 Draw trait 的具体类型，比如 Box<Button>
    pub components: Vec<Box<dyn Draw>>,
}
```

```
impl Screen {
    pub fn run(&self) {
        for component in self.components.iter() {
            component.draw();//动态派发，运行时，通过 trait 对象内部的指针去定位具体调动那个方法；单态化是静态派发
        }
    }
}
pub struct Button {
    pub width: u32,
    pub height: u32,
    pub label: String,
}
impl Draw for Button {
    fn draw(&self) {
        // 实际绘制按钮的代码
    }
}
use gui::Draw;
struct SelectBox {
    width: u32,
    height: u32,
    options: Vec<String>,
}
impl Draw for SelectBox {
    fn draw(&self) {
        // code to actually draw a select box
    }
}
use gui::{Screen, Button};
fn main() {
    let screen = Screen {
        components: vec![
            Box::new(SelectBox {//类型检测，实现了 Draw 的类型
                width: 75,
                height: 10,
                options: vec![
                    String::from("Yes"),
                    String::from("Maybe"),
                    String::from("No")
                ],
```

```
            }),
            Box::new(Button {
                width: 50,
                height: 10,
                label: String::from("OK"),
            }),
        ],
    };

    screen.run();
}
```

面向对象（3）：trait 对象动态分发与对象安全

动态分发：

单态化所产生的代码进行静态分发（static dispatch）。静态分发发生于编译器在编译时就知晓调用了什么方法的时候。这与 动态分发 （dynamic dispatch）相对，这时编译器在编译时无法知晓调用了什么方法。在动态分发的情况下，编译器会生成在运行时确定调用了什么方法的代码。

当使用 trait 对象时，Rust 必须使用动态分发。编译器无法知晓所有可能用于 trait 对象代码的类型，所以它也不知道应该调用哪个类型的哪个方法实现。为此，Rust 在运行时使用 trait 对象中的指针来知晓需要调用哪个方法。动态分发也阻止编译器有选择的内联方法代码，这会相应的禁用一些优化。

对象安全：

Trait 对象要求对象安全。只有 对象安全（object safe）的 trait 才可以组成 trait 对象。如果一个 trait 中所有的方法有如下属性时，则该 trait 是对象安全的：

返回值类型不为 Self

方法没有任何泛型类型参数

Self 关键字是我们要实现 trait 或方法的类型的别名。对象安全对于 trait 对象是必须的，因为一旦有了 trait 对象，就不再知晓实现该 trait 的具体类型是什么了。如果 trait 方法返回具体的 Self 类型，但是 trait 对象忘记了其真正的类型，那么方法不可能使用已经忘却的原始具体类型。同理对于泛型类型参数来说，当使用 trait 时其会放入具体的类型参数：此具体类型变成了实现该 trait 的类型的一部分。当使用 trait 对象时其具体类型被抹去了，故无从得知放入泛型参数类型的类型是什么。

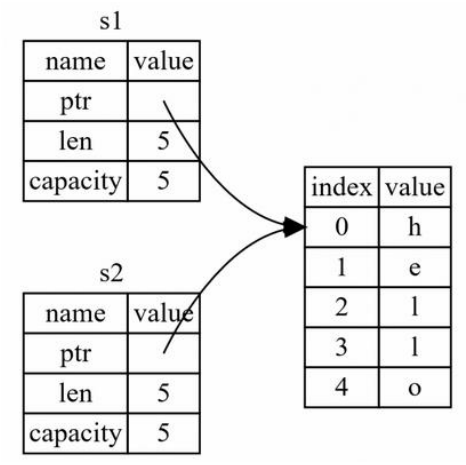
一个 trait 的方法不是对象安全的例子是标准库中的 Clone trait。Clone trait 的 clone 方法的参数签名看起来像这样：

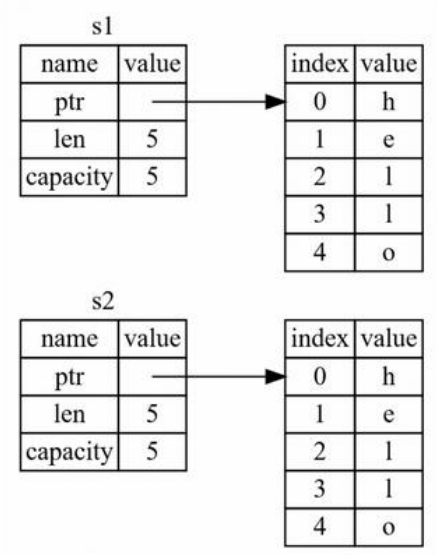
```
pub trait Clone {
    fn clone(&self) -> Self;
}
pub struct Screen {
    pub components: Vec<Box<dyn Clone>>,//err
}
```

所有权

（1）
所有权（ownership）是 Rust 最独特的功能，让 Rust 无需垃圾回收（garbage collector）即可保障内存安全
垃圾回收，在程序运行时不断地寻找不再使用的内存；
亲自分配和释放内存。
Rust ：通过所有权系统管理内存，编译器在编译时会根据一系列的规则进行检查。在运行时，所有权系统的任何功能都不会减慢程序。内存在拥有它的变量离开作用域后就被自动释放
栈与堆：内存回收
所有权系统：跟踪哪部分代码正在使用堆上的哪些数据，最大限度的减少堆上的重复数据的数量，以及清理堆上不再使用的数据确保不会耗尽空间。所有权的存在就是为了管理堆数据。
所有权规则：
Rust 中的每一个值都有一个被称为其所有者（owner）的变量。
值有且只有一个所有者。
当所有者（变量）离开作用域，这个值将被丢弃。

```
{//一个作用域
    let s = String::from("hello"); // 从此处起，s 是有效的
    ...// 使用 s
} // 此作用域已结束，s 不再有效，调用 drop 释放内存

//拷贝，固定大小，存放在栈上
let x = 5;//5 绑定到 x 上
let y = x;//创建 x 的拷贝，绑定到 y 上，固定大小，栈上

//移动 move（浅拷贝），堆上
let s1 = String::from("hello");
let s2 = s1;
println!("{}, world!", s1);
//s1 在栈上，s2 只从栈上拷贝了它的指针、长度和容量
//s1 失效，s1 被移动到了 s2 中，避免了浅拷贝中的二次释放

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





所有权（2）：Copy Trait

像整型这样的在编译时已知大小的类型被整个存储在栈上，所以拷贝其实际的值是快速的。只在栈上的数据：拷贝。如果一个类型拥有 Copy trait，一个旧的变量在将其赋值给其他变量后仍然可用。

Rust 不允许自身或其任何部分实现了 Drop trait 的类型使用 Copy trait。

作为一个通用的规则，任何简单标量值的组合可以是 Copy 的，不需要分配内存或某种形式资源的类型是 Copy 的。如下是一些 Copy 的类型：

所有整数类型，比如 u32。

布尔类型，bool，它的值是 true 和 false。

所有浮点数类型，比如 f64。

字符类型，char。

元组，当且仅当其包含的类型也都是 Copy 的时候。比如，(i32, i32) 是 Copy 的，但 (i32, String) 就不是。

所有权（3）：传参与返回值

```
fn main() {
    let s = String::from("hello");   // s 进入作用域
    takes_ownership(s);   // s 的值移动到函数里 ...
    // ... 所以到这里不再有效

    let x = 5;                    // x 进入作用域
    makes_copy(x); // x 应该移动函数里，
                                  // 但 i32 是 Copy 的，所以在后面可继续使用 x
} // 这里, x 先移出了作用域，然后是 s。但因为 s 的值已被移走，
  // 所以不会有特殊操作

fn takes_ownership(some_string: String) {
    // some_string 进入作用域
    println!("{}", some_string);
} // 这里，some_string 移出作用域并调用 `drop` 方法。占用的内存被释放

fn makes_copy(some_integer: i32) {
    // some_integer 进入作用域
    println!("{}", some_integer);
} // 这里，some_integer 移出作用域。不会有特殊操作
//返回值也可以转移所有权
fn main() {

    let s1 = gives_ownership();          // gives_ownership 将返回值移给 s1
    let s2 = String::from("hello");      // s2 进入作用域
    // s2 被移动到 akes_and_gives_back 中,  它也将返回值移给 s3
    let s3 = takes_and_gives_back(s2);

} // 这里, s3 移出作用域并被丢弃。s2 也移出作用域，但已被移走，
  // 所以什么也不会发生。s1 移出作用域并被丢弃

fn gives_ownership() -> String {   // gives_ownership 将返回值移动给调用它的函数
    let some_string = String::from("hello"); // some_string 进入作用域.
    some_string                                  // 返回 some_string 并移出给调用的函
数
}

// takes_and_gives_back 将传入字符串并返回该值
fn takes_and_gives_back(a_string: String) -> String { // a_string 进入作用域
    a_string   // 返回 a_string 并移出给调用的函数
}
```

变量的所有权总是遵循相同的模式：将值赋给另一个变量时移动它。当持有堆中数据值的变量离开作用域时，其值将通过 drop 被清理掉，除非数据被移动为另一个变量所有。
所有权（4）：引用和借用
借用：
以一个对象的引用作为参数而不是获取值的所有权

```
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()
}
```

我们将获取引用作为函数参数称为借用（borrowing），归还
可变引用：

```
fn main() {
    let mut s = String::from("hello");
    change(&mut s);
}
fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

限制：在特定作用域中的特定数据有且只有一个可变引用

