

# Rust 教程



Rust 语言是一种高效、可靠的通用高级语言。其高效不仅限于开发效率，它的执行效率也是令人称赞的，是一种少有的兼顾开发效率和执行效率的语言。

Rust 语言由 Mozilla 开发，最早发布于 2014 年 9 月。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 官方在线工具: <https://play.rust-lang.org/>.

Rust 系列文章内容由 **Sobin** 收集整理。

## Rust语言的特点

- 高性能** - Rust 速度惊人且内存利用率极高。由于没有运行时和垃圾回收，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。
- 可靠性** - 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，让您在编译期就能够消除各种各样的错误。
- 生产力** - Rust 拥有出色的文档、友好的编译器和清晰的错误提示信息，还集成了一流的工具——包管理器和构建工具，智能地自动补全和类型检验的多编辑器支持，以及自动格式化代码等等。

## Rust的应用

Rust 语言可以用于开发：

- 传统命令行程序** - Rust 编译器可以直接生成目标可执行程序，不需要任何解释程序。
- Web 应用** - Rust 可以被编译成 WebAssembly，WebAssembly 是一种 JavaScript 的高效替代品。
- 网络服务器** - Rust 用极低的资源消耗做到安全高效，且具备很强的大规模并发处理能力，十分适合开发普通或极端的服务器程序。
- 嵌入式设备** - Rust 同时具有 JavaScript 一般的高效开发语法和 C 语言的执行效率，支持底层平台的开发。

## 谁适合阅读本教程？

本教程对于初级的编程知识将默认读者已经掌握，所以如果你阅读本教程，你需要对初级的编程知识有一定的了解（最好已经初识 C/C++ 或 JavaScript 编程语言）。

### 第一个 Rust 程序

Rust 语言代码文件后缀名为 `.rs`，如 `runoob.rs`。

#### 实例: `runoob.rs` 文件

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

[运行实例 »](#)

使用 `rustc` 命令编译 `runoob.rs` 文件：

```
$ rustc runoob.rs # 编译 runoob.rs 文件
```

编译后会生成 `runoob` 可执行文件：

```
$ ./runoob # 执行 runoob
Hello World!
```

## 参考链接

- Rust 官方网站: <https://www.rust-lang.org/zh-CN>
- Rust 官方文档: <https://doc.rust-lang.org/>
- Rust Play: <https://play.rust-lang.org/>
- Visual Studio Code: <https://code.visualstudio.com/>

## Rust 环境搭建

Rust 支持很多的集成开发环境 (IDE) 或开发专用的文本编辑器。

官方网站公布支持的工具如下 (<https://www.rust-lang.org/zh-CN/tools>) :



本教程将使用 Visual Studio Code 作为我们的开发环境 (Eclipse 有专用于 Rust 开发的版本, 对于初学者也是不错的选择)。

注意: IntelliJ IDEA 安装插件之后难以调试, 所以推荐习惯使用 IDEA 的开发者使用 CLion, 但 CLion 不是免费的。

### 搭建 Visual Studio Code 开发环境

首先, 需要安装最新版的 Rust 编译工具和 Visual Studio Code。

Rust 编译工具: <https://www.rust-lang.org/zh-CN/tools/install>

Visual Studio Code: <https://code.visualstudio.com/Download>

Rust 的编译工具依赖 C 语言的编译工具, 这意味着你的电脑上至少已经存在一个 C 语言的编译环境。如果你使用的是 Linux 系统, 往往已经具备了 GCC 或 clang。如果你使用的是 macOS, 需要安装 Xcode。如果你是用的是 Windows 操作系统, 你需要安装 Visual Studio 2013 或以上的环境 (需要 C/C++ 支持) 以便使用 MSVC 或安装 MinGW + GCC 编译环境 (Cygwin 还没有测试)。

### 安装 Rust 编译工具

Rust 编译工具推荐使用刚才从上方链接中下载的 Rustup 安装。下载好的 Rustup 在 Windows 上是一个可执行程序 rustup-init.exe。

(在其他平台上应该是 `rustup-init.sh` )。

现在执行 rustup-init 文件:

```
C:\Users\Sobin\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
```

上图显示的是一个命令行安装向导。

如果你已经安装 MSVC (推荐), 那么安装过程会非常的简单, 输入 1 并回车, 直接进入第二步。

如果你安装的是 MinGW, 那么你需要输入 2 (自定义安装), 然后系统会询问你 Default host triple?, 请将上图中 default host triple 的 "msvc" 改为 "gnu" 再输入安装程序:

```
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
>2
I'm going to ask you the value of each of these installation options.
You may simply press the Enter key to leave unchanged.
Default host triple?
x86_64-pc-windows-gnu.
```

其它属性都默认。

设置完所有选项, 会回到安装向导界面 (第一张图), 这是我们输入 1 并回车即可。

```
rustup -V      # 注意的大写的 V
命令提示符 - D:\Downloads\rustup-init.exe

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
>1
info: profile set to 'default'
info: setting default host triple to x86_64-pc-windows-msvc
info: updating existing rustup installation

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.
```

进行到这一步就完成了 Rust 的安装, 可以通过以下命令测试:

```
rustc -V      # 注意的大写的 V
命令提示符

C:\>rustc -V
rustc 1.41.0 (5e1a79984 2020-01-27)

C:\>cargo -V
cargo 1.41.0 (626f0f40e 2019-12-03)

C:\>
```

如果以上两个命令能够输出你安装的版本号, 就是安装成功了。

### 搭建 Visual Studio Code 开发环境

下载完 Visual Studio Code 安装包之后启动安装向导安装 (此步骤不在此赘述)。

安装完 Visual Studio Code (下文简称 VSCode) 之后运行 VSCode。



用同样的方法再安装 rust-analyzer 和 Native Debug 两个扩展。



重新启动 VSCode, Rust 的开发环境就搭建好了。

现在新建一个文件夹, 如 runoob-greeting。



在 VSCode 中打开新建的文件夹:



当前文件下会构建一个名叫 greeting 的 Rust 工程目录。



现在在终端里输入以下三个命令:



系统在创建工程时会生成一个 Hello, world 源程序 main.rs, 这时会被编译并运行:

```
PS D:\Developments\rust\runoob-greeting> cd ./greeting
PS D:\Developments\rust\runoob-greeting> cargo build
    Compiling greeting v0.1.0 (D:\Developments\rust\runoob-greeting\greeting)
    Finished dev [unoptimized + debuginfo] target(s) in 0.81s
```

```
PS D:\Developments\rust\runoob-greeting> cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
        Running `target\debug\greeting.exe`
```

Hello, world!

至此, 你成功的构建了一个 Rust 命令行程序!

有关在 VSCode 中调试程序的问题, 详见 [Cargo 教程](#)。

## Cargo 教程

### Cargo 是什么

Cargo 是 Rust 的构建系统和包管理器。

Rust 开发者常用 Cargo 来管理 Rust 工程和获取工程所依赖的库。在上个教程中我们曾使用 cargo new greeting 命令创建了一个名为 greeting 的工程，Cargo 新建了一个名为 greeting 的文件夹并在里面部署了一个 Rust 工程最典型的文件结构。这个 greeting 文件夹就是工程本身。

### Cargo 功能

Cargo 除了创建工程以外还具备构建 (build) 工程、运行 (run) 工程等一系列功能，构建和运行分别对应以下命令：

```
cargo build  
cargo run
```

Cargo 还具有获取包、打包、高级构建等功能，详细使用方法参见 Cargo 命令。

### 在 VSCode 中配置 Rust 工程

Cargo 是一个不错的构建工具，如果使 VSCode 与它相配合那么 VSCode 将会是一个十分便捷的开发环境。

在上一章中我们建立了 greeting 工程，现在我们用 VSCode 打开 greeting 文件夹（注意不是 runoob-greeting）。

打开 greeting 之后，在里面新建一个新的文件夹 .vscode（注意 vscode 前面的点，如果有这个文件夹就不需要新建了）。在新建的 vscode 文件夹里新建两个文件 tasks.json 和 launch.json，文件内容如下：

#### tasks.json 文件

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "build",
      "type": "shell",
      "command": "cargo",
      "args": ["build"]
    }
  ]
}
```

#### launch.json 文件 (适用于 Windows 系统上)

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "(Windows) 启动",
      "preLaunchTask": "build",
      "type": "cppvsdbg",
      "request": "launch",
      "program": "${workspaceFolder}/target/debug/${workspaceFolderBasename}.exe",
      "args": [],
      "stopAtEntry": false,
      "cwd": "${workspaceFolder}",
      "environment": [],
      "externalConsole": false
    },
    {
      "name": "(gdb) 启动",
      "type": "cppdbg",
      "request": "launch",
      "program": "${workspaceFolder}/target/debug/${workspaceFolderBasename}.exe",
      "args": [],
      "stopAtEntry": false,
      "cwd": "${workspaceFolder}",
      "environment": [],
      "externalConsole": false,
      "MIIMode": "gdb",
      "miDebuggerPath": "这里填GDB所在的目录",
      "setupCommands": [
        {
          "description": "为 gdb 启用整齐打印",
          "text": "-enable-pretty-printing",
          "ignoreFailures": true
        }
      ]
    }
  ]
}
```

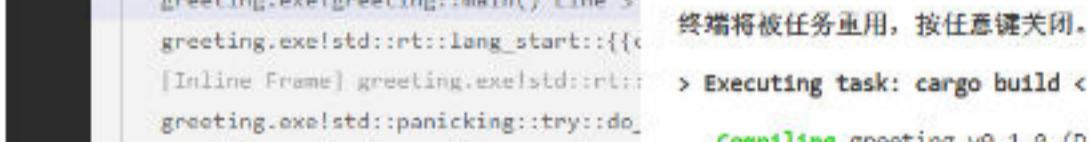
#### launch.json 文件 (适用于 Linux 系统上)

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Debug",
      "type": "gdb",
      "preLaunchTask": "build",
      "request": "launch",
      "target": "${workspaceFolder}/target/debug/${workspaceFolderBasename}",
      "cwd": "${workspaceFolder}"
    }
  ]
}
```

然后点击 VSCode 左栏的“运行”。

如果你使用的是 MSVC 选择“(Windows) 启动”。

如果使用的是 MinGW 且安装了 GDB 选择“(gdb) 启动”，gdb 启动前请注意填写 launch.json 中的 “miDebuggerPath”。

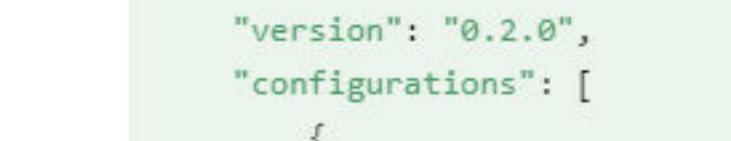


程序就会开始调试运行了。运行输出将出现在“调试控制台”中：



### 在 VSCode 中调试 Rust

调试程序的方法与其它环境相似，只需要在行号的左侧点击红点就可以设置断点，在运行中遇到断点会暂停，以供开发者监视实时变量的值。



另外 cargo build/run --release 使用 release 编译会比默认的 debug 编译性能提升 10 倍以上，但是 release 缺点是编译速度较慢，而且不会显示 panic backtrace 的具体行号

[cargo expand](#) 3年前 [2020-10-13]

使用 M1 M2 芯片的苹果电脑，还需要安装 CodeLLDB 插件，并在 launch.json 中指定 targetArchitecture 为 arm64：

```
{
  // Use IntelliSense to learn about possible attributes.  
// Hover to view descriptions of existing attributes.  
// For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387  
"version": "0.2.0",  
"configurations": [  
  {
    "name": "clang++ - Build and debug active file",
    "type": "lldb",
    "request": "launch",
    "program": "./benchncnn",
    "stopAtEntry": true,
    "cwd": "/Users/oldpan/deps/ncnn/build_debug/benchmark/",
    "environment": [],
    "externalConsole": false,
    "MIIMode": "lldb",
    "targetArchitecture": "arm64"
  }
]
```

apple@mt: 11个月前 [10-26]

## 2 篇笔记

## 写笔记

- 再补充几个 cargo 的重要子命令：
    - cargo clippy: 类似eslint，lint 工具检查代码可以优化的地方
    - cargo fmt: 类似 go fmt，代码格式化
    - cargo tree: 查看第三方库的版本和依赖关系
    - cargo bench: 运行 benchmark(基准测试, 性能测试)
    - cargo udeps(第三方): 检查项目中未使用的依赖
- 另外 cargo build/run --release 使用 release 编译会比默认的 debug 编译性能提升 10 倍以上，但是 release 缺点是编译速度较慢，而且不会显示 panic backtrace 的具体行号
- [cargo expand](#) 3年前 [2020-10-13]

使用 M1 M2 芯片的苹果电脑，还需要安装 CodeLLDB 插件，并在 launch.json 中指定 targetArchitecture 为 arm64：

```
{
  // Use IntelliSense to learn about possible attributes.  
// Hover to view descriptions of existing attributes.  
// For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387  
"version": "0.2.0",  
"configurations": [  
  {
    "name": "clang++ - Build and debug active file",
    "type": "lldb",
    "request": "launch",
    "program": "./benchncnn",
    "stopAtEntry": true,
    "cwd": "/Users/oldpan/deps/ncnn/build_debug/benchmark/",
    "environment": [],
    "externalConsole": false,
    "MIIMode": "lldb",
    "targetArchitecture": "arm64"
  }
]
```

apple@mt: 11个月前 [10-26]

# Rust 输出到命令行

在正式学习 Rust 语言以前，我们需要先学会怎样输出一段文字到命令行，这几乎是学习每一门语言之前必备的技能，因为输出到命令行几乎是语言学习阶段程序表达结果的唯一方式。

在之前的 Hello, World 程序中大概已经告诉了大家输出字符串的方式，但并不全面，大家可能很疑惑为什么 `println!( "Hello World" )` 中的 `println` 后面还有一个 `!` 符号，难道 Rust 函数之后都要加一个感叹号？显然并不是这样。`println` 不是一个函数，而是一个宏规则。这里不需要更深刻的挖掘宏规则是什么，后面的章节中会专门介绍，并不影响接下来的一段学习。

Rust 输出文字的方式主要有两种：`println!()` 和 `print!()`。这两个“函数”都是向命令行输出字符串的方法，区别仅在于前者会在输出的最后附加输出一个换行符。当用这两个“函数”输出信息的时候，第一个参数是格式字符串，后面是一串可变参数，对应着格式字符串中的“占位符”，这一点与 C 语言中的 `printf` 函数很相似。但是，Rust 中格式字符串中的占位符不是 “% + 字母”的形式，而是一对 `{}`。

## 实例：runoob.rs 文件

```
fn main() {
    let a = 12;
    println!("a is {}", a);
}
```

使用 `rustc` 命令编译 runoob.rs 文件：

