一、Rust 基础教程

1.1 Rust 语言简介

Rust 是一门系统级别的编程语言。

Rust 由 Graydon Hoare 开发并在被 Mozilla 实验室收购后发扬光大。

1.2 应用程序及所采用的语言

总所周知，Java 和 C# 语言 一般用于构建面向用户服务的软件。比如电子表格，文字处理器，Web 应用程序或移动应用程序等业务应用程序。

至于面向机器的那些系统级别应用程序，因为对性能和并发由极高的要求，系统、软件以及软件平台所采用的语言几乎都是清一色的 C 语言 和 C++ 语言，比如它们操作系统，游戏引擎，编译器等。这些编程语言需要很大程度的硬件交互。

系统级别和应用程序级别编程语言面临两个主要问题：

- 难以写出高安全的语言，尤其是 C/C++ 指针带来的悬空指针，缓冲区溢出和内存泄漏等问题
- 缺少语言级别的高并发特性。

1.3 为什么选择 Rust？

为什么选择 Rust？

估计我们能说出一大堆理由来，然而，对我来说，最大的理由就是 字节跳动公司的飞聊团队已经用上了 Rust 语言了。这意味着学好 Rust 语言就有机会找到高薪的工作。

此外，正如 Rust 语言自己说的那样，Rust 语言有三大愿景：

- 高安全
- 高性能
- 高并发

Rust 语言旨在以简单的方式开发高度可靠和快速的软件。

Rust 语言支持用现代语言特性来写一些系统级别乃至机器级别的程序。

1.3.1 高性能

高性能是所有语言的最高追求，Rust 也不例外。

为了追求极致的性能，Rust 抛弃了 C/C++ 之外的语言都有的 垃圾回收器（ Garbage Collector (GC)）。

也就是消除了垃圾回收机制带来的性能损耗。

1.3.2 编译时内存安全

Rust 虽然也有指针的概念，但这个概念被大大的弱化，因此它没有 C/C++ 那种悬空指针，缓冲区溢出和内存泄漏等等问题。

1.3.3 天生多线程安全运行程序

Rust 是为多线程高并发而设计的系统级别语言

Rust 的 **拥有者(ownership)** 概念和 **内存安全** 规则使得它天生支持高并发，而且是支持没有数据竞争的高并发。

1.3.4 Rust 语言支持 Web Assembly (WASM) 语言

Rust 的目标是成为高并发且高安全的系统级语言，但这并不代表它就不能开发 Web 应用。

Rust 支持通过把代码编译成 **Web Assembly (WASM)** 语言从而能够在浏览器端以实现快速，可靠的运行。

Web Assembly (WASM) 语言是被设计用来在浏览器端/嵌入式设别上运行的，用于执行 CPU 计算密集型的语言。

也就是说 **Web Assembly (WASM) 语言** 的目标是和 Javascript 一样能够在浏览器里运行，但因为是编译型，所以更高效。

二、Rust 开发环境配置

Rust 语言的环境配置还是相对简单的，因为官方提供了 rustup 这个一步配置工具。rustup 是 Rust 官方推出的基于 终端/控制台/shell 的工具，可用于管理 Rust 版本和相关工具

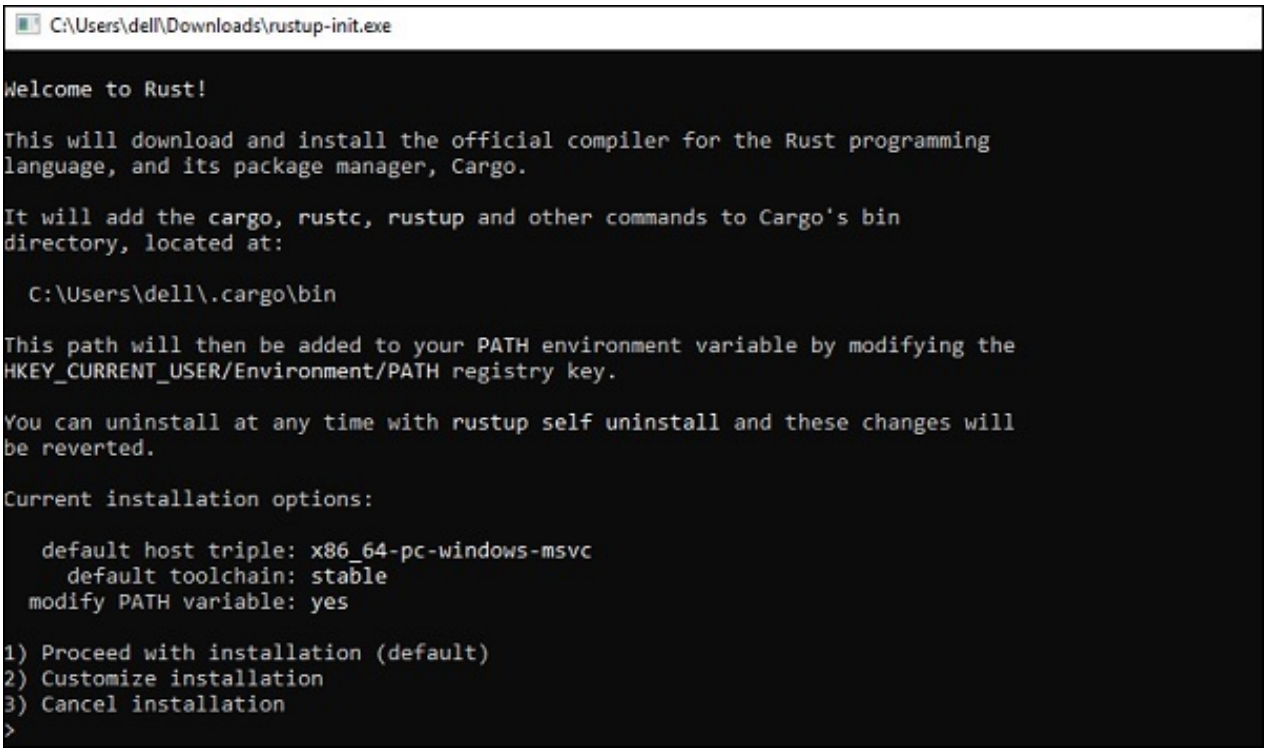
2.1 Windows 上安装 Rust

Windows 上安装任何语言的开发环境都有一点复杂，Rust 也无法避免这一点：

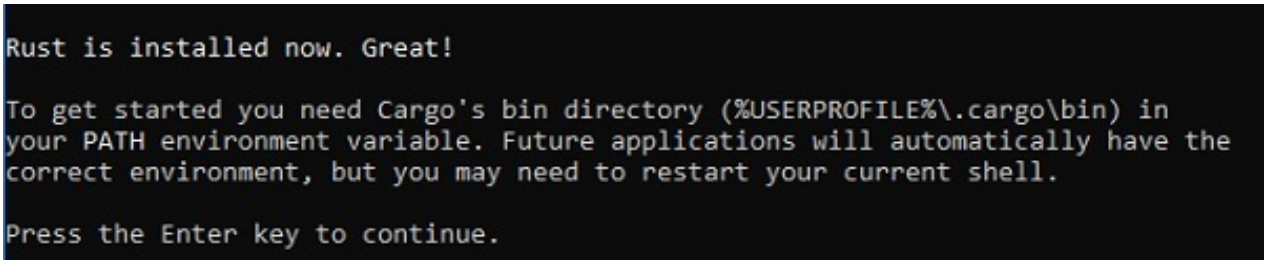
1. Windows 上运行 Rust 编译器需要 C++ 开发环境。

我们推荐的做法是安装 Visual Studio 2013 或更高的版本。你可以点击 VS 2013 Express 链接下载 Visual Studio 2013 并安装，详细的安装流程请参考 Visual Studio 2013。

2. 点击 Rust 安装工具 下载 Windows 版本的 rustup-init.exe 工具。
3. 双击下载好的 rustup-init.exe 文件，然后你就会看到如下的界面。



4. 从上面的界面中我们可以看到三个安装选项，一般情况下我们选择默认，也就是1。 选择默认什么都不要做，直接按下 **回车键** 就好。按下回车键后，Rust 就会开始安装，一般情况下，只要静静地等待安装完成即可。
5. 安装完成后会出现如下界面。



6. 从安装成功后的提示来看，Rust 被安装到目录 C:\Users{PC}.cargo\bin 里打开你自己的 C:\Users{PC}.cargo\bin 目录，可以看到下面的文件列表

```
cargo-fmt.exe
cargo.exe
rls.exe
rust-gdb.exe
rust-lldb.exe
rustc.exe // this is the compiler for rust
rustdoc.exe
rustfmt.exe
rustup.exe
```

我们简单的对上面的文件做个介绍

- Cargo 是 Rust 的包管理器。类似于 Nodejs 中的 npm 或 Python 中的 pip 或者 PHP 中的 composer。 我们可以通过运行下面的命令来检查 Cargo 是否安装正确和已经安装的版本。

```
C:\Users\Admin>cargo -V
cargo 1.29.0 (524a578d7 2018-08-05)
```

- rustc 是 Rust 的编译器，类似于 Java 中的 Javac 或 C/C++ 中的 gcc。 我们可以通过下面的命令来检查 rustc 是否安装正确和已经安装的版本。

```
C:\Users\Admin>cargo -V
cargo 1.29.0 (524a578d7 2018-08-05)
```

2.2 Linux 或 Mac 上安装

Linux 或 Mac 上安装 Rust 和 rustup 真的是再简单不过了，只要打开 终端 / Shell 输入下面的命令然后按下回车键

```
curl https://sh.rustup.rs -sSf | sh
```

上面那段脚本命令`curl https://sh.rustup.rs -sSf | sh`会下载必须的脚本并开始安装 rustup 工具，同时会安装当前最新的稳定版的的 Rust。

因为安装可能需要管理权限，因此可能会询问你的登陆密码。

安装完成后 终端 / shell 中会出现一下文字

```
Rust is installed now. Great!
```

上面这个安装命令会自动将 rustc 和 cargo 等命令添加到 **PATH** 路径中，我们只要重启终端/shell 就能立即使用。

如果你不想重启终端，那么你只需要运行下面的 shell 命令重新加载 **环境变量** 即可

```
source $HOME/.cargo/env
```

或者，你可以运行下面的命令手动将 rust 的安装目录临时添加到 **PATH** 中

```
export PATH="$HOME/.cargo/bin:$PATH"
```

> 注意: 因为安装过程中可能出现各种错误。如果你尝试编译 Rust 程序却输出链接器无法执行的错误时，意味着你的系统上未安装链接器，这个需要我们手动安装。

三、Rust HelloWorld

本章节我们通过一个小小的 Hello World 程序来看看和解释 Rust 最小程序的构成。

首先创建一个 HelloWorld 的目录并在 命令行/终端 中使用 cd 命令并进入到 HelloWorld 目录

```
C:\Users\Admin>mkdir HelloWorld
C:\Users\Admin>cd HelloWorld
C:\Users\Admin\HelloWorld>
```

使用你趁手的编辑器创建一个文件 Hello.rs，比如我使用 Visual Studo Code 作为代码编辑器

```
C:\Users\Admin\HelloWorld-App>code Hello.rs
```

Rust 语言的源代码文件以 .rs 作为文件扩展名。

上面的命令会在当前目录下创建一个 Hello.rs 的空文件，并在 Visual Studo Code 中打开。

在 Hello.rs 文件中输入以下代码，然后保存。

```
fn main(){
   println!("Hello World");
   println!("你好，零基础教程");
}
```

上面这段代码定义了一个名为 main 的函数。Rust 语言使用 fn 关键字定义函数。

main() 函数是一个预定义函数，充当 Rust 程序的入口点，类似于 C 语言 中的 main() 函数。

接下来的 println!() 是 Rust 语言中的一个 **预定义的宏**。这个 println!() 用于将传递给它的参数输出到 标准输出。

Rust 语言中的 宏 都会以 **感叹号(！)** 结尾。也就是说，你以后看到以 ! 结尾的类似函数调用，都是 **宏调用**。

使用 rustc 编译工具编译我们刚刚的 Hello.rs

```
C:\Users\Admin\HelloWorld>rustc Hello.rs
```

如果编译成功，rustc 命令会生成一个跟 **源代码文件** 同名的 .exe 二进制可执行文件。比如上面的命令会生成一个 Hello.exe 文件。

```
// 列出当前目录下的文件

C:\Users\Admin\HelloWorld>dir
Hello.exe
Hello.pdb
Hello.rs
```

为了检查编译是否成功，你可以双击 Hello.exe 查看运行结果。

3.1 什么是宏？

Rust 提供了一个功能非常强大的 **宏** 体系，通过这些 宏，我们可以很方便的进行**元编程**。

Rust 中的 **宏** 有点类似于 **函数**。它们跟函数还是有很大的区别的：

1. 宏以感叹号 (!) 结尾，正如上面我们所见到的那样。
2. 宏在编译时并不会生成一个 **函数调用**，而是直接对 宏 的源代码进行展开。这个和 C 语言 中宏是一样的。

从某些方面说，我们可以将 **宏** 理解为 **函数的加强版**

3.1.1 范例

下面的三行代码，演示了宏的使用

```
println!(); // 输出一个空行
println!("hello "); // 新的一行输出 hello
println!("format {} arguments", "some"); // 新的一行输出 format some arguments
```

3.2 Rust 语言中的注释

注释有助于改善程序代码的可读性。注释类似于我们读书时做的评注。

程序中的注释一般用于标注程序的一些基本信息，对一些难以理解的代码、类、函数或返回值进行说明等。

Rust 编译器会在编译程序的时候 主动忽略 代码中的注释，也就是说，注释不会影响代码的编译和执行。

Rust 语言支持以下两种注释，也是很多语言都有的默认注释方法

3.2.1 单行注释 / 行内注释

单行注释以 // 开头，之后说有的文字包括代码都属于注释。

也就是说，以 // 开头，直到行尾的所有文字都属于注释，包括中间的任何 //

例如:

```
// 这个单行注释单独一行

1+1; // 这是注释，但前面的 1+1; 是正常的代码

// 1+1; // 这个整行也是注释，// 可以任意多个
```

3.2.2 多行注释 / 块注释

多行注释 就是可以跨越多行，当然也可以只有一行。

多行注释以 /* 开头，以 */ 结尾。

块注释中可以有任意数量的 /*，但是，一旦遇到 */ 则表示 块注释结束

例如:

```
/* 这是块注释 */

/* 这
     是
       多行
          注释 */

/* /* /*
    这也是多行注释
*/
```

四、Rust 数据类型

类型系统 对于任何一门语言都是重中之重，因为它体现了语言所支持的不同类型的值。

类型系统 也是 IT 初学者最难啃的三座大山之一，而类型系统之所以难以理解，主要是没有合适的现成的参考体系。

举个例子，比如我们说 **类型系统** 存在的目的，就是 程序在存储或操作某个数之前检查这个数的有效性。

就这一句话，简简单单，看起来每个文字都懂，如果把它们放在一起，就有那么一点天书的味道了。

类型系统在程序存储或操作之前检查所提供值的有效性。这可以保证程序运行时给变量提供的数据不会发生类型错误。

更近一步说，类型系统可以允许编辑器时时报告错误或者自动提示。

Rust 是一个静态的严格数据类型的语言。每个值都有唯一的数据类型，要么是整型，要么是浮点型等等。

Rust 语言在赋值时并不强制要求指定变量的数据类型，Rust 编译器可以根据分配给它的值自动推断变量的数据类型。

4.1 声明/定义一个变量

Rust 语言使用 **let** 关键字来声明和定义一个变量。

Rust 语言中声明变量的语法格式如下

```
let variable_name = value;
```

这里我们只会粗略带过变量的定义和声明，后面的章节我们会详细介绍。

下面的代码演示了如何定义变量

```
fn main() {
   let company_string = "TutorialsPoint";  // string 字符串类型
   let rating_float = 4.5;                 // float 类型
   let is_growing_boolean = true;          // boolean 类型
   let icon_char = '♥';                    //unicode character 类型

   println!("company name is:{}",company_string);
   println!("company rating on 5 is:{}",rating_float);
   println!("company is growing :{}",is_growing_boolean);
   println!("company icon is:{}",icon_char);
}
```

上面的代码中，我们并没有为每一个变量指定它们的数据类型。Rust 编译器会自动从 **等号=** 右边的值中推断出该变量的类型。例如 Rust 会自动将 *双引号* 阔起来的数据推断为 **字符串**，把 *没有小数点的数字* 自动推断为 **整型**。把 *true* 或 *false* 值推断为 **布尔类型**。

println!() 是一个宏，而不是一个函数，区分函数和宏的唯一办法，就是看函数名/宏名最后有没有感叹号 !. 如果有感叹号则是宏，没有则是函数。

println!() 宏接受两个参数：

- 第一个参数是格式化符，一般是 {}，如果是复杂类型，则是 {:?}。
- 第二个参数是变量名或者常量名。

编译运行以上 Rust 代码，输出结果如下

```
company name is: TutorialsPoint
company rating on 5 is:4.5
company is growing: true
company icon is: ♥
```

4.2 标量数据类型

标量数据类型 又称为 基本数据类型。标量数据类型只能存储单个值，例如 10 或 3.14 或 c。

Rust 语言中有四种标量数据类型：

- 整型
- 浮点型
- 布尔类型
- 字符类型

接下来我们会对每种标量数据类型做一个简单的介绍。

4.3 整型

整数就是没有小数点的数字，比如说 0，1，-1，-2，9999999 等，但是 0.0 、1.0 和 -1.11 等都不是整数。

整型 能够囊括所有的数字，虽然不可能无穷大，但已经大到足够使用了。

最大的整型为 340282366920938463463374607431768211455，由 std::u128:MAX 定义。

最小的整型为 -170141183460469231731687303715884105728，由 std::i128:MIN 定义。

它们看起来是不是一个天文数字?