```
let mut s = String::from("hello");
let   r1 = &mut s; //
let r2 = &mut s; //err
```

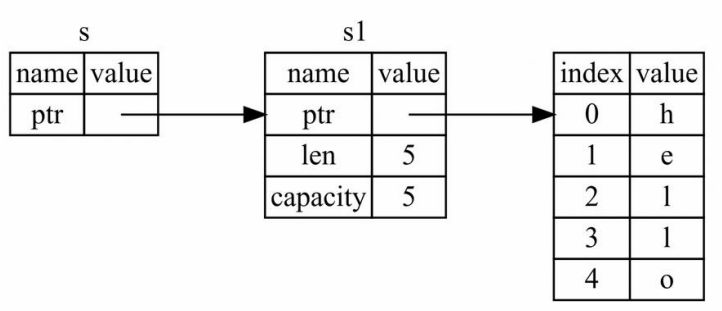
在编译时就避免数据竞争。数据竞争（data race）类似于竞态条件，它可由这三个行为造成：
    两个或更多指针同时访问同一数据。
    至少有一个指针被用来写入数据。
    没有同步数据访问的机制。

```
let mut s = String::from("hello");
let r1 = &s; // no problem
let r2 = &s; // no problem
let r3 = &mut s; // err，不能同时拥有不变引用与可变引用
```

悬垂引用：在 Rust 中编译器确保引用永远也不会变成悬垂状态：当你拥有一些数据的引用，编译器确保数据不会在其引用之前离开作用域。

```
fn main() {
    let reference_to_nothing = dangle();
}
fn dangle() -> &String { //err
   // dangle 返回一个字符串的引用
```

```
    let s = String::from("hello");
   // s 是一个新字符串
    &s // 返回字符串 s 的引用
} // 这里 s 离开作用域并被丢弃。其内存被释放。危险！

//ok
fn no_dangle() -> String {
    let s = String::from("hello");
    s
}
```



所有权（5）：&str 字符串切片

字符串切片 slice 的类型声明写作 &str

```
let s = "Hello, world!";// 字符串字面值就是 slice, s 的类型是 &str
fn first_word(s: &String) -> &str {
    let bytes = s.as_bytes();//获取字节数组
    //iter 创建迭代器返回结合中每一个元素
    //enumerate 将 iter 每个输出作为元素封装在元组中，第一个为索引
    //第二个为指向集合中字节的引用
    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }
    &s[..]
}
    let mut s = String::from("hello world");
    let word = first_word(&s);//s 的不可变引用
    s.clear(); // s 的可变引用，error!
//字符串切片做参数更加通用，可以对 String 值和 &str 值使用相同的函数
fn first_word(s: &str) -> &str {}
let s1 = String::from("hello world");
first_word(&s1[..]);
let s2 = "hello world";
```

first_word(s2);

字符串切片指向 String 某个连续部分的引用

```
let s = String::from("hello world");
let hello = &s[0..5];
let world = &s[6..11];
```

start..end 语法代表一个以 start 开头并一直持续到但不包含 end 的 range。如果需要包含 end，可以使用 ..= 而不是 ..。

如果想要从第一个索引（0）开始，可以不写两个点号之前的值。如果 slice 包含 String 的最后一个字节，也可以舍弃尾部的数字。也可以同时舍弃这两个值来获取整个字符串的 slice。

```
let s = String::from("hello");
let slice = &s[3..];

let s = String::from("hello");
let slice = &s[..];
```

字符串切片必须切在有效的 utf8 字符位置

其它切片：数组，动态数组等

生命周期

（1）：借用检查器

Copy trait 拷贝，Drop trait 移动，引用与借用

Rust 中的每一个引用都有其生命周期（lifetime），也就是引用保持有效的作用域。大部分时候生命周期是隐含并可以推断，有时候需要使用泛型生命周期参数来注明他们的关系，手动标注生命周期。生命周期是 Rust 最与众不同的功能

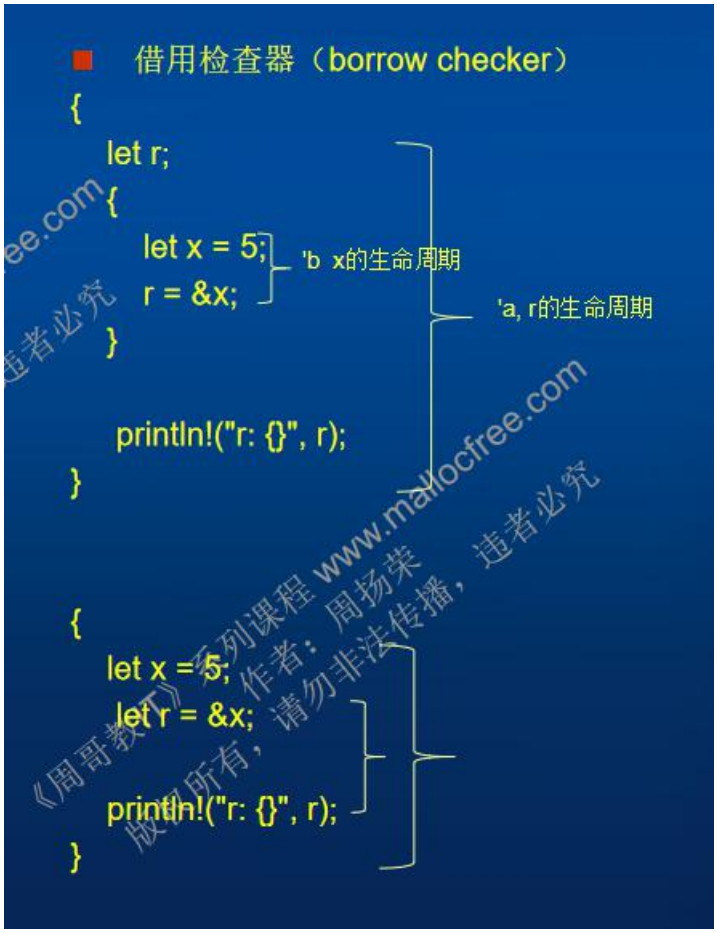
生命周期的主要目标：避免悬垂引用

```
{
    let r;
    {
        let x = 5;
        r = &x;
    }
    println!("r: {}", r);//不能编译
}
```

借用检查器（borrow checker）

```
{
    let r;
    {
        let x = 5;
        r = &x;
```

```
    }

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



{
    let x = 5;
     let r = &x;

    println!("r: {}", r);
}
```



生命周期（2）：函数中的泛型生命周期

//返回值需要一个泛型生命周期参数，不知道 x 和 y 的生命周期是如何与返回值的生命周期相关联

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
```

```
    } else {
        y
    }
}
```

生命周期注解语法

```
&i32          // 引用
&'a i32       // 带有显式生命周期的引用
&'a mut i32 // 带有显式生命周期的可变引用
```

泛型生命周期参数需要声明在函数名和参数列表间的尖括号中。这里我们想要告诉 Rust 关于参数中的引用和返回值之间的限制是他们都必须拥有相同的生命周期

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

在函数签名中指定生命周期参数时，并没有改变任何传入后返回的值的生命周期。而是指出任何不遵守这个协议的传入值都将被借用检查器拒绝。这些注解出现在函数签名中，而不存在于函数体中的任何代码中。这是因为 Rust 能够分析函数中代码而不需要任何协助，不过当函数引用或被函数之外的代码引用时，让 Rust 自身分析出参数或返回值的生命周期几乎是不可能的。这些生命周期在每次函数被调用时都可能不同。这也就是为什么我们需要手动标记生命周期

```
{
    let string1 = String::from("long string is long");
    {
        let string2 = String::from("xyz");
        let result = longest(string1.as_str(), string2.as_str());
        println!("The longest string is {}", result);//ok
    }
}
{
    let string1 = String::from("long string is long");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
    println!("The longest string is {}", result);//err
}
```

通过生命周期参数告诉 Rust ： longest 函数返回的引用的生命周期应该与传入参数的生命

周期中较短那个保持一致。因此，借用检查器不允许示例中的代码，因为它可能会存在无效的引用。

```
fn longest<'a>(x: &'a str, y: &str) -> &'a str {//y 不需要标注
    x
}
```

当从函数返回一个引用，返回值的生命周期参数需要与一个参数的生命周期参数相匹配。如果返回的引用 没有 指向任何一个参数，那么唯一的可能就是它指向一个函数内部创建的值，它将会是一个悬垂引用

```
fn longest<'a>(x: &str, y: &str) -> &'a str {
    let result = String::from("really long string");
    result.as_str()
}
```

生命周期（3）：结构体定义中的生命周期标注

包含引用的结构体

```
struct ImportantExcerpt<'a> {//摘要
    part: &'a str,//ImportantExcerpt 的实例不能比其 part 字段中的引用存在的更久。
}
fn main() {
    let novel = String::from("Call me bluepill. Some years ago...");
    let first_sentence = novel.split('.')
        .next()
        .expect("Could not find a '.'");
    let i = ImportantExcerpt { part: first_sentence };
}
```

生命周期（4）：生命周期省略

生命周期省略（Lifetime Elision）：函数或方法的参数的生命周期被称为 输入生命周期（input lifetimes），而返回值的生命周期被称为 输出生命周期（output lifetimes）。RUST 编译器通过如下 3 条规则来计算函数签名中引用的生命周期：

第一条规则：每一个是引用的参数都有它自己的生命周期参数。有一个引用参数的函数就有一个生命周期参数：fn foo<'a>(x: &'a i32)，有两个引用参数的函数有两个不同的生命周期参数，fn foo<'a, 'b>(x: &'a i32, y: &'b i32)，依此类推。