```
$ rustc runoob.rs # 编译 runoob.rs 文件
```

编译后会生成 `runoob` 可执行文件：

```
$ ./runoob # 执行 runoob
```

以上程序的输出结果是：

```
a is 12
```

如果我想把 `a` 输出两遍，那岂不是要写成：

```
println!("a is {}, a again is {}", a, a);
```

其实有更好的写法：

```
println!("a is {0}, a again is {0}", a);
```

在 `{}` 之间可以放一个数字，它将把之后的可变参数当作一个数组来访问，下标从 0 开始。

如果要输出 `{` 或 `}` 怎么办呢？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。

```
fn main() {
    println!("{{}}");
}
```

以上程序的输出结果是：

```
{}
```

# Rust 基础语法

变量，基本类型，函数，注释和控制流，这些几乎是每种编程语言都具有的编程概念。

这些基础概念将存在于每个 Rust 程序中，及早学习它们将使你以最快的速度学习 Rust 的使用。

## 变量

首先必须说明，Rust 是强类型语言，但具有自动判断变量类型的能力。这很容易让人与弱类型语言产生混淆。

如果要声明变量，需要使用 `let` 关键字。例如：

```
let a = 123;
```

只学习过 JavaScript 的开发者对这句话很敏感，只学习过 C 语言的开发者对这句话很不理解。

在这句声明语句之后，以下三行代码都是被禁止的：

```
a = "abc";
a = 4.56;
a = 456;
```

第一行的错误在于当声明 `a` 是 `123` 以后，`a` 就被确定为整型数字，不能把字符串类型的值赋给它。

第二行的错误在于自动转换数字精度有损失，Rust 语言不允许精度有损失的自动数据类型转换。

第三行的错误在于 `a` 不是个可变变量。

前两种错误很容易理解，但第三个是什么意思？难道 `a` 不是个变量吗？

这就牵扯到了 Rust 语言为了高并发安全而做的设计：在语言层面尽量少的让变量的值可以改变。所以 `a` 的值不可变。但这不意味着 `a` 不是“变量”（英文中的 variable），官方文档称 `a` 这种变量为“不可变变量”。

如果我们编写的程序的一部分在假设值永远不会改变的情况下运行，而我们代码的另一部分在改变该值，那么代码的第一部分可能就不会按照设计的意图去运转。由于这种原因造成的选择很难在事后找到。这是 Rust 语言设计这种机制的原因。

当然，使变量变得“可变”（mutable）只需一个 `mut` 关键字。

```
let mut a = 123;
a = 456;
```

这个程序是正确的。

## 常量与不可变变量的区别

既然不可变变量是不可变的，那不就是常量吗？为什么叫变量？

变量和常量还是有区别的。在 Rust 中，以下程序是合法的：

```
let a = 123; // 可以编译，但可能有警告，因为该变量没有被使用
let a = 456;
```

但是如果 `a` 是常量就不合法：

```
const a: i32 = 123;
let a = 456;
```

变量的值可以“重新绑定”，但在“重新绑定”以前不能私自被改变，这样可以确保在每一次“绑定”之后的区域里编译器可以充分的推理程序逻辑。虽然 Rust 有自动判断类型的功能，但有些情况下声明类型更加方便：

```
let a: u64 = 123;
```

这里声明了 `a` 为无符号 64 位整型变量，如果没有声明类型，`a` 将自动被判断为有符号 32 位整型变量，这对于 `a` 的取值范围有很大的影响。

## 重影 (Shadowing)

重影的概念与其他面向对象语言里的“重写”（Override）或“重载”（Overload）是不一样的。重影就是刚才讲述的所谓“重新绑定”，之所以加引号就是为了在没有介绍这个概念的时候代替一下概念。

重影就是指变量的名称可以被重新使用的机制：

### 实例

```
fn main() {
    let x = 5;
    let x = x + 1;
    let x = x * 2;
    println!("The value of x is: {}", x);
}
```

这段程序的运行结果：

```
The value of x is: 12
```

重影与可变变量的赋值不是一个概念，重影是指用同一个名字重新代表另一个变量实体，其类型、可变属性和值都可以变化。但可变变量赋值仅能发生值的变化。

```
let mut s = "123";
s = s.len();
```

这段程序会出错：不能给字符串变量赋整型值。

# Rust 数据类型

Rust 语言中的基础数据类型有以下几种。

## 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

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。

整数的表述方法有以下几种：

进制	例
十进制	98_222
十六进制	0xff
八进制	0o77
二进制	0b1111_0000
字节(只能表示 u8 型)	b'A'

很显然，有的整数中间存在一个下划线，这种设计可以让人们在输入一个很大的数字时更容易判断数字的值大概是多少。

## 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。

### 实例

```
fn main() {
    let x = 2.0; // f64
    let y: f32 = 3.0; // f32
}
```

## 数学运算

用一段程序反映数学运算：

### 实例

```
fn main() {
    let sum = 5 + 10; // 加
    let difference = 95.5 - 4.3; // 减
    let product = 4 * 30; // 乘
    let quotient = 56.7 / 32.2; // 除
    let remainder = 43 % 5; // 求余
}
```

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。

注意：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。

## 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

元组是用一对 () 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
```

数组用一对 [] 包括的同类型数据。

### 实例

```
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

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
```

# Rust 注释

Rust 中的注释方式与其它语言 (C、Java) 一样，支持两种注释方式：

## 实例

```
// 这是第一种注释方式

/* 这是第二种注释方式 */

/*
 * 多行注释
 * 多行注释
 * 多行注释
 */
```

## 用于说明文档的注释

在 Rust 中使用 `//` 可以使其之后到第一个换行符的内容变成注释。

在这种规则下，三个正斜杠 `///` 依然是合法的注释开始。所以 Rust 可以用 `///` 作为说明文档注释的开头：

## 实例

```
/// Adds one to the number given.
///
/// # Examples
///
/// ```
/// let x = add(1, 2);
/// ```
/// ```

fn add(a: i32, b: i32) -> i32 {
    return a + b;
}

fn main() {
    println!("{}", add(2, 3));
}
```

程序中的函数 `add` 就会拥有一段优雅的注释，并可以显示在 IDE 中：

The screenshot shows the Visual Studio Code interface with the 'main.rs' file open. The code contains Rust functions and documentation blocks. A tooltip is displayed over the documentation block for the `fn add` function, showing the docstring and the word 'Examples'.

**Tip:** Cargo 具有 cargo doc 功能，开发者可以通过这个命令将工程中的说明注释转换成 HTML 格式的说明文档。

## Rust 函数

函数在 Rust 语言中是普遍存在的。

通过之前的章节已经可以了解到 Rust 函数的基本形式：

```
fn <函数名> (<参数>) <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：

### 实例

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

fn another_function() {
    println!("Hello, runoob!");
}
```

运行结果：

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

注意，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。

### 函数参数

Rust 中定义函数如果需要具备参数必须声明参数名称和类型：

### 实例

```
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);
}
```

运行结果：

```
x 的值为 : 5
y 的值为 : 6
```

### 函数体的语句和表达式

Rust 函数体由一系列可以以表达式 (Expression) 结尾的语句 (Statement) 组成。到目前为止，我们仅见到了没有以表达式结尾的函数，但已经将表达式用作语句的一部分。

语句是执行某些操作且没有返回值的步骤。例如：

```
let a = 6;
```

这个步骤没有返回值，所以以下语句不正确：

```
let a = (let b = 2);
```

表达式有计算步骤且有返回值。以下是表达式（假设出现的标识符已经被定义）：

```
a = 7
b + 2
c * (a + b)
```

Rust 中可以在一个用 {} 包括的块里编写一个较为复杂的表达式：

### 实例

```
fn main() {
    let x = 5;

    let y = {
        let x = 3;
        x + 1
    };

    println!("x 的值为 : {}", x);
    println!("y 的值为 : {}", y);
}
```

运行结果：

```
x 的值为 : 5
y 的值为 : 4
```

很显然，这段程序中包含了一个表达式块：

```
{ 
    let x = 3;
    x + 1
};
```

而且在块中可以使用函数语句，最后一个步骤是表达式，此表达式的结果值是整个表达式块所代表的值。这种表达式块叫做函数体表达式。

注意：x + 1 之后没有分号，否则它将变成一条语句！

这种表达式块是一个合法的函数体。而且在 Rust 中，函数定义可以嵌套：

### 实例

```
fn main() {
    fn five() -> i32 {
        5
    }
    println!("five() 的值为: {}", five());
}
```

### 函数返回值

在上一个嵌套的例子中已经显示了 Rust 函数声明返回值类型的方式：在参数声明之后用 -> 来声明函数返回值的类型（不是 :）。

在函数体中，随时都可以以 return 关键字结束函数运行并返回一个类型合适的值。这也是最接近大多数开发者经验的做法：

### 实例

```
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
```

但是 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。

注意：函数体表达式并不能等同于函数体，它不能使用 return 关键字。

# Rust 条件语句

在 Rust 语言中的条件语句是这种格式的：

## 实例

```
fn main() {
    let number = 3;
    if number < 5 {
        println!("条件为 true");
    } else {
        println!("条件为 false");
    }
}
```

在上述程序中有条件 if 语句，这个语法在很多其它语言中很常见，但也有一些区别：首先，条件表达式 `number < 5` 不需要用小括号括（注意，不需要不是不允许）；但是 Rust 中的 if 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。尽管如此，Rust 还是支持传统 else-if 语法的：

## 实例

```
fn main() {
    let a = 12;
    let b;
    if a > 0 {
        b = 1;
    }
    else if a < 0 {
        b = -1;
    }
    else {
        b = 0;
    }
    println!("b is {}", b);
}
```

运行结果：

b 为 1

Rust 中的条件表达式必须是 bool 类型，例如下面的程序是错误的：

## 实例

```
fn main() {
    let number = 3;
    if number { // 报错, expected `bool`, found integer rustc(E0308)
        println!("Yes");
    }
}
```

虽然 C/C++ 语言中的条件表达式用整数表示，非 0 即真，但这个规则在很多注重代码安全性的语言中是被禁止的。

结合之前章学习的函数体表达式我们加以联想：

`if <condition> { block 1 } else { block 2 }`

这种语法中的 { block 1 } 和 { block 2 } 可不可以是函数体表达式呢？

答案是肯定的！也就是说，在 Rust 中我们可以使用 if-else 结构实现类似于三元条件运算表达式 (A ? B : C) 的效果：

## 实例

```
fn main() {
    let a = 3;
    let number = if a > 0 { 1 } else { -1 };
    println!("number 为 {}", number);
}
```

运行结果：

number 为 1

注意：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。

## Rust 循环

Rust 除了灵活的条件语句以外，循环结构的设计也十分成熟。这一点作为身经百战的开发者应该能感觉出来。

### while 循环

while 循环是最典型的条件语句循环：

#### 实例

```
fn main() {
    let mut number = 1;
    while number != 4 {
        println!("{}", number);
        number += 1;
    }
    println!("EXIT");
}
```

运行结果：

```
1
2
3
EXIT
```

Rust 语言到此教程编撰之日还没有 do-while 的用法，但是 do 被规定为保留字，也许以后的版本中会用到。

在 C 语言中 for 循环使用三元语句控制循环，但是 Rust 中没有这种用法，需要用 while 循环来代替：

#### C 语言

```
int i;
for (i = 0; i < 10; i++) {
    // 循环体
}
```

#### Rust

```
let mut i = 0;
while i < 10 {
    // 循环体
    i += 1;
}
```

### for 循环

for 循环是最常用的循环结构，常用来遍历一个线性数据结构（比如数组）。for 循环遍历数组：

#### 实例

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

运行结果：

```
值为：10
值为：20
值为：30
值为：40
值为：50
```

这个程序中的 for 循环完成了对数组 a 的遍历。a.iter() 代表 a 的迭代器 (iterator)，在学习有关于对象的章节以前不做赘述。

当然，for 循环其实是可以通过下标来访问数组的：

#### 实例

```
fn main() {
    let a = [10, 20, 30, 40, 50];
    for i in 0..5 {
        println!("a[{}] = {}", i, a[i]);
    }
}
```

运行结果：

```
a[0] = 10
a[1] = 20
a[2] = 30
a[3] = 40
a[4] = 50
```

### loop 循环

身经百战的开发者一定遇到过几次这样的情况：某个循环无法在开头和结尾判断是否继续进行循环，必须在循环体中间某处控制循环的进行。如果遇到这种情况，我们经常会在一个 while (true) 循环体里实现中途退出循环的操作。

Rust 语言有原生的无限循环结构——loop：

#### 实例

```
fn main() {
    let s = ['R', 'U', 'N', 'O', 'O', 'B'];
    let mut i = 0;
    loop {
        let ch = s[i];
        if ch == 'O' {
            break;
        }
        println!("'{}'", ch);
        i += 1;
    }
}
```

运行结果：

```
'R'
'U'
'N'
```

loop 循环可以通过 break 关键字类似于 return 一样使整个循环退出并给予外部一个返回值。这是一个十分巧妙的设计，因为 loop 这样的循环常被用来当作查找工具使用，如果找到了某个东西当然要将这个结果交出去：

#### 实例

```
fn main() {
    let s = ['R', 'U', 'N', 'O', 'O', 'B'];
    let mut i = 0;
    let location = loop {
        let ch = s[i];
        if ch == 'O' {
            break i;
        }
        i += 1;
    };
    println!("'O' 的索引为 {}", location);
}
```

运行结果：

```
'O' 的索引为 3
```

## Rust 所有权

计算机程序必须在运行时管理它们所使用的内存资源。

大多数的编程语言都有管理内存的功能：

C/C++ 这样的语言主要通过手动方式管理内存，开发者需要手动的申请和释放内存资源。但为了提高开发效率，只要不影响程序功能的实现，许多开发者没有及时释放内存的习惯，所以手动管理内存的方式常常会造成资源浪费。

Java 语言编写的程序在虚拟机（JVM）中运行，JVM 具备自动回收内存资源的功能。但这种方式常常会降低运行时效率，所以 JVM 尽可能少的回收资源，这样也会使程序占用较大的内存资源。

所有权对大多数开发者而言是一个新颖的概念，它是 Rust 语言为高效使用内存而设计的语法规则。所有权概念是为了让 Rust 在编译阶段更有效地分析内存资源的有用性以实现内存管理而诞生的概念。

### 所有权规则

所有权有以下三条规则：

- Rust 中的每个值都有一个变量，称为其所有者。
- 一次只能有一个所有者。
- 当所有者不在程序运行范围时，该值将被删除。

这三条规则是所有权概念的基础。

接下来将介绍与所有权概念有关的概念。

### 变量范围

我们用下面这段程序描述变量范围的概念：

```
{
    // 在声明以前，变量 s 无效
    let s = "runoob";
    // 这里是变量 s 的可用范围
}
// 变量范围已经结束，变量 s 无效
```