整型可以进一步分为 **有符号整型** 和 **无符号整型** 两种:

- 有符号整型，英文 signed，既可以存储正数，也可以存储负数。
- 无符号整型，因为 unsigned，只能存储正数。

按照存储空间来说，整型可以进一步划分为 1字节、2字节、4字节、8字节、16字节。

1 字节 = 8 位，每一位能只能存储二进制 0 或 1，因此每一个字节能够存储的最大数字是 256，而最小数字则是 -127。

有点复杂了，详细的看 C 语言的数据可能会更简单。

下表列出了整型所有的细分类型：

大小	有符号	无符号
8bit	i8	u8
16bit	i16	u16
32bit	i32	u32
64bit	i64	u64
128bit	i128	u128
Arch	isize	usize

i32 是默认的整型，如果我们直接说出一个数字而不说它的数据类型，那么它默认就是 i32 。

整型的长度还可以是 arch。arch 是由 CPU 构架决定的大小的整型类型。大小为 arch 的整数在 x86 机器上为 32 位，在 x64 机器上为 64 位。

Arch 整型通常用于表示容器的大小或者数组的大小，或者数据在内存上存储的位置。

4.3.1 范例：如何定义各种整型的变量

定义整型变量的时候要注意每种整型的最大值和最小值，如果超出可能会赋值失败，也有可能结果不是我们想要的。

```
fn main() {
   let result = 10;    // i32 默认
   let age:u32 = 20;
   let sum:i32 = 5-15;
   let mark:isize = 10;
   let count:usize = 30;
   println!("result value is {}",result);
   println!("sum is {} and age is {}",sum,age);
   println!("mark is {} and count is {}",mark,count);
}
```

编译运行以上 Rust 代码，输出结果如下

```
result value is 10
sum is -10 and age is 20
mark is 10 and count is 30
```

整型只能保存整数，不能保存有小数点的数字，哪怕小数点后面全是 0。如果把一个有小数的数字赋值给整型，会报编译错误

```
fn main() {
   let age:u32 = 20.0;
}
```

编译运行以上 Rust 代码，报错信息如下

```
error[E0308]: mismatched types
 --> src/main.rs:2:18
  |
2 |    let age:u32 = 20.0;
  |                  ^^^^ expected u32, found floating-point number
  |
  = note: expected type `u32`
             found type `{float}`
```

从报错信息中可以看出，Rust 语言将 20.0 这种有小数点的数字称为 浮点数。使用 float 来表示。

我们不能将一个 float 类型的数字赋值给 u32 类型。

4.3.2 整型范围

每种整型并不都是能存储任意数字的，每种整型只能装下固定大小的数字，但总体上，大的整型能装下小的整型。

每种 **有符号整型** 能够存储的最小值为 **-(2^(n-1)**，能够存储的最大值为 **2^(n-1) -1**。

每种 **无符号整型** 能够存储的最小值为 **0**，能够存储的最大值为 **2^n - 1**。

其中 n 是指数据类型的大小。

例如一个 8 位有符号整型 i8，它能够存储的最小值为 -(2^(8-1)) = -128。最大值为 (2^(8-1)-1) = 127。

例如一个 8 位无符号整型 u8，它能够存储的最小值为 0，能够存储的最大值为 2^8-1 = 255。

4.3.3 整型溢出

每种整型并不都是能存储任意数字的，每种整型只能装下固定大小的数字。如果给予的数字超出了整型的范围则会发生溢出。

比如一个 i8 能够存储的最小值是 0，如果我们让它来存储 -1 则会发生溢出。

当发生数据溢出时，Rust 抛出一个错误指示数据溢出。

例如下面的代码，编译会报错。

```
fn main() {
   let age:u8 = 255;

   // u8 只能存储 0 to 255
   let weight:u8 = 256;   // 溢出值为 0
   let height:u8 = 257;   // 溢出值为 1
   let score:u8 = 258;    // 溢出值为 2

   println!("age is {} ",age);
   println!("weight is {}",weight);
   println!("height is {}",height);
   println!("score is {}",score);
}
```

编译运行以上代码，报错信息如下

```
error: literal out of range for u8
 --> src/main.rs:5:20
  |
5 |    let weight:u8 = 256;   // 溢出值为 0
  |                     ^^^
  |
  = note: #[deny(overflowing_literals)] on by default

error: literal out of range for u8
 --> src/main.rs:6:20
  |
6 |    let height:u8 = 257;   // 溢出值为 1
  |                     ^^^

error: literal out of range for u8
 --> src/main.rs:7:19
  |
7 |    let score:u8 = 258;    // 溢出值为 2
  |                   ^^^
```

从错误信息来看，三个溢出的地方都报错了。提示赋值的数字超出了 u8 的范围。

4.4 浮点型：f32 和 f64

前面我们提到过，整型只能保存没有小数点的数字。而对于有小数点的数字，Rust 提供了浮点型。

Rust 区分整型和浮点型的唯一指标就是 **有没有小数点**。

Rust 中的整型和浮点型是严格区分的，不能相互转换。

也就是说，我们不能将 0.0 赋值给任意一个整型，也不能将 0 赋值给任意一个浮点型。

按照存储大小，我们把浮点型划分为 **f32** 和 **f64**。其中 f64 是默认的浮点类型。

- f32 又称为 单精度浮点型。
- f64 又称为 双精度浮点型，它是 Rust 默认的浮点类型.

4.3.1 范例：如何定义各种浮点型的变量

定义浮点型变量的时候要注意每种浮点型的最大值和最小值，如果超出可能会赋值失败，也有可能结果不是我们想要的。

```
fn main() {
   let result = 10.00;         // 默认是 f64
   let interest:f32 = 8.35;
   let cost:f64 = 15000.600;  // 双精度浮点型

   println!("result value is {}",result);
   println!("interest is {}",interest);
   println!("cost is {}",cost);
}
```

编译运行以上 Rust 代码，报错信息如下

```
interest is 8.35
cost is 15000.6
```

4.4 不允许类型自动转换

Rust 中的数字类型与 C/C++ 中不同的是 Rust 语言不允许类型自动转换。

例如，把一个 *整型* 赋值给一个 *浮点型* 是会报错的。

4.4.1 范例

```
fn main() {
   let interest:f32 = 8;   // integer assigned to float variable
   println!("interest is {}",interest);
}
```

编译上面的代码，会抛出 mismatched types error 错误

```
error[E0308]: mismatched types
   --> main.rs:2:22
   |
 2 | let interest:f32=8;
   |    ^ expected f32, found integral variable
   |
   = note: expected type `f32`
      found type `{integer}`
error: aborting due to previous error(s)
```

4.5 数字可读性分隔符 _

为了方便阅读超大的数字，Rust 语言允许使用一个 *虚拟的分隔符* 也就是 *下划线（_）* 来对数字进行可读性分隔符。

比如为了提高 50000 的可读性，我们可以写成 50_000 。

Rust 语言会在编译时移除数字可读性分隔符 _

4.5.1 范例

我们写几个例子来演示下 数字分隔符，从结果中可以看出，分隔符对数字没有造成任何影响。

```
fn main() {
   let float_with_separator = 11_000.555_001;
   println!("float value {}",float_with_separator);

   let int_with_separator = 50_000;
   println!("int value {}",int_with_separator);
}
```

编译运行上面的代码，输出结果如下

```
float value 11000.555001
int value 50000
```

4.6 布尔类型 bool

布尔类型 只有两个可能的取值 true 或 false 。

Rust 使用 bool 关键字来声明一个 **布尔类型** 的变量。

4.6.1 范例

```
fn main() {
   let isfun:bool = true;
   println!("Is Rust Programming Fun ? {}",isfun);
}
```

编译运行上面的代码，输出结果如下

```
Is Rust Programming Fun ? true
```

4.7 字符类型 char

字符，简单的来说，就是字符串的基本组成部分，也就是单个字符或字。

Rust 使用 char 作为 字符数据类型。这点可谓是继承了 C / C++。

但与 C / C++ 不同的是：Rust 使用 UTF-8 作为底层的编码 ，而不是常见的使用 ASCII 作为底层编码。

也就是说，Rust 中的 **字符数据类型** 包含了数字、字母、Unicode 和 其它特殊字符。

Rust 选用 UTF-8 作为底层编码可谓是顺应时代的潮流。因为编程和互联网早就不极限于拉丁语系的国家，像中国、印度、日本等国家都有大量的程序员和网民。

Unicode 编码的标量值的范围从 U+0000 到 U+D7FF， U+E000 到 U+10FFFF（含）

4.7.1 范例

我们输出几个不同语系的字符来演示下 Rust 中的 char 字符类型。

```
fn main() {
   let special_character = '@'; //default
   let alphabet:char = 'A';
   let emoji:char = 'c'; // 笑脸的那个图

   println!("special character is {}",special_character);
   println!("alphabet is {}",alphabet);
   println!("emoji is {}",emoji);
}
```

编译运行上面的代码，输出结果如下

```
special character is @
alphabet is A
emoji is c
```

五、Rust 变量定义

现实生活中，我们肯定不会使用 隔壁邻居的儿子、隔壁的隔壁邻居的女儿 、隔壁的隔壁的隔壁邻居的儿子这种称呼，多难记啊。

我们都习惯于给别人取一个小名，比如 小明 = 隔壁邻居的儿子、小红 = 隔壁的隔壁邻居的女儿、小王 = 隔壁的隔壁的隔壁邻居的儿子。

在计算机中，在编程时也是一样的。

我们肯定不会使用 内存位置1、内存位置2、内存位置3 这种形式来记录内存中存储的数据。

我们喜欢对内存进行标记，比如 name = 内存位置1、age = 内存位置2、address = 内存位置3

我们把 name、age 和 address 这种标记称之为 **变量**

变量 是对 **内存** 的标记。

因为内存中存储的数据是有数据类型的，所以变量也是有数据类型的。

变量的数据类型不仅用于标识内存中存储的数据类型，还用于标识内存中存储的数据的大小。同时也标识内存中存储的数据可以进行的操作。

5.1 Rust 语言中变量的命名规则

Rust 中的变量名并不是随便什么字符都可以的，它遵循着一套规则

1. 变量名中可以包含 字母、数字 和 下划线。也就只能是 abcdefghijklmnopqrstuvwxyz013456789_ 以及大写字母。
2. 变量名必须以 字母 或 下划线 开头。
3. 也就是不能以 数字 开头。
4. 变量名是 区分大小 写的。也就是大写的 A 和小写的 a 是两个不同的字符

5.2 Rust 中变量的声明语法

Rust 语言中声明变量时的 数据类型 是可选的，也就是可以忽略的。如果忽略了，那么 Rust 就会自动通过 **值** 来推断变量的类型。

就像我们平时说 1 ，大家默认都会把它当作数字看待那样。

Rust 语言中声明变量的语法格式如下

```
let variable_name = value;            // 不指定变量类型
let variable_name:dataType = value;   // 指定变量类型
```

5.2.1 范例

```
fn main() {
   let fees = 25_000;
   let salary:f64 = 35_000.00;
   println!("fees is {} and salary is {}",fees,salary);
}
```

编译运行以上 Rust 代码，输出结果如下

```
fees is 25000 and salary is 35000
```

5.3 let 变量的不可变性

默认情况下，Rust 语言中的变量是不可变的。

Rust 语言中使用 let 声明的变量，在第一次赋值之后，是不可变更不可重新赋值的，变成了 **只读** 状态。

就像我们身份证上的名字，不是轻易能够改变的，不是自己涂涂写写就能改变的。

这点不同于很多语言，倒是跟 Erlang 有点像。

如果你觉得有点难以理解，那么上代码来的更直接

```
fn main() {
   let fees = 25_000;
   println!("fees is {} ",fees);
   fees = 35_000;
   println!("fees changed is {}",fees);
}
```

编译上面这段 Rust 代码，是会报错的

```
error[E0384]: re-assignment of immutable variable `fees`
 --> main.rs:6:3
   |
 3 | let fees = 25_000;
   | ---- first assignment to `fees`
...
 6 | fees=35_000;
   | ^^^^^^^^^^ re-assignment of immutable variable

error: aborting due to previous error(s)
```

上面这段错误消息告诉我们：我们不能对变量 fees 进行第二次赋值。

这估计是 Rust 语言中最大的变更了。也是并发编程安全性的重要组成部分。

但也是 Rust 程序员最容易出错的地方。

5.4 可变变量

let 关键字声明的变量默认情况下是 **不可变更** 的，在定义了一个变量之后，默认情况下我们是不能通过赋值再改变它的值的。

```
fn main() {
   let fees:i32 = 25_000;
   println!("fees is {} ",fees);
   fees = 35_000;
   println!("fees changed is {}",fees);
}
```

上面这段代码，编译会报错的。

```
error[E0384]: re-assignment of immutable variable `fees`
 --> main.rs:6:3
   |
 3 | let fees = 25_000;
   | ---- first assignment to `fees`
...
 6 | fees=35_000;
   | ^^^^^^^^^^^ re-assignment of immutable variable

error: aborting due to previous error(s)
```

但有时候，我们又需要给一个变量重新赋值，这要怎么做呢？

Rust 语言提供了 **mut** 关键字表示 **可变更**。 我们可以在变量名的前面加上 mut 关键字来告诉编译器这个变量是可以重新赋值的。

5.4.1 可更变量的声明语法

```
let mut variable_name:dataType = value;
```

或者让编译器自动推断类型

```
let mut variable_name = value;
```

5.4.2 范例

我们使用 mut 关键字定义一个变量 fees 然后再重新赋值

```
fn main() {
   let mut fees:i32 = 25_000;
   println!("fees is {} ",fees);
   fees = 35_000;
   println!("fees changed is {}",fees);
}
```

编译运行以上 Rust 代码，输出结果如下

```
fees is 25000
fees changed is 35000
```

六、Rust 常量

常量就是那些值不能被改变的变量。一旦我们定义了一个常量，那么就再也没有任何方法可以改变常量的值了。

Rust 语言中使用 const 关键字来定义一个常量。定义常量时需要明确指定常量的数据类型。

6.1 定义常量的语法

Rust 中定义常量的语法格式如下

```
const VARIABLE_NAME:dataType = value;
```

从语法上来看，定义常量和定义变量的语法是类似的：

- 定义常量的关键字是 const ，而定义变量的关键字是 let。
- 定义常量时必须指定数据类型，而定义变量时数据类型可以省略，因为会自动推导。
- 常量名的命名规则可变量的命名规则一样，但常量名一般都是 大写字母。