第二条规则：如果只有一个输入生命周期参数，那么它被赋予所有输出生命周期参数：fn foo<'a>(x: &'a i32) -> &'a i32。

第三条规则：如果方法有多个输入生命周期参数，不过其中之一因为方法的缘故为 &self 或 &mut self，那么 self 的生命周期被赋给所有输出生命周期参数。通过此规则可以使得大部分方法签名中省略生命周期标注。

如果编译器检查完这三条规则后仍然存在没有计算出生命周期的引用，编译器将会停止并生成错误

```
fn first_word(s: &str) -> &str {}//无须标注生命周期
fn longest(x: &str, y: &str) -> &str {}//无法计算签名中所有引用的生命周期
```

生命周期（5）：方法定义中的生命周期标注

第三条规则：如果方法有多个输入生命周期参数，不过其中之一因为方法的缘故为 &self 或 &mut self，那么 self 的生命周期被赋给所有输出生命周期参数。通过此规则可以使得大部分方法签名中省略生命周期标注。

```
struct ImportantExcerpt<'a> {
        part: &'a str,
 }

impl<'a> ImportantExcerpt<'a> {//声明在 impl 及类型名称之后的生命周期不能省略
      fn level(&self) -> i32 {//不用为 self 引用标注生命周期
            3
      }

      fn announce_and_return_part(&self, announcement: &str) -> &str {//规则 1 和规则 3
            println!("Attention please: {}", announcement);
            self.part
      }

}
```

生命周期（6）：静态生命周期

静态生命周期

'static，其生命周期存活于整个程序期间。所有的字符串字面值都拥有 'static 生命周期

```
let s: &'static str = "I have a static lifetime.";
```

生命周期（7）：泛型参数、trait bounds 和生命周期

结合泛型类型参数、trait bounds 和生命周期

```
use std::fmt::Display;

fn longest_with_an_announcement<'a, T>(x: &'a str, y: &'a str, ann: T) -> &'a str
      where T: Display
{
      println!("Announcement! {}", ann);
      if x.len() > y.len() {
            x
      } else {
            y
      }
}
```

生命周期（8）:生命周期子类型

生命周期子类型（lifetime subtyping）功能，这是一个指定一个生命周期不会短于另一个的方法。在声明生命周期参数的尖括号中，可以照常声明一个生命周期 'a，并通过语法 'b: 'a 声明一个不短于 'a 的生命周期 'b。

```
struct Context<'s>(&'s str);
struct Parser<'c, 's> {
	context: &'c Context<'s>,
}
struct Parser<'c, 's:'c> {
	context: &'c Context<'s>,
}
impl<'c, 's> Parser<'c, 's> {
	fn parse(&self) -> Result<(), &'s str> {
		Err(&self.context.0[1..])
	}
}
fn parse_context(context: Context) -> Result<(), &str> {
	Parser { context: &context }.parse()
}
```

生命周期（9）:trait 对象生命周期的推断

```
trait Red { }

struct Ball<'a> {
	diameter: &'a i32,
}

impl<'a> Red for Ball<'a> { }

fn main() {
	let num = 5;

	let obj = Box::new(Ball { diameter: &num }) as Box<dyn Red>;//没有为 obj 指定生命周期
}
```

生命周期与 trait 对象必须遵守的规则：

trait 对象的默认生命周期是 'static。
如果有 &'a X 或 &'a mut X，则默认生命周期是 'a。
如果只有 T: 'a 从句， 则默认生命周期是 'a。
如果有多个类似 T: 'a 的从句，则没有默认生命周期；必须明确指定。

当必须明确指定时，可以为像 Box<Red> 这样的 trait 对象增加生命周期 bound，根据需要使用语法 Box<Foo + 'a> 或 Box<Foo + 'static>。正如其他的 bound，这意味着任何 Red trait 的实现如果在内部包含有引用，这些引用就必须拥有与 trait 对象 bound 中所指定的相同

的生命周期。

生命周期（10）：匿名生命周期

```
struct StrWrap<'a>(&'a str);

fn foo<'a>(string: &'a str) -> StrWrap<'a> {
    StrWrap(string)
}

fn foo(string: &str) -> StrWrap<'_> {// StrWrap 包含一个引用，不过无需所有的生命周期注解来知道
    StrWrap(string)
}
// 冗余
impl<'a> fmt::Debug for StrWrap<'a> {}

// 省略
impl fmt::Debug for StrWrap<'_> { }
```

swap 函数

```
fn swap<'a>(mut x:&'a mut i32, mut y:&'a mut i32){
    println!("x:{:p}, y:{:p}", x, y);
    let temp:&mut i32 = x;
    x = y;
    y = temp;
    println!("x:{:p}, y:{:p}", x, y);
}

fn main() {
    let mut x:i32 = 5;
    let mut y:i32 = 10;

    swap(&mut x, &mut y);
    println!("x:{}, y:{}", x, y);
}
```

函数式编程

（1）：闭包

函数式编程风格通常包含将函数作为参数、其他函数的返回值、将函数赋值给变量以供之后执行等等

闭包（Closures）：一个可以储存在变量里的类似函数的结构，可以捕获环境的匿名函数

```
fn    add_one_v1      (x: u32) -> u32 { x + 1 }
let add_one_v2 = |x: u32| -> u32 { x + 1 };
let add_one_v3 = |x|                    { x + 1 };//类型推断
let add_one_v4 = |x|                      x + 1    ;
```

闭包会捕获其环境：
闭包周围的作用域被称为其环境，environment

```
fn main() {
    let x = 4;
    let equal_to_x = |z| z == x;
    let y = 4;
    assert!(equal_to_x(y));
}
```

FnOnce， FnMut，Fn
当闭包从环境中捕获一个值，闭包会在闭包体中储存这个值以供使用。这会使用内存并产生额外的开销。
闭包可以通过三种方式捕获其环境，他们直接对应函数的三种获取参数的方式：获取所有权，可变借用和不可变借用。这三种捕获值的方式被编码为如下三个 Fn trait：

FnOnce 消费从周围作用域捕获的变量。为了消费捕获到的变量，闭包必须获取其所有权并在定义闭包时将其移动进闭包。其名称的 Once 部分代表了闭包不能多次获取相同变量的所有权的事实，所以它只能被调用一次。

FnMut 获取可变的借用值所以可以改变其环境

Fn 从其环境获取不可变的借用值

当创建一个闭包时，Rust 根据其如何使用环境中变量来推断我们希望如何引用环境。由于所有闭包都可以被调用至少一次，所以所有闭包都实现了 FnOnce 。那些并没有移动被捕获变量的所有权到闭包内的闭包也实现了 FnMut ，而不需要对被捕获的变量进行可变访问的闭包则也实现了 Fn。
move
如果希望强制闭包获取其使用的环境值的所有权，可以在参数列表前使用 move 关键字。这在将闭包传递给新线程以便将数据移动到新线程中时最为实用

```
    let x = vec![1, 2, 3];
    let equal_to_x = move |z| z == x;
    println!("can't use x here: {:?}", x);
    let y=vec![1,2,3];
    assert!(equal_to_x(y));
```

函数式编程（2）：返回闭包
返回闭包：Box<dyn Fn(i32)->i32>

```
fn returns_closure() -> Fn(i32) -> i32 {//err，无法知道多大的空间存储返回的闭包
    |x| x + 1
}
```

Rust 并不知道需要多少空间来储存闭包。可以使用 trait 对象：

```
fn returns_closure() -> Box<dyn Fn(i32) -> i32> {
    Box::new(|x| x + 1)
}
```

函数式编程（3）：迭代器

迭代器（iterator）：一种处理元素序列的方式，负责遍历序列中的每一项和决定序列何时结束的逻辑

```
let v1 = vec![1, 2, 3];
let v1_iter = v1.iter();
for val in v1_iter {//获得了 v1_iter 所有权
    println!("Got: {}", val);
}
```

迭代器都实现了一个叫做 Iterator 的定义于标准库的 trait

```
trait Iterator {
    type Item;//关联类型，next 返回的类型
    fn next(&mut self) -> Option<Self::Item>;
    // 此处省略了方法的默认实现
}
```

next ： Iterator 实现者被要求定义的唯一方法。next 一次返回迭代器中的一个项，封装在 Some 中，当迭代器结束时，它返回 None

```
let v1 = vec![1, 2, 3];
let mut v1_iter = v1.iter();
//iter：next 获取的是 vector 元素的不变引用
//into_iter：获得 v 所有权，并返回元素本身
// iter_mut，获得元素的可变引用
assert_eq!(v1_iter.next(), Some(&1));
```

sum 方法：调用 next 方法的方法被称为 消费适配器（consuming adaptors），因为调用他们会消耗迭代器。一个消费适配器的例子是 sum 方法。这个方法获取迭代器的所有权并反复调用 next 来遍历迭代器，因而会消费迭代器。当其遍历每一个项时，它将每一个项加总到一个总和并在迭代完成时返回总和。

```
let v1 = vec![1, 2, 3];
let v1_iter = v1.iter();
let total: i32 = v1_iter.sum();
```

map 方法：使用闭包来调用每个元素以生成新的迭代器。 collect 方法消费迭代器并将结果收集到一个数据结构中。

```
let v1: Vec<i32> = vec![1, 2, 3];
let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();
assert_eq!(v2, vec![2, 3, 4]);
```

filter 方法：使用返回布尔值的闭包来过滤迭代器中的每一个项。如果闭包返回 true，其值将会包含在 filter 提供的新迭代器中。如果闭包返回 false，其值不会包含在结果迭代器中。

```
let v1: Vec<i32> = vec![1, 2, 3];
let v2: Vec<_> = v1.iter().filter(|x| *x%2==0).collect();
assert_eq!(v2, vec![2]);
```