变量范围是变量的一个属性，其代表变量的可见域，默认从声明变量开始有效直到变量所在域结束。

## 内存和分配

如果我们定义了一个变量并给它赋值，这个变量的值存在于内存中。这种情况很普遍。但如果我们要储存的数据长度不确定（比如用户输入的一串字符串），我们就无法在定义时明确数据长度，也就无法在编译阶段令程序分配固定长度的内存空间供数据储存使用。（有人说分配尽可能大的空间可以解决问题，但这个方法很不文明）。这就需要提供一种在程序运行时程序员自己申请使用内存的机制——堆。本章讲述的所有“内存操作”都指的是堆所占有的内存空间。

有分配就有释放，程序不能一直占用某个内存资源。因此决定资源是否浪费的关键因素就是资源有没有及时的释放。

我们把字符串样例程序用 C 语言等价编写：

```
{
    char *s = strdup("runoob");
    free(s); // 释放 s 资源
}
```

很显然，Rust 中没有调用 free 函数来释放字符串 s 的资源（我知道这样在 C 语言中是不正确的写法，因为“runoob”不在堆中，这里假设它在）。Rust之所以没有明示释放的步骤是因为在变量范围结束的时候，Rust 编译器自动添加了调用释放资源函数的步骤。

这种机制看似很简单：它不过是帮助程序员在适当的地方添加了一个释放资源的函数调用而已。但这种简单的机制可以有效地解决一个史上最令程序员头疼的编码问题。

## 变量与数据交互的方式

变量与数据交互方式主要有移动（Move）和克隆（Clone）两种：

### 移动

多个变量可以在 Rust 中以不同的方式与相同的数据交互：

```
let x = 5;
let y = x;
```

这个程序将值 5 绑定到变量 x，然后将 x 的值复制并赋值给变量 y。现在栈中将有两个值 5。此情况中的数据是“基本数据”类型的数据，不需要存储到堆中，仅在栈中的数据的“移动”方式是直接复制，这不会花费更长的时间或更多的存储空间。“基本数据”类型有这些：

- 所有整数类型，例如 i32、u32、i64 等。
- 布尔类型 bool，值为 true 或 false。
- 所有浮点类型，f32 和 f64。
- 字符类型 char。
- 仅包含以上类型数据的元组（Tuples）。

但如果发生交互的数据在堆中就是另外一种情况：

```
let s1 = String::from("hello");
let s2 = s1;
```

第一步产生一个 String 对象，值为 "hello"。其中 "hello" 可以认为是类似于长度不确定的数据，需要在堆中存储。

第二步的情况略有不同（这不是完全真的，仅用来对比参考）：



如图所示：两个 String 对象在栈中，每个 String 对象都有一个指针指向堆中的 "hello" 字符串。在给 s2 赋值时，只有栈中的数据被复制了，堆中的字符串依然还是原来的字符串。

前面我们说过，当变量超出范围时，Rust 自动调用释放资源函数并清理该变量的堆内存。但是 s1 和 s2 都被释放的话堆区中的 "hello" 被释放两次，这是不被系统允许的。为了确保安全，在给 s2 赋值时 s1 已经失效了。没错，在把 s1 的值赋给 s2 以后 s1 将不可以再被使用。下面这段程序是错的：

```
let s1 = String::from("hello");
let s2 = s1;
println!("{}{}, world!", s1, s2); // 错误！s1 已经失效
```

所以实际情况是：



s1 名存实亡。

### 克隆

Rust 尽可能地降低程序的运行成本，所以默认情况下，长度较大的数据存放在堆中，且采用移动的方式进行数据交互。但如果需要将数据单纯的复制一份以供他用，可以使用数据的第二种交互方式——克隆。

### 实例

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

运行结果：

```
s1 = hello, s2 = hello
```

这里是将堆中的 "hello" 复制了一份，所以 s1 和 s2 都分别绑定了一个值，释放的时候也会被当作两个资源。

当然，克隆仅在需要复制的情况下使用，毕竟复制数据会花费更多的时间。

## 涉及函数的所有权机制

对于变量来说这是最复杂的情况了。

如果将一个变量当作函数的参数传给其他函数，怎样安全的处理所有权呢？

下面这段程序描述了这种情况下的所有权机制的运行原理：

### 实例

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1.clone();
    takes_ownership(s1);
    // s1 的值被当作参数传入函数
    // 所以 s2 依然有效，从这里开始已经无效
}
```

函数结束后，x 无效，然后是 s，但 s 已被移动，所以不用被释放

```
fn takes_ownership(some_string: String) {
    // 一个 String 参数 some_string 传入，有效
    println!("{}!", some_string);
} // 函数结束，参数 some_string 在这里释放
```

```
fn makes_copy(some_integer: i32) {
    // x 的值被当作参数传入函数
    // x 是基本类型，依然有效
    // 在这里依然可以使用 x 却不能使用 s
}
```

```
} // 函数结束，x 无效，然后是 s，但 s 已被移动，所以不用被释放
```

如果将变量当作参数传入函数，那么它和移动的效果是一样的。

### 函数返回值的所有权机制

#### 实例

```
fn main() {
    let s1 = gives_ownership();
    // gives_ownership 移动它的返回值到 s1
}
```

运行结果：

```
s1 = hello, s2 = hello
```

这里是真的将堆中的 "hello" 复制了一份，所以 s1 和 s2 都分别绑定了一个值，释放的时候也会被当作两个资源。

当然，克隆仅在需要复制的情况下使用，毕竟复制数据会花费更多的时间。

## 涉及函数的所有权机制

对于变量来说这是最复杂的情况了。

如果将一个变量当作函数的参数传给其他函数，怎样安全的处理所有权呢？

下面这段程序描述了这种情况下的所有权机制的运行原理：

### 实例

```
fn main() {
    let s1 = String::from("hello");
    let s2 = &s1;
    takes_and_gives_back(s2);
    // s2 被当作参数移动，s2 被修改后返回值所有权
} // s3 无法被释放，s2 被移动，s1 无法被释放
```

如果尝试利用租借的权利来修改数据会被阻止：

```
fn main() {
    let s1 = String::from("hello");
    let s2 = &s1;
    takes_and_gives_back(s2);
    // s2 被当作参数移动，s2 被修改后返回值所有权
} // s3 无法被释放，s2 被移动，s1 无法被释放
```

这段程序不正确，因为 s2 租借的 s1 已经将所有权移动到 s3，所以 s2 将无法继续租借使用 s1 的所有权。如果需要使用 s2 使用该值，必须重新租借：

```
fn main() {
    let s1 = String::from("hello");
    let s2 = &s1;
    let s3 = s1; // 重新从 s1 租借所有权
    takes_and_gives_back(s3);
    // s3 被当作参数移动，s3 被修改后返回值所有权
} // s3 无法被释放，s2 被移动，s1 无法被释放
```

这段程序是正确的，即使它租借了所有权，它也只享有使用权（这跟租房子是一个道理）。

如果尝试利用租借的权利来修改数据会被阻止：

```
fn main() {
    let s1 = String::from("hello");
    let s2 = &s1;
    takes_and_gives_back(s2);
    // s2 被当作参数移动，s2 被修改后返回值所有权
} // s3 无法被释放，s2 被移动，s1 无法被释放
```

这段程序不正确，因为 s2 租借的 s1 已经将所有权移动到 s3，所以 s2 将无法继续租借使用 s1 的所有权。如果需要使用 s2 使用该值，必须重新租借：

```
fn main() {
    let s1 = String::from("hello");
    let s2 = &s1;
    let s3 = s1; // 重新从 s1 租借所有权
    takes_and_gives_back(s3);
    // s3 被当作参数移动，s3 被修改后返回值所有权
} // s3 无法被释放，s2 被移动，s1 无法被释放
```

这段程序是正确的，即使它租借了所有权，它也只享有使用权（这跟租房子是一个道理）。

如果尝试利用租借的权利来修改数据会被阻止：

```
fn main() {
    let s1 = String::from("hello");
    let s2 = &s1;
    takes_and_gives_back(s2);
    // s2 被当作参数移动，s2 被修改后返回值所有权
} // s3 无法被释放，s2 被移动，s1 无法被释放
```

这段程序不正确，因为 s2 租借的 s1 已经将所有权移动到 s3，所以 s2 将无法继续租借使用 s1 的所有权。如果需要使用 s2 使用该值，必须重新租借：

```
fn main() {
    let s1 = String::from("hello");
    let s2 = &s1;
    let s3 = s1; // 重新从 s1 租借所有权
    takes_and_gives_back(s3);
    // s3 被当作参数移动，s3 被修改后返回值所有权
} // s3 无法被释放，s2 被移动，s1 无法被释放
```

这段程序是正确的，即使它租借了所有权，它也只享有使用权（这跟租房子是一个道理）。

如果尝试利用租借的权利来修改数据会被阻止：

```
fn main() {
    let s1 = String::from("hello");
    let s2 = &s1;
    takes_and_gives_back(s2);
    // s2 被当作参数移动，s2 被修改后返回值所有权
} // s3 无法被释放，s2 被移动，s1 无法被释放
```

这段程序不正确，因为 s2 租借的 s1 已经将所有权移动到 s3，所以 s2 将无法继续租借使用 s1 的所有权。如果需要使用 s2 使用该值，必须重新租借：

```
fn main() {
    let s1 = String::from("hello");
    let s2 = &s1;
    let s3 = s1; // 重新从 s1 租借所有权
    takes_and_gives_back(s3);
    // s3 被当作参数移动，s3 被修改后返回值所有权
} // s3 无法被释放，s2 被移动，s1 无法被释放
```

这段程序是正确的，即使它租借了所有权，它也只享有使用权（这跟租房子是一个道理）。

如果尝试利用租借的权利来修改数据会被阻止：

```
fn main() {
    let s1 = String::from("hello");
    let s2 = &s1;
    takes_and_gives_back(s2);
    // s2 被当作参数移动，s2 被修改后返回值所有权
} // s3 无法被释放，s2 被移动，s1 无法被释放
```

这段程序不正确，因为 s2 租借的 s1 已经将所有权移动到 s3，所以 s2 将无法继续租借使用 s1 的所有权。如果需要使用 s2 使用该值，必须重新租借：

```
fn main() {
    let s1 = String::from("hello");
    let s2 = &s1;
    let s3 = s1; // 重新从 s1 租借所有权
    takes_and_gives_back(s3);
    // s3 被当作参数移动，s3 被修改后返回值所有权
} // s3 无法被释放，s2 被移动，s1 无法被释放
```

这段程序是正确的，即使它租借了所有权，它也只享有使用权（这跟租房子是一个道理）。

如果尝试利用租借的权利来修改数据会被阻止：

```
fn main() {
    let s1 = String::from("hello");
    let s2 = &s1;
    takes_and_gives_back(s2);
    // s2 被当作参数移动，s2 被修改后返回值所有权
} // s3 无法被释放，s2 被移动，s1 无法被释放
```

这段程序不正确，因为 s2 租借的 s1 已经将所有权移动到 s3，所以 s2 将无法继续租借使用 s1 的所有权。如果需要使用 s2 使用该值，必须重新租借：

```
fn main() {
    let s1 = String::from("hello");
    let s2 = &s1;
    let s3 = s1; // 重新从 s1 租借所有权
    takes_and_gives_back(s3);
    // s3 被当作参数移动，s3 被修改后返回值所有权
} // s3 无法被释放，s2 被移动，s1 无法被释放
```

这段程序是正确的，即使它租借了所有权，它也只享有使用权（这跟租房子是一个道理）。

如果尝试利用租借的权利来修改数据会被阻止：

```
fn main() {
    let s1 = String::from("hello");
    let s2 = &s1;
    takes_and_gives_back(s2);
    // s2 被当作参数移动，s2 被修改后返回值所有权
} // s3 无法被释放，s2 被移动，s1 无法被释放
```

这段程序不正确，因为 s2 租借的 s1 已经将所有权移动到 s3，所以 s2 将无法继续租借使用 s1 的所有权。如果需要使用 s2 使用该值，必须重新租借：

```
fn main() {
    let s1 = String::from("hello");
    let s2 = &s1;
    let s3 = s1; // 重新从 s1 租借所有权
    takes_and_gives_back(s3);
    // s3 被当作参数移动，s3 被修改后返回值所有权
} // s3 无法被释放，s2 被移动，s1 无法被释放
```

这段程序是正确的，即使它租借了所有权，它也只享有使用权（这跟租房子是一个道理）。

如果尝试利用租借的权利来修改数据会被阻止：

```
fn main() {
    let s1 = String::from("hello");
    let s2 = &s1;
    takes_and_gives_back(s2);
    // s2 被当作参数移动，s2 被修改后返回值所有权
} // s3 无法被释放，s2 被移动，s1 无法被释放
```

这段程序不正确，因为 s2 租借的 s1 已经将所有权移动到 s3，所以 s2 将无法继续租借使用 s1 的所有权。如果需要使用 s2 使用该值，必须重新租借：

&lt;pre

## Rust Slice (切片) 类型

切片 (Slice) 是对数据值的部分引用。

切片这个名字往往出现在生物课上，我们做样本玻片的时候要从生物体上获取切片，以供在显微镜上观察。在 Rust 中，切片的意思大致也是这样，只不过它从数据取材引用。

### 字符串切片

最简单、最常用的数据切片类型是字符串切片 (String Slice)。

#### 实例

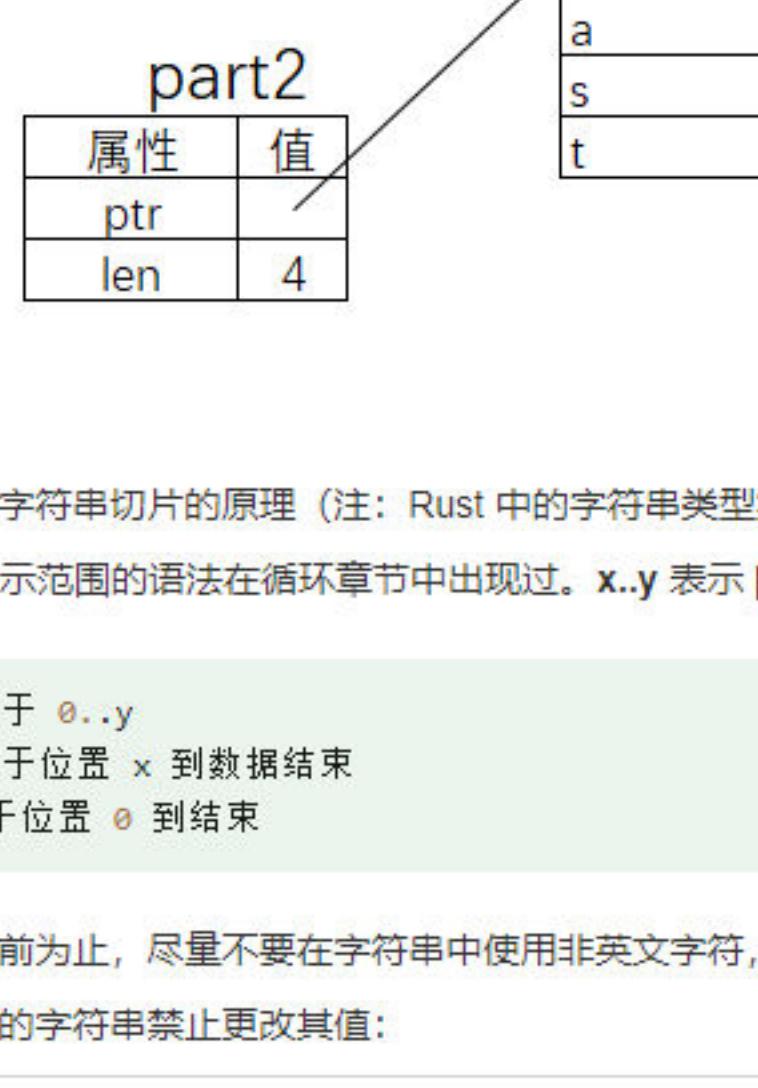
```
fn main() {
    let s = String::from("broadcast");

    let part1 = &s[0..5];
    let part2 = &s[5..9];

    println!("{}={}{+}{}", s, part1, part2);
}
```

运行结果：

```
broadcast=broad+cast
```



上图解释了字符串切片的原理（注：Rust 中的字符串类型实质上记录了字符在内存中的起始位置和其长度，我们暂时了解到这一点）。

使用 `..` 表示范围的语法在循环章节中出现过。`x..y` 表示  $[x, y)$  的数学含义。`..` 两边可以没有运算数：