6.2 Rust 常量命名规则

有点重复了，不过我们还是得重复说一下 常量的命名规则，常量命名规则和变量的命名规则是类似的，除了以下几点：

- 常量名通常都是 大写字母。
- 使用 const 关键字而不是 let 关键字来定义一个常量。
- 根据这些命名规则，我们来看一些定义常量的范例

```
fn main() {
   const USER_LIMIT:i32 = 100;    // 定义了一个 i32 类型的常量
   const PI:f32 = 3.14;           // 定义了一个 float 类型的常量

   println!("user limit is {}",USER_LIMIT);  // 显示常量的值
   println!("pi value is {}",PI);            // 显示常量的值
}
```

6.3 Rust 中常量与变量的不同之处

Rust 中常量与变量的不同之处是面试必问的经典题目之一。

下面我们就来详细讲讲这个面试题的答案吧。

1. 声明常量使用的是 const 关键字，声明变量使用的是 let 关键字。
2. 声明变量时 数据类型可以忽略，但声明常量时数据类型不可以忽略。这就意味着 const USER_LIMIT=100 这种常量声明会导致编译错误。
3. 虽然声明变量时使用 let 关键字也会导致 变量不可以重新赋值，但我们可以加上 mut 关键字来让变量可以被重新赋值。然而常量却没有这种机制，常量一旦定义就永远不可变更和重新赋值。
4. 常量只能 被赋值为 常量表达式/数学表达式，不能是 函数返回值 或者其它只有在运行时才能确定的值。这是因为 常量 只是一个符号，会在 编译时 替换为具体的值，这个有点类似于 C 语言中的 #define 定义的符号。
5. 常量可以在任意作用域里定义，包括全局作用域。也就是可以在任何地方定义。因此我们可以在需要的地方定义，以明确我们为什么需要定义一个常量。
6. 不能出现同名常量，变量可以（隐藏/屏蔽）

6.4 常量和变量的隐藏/屏蔽

Rust 语言中允许重复定义一个相同变量名的变量。这种重名的变量的规则是 后面定义的变量会重写/屏蔽 前面定义的同名变量。

我们使用一个范例来演示下

```
fn main() {
   let salary = 100.00;
   let salary = 1.50 ;
   // 输出薪水
   println!("salary 变量的值为:{}",salary);
}
```

编译运行以上 Rust 代码，输出结果如下

```
salary 变量的值为:1.5
```

上面的代码，我们定义了两个同名的变量 salary，第一次 salary 我们赋值 100.00 ，第二次 salary 我们赋值为 1.5。

从输出结果中可以看出，第二个 salary 会隐藏/屏蔽第一次定义的变量。

6.5 同名变量可以有不同的数据类型

Rust 支持在同一个作用域/内层作用域内定义多个同名变量，而且每个同名变量的类型还可以不一样。

我们使用一个简单的范例来演示下 同名变量可以有不同的数据类型

```
fn main() {
   let uname = "Mohtashim";
   let uname = uname.len();
   println!("uname 字符串的字符数是: {}",uname);
}
```

编译运行以上 Rust 代码，输出结果如下

```
uname 字符串的字符数是:: 9
```

在上面这个范例中，我们定义了两个同名变量 uname，但它们有着不同的数据类型。第一次 uname 的数据类型是 string 也就是字符串，而第二次则变成了 i32 是一个整数了。

> len() 函数返回字符串中字符的个数。

6.6 不能出现同名常量

常量与变量的另一个不同点是： 常量不能被屏蔽/遮挡，也不能被重复定义。

也就是说不存在内层/后面作用域定义的常量屏蔽外层/前面定义的同名常量。

这是因为 Rust 语言中不允许有同名的常量。

6.6.1 范例

我们将上面的范例改造下，使用 const 关键字来定义同名的常量

```
fn main() {
   const NAME:&str = "Mohtashim";
   const NAME:usize = NAME.len();
   //Error : `NAME` already defined
   println!("改变 name 常量的类型: {}",NAME);
}
```

七、Rust 字符串

Rust 语言提供了两种字符串

- 字符串字面量 **&str**。它是 Rust 核心内置的数据类型。
- 字符串对象 **String**。它不是 Rust 核心的一部分，只是 Rust 标准库中的一个 **公开 pub 结构体**。

7.1 字符串字面量 &str

字符串字面量 &str 就是在 编译时 就知道其值的字符串类型，是 Rust 语言核心的一部分。 字符串字面量 &str 是字符的集合，被硬编码赋值给一个变量。

```
let name1 = "你好，零基础教程 简单编程";
```

字符串字面量的核心代码可以在模块 *std::str* 中找到，如果你有兴趣，可以阅读一二。

Rust 中的字符串字面量被称之为 **字符串切片**。因为它的底层实现是 **切片**。

7.1.1 范例

下面的代码，我们定义了两个字符串字面量 company 和 location

```
fn main() {
   let company:&str="零基础教程";
   let location:&str = "中国";
   println!("公司名 : {} 位于 :{}",company,location);
}
```

字符串字面量模式是 **静态** 的。 这就意味着字符串字面量从创建时开始会一直保存到程序结束。

因为默认是 **静态** 的，我们也可以主动添加 static 关键字。只不过语法格式有点怪，所以日常使用还是忽略吧。

```
fn main() {
   let company:&'static str = "零基础教程";
   let location:&'static str = "中国";
   println!("公司名 : {} 位于 :{}",company,location);
}
```

编译运行以上 Rust 代码，输出结果如下

```
公司名 : 零基础教程 位于 :中国
```

7.2 字符串对象

字符串对象是 Rust 标准库提供的内建类型。

与字符串字面量不同的是：字符串对象并不是 Rust 核心内置的数据类型，它只是标准库中的一个 公开 pub 的结构体。

字符串对象在标准库中的定义语法如下

```
pub struct String
```

字符串对象是是一个 长度可变的集合，它是 **可变** 的而且使用 UTF-8 作为底层数据编码格式。

字符串对象在 **堆 heap** 中分配，可以在运行时提供字符串值以及相应的操作方法。

7.2.1 创建字符串对象的语法

要创建一个字符串对象，有两种方法：

1. 一种是创建一个新的空字符串，使用 String::new() 静态方法

```
String::new()
```

2. 另一种是根据指定的字符串字面量来创建字符串对象，使用 String::from() 方法

```
String::from()
```

7.2.2 范例

下面，我们分别使用 String::new() 方法和 String::from() 方法创建字符串对象，并输出字符串对象的长度

```
fn main(){
   let empty_string = String::new();
   println!("长度是 {}",empty_string.len());

   let content_string = String::from("零基础教程");
   println!("长度是 {}",content_string.len());
}
```

编译运行以上 Rust 代码，输出结果如下

```
长度是 0
长度是 12
```

The above example creates two strings − an empty string object using the new method and a string object from string literal using the from method.

7.3 字符串对象常用的方法

方法	原型	说明
new()	pub const fn new() -> String	创建一个新的字符串对象
to_string()	fn to_string(&self) -> String	将字符串字面量转换为字符串对象
replace() pub fn replace<'a, P>(&'a self, from: P, to: &str) -> String	搜索指定模式并替换	
as_str()	pub fn as_str(&self) -> &str	将字符串对象转换为字符串字面量
push()	pub fn push(&mut self, ch: char)	再字符串末尾追加字符
push_str()	pub fn push_str(&mut self, string: &str)	再字符串末尾追加字符串
len()	pub fn len(&self) -> usize	返回字符串的字节长度
trim()	pub fn trim(&self) -> &str	去除字符串首尾的空白符
split_whitespace()	pub fn split_whitespace(&self) -> SplitWhitespace	根据空白符分割字符串并返回分割后的迭代器
split()	pub fn split<'a, P>(&'a self, pat: P) -> Split<'a, P>	根据指定模式分割字符串并返回分割后的迭代器。模式 P 可以是字符串字面量或字符或一个返回分割符的闭包
chars()	pub fn chars(&self) -> Chars	返回字符串所有字符组成的迭代器

7.4 创建一个新的空字符串对象 new()

如果要创建一个新的空字符串对象，我们可以调用 new() 方法。

```
fn main(){
   let mut z = String::new();
   z.push_str("零基础教程 简单编程");
   println!("{}",z);
}
```

编译运行以上 Rust 代码，输出结果如下

```
零基础教程 简单编程
```

7.5 字符串字面量转换为字符串对象 to_string()

字符串字面量是没有任何操作方法的，它仅仅只保存了 字符串 本身。

其实是有的，就是 to_string()

如果要对字符串进行一些操作，就必须将字符串转换为字符串对象。而这个转换过程，可以通过调用 to_string() 方法

```
fn main(){
   let name1 = "你好，零基础教程
   简单编程".to_string();
   println!("{}",name1);
}
```

编译运行以上 Rust 代码，输出结果如下

```
你好，零基础教程
   简单编程
```

7.6 字符串替换 replace()

如果要一个字符串对象中的指定字符串子串替换成另一个字符串，可以调用 **replace()** 方法。

replace() 方法接受两个参数：

- 第一个参数是要被替换的字符串子串或模式。
- 第二个参数是新替换的字符串。

> 注意: replace() 会搜索和替换所有要被替换的字符串子串或模式。

7.6.1 范例

下面的代码，我们搜索字符串中所有的 程 字，并替换成 www.badu.com

```
fn main(){
   let name1 = "零基础教程 简单编
   程".to_string(); //原字符串对象
   let name2 = name1.replace("程","www.badu.com");    // 查找并替换
   println!("{}",name2);
}
```

编译运行以上 Rust 代码，输出结果如下

```
简单教www.badu.com 简单编
   www.badu.com
```

7.7 将字符串对象转换为字符串字面量 as_str()

字符串字面量就是字符串那些字符，比如

```
let name1 = "零基础教程";
```

name1 是一个字符串字面量，它只包含 零基础教程 四个字本身。

字符串字面量只包含字符串本身，并没有提供相应的操作方法。

如果要返回一个字符串对象的 **字符串** 字面量，则可以调用 **as_str()** 方法

```
fn main() {
   let example_string = String::from("零基础教程");
   print_literal(example_string.as_str());
}
fn print_literal(data:&str ){
   println!("显示的字符串字面量是: {}",data);
}
```

编译运行以上 Rust 代码，输出结果如下

```
显示的字符串字面量是: 零基础教程
```

7.8 原字符串后追加字符 push()

如果要在一个字符串后面追加字符则首先需要将该字符串声明为 **可变** 的，也就是使用 mut 关键字。然后再调用 **push()** 方法。

push() 是在原字符上追加，而不是返回一个新的字符串

```
fn main(){
   let mut company = "零基础教程".to_string();
   company.push('t');
   println!("{}",company);
}
```

编译运行以上 Rust 代码，输出结果如下

```
零基础教程t
```

7.9 原字符串后追加字符串 push_str()

如果要在一个字符串后面追加字符串则首先需要将该字符串声明为 **可变** 的，也就是使用 mut 关键字。然后再调用 **push_str()** 方法。

push_str() 是在原字符上追加，而不是返回一个新的字符串

```
fn main(){
   let mut company = "零基础教程".to_string();
   company.push_str(" 简单编程");
   println!("{}",company);
}
```

编译运行以上 Rust 代码，输出结果如下

```
零基础教程 简单编程
```

7.10 获取字符串长度 len()

如果要返回字符串中的总字节数可以使用 **len()** 方法。

len() 方法会统计所有的字符，包括空白符。

空白符是指 制表符 \t、空格 、回车 \r、换行 \n 和回车换行 \r\n 等等。

```
fn main() {
   let fullname = " 零基础教程 简单编程 www.badu.com";
   println!("length is {}",fullname.len());
}
```

编译运行以上 Rust 代码，输出结果如下

```
length is 38
```

7.11 去除字符串头尾的空白符 trim()

空白符是指 制表符 \t、空格 、回车 \r、换行 \n 和回车换行 \r\n 等等。

如果要去掉字符串头尾的空白符，可以使用 **trim()** 方法。

该方法并不会去掉不是头尾的空白符，而且该方法会返回一个新的字符串。

```
fn main() {
   let fullname = " \t简单\t教程 \r\n简单
   编程\r\n     ";
   println!("Before trim ");
   println!("length is {}",fullname.len());
   println!();
   println!("After trim ");
   println!("length is {}",fullname.trim().len());
   println!("string is :{}",fullname.trim());
}
```

编译运行以上 Rust 代码，输出结果如下

```
Before trim
length is 41

After trim
length is 32
string is :简单  教程
简单
   编程
```

7.12 使用空白符分割字符串 split_whitespace()

空白符是指 制表符 \t、空格 、回车 \r、换行 \n 和回车换行 \r\n 等等。

根据空白符分割字符串是最常用的操作之一，为此，Rust 语言为字符串提供了 **split_whitespace()** 用于根据空白符 分割一个字符串并返回一个迭代器。

我们可以使用这个迭代器来访问分割后的字符串。

```
fn main(){
   let msg = "零基础教程 简单编程 www.badu.com
   https://www.badu.com".to_string();
   let mut i = 1;

   for token in msg.split_whitespace(){
      println!("token {} {}",i,token);
      i+=1;
   }
}
```

编译运行以上 Rust 代码，输出结果如下

```
token 1 零基础教程
token 2 简单编程
token 3 www.badu.com
token 4 https://www.badu.com
```

7.13 根据指定模式分割字符串 split()

如果要将字符串根据某些指定的 字符串子串 分割，则可以使用 **split()** 方法。

split() 会根据传递的指定 模式 （字符串分割符） 来分割字符串，并返回分割后的字符串子串组成的切片上的迭代器。我们可以通过这个迭代器来迭代分割的字符串子串。

split() 方法最大的缺点是不可重入迭代，也就是迭代器一旦使用，则需要重新调用才可以再用。

但我们可以先在迭代器上调用 collect() 方法将迭代器转换为 向量 Vector ，这样就可以重复使用了。

```
fn main() {
   let fullname = "李白，诗仙，唐朝";

   for token in fullname.split("，"){
      println!("token is {}",token);
   }

   // 存储在一个向量中
   println!("\n");
   let tokens:Vec<&str>= fullname.split("，").collect();
   println!("姓名 is {}",tokens[0]);
   println!("称号 {}",tokens[1]);
   println!("朝代 {}",tokens[2]);
}
```

编译运行以上 Rust 代码，输出结果如下

```
token is 李白
token is 诗仙
token is 唐朝

姓名 is 李白
称号 诗仙
朝代 唐朝
```

7.14 将字符串打散为字符数组 chars()

如果要将一个字符串打散为所有字符组成的数组，可以使用 **chars()** 方法。

从某些方面说，如果我们要迭代字符串中的每一个字符，则必须首先将它打散为字符数组，然后才能遍历。

```
fn main(){
   let n1 = "零基础教程 www.badu.com".to_string();

   for n in n1.chars(){
      println!("{}",n);
   }
}
```

编译运行以上 Rust 代码，输出结果如下

```
简
单
教
程

w
w
w
.
t
w
l
e
.
c
n
```

7.15 字符串连接符 +

将一个字符串追加到另一个字符串的末尾，创建一个新的字符串，我们将这种操作称之为 **连接**，有时候也称之为 **拼接** 或 **差值**。

连接 的结果是创建一个新的字符串对象。

Rust 语言使用 **加号 +** 来完成这种 **连接**，我们称之为 **字符串连接符**。

7.15.1 字符串连接符 + 的内部实现

字符串连接符 **+** 的内部实现，其实是重写了 **add()** 方法。该方法接受两个参数，第一个参数是当前的字符串对象 self ，也就是 + 左边的字符串，第二个参数是一个引用，它指向了 + 右边的字符串。

具体的方法原型如下

```
//字符串拼接符的实现
add(self,&str)->String {
   // 返回一个新的字符串对象
}
```

7.15.2 范例

下面的代码，我们使用 字符串拼接符 + 将连个字符串变量拼接成一个新的字符串

```
fn main(){
   let n1 = "零基础教程".to_string();
   let n2 = "简单编程".to_string();

   let n3 = n1 + &n2; // 需要传递 n2 的引用
   println!("{}",n3);
}
```

编译运行以上 Rust 代码，输出结果如下

```
零基础教程 简单编程
```

7.16 类型转换 to_string()

如果需要将其它类型转换为字符串类型，可以直接调用 **to_string()** 方法。

例如可以调用一个数字类型的变量的 to_string() 方法将当前变量转换为字符串类型。

```
fn main(){
   let number = 2020;
   let number_as_string = number.to_string();

   // 转换数字为字符串类型
   println!("{}",number_as_string);
   println!("{}",number_as_string=="2020");
}
```

编译运行以上 Rust 代码，输出结果如下

```
2020
true
```

7.17 格式化宏 format!