函数式编程（4）:实现 Iterator trait

```
struct Counter {
	count: u32,
}
impl Counter {
	fn new() -> Counter {
		Counter { count: 0 }
	}
}
impl Iterator for Counter {
	type Item = u32;
	fn next(&mut self) -> Option<Self::Item> {
		self.count += 1;
		if self.count < 6 {
			Some(self.count)
		} else {
			None
		}
	}
}
let sum: u32 = Counter::new().zip(Counter::new().skip(1))//迭代器合并，生产元组迭代器
						.map(|(a, b)| a * b)
						.filter(|x| x % 3 == 0)
						.sum();//(1,2) (2,3)(3,4)(4,5) (5, None)
```

指定了 next 方法如何工作，而标准库则提供了其它调用 next 的方法的默认实现

关联类型（associated types）是一个将类型占位符与 trait 相关联的方式，这样 trait 的方法签名中就可以使用这些占位符类型。trait 的实现者会针对特定的实现在这个类型的位置指定相应的具体类型。如此可以定义一个使用多种类型的 trait，直到实现此 trait 时都无需知道这些类型具体是什么。

```
pub trait Iterator {
	type Item;

	fn next(&mut self) -> Option<Self::Item>;
}

impl Iterator for Counter {
	type Item = u32;
```

```
    fn next(&mut self) -> Option<Self::Item> {
        // --snip--
}
```

函数式编程（5）:迭代器与循环性能分析

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
    }

    results
}

pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    contents.lines()
        .filter(|line| line.contains(query))
        .collect()
}
```

通过使用迭代器适配器方法来编写更简明的代码

性能：迭代器版本还要稍微快一点

迭代器，作为一个高级的抽象，被编译成了与手写的底层代码大体一致性能代码。迭代器是 Rust 的 零成本抽象（zero-cost abstractions）之一，它意味着抽象并不会引入运行时开销

智能指针

（1）:基本概念

智能指针是 Rust 中一种特殊的数据结构。它与普通指针的本质区别在于普通指针(引用）是对值的借用，而智能指针通常拥有对数据的所有权。如果想要在堆内存中定义一个对象，并不是像 Java 中那样直接 new 一个，也不是像 C 语言中那样需要手动 malloc 函数来分配内存空间。Rust 中使用的是 Box::new 等来对数据进行封装。

智能指针通常使用结构体实现。智能指针区别于常规结构体的显著特性在于其实现了 Deref 和 Drop trait。实现 Deref 可以使智能指针能够解引用，而实现 Drop 则使智能指针具有自动析构的能力。

使用 Box 在堆上储存数据

```
fn main() {
```

```
    let a = 10;
    let x = &a;
    println!("x:{}, *x={}", x, *x);

    let b = Box::new(5);
    println!("b = {}, *b={}", b, *b);
}
```

Deref 有一个特性是强制隐式转换

```
let x = Box::new("hello");
println!("{:?}", x.chars());//Box 里没有 chars()，x 自动解引用
```

Rust 标准库中提供的智能指针有：Box<T>、Rc<T>(Arc<T>)、Ref<T>、RefCell<T>等等

智能指针（2）：Box

Box 类型是一个智能指针，因为它实现了 Deref trait，它允许 Box 值被当作引用对待。当 Box 值离开作用域时，由于 Box 类型 Drop trait 的实现，box 所指向的堆数据也会被清除。

使用 Box 在堆上储存数据

```
fn main() {
    let b = Box::new(5);
    println!("b = {}", b);
}
```

单一所有权，即同时只能有一个人拥有对其指向数据的所有权，并且同时只能存在一个可变引用或多个不可变引用

```
let a = Box::new(1);    // Immutable
println!("{}", a);       // Output: 1

let mut b = Box::new(1);    // Mutable
*b += 1;
println!("{}", b);       // Output: 2

let a = Box::new(1);    // Owned by a
let b = a;    // Now owned by b
println!("{}", a);    // Error: value borrowed here after move

let c = &mut a;
let d = &a;
```

println!("{}, {}", c, d); // Error: cannot borrow `a` as immutable because it is also borrowed as mutable

Box 允许创建递归类型。一种无法在编译时知道大小的类型，其值的一部分可以是相同类型的另一个值。这种值的嵌套，理论上可以无限的进行下去，所以 Rust 不知道递归类型需要多少空间。不过 box 有一个已知的大小，所以通过在循环类型定义中插入 box，就可以创建递归类型了

链接列表(cons list)：两个参数组成一个二元组，二元组由一个值和另一个二元组组成，不断嵌套最终组成一个列表。最后一项为 Nil，作为终止标记。

```
enum List {
	Cons(i32, List),//err
	Nil,
}
use crate::List::{Cons, Nil};
fn main() {
	let list = Cons(1, Cons(2, Cons(3, Nil)));
}
enum List {
	Cons(i32, Box<List>),
	Nil,
}
use crate::List::{Cons, Nil};
fn main() {
	let list = Cons(1,
		Box::new(Cons(2,
			Box::new(Cons(3,
				Box::new(Nil))))));
}
```

智能指针（3）：deref 与 drop

解引用：

```
struct MyBox<T>(T);
impl<T> MyBox<T> {
	fn new(x: T) -> MyBox<T> {
		MyBox(x)
	}
}

pub fn new(x: T) -> Box<T> {
   box x
}
```

有一个 box 关键字，这个关键字是用来进行堆内存分配的，它只能在 Rust 源码内部使用。box 关键字会调用 Rust 内部的 exchange_malloc 和 box_free 方法来管理内存

```
impl<T> Deref for MyBox<T> {
	type Target = T;//关联类型
	fn deref(&self) -> &T {
		&self.0//deref 方法体中写入了 &self.0，这样 deref 返回了我希望通过 * 运算符访问的值的引用
```

```
    }
}
```

输入 *y 时,等同于*(y.deref()), Rust 将 * 运算符替换为先调用 deref 方法再进行直接引用的操作

函数和方法的隐式解引用强制多态。特定类型的引用作为实参传递给和形参类型不同的函数或方法时，解引用强制多态将自动发生。这时会有一系列的 deref 方法被调用，把我们提供的类型转换成了参数所需的类型

```
fn hello(name: &str) {
    println!("Hello, {}!", name);
}
fn main() {
    let m = MyBox::new(String::from("Rust"));
    hello(&m);
}
```

Rust 可以通过 deref 调用将 &MyBox<String> --> &String-->再次调用 deref --> &str，这就符合 hello 函数的定义了

使用 Deref trait 重载不可变引用的 * 运算符，Rust 提供了 DerefMut trait 用于重载可变引用的 * 运算符。

```
struct CustomSmartPointer {
    data: String,
}
impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        println!("Dropping CustomSmartPointer with data `{}`!", self.data);
    }
}
fn main() {
    let c = CustomSmartPointer { data: String::from("my stuff") };
    let d = CustomSmartPointer { data: String::from("other stuff") };
   std::mem::drop(c);
    println!("CustomSmartPointers created.");
}
```

Rust 并不允许我们主动调用 Drop trait 的 drop 方法；当我们希望在作用域结束之前就强制释放变量的话，我们应该使用的是由标准库提供的 std::mem::drop

智能指针（4）：RC

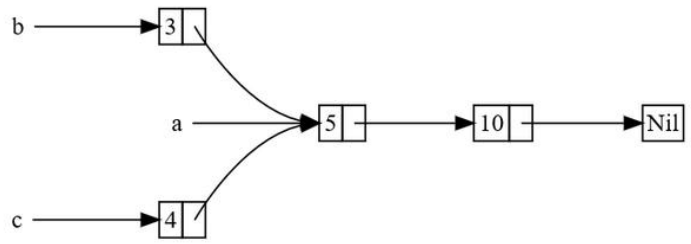
基本使用方法：有些情况单个值可能会有多个所有者，为了启用多所有权，Rust 有一个叫做 Rc 的类型。其名称为 引用计数（reference counting）。Rc::strong_count(&a)显示引用计数个数。Rc 允许通过不可变引用来只读的在程序的多个部分共享数据。每次调用 Rc::clone（或 a.clone()），Rc 中数据的引用计数都会增加

```
    let a = Rc::new(1);
```

```
    println!("count after creating a = {}", Rc::strong_count(&a));
    let b = Rc::clone(&a);
    println!("count after creating b = {}", Rc::strong_count(&a));
    {
        let c = Rc::clone(&a);
        println!("count after creating c = {}", Rc::strong_count(&a));
    }
    println!("count after c goes out of scope = {}", Rc::strong_count(&a));
```

共享链接列表：

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use crate::List::{Cons, Nil};
fn main() {
    let a = Cons(5,
        Box::new(Cons(10,
            Box::new(Nil))));
    let b = Cons(3, Box::new(a));//a 列表所有权移动到 b
    let c = Cons(4, Box::new(a));//a 已经被移走了，错误
}
enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));//a 为指向列表（5,10,Nil）的智能指针
    let b = Cons(3, Rc::clone(&a));//克隆智能指针 a，共享 a 中数据，引用计数变为 2
    let c = Cons(4, Rc::clone(&a));//同上，引用计数变为 3，只有当引用计数变为 0，a 中的数据才会清理
}
```