```
..y 等价于 0..y
x.. 等价于位置 x 到数据结束
.. 等价于位置 0 到结束
```

注意：到目前为止，尽量不要在字符串中使用非英文字符，因为编码的问题。具体原因会在“字符串”章节叙述。

被切片引用的字符串禁止更改其值：

#### 实例

```
fn main() {
    let mut s = String::from("runoob");
    let slice = &s[0..3];
    s.push_str("yes!"); // 错误
    println!("slice = {}", slice);
}
```

这段程序不正确。

`s` 被部分引用，禁止更改其值。

实际上，到目前为止你一定疑惑为什么每一次使用字符串都要这样写 `String::from("runoob")`，直接写 "runoob" 不行吗？

事已至此我们必须分辨这两者概念的区别了。在 Rust 中有两种常用的字符串类型：str 和 String。str 是 Rust 核心语言类型，就是本章一直在讲的字符串切片 (String Slice)，常常以引用的形式出现 (`&str`)。

凡是用双引号包括的字符串常量整体的类型性质都是 `&str`：

```
let s = "hello";
```

这里的 `s` 就是一个 `&str` 类型的变量。

`String` 类型是 Rust 标准公共库提供的一种数据类型，它的功能更完善——它支持字符串的追加、清空等实用的操作。`String` 和 `str` 除了同样拥有一个字符开始位置属性和一个字符串长度属性以外还有一个容量 (capacity) 属性。

`String` 和 `str` 都支持切片，切片的结果是 `&str` 类型的数据。

注意：切片结果必须是引用类型，但开发者必须自己明示这一点：

```
let slice = &s[0..3];
```

有一个快速的办法可以将 `String` 转换成 `&str`：

```
let s1 = String::from("hello");
let s2 = &s1[..];
```

## 非字符串切片

除了字符串以外，其他一些线性数据结构也支持切片操作，例如数组：

#### 实例

```
fn main() {
    let arr = [1, 3, 5, 7, 9];
    let part = &arr[0..3];
    for i in part.iter() {
        println!("{}", i);
    }
}
```

运行结果：

```
1
3
5
```

## Rust 结构体

Rust 中的结构体 (Struct) 与元组 (Tuple) 都可以将若干个类型不一定相同的数据捆绑在一起形成整体，但结构体的每个成员和其本身都有一个名字，这样访问它成员的时候就不用记住下标了。元组常用于非定义的多值传递，而结构体用于规范常用的数据结构。结构体的每个成员叫做“字段”。

### 结构体定义

这是一个结构体定义：

```
struct Site {
    domain: String,
    name: String,
    nation: String,
    found: u32
}
```

注意：如果你常用 C/C++，请记住在 Rust 里 struct 语句仅用来定义，不能声明实例，结尾不需要 ; 符号，而且每个字段定义之后用 , 分隔。

### 结构体实例

Rust 很多地方受 JavaScript 影响，在实例化结构体的时候用 JSON 对象的 key: value 语法来实现定义：

#### 实例

```
let runoob = Site {
    domain: String::from("www.runoob.com"),
    name: String::from("RUNOOB"),
    nation: String::from("China"),
    found: 2013
};
```

如果你不了解 JSON 对象，你可以不用管它，记住格式就可以了：

```
结构体类名 {
    字段名 : 字段值,
    ...
}
```

这样的好处是不仅使程序更加直观，还不需要按照定义的顺序来输入成员的值。

如果正在实例化的结构体有字段名称和现存变量名称一样的，可以简化书写：

#### 实例

```
let domain = String::from("www.runoob.com");
let name = String::from("RUNOOB");
let runoob = Site {
    domain, // 等同于 domain: domain,
    name, // 等同于 name: name,
    nation: String::from("China"),
    traffic: 2013
};
```

有这样一种情况：你想要新建一个结构体的实例，其中大部分属性需要被设置成与现存的一个结构体属性一样，仅需更改其中的一两个字段的值，可以使用结构体更新语法：

```
let site = Site {
    domain: String::from("www.runoob.com"),
    name: String::from("RUNOOB"),
    ..runoob
};
```

注意：..runoob 后面不可以有逗号。这种语法不允许一成不变的复制另一个结构体实例，意思就是说至少重新设定一个字段的值才能引用其他实例的值。

### 元组结构体

有一种更简单的定义和使用结构体的方式：元组结构体。

元组结构体是一种形式是元组的结构体。

与元组的区别是它有名字和固定的类型格式。它存在的意义是为了处理那些需要定义类型（经常使用）又不想太复杂的简单数据：

```
struct Color(u8, u8, u8);
struct Point(f64, f64);
```

```
let black = Color(0, 0, 0);
let origin = Point(0.0, 0.0);
```

“颜色”和“点坐标”是常用的两种数据类型，但如果实例化时写个大括号再写上两个名字就为了可读性牺牲了便捷性，Rust 不会遗留这个问题。元组结构体对象的使用方式和元组一样，通过 . 和下标来进行访问：

#### 实例

```
fn main() {
    struct Color(u8, u8, u8);
    struct Point(f64, f64);

    let black = Color(0, 0, 0);
    let origin = Point(0.0, 0.0);
}
```

运行结果：

```
black = (0, 0, 0)
origin = (0, 0)
```

## 结构体所有权

结构体必须掌握字段值所有权，因为结构体失效的时候会释放所有字段。

这就是为什么本章的案例中使用了 String 类型而不使用 &str 的原因。

但这不意味着结构体中不定义引用型字段，这需要通过“生命周期”机制来实现。

但现在还难以说明“生命周期”概念，所以只能在后面章节说明。

### 输出结构体

调试中，完整地显示出一个结构体实例是非常有用的。但如果手动的书写一个格式会非常的不方便。所以 Rust 提供了一个方便地输出一个整个结构体的方法：

#### 实例

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}
```

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}
```

```
fn main() {
    let rect1 = Rectangle { width: 30, height: 50 };
    println!("rect1's area is {}", rect1.area());
}
```

如第一行所示：一定要导入调试库 #[derive(Debug)]，之后在 println 和 print 宏中就可以用 {:?} 占位符输出一整个结构体：

```
rect1 is Rectangle { width: 30, height: 50 }
```

如果属性较多的话可以使用另一个占位符 {:#?}。

输出结果：

```
rect1 is Rectangle {
    width: 30,
    height: 50
}
```

### 结构体方法

方法 (Method) 和函数 (Function) 类似，只不过它是用来操作结构体实例的。

如果你学习过一些面向对象的语言，那你一定很清楚函数一般放在类定义里并在函数中用 this 表示所操作的实例。

Rust 语言不是面向对象的，从它所有权机制的创新可以看出这一点。但是面向对象的珍贵思想可以在 Rust 实现。

结构体方法的第一个参数必须是 &self，不需声明类型，因为 self 不是一种风格而是关键字。

计算一个矩形的面积：

#### 实例

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}
```

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}
```

```
fn main() {
    let rect1 = Rectangle { width: 30, height: 50 };
    println!("rect1's area is {}", rect1.area());
}
```

输出结果：

```
rect1 is Rectangle { width: 30, height: 50 }
```

请注意，在调用结构体方法的时候不需要填写 self，这是出于对使用方便性的考虑。

一个参数的例子：

#### 实例

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

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}
```

```
fn main() {
    let rect1 = Rectangle { width: 30, height: 50 };
    let rect2 = Rectangle { width: 40, height: 20 };
    println!("rect1's area is {}", rect1.area());
}
```

输出结果：

```
rect1 is Rectangle { width: 30, height: 50 }
```

贴士：结构体 impl 块可以写几次，效果相当于它们内容的拼接！

### 单元结构体

结构体可以只作为一种象征而无需任何成员：

```
struct UnitStruct;
```

我们称这种没有身体的结构体为单元结构体 (Unit Struct)。

## 1 篇笔记

## 写笔记

引用结构体成员给其他变量赋值时，要注意：所有权的转移可能会破坏结构体变量的完整性。例如：

55

```
struct Dog {
    name: String,
    age: i8
}
```

```
fn main() {
    let mydog = Dog {
        name: String::from("wangcai"),
        age: 3,
    };
    let str = mydog.name;
    println!("str={}", str);
    println!("mydog: name={}, age={}", mydog.name, mydog.age);
}
```

编译会出错：

```
11 |     let str = mydog.name;
|          ----- value moved here
12 |     println!("str={}", str);
13 |     println!("mydog: name={}, age={}", mydog.name, mydog.age);
|          ^^^^^^^^^^ value borrowed here after move
```

11行，用mydog.name给str赋值时，所有权就move到的str变量。

13行，打印时引用mydog.name，此时已经不存在，无法再使用。

11行应该改为：

```
let str = mydog.name.clone();
```

clone()会创建mydog.name的一个副本。

lushidegreen 1年前 (2022-06-10)

这个程序计算 rect1 是否比 rect2 更宽。

## 结构体关联函数

之所以“结构体方法”不叫“结构体函数”是因为“函数”这个名字留给了这种函数：它在 impl 块中却没有 &self 参数。

如果你学习过一些面向对象的语言，那你一定很清楚函数一般放在类定义里并在函数中用 this 表示所操作的实例。

Rust 语言不是面向对象的，从它所有权机制的创新可以看出这一点。但是面向对象的珍贵思想可以在 Rust 实现。

结构体方法的第一个参数必须是 &self，不需声明类型，因为 self 不是一种风格而是关键字。

计算一个矩形的面积：

#### 实例

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}
```

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}
```

```
fn main() {
    let rect1 = Rectangle { width: 30, height: 50 };
    let rect2 = Rectangle { width: 40, height: 20 };
    println!("rect1's area is {}", rect1.area());
}
```

输出结果：

```
rect1 is Rectangle { width: 30, height: 50 }
```

贴士：结构体 impl 块可以写几次，效果相当于它们内容的拼接！

### 单元结构体

结构体可以只作为一种象征而无需任何成员：

```
struct UnitStruct;
```

我们称这种没有身体的结构体为单元结构体 (Unit Struct)。

## Rust 枚举类

枚举类在 Rust 中并不像其他编程语言中的概念那样简单，但依然可以十分简单的使用：

### 实例

```
#[derive(Debug)]
enum Book {
    Papery,
    Electronic
}

fn main() {
    let book = Book::Papery;
    println!("{}: {:?}", book);
}
```

运行结果：

Papery

书分为纸质书 (Papery book) 和电子书 (Electronic book)。

如果你现在正在开发一个图书管理系统，你需要描述两种书的不同属性（纸质书有索书号，电子书只有 URL），你可以为枚举类成员添加元组属性描述：

```
enum Book {
    Papery(u32),
    Electronic(String),
}

let book = Book::Papery(1001);
let ebook = Book::Electronic(String::from("url://..."));
```

如果你想为属性命名，可以用结构体语法：

```
enum Book {
    Papery { index: u32 },
    Electronic { url: String },
}
let book = Book::Papery{index: 1001};
```

虽然可以如此命名，但请注意，并不能像访问结构体字段一样访问枚举类绑定的属性。访问的方法在 match 语法中。

### match 语法

枚举的目的是对某一类事物的分类，分类的目的是为了对不同的情况进行描述。基于这个原理，往往枚举类最终都会被分支结构处理（许多语言中的 switch）。switch 语法很经典，但在 Rust 中并不支持，很多语言摒弃 switch 的原因都是因为 switch 容易存在因忘记添加 break 而产生的串接运行问题，Java 和 C# 这类语言通过安全检查杜绝这种情况出现。

Rust 通过 match 语句来实现分支结构。先认识一下如何用 match 处理枚举类：

### 实例

```
fn main() {
    enum Book {
        Papery { index: u32 },
        Electronic { url: String },
    }

    let book = Book::Papery{index: 1001};
    let ebook = Book::Electronic{url: String::from("url...")};

    match book {
        Book::Papery { index } => {
            println!("Papery book {}", index);
        },
        Book::Electronic { url } => {
            println!("E-book {}", url);
        }
    }
}
```

运行结果：

Papery book 1001

match 块也可以当作函数表达式来对待，它也是可以有返回值的：

```
match 枚举类实例 {
    分类1 => 返回值表达式,
    分类2 => 返回值表达式,
    ...
}
```

但是所有返回值表达式的类型必须一样！

如果把枚举类附加属性定义成元组，在 match 块中需要临时指定一个名字：

### 实例

```
enum Book {
    Papery(u32),
    Electronic {url: String},
}
let book = Book::Papery(1001);

match book {
    Book::Papery(i) => {
        println!("{}", i);
    },
    Book::Electronic { url } => {
        println!("{}", url);
    }
}
```

match 除了能够对枚举类进行分支选择以外，还可以对整数、浮点数、字符和字符串切片引用 (&str) 类型的数据进行分支选择。其中，浮点数类型被分支选择虽然合法，但不推荐这样使用，因为精度问题可能会导致分支错误。

对非枚举类进行分支选择时必须注意处理例外情况，即使在例外情况下没有任何要做的事，例外情况用下划线 \_ 表示：

### 实例

```
fn main() {
    let t = "abc";
    match t {
        "abc" => println!("Yes"),
        _ => {},
    }
}
```

运行结果：

Hello

如果你的变量刚开始是空值，你体谅一下编译器，它怎么知道值不为空的时候变量是什么类型的呢？

所以初始值为空的 Option 必须明确类型：

### 实例

```
fn main() {
    let opt: Option<&str> = Option::None;
    match opt {
        Option::Some(something) => {
            println!("{}: {}", something);
        },
        Option::None => {
            println!("opt is nothing");
        }
    }
}
```

