Rust权威指南

【美】Steve Klabnik Carol Nichols 摘录

环境搭建：
pycharm搜不到rust插件，在vscode里面有

一些cmd命令：
`cargo build` 编译
`cargo run` 编译加运行
`cargo check` 检查是否能通过编译
`cargo build --release` 生成的可执行文件会被放置在`target/release`目录下
`cargo doc --open` 很神奇，可以打开本项目的说明文档，或者参阅包的官方文档

Rust的特点：
变量都是默认不可变的；自动进行类型推导的能力；是一门静态类型语言，在编译程序的过程中需要知道所有变量的具体类型，当其无法推导出类型时会报错；将表达式和语句区分为两个不同的东西，语句偏向指令，无返回值，而表达式偏向于能产生结果（Rust中，x = y = 6; 是错误的）；
再例如：

```
let y = {
        let x = 3;
        x + 1
}; 其中{}及以内也是表达式
```

无垃圾回收机制（GC），通过所有权来保障内存安全；

变量与常量的区别：
常量总是需要标注类型，声明必须用const, 而不是let，总是不变的
例如：`const MAX_NUM: u32 = 100_000;`

一些概念：
`&mut guess` 来声明一个可变引用
通过`Cargo.lock`文件确保我们的构建可重现
类型，变体，match表达式，绑定（不一定就是存储了值的地址，只是相关联而已），命名空间，二次释放，类型系统
硬编码：即使用常量值，在大型项目中需要避免
包package, 单元包crate, 模块module, 路径path
模块系统用于管理作用域

隐藏：
通过let对一个变量进行变换操作，并保持其自身的不变性。
不同于
`let x = 3;`
`x = 5;`
这是直接赋值，会导致报错。
名字复用，而类型可变，这与let mut也有很大区别：
`let mut x = " ";`
`x = x.len();`
这会报错。

数据类型：
两大类：标量类型（scalar) 和符合类型（compound), 均将数据存储在栈上
scalar: 整数(u16, i16, u32)，浮点（f64, f32)，字符 (只能用单引号），布尔值 (true, false)
Rust的char类型占4个字节
compound：元组，数组，可被看作一个单独的符合元素

元组：
模式匹配 获得元组的单个值，也称之为： 解构
`let tup: (i32, f64, u8) = (500, 1.1, 1);`
`(x, y, z) = tup;`
也可索引取值：
`y = tup.1;`
`(1..4)` 表示元组 `(1,2,3)`

数组：
固定元素个数，每个元素类型必须相同
动态数组vector
`let a: [i32; 5] = [1,2,3,4,5];`
`let x = a[1];`
遍历：`for element in a.iter() {}`

函数：
函数名命名规则：蛇形（num_game）
不顾函数定义的位置，只要在可见区域即可
函数隐式返回：可以不用return语句。
必须标明返回值类型。

```
fn main() {
    let x = plus_one(5);
    println!("the value of x is: {}", x);
}
```

```
fn plus_one(x:i32) -> i32 {
    x + 1         //此处若添加semicolon，  则报错found () 表示返回的是一个空元组
}
```

注释：
只能是 //

控制流：
分支：
if 表达式必须依据显式的bool值，而不能把非零值视为bool值
if-else if-else与C格式相同，区别：单条语句依然需要加上{}
`let number = if cond {5} else {6};` 这表明：代码块输出的值就是其中最后一个表达式的值；Rust在编译时就必须确定number的类型，故两个分支的数据类型必须保持一致，否则报错。
循环：
有loop, while 和 for
`let result = loop {break 2;};`
`while codition {}`
for循环的安全性：避免index错误引用

所有权：
Rust语言的核心功能；涉及到堆栈；所有存储在栈中的数据都必须拥有一个已知的固定的大小，对于编译期间无法确定大小的数据，只能把它们存储在堆中，在堆中叫“分配”，同时将得到的空间地址的指针存储在栈中。所有这些堆上操作均可由所有权来解决。
访问堆上的数据要慢于栈
所有权规则：