如果要把不同的变量或对象拼接成一个字符串，我们可以使用 **格式化宏 (format!)**

格式化宏 format! 的使用方法如下

```
fn main(){
   let n1 = "零基础教程".to_string();
   let n2 = "简单编程".to_string();
   let n3 = format!("{} {}",n1,n2);
   println!("{}",n3);
}
```

编译运行以上 Rust 代码，输出结果如下

```
零基础教程 简单编程
```

八、Rust 运算符

运算符用于对数据执行一些操作。

被 **运算符** 执行操作的数据我们称之为 操作数。

例如我们常见的加法运算，那么 **加号 （ + ）** 就是一个运算符。

例如

```
7 + 5 = 12
```

7 和 5 我们称为 **运算符加号（ + ）** 的操作数，而 12 则运算符操作的结果。

Rust 语言支持以下四种运算符

- 算术运算符
- 位运算符
- 关系运算符
- 逻辑运算符

8.1 算术运算符

算术运算符就是我们日常所使用的 **加减乘除求余** 五则运算。

下表列出了 Rust 语言支持的所有算术运算符。

在下表中，我们假设 a = 10 且 b = 5。

名称	运算符	范例
加	+	a+b 的结果为 15
减	-	a-b 的结果为 5
乘	*	a*b 的结果为 50
除	/	a / b 的结果为 2
求余	%	a % b 的结果为 0

> 注意: Rust 语言不支持自增自减运算符 ++ 和 --。

8.1.1 范例

下面的范例演示了我们上面提到的所有算术运算符。

```
fn main() {
   let num1 = 10 ;
   let num2 = 2;
   let mut res:i32;

   res = num1 + num2;
   println!("Sum: {} ",res);

   res = num1 - num2;
   println!("Difference: {} ",res) ;

   res = num1*num2 ;
   println!("Product: {} ",res) ;

   res = num1/num2 ;
   println!("Quotient: {} ",res);

   res = num1%num2 ;
   println!("Remainder: {} ",res);
}
```

编译运行以上 Rust 代码，输出结果如下

```
Sum: 12
Difference: 8
Product: 20
Quotient: 5
Remainder: 0
```

8.2 关系运算符

关系运算符测试或定义两个实体之间的关系类型。

关系运算符用于比较两个或多个值之间的关系，是大于，是等于还是小于。

关系运算符的返回结果为 **布尔类型**。

下表列出了 Rust 语言支持的所有关系运算符。

在下表中，我们假设 A = 10 且 B = 20。

名称	运算符	说明	范例
大于	>	如果左操作数大于右操作数则返回 true 否则返回 false	(A > B) 返回 false
小于	<	如果左操作数小于于右操作数则返回 true 否则返回 false	(A < B) 返回 true
大于等于	>=	如果左操作数大于或等于右操作数则返回 true 否则返回 false	(A >= B) 返回 false
小于等于	<=	如果左操作数小于或等于右操作数则返回 true 否则返回 false	(A <= B) 返回 true
等于	==	如果左操作数等于右操作数则返回 true 否则返回 false	(A == B) 返回 true
不等于	!=	如果左操作数不等于右操作数则返回 true 否则返回 false	(A != B) 返回 false

8.2.1 范例

下面我们就用一小段代码来演示下上面提到的这些关系运算符的作用和结果

```
fn main() {
   let A:i32 = 10;
   let B:i32 = 20;

   println!("Value of A:{} ",A);
   println!("Value of B : {} ",B);

   let mut res = A>B ;
   println!("A greater than B: {} ",res);

   res = A<B ;
   println!("A lesser than B: {} ",res) ;

   res = A>=B ;
   println!("A greater than or equal to B: {} ",res);
```

```
    res = A<=B;
    println!("A lesser than or equal to B: {}",res) ;

    res = A==B ;
    println!("A is equal to B: {}",res) ;

    res = A!=B ;
    println!("A is not equal to B: {} ",res);
}
```

编译运行以上 Rust 代码，输出结果如下

```
Value of A:10
Value of B : 20
A greater than B: false
A lesser than B: true
A greater than or equal to B: false
A lesser than or equal to B: true
A is equal to B: false
A is not equal to B: true
```

8.3 逻辑运算符

逻辑运算符用于组合两个或多个条件。

逻辑运算符的返回结果也是布尔类型。

下表列出了 Rust 语言支持的所有逻辑运算符。

在下表中，我们假设 A = 10 且 B = 20。

名称	运算符	说明	范例
逻辑与	&&	两边的条件表达式都为真则返回 true 否则返回 false	(A > 10 && B > 10) 的结果为 false
逻辑或	||	两边的条件表达式只要有一个为真则返回 true 否则返回 false	(A > 10
逻辑非	!	如果表达式为真则返回 false 否则返回 true	!(A >10) 的结果为 true

8.3.1 范例

逻辑运算符很简单，因为只有三个。

我们写一小段代码演示下如何使用逻辑运算符以及它们的计算结果。

```
fn main() {
   let a = 20;
   let b = 30;

   if (a > 10) && (b > 10) {
      println!("true");
   }
   let c = 0;
   let d = 30;

   if (c>10) || (d>10){
      println!("true");
   }
   let is_elder = false;

   if !is_elder {
      println!("Not Elder");
   }
}
```

编译运行以上 Rust 代码，输出结果如下

```
true
true
Not Elder
```

8.4 位运算符

下表列出了 Rust 支持的所有位运算操作。

我们假设变量 A = 2 且变量 B = 3。

```
A 的二进制格式为

0 0 0 0 0 0 1 0

B 的二进制位格式为

0 0 0 0 0 0 1 1
```

名字	运算符	说明	范例
位与	&	相同位都是 1 则返回 1 否则返回	0 (A & B) 结果为 2
位或	|	相同位只要有一个是 1 则返回 1 否则返回 0	(A
异或	^	相同位不相同则返回 1 否则返回 0	(A ^ B) 结果为 1
位非	! 把位中的 1 换成 0 ， 0 换成 1	(!B) 结果 -4	
左移	<<	操作数中的所有位向左移动指定位数，右边的位补 0	(A << 1) 结果为 4
右移	>>	操作数中的所有位向右移动指定位数，左边的位补 0	(A >> 1) 结果为 1

8.4.1 范例

下面的范例演示了我们上面提到的所有位运算符。

```
fn main() {
   let a:i32 = 2;     // 二进制表示为 0 0 0 0 0 0 1 0
   let b:i32 = 3;     // 二进制表示为 0 0 0 0 0 0 1 1

   let mut result:i32;

   result = a & b;
   println!("(a & b) => {} ",result);

   result = a | b;
   println!("(a | b) => {} ",result) ;

   result = a ^ b;
   println!("(a ^ b) => {} ",result);

   result = !b;
   println!("(!b) => {} ",result);

   result = a << b;
   println!("(a << b) => {}",result);

   result = a >> b;
   println!("(a >> b) => {}",result);
}
```

编译运行以上 Rust 代码，输出结果如下

```
(a & b) => 2
(a | b) => 3
(a ^ b) => 1
(!b) => -4
(a << b) => 16
(a >> b) => 0
```

九、Rust 条件判断

编程语言是模拟现实生活的，如果你有哪块搞不清楚，那么只要从现实生活中找例子即可。

我们的生活，大部分都是流水线似的，比如

```
睡醒 -> 早餐 -> 上班/上学 -> 午餐 -> 上班/上学 -> 回家 -> 晚饭 -> 睡觉
```

虽然站在每天的角度考虑，生活真的是顺风顺水，流水作业，但是，精确到每件事，就会有很多小插曲

比如

```
睡醒 -> (如果天还没亮则继续睡，否则起床吃早餐)
```

比如

```
如果是周六或周日，就不用上班或上课
```

也就是说，具体到每件事，都是根据一个条件作出判断，如果条件为真则怎么样，如果条件为假又怎么样，等等。

因为编程语言也是模拟生活的，所以，编程语言一般情况下，从入口代码开始，一条一条往下执行，比如

```
fn main()
{
    let name = "小明";         // 名字
    let firstMeet = false;    // 是否第一次见

    println!("{},你好!",name);
    println!("很高兴见到你");
    println!("我是零基础教程");
    println!("你叫什么名字");
}
```

上面这段代码，会从上往下一行一行的执行。所以输出的结果就是

```
你好!
很高兴见到你
我是零基础教程
你叫什么名字
```

但有时候，我们会想： 如果是第一次见则输出问候语，如果不是第一次见，则直接输出名字即可

对于这种现实生活中常见的需求，我们在程序中有要怎么模拟呢？

9.1 编程语言中的条件判断

我们说过编程语言也是模拟现实生活，针对上面提到的 如果...就... 或者 如果...就...否则就... 选择判断，我们称之为 **条件判断**

也就是说，条件判断就是

```
如果 ....  就 ...
```

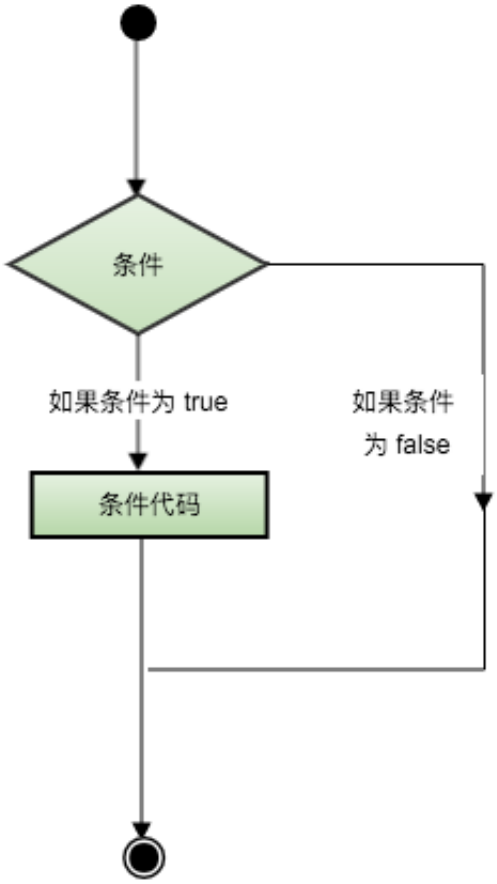
或者

```
如果 ....  就 ... 否则 ....
```

或者

```
如果 .... 就 .... 否则如果 ... 就 ... 否则如果 ... 就 .... 否则 ....
```

绝大多数的编程语言都有着相同的条件判断语句，它们都是针对一个条件的真或假作出不同的选择，基本的流程图如下



条件判断语句	说明
if 语句	if 语句用于模拟现实生活中的 如果...就...
if...else 语句	if...else 语句用于模拟 如果...就...否则...
else...if 和嵌套if 语句	嵌套if 语句用于模拟 如果...就...如果...就...
match 语句	match 语句用于模拟现实生活中的 老师点名 或 银行叫号

9.2 if 语句

我们经常会说 如果肚子饿了就吃饭，如果今天天晴，我们就去郊游，如果....。

我们把这种 如果...就 语句叫做 条件语句，其中 肚子饿了、天晴 等叫做 条件，而 吃饭、郊游 叫做条件为真时要执行的 动作。

Rust 语言中使用 if 语句来模拟现实生活中这种 **如果...就** 的情况。

9.2.1 if 语句语法

```
if boolean_expression {
    // boolean_expression 为 true 时要执行的语句
    // statement(s)
}
```

因此，if 语句简单来说，就是

1. 如果 boolean_expression 为 true ，则执行大括号里的 statement(s)。
2. 如果 boolean_expression 为 false ，则执行大括号后面的第一条语句。

把 **如果肚子饿了就吃饭** 格式化为 if 语句就是

```
if 肚子饿了 {
   吃饭
}
```

9.2.2 范例

我们写一个范例，使用 if 语句来模拟 如果数字大于 0 则输出 正数。

```
fn main(){
   let num:i32 = 5;
   if num > 0 {
      println!("正数");
   }
}
```

编译运行以上 Rust 代码，输出结果如下

```
正数
```

9.3 if else 语句

生活既有着流水线式的日复一日，也有着只有一个选择题的如果。 然而只有一个选择题的如果真的只有一个选择吗?

不是的，不是还有另一个选择，就是什么都不做吗?

如果另一个选择是做些什么，那么，不就变成了

```
如果 ... 就 ... 否则就...
```

编程语言的创建者早就想到了这种二选一的情况，特意为 if 语句后面跟着一个可选的 else 语句。

这种二选一的 if 语句，我们称之为 if else 语句。

9.3.3 if..else 语句语法格式

```
if boolean_expression {
   // 如果 boolean_expression 为真则执行这里的代码
} else {
   // 如果 boolean_expression 为假则执行这里的代码
}
```

9.3.4 if..else 语句流程图

1. 在 if else 语句中，if 语句才是最主要的。如果 条件 为真，就没 else 语句啥事了。
2. 其实 if 语句后面的 else 语句是可选的。就像我们所说的，如果条件为假就什么都不做，那要 else 语句有什么用呢？else 语句的唯一作用，就是 if 语句中的 条件 为假时做些什么，执行些什么

9.3.5 范例

我们写一段代码，使用 if else 语句来判断一个数是否偶数或奇数，如果是偶数则输出 偶数 如果是奇数则输出 奇数

```
fn main() {
   let num = 12;
   if num % 2==0 {
      println!("偶数");
   } else {
      println!("奇数");
   }
}
```

编译运行以上 Rust 代码，输出结果如下

```
偶数
```

9.4 嵌套 If 语句

现实生活中肯定不会只有 如果...不然就 这种二选一的情况，还存在多个 如果 的情况。

比如 如果今天天晴，我们就去郊游，如果下雪，我们就堆雪人，如果下雨，我们就逛商场，如果下暴雨，我们就在家看电视。

面对多个 如果 这种情况，我们可以使用 嵌套 if 语句。

嵌套 if 语句用于测试多种条件。

9.4.1 语法

嵌套 If 语句的语法格式如下

```
if boolean_expression1 {
   // 当 boolean_expression1 为 true 时要执行的语句
} else if boolean_expression2 {
   // 当 boolean_expression2 为 true 时要执行的语句
} else {
   // 如果 boolean_expression1 和 boolean_expression2 都为 false 时要执行的语句
}
```

是不是看起来有点复杂了。

使用嵌套 if 语句，也就是 if...else if... 语句时需要牢记几个点：

- 任何一个 if 或者嵌套 if 语句可以有 0 个或 1 个 else 语句，但 else 语句必须出现在 if else 后面，也就是出现在最后。
- 任何一个 if 或者嵌套 if 语句可以有 0 个或多个 if else 语句，但所有的 if else 语句都必须出现在 else 语句之前。
- 一旦某个 else if 中的条件 boolean_expression1 返回 true，那么后面的 else if 和 else 语句都不会运行。

9.4.2 范例

我们使用嵌套 if 语句来写一段代码，判断某个值是 大于、小于、等于 0。

```
fn main() {
   let num = 2 ;
   if num > 0 {
      println!("{} is positive",num);
   } else if num < 0 {
      println!("{} is negative",num);
   } else {
      println!("{} is neither positive nor negative",num) ;
   }
}
```