放入主函数运行结果：

zero

这段程序的目的是判断 i 是否是数字 0，如果是就打印 zero。

现在用 if let 语法缩短这段代码：

```
let i = 0;
if let 0 = i {
    println!("zero");
}
```

if let 语法格式如下：

```
if let 变量名 = 源变量 {
    语句块
}
```

可以在之后添加一个 else 块来处理例外情况。

if let 语法可以认为是只区分两种情况的 match 语句的“语法糖”（语法糖指的是某种语法的原理相同的便捷替代品）。

对于枚举类依然适用：

### 实例

```
fn main() {
    enum Book {
        Papery(u32),
        Electronic(String)
    }

    let book = Book::Electronic(String::from("url"));

    if let Book::Papery(index) = book {
        println!("Papery {}", index);
    } else {
        println!("Not papery book");
    }
}
```

运行结果：

Not papery book

点我分享笔记

这种设计会让空值编程变得不容易，但这正是构建一个稳定高效的系统所需要的。由于 Option 是 Rust 编译器默认引入的，在使用时可以省略 Option:: 直接写 None 或者 Some()。

Option 是一种特殊的枚举类，它可以含值分支选择：

### 实例

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

如果你想定义一个可以为空值的类，你可以这样：

```
let opt = Option::Some("Hello");
```

如果你想针对 opt 执行某些操作，你必须先判断它是否是 Option::None：

### 实例

```
fn main() {
    let opt: Option<&str> = Option::None;
    match opt {
        Option::Some(something) => {
            println!("{}: {}", something);
        },
        Option::None => {
            println!("opt is nothing");
        }
    }
}
```

这种设计会让空值编程变得不容易，但这正是构建一个稳定高效的系统所需要的。由于 Option 是 Rust 编译器默认引入的，在使用时可以省略 Option:: 直接写 None 或者 Some()。

Option 是一种特殊的枚举类，它可以含值分支选择：

### 实例

```
fn main() {
    enum Book {
        Papery(u32),
        Electronic(String)
    }

    let book = Book::Electronic(String::from("url"));

    if let Book::Papery(index) = book {
        println!("Papery {}", index);
    } else {
        println!("Not papery book");
    }
}
```

运行结果：

Not papery book

点我分享笔记

这种设计会让空值编程变得不容易，但这正是构建一个稳定高效的系统所需要的。由于 Option 是 Rust 编译器默认引入的，在使用时可以省略 Option:: 直接写 None 或者 Some()。

Option 是一种特殊的枚举类，它可以含值分支选择：

### 实例

```
fn main() {
    enum Book {
        Papery(u32),
        Electronic(String)
    }

    let book = Book::Electronic(String::from("url"));

    if let Book::Papery(index) = book {
        println!("Papery {}", index);
    } else {
        println!("Not papery book");
    }
}
```

运行结果：

Not papery book

点我分享笔记

这种设计会让空值编程变得不容易，但这正是构建一个稳定高效的系统所需要的。由于 Option 是 Rust 编译器默认引入的，在使用时可以省略 Option:: 直接写 None 或者 Some()。

Option 是一种特殊的枚举类，它可以含值分支选择：

### 实例

```
fn main() {
    enum Book {
        Papery(u32),
        Electronic(String)
    }

    let book = Book::Electronic(String::from("url"));

    if let Book::Papery(index) = book {
        println!("Papery {}", index);
    } else {
        println!("Not papery book");
    }
}
```

运行结果：

Not papery book

点我分享笔记

这种设计会让空值编程变得不容易，但这正是构建一个稳定高效的系统所需要的。由于 Option 是 Rust 编译器默认引入的，在使用时可以省略 Option:: 直接写 None 或者 Some()。

Option 是一种特殊的枚举类，它可以含值分支选择：

### 实例

```
fn main() {
    enum Book {
        Papery(u32),
        Electronic(String)
    }

    let book = Book::Electronic(String::from("url"));

    if let Book::Papery(index) = book {
        println!("Papery {}", index);
    } else {
        println!("Not papery book");
    }
}
```

运行结果：

Not papery book

点我分享笔记

这种设计会让空值编程变得不容易，但这正是构建一个稳定高效的系统所需要的。由于 Option 是 Rust 编译器默认引入的，在使用时可以省略 Option:: 直接写 None 或者 Some()。

Option 是一种特殊的枚举类，它可以含值分支选择：

### 实例

```
fn main() {
    enum Book {
        Papery(u32),
        Electronic(String)
    }

    let book = Book::Electronic(String::from("url"));

    if let Book::Papery(index) = book {
        println!("Papery {}", index);
    } else {
        println!("Not papery book");
    }
}
```

运行结果：

Not papery book

点我分享笔记

这种设计会让空值编程变得不容易，但这正是构建一个稳定高效的系统所需要的。由于 Option 是 Rust 编译器默认引入的，在使用时可以省略 Option:: 直接写 None 或者 Some()。

Option 是一种特殊的枚举类，它可以含值分支选择：

### 实例

```
fn main() {
    enum Book {
        Papery(u32),
        Electronic(String)
    }

    let book = Book::Electronic(String::from("url"));

    if let Book::Papery(index) = book {
        println!("Papery {}", index);
    } else {
        println!("Not papery book");
    }
}
```

运行结果：

Not papery book

点我分享笔记

这种设计会让空值编程变得不容易，但这正是构建一个稳定高效的系统所需要的。由于 Option 是 Rust 编译器默认引入的，在使用时可以省略 Option:: 直接写 None 或者 Some()。

Option 是一种特殊的枚举类，它可以含值分支选择：

### 实例

```
fn main() {
    enum Book {
        Papery(u32),
        Electronic(String)
    }

    let book = Book::Electronic(String::from("url"));

    if let Book::Papery(index) = book {
        println!("Papery {}", index);
    } else {
        println!("Not papery book");
    }
}
```

运行结果：

Not papery book

点我分享笔记

这种设计会让空值编程变得不容易，但这正是构建一个稳定高效的系统所需要的。由于 Option 是 Rust 编译器默认引入的，在使用时可以省略 Option:: 直接写 None 或者 Some()。

Option 是一种特殊的枚举类，它可以含值分支选择：

### 实例

```
fn main() {
    enum Book {
        Papery(u32),
        Electronic(String)
    }

    let book = Book::Electronic(String::from("url"));

    if let Book::Papery(index) = book {
        println!("Papery {}", index);
    } else {
        println!("Not papery book");
    }
}
```

运行结果：

Not papery book

点我分享笔记

这种设计会让空值编程变得不容易，但这正是构建一个稳定高效的系统所需要的。由于 Option 是 Rust 编译器默认引入的，在使用时可以省略 Option:: 直接写 None 或者 Some()。

Option 是一种特殊的枚举类，它可以含值分支选择：

### 实例

```
fn main() {
    enum Book {
        Papery(u32),
        Electronic(String)
    }

    let book = Book::Electronic(String::from("url"));

    if let Book::Papery(index) = book {
        println!("Papery {}", index);
    } else {
        println!("Not papery book");
    }
}
```

运行结果：

Not papery book

点我分享笔记

这种设计会让空值编程变得不容易，但这正是构建一个稳定高效的系统所需要的。由于 Option 是 Rust 编译器默认引入的，在使用时可以省略 Option:: 直接写 None 或者 Some()。

Option 是一种特殊的枚举类，它可以含值分支选择：

### 实例

```
fn main() {
    enum Book {
        Papery(u32),
        Electronic(String)
    }

    let book = Book::Electronic(String::from("url"));

    if let Book::Papery(index) = book {
        println!("Papery {}", index);
    } else {
        println!("Not papery book");
    }
}
```

运行结果：

Not papery book

点我分享笔记

这种设计会让空值编程变得不容易，但这正是构建一个稳定高效的系统所需要的。由于 Option 是 Rust 编译器默认引入的，在使用时可以省略 Option:: 直接写 None 或者 Some()。

Option 是一种特殊的枚举类，它可以含值分支选择：

### 实例

```
fn main() {
    enum Book {
        Papery(u32),
        Electronic(String)
    }

    let book = Book::Electronic(String::from("url"));

    if let Book::Papery(index) = book {
        println!("Papery {}", index);
    } else {
        println!("Not papery book");
    }
}
```

运行结果：

Not papery book

点我分享笔记

这种设计会让空值编程变得不容易，但这正是构建一个稳定高效的系统所需要的。由于 Option 是 Rust 编译器默认引入的，在使用时可以省略 Option:: 直接写 None 或者 Some()。

Option 是一种特殊的枚举类，它可以含值分支选择：

### 实例

## Rust 组织管理

任何一门编程语言如果不能组织代码都是难以深入的，几乎没有一个软件产品是由一个源文件编译而成的。

本教程到目前为止所有的程序都是在一个文件中编写的，主要是为了方便学习 Rust 语言的语法和概念。

对于一个工程来讲，组织代码是十分重要的。

Rust 中有三个重要的组织概念：箱、包、模块。

### 箱 (Crate)

"箱"是二进制程序文件或者库文件，存在于"包"中。

"箱"是树状结构的，它的树根是编译器开始运行时编译的源文件所编译的程序。

注意："二进制程序文件"不一定是"二进制可执行文件"，只能确定是包含目标机器语言的文件，文件格式随编译环境的不同而不同。

### 包 (Package)

当我们使用 Cargo 执行 new 命令创建 Rust 工程时，工程目录下会建立一个 Cargo.toml 文件。工程的实质就是一个包，包必须由一个 Cargo.toml 文件来管理，该文件描述了包的基本信息以及依赖项。

一个包最多包含一个库"箱"，可以包含任意数量的二进制"箱"，但是至少包含一个"箱"（不管是库还是二进制"箱"）。

当使用 cargo new 命令创建完包之后，src 目录下会生成一个 main.rs 源文件，Cargo 默认这个文件为二进制箱的根，编译之后的二进制箱将与包名相同。

### 模块 (Module)

对于一个软件工程来说，我们往往按照所使用的编程语言的组织规范来进行组织，组织模块的主要结构往往是树。Java 组织功能模块的主要单位是类，而 JavaScript 组织模块的主要方式是 function。

这些先进的语言的组织单位可以层层包含，就像文件系统的目录结构一样。Rust 中的组织单位是模块 (Module)。

```
mod nation {
    mod government {
        fn govern() {}
    }
    mod congress {
        fn legislate() {}
    }
    mod court {
        fn judicial() {}
    }
}
```

这是一段描述法治国家的程序：国家 (nation) 包括政府 (government)、议会 (congress) 和法院 (court)，分别有行政、立法和司法的功能。我们可以把它转换成树状结构：

```
nation
|--- government
|   |--- govern
|--- congress
|   |--- legislate
|--- court
|   |--- judicial
```

在文件系统中，目录结构往往以斜杠在路径字符串中表示对象的位置，Rust 中的路径分隔符是 ::。

路径分为绝对路径和相对路径。绝对路径从 crate 关键字开始描述。相对路径从 self 或 super 关键字或一个标识符开始描述。例如：

```
crate::nation::government::govern();
```

是描述 govern 函数的绝对路径，相对路径可以表示为：

```
nation::government::govern();
```

现在你可以尝试在一个源程序里定义类似的模块结构并在主函数中使用路径。

如果你这样做，你一定会发现它不正确的地方：government 模块和其中的函数都是私有的 (private) 的，你不被允许访问它们。

### 访问权限

Rust 中有两种简单的访问权：公共 (public) 和私有 (private)。

默认情况下，如果不加修饰符，模块中的成员访问权将是私有的。

如果想使用公共权限，需要使用 pub 关键字。

对于私有的模块，只有在与其平级的位置或下级的位置才能访问，不能从其外部访问。

#### 实例

```
mod nation {
    pub mod government {
        pub fn govern() {}
    }

    mod congress {
        pub fn legislate() {}
    }

    mod court {
        fn judicial() {
            super::congress::legislate();
        }
    }
}

fn main() {
    nation::government::govern();
}
```

这段程序是能通过编译的。请注意观察 court 模块中 super 的访问方法。

如果模块中定义了结构体，结构体除了其本身是私有的以外，其字段也默认是私有的。所以如果想使用模块中的结构体以及其字段，需要 pub 声明：

#### 实例

```
mod back_of_house {
    pub struct Breakfast {
        pub toast: String,
        seasonal_fruit: String,
    }

    impl Breakfast {
        pub fn summer(toast: &str) -> Breakfast {
            Breakfast {
                toast: String::from(toast),
                seasonal_fruit: String::from("peaches"),
            }
        }
    }
}

pub fn eat_at_restaurant() {
    let mut meal = back_of_house::Breakfast::summer("Rye");
    meal.toast = String::from("wheat");
    println!("I'd like {} toast please", meal.toast);
}

fn main() {
    eat_at_restaurant();
}
```

运行结果：

```
I'd like wheat toast please
```

枚举类枚举项可以内含字段，但不具备类似的性质。

#### 实例

```
mod SomeModule {
    pub enum Person {
        King {
            name: String
        },
        Queen
    }
}

fn main() {
    let person = SomeModule::Person::King{
        name: String::from("Blue")
    };
    match person {
        SomeModule::Person::King {name} => {
            println!("{}", name);
        }
        _ => {}
    }
}
```

运行结果：

```
Blue
```

## 难以发现的模块

使用过 Java 的开发者在编程时往往非常讨厌最外层的 class 块——它的名字与文件名一模一样，因为它就表示文件容器，尽管它很繁琐但我们不得不写一遍来强调“这个类是文件所包含的类”。

不过这样有一些好处：起码它让开发者明明白白的意识到了类包装的存在，而且可以明确的描述类的继承关系。

在 Rust 中，模块就像是 Java 中的类包装，但是文件一开头就可以写一个主函数，这该如何解释呢？

每一个 Rust 文件的内容都是一个“难以发现”的模块。

让我们用两个文件来揭示这一点：

#### main.rs 文件

```
// main.rs
mod second_module;

fn main() {
    println!("This is the main module.");
    println!("{}", second_module::message());
}
```

#### second\_module.rs 文件

```
// second_module.rs
pub fn message() -> String {
    String::from("This is the 2nd module.")
}
```

运行结果：

```
This is the main module.
This is the 2nd module.
```

枚举类枚举项可以内含字段，但不具备类似的性质。

#### 实例

```
mod nation {
    pub mod government {
        pub fn govern() {}
    }
    pub fn govern() {}
}

use crate::nation::government::govern;
use crate::nation::govern as nation_govern;

fn main() {
    nation_govern();
    govern();
}
```

这里有两个 govern 函数，一个是 nation 下的，一个是 government 下的，我们用 as 将 nation 下的取别名 nation\_govern。两个名称可以同时使用。

use 关键字可以与 pub 关键字配合使用：

#### 实例

```
mod nation {
    pub mod government {
        pub fn govern() {}
    }
    pub use government::govern;
}

fn main() {
    nation::govern();
}
```

所有的系统库模块都是被默认导入的，所以在使用的时候只需要使用 use 关键字简化路径就可以方便的使用了。

#### use 关键字

use 关键字能够将模块标识符引入当前作用域：

#### 实例

```
mod nation {
    pub mod government {
        pub fn govern() {}
    }
}

use crate::nation::government::govern;
use crate::nation::govern as nation_govern;
```