注意事项：

Rc<T> 是完全不可变的，可以将其理解为对同一内存上的数据同时存在的多个只读指针。

其次，Rc<T> 是只适用于单线程内的，尽管从概念上讲不同线程间的只读指针是完全安全的，但由于 Rc<T> 没有实现在多个线程间保证计数一致性

智能指针（5）：Cell<T>

通常情况下，我们只能修改一个人的名称或者年龄，而不能修改用户的 id。如果我们把 User 的实例设置成了可变状态，那就不能保证别人不会去修改 id。

```
struct User {
    id: i32,
    name: str,
    age: u8,
}
let mut user = ....;
```

特定情况下在值的方法内部能够修改自身是很有用的；而在其他代码中，此时值仍然是不可变。值方法外部的代码不能修改其值。为了应对这种情况，Rust 为我们提供了 Cell<T>和 RefCell<T>。提供内部可变性的容器。

一般来说，当有一个不可变值时，不能可变的借用它。例如，如下代码不能编译：

```
let x = 5;
let y = &mut x;
```

Cell：

```
let x = Cell::new(1);//x 对外部代码，是不可变值
let y = &x;//y 是对 x 的引用
 let z = &x;
 x.set(2); //内部可变
 y.set(3);
 z.set(4);
 println!("{}", x.get());
use std::cell::Cell;
struct Foo {
    x: u32,
    y: Cell<u32>,
}
```

```
fn main() {
    let foo = Foo { x: 1, y: Cell::new(3)};
    assert_eq!(1, foo.x);
    assert_eq!(3, foo.y.get());
    foo.y.set(5);
    assert_eq!(5, foo.y.get());
}
```

Cell<T> 允许多个共享引用对其内部值进行更改，实现了内部可变性。

Cell<T> 对 T 的限制：只能作用于实现了 Copy 的类型，几乎等同于会分配在栈上的数据（可以直接按比特进行连续 n 个长度的复制），所以对其随意进行改写是十分安全的，不会存在堆数据泄露的风险。

智能指针（6）：RefCell<T>

对于没有实现 Copy 的类型，使用 Cell<T>还是比较不方便的，Rust 还提供了 RefCell<T>

```
use std::cell::RefCell;
fn main() {
    let x = RefCell::new(vec![1, 2, 3]);
    println!("{:?}", x.borrow());
    x.borrow_mut().push(5);
    println!("{:?}", x.borrow());
}
```

RefCell<T> 实现了运行时借用，这个借用是临时的，而且 Rust 的 Runtime 也会随时紧盯 RefCell<T> 的借用行为：同时只能有一个可变借用存在，否则直接 Painc。也就是说 RefCell<T> 不会像常规时一样在编译阶段检查引用借用的安全性，而是在程序运行时动态的检查，从而提供在不安全的行为下出现一定的安全场景的可行性。

RefCell 正是用于当你确信代码遵守借用规则，而编译器不能理解和确定的时候。

当创建不可变和可变引用时，我们分别使用 & 和 &mut 语法。对于 RefCell 来说，则是 borrow 和 borrow_mut 方法，这属于 RefCell 安全 API 的一部分。borrow 方法返回 Ref 类型的智能指针，borrow_mut 方法返回 RefMut 类型的智能指针。这两个类型都实现了 Deref 所以可以当作常规引用对待。RefCell 记录当前有多少个活动的 Ref 和 RefMut 智能指针。每次调用 borrow，RefCell 将活动的不可变借用计数加一。当 Ref 值离开作用域时，不可变借用计数减一。就像编译时借用规则一样，RefCell 在任何时候只允许有多个不可变借用或一个可变借用。如果我们尝试违反这些规则，相比引用时的编译时错误，RefCell 的实现会在运行时 panic!

```
    thread::spawn(move || {
        let c = RefCell::new(5);
        let m = c.borrow();
        let b = c.borrow_mut();
    }).join();
```

类似于 Rc，RefCell 只能用于单线程场景。如果尝试在多线程上下文中使用 RefCell，会得到一个编译错误。 Box，Rc 或 RefCell 的区别：

Rc 允许相同数据有多个所有者；Box 和 RefCell 有单一所有者。

Box 允许在编译时执行不可变或可变借用检查；Rc 仅允许在编译时执行不可变借用检查；RefCell 允许在运行时执行不可变或可变借用检查。

因为 RefCell 允许在运行时执行可变借用检查，所以我们可以在即便 RefCell 自身是不可变的情况下修改其内部的值。

智能指针（7）：Rc 和 RefCell 合用

实现这样一个需求：一份数据同时存在多个不同所有者，但每个所有者又可以随时修改其内容，且这个内容类型 T 没有实现 Copy 的情况。通过 Rc<T> 保证了多所有权，而通过 RefCell<T> 则保证了内部数据的可变性

结合 Rc 和 RefCell 来拥有多个可变数据所有者

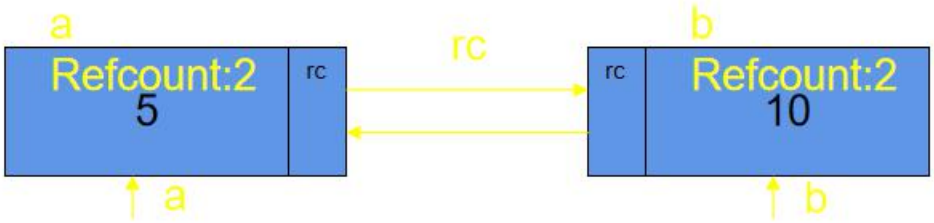
```
let value = Rc::new(RefCell::new(5));
let value2 = Rc::clone(&value);
*value.borrow_mut() += 10;
println!("{}", value);

    let shared_vec: Rc<RefCell<_>> = Rc::new(RefCell::new(Vec::new()));
    // Output: []
    println!("{:?}", shared_vec.borrow());
    {
        let b = Rc::clone(&shared_vec);
        b.borrow_mut().push(1);
        b.borrow_mut().push(2);
    }
    shared_vec.borrow_mut().push(3);
    // Output: [1, 2, 3]
    println!("{:?}", shared_vec.borrow());
```

智能指针（8）：循环引用

Rust 并不保证完全地避免内存泄漏。这一点可以通过 Rc 和 RefCell 看出：有可能会创建个个项之间相互引用的引用。这会造成内存泄漏，因为每一项的引用计数将永远也到不了 0，其值也永远也不会被丢弃。

```
use std::rc::Rc;
use std::cell::RefCell;
use crate::List::{Cons, Nil};
enum List {
    Cons(i32, RefCell<Rc<List>>),
    Nil,
}
impl List {
    fn tail(&self) -> Option<&RefCell<Rc<List>>> {
        match self {
```

```
                Cons(_, item) => Some(item),
                Nil => None,
            }
        }
}
fn main() {
    let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));
    println!("a initial rc count = {}", Rc::strong_count(&a));
    println!("a next item = {:?}", a.tail());
    let b = Rc::new(Cons(10,
        RefCell::new(Rc::clone(&a))));
    println!("a rc count after b creation = {}",    Rc::strong_count(&a));
    println!("b initial rc count = {}", Rc::strong_count(&b));
    println!("b next item = {:?}", b.tail());
    if let Some(link) = a.tail() {
        *link.borrow_mut() = Rc::clone(&b);
    }
    println!("b rc count after changing a = {}", Rc::strong_count(&b));
    println!("a rc count after changing a = {}",    Rc::strong_count(&a));
    // 取消如下行的注释来观察引用循环；
    // 这会导致栈溢出
    // println!("a next item = {:?}", a.tail());
}
```



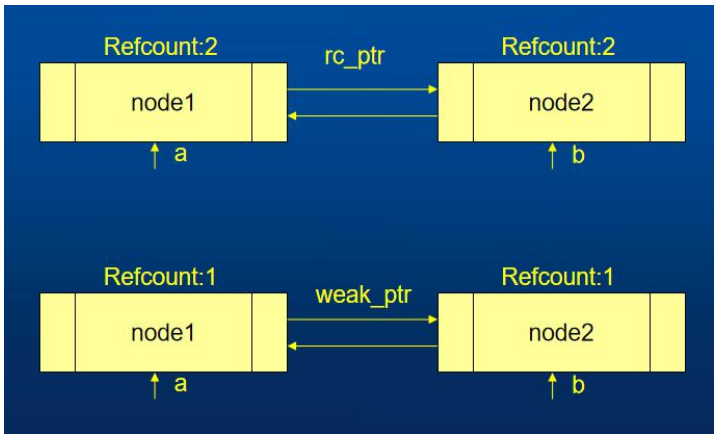
智能指针（9）：Weak

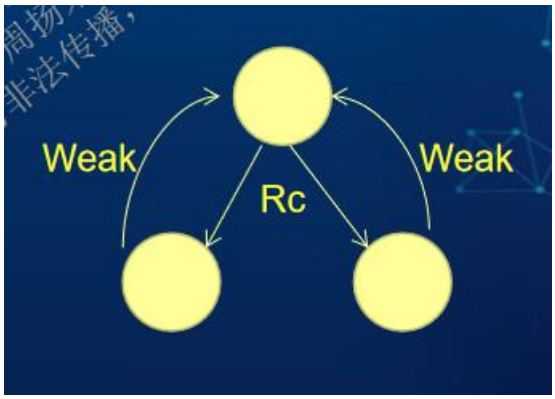
避免引用
循环：将 Rc 变为 Weak
Rc 类型使用 weak_count 来记录其存在多少个 Weak 引用，类似于 strong_count。其区别在于 weak_count 无需计数为 0 就能使 Rc 实例被清理。
强引用代表如何共享 Rc 实例的所有权。弱引用并不代表所有权关系。他们不会造成引用循环，因为任何引入了弱引用的循环一旦所涉及的强引用计数为 0 就会被打破。
调用 Rc::downgrade 时会得到 RC 实例的 Weak 类型的智能指针
调用 Weak 实例的 upgrade 方法，这会返回 Option<Rc>。如果 Rc 值还未被丢弃则结果是