编译运行以上 Rust 代码，输出结果如下

```
2 is positive
```

9.5 match 语句

match 语句用于检查某个当前的值是否匹配一组/列值 中的某一个。

如果放到现实生活中，那么 match 语句类似于 老师点名 或者 银行叫号。当老师叫一个名字，比如 小明 时，叫 小明 的那个人就会应答，比如说 到。

如果你会 C 语言，那么 Rust 中的 match 表达式则类似于 C 语言中的 switch 语句。

对 match 语句有了个大概的印象之后，我们来看看 Rust 中的 switch 语句的语法格式

9.5.1 match 语句语法格式

Rust 中一个基本的 match 语句语法格式如下

```
match variable_expression {
   constant_expr1 => {
      // 语句;
   },
   constant_expr2 => {
      // 语句;
   },
   _ => {
      // 默认
      // 其它语句
   }
};
```

match 语句有返回值，它把 匹配值 后执行的最后一条语句的结果当作返回值。

```
let expressionResult = match variable_expression {
   constant_expr1 => {
      // 语句;
   },
   constant_expr2 => {
      // 语句;
   },
   _ => {
      // 默认
      // 其它语句
   }
};
```

我们来分析下上面这两个 match 语句的语法：

1. 首先要说明的是 match 关键字后面的表达式不必括在括号中。也就是 variable_expression 不需要用一对 括号(()) 括起来。
2. 其次，match 语句在执行的时候，会计算 variable_expression 表达式的值，然后把计算后的结果和每一个 constant_exprN 匹配，使用的是 全等于 也就是 === 来匹配。如果匹配成功则执行 => {} 里面的语句。
3. 如果 variable_expression 表达式的值没有和任何一个 constant_exprN 匹配，那么它会默认匹配 _。因此，当没有匹配时，默认会执行 _ => {} 中的语句。
4. match 语句有返回值，它把 匹配值 后执行的最后一条语句的结果当作返回值。
5. _ => {} 语句是可选的，也就是说 match 语句可以没有它。
6. 如果只有一条语句，那么每个 constant_expr2 => {} 中的 {} 是可以省略的。

9.5.2 范例1

看起来 match 语句有点复杂，我们直接拿几个范例来说明下

```
fn main(){
   let state_code = "MH";
   let state = match state_code {
      "MH" => {println!("Found match for MH"); "Maharashtra"},
      "KL" => "Kerala",
      "KA" => "Karnadaka",
      "GA" => "Goa",
      _ => "Unknown"
   };
   println!("State name is {}",state);
}
```

编译运行以上 Rust 代码，输出结果如下

```
Found match for MH
State name is Maharashtra
```

上面的范例中，state_code 对应着语法中的 variable_expression，MH、KL、KA、GA 对应这语法中的 constant_expr1 、constant_expr2 等等。

因为我们的 variable_expression 的值为 MH 和值为 MH 的第一个 constant_expr1 匹配，因此会执行 `{println!("Found match for MH"); "Maharashtra"}`。 然后将执行的最后一条语句的结果，也就是 "Maharashtra" 作为整个表达式的值返回。

这样，我们的 state 变量就被赋值为 Maharashtra。

9.5.3 范例 2

上面这个范例是匹配的情况，如果不匹配，那么就会执行 _ => 语句

```
fn main(){
   let state_code = "MS";
   let state = match state_code {
      "MH" => {println!("Found match for MH"); "Maharashtra"},
      "KL" => "Kerala",
      "KA" => "Karnadaka",
      "GA" => "Goa",
      _ => "Unknown"
   };
   println!("State name is {}",state);
}
```

编译运行以上 Rust 代码，输出结果如下

```
Unknown
State name is Unknown
```

十、Rust 循环语句

现实的世界很无奈，因为它是线性的，流水线形式的。

过完今天，还有明天，过完明天，还有后天。

具体到每一天，无外乎是

```
睡醒 -> 早餐 -> 上班/上学 -> 午餐 -> 上班/上学 -> 回家 -> 晚饭 -> 睡觉
```

虽然站在每天的角度考虑，生活真的是顺风顺水，流水作业。但如果站在上帝的视角，白天生活无外乎是

```
吃饭 -> 工作 -> 吃饭 -> 工作 -> 吃饭 -> 工作
```

好像陷入了一种死循环的状态。

如果我们把时间缩短到每个小时，那么白天的生活几乎就是

```
9:30  上班
10:30 上班
11:30 上班
12:00 下班
13:30 上班
14:30 上班
15:30 上班
16:30 上班
17:30 上班
18:30 下班
```

如果你撇开时间不看，那白天都在上班上班上班....这样循环下去。

即使加入了时间，看起来除了 12:00 和 18:30 是下班外，每个小时都在循环工作的赶脚。

循环其实就是一种重复，在满足指定的条件下，重复的做某些事情。就好比如只要时间没到 18:30，那么我们一直在重复的上班。

编程语言是模拟现实生活的，如果你有哪块搞不清楚，那么只要从现实生活中找例子即可。

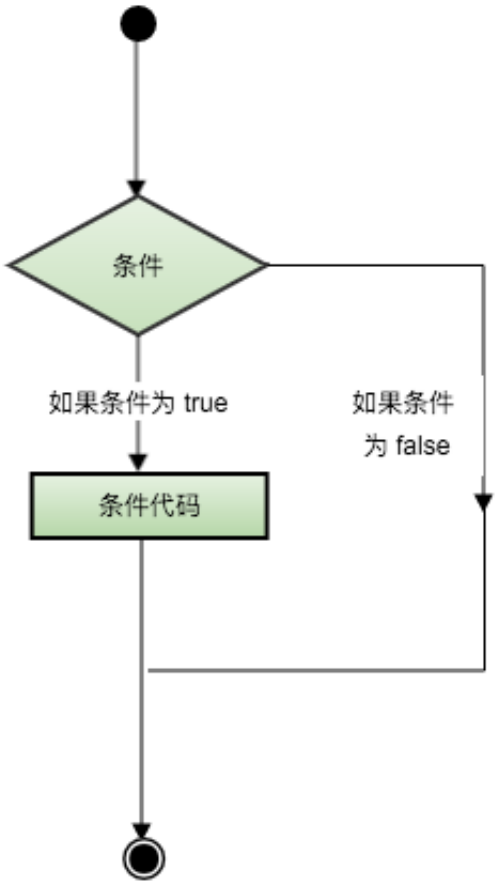
编程语言也有循环这种概念，也可以在满足指定的条件下，重复的执行一部分代码，我们把这种重复执行的代码称之为 转换语句

10.1 Rust 中的循环

循环语句允许我们多次执行语句或语句组。

循环语句就是在指定条件为真的情况下重复的执行代码。

这个重复的过程，我们使用一张图来描绘就会像下面这样子



有时候不得不感慨中文的博大精深，就一个重复执行的动作，我们有很多种描述方法

1. 太阳每天都东升西落。在人的生命周期中，永远看不到结束的时候，这是一种死循环。
2. 只要还趴下住院，你就一直要重复的工作...。这是一种有条件的循环，只要条件为真就要重复执行。
3. 做错了事，总是被罚站 30 分钟。1分钟...2分钟...30 分钟。这种循环有初始化，有临界条件，还会自动递增时间。

对应着上面三种重复，Rust 语言中也有三种表示 **循环** 的语句：

- loop 语句。一种重复执行且永远不会结束的循环。
- while 语句。一种在某些条件为真的情况下就会永远执行下去的循环。
- for 语句。一种有确定次数的循环。

三种语句都能实现循环功能，只不过侧重点不一样。

从上面的描述中，根据循环是否可能结束，我们把循环分为两大类

1. 能确定次数的循环，比如 for 循环。
2. 满足条件就是永动机的循环。比如 while 循环。
3. 死循环。比如 loop 循环。

10.2 for 循环语句

for 语句是一种能确定循环次数的循环。

for 语句用于执行代码块指定的次数。

可能和其它语言有所不同，Rust 中的 for 循环只有 for..in 这种格式，常用于迭代一组固定的值，例如数组、向量等。

10.2.1 for 循环语句的语法格式

```
for temp_variable in lower_bound..upper_bound {
   // action 要重复执行的代码
}
```

- lower_bound..upper_bound 用于指定循环区间，是一个左闭又开的区间，比如 1..3 则只有 1 和 2 不包含 3
- temp_variable 是循环迭代的每一个元素。
- temp_variable 变量的作用域仅限于 for...in 语句，超出则会报错

10.2.2 范例: 基本的 for...in 循环

下面的代码，使用 for...in 循环，重复输出 1 到 11 之间的数字（不包括 11 ）

```
fn main(){
   for x in 1..11{
      println!("x is {}",x);
   }
}
```

编译运行以上 Rust 代码，输出结果如下

```
x is 1
x is 2
x is 3
x is 4
x is 5
x is 6
x is 7
x is 8
x is 9
x is 10
```

10.3 while 循环

while 语句是一种只要条件为真就一直会重复执行下去的循环。

while 循环会在每次重复执行前先判断条件是否满足，满足则执行，不满足则退出。

10.3.1 while 语句的语法格式

```
while ( condition ) {
    // action 要重复执行的代码
}
```

condition 是一个表达式，返回的结果会转换为 布尔类型。只要 condition 返回真，那么循环就会一直执行。

10.3.2 范例： 基本的 while 循环

下面的代码，使用 while 循环重写下上面的代码，重复输出 1 到 11 之间的数字（不包括 11 ）

```
fn main(){
   let mut x = 1;
   while x < 11{
      println!("inside loop x value is {}",x);
      x+=1;
   }
   println!("outside loop x value is {}",x);
}
```

编译运行以上 Rust 代码，输出结果如下

```
inside loop x value is 1
inside loop x value is 2
inside loop x value is 3
inside loop x value is 4
inside loop x value is 5
inside loop x value is 6
inside loop x value is 7
inside loop x value is 8
inside loop x value is 9
inside loop x value is 10
outside loop x value is 11
```

看到最后的 outside loop x value is 11 没，因为当 x 递增到 11 就不再满足条件 x < 11 了。

10.4 loop 循环

loop 语句代表着一种死循环。它没有循环条件，也没有循环次数，它就是一个永动机。

10.4.1 loop 语句的语法格式

```
loop {
    // action 要重复执行的代码
}
```

10.4.2 范例：loop 循环的基本使用

下面的语句，我们使用 loop 输出 1 到无限大的数字。

实际上是不可能的

```
fn main(){
   let mut x = 0;
   loop {
      x+=1;
      println!("x={}",x);
   }
}
```

编译运行以上 Rust 代码，输出结果如下

```
x=1
x=2
x=3

...
...

x=61^Cx=612514
x=612515
x=612516
```

这是一个死循环，如果不出意外是不会停下来的。需要我们手动退出程序，这时候你可以按下 CTRL + C 组合建退出程序。

> 注意: 任何程序，如果进入死循环，你可以按下 CTRL + C 组合键退出程序。

10.5 循环控制语句 break

好啦好啦，我们已经学会了三大循环语句了，大家用起来感觉如何？？？

在日常的编程中，不知道大家有没有这样的请求，如果 action 语句块中也能退出循环就好了。

哈哈，为了响应这个请求，语言的开发者们早点就想到了这一块了，于是有了 break 语句。

也就是说 break 语句的出现，就是为了在 action 语句块中可以退出循环语句。

10.5.1 break 语句的语法

break 语句的语法格式如下

```
break;
```

很简洁，不拖泥带水的。

10.5.2 范例

小试牛刀一下，我们使用 break 语句改造下我们的 loop 循环，在 x 大于 11 就退出循环。

也就是使用 loop 循环实现 while 循环

```
fn main(){
   let mut x = 0;
   loop {
      x+=1;
      if x > 10 {
        break;
      }
      println!("x={}",x);
   }
}
```

编译运行以上 Rust 代码，输出结果如下

```
x=1
x=2
x=3
x=4
x=5
x=6
x=7
x=8
x=9
x=10
```

10.6 循环控制语句 continue

break 语句让我们尝到了甜头，有时候我们就会突发奇想，有没有另一个关键字，不像 break 语句那样直接退出整个循环，而仅仅是退出当前循环。也就是剩下的语句不执行了，开始下一个迭代。

创造了那些语言的前辈们，自然也会有这个想法，于是造出了 continue 语句。

continue 语句，简单的说，就是停止执行剩下的语句，直接进入下一个循环。

10.6.1 continue 语句的语法格式

continue 语句的语法格式如下

```
continue;
```

10.6.2 范例

下面的代码，我们使用 for 循环输出 1 到 11 之间的数字，但是跳过数字 5

```
fn main(){
    for x in 1..11{
        if 5 == x {
            continue;
        }
        println!("x is {}",x);
    }
}
```

编译运行以上 Rust 代码，输出结果如下

```
x is 1
x is 2
x is 3
x is 4
x is 6
x is 7
x is 8
x is 9
x is 10
```

十一、Rust 函数 fn

函数是一组一起执行一个任务的语句块。

函数是一段可读，可维护和可重用代码的多条语句。

每个 Rust 程序都至少有一个函数，即主函数 main()。

除了使用 Rust 核心和标准库提供的函数外，我们还可以自己定义自己的函数。

11.1 划分代码到函数中

我们可以把代码划分到不同的函数中，这样可以使得代码可读性更强，逻辑更简单。

虽然划分代码到不同的函数中没有一个统一的规范，但实践证明，在逻辑上，划分的标准是每个函数执行一个特定的任务的

函数声明就是告诉编译器一个函数的名称、变量、和返回值类型。这三个合在一起组成了函数的签名，函数签名的作用就是防止出现两个相同的函数。

函数定义，就是提供了函数的具体实现

术语	说明
函数定义	函数定义其实就是定义一个任务要以什么方式来完成
函数调用	函数只有被调用才会运行起来
函数返回值	函数在执行完成后可以返回一个值给它的调用者
函数参数	函数参数用于携带外部的值给函数内部的代码

11.2 函数定义

函数定义其实就是定义一个任务要以什么方式来完成。

因此，定义函数时首先想的并不是我要定义一个函数，而是我这个任务要怎么做，要定义哪些函数来完成。

函数也不会凭空出现的，在使用一个函数前，我们必须先定义它。

定义函数时必须以 **fn** 关键字开头，**fn** 关键字是 **function** 的缩写。

函数内部必须包含函数要执行的具体代码，我们把这些代码称之为 **函数体**。

函数名称的命名规则和变量的命名规则一致。

11.2.1 定义函数的语法

定义函数的语法如下，定义函数时必须使用 fn 关键字开头，后面跟着要定义的函数名。

```
fn function_name([param1:data_type1,param2..paramN]) {
   // 函数代码
}
```

参数用于将值传递给函数内部的语句。函数定义时可以自由选择包含参数与否。

11.2.2 范例：定义一个简单的函数

下面的代码，我们定义了一个函数名为 fn_hello 的函数，用于输出一些信息

```
// 函数定义
fn fn_hello(){
   println!("hello from function fn_hello ");
}
```

11.3 函数调用

为了运行一个函数首先必须调用它。函数不像普通的语句，写完了会自动执行，函数需要调用才会被执行。否则看起来就像是多余的没有用的代码。

让函数运行起来的过程我们称之为 函数调用。

如果函数定义了参数，那么在 函数调用 时必须传递指定类型的参数。

在一个函数 fn1 内部调用其它函数 fn2，那么这个 fn1 函数就称为 调用者函数，简称为 调用者。

调用者函数有点拗口，我们一般都称呼为 函数调用者。

11.3.1 函数调用的语法格式

```
function_name(val1,val2,valN)
```

例如我们在 main() 函数内部调用函数 fn_hello()

```
fn main(){
   //调用函数
   fn_hello();
}
```

这时候，函数 main() 就是 调用者函数，也就是 调用者。

11.3.2 范例

下面的代码，我们定义了一个函数 fn_hello() 用于输出一些信息。 然后我们在 main() 函数对 fn_hello() 进行调用

```
fn main(){
   // 调用函数
   fn_hello();
}


// 定义函数
fn fn_hello(){
   println!("hello from function fn_hello ");
}
```