这段程序能够通过编译。

因为 use 关键字把 govern 标识符导入到了当前的模块下，可以直接使用。

当然，有些情况下存在两个相同的名称，且同样需要导入，我们可以使用 as 关键字为标识符添加别名：

#### 实例

```
mod nation {
    pub mod government {
        pub fn govern() {}
    }
}

use std::f64::consts::PI;

fn main() {
    println!("{} ({PI / 2.0}.sin())", PI);
}
```

运行结果：

```
1
```

所有的系统库模块都是被默认导入的，所以在使用的时候只需要使用 use 关键字简化路径就可以方便的使用了。

#### 引用标准库

Rust 官方标准库字典：<https://doc.rust-lang.org/stable/std/all.html>

在学习了本章的概念之后，我们可以轻松的导入系统库来方便的开发程序了：

#### 实例

```
use std::f64::consts::PI;

fn main() {
    println!("{} ({PI / 2.0}.sin())", PI);
}
```

运行结果：

```
1
```

所有的系统库模块都是被默认导入的，所以在使用的时候只需要使用 use 关键字简化路径就可以方便的使用了。

#### use 关键字

use 关键字能够将模块标识符引入当前作用域：

#### 实例

```
mod nation {
    pub mod government {
        pub fn govern() {}
    }
}

use crate::nation::government::govern;
use crate::nation::govern as nation_govern;
```

这段程序能够通过编译。

因为 use 关键字把 govern 标识符导入到了当前的模块下，可以直接使用。

当然，有些情况下存在两个相同的名称，且同样需要导入，我们可以使用 as 关键字为标识符添加别名：

#### 实例

```
mod nation {
    pub mod government {
        pub fn govern() {}
    }
}

use std::f64::consts::PI;

fn main() {
    nation::govern();
}
```

运行结果：

```
1
```

所有的系统库模块都是被默认导入的，所以在使用的时候只需要使用 use 关键字简化路径就可以方便的使用了。

#### use 关键字

use 关键字能够将模块标识符引入当前作用域：

#### 实例

```
mod nation {
    pub mod government {
        pub fn govern() {}
    }
}

use crate::nation::government::govern;
use crate::nation::govern as nation_govern;
```

这段程序能够通过编译。

因为 use 关键字把 govern 标识符导入到了当前的模块下，可以直接使用。

当然，有些情况下存在两个相同的名称，且同样需要导入，我们可以使用 as 关键字为标识符添加别名：

#### 实例

```
mod nation {
    pub mod government {
        pub fn govern() {}
    }
}

use std::f64::consts::PI;

fn main() {
    nation::govern();
}
```

运行结果：

```
1
```

所有的系统库模块都是被默认导入的，所以在使用的时候只需要使用 use 关键字简化路径就可以方便的使用了。

#### use 关键字

use 关键字能够将模块标识符引入当前作用域：

#### 实例

```
mod nation {
    pub mod government {
        pub fn govern() {}
    }
}

use crate::nation::government::govern;
use crate::nation::govern as nation_govern;
```

这段程序能够通过编译。

因为 use 关键字把 govern 标识符导入到了当前的模块下，可以直接使用。

当然，有些情况下存在两个相同的名称，且同样需要导入，我们可以使用 as 关键字为标识符添加别名：

#### 实例

```
mod nation {
    pub mod government {
        pub fn govern() {}
    }
}

use std::f64::consts::PI;

fn main() {
    nation::govern();
}
```

运行结果：

```
1
```

所有的系统库模块都是被默认导入的，所以在使用的时候只需要使用 use 关键字简化路径就可以方便的使用了。

#### use 关键字

use 关键字能够将模块标识符引入当前作用域：

#### 实例

```
mod nation {
    pub mod government {
        pub fn govern() {}
    }
}

use crate::nation::government::govern;
use crate::nation::govern as nation_govern;
```

这段程序能够通过编译。

因为 use 关键字把 govern 标识符导入到了当前的模块下，可以直接使用。

当然，有些情况下存在两个相同的名称，且同样需要导入，我们可以使用 as 关键字为标识符添加别名：

#### 实例

```
mod nation {
    pub mod government {
        pub fn govern() {}
    }
}

use std::f64::consts::PI;

fn main() {
    nation::govern();
}
```

运行结果：

```
1
```

所有的系统库模块都是被默认导入的，所以在使用的时候只需要使用 use 关键字简化路径就可以方便的使用了。

#### use 关键字

use 关键字能够将模块标识符引入当前作用域：

#### 实例

```
mod nation {
    pub mod government {
        pub fn govern() {}
    }
}

use crate::nation::government::govern;
use crate::nation::govern as nation_govern;
```

这段程序能够通过编译。

因为 use 关键字把 govern 标识符导入到了当前的模块下，可以直接使用。

当然，有些情况下存在两个相同的名称，且同样需要导入，我们可以使用 as 关键字为标识符添加别名：

#### 实例

```
mod nation {
    pub mod government {
        pub fn govern() {}
    }
}

use std::f64::consts::PI;

fn main() {
    nation::govern();
}
```

运行结果：

```
1
```

## 错误处理

Rust 有一套独特的处理异常情况的机制，它并不像其它语言中的 try 机制那样简单。

首先，程序中一般会出现两种错误：可恢复错误和不可恢复错误。

可恢复错误的典型案例是文件访问错误，如果访问一个文件失败，有可能是因为它正在被占用，是正常的，我们可以通过等待来解决。

但还有一种错误是由编程中无法解决的逻辑错误导致的，例如访问数组末尾以外的位置。

大多数编程语言不区分这两种错误，并用 Exception（异常）类来表示错误。在 Rust 中没有 Exception。

对于可恢复错误用 Result<T, E> 类来处理，对于不可恢复错误使用 panic! 宏来处理。

### 不可恢复错误

本章以前没有专门介绍 Rust 宏的语法，但已经使用过了 println! 宏，因为这些宏的使用较为简单，所以暂时不需要彻底掌握它，我们可以用同样的方法先学会使用 panic! 宏的使用方法。

#### 实例

```
fn main() {
    panic!("error occurred");
    println!("Hello, Rust");
}
```

运行结果：

```
thread 'main' panicked at 'error occurred', src\main.rs:3:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace.
```

很显然，程序并不能如约运行到 println!("Hello, Rust")，而是在 panic! 宏被调用时停止了运行。

不可恢复的错误一定会导致程序受到致命的打击而终止运行。

让我们注视错误输出的两行：

- 第一行输出了 panic! 宏调用的位置以及其输出的错误信息。
- 第二行是一句提示，翻译成中文就是“通过 ‘RUST\_BACKTRACE=1’ 环境变量运行以显示回溯”。接下来我们将介绍回溯（backtrace）。

紧接着刚才的例子，我们在 VSCode 中新建一个终端：



在新建的终端里设置环境变量（不同的终端方法不同，这里介绍两种主要的方法）：

如果在 Windows 7 及以上的 Windows 系统版本中，默认使用的终端命令行是 Powershell，请使用以下命令：

```
$env:RUST_BACKTRACE=1 ; cargo run
```

如果你使用的是 Linux 或 macOS 等 UNIX 系统，一般情况下默认使用的是 bash 命令行，请使用以下命令：

```
RUST_BACKTRACE=1 cargo run
```

然后，你会看到以下文字：

```
thread 'main' panicked at 'error occurred', src\main.rs:3:5
stack backtrace:
...
11: greeting::main
    at .\src\main.rs:3
...
```

回溯是不可恢复错误的另一种处理方式，它会展开运行的栈并输出所有的信息，然后程序依然会退出。上面的省略号省略了大量的输出信息，我们可以找到我们编写的 panic! 宏触发的错误。

### 可恢复的错误

此概念十分类似于 Java 编程语言中的异常。实际上在 C 语言中我们就常常将函数返回值设置成整数来表达函数遇到的错误，在 Rust 中通过 Result<T, E> 枚举类作返回值来进行异常表达：

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

在 Rust 标准库中可能产生异常的函数的返回值都是 Result 类型的。例如：当我们尝试打开一个文件时：

#### 实例

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");
    match f {
        Ok(file) => {
            println!("File opened successfully.");
        },
        Err(err) => {
            println!("Failed to open the file.");
        }
    }
}
```

如果 hello.txt 文件不存在，会打印 "Failed to open the file."。

当然，我们在枚举类章节讲到的 if let 语法可以简化 match 语句块：

#### 实例

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");
    if let Ok(file) = f {
        println!("File opened successfully.");
    } else {
        println!("Failed to open the file.");
    }
}
```

这段程序相当于在 Result 为 Err 时调用 panic! 宏。两者的区别在于 expect 能够向 panic! 宏发送一段指定的错误信息。

### 可恢复的错误的传递

之前所讲的是接收到错误的处理方式，但是如果我们自己编写一个函数在遇到错误时想传递出去怎么办呢？

#### 实例

```
fn f(i: i32) -> Result<i32, bool> {
    if i >= 0 { Ok(i) }
    else { Err(false) }
}

fn main() {
    let r = f(10000);
    if let Ok(v) = r {
        println!("Ok: f(-1) = {}", v);
    } else {
        println!("Err");
    }
}
```

运行结果：

```
Ok: f(-1) = 10000
```

这段程序中函数 f 是错误的根源，现在我们再写一个传递错误的函数 g：

#### 实例

```
use std::fs::File;

fn main() {
    let f1 = File::open("hello.txt").unwrap();
    let f2 = File::open("hello.txt").expect("Failed to open.");
}
```

如果想使一个可恢复错误接不可恢复错误处理，Result 类提供了两个办法：unwrap() 和 expect(message: &str)：

#### 实例

```
use std::fs::File;

fn main() {
    let f1 = File::open("hello.txt").unwrap();
    let f2 = File::open("hello.txt").expect("Failed to open.");
}
```

这段程序相当于在 Result 为 Err 时调用 panic! 宏。两者的区别在于 expect 能够向 panic! 宏发送一段指定的错误信息。

### kind 方法

到此为止，Rust 似乎没有像 try 块一样可以令任何位置发生的同类异常都直接得到相同的解决的语法，但这并不意味着 Rust 实现不了：我们完全可以把 try 块在独立的函数中实现，将所有的异常都传递出去解决。实际上这才是一个分化良好的程序应当遵循的编程方法：应该注重独立功能的完整性。

但是这样需要判断 Result 的 Err 类型，获取 Err 类型的函数是 kind()。

#### 实例

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

fn read_text_from_file(path: &str) -> Result<String, io::Error> {
    let mut f = File::open(path)?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    Ok(s)
}

fn main() {
    let str_file = read_text_from_file("hello.txt");
    match str_file {
        Ok(s) => println!("{}: {}", s),
        Err(e) => {
            match e.kind() {
                io::ErrorKind::NotFound => {
                    println!("No such file");
                },
                _ => {
                    println!("Cannot read the file");
                }
            }
        }
    }
}
```

运行结果：

```
No such file
```

## Rust 泛型与特性

泛型是一个编程语言不可或缺的机制。

C++ 语言中用“模板”来实现泛型，而 C 语言中没有泛型的机制，这也导致 C 语言难以构建类型复杂的工程。

泛型机制是编程语言用于表达类型抽象的机制，一般用于功能确定、数据类型待定的类，如链表、映射表等。

### 在函数中定义泛型

这是一个对整型数字选择排序的方法：

#### 实例

```
fn max(array: &[i32]) -> i32 {
    let mut max_index = 0;
    let mut i = 1;
    while i < array.len() {
        if array[i] > array[max_index] {
            max_index = i;
        }
        i += 1;
    }
    array[max_index]
}

fn main() {
    let a = [2, 4, 6, 3, 1];
    println!("max = {}", max(&a));
}
```

运行结果：

```
max = 6
```

这是一个简单的取最大值程序，可以用于处理 i32 数字类型的数据，但无法用于 f64 类型的数据。通过使用泛型我们可以使这个函数可以利用到各个类型中去。但实际上并不是所有的数据类型都可以比大小，所以接下来一段代码并不是用来运行的，而是用来描述一下函数泛型的语法格式：

#### 实例

```
fn max<T>(array: &[T]) -> T {
    let mut max_index = 0;
    let mut i = 1;
    while i < array.len() {
        if array[i] > array[max_index] {
            max_index = i;
        }
        i += 1;
    }
    array[max_index]
}
```

### 结构体与枚举类中的泛型

在之前我们学习的 Option 和 Result 枚举类就是泛型的。

Rust 中的结构体和枚举类都可以实现泛型机制。

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

这是一个点坐标结构体，T 表示描述点坐标的数字类型。我们可以这样使用：

```
let p1 = Point {x: 1, y: 2};
let p2 = Point {x: 1.0, y: 2.0};
```

使用时并没有声明类型，这里使用的是自动类型机制，但不允许出现类型不匹配的情况如下：

```
let p = Point {x: 1, y: 2.0};
```

x 与 1 绑定时就已经将 T 设定为 i32，所以不允许再出现 f64 的类型。如果我们想让 x 与 y 用不同的数据类型表示，可以使用两个泛型标识符：

```
struct Point<T1, T2> {
    x: T1,
    y: T2
}
```

在枚举类中表示泛型的方法诸如 Option 和 Result：

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

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

结构体与枚举类都可以定义方法，那么方法也应该实现泛型的机制，否则泛型的类将无法被有效的方法操作。

#### 实例

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

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}

fn main() {
    let p = Point {x: 1, y: 2};
    println!("p.x = {}", p.x());
}
```

运行结果：

```
p.x = 1
```

注意，impl 关键字的后方必须有 <T>，因为它后面的 T 是以之为榜样的。但我们也可以为其中的一种泛型添加方法：

```
impl Point<f64> {
    fn x(&self) -> f64 {
        self.x
    }
}
```

impl 块本身的泛型并没有阻碍其内部方法具有泛型的能力：

```
impl<T, U> Point<T, U> {
    fn mixup<V, W>(self, other: Point<V, W>) -> Point<T, W> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}
```

方法 mixup 将一个 Point<T, U> 点的 x 与 Point<V, W> 点的 y 融合成一个类型为 Point<T, W> 的新点。

## 特性

特性 (trait) 概念接近于 Java 中的接口 (Interface)，但两者不完全相同。特性与接口相同的地方在于它们都是一种行为规范，可以用子标注哪些类有哪些方法。

特性在 Rust 中用 trait 表示：

```
trait Descriptive {
    fn describe(&self) -> String;
}
```

Descriptive 规定了实现者必需有 `describe(&self) -> String` 方法。

我们用它实现一个结构体：

#### 实例

```
struct Person {
    name: String,
    age: u8
}
```

```
impl Descriptive for Person {
    fn describe(&self) -> String {
        format!("{} {}", self.name, self.age)
    }
}
```