Some，如果 Rc 已经被丢弃则结果是 None

父结点应该拥有其子结点：如果父结点被丢弃了，其子结点也应该被丢弃。然而子结点不应该拥有其父结点：如果丢弃子结点，其父结点应该依然存在。这正是弱引用的例子

```
use std::rc::{Rc, Weak};
use std::cell::RefCell;
struct Node {
    value: i32,
    parent: RefCell<Weak<Node>>,
    children: RefCell<Vec<Rc<Node>>>,
}
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });
    println!("leaf parent = {:?}",
        leaf.parent.borrow().upgrade());//None
    let branch = Rc::new(Node {
        value: 5,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });
    *leaf.parent.borrow_mut() =
        Rc::downgrade(&branch);
    println!("leaf parent = {:?}",leaf.parent.borrow().upgrade());
}
Rc::strong_count(&leaf)
Rc::weak_count(&leaf)
```





并发编程

（1）:线程创建

并发编程（Concurrent programming），代表程序的不同部分相互独立的执行，而并行编程（parallel programming）代表程序不同部分于同时执行。

无畏并发（fearless concurrency）：通过改进所有权和类型检查，Rust 很多并发错误都是 编译时 错误，而非运行时错误。因此，相比花费大量时间尝试重现运行时并发 bug 出现的特定情况，Rust 会拒绝编译不正确的代码并提供解释问题的错误信息

编程语言提供的线程被称为 绿色（green）线程（协程），使用绿色线程的语言会在不同数量的 OS 线程的上下文中执行它们。为此，绿色线程模式被称为 M:N 模型：M 个绿色线程对应 N 个 OS 线程(_beginthreadex(), pthread_create())，这里 M 和 N 不必相同。M：N 模型需要较大的运行时来管理线程。运行时代表二进制文件中包含的由语言自身提供的代码。这些代码根据语言的不同可大可小，不过任何非汇编语言都会有一定数量的运行时代码。Rust 需要做到几乎没有运行时，同时为了保持高性能必需能够调用 C 语言。

Rust 标准库只提供了 1:1 线程模型实现。Rust 是足够底层的语言，所以由相应的 crate 实现了 M:N 线程模型

```
use std::thread;
use std::time::Duration;
fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });
    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
```

```
    handle.join().unwrap();//注意这个 join
}
```

move 闭包，经常与 thread::spawn 一起使用，因为它允许我们在一个线程中使用另一个线程的数据。

```
use std::thread;
fn main() {
    let v = vec![1, 2, 3];
    let handle = thread::spawn(move || {
        println!("Here's a vector: {:?}", v);
    });
    handle.join().unwrap();
}
//增加 move 关键字，我们强制闭包获取其使用的值的所有权
```

并发编程会带来相应的安全问题：共享资源竞争，死锁等。

并发编程（2）：消息传递与通道

线程之间需要通信。消息传递保证并发安全和通信越来越主流。GO：不要共享内存来通讯；而是要通讯来共享内存。

rust 中一个实现消息传递并发的主要工具是 通道（channel），类似于河流小溪

channel 中的 send 函数获取其参数的所有权并移动这个值归接收者所有。这个意味着不可能在发送后再次使用这个值。

```
use std::thread;
use std::sync::mpsc;
//Multi-producer, single-consumer FIFO
fn main() {
    //创建一个通道，两端赋值给 tx 和 rx
    //transmitter, receiver
    let (tx, rx) = mpsc::channel();
    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
        //println!("val is {}", val);//err
    });
    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
use std::thread;
use std::sync::mpsc;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();
```

```
    thread::spawn(move || {
        let vals = vec![
            String::from("1"),
            String::from("2"),
            String::from("3"),
            String::from("4"),
        ];
        for val in vals {
            tx.send(val).unwrap();//批量发送信息
            thread::sleep(Duration::from_secs(1));
        }
    });
    for received in rx {
        println!("Got: {}", received);
    }
}
let (tx, rx) = mpsc::channel();
let tx1 = mpsc::Sender::clone(&tx);//克隆多个生产者
thread::spawn(move || {//线程 1
    let vals = vec![
        String::from("hi from 1"),
        String::from("hello from 1"),
    ];
    for val in vals {
        tx1.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
});
thread::spawn(move || {//线程 2
    let vals = vec![
        String::from("hi from 2"),
        String::from("hello from 2"),
     ];
    for val in vals {
        tx.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
});
for received in rx {
    println!("Got: {}", received);
```

```
}
```

并发编程（3）：数据共享与 Mutex

在线程间数据共享 Mutex：Mutex 是一个智能指针。lock 调用 返回 一个叫做 MutexGuard 的智能指针。这个智能指针实现了 Deref 来指向其内部数据；其也提供了一个 Drop 实现当 MutexGuard 离开作用域时自动释放锁

```
use std::sync::Mutex;
use std::thread;
fn main() {
    let counter = Mutex::new(0);
    let mut handles = vec![];
    for _ in 0..10 {
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();//err，不能将 counter 的所有权移动到多个线程中

            *num += 1;
        });
        handles.push(handle);
    }
    for handle in handles {
        handle.join().unwrap();
    }
    println!("Result: {}", *counter.lock().unwrap());
}
```

通过使用智能指针 Rc 来创建引用计数的值，以便拥有多所有者。

```
use std::rc::Rc;
use std::sync::Mutex;
use std::thread;
fn main() {
    let counter = Rc::new(Mutex::new(0));//err，Rc 不能安全的在线程间共享。当 Rc 在每一个 clone 调用时增加计数，并在每一个克隆被丢弃时减少计数。Rc 并没有使用任何并发原语，来确保改变计数的操作不会被其他线程打断。Rc 不满足 send 约束
    let mut handles = vec![];
    for _ in 0..10 {
        let counter = Rc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();
            *num += 1;
        });
        handles.push(handle);
    }
```

```
    for handle in handles {
        handle.join().unwrap();
    }
    println!("Result: {}", *counter.lock().unwrap());
}
```

Arc 可以安全的用于并发环境的类型。原子性（atomic），所以这是一个原子引用计数（atomically reference counted）类型

```
use std::sync::{Mutex, Arc};
use std::thread;
fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];
    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();
            *num += 1;
        });
        handles.push(handle);
    }
    for handle in handles {
        handle.join().unwrap();
    }
    println!("Result: {}", *counter.lock().unwrap());
}
```

counter 是不可变的，不过可以获取其内部值的可变引用；这意味着 Mutex 提供了内部可变性

并发编程（4）：send 和 sync

Send trait

类型的所有权可以在线程间传递。几乎所有的 Rust 类型都是 Send 的，不过有一些例外，包括 Rc：这是不能 Send 的，因为如果克隆了 Rc 的值并尝试将克隆的所有权转移到另一个线程，这两个线程都可能同时更新引用计数。为此，Rc 被实现为用于单线程场景，这时不需要为拥有线程安全的引用计数而付出性能代价。

Rust 类型系统和 trait bound 确保永远也不会意外的将不安全的 Rc 在线程间发送(编译错误）

任何完全由 Send 的类型组成的类型也会自动被标记为 Send。几乎所有基本类型都是 Send 的，除了裸指针（raw pointer）。

Sync trait

表明一个实现了 Sync 的类型可以安全的在多个线程中拥有其值的引用。换一种方式来说，对于任意类型 T，如果 &T（T 的引用）是 Send 的话 ，T 就是 Sync 的，这意味着其引用就可以安全的发送到另一个线程

基本类型是 Sync 的，完全由 Sync 的类型组成的类型也是 Sync 的。
Rc 也不是 Sync 的，出于其不是 Send 相同的原因。RefCell 和 Cell 系列类型不是 Sync 的。RefCell 在运行时所进行的借用检查也不是线程安全的。Mutex 是 Sync 的，它可以被用来在多线程中共享访问。
具体的类型：
原始类型（比如： u8, f64），都是 Sync，都是 Copy，因此都是 Send；
只包含原始类型的复合类型，都是 Sync，都是 Copy，因此都是 Send；
当 T: Sync，Box<T>, Vec<T> 等集合类型是 Sync；
具有内部可变性的的指针，不是 Sync 的，比如 Cell, RefCell, UnsafeCell；
Rc 不是 Sync。因为克隆一个新引用，它会以非原子性的方式修改引用计数，所以是不安全的；
被 Mutex 和 RWLock 锁住的类型 T: Send，是 Sync 的；
原始指针既不是 Send 也不是 Sync；
通常并不需要手动实现 Send 和 Sync trait，因为由 Send 和 Sync 的类型组成的类型，自动就是 Send 和 Sync 的。手动实现 Send 和 Sync 是不安全的
Rust 正是通过这两大武器：所有权和生命周期 + Send 和 Sync（本质上为类型系统）来为并发编程提供了安全可靠的基础设施。使得程序员可以放心在其上构建稳健的并发模型