> Rust中的每一个值都有一个对应的变量作为它的所有者。
> 在同一时间内，值有且仅有一个所有者。
> 当所有者离开自己的作用域时，它持有的值就会被释放掉。

String类型存储在堆上。字符串字面量类型和String类型是不同的两个类型, String:: from()做了申请内存的事情
栈上数据s2 = s1, 是深拷贝；堆上数据是浅拷贝，即指向的数据内容是不变的；但是与浅度拷贝不同的是，s1 被弃用（为解决二次释放的问题），如：

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

会报错，因此更贴切的叫法为：`移动`， 深拷贝：`let s2 = s1.clone();`
所有权与函数：
参数传递和返回值会触发移动或者复制
当一个持有堆数据的变量离开作用域时，它的数据就会被drop清理回收，除非已被`move`
为继续使用形参，引入了`引用`这个类型概念
引用：
`&`保证你在不获取所有权的情况下使用值，（相当于s2.ptr 指向s1.addr，而s1.ptr 指向data）
`*` 解引用
这种通过引用传递参数给函数的方法也被称为`借用`
引用默认是不可变的，Rust不允许我们去修改引用指向的值
可变引用：
加入mut修饰符。但是可变引用是有限制的。对于特定作用域中的特定数据来说，一次只能声明一个可变引用。即`let s1 = &mut s; let s2 = &mut s;` 会报错的。可避免data race，存在数据竞争的代码连编译检查都无法通过，并且难以debug； 但是`let s1 = &s; let s2 = &s`是可行的。
可变引用一改全改
不能在存在不可变引用的情况下创建可变引用；即在任一段时间内，要么只拥有一个可变引用，要么只能拥有任意数量的不可变引用。
引用总是有效的，若通过了编译。编译能确保它不出现悬垂指针

一种名为Copy的trait，用于栈上数据的复制，在其被赋值给其他变量之后依然保持可用性。一般来说，任何需要分配内存和资源的类型都不会是Copy的。

切片：
`let s = "hello"` 其中`s`的类型为`&str`
字符串字面量就是切片
str 与String的区别：
没有str数据类型，只有&str, &str不可变，不拥有数据的所有权

```
// 创建一个 String
let mut hello = String::from("Hello");
// 修改 String
hello.push_str(", world!");
// 将 String 转换为 &str
let greeting = &hello[..];
// 创建一个 &str
let greeting_literal = "Hello, world!";
// 从 String 到 &str 的转换
let string_slice: &str = hello.as_str();
```

```
// 从 &str 到 String 的转换
let owned_string: String = greeting_literal.to_string();
```

对String切片的引用也是`&str`类型

结构体：
结构体与元组的区别：结构体每个字段都必须要有明确的名字。
结构体的实例一旦可变，则各自段皆可变
元组结构体：只标明类型
空结构体：只想在某个类型上实现一个trait， 而不存储数据时使用
所有权：一种是所有字段持有所有权，还有一种指定生命周期的引用
结构体还有一个语义作用：表示相关联，增加有意义的描述信息
可以使用调试模式`#[derive(Debug)]`打印结构体

方法：
方法总是定义在某个结构体，枚举类型，trait对象上下文中，第一个参数总是self，代指自身实例，比函数更加有助于组织代码结构
自动调用解引用，即：object.something() 等价于 (&object).something() 或者 `(*object).something()`
方法放在结构体的impl块中，一个结构体可以有多个impl块
关联函数：结构体中的无需self参数的方法，不作用于某个具体的结构体实例，常用作构造器
例如`Rectangle::square(3);`其中`::`不仅被用于关联函数，还用于模块创建的命名空间

枚举与模式匹配：
枚举类型的值只能是变体中的一个成员，变体可以是不同类型数据
枚举变体：所有可能的值；依变体创建实例
可以将数据附加到枚举的每个变体中，这样便不需要额外使用结构体
枚举的变体中还可以嵌入另一个枚举
IPv 枚举类型有专门的库
枚举较之结构体的优势：可以统揽多个结构体为一个类型，例如

```
enum Message {
        Quit,
        Move{x: i32, y:i32},
        Write(String),
        ChangeColor(i32, i32, i32),
}

等同于以下的集合：

struct QuitMessage;
```

```
struct MoveMessage {
        x: i32,
        y: i32,
}
struct WriteMessage(String);
struct ChangeColorMessage(i32, i32, i32);
```