格式是：

```
impl <特性名> for <所实现的类型名>
```

Rust 同一个类可以实现多个特性，每个 impl 块只能实现一个。

### 默认特性

这是特性与接口的不同点：接口只能规范方法而不能定义方法，但特性可以定义方法作为默认方法，因为是“默认”，所以对象既可以重新定义方法，也可以不重新定义方法使用默认的方法：

#### 实例

```
trait Descriptive {
    fn describe(&self) -> String {
        String::from("[Object]")
    }
}
```

```
struct Person {
    name: String,
    age: u8
}
```

```
impl Descriptive for Person {
    fn describe(&self) -> String {
        format!("{} {}", self.name, self.age)
    }
}
```

```
fn main() {
    let cali = Person {
        name: String::from("Cali"),
        age: 24
    };
    println!("{} {}", cali.describe());
}
```

运行结果：

```
Cali 24
```

如果我们将 `impl Descriptive for Person` 块中的内容去掉，那么运行结果就是：

```
[Object]
```

### 特性做参数

很多情况下我们需要传递一个函数做参数，例如回调函数、设置按钮事件等。在 Java 中函数必须以接口实现的类实例来传递，在 Rust 中可以通过传递特性参数来实现：

```
fn output(object: impl Descriptive) {
    println!("{} {}", object.describe());
}
```

任何实现了 Descriptive 特性的对象都可以作为这个函数的参数，这个函数没必要了解传入对象有没有其他属性或方法，只需要了解它一定有 Descriptive 特性规范的方法就可以了。当然，此函数内也无法使用其他的属性与方法。

特性参数还可以用这种等效语法实现：

```
fn output<T: Descriptive>(object: T) {
    println!("{} {}", object.describe());
}
```

这是一种风格类似泛型的语法糖，这种语法糖在有多个参数类型均是特性的情况下十分实用：

```
fn output_two<T: Descriptive>(arg1: T, arg2: T) {
    println!("{} {}", arg1.describe());
    println!("{} {}", arg2.describe());
}
```

特性作类型表示时如果涉及多个特性，可以用 + 符号表示，例如：

```
fn some_function<T: Display + Clone, U: Clone + Debug>(t: T, u: U)
```

可以简化成：

```
fn some_function<T, U>(t: T, u: U) -> i32
    where T: Display + Clone,
          U: Clone + Debug
```

在了解这个语法之后，泛型章节中的“取最大值”案例就可以真正实现了：

```
trait Comparable {
    fn compare(&self, object: &Self) -> i8;
}
```

```
fn max<T: Comparable>(array: &[T]) -> &T {
    let mut max_index = 0;
    let mut i = 1;
    while i < array.len() {
        if array[i].compare(&array[max_index]) > 0 {
            max_index = i;
        }
        i += 1;
    }
    &array[max_index]
}
```

```
impl Comparable for f64 {
    fn compare(&self, object: &f64) -> i8 {
        if &self > &object { 1 }
        else if &self == &object { 0 }
        else { -1 }
    }
}
```

```
fn main() {
    let arr = [1.0, 3.0, 5.0, 4.0, 2.0];
    println!("maximum of arr is {}", max(&arr));
}
```

运行结果：

```
maximum of arr is 5
```

**Tip:** 由于需要声明 `compare` 函数的第二参数必须与实现该特性的类型相同，所以 `Self`（注意大小写）关键字就代表了当前类型（不是实例）本身。

### 特性做返回值

特性做返回值格式如下：

```
fn person() -> impl Descriptive {
    Person {
        name: String::from("Cali"),
        age: 24
    }
}
```

但是有一点，特性做返回值只接受实现了该特性的对象做返回值且在同一个函数中所有可能的返回值类型必须完全一样。比如结构体 A 与结构体 B 都实现了特性 Trait，下面这个函数就是错误的：

```
fn some_function(bool b) -> impl Descriptive {
    if b {
        return A {};
    } else {
        return B {};
    }
}
```

### 有条件实现方法

impl 功能十分强大，我们可以用它实现类的方法。但对于泛型类来说，有时我们需要区分一下它所属的泛型已经实现的方法来决定它接下来该实现的方法：

```
struct A<T> {}

impl<T: B + C> A<T> {
    fn d(&self) {}
}
```

这段代码声明了 `A<T>` 类型必须在 T 已经实现 B 和 C 特性的前提下才能有效实现此 impl 块。

## Rust 生命周期

Rust 生命周期机制是与所有权机制同等重要的资源管理机制。

之所以引入这个概念主要是应对复杂类型系统中资源管理的问题。

引用是对待复杂类型时必不可少的机制，毕竟复杂类型的数据不能被处理器轻易地复制和计算。

但引用往往导致极其复杂的资源管理问题，首先认识一下悬引用：

### 实例

```
{
    let r;

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

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

这段代码是不会通过 Rust 编译器的，原因是 r 所引用的值已经在使用之前被释放。

上图中的绿色范围 'a' 表示 r 的生命周期，蓝色范围 'b' 表示 x 的生命周期。很显然，'b' 比 'a' 小得多，引用必须在值的生命周期以内才有效。

一直以来我们都在结构体中使用 String 而不用 &str，我们用一个案例解释原因：

### 实例

```
fn longer(s1: &str, s2: &str) -> &str {
    if s2.len() > s1.len() {
        s2
    } else {
        s1
    }
}
```

longer 函数取 s1 和 s2 两个字符串切片中较长的一个返回其引用值。但只这段代码不会通过编译，原因是返回值引用可能会返回过期的引用：

### 实例

```
fn main() {
    let r;
    {
        let s1 = "rust";
        let s2 = "ecmascript";
        r = longer(s1, s2);
    }
    println!("{} is longer", r);
}
```

这段程序中虽然经过了比较，但 r 被使用的时候源值 s1 和 s2 都已经失效了。当然我们可以把 r 的使用移到 s1 和 s2 的生命周期范围以内防止这种错误的发生，但对于函数来说，它并不能知道自己以外的地方是什么情况，它为了保障自己传递出去的值是正常的，必选所有权原则消除一切危险，所以 longer 函数并不能通过编译。

### 生命周期注释

生命周期注释是描述引用生命周期的办法。

虽然这样并不能够改变引用的生命周期，但可以在合适的地方声明两个引用的生命周期一致。

生命周期注释用单引号开头，跟着一个小写字母单词：

```
&i32      // 常规引用
&'a i32   // 含有生命周期注释的引用
&'a mut i32 // 可变型含有生命周期注释的引用
```

让我们用生命周期注释改造 longer 函数：

### 实例

```
fn longer<'a>(s1: &'a str, s2: &'a str) -> &'a str {
    if s2.len() > s1.len() {
        s2
    } else {
        s1
    }
}
```

我们需要用泛型声明来规范生命周期的名称，随后函数返回值的生命周期将与两个参数的生命周期一致，所以在调用时可以这样写：

### 实例

```
fn main() {
    let r;
    {
        let s1 = "rust";
        let s2 = "ecmascript";
        r = longer(s1, s2);
        println!("{} is longer", r);
    }
}
```

以上两段程序结合的运行结果：

```
ecmascript is longer
```

注意：别忘记了自动类型判断的原则。

### 结构体中使用字符串切片引用

这是之前留下的疑问，在此解答：

### 实例

```
fn main() {
    struct Str<'a> {
        content: &'a str
    }
    let s = Str {
        content: "string_slice"
    };
    println!("s.content = {}", s.content);
}
```

运行结果：

```
s.content = string_slice
```

如果对结构体 Str 有方法定义：

### 实例

```
impl<'a> Str<'a> {
    fn get_content(&self) -> &str {
        self.content
    }
}
```

这里返回值并没有生命周期注释，但是加上也无妨。这是一个历史问题，早期 Rust 不支持生命周期自动判断，所有的生命周期必须严格声明，但主流稳定版本的 Rust 已经支持了这个功能。

### 静态生命周期

生命周期注释有一个特别的：'static。所有用双引号括的字符串常量所代表的精确数据类型都是 &static str，'static 所表示的生命周期从程序运行开始到程序运行结束。

## 泛型、特性与生命周期协同作战

### 实例

```
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
    }
}
```

这段程序出自 Rust 圣经，是一个同时使用了泛型、特性、生命周期机制的程序，不强求，可以体验，毕竟早晚用得到！

## 2 篇笔记

## 写笔记

### longer 添加生命周期后：

```
fn longer<'a>(s1: &'a str, s2: &'a str) -> &'a str
```

67 下面的代码可以运行，这里 s1, s2 是被 copy 给了 r。

```
fn main() {
    let r;
    {
        let s1 = "rust";
        let s2 = "ecmascript";
        r = longer(s1, s2);
    }
    println!("{} is longer", r);
}
```

如果是下面这样则不可以编译通过，因为 s1, s2 没有 copy trait，内存里的值会在 {} 执行完后通过 drop 自动清理，把 println 语句移动到 {} 作用域内则可以正常运行。

```
fn main() {
    let r;
    {
        let s1 = String::from("rust");
        let s2 = String::from("ecmascript");
        r = longer(s1, s2);
    }
    println!("{} is longer", r);
}
```

沙发 3年前 [2021-03-22]

### 细化下前面同学的论述：

```
fn longer<'a>(s1: &'a str, s2: &'a str) -> &'a str
```

17 fn main() {
 let r;
 {

```
        let s1 = "rust";
        let s2 = "ecmascript";
        r = longer(s1, s2);
    }
    println!("{} is longer", r);
}
```

这里的字符串“rust”、“ecmascript”都是字符串字面常量(string literals)，生命周期持续到整个程序运行期间。s1、s2 都是借用形式&str，r 没有拷贝整个字符串，只是“拷贝”了对字符串字面常量的引用；因此执行效率还是很高的；并且 r 内的值，就算出了 println 前的花括号，生命周期还在持续。

Keilion 8个月前 [01-10]

## Rust 文件与 IO

本章介绍 Rust 语言的 I/O 操作。

### 接收命令行参数

命令行程序是计算机程序最基础的存在形式，几乎所有的操作系统都支持命令行程序并将可视化程序的运行基于命令行机制。

命令行程序必须能够接收来自命令行环境的参数，这些参数往往在一条命令行的命令之后以空格符分隔。

在很多语言中（如 Java 和 C/C++）环境参数是以主函数的参数（常常是一个字符串数组）传递给程序的，但在 Rust 中主函数是个无参数，环境参数需要开发者通过 std::env 模块取出，过程十分简单：

#### 实例

```
fn main() {
    let args = std::env::args();
    println!("{}: {}", args);
}
```

现在直接运行程序：

```
Args { inner: ["D:\\rust\\greeting\\target\\debug\\greeting.exe"] }
```

也许你得到的结果比这个要长的多，这很正常，这个结果中 Args 结构体中有一个 inner 数组，只包含唯一的字符串，代表了当前运行的程序所在的位置。

但这个数据结构令人难以理解，没关系，我们可以简单地遍历它：

#### 实例

```
fn main() {
    let args = std::env::args();
    for arg in args {
        println!("{}: {}", arg);
    }
}
```

运行结果：

```
D:\\rust\\greeting\\target\\debug\\greeting.exe
```

一般参数们就是用来被遍历的，不是吗。

现在我们打开许久未碰的 launch.json，找到 "args": []，这里可以设置运行时的参数，我们将它写成 "args": ["first", "second"]，然后保存、再次运行刚才的程序，运行结果：

```
D:\\rust\\greeting\\target\\debug\\greeting.exe
first
second
```

作为一个真正的命令行程序，我们从未真正使用过它，作为语言教程不在此叙述如何用命令行运行 Rust 程序。但如果你是个训练有素的开发者，你应该可以找到可执行文件的位置，你可以尝试进入目录并使用命令行命令来测试程序接收命令行环境参数。

### 命令行输入

早期的章节详细讲述了如何使用命令行输出，这是由于语言学习的需要，没有输出是无法调试程序的。但从命令行获取输入的信息对于一个命令行程序来说依然是相当重要的。

在 Rust 中，std::io 模块提供了标准输入（可认为是命令行输入）的相关功能：

#### 实例

```
use std::io::stdin;

fn main() {
    let mut str_buf = String::new();

    stdin().read_line(&mut str_buf)
        .expect("Failed to read line.");

    println!("Your input line is \\n{}", str_buf);
}
```

令 VSCode 环境支持命令行输入是一个非常繁琐的事情，牵扯到跨平台的问题和不可调试的问题，所以我们直接在 VSCode 终端中运行程序。在命令行中运行：

```
D:\\rust\\greeting> cd ./target/debug
D:\\rust\\greeting\\target\\debug> ./greeting.exe
RUNOOB
Your input line is
RUNOOB
```

std::io::Stdio 包含 read\_line 读取方法，可以读取一行字符串到缓冲区，返回值都是 Result 枚举类，用于传递读取中出现的错误，所以常用 expect 或 unwrap 函数来处理错误。

注意：目前 Rust 标准库还没有提供直接从命令行读取数字或格式化数据的方法，我们可以读取一行字符串并使用字符串识别函数处理数据。

### 文件读取

我们在计算机的 D:\\ 目录下建立文件 text.txt，内容如下：

```
This is a text file.
```

这是一个将文本文件内容读入字符串的程序：

#### 实例

```
use std::fs;

fn main() {
    let text = fs::read_to_string("D:\\text.txt").unwrap();
    println!("{}: {}", text);
}
```

运行结果：

```
This is a text file.
```

在 Rust 中读取内存可容纳的一整个文件是一件极度简单的事情，std::fs 模块中的 read\_to\_string 方法可以轻松完成文本文件的读取。但如果要读取的文件是二进制文件，我们可以用 std::fs::read 函数读取 u8 类型集合：

#### 实例

```
use std::fs;

fn main() {
    let content = fs::read("D:\\text.txt").unwrap();
    println!("{}: {}", content);
}
```

运行结果：

```
[84, 104, 105, 115, 32, 105, 115, 32, 97, 32, 116, 101, 120, 116, 32, 102, 105, 108, 101, 46]
```

以上两种方式是一次性读取，十分适合 Web 应用的开发。但是对于一些底层程序来说，传统的按流读取的方式依然是无法被取代的，因为更多情况下文件的大小可能远超内存容量。

Rust 中的文件流读取方式：

#### 实例

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

fn main() {
    let mut buffer = [0u8; 5];
    let mut file = fs::File::open("D:\\text.txt").unwrap();
    file.read(&mut buffer).unwrap();
    println!("{}: {}", buffer);
    file.read(&mut buffer).unwrap();
    println!("{}: {}", buffer);
}
```

运行结果：

```
[84, 104, 105, 115, 32]
[105, 115, 32, 97, 32]
```

std::fs 模块中的 File 类是描述文件的类，可以用于打开文件，再打开文件之后，我们可以使用 File 的 read 方法按流读取文件的下面一些字节到缓冲区（缓冲区是一个 u8 数组），读取的字节数等于缓冲区的长度。