不安全 rust

（1）裸指针和不安全函数
代码都有 Rust 在编译时会强制执行的内存安全保证。然而，Rust 还隐藏有第二种语言，它不会强制执行这类内存安全保证：这被称为 不安全 Rust（unsafe Rust）
静态分析本质上保守，有时代码可能是合法的，但是 Rust 不这么认为！在这种情况下，可以使用不安全代码告诉编译器，相信我，我知道我在干什么。这么做的缺点就是你只能靠自己
底层计算机硬件固有的不安全性。如果 Rust 不允许进行不安全操作，那么有些任务则根本完成不了。Rust 需要能直接与操作系统交互，底层系统编程。在 rust 里，要求这四类操作必须位于标记为 unsafe 的块中
解引用裸指针
调用不安全的函数或方法
访问或修改可变静态变量
实现不安全 trait
unsafe 并不会关闭借用检查器或禁用任何其他 Rust 安全检查：如果在不安全代码中使用引用，其仍会被检查。unsafe 关键字只是提供了那四个不会被编译器检查内存安全的功能。你仍然能在不安全块中获得某种程度的安全。
裸指针：是可变或不可变的，分别写作 *const T 和 *mut T，允许忽略借用规则，可以同时拥有不可变和可变的指针，或多个指向相同位置的可变指针
不保证指向有效的内存
允许为空
不能实现任何自动清理功能

用 as 将不可变和可变引用强转为对应的裸指针类型

```
let mut num = 5;
let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;
unsafe {
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", *r2);
}
```

可以在安全代码中 创建 裸指针，只是不能在不安全块之外 解引用 裸指针

创建了同时指向相同内存位置 num 的裸指针 *const i32 和 *mut i32。相反如果尝试创建 num 的不可变和可变引用，这将无法编译因为 Rust 的所有权规则不允许拥有可变引用的同时拥有不可变引用。通过裸指针，就能够同时创建同一地址的可变指针和不可变指针，若通过可变指针修改数据，则可能潜在造成数据竞争

不能确定其有效性的裸指针

```
let address = 0x012345usize;
let r = address as *const i32;
```

一个主要的应用场景便是调用 C 代码接口

另一个场景是构建借用检查器无法理解的安全抽象

调用不安全函数或方法：通过在 unsafe 块中调用不安全函数，我们表明已经阅读过此函数的文档并对其是否满足函数自身的契约负责。

```
unsafe fn dangerous() {}

unsafe {
    dangerous();
}
```

不安全 rust（2）：安全抽象

创建不安全代码的安全抽象：仅仅因为函数包含不安全代码并不意味着整个函数都需要标记为不安全的。事实上，将不安全代码封装进安全函数是一个常见的抽象

Rust 的借用检查器不能理解我们要借用这个 slice 的两个不同部分：它只知道我们借用了同一个 slice 两次。本质上借用 slice 的不同部分是可以的，因为结果两个 slice 不会重叠，不过 Rust 还没有智能到能够理解这些。当我们知道某些事是可以的而 Rust 不知道的时候，就是触及不安全代码的时候了

```
let mut v = vec![1, 2, 3, 4, 5, 6];
let r = &mut v[..];
let (a, b) = r.split_at_mut(3);
assert_eq!(a, &mut [1, 2, 3]);
assert_eq!(b, &mut [4, 5, 6]);

fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
    let len = slice.len();
```

```
    assert!(mid <= len);
    (&mut slice[..mid],
     &mut slice[mid..])
}
//cannot borrow `*slice` as mutable more than once at a time
use std::slice;
fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
    let len = slice.len();
    let ptr = slice.as_mut_ptr();
    assert!(mid <= len);
    unsafe {
        (slice::from_raw_parts_mut(ptr, mid),
         slice::from_raw_parts_mut(ptr.offset(mid as isize), len - mid))
    }
}
```

slice::from_raw_parts_mut 函数是不安全的因为它获取一个裸指针，并必须确信这个指针是有效的。裸指针上的 offset 方法也是不安全的，因为其必须确信此地址偏移量也是有效的指针。因此必须将 slice::from_raw_parts_mut 和 offset 放入 unsafe 块中以便能调用它们。通过观察代码，和增加 mid 必然小于等于 len 的断言，我们可以说 unsafe 块中所有的裸指针将是有效的 slice 中数据的指针。这是一个可以接受的 unsafe 的恰当用法。无需将 split_at_mut 函数的结果标记为 unsafe，并可以在安全 Rust 中调用此函数。我们创建了一个不安全代码的安全抽象，其代码以一种安全的方式使用了 unsafe 代码，因为其只从这个函数访问的数据中创建了有效的指针

不安全 rust（3）：extern 调用 C 函数

使用 extern 函数调用外部代码：有助于创建和使用 外部函数接口（Foreign Function Interface， FFI）。在 extern 块中声明的函数在 Rust 代码中总是不安全的。因为其他语言不会强制执行 Rust 的规则且 Rust 无法检查它们，所以确保其安全是程序员的责任

```
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    unsafe {
        println!("Absolute value : {}", abs(-3));
    }
}
```

"C" 部分定义了外部函数所使用的 应用程序接口（application binary interface，ABI）。ABI 定义了如何在汇编语言层面调用此函数。"C" ABI 是最常见的，并遵循 C 编程语言的 ABI。

Rust 程序可以编译为动态库并从 C 语言中链接，call_from_c 函数就能够在 C 代码中访问

```
#[no_mangle]
pub extern "C" fn call_from_c() {
    println!("Just called a Rust function from C!");
}
```

不安全 rust（4）：静态变量和不安全 trait

全局变量在 Rust 中被称为 静态（static）变量

```
static HELLO_WORLD: &str = "Hello, world!";//'static, 访问不可变静态变量是安全的

fn main() {
    println!("name is: {}", HELLO_WORLD);
}
```

常量与不可变静态变量可能看起来很类似，不过一个微妙的区别是静态变量中的值有一个固定的内存地址。使用这个值总是会访问相同的地址。另一方面，常量则允许在任何被用到的时候复制其数据。

常量与静态变量的另一个区别在于静态变量可以是可变的。访问和修改可变静态变量都是不安全 的

```
static mut COUNTER: u32 = 0;

fn add_to_count(inc: u32) {
    unsafe {
        COUNTER += inc;
    }
}

fn main() {
    add_to_count(3);

    unsafe {
        println!("COUNTER: {}", COUNTER);
    }
}
```

实现不安全 trait

```
unsafe trait Foo {
    // methods go here
}

unsafe impl Foo for i32 {
    // method implementations go here
```

```
}
```

当至少有一个方法中包含编译器不能验证的不安全因素时，这个 trait 就是不安全的

网络编程

服务端：

```
    let listener = TcpListener::bind("127.0.0.1:8080")?;
    let mut thread_vec: Vec = Vec::new();
    for stream in listener.incoming() {
        let stream = stream.expect("failed!");
        let handle = thread::spawn(move || {
            handle_client(stream).unwrap_or_else(|error| eprintln!("{:?}", error));
        });
        thread_vec.push(handle);
    }
    for handle in thread_vec {
        handle.join().unwrap();
    }
fn handle_client(mut stream: TcpStream) -> Result<(), Box<dyn Error>{
    let mut buf = [0; 512];
    for _ in 0..1000 {
        let bytes_read = stream.read(&mut buf)?;
        if bytes_read == 0 {
            return Ok(());
        }
        stream.write(&buf[..bytes_read])?;
        thread::sleep(time::Duration::from_secs(1 as u64));
    }
    Ok(())
}
```

客户端：

```
    let mut stream = TcpStream::connect("127.0.0.1:8080")?;
    for _ in 0..10 {

        let mut input = String::new();
        io::stdin().read_line(&mut input).expect("Failed to read from stdin");

        stream.write(input.as_bytes()).expect("Failed to write to stream");

        let mut reader = BufReader::new(&stream);
```

```
        let mut buffer: Vec = Vec::new();
        reader.read_until(b'\n', &mut buffer).expect("Could not read into buffer");

        println!("{}",str::from_utf8(&buffer).expect("Could not write buffer as string"));
        println!("");

    }
```

//简单的通信模型，async/.await　Future 异步编程，或者线程池等技术

数据库编程

库：
[dependencies]
mysql = "*"
Blocking waiting for file lock on package cache
DSN:

```
  let url = "mysql://root:mfroot911@localhost/mfwebsecdb";
  let pool = Pool::new(url)?;
  let mut conn = pool.get_conn()?;
```

query /// Performs text query and collects the first result set.
query_first /// Performs text query and returns the firt row of the first result set.
query_map/// Performs text query and maps each row of the first result set.
query_drop　/// Performs text query and drops the query result.

exec_batch/// Prepares the given statement, and executes it with each item in the given params iterator.
exec /// Executes the given `stmt` and collects the first result set.
exec_first　/// Executes the given `stmt` and returns the first row of the first result set.exec_first
exec_map　/// Executes the given `stmt` and maps each row of the first result set.
exec_drop /// Executes the given `stmt` and drops the result.
事务：

```
    let txopts = TxOpts::default();
    txopts.set_isolation_level(Some(IsolationLevel::RepeatableRead));
    let mut tx = conn.start_transaction(txopts)?;

    tx.query_drop("CREATE TEMPORARY TABLE tmp (a TEXT)")?;
    tx.exec_drop("INSERT INTO tmp (a) VALUES (?)", ("foo",))?;
    let val: Option<String> = tx.query_first("SELECT a from tmp")?;
```

```
    assert_eq!(val.unwrap(), "foo");

    // Note, that transaction will be rolled back implicitly on Drop, if not committed.

    //tx.rollback()?;
    tx.commit()?;
```

json 支持

JSON(JavaScript Object Notation) 是一种轻量级的数据交换（程序与程序之间，网络之间）格式。已经是行业标准，得到各种语言的广泛支持。http://www.json.org/