编译运行以上 Rust 代码，输出结果如下

```
hello from function fn_hello
```

11.4 函数返回值

函数可以返回值给它的调用者。我们将这些值称为 函数返回值。

也就是说，函数在代码执行完成后，除了将控制权还给调用者之外，还可以携带值给它的调用者。

如果一个函数需要返回值给它的调用者，那么我们在函数定义时就需要明确中函数能够返回什么类型的值。

11.4.1 函数返回值语法格式

Rust 语言的返回值定义语法与其它语言有所不同，它是通过在 小括号后面使用 **箭头 (->) 加上数据类型** 来定义的。

同时在函数的代码中，可以使用 return 关键字指定要返回的值。

如果函数代码中没有使用 return 关键字，那么函数会默认使用最后一条语句的执行结果作为返回值。

因此，千万注意，return 中返回的值或最后一条语句的执行结果 必须和函数定义时的返回数据类型一样，不然会编译会出错

具有返回值的函数的完整定义语法如下

1. 有 return 语句

```
function function_name() -> return_type {
   // 其它代码

   // 返回一个值
   return value;
}
```

2. 没有 return 语句则使用最后一条语句的结果作为返回值

函数中最后用于返回值的语句不能有 **分号 ;** 结尾，否则就不会时返回值了。

function function_name() -> return_type { // 其它代码

value // 没有分号则表示返回值 }

11.4.2 范例1

下面的代码，我们定义了两个相同功能的 get_pi() 和 get_pi2() 函数，一个使用 return 语句返回值，另一个则使用没有分号的最后一条语句作为返回值。

```
fn main(){
   println!("pi value is {}",get_pi());
   println!("pi value is {}",get_pi2());
}

fn get_pi()->f64 {
   22.0/7.0
}

fn get_pi2()->f64 {
   return 22.0/7.0;
}
```

编译运行以上 Rust 代码，输出结果如下

```
pi value is 3.142857142857143
pi value is 3.142857142857143
```

11.4.3 范例 2

我们修改下上面的代码，在没有 return 的那个函数的最后一条语句添加一个分号，看看执行结果如何

```
fn main(){
   println!("pi value is {}",get_pi());
}

fn get_pi()->f64 {
   22.0/7.0;
}
```

编译运行上面的代码，会报错，错误信息如下

```
error[E0308]: mismatched types
 --> src/main.rs:5:14
  |
5 | fn get_pi()->f64 {
  |    ------    ^^^ expected f64, found ()
  |    |
  |    this function's body doesn't return
6 |    22.0/7.0;
  |             - help: consider removing this semicolon
  |
  = note: expected type `f64`
             found type `()`
```

从错误信息中可以看出，函数定义了返回值，但我们却没有返回值。也就是说，函数的返回值必须没有分号 结尾。

11.4.4 范例3： 函数返回值接收变量

我们还可以将函数的返回值赋值给一个变量。

例如下面的代码，我们定义了变量 pi 来接收函数的返回值

```
fn main(){
   let pi = get_pi();
   println!("pi value is {}",pi);
}

fn get_pi()->f64 {
   22.0/7.0
}
```

编译运行以上 Rust 代码，输出结果如下

```
pi value is 3.142857142857143
```

11.5 函数参数

函数参数 是一种将外部变量和值带给函数内部代码的一种机制。函数参数是函数签名的一部分。

函数签名的最大作用，就是防止定义两个相同的签名的函数。

当一个函数定义了函数参数，那么在调用该函数的之后就可以把变量/值传递给函数。

我们把函数定义时指定的参数名叫做 **形参**。同时把调用函数时传递给函数的变量/值叫做 **实参**。

除非特别指定，函数调用时传递的 **实参** 数量和类型必须与 **形参** 数量和类型必须相同。 否则会编译错误。

函数参数有两种传值方法，一种是把值直接传递给函数，另一种是把值在内存上的保存位置传递给函数。

11.5.1 传值调用

传值调用 就是简单的把传递的变量的值传递给函数的 **形参**，从某些方面说了，就是把函数参数也赋值为传递的值。 因为是赋值，所以函数参数和传递的变量其实是各自保存了相同的值，互不影响。因此函数内部修改函数参数的值并不会影响外部变量的值。

11.5.2 范例

我们定义了一个函数 mutate_no_to_zero()，它接受一个参数并将参数重新赋值为 0 。

我们还定义了一个变量 no 并初始化它的值为 5。然后将该变量传递给函数 mutate_no_to_zero()。

虽然我们在函数中将变量的值改成了 0，但当调用完成后，我们的 no 变量的值仍然是 5。

这是因为传值调用传递的是变量的一个副本，也就是重新创建了一个变量。

```
fn main(){
   let no:i32 = 5;
   mutate_no_to_zero(no);
   println!("The value of no is:{}",no);
}

fn mutate_no_to_zero(mut param_no: i32) {
   param_no = param_no*0;
   println!("param_no value is :{}",param_no);
}
```

编译运行以上 Rust 代码，输出结果如下

```
param_no value is :0
The value of no is:5
```

11.5.3 引用调用

值传递变量导致重新创建一个变量。但引用传递则不会，引用传递把当前变量的内存位置传递给函数。

对于引用传递来说，传递的变量和函数参数都共同指向了同一个内存位置。

引用传递需要函数定义时在参数类型的前面加上 **&** 符号，语法格式如下

```
fn function_name(parameter: &data_type) {
   // 函数的具体代码
}
```

11.5.4 范例

我们对刚刚传值调用的代码做一些修改。

我们定义了一个函数 mutate_no_to_zero()，它接受一个可变引用作为参数，并把传递的引用变量重新赋值为 0 。

同时定义了一个 可变变量 no 并初始化它的值为 5。然后将该变量的一个引用传递给函数 mutate_no_to_zero()。

当函数执行完成后，可变变量 no 的值就变成 0 了。

```
fn main() {
   let mut no:i32 = 5;
   mutate_no_to_zero(&mut no);
   println!("The value of no is:{}",no);
}
fn mutate_no_to_zero(param_no:&mut i32){
   *param_no = 0; //解引用操作
}
```

编译运行以上 Rust 代码，输出结果如下

```
The value of no is 0.
```

上面的代码中，星号 (*) 用于访问变量 param_no 指向的内存位置上存储的变量的值。这种操作也称为 解引用。 因此 星号（*） 也称为 解引用操作符。

11.6 传递复合类型给函数做参数

对于复合类型，比如字符串，如果按照普通的方法传递给函数后，那么该变量将不可再访问

例如下面的代码编译会报错

```
fn main(){
   let name:String = String::from("TutorialsPoint");
   display(name);
   println!("after function name is: {}",name);
}

fn display(param_name:String){
   println!("param_name value is :{}",param_name);
}
```

编译上面的代码会出错，错误信息如下

```
error[E0382]: borrow of moved value: `name`
 --> src/main.rs:4:42
  |
2 |    let name:String = String::from("TutorialsPoint");
  |        ---- move occurs because `name` has type `std::string::String`, which does
not implement the `Copy` trait
3 |    display(name);
  |            ---- value moved here
4 |    println!("after function name is: {}",name);
  |                                          ^^^^ value borrowed here after move
```

修复的方法之一就是去掉后面的 println!() 语句

```
fn main(){
   let name:String = String::from("TutorialsPoint");
   display(name);
}

fn display(param_name:String){
   println!("param_name value is :{}",param_name);
}
```

编译运行以上 Rust 代码，输出结果如下

```
param_name value is :TutorialsPoint
```

后面的章节，我们会讨论如何解决这个问题，本章节这不是重点。

十二、Rust 元组 tuple

Rust 支持元组 tuple。而且元组是一个 **复合类型** 。

标量类型 vs 复合类型

1. 基础类型/标量类型 只能存储一种类型的数据。例如一个 i32 的变量只能存储一个数字。
2. **复合类型** 可以存储多个不同类型的数据。

复合类型就像我们的菜篮子，里面可以放各种类型的菜。

12.1 元组

元组有着固定的长度。而且一旦定义，就不能再增长或缩小。

元组的下标从 0 开始。

12.1.1 元组定义语法

Rust 语言中元组的定义语法格式如下

```
let tuple_name:(data_type1,data_type2,data_type3) = (value1,value2,value3);
```

定义元组时也可以忽略数据类型

```
let tuple_name = (value1,value2,value3);
```

Rust 中元组的定义很简单，就是使用一对小括号 () 把所有元素放在一起，元素之间使用逗号 , 分隔。

定义元组数据类型的时候也是一样的。

但需要注意的是，如果显式指定了元组的数据类型，那么数据类型的个数必须和元组的个数相同，否则会报错。

12.1.2 范例 1

如果要输出元组中的所有元素，必须使用 {:?} 格式化符。

```
fn main() {
   let tuple:(i32,f64,u8) = (-325,4.9,22);
   println!("{:?}",tuple);
}
```

编译运行以上 Rust 代码，输出结果如下

```
(-325, 4.9, 22)
```

仅仅使用下面的输出语句是不能输出元组中的元素的。

```
println!("{ }",tuple)
```

因为元组是一个 **复合类型**，要输出复合类型的数据，必须使用 println!("{:?}", tuple_name)

12.1.3 访问元组中的单个元素

我们可以使用 **元组名.索引数字** 来访问元组中相应索引位置的元素。索引从 0 开始。

例如下面这个拥有 3 个元素的元组

```
let tuple:(i32,f64,u8) = (-325,4.9,22);
```

我们可以通过下面的方式访问各个元素

```
tuple.0  // -325

tuple.1  // 4.9

tuple.2  // 22
```

12.1.4 范例

下面的范例演示了如何通过 **元组名.索引数字** 方式输出元组中的各个元素

```
fn main() {
   let tuple:(i32,f64,u8) = (-325,4.9,22);
   println!("integer is :{:?}",tuple.0);
   println!("float is :{:?}",tuple.1);
   println!("unsigned integer is :{:?}",tuple.2);
}
```

编译运行以上 Rust 代码，输出结果如下

```
integer is :-325
float is :4.9
unsigned integer is :2
```

12.2 元组也可以作为函数的参数

Rust 语言中，元组也可以作为函数的参数。

函数参数中元组参数的声明语法和声明一个元素变量是相似的

```
fn function_name(tuple_name:(i32,bool,f64)){}
```

12.2.1 范例

下面这段代码，我们声明一个函数 print，它接受一个元组作为参数并打印元组中的所有元素

```
fn main(){
   let b:(i32,bool,f64) = (110,true,10.9);
   print(b);
}

// 使用元组作为参数
fn print(x:(i32,bool,f64)){
   println!("Inside print method");
   println!("{:?}",x);
}
```

编译运行以上 Rust 代码，输出结果如下

```
Inside print method
(110, true, 10.9)
```

12.2.2 元组解构赋值 (destructing) =

解构赋值 (destructing) 就是把元组 (tuple) 中的每一个元素按照顺序一个一个赋值给变量。

12.2.3 元组解构赋值 (destructing) 语法格式

元组解构赋值 (destructing) 的语法格式如下

```
(age,is_male,cgpa) = (30,true,7.9);
```

上面这种赋值操作称之为 元组解构赋值，它会把等号 (=) 右边的元组的元素按照顺序一个一个赋值给等号左边元组里的变量。

赋值完成后，左边的各个变量的值为

```
age      = 30;
is_male  = true;
cgpa     = 7.9;
```

解构 操作是 Rust 语言的一个特性，最新的 JavaScript 语言也有解构操作。

12.2.4 范例

```
fn main(){
   let b:(i32,bool,f64) = (30,true,7.9);
   print(b);
}
fn print(x:(i32,bool,f64)){
   println!("Inside print method");
   let (age,is_male,cgpa) = x; //assigns a tuple to
   distinct variables
   println!("Age is {} , isMale? {},cgpa is
   {}",age,is_male,cgpa);
}
```

编译运行以上 Rust 代码，输出结果如下

```
Inside print method
Age is 30 , isMale? true,cgpa is 7.9
```

十三、Rust 数组

虽然我们看到的绝大多数变量都是基本数据类型。虽然这些基本数据类型的能够满足大部分的工作，但它们不是万能的。

基本数据类型的变量也有它们的局限性。

- 基本数据类型的变量本质上是 标量。这意味着每个基本数据类型的变量一次只能存储一个值。 因此，当我们需要存储多个值的时候，我们不得不重复定义多个变量。比如 a1、a2、a3 .... 如果我们要存储的值非常多，成百上千，这种重复定义变量的方法是行不通的。
- 基本数据类型的变量的值在内存中的位置是随机的。多个按照顺序定义的变量，它们在内存中的存储位置不一定是连续的。因此我们可能按照变量的声明顺序来获取它们的值。

数组 是用来存储一系列数据，但它往往被认为是一系列相同类型的变量，也就是说，数组 是可以存储一个固定大小的相同类型元素的顺序集合。

数组的声明并不是声明一个个单独的变量，比如 number0、number1、...、number99，而是声明一个数组变量，比如 numbers，然后使用 numbers[0]、numbers[1]、...、numbers[99] 来代表一个个单独的变量。 数组中的特定元素可以通过索引访问

数组可以理解为相同数据类型的值的集合。

13.1 数组的特性

- 数组的定义其实就是为分配一段 **连续的相同数据类型** 的内存块。
- 数组是静态的。这意味着一旦定义和初始化，则永远不可更改它的长度。
- 数组的元素有着相同的数据类型，每一个元素都独占者数据类型大小的内存块。 也就是说。数组的内存大小等于数组的长度乘以数组的数据类型。
- 数组中的每一个元素都按照顺序依次存储，这个顺序号既代表着元素的存储位置，也是数组元素的

唯一标识。我们把这个标识称之为 数组下标 。 注意，数组下标从 0 开始。
- 填充数组中的每一个元素的过程称为 **数组初始化**。也就是说 数组初始化 就是为数组中的每一个元素赋值。
- 可以更新或修改数组元素的值，但不能删除数组元素。如果要删除功能，你可以将它的值赋值为 0 或其它表示删除的值。

13.2 声明和初始化数组

Rust 语言为数组的声明和初始化提供了 3 中语法

1. 最基本的语法，指定每一个元素的初始值

```
let variable_name:[dataType;size] = [value1,value2,value3];
```

例如

```
let arr:[i32;4] = [10,20,30,40];
```

2. 省略数组类型的语法

因为指定了每一个元素的初始值，所以可以从初始值中推断出数组的类型

```
let variable_name = [value1,value2,value3];
```

例如

```
let arr = [10,20,30,40];
```

指定默认初始值的语法，这种语法有时候称为 默认值初始化。

如果不想为每一个元素指定初始值，则可以为所有元素指定一个默认的初始值。

```
let variable_name:[dataType;size] = [default_value_for_elements,size];
```

例如下面的代码为每一个元素指定初始值为 -1

```
let arr:[i32;4] = [-1;4];
```

13.2.1 数组初始化：简单的数组

数组初始化的语法一般如下

```
let variable_name = [value1,value2,value3];
```

这种一种最基本的初始化方法，也是字符最长的初始化方法，除了明确指定了数组的类型外，还未每一个数组元素指定了初始值。

数组是一个复合类型，因此输出数组的时候需要使用 **{:?}** 格式符。

Rust 还提供了 len() 方法则用于返回数组的长度，也就是元素的格式。

```
fn main(){
   let arr:[i32;4] = [10,20,30,40];
   println!("array is {:?}",arr);
   println!("array size is :{}",arr.len());
}
```

编译运行以上 Rust 代码，输出结果如下

```
array is [10, 20, 30, 40]
array size is :4
```

13.2.2 数组初始化：忽略元素数据类型

数组初始化时如果为每一个元素都指定了它的初始值，那么在定义数组时可以忽略数组的数据类型。

因为这时候，编译器可以通过元素的数据类型自动推断出数组的数据类型。

例如下面的代码，我们的数组长度为 4，因为初始化的时候为 4 个元素都指定了初始值为整型，那么声明数组变量的时候就可以忽略数组的数据类型。

数组的 len() 函数用于返回数组的长度。

```
fn main(){
   let arr = [10,20,30,40];
   println!("array is {:?}",arr);
   println!("array size is :{}",arr.len());
}
```

编译运行以上 Rust 范例，输出结果如下

```
array is [10, 20, 30, 40]
array size is :4
```

13.3 数组默认值

在数组初始化时，如果不想为数组中的每个元素指定值，我们可以为数组设置一个默认值，也就是使用默认值初始化 语法。

当使用 默认值初始化 语法初始化数组时，数组中的每一个元素都被设置为默认值。

例如下面的代码，我们将数组中所有的元素的值初始化为 -1

```
fn main() {
   let arr:[i32;4] = [-1;4];
   println!("array is {:?}",arr);
   println!("array size is :{}",arr.len());
}
```