注意：VSCode 目前还不具备自动添加标准库引用的功能，所以有时出现“函数或方法不存在”的错误有可能是标准库引用的问题。

我们可以查看标准库的注释文档（鼠标放到上面会出现）来手动添加标准库。

std::fs::File 的 open 方法是“只读”打开文件，并且没有配套的 close 方法，因为 Rust 编译器可以在文件不再被使用时自动关闭文件。

### 文件写入

文件写入分为一次性写入和流式写入。流式写入需要打开文件，打开方式有“新建”（create）和“追加”（append）两种。

一次性写入：

#### 实例

```
use std::fs;

fn main() {
    fs::write("D:\\text.txt", "FROM RUST PROGRAM")
        .unwrap();
}
```

这和一次性读取一样简单方便。执行程序之后，D:\\text.txt 文件的内容将会被重写为 FROM RUST PROGRAM。所以，一次性写入请谨慎使用！因为它会直接删除文件内容（无论文件多大）。如果文件不存在就会创建文件。

如果想使用流的方式写入文件内容，可以使用 std::fs::File 的 create 方法：

#### 实例

```
use std::io::prelude::*;
use std::fs::OpenOptions;

fn main() {
    let mut file = OpenOptions::new()
        .append(true).open("D:\\text.txt")?;

    file.write(b"APPEND WORD")?;
}
```

运行之后，D:\\text.txt 文件内容将变成：

```
FROM RUST PROGRAM APPEND WORD
```

OpenOptions 是一个灵活的打开文件的方法，它可以设置打开权限，除 append 权限以外还有 read 权限和 write 权限，如果我们想以读写权限打开一个文件可以这样写：

#### 实例

```
use std::io::prelude::*;
use std::fs::OpenOptions;

fn main() {
    let mut file = OpenOptions::new()
        .read(true).write(true).open("D:\\text.txt")?;

    file.write(b"COVER")?;
}
```

运行之后，D:\\text.txt 文件内容将变成：

```
COVER RUST PROGRAM APPEND WORD
```

## Rust 集合与字符串

集合 (Collection) 是数据结构中最普遍的数据存放形式，Rust 标准库中提供了丰富的集合类型帮助开发者处理数据结构的操作。

### 向量

向量 (Vector) 是一个存放多值的单数据结构，该结构将相同类型的值线性的存放在内存中。

向量是线性表，在 Rust 中的表示是 `Vec<T>`。

向量的使用方式类似于列表 (List)，我们可以通过这种方式创建指定类型的向量：

```
let vector: Vec<i32> = Vec::new(); // 创建类型为 i32 的空向量
let vector = vec![1, 2, 4, 8]; // 通过数组创建向量
```

我们使用线性表常常会用到追加的操作，但是追加和栈的 push 操作本质是一样的，所以向量只有 push 方法来追加单个元素：

#### 实例

```
fn main() {
    let mut vector = vec![1, 2, 4, 8];
    vector.push(16);
    vector.push(32);
    vector.push(64);
    println!("{:?}", vector);
}
```

运行结果：

```
[1, 2, 4, 8, 16, 32, 64]
```

`append` 方法用于将一个向量拼接到另一个向量的尾部：

#### 实例

```
fn main() {
    let mut v1: Vec<i32> = vec![1, 2, 4, 8];
    let mut v2: Vec<i32> = vec![16, 32, 64];
    v1.append(&mut v2);
    println!("{:?}", v1);
}
```

运行结果：

```
[1, 2, 4, 8, 16, 32, 64]
```

`get` 方法用于取出向量中的值：

#### 实例

```
fn main() {
    let mut v = vec![1, 2, 4, 8];
    println!("{}: {:?}", 0, v.get(0));
    Some(value) => value.to_string(),
    None => "None".to_string()
});
```

运行结果：

```
1
```

因为向量的长度无法从逻辑上推断，`get` 方法无法保证一定取到值，所以 `get` 方法的返回值是 Option 枚举类，有可能为空。

这是一种安全的取值方法，但是书写起来有些麻烦。如果你能够保证取值的下标不会超出向量下标取值范围，你也可以使用数组取值语法：

#### 实例

```
fn main() {
    let v = vec![1, 2, 4, 8];
    println!("{}: {}", 0, v[0]);
}
```

运行结果：

```
100
32
57
```

如果遍历过程中需要更改变量的值：

#### 实例

```
fn main() {
    let v = vec![100, 32, 57];
    for i in &v {
        *i += 50;
    }
}
```

运行结果：

```
150
82
107
```

但如果我们将 `v[4]` 改为 `v[4]`，那么向量会返回错误。

遍历向量：

#### 实例

```
fn main() {
    let v = vec![100, 32, 57];
    for i in &v {
        println!("{}: {}", i);
    }
}
```

运行结果：

```
100
32
57
```

如果遍历过程中需要更改变量的值：

#### 实例

```
fn main() {
    let v = vec![100, 32, 57];
    for i in &mut v {
        *i += 50;
    }
}
```

运行结果：

```
150
82
107
```

注意：`nth` 函数是从迭代器中取出某值的方法，请不要在遍历中这样使用！因为 UTF-8 每个字符的长度不一定相等！

如果想截取字符串串：

#### 实例

```
fn main() {
    let s = String::from("Hello, 世界");
    let sub = &s[0..2];
    println!("{}: {}", 0, sub);
}
```

运行结果：

```
EN
```

但是请注意此用法有可能肢解一个 UTF-8 字符！那样会报错：

#### 实例

```
fn main() {
    let s = String::from("Hello, 世界");
    let sub = &s[0..2];
    println!("{}: {}", 0, sub);
}
```

运行结果：

```
thread 'main' panicked at 'byte index 3 is not a char boundary; it is inside '中' (bytes 2..5) of 'EN中文'',
src\libcore\str\mod.rs:2069:5
note: run with 'RUST_BACKTRACE=1' environment variable to display a backtrace.
```

映射表

映射表 (Map) 在其他语言中广泛存在。其中应用最普遍的就是键值对映射表 (Hash Map)。

新建散列映射表：

#### 实例

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

fn main() {
    let mut map = HashMap::new();

    map.insert("color", "red");
    map.insert("size", "10 m^2");

    println!("{}: {}", "color", map.get("color").unwrap());
}
```

注意：这里没有声明散列表的泛型，是因为 Rust 的自动判断类型机制。

运行结果：

```
red
```

插入方法和 `get` 方法是映射表最常用的两个方法。

映射表支持迭代器：

#### 实例

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

fn main() {
    let mut map = HashMap::new();

    map.insert("color", "red");
    map.insert("size", "10 m^2");

    for p in map.iter() {
        println!("{}: {}", p);
    }
}
```

运行结果：

```
("color", "red")
("size", "10 m^2")
```

迭代元素是表示键值对的元组。

Rust 的映射表是十分方便的数据结构，当使用 `insert` 方法添加新的键值对的时候，如果已经存在相同的键，会直接覆盖对应的值。

如果你想“安全地插入”，就是在确认当前不存在某个键时才执行的插入动作，可以这样：

```
map.entry("color").or_insert("red");
```

这句话的意思是如果没有键为 “color”的键值对就添加它并设置值为 “red”，否则将跳过。

在已经确定有某个键的情况下如果想直接修改对应的值，有更好的办法：

#### 实例

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

fn main() {
    let mut map = HashMap::new();
    map.insert(1, "a");

    if let Some(x) = map.get_mut(&1) {
        *x = "b";
    }
}
```

运行结果：

```
EN
```

但是请注意此用法有可能肢解一个 UTF-8 字符！那样会报错：

#### 实例

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

fn main() {
    let mut map = HashMap::new();
    map.insert(1, "a");

    if let Some(x) = map.get_mut(&1) {
        *x = "b";
    }
}
```

运行结果：

```
thread 'main' panicked at 'byte index 3 is not a char boundary; it is inside '中' (bytes 2..5) of 'EN中文'',
src\libcore\str\mod.rs:2069:5
note: run with 'RUST_BACKTRACE=1' environment variable to display a backtrace.
```

# Rust 面向对象

面向对象的编程语言通常实现了数据的封装与继承并能基于数据调用方法。

Rust 不是面向对象的编程语言，但这些功能都得以实现。

## 封装

封装就是对外显示的策略，在 Rust 中可以通过模块的机制来实现最外层的封装，并且每一个 Rust 文件都可以看作一个模块，模块内的元素可以通过 pub 关键字对外明示。这一点在“组织管理”章节详细叙述过。

“类”往往是面向对象的编程语言中常用到的概念。“类”封装的是数据，是对同一类数据实体以及其处理方法的抽象。在 Rust 中，我们可以使用结构体或枚举类来实现类的功能：

### 实例

```
pub struct ClassName {
    pub field: Type,
}

pub impl ClassName {
    fn some_method(&self) {
        // 方法函数体
    }
}

pub enum EnumName {
    A,
    B,
}

pub impl EnumName {
    fn some_method(&self) {
    }
}
```

下面建造一个完整的类：

### 实例

```
second.rs
pub struct ClassName {
    field: i32,
}

impl ClassName {
    pub fn new(value: i32) -> ClassName {
        ClassName {
            field: value
        }
    }

    pub fn public_method(&self) {
        println!("from public method");
        self.private_method();
    }

    fn private_method(&self) {
        println!("from private method");
    }
}

main.rs
mod second;
use second::ClassName;

fn main() {
    let object = ClassName::new(1024);
    object.public_method();
}
```

输出结果：

```
from public method
from private method
```

## 继承

几乎其他的面向对象的编程语言都可以实现“继承”，并用“extend”词语来描述这个动作。

继承是多态（Polymorphism）思想的实现，多态指的是编程语言可以处理多种类型数据的代码。在 Rust 中，通过特性（trait）实现多态。有关特性的细节已在“特性”章节给出。但是特性无法实现属性的继承，只能实现类似于“接口”的功能，所以想继承一个类的方法最好在“子类”中定义“父类”的实例。

总结地说，Rust 没有提供跟继承有关的语法糖，也没有官方的继承手段（完全等同于 Java 中的类的继承），但灵活的语法依然可以实现相关功能。

## Rust 并发编程

安全高效的处理并发是 Rust 诞生的目的之一，主要解决的是服务器高负载承受能力。

并发 (concurrent) 的概念是指程序不同的部分独立执行，这与并行 (parallel) 的概念容易混淆，并行强调的是“同时执行”。

并发往往会造成并行。

本章讲述与并发相关的编程概念和细节。

### 线程

线程 (thread) 是一个程序中独立运行的一个部分。

线程不同于进程 (process) 的地方是线程是程序以内的概念，程序往往是在一个进程中执行的。

在有操作系统的环境中进程往往被交替地调度得以执行，线程则在进程以内由程序进行调度。

由于线程并发很有可能出现并行的情况，所以在并行中可能遇到的死锁、延宕错误常出现于含有并发机制的程序。

为了解决这些问题，很多其它语言（如 Java、C#）采用特殊的运行时 (runtime) 软件来协调资源，但这样无疑极大地降低了程序的执行效率。

C/C++ 语言在操作系统的最底层也支持多线程，且语言本身以及其编译器不具备侦察和避免并行错误的能力，这对于开发者来说压力很大，开发者需要花费大量的精力避免发生错误。

Rust 不依靠运行时环境，这一点像 C/C++ 一样。

但 Rust 在语言本身就设计了包括所有权机制在内的手段来尽可能地把最常见的错误消灭在编译阶段，这一点其他语言不具备。

但这不意味着我们编程的时候可以不小心，迄今为止由于并发造成的问题还没有在公共范围内得到完全解决，仍有可能出现错误，并发编程时要尽量小心！

Rust 中通过 `std::thread::spawn` 函数创建新线程：

#### 实例

```
use std::thread;
use std::time::Duration;

fn spawn_function() {
    for i in 0..5 {
        println!("spawned thread print {}", i);
        thread::sleep(Duration::from_millis(1));
    }
}

fn main() {
    thread::spawn(spawn_function);

    for i in 0..3 {
        println!("main thread print {}", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

运行结果：

```
main thread print 0
spawned thread print 0
main thread print 1
spawned thread print 1
main thread print 2
spawned thread print 2
```

这个结果在某些情况下顺序有可能变化，但总体上是这样打印出来的。

此程序有一个子线程，目的是打印 5 行文字，主线程打印三行文字，但很显然随着主线程的结束，spawn 线程也随之结束了，并没有完成所有打印。

`std::thread::spawn` 函数的参数是一个无参函数，但上述写法不是推荐的写法，我们可以使用闭包 (closures) 来传递函数作为参数：

#### 实例

```
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 0..5 {
            println!("spawned thread print {}", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 0..3 {
        println!("main thread print {}", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

闭包是可以保存进变量或作为参数传递给其他函数的匿名函数。闭包相当于 Rust 中的 Lambda 表达式，格式如下：

```
|参数1, 参数2, ...| -> 返回值类型 {
    // 函数体
}
```

例如：

#### 实例

```
fn main() {
    let inc = |num: i32| -> i32 {
        num + 1
    };
    println!("inc(5) = {}", inc(5));
}
```

运行结果：

```
inc(5) = 6
```

闭包可以省略类型声明使用 Rust 自动类型判断机制：

#### 实例

```
fn main() {
    let inc = |num| {
        num + 1
    };
    println!("inc(5) = {}", inc(5));
}
```

结果没有变化。

### join 方法

#### 实例

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 0..5 {
            println!("spawned thread print {}", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 0..3 {
        println!("main thread print {}", i);
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap();
}
```

运行结果：

```
main thread print 0
spawned thread print 0
spawned thread print 1
main thread print 1
spawned thread print 2
main thread print 2
spawned thread print 3
spawned thread print 4
```

join 方法可以使子线程运行结束后再停止运行程序。

### move 强制所有权迁移

这是一个经常遇到的情况：

#### 实例

```
use std::thread;
use std::sync::mpsc;
```

```
fn main() {
    let s = "hello";

    let handle = thread::spawn(|| {
        println!("{}", s);
    });
}
```

std::sync::mpsc 包含了消息传递的方法：

#### 实例

```
use std::thread;
use std::sync::mpsc;
```

```
fn main() {
    let s = "hello";

    let handle = thread::spawn(move || {
        println!("{}", s);
    });
}
```

在子线程中尝试使用当前函数的资源，这一定是错误的！因为所有权机制禁止这种危险情况的产生，它将破坏所有权机制销毁资源的一致性。我们可以使用闭包的 move 关键字来处理：

#### 实例

```
use std::thread;
use std::sync::mpsc;
```

```
fn main() {
    let s = "hello";

    let handle = thread::spawn(move || {
        println!("{}", s);
    });
}
```

### 消息传递

Rust 中一个实现消息传递并发的主要工具是通道 (channel)，通道有两部分组成，一个发送者 (transmitter) 和一个接收者 (receiver)。

std::sync::mpsc 包含了消息传递的方法：

#### 实例

```
use std::thread;
use std::sync::mpsc;
```

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

    thread::spawn(move || {
        tx.send("hi").unwrap();
    });
}
```

运行结果：

```
Got: hi
```

子线程获得了主线程的发送者 tx，并调用了它的 send 方法发送了一个字符串，然后主线程就通过对应的接收者 rx 接收到了。