Key-value 结构存储的字符串

{"name":"tom", "age":30,"marriage":false}

[{"a":0},{"b":1},{"c":2},{"d":3}]

序列化与反序列化：

把 rust 对象转化为 json 字符串，叫序列化

stu:=Student{"tom",30,false} "{\"name\":\"tom\", \"age\":30,\"marriage\":false}"

把 json 字符串转化为 rust 对象，叫反序列化

"{\"name\":\"tom\", \"age\":30,\"marriage\":false}" Student{"tom",30,false}

[dependencies]

rustc-serialize = "0.3.18"

装 Rust 之后，会在你的用户目录之下生成一个.cargo 文件夹，进入这个文件夹

在.cargo 文件夹下，创建一个 config 文件

[registry]

index = "git://crates.mirrors.ustc.edu.cn/index"

```
// 引入 rustc_serialize 模块
use rustc_serialize::json;
// Automatically generate `RustcDecodable` and `RustcEncodable` trait
// implementations
// 定义 TestStruct
#[derive(RustcDecodable, RustcEncodable)]
pub struct TestStruct  {
    data_int: u8,
    data_str: String,
    data_vector: Vec<u8>,
}
fn main() {
    // 初始化 TestStruct
    let object = TestStruct {
        data_int: 1,
```

```
            data_str: "homura".to_string(),
            data_vector: vec![2,3,4,5],
        };

        // Serialize using `json::encode`
        // 将 TestStruct 转意为字符串
        let encoded = json::encode(&object).unwrap();
        println!("{}",encoded);
        // Deserialize using `json::decode`
        // 将 json 字符串中的数据转化成 TestStruct 对应的数据，相当于初始化
        let decoded: TestStruct = json::decode(&encoded).unwrap();
        println!("{:?}",decoded.data_vector);
}
serde_json
serde_json
use serde::{Deserialize, Serialize};
use serde_json::{Result};
struct User {
   name: String,
   age: u8,
   blog: String,
   addr: String,
}
let u: User = serde_json::from_str(json).unwrap();
let json_str = serde_json::to_string(&u).unwrap();

[dependencies]
serde_json = "1.0"
serde = { version = "1.0", features = ["derive"] }
```

redis

redis 是一个 key-value 存储系统（NoSQL 数据库）。它支持存储的 value 类型包括 string、hash、list、set（元素不能重复）、zset(sorted set 有序集合)等。这些类型都具有丰富的操作命令用于存取数据，而且这些操作都是原子性的。与 memcached 一样，为了保证效率，数据都是缓存在内存中。区别的是 redis 会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了 master-slave(主从)同步。(暗示了奴隶制度，被迫改名，master-replica)

性能极高 – Redis 能读的速度是 110,000 次/s,写的速度是 81,000 次/s ，后台开发必备

http://redis.io/

http://redis.io/commands

http://www.runoob.com/redis/redis-commands.html

安装 redis：
https://redis.io/
https://github.com/microsoftarchive/redis/releases/tag/win-3.2.100
cd /d C:\Program Files\Redis
redis-server.exe redis.windows.conf
redis-cli.exe -h 127.0.0.1 -p 6379 -a mf123456
config set stop-writes-on-bgsave-error no
Redis is configured to save RDB snapshots, but is currently not able to persist on disk.

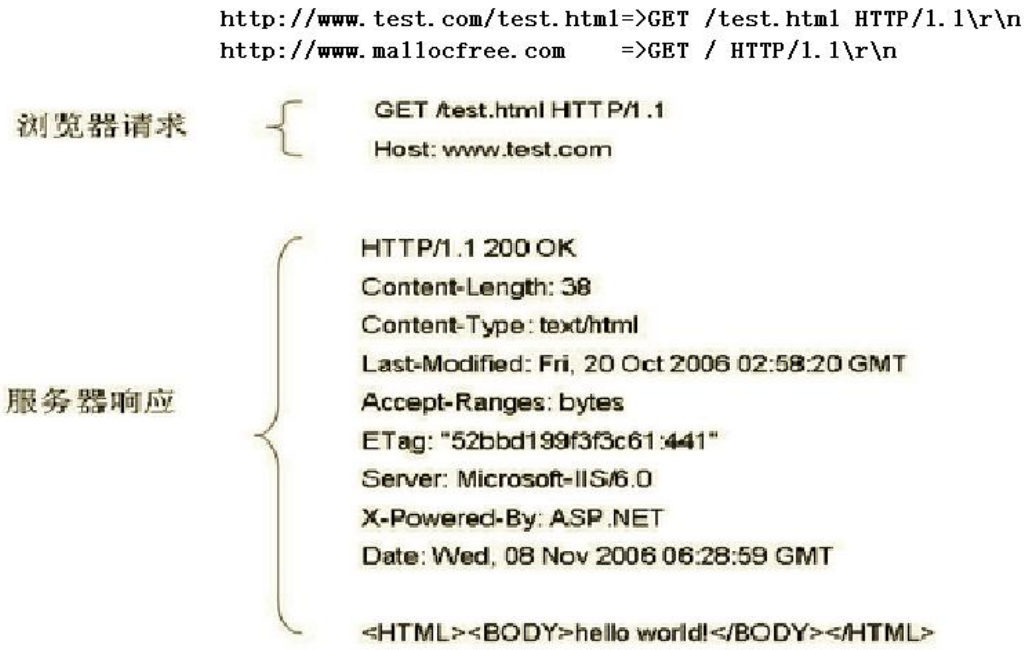
```
[dependencies]
simple_redis = "0.3.44"
  let mut clients = simple_redis::create("redis://:mf123456@127.0.0.1:6379")?;
    clients.set("mf_key", "mallocfree.com")?;
    clients.set("mf_num", 1999_i32)?;

    let str_val = clients.get::<String>("mf_key")?;
    let num_val = clients.get::<i32>("mf_num")?;

    println!("strval:{}, numval:{}", str_val, num_val);

    clients.lpush("mf_list", "l1")?;
    clients.rpush("mf_list", "r1")?;
    clients.lpushx("mf_list", "lx")?;//key should exist
    clients.rpushx("mf_list", "rx")?;

    let list:Vec<String> = clients.lrange("mf_list", 0, -1)?;
    let val = clients.lpop::<String>("mf_list")?;
    let val2 = clients.rpop::<String>("mf_list")?;

    println!("list:{:?}, val:{}, val2:{}", list, val, val2);
    clients.hset("mf_hash", "a", 1)?;
    clients.hsetnx("mf_hash", "b", 1)?;//only if field does not yet exist

    let vali32 = clients.hget::<i32>("mf_hash", "a")?;    // 1      i32 类型
    let valstring = clients.hget_string("mf_hash", "a")?;    // "1"      String 类型
    let valhashmap = clients.hgetall("mf_hash")?;    // HashMap<String,String>类型

    println!("vali32:{}, valstring:{}, valhashmap:{:?}", vali32, valstring, valhashmap);
```

也可以直接使用 redis-rs。

[dependencies]
redis = "0.12.0"

实战篇-web 服务器

实战篇-web 服务器（1）单线程版本



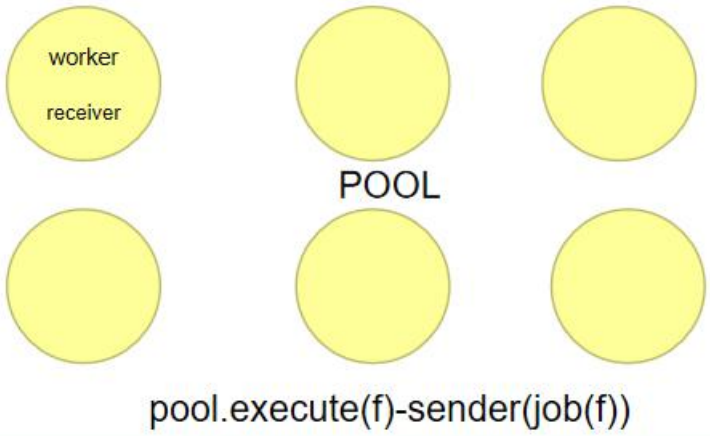

```
let listener = TcpListener::bind("127.0.0.1:8080").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        handle_connection(stream);
    }

fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 512];
    stream.read(&mut buffer).unwrap();

    let get = b"GET / HTTP/1.1\r\n";

    let (status_line, filename) = if buffer.starts_with(get) {
        ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
```

```
    } else {
        ("HTTP/1.1 404 NOT FOUND\r\n\r\n", "404.html")
    };

    let contents = fs::read_to_string(filename).unwrap();

    let response = format!("{}{}", status_line, contents);

    stream.write(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}
```

实战篇-web 服务器（2）多线程版本

```
    let listener = TcpListener::bind("127.0.0.1:8080").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        thread::spawn(|| {
            handle_connection(stream);
        });
    }
```

实战篇-web 服务器（3）线程池版本

```
let listener = TcpListener::bind("127.0.0.1:8080").unwrap();
let pool = ThreadPool::new(4);

for stream in listener.incoming().take(2) {
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
}
pub struct ThreadPool {
    workers: Vec<Worker>, //线程池
    sender: mpsc::Sender<Message>,
```

```
}

struct Worker {
    id: usize,
    thread: Option<thread::JoinHandle<()>>,
}
```