枚举也可以定义方法，同结构体
Option枚举：
预导入模块不需要显式引入作用域
Option具有some和None两个变体
因为 `let y: Option<i8> = Some(5)` 和 `let x: i8 = 5` 是两个不同类型，需要相加之前把Option 类型转化为T类型，这里会进行一个空值检验。
只要不是Option类型的值，我们就可以把它假设为不是非空的。
match：控制流运算符
match value {
mode => code or expression,
}
模式匹配：利用match， 直接在模式中写入变量，若变体确实包含这个字段，则该字段的值赋给这个变量，从而实现了数据的提取
匹配必须穷举所有的可能
通配符：
`_ => (),` 用于处理不关心的模式和值
简化版的match： `if let`

```
let mut count = 0;
if let Coin::Quarter(state) = coin {
        println!("STate quarter from {:?}!", state);
} else {
        count += 1;
}
```

当枚举中包含数据时候，可以使用match或者if let来抽取值

包和模块：
运行命令 `cargo new --lib restaurant` 创建一个名为restaurant的库，这时根节点crate是lib.rs
一个包内最多只能又1个库单元包
模块中的任何条目都是默认私有的，但是父模块中的条目对子来说可见，反之不可见。可实现：默认隐藏内部的实现细节。可以用public依次暴露路径上的所有条目
在模块面前加public，但是模块内部的条目依然是私有的

修饰符mod定义模块
使用super关键字构造相对路径：

```
fn s() { }
mod b {
        fn f( ) {
        super::s();       // 这里super之后跳到了crate
        }
        fn c(){ }
}
```

某模块如果存在私有字段，又没有公共的关联函数来构造其实例，会报错，无法实例化。
将一个枚举声明为public时，其中所有字段都为public；但struct 不同
use关键字：
`use crate::` 或者`use self::` 开头
一些迷思：

```
use crate :: front_of_house :: hosting :: add_to_waitlist;
pub fn e() {
        add_to_waitlist();
}
```

这种引入方式不可取，因为无法直观的表现出add_to_waitlist的定义位置，它看起来很像在本文件中定义的，同时完整路径过于冗长，同时还不能区分来自不同模块的同名函数（当然可以使用as重命名）。hosting::add_to_waitlist()；更好些
当我们引入结构体，枚举等时，我们通过第一种方法引入，即完整路径
重导出：不仅可以被导入，还可以被导出
使用pub use作修饰符：`pub use crate::front_of_house::hosting`，使得该作用域外部的代码也可以使用这个模块
使用外部包：
简写：`use std::io::{self, Write};` 等同于`use std::io;` `use std::io::Write;`
通配符：
`use std::collections::*` 全部引入，常用于测试，和预导入模块
将模块拆分为不同的文件，详见module_dir/

通用集合类型：
包括动态数组，String, 哈希映射
动态数组：
动态数组只能存储相同类型的值，可以用枚举来包括不同类型，但枚举的实例是相同类型。
String:

Rust中的字符串使用了utf-8编码
某些第三方包提供了其他字符串类型，适用于不同的场合
Rust字符串包含了str和String，不支持索引（字符串索引操作返回类型不明确，字节，字符，或字形簇，或字符串切片）
切片：
right - left 必须是其语言“最小单位”的整数倍，可以通过len() 除以‘字符’个数来计算最小单位。例如：`let s = &hello[0..4]`
遍历：
`for c in "".chars() {}` 和 `for b in "".bytes() {}`
合法的Unicode标量值可能会需要占据1字节以上的空间
需要理解UTF-8数据的存储流程
哈希映射：
Python中的字典也是哈希映射的一种

错误处理：
`$env:RUST_BACKTRACE=1 ; cargo run` 当程序崩溃时输出回溯信息
(未完待续.....)