编译运行以上 Rust 代码，输出结果如下

```
array is [-1, -1, -1, -1]
array size is :4
```

13.4 数组长度 len()

Rust 为数组提供了 **len()** 方法用于返回数组的长度。

len() 方法的返回值是一个整型。

例如下面的代码，我们使用 len() 求数组的长度

```
fn main() {
   let arr:[i32;4] = [-1;4];
   println!("array size is :{}",arr.len());
}
```

编译运行以上 Rust 代码，输出结果如下

```
array size is :4
```

13.5 for in 循环遍历数组

在其它语言中，一般使用 for 循环来遍历数组，Rust 语言也可以，只不过时使用 for 语句的变种 for ... in .. 语句。

因为数组的长度在编译时就时已知的，因此我们可以使用 for ... in 语句来遍历数组。

注意 for in 语法中的左闭右开法则。

```
fn main(){
   let arr:[i32;4] = [10,20,30,40];
   println!("array is {:?}",arr);
   println!("array size is :{}",arr.len());

   for index in 0..4 {
      println!("index is: {} & value is : {}",index,arr[index]);
   }
}
```

编译运行以上 Rust 代码，输出结果如下

```
array is [10, 20, 30, 40]
array size is :4
index is: 0 & value is : 10
index is: 1 & value is : 20
index is: 2 & value is : 30
index is: 3 & value is : 40
```

13.6 迭代数组 iter()

我们可以使用 **iter()** 函数为数组生成一个迭代器。

然后就可以使用 for in 语法来迭代数组。

```
fn main(){

   let arr:[i32;4] = [10,20,30,40];
   println!("array is {:?}",arr);
   println!("array size is :{}",arr.len());

   for val in arr.iter(){
      println!("value is :{}",val);
   }
}
```

编译运行以上 Rust 代码，输出结果如下

```
array is [10, 20, 30, 40]
array size is :4
value is :10
value is :20
value is :30
value is :40
```

13.7 可变数组

使用 let 声明的变量，默认是只读的，数组也不例外。也就是说，默认情况下，数组是不可变的。

```
fn main(){
   let arr:[i32;4] = [10,20,30,40];
   arr[1] = 0;
   println!("{:?}",arr);
}
```

上面的代码运行会出错，错误信息如下

```
error[E0594]: cannot assign to `arr[_]`, as `arr` is not declared as mutable
 --> src/main.rs:3:4
  |
2 |    let arr:[i32;4] = [10,20,30,40];
  |        --- help: consider changing this to be mutable: `mut arr`
3 |    arr[1] = 0;
  |    ^^^^^^^^^^ cannot assign

error: aborting due to previous error
```

数组的不可变，表现为两种形式：变量不可重新赋值为其它数组、数组的元素不可以修改。

如果要让数组的元素可以修改，就需要添加 mut 关键字。例如

```
let mut arr:[i32;4] = [10,20,30,40];
```

我们将刚刚错误的代码修改下

```
fn main(){
   let mut arr:[i32;4] = [10,20,30,40];
   arr[1] = 0;
   println!("{:?}",arr);
}
```

就可以正常运行，输出结果如下

```
[10, 0, 30, 40]
```

13.8 数组作为函数参数

数组可以作为函数的参数。而传递方式有 **传值传递** 和 **引用传递** 两种方式。

传值传递 就是传递数组的一个副本给函数做参数，函数对副本的任何修改都不会影响到原来的数组。

引用传递 就是传递数组在内存上的位置给函数做参数，因此函数对数组的任何修改都会影响到原来的数组。

13.8.1 范例1：传值传递

下面的代码，我们使用传值方式将数组传递给函数做参数。函数对参数的任何修改都不会影响到原来的数组。

```
fn main() {
   let arr = [10,20,30];
   update(arr);

   println!("Inside main {:?}",arr);
}
fn update(mut arr:[i32;3]){
   for i in 0..3 {
      arr[i] = 0;
   }
   println!("Inside update {:?}",arr);
}
```

编译运行以上 Rust 代码，输出结果如下

```
Inside update [0, 0, 0]
Inside main [10, 20, 30]
```

13.8.2 范例2: 引用传递

下面的代码，我们使用引用方式将数组传递给函数做参数。函数对参数的任何修改都会影响到原来的数组

```
fn main() {
   let mut arr = [10,20,30];
   update(&mut arr);
   println!("Inside main {:?}",arr);
}
fn update(arr:&mut [i32;3]){
   for i in 0..3 {
      arr[i] = 0;
   }
   println!("Inside update {:?}",arr);
}
```

编译运行以上 Rust 代码，输出结果如下

```
Inside update [0, 0, 0]
Inside main [0, 0, 0]
```

13.9 数组声明和常量

数组 (array) 有一个唯一的弱点，它的长度必须在编译时就是固定的已知的。

声明数组时长度必须指定为整数字面量或者整数常量。

如果数组长度是一个变量，则会报编译错误。例如下面的代码

```
fn main() {
   let N: usize = 20;
   let arr = [0; N]; //错误: non-constant used with constant
   print!("{}",arr[10])
}
```

编译上面的 Rust 代码报错

```
error[E0435]: attempt to use a non-constant value in a constant
 --> main.rs:3:18
  |
3 |    let arr = [0; N]; //错误: non-constant used with constant
  |                  ^ non-constant value

error: aborting due to previous error

For more information about this error, try `rustc --explain E0435`.
```

报错的原因是： N 不是一个常量。

注意，虽然 N 默认是只读的，但它仍然是一个变量，只不过是一个只读变量而已，只读变量不是常量。因为变量的值是在运行时确定的，而常量的值是在编译器确定的。

变量不可用做数组的长度。

如果我们将 let 关键字修改为 const 关键字，编译就能通过了。

```
fn main() {
   const N: usize = 20;
   // 固定大小
   let arr = [0; N];

   print!("{}",arr[10])
}
```

编译运行以上 Rust 代码，输出结果如下

```
0
```

usize 是一个指针所占用的大小。它的实际大小取决于你编译程序的 cpu 体系结构。

十四、 Rust 所有权 Ownership

编程语言把内存分为两大类：

- 栈 stack

- 堆 heap

当然了，这两种分类并没有对实际的内存做什么，只是把系统分给应用程序的内存标识为上面的两大类而已。

14.1 栈 stack

栈 stack 是一种 后进先出 容器。就像我们的存储罐子，后面放进去的只能先拿出来（后面放进去的会放在上面）。

栈 stack 上存储的元素大小必须是已知的，也就是说如果一个变量或数据要放到栈上，那么它的大小在编译是就必须是明确的。

例如，对于一个数据类型为 i32 的变量，它的大小是可预知的，只占用 4 个字节。

Rust 语言中所有的标量类型都可以存储到栈上，因为它们的大小都是固定的。

而对于字符串这种复合类型，它们在运行时才会赋值，那么在编译时的大小就是未知的。那么它们就不能存储在栈上，而只能存储在 **堆** 上。

14.2 堆 heap

堆 heap 用于存储那些在编译时大小未知的数据，也就是那些只有在运行时才能确定大小的数据。

我们一般在堆 heap 上存储那些动态类型的数据。简而言之，我们一般在堆上存储那些可能在程序的整个生命周期中发生变化的数据。

堆 是不受系统管理的，由用户自己管理，因此，使用不当，内存溢出的可能性就大大增加了。

14.3 什么是所有权？

所有权就是一个东西属不属于你，你有没有权力随意处理它，比如送人，比如扔掉。

Rust 语言中每一值都有一个对应的变量，这个变量就成为这个值的 所有者。从某些方面说，定义一个变量就是为这个变量和它存储的数据定义一种所有者管理，声明这个值由这个变量所有。

例如，对于 let age = 30 这条语句，相当于声明 30 这个值由变量 age 所有。

这个比喻是不恰当的。变量并不是对 30 这个数字拥有，而是某个内存块的所有者，而这个内存块上存储者 30 这个数。

任何东西只有一个所有者，Rust 中是不允许有共同所有者这个概念的。

Rust 中，任何特定时刻，一个数据只能有一个所有者。

Rust 中，不允许两个变量同时指向同一块内存区域。变量必须指向不同的内存区域。

14.3.1 转让所有权

既然所有权就是一个东西属不属于你，你有没有权力随意处理它，比如送人，比如扔掉。

那么转让所有权就会时不时的发生。

Rust 语言中转让所有权的方式有以下几种：

把一个变量赋值给另一个变量。重要

把变量传递给函数作为参数。

函数中返回一个变量作为返回值。

接下来我们分别对这三种方式做详细的介绍

14.3.2 把一个变量赋值给另一个变量

Rust 自己宣称的最大卖点是它的 内存安全，这也是它认为能够取代 C++ 作为系统级别语言的自信之一。

Rust 为了实现内存安全，Rust 严格控制谁可以使用内存和什么时候应该限制使用内存。

说起来有点晦涩难懂，我们直接看代码，然后通过代码来解释

```
fn main(){

    // 向量 v 拥有堆上数据的所有权
    // 每次只能有一个变量对堆上的数据拥有所有权
    let v = vec![1,2,3];



    // 赋值会导致两个变量都对同一个数据拥有所有权
    // 因为两个变量指向了相同的内存块
    let v2 = v;

    // Rust 会检查两个变量是否同时拥有堆上内存块的所有权。
    // 如果发生所有权竞争，它会自动将所有权判给给新的变量
    // 运行出错，因为 v 不再拥有数据的所有权
    println!("{:?}",v);
}
```

上面的代码中我们首先声明了一个向量 v。所有权的概念是只有一个变量绑定到资源，v 绑定到资源或 v2 绑定到资源。

上面的代码会发生编译错误 use of moved value: v。这是因为赋值操作会将资源的所有权转移到了 v2。这意味着所有权从 v 移至 v2 （ v2 = v ），移动后 v 就会变得无效。

14.3.3 把变量传递给函数作为参数

将堆中的对象传递给闭包或函数时，值的所有权也会发生变更

```
fn main(){
   let v = vec![1,2,3];     // 向量 v 拥有堆上数据的所有权
   let v2 = v;              // 向量 v 将所有权转让给 v2
   display(v2);             // v2 将所有权转让给函数参数 v , v2 将变得不可用
   println!("In main {:?}",v2);    // v2 变得不可用
}
fn display(v:Vec<i32>){
   println!("inside display {:?}",v);
}
```

编译运行以上 Rust 代码，报错如下

```
error[E0382]: borrow of moved value: `v2`
 --> src/main.rs:5:28
  |
3 |    let v2 = v;                // 向量 v 将所有权转让给 v2
  |        -- move occurs because `v2` has type `std::vec::Vec<i32>`, which does not
implement the `Copy` trait
4 |    display(v2);
  |            -- value moved here
5 |    println!("In main {:?}",v2);
  |                            ^^ value borrowed here after move
```

修复的关键，就是注释掉最后的输出 v2

```
fn main(){
   let v = vec![1,2,3];     // 向量 v 拥有堆上数据的所有权
   let v2 = v;              // 向量 v 将所有权转让给 v2
   display(v2);             // v2 将所有权转让给函数参数 v , v2 将变得不可用
   //println!("In main {:?}",v2);    // v2 变得不可用
}

fn display(v:Vec<i32>){
   println!("inside display {:?}",v);
}
```

编译运行以上 Rust 代码，报错如下

```
inside display [1, 2, 3]
```

14.3.4 函数中返回一个变量作为返回值

传递给函数的所有权将在函数执行完成时失效。

也就是函数的形参获得的所有权将在离开函数后就失效了。失效了数据就再也访问不到的了。

为了解决所有权失效的问题，我们可以让函数将拥有的对象返回给调用者。

```
fn main(){
   let v = vec![1,2,3];       // 向量 v 拥有堆上数据的所有权
   let v2 = v;                // 向量 v 将所有权转让给 v2
   let v2_return = display(v2);
   println!("In main {:?}",v2_return);
}

fn display(v:Vec<i32>)-> Vec<i32> {
   // 返回同一个向量
   println!("inside display {:?}",v);
   return v;
}
```

编译运行上面的 Rust 代码，输出结果如下

```
inside display [1, 2, 3]
In main [1, 2, 3]
```

14.4 所有权和基本（原始）数据类型

所有的基本数据类型，把一个变量赋值给另一个变量，并不是所有权转让，而是把数据复制给另一个对象。简单的说，就是在内存上重新开辟一个区域，存储复制来的数据，然后把新的变量指向它。

这样做的原因，是因为原始数据类型并不需要占用那么大的内存。

```
fn main(){
   let u1 = 10;
   let u2 = u1;  // u1 只是将数据拷贝给 u2

   println!("u1 = {}",u1);
}
```

编译运行以上 Rust 代码，输出结果如下

```
u1 = 10
```

> 注意：所有权只会发生在堆上分配的数据，对比 C++，可以说所有权只会发生在指针上。基本类型的存储都在栈上，因此没有所有权的概念。

十五、Rust 借用 Borrowing

上一章节我们学习了 **所有权 （ ownership ）** 这个改变，知道了在 **堆（ heap ）** 上分配的变量都有所有权。

上一章节，我们也以一种艰难的方式将具有所有权的变量，比如字符串变量、向量变量作为参数传递给函数，同时为了保证函数调用之后变量仍然具有所有权，又在函数内返回变量。

这样的过程，不但没有减轻我们的负担，反而觉得越来越难以使用的感觉。

使用的过程中，我就一直在想，为什么不多支持一个 **借用所有权** 或者 **租借所有权** 的概念呢？

把具有所有权的变量传递给函数作为参数时，就是临时出租所有权，当函数执行完后就会自动收回所有权。就像现实生活中，我可以把某个工具临时借用给其它人，当他们使用完了之后还给我们就可以了。

随着对 Rust 的深入了解，觉得 Rust 语言的开发者也是不笨的，他们也想到了 借用所有权 这个概念。

Rust 支持对所有权的 **出借 borrowing**。当把一个具有所有权的变量传递给函数时，就是把所有权借用给函数的参数，当函数返回后则自动收回所有权。

下面的代码，我们并没有使用上一章节的 所有权 转让规则收回所有权，所以程序会报错

```
fn main(){

    let v = vec![10,20,30]; // 声明一个向量，变量 v 具有数据的所有权
    print_vector(v);
    println!("{}",v[0]);    // 这行会报错
}

fn print_vector(x:Vec<i32>){
    println!("Inside print_vector function {:?}",x);
}
```

上面这段代码中，我们定义了两个函数 main() 和 print_vector() ，前者是应用程序的入口函数，而后者则用于输出一个 向量。

我们在 main() 函数中定义了一个向量，同时将这个向量传递给 print_vector() 作为参数。 因为参数的传递会触发所有权的转让。因此将 v 传递给 print_vector() 函数时，数据的所有权就从 v 转让到了 参数 x 上。

但函数返回时我们并没有把 x 对数据的所有权转让回 v 变量，因此上面这段代码编译的时候编译的时候就会报错了。

```
error[E0382]: borrow of moved value: `v`
 --> src/main.rs:5:18
  |
3 |    let v = vec![10,20,30]; // 声明一个向量，变量 v 具有数据的所有权
  |        - move occurs because `v` has type `std::vec::Vec<i32>`, which does not
implement the `Copy` trait
4 |    print_vector(v);
  |                 - value moved here
5 |    println!("{}",v[0]);    // 这行会报错
  |                   ^ value borrowed here after move

error: aborting due to previous error
```

重复的讲解这个例子，并不是为了凑字数，而是我们会有一种更好的解决方案，这个方案只要修改一点点就能让程序运行。

15.1 什么是借用 Borrowing？

借用 Borrowing 或者说 出借 应该不用我再详细解释了吧，很简单的，就是 临时性的把东西借给别人，当别人用完了之后就要还回来。

这里的重点有两个字： 借 和 还。

- 借： 把东西借给他人后，自己就暂时性的失去了东西的所有权（现实中是失去了使用权）。
- 还： 借了别人的东西要主动还，这应该养成一个良好的习惯，如果不还，就是 占为己有 了。

了解了 借用、借、还 的概念后，对 Rust 语言的 **借用 Borrowing** 概念就很清晰了。

Rust 语言中，借用 就是一个函数中将一个变量传递给另一个函数作为参数暂时使用。

同时，Rust 也引用了自动 还 的概念，就是要求函数的参数离开其作用域时需要将 所有权 还给当初传递给他的变量，这个过程，我们需要将函数的参数定义为 &variable_name，同时传递参数时，需要传递 &variable_name。

站在 C++ 语言的角度考虑，就是将 函数的参数定义为引用，同时传递变量的引用。

有了 借用 Borrowing 也就是引用的概念后，我们只要修改两处就能让上面的代码运行起来

```
fn print_vector(x:&Vec<i32>){  // 1. 第一步，定义参数接受一个引用
    println!("Inside print_vector function {:?}",x);
}

fn main(){

    let v = vec![10,20,30];     // 声明一个向量，变量 v 具有数据的所有权
    print_vector(&v);           // 第二步，传递变量的引用给函数
    println!("{}",v[0]);        // 这行会报错
}
```

编译运行以上 Rust 代码，输出结果如下

```
Inside print_vector function [10, 20, 30]
Printing the value from main() v[0] = 10
```

15.2 可变引用

借用 Borrowing 或者说引用默认情况下是只读的，也就是我们不能修改引用的的变量的值。

例如下面的代码

```
fn add_one(e: &i32) {
   *e+= 1;
}

fn main() {
   let i = 3;
   println!("before {}",i);
   add_one(&i);
   println!("after {}", i);
}
```

编译运行以上 Rust 代码，输出结果如下

```
error[E0594]: cannot assign to `*e` which is behind a `&` reference
 --> src/main.rs:2:4
  |
1 | fn add_one(e: &i32) {
  |               ---- help: consider changing this to be a mutable reference: `&mut
i32`
2 |    *e+= 1;
  |    ^^^^^^ `e` is a `&` reference, so the data it refers to cannot be written

error: aborting due to previous error
```

我们尝试在函数 add_one() 将引用的变量 +1 但却编译失败了。

而失败的原因，就像错误信息里说的那样，引用 默认情况下是不可编辑的。

Rust 中，要让一个变量可编辑，唯一的方式就是给他加上 mut 关键字。

因此，我们可以将上面的代码改造下，改成下面这样

```
fn add_one(e: &mut i32) {
   *e+= 1;
}

fn main() {
   let mut i = 3;
   println!("before {}",i);
   add_one(&mut i);
   println!("after {}", i);
}
```

编译运行以上 Rust 代码，输出结果如下

```
before 3
after 4
```

从上面的代码中可以看出： 借用 Borrowing 或者说引用的变量如果要变更，必须符合满足三个要求：

1. 变量本身是可变更的，也就是定义时必须添加 mut 关键字。
2. 函数的参数也必须定义为可变更的，加上 借用 Borrowing 或者说引用，也就是必须添加 &mut 关键字。
3. 传递 借用 Borrowing 或者说引用也必须声明时 可变更传递，也就是传递参数时必须添加 &mut 关键字。

以上三个条件，任意一个不满足，都会报错。比如第三个条件不满足时

```
fn add_one(e: &mut i32) {
   *e+= 1;
}

fn main() {
   let mut i = 3;
   println!("before {}",i);
   add_one(& i);
   println!("after {}", i);
}
```

报错信息如下

```
error[E0308]: mismatched types
 --> src/main.rs:8:12
  |
8 |    add_one(& i);
  |            ^^^ types differ in mutability
  |
  = note: expected type `&mut i32`
             found type `&{integer}`

error: aborting due to previous error
```

> 注意: 可变引用只能操作可变变量

15.2.1 范例：字符串的可变引用

上面的范例，我们操作的是基础的数据类型，如果是堆上分配的变量又会怎么样呢？ 比如字符串。

其实堆上分配的变量的 借用 Borrowing 或者说引用跟基础类型的变量一样，我们来看一段代码

```
fn main() {
   let mut name:String = String::from("TutorialsPoint");
   display(&mut name);  // 传递一个可变引用
   println!("The value of name after modification is:{}",name);
}

fn display(param_name:&mut String){
   println!("param_name value is :{}",param_name);
   param_name.push_str(" Rocks"); // 修改字符串，追加一些字符
}
```

编译运行以上 Rust 代码，输出结果如下

```
param_name value is :TutorialsPoint
The value of name after modification is:TutorialsPoint Rocks
```

十六、Rust 切片 Slice

一个 **切片（slice）** 就是指向一段内存的指针。

因此切片可用于访问内存块中连续区间内的数据。

一般情况下，能够在内存中连续区间存储数据的数据结构有: 数组 array、向量 vector、字符串 string。

也就是说，**切片** 可以和数组、向量、字符串一起使用，它使用 数字索引 （类似于数组的下标索引）来访问它所指向的数据。

例如，**切片** 可以和字符串一起使用，切片 可以指向字符串中连续的一部分。这种 字符串切片 实际上是指向字符串的指针。因为是指向字符串的连续区间，所以我们要指定字符串的开始和结束位置。

访问切片内容的时候，下标索引是从 0 开始的。

切片 的大小是运行时才可知的，并不是数组那种编译时就必须告知的。

16.1 定义切片的语法

经过上面晦涩难懂的解释，我们知道切片就是指向数组、向量、字符串的连续区间。

定义一个切片的语法格式如下

```
let sliced_value = &data_structure[start_index..end_index]
```

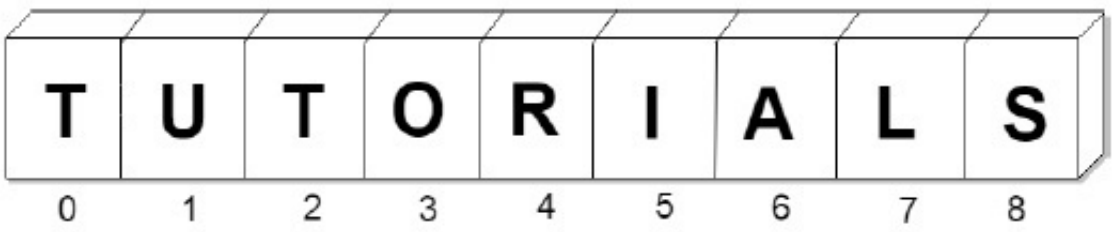
定义切片时的几个注意事项:

1. [start_index..end_index] 是一个左闭又开区间 [start_index,end_index)
2. 要注意区间的语法两个点 .. 。
3. start_index 的最小值为 0，这是因为数组、向量、字符串它们的下标访问方式也是从 0 开始的。
4. end_index 的最大值是数组、向量、字符串的长度。

5. end_index 所表示的索引的字符并不包含在切片里面。

16.2 切片图例

数组、向量、字符串在内存中是连续存储的，一个字符串 Tutorials 在内存中的存储类似于下图



从图中很直观的可以看出，字符串 Tutorials 的长度为 9，第一个字符的下标索引为 0，最后一个字符的下标索引为 8。

如果我们想访问字符串 Tutorials 中第 4 个字符开始的连续 5 个字符，使用切片，我们可以这么做

```
fn main() {
   let n1 = "Tutorials".to_string();
   println!("length of string is {}",n1.len());
   let c1 = &n1[4..9];

   // fetches characters at 4,5,6,7, and 8 indexes
   println!("{}",c1);
}
```

编译运行以上 Rust 代码，输出结果如下

```
length of string is 9
rials
```

16.3 切片作为函数参数

切片还可以作为函数的参数。

使用切片可以把数组、向量、字符串中的连续子集通过引用的方式传递给函数。

我们先来看一段简单的代码

```
fn main(){
   let data = [10,20,30,40,50];
   use_slice(&data[1..4]);
   //this is effectively borrowing elements for a while
}
fn use_slice(slice:&[i32]) {
   // is taking a slice or borrowing a part of an array of i32s
   println!("length of slice is {:?}",slice.len());
   println!("{:?}",slice);
}
```

编译运行以上 Rust 代码，输出结果如下

```
length of slice is 3
[20, 30, 40]
```

上面这段代码中

1. 声明了两个函数 main() 和 use_slice()，后者接受一个切片并打印切片的长度。
2. 首先在 main() 函数中声明了一个有 5 个元素的数组。
3. 然后调用函数 use_slice() 并把数组的一个切片（ 从下标 1 开始到下标 3 之间的元素 ）作为参数。

16.4 可变更切片

默认情况下 **切片** 是不可变更的。

虽然，看起来切片是指向原数据，但是默认情况下我们并不能改变切片的元素。

也就说默认情况下不能通过更改切片的元素来影响原数据。

但这不是绝对的，如果我们声明的原数据是可变的，同时定义切片的时候添加了 &mut 关键字，那么我们就可以通过更改切片的元素来影响原数据。

```
fn main(){
   let mut data = [10,20,30,40,50];
   use_slice(&mut data[1..4]);
   // passes references of
   20, 30 and 40
   println!("{:?}",data);
}
fn use_slice(slice:&mut [i32]) {
   println!("切片的长度为：{:?}",slice.len());
   println!("{:?}",slice);
   slice[0] = 1010; // replaces 20 with 1010
}
```

编译运行以上 Rust 代码，输出结果如下

```
切片的长度为：3
[20, 30, 40]
[10, 1010, 30, 40, 50]
```

从上面的代码中可以看出，只要原数据是可变的，且切片声明时添加了 &mut 关键字，那么切片就是可变更的。

十七、Rust 结构体 Struct

数组永远只能保存相同类型的元素。从某些方面说，数组是相同类型元素的集合。

但世界并不是那么美好的，永远都是复制克隆类的东西，很多东西往往比较复杂。

比如我们要描述一个人，它有着年龄，有着姓名，有着居住地址。如果我们要存储这些东西，数组是指望不上了。

为了解决这里问题，语言的开发者们想到了另一种可以让用户自定义类型的方法，那就是 **结构体（struct）**。

结构体，其实就是 **结构**，日常生活中，当我们提及某某结构的时候，都会分析说它是由什么什么组成的。比如当我们分析一张 桌子 的时候，我们会说它有 4 条腿，有一个桌面，有几个横杠...

结构体 就是可以组合不同类型的数据项，包括另一个结构。

17.1 定义一个结构体

几乎所有有结构体这个概念的语言，定义结构体的关键字都是一样的，那就是struct。

因为结构体是一个集合，也就是复合类型。结构体中的所有元素/字段也必须明确指明它的数据类型。

定义一个结构体的语法格式如下

```
struct Name_of_structure {
   field1:data_type,
   field2:data_type,
   field3:data_type
}
```

定义一个结构体时：

- 结构体名 Name_of_structure 和元素/字段名 fieldN 遵循普通变量的命名语法。
- 结构体中中的每一个元素/字段都必须明确指定数据类型。可以是基本类型，也可以是另一个结构体。

17.1.1 范例

下面的代码，我们定义了一个结构体 Employee ，它有着三个元素/字段，分别是姓名、年龄、公司。

```
struct Employee {
   name:String,
   company:String,
   age:u32
}
```

17.2 创建结构体的实例（也称为结构体初始化）

创建结构体的实例或者说结构体初始化本质上就是创建一个变量。使用 let 关键字创建一个变量。

创建结构体的一个实例和定义结构体的语法真的很类似。但说起来还是有点复杂，我们先看具体的语法

17.2.1 结构体初始化语法

```
let instance_name = Name_of_structure {
   field1:value1,
   field2:value2,
   field3:value3
};
```

从语法中可以看出，初始化结构体时的等号右边，就是把定义语法中的元素类型换成了具体的值。

结构体初始化，其实就是对结构体中的各个元素进行赋值。

> 注意: 千万不要忘记结尾的分号 ;

17.2.2 访问结构体实例元素的语法

如果要访问结构体实例的某个元素，我们可以使用 元素访问符，也就是 点号 (.)

具体的访问语法格式如下

```
struct_name_instance.field_name
```

例如，如果要访问 Employee 的实例 emp1 中的 name 元素，可以如下使用

```
emp1.name
```

17.2.3 范例

下面的代码，我们定义了一个有三个元素的结构体 Employee，然后初始化一个实例 emp1，最后通过元素访问符来访问 emp1 的三个元素。

```
struct Employee {
   name:String,
   company:String,
   age:u32
}
```

```
fn main() {
   let emp1 = Employee {
      company:String::from("TutorialsPoint"),
      name:String::from("Mohtashim"),
      age:50
   };
   println!("Name is :{} company is {} age is {}",emp1.name,emp1.company,emp1.age);
}
```

编译运行以上 Rust 代码，输出结果如下

```
Name is :Mohtashim company is TutorialsPoint age is 50
```

17.3 修改结构体实例

修改结构体实例就是对结构体的个别元素 **重新赋值**。

结构体实例默认是 **不可修改的**，因为结构体实例也是一个使用 let 定义的变量。

如果要修改结构体实例，就必须在创建时添加 mut 关键字，让它变成可修改的。

因为没啥新的知识内容，我们就直接上范例吧。

17.3.1 范例

下面的范例给 Employee 的实例 emp1 添加了 mut 关键字，因此我们可以修改 emp1 的内部元素。

```
struct Employee {
   name:String,
   company:String,
   age:u32
}

let mut emp1 = Employee {
   company:String::from("TutorialsPoint"),
   name:String::from("Mohtashim"),
   age:50
};
emp1.age = 40;
println!("Name is :{} company is {} age is
{}",emp1.name,emp1.company,emp1.age);
```

编译运行以上 Rust 代码，输出结果如下

```
Name is :Mohtashim company is TutorialsPoint age is 40
```

17.4 结构体作为函数的参数

结构体的用途之一就是可以作为参数传递给函数。

定一个结构体参数和定义其它类型的参数的语法是一样的。我们这里就不多介绍了，直接看范例

下面的代码定义了一个函数 display ， 它接受一个 Employee 结构体实例作为参数并输出结构体的所有元素

```
fn display( emp:Employee) {
   println!("Name is :{} company is {} age is
   {}",emp.name,emp.company,emp.age);
}
```

完整的可运行的实例代码如下

```
//定义一个结构体
struct Employee {
   name:String,
   company:String,
   age:u32
}
fn main() {
   //初始化结构体
   let emp1 = Employee {
      company:String::from("TutorialsPoint"),
      name:String::from("Mohtashim"),
      age:50
   };
   let emp2 = Employee{
      company:String::from("TutorialsPoint"),
      name:String::from("Kannan"),
      age:32
   };
   //将结构体作为参数传递给 display
   display(emp1);
   display(emp2);
}

// 使用点号(.) 访问符访问结构体的元素并输出它么的值
fn display( emp:Employee){
   println!("Name is :{} company is {} age is
   {}",emp.name,emp.company,emp.age);
}
```

编译运行以上 Rust 代码，输出结果如下

```
Name is :Mohtashim company is TutorialsPoint age is 50
Name is :Kannan company is TutorialsPoint age is 32
```

17.5 结构体实例作为函数的返回值

Rust 中的结构体不仅仅可以作为函数的参数，还可以作为 函数的返回值。

函数返回结构体实例需要实现两个地方：

1. 在 箭头 -> 后面指定结构体作为一个返回参数。
2. 在函数的内部返回 结构体的实例

17.5.1 结构体实例作为函数的返回值的语法格式

```
struct My_struct {}

fn function_name([parameters]) -> My_struct {
   // 其它的函数逻辑
   return My_struct_instance;
}
```

17.5.2 范例

下面的代码，我们首先定义一个结构体 Employee ， 接着定义一个方法 who_is_elder ， 传入两个结构体 Employee 作为参数并返回年龄个大的那个。

```
fn main() {

   let emp1 = Employee{
      company:String::from("TutorialsPoint"),
      name:String::from("Mohtashim"),
      age:50
   };
   let emp2 = Employee {
      company:String::from("TutorialsPoint"),
      name:String::from("Kannan"),
      age:32
   };
   let elder = who_is_elder(emp1,emp2);
   println!("elder is:");

   display(elder);
}

//接受两个 Employee 的实例作为参数并返回年长的那个
fn who_is_elder (emp1:Employee,emp2:Employee)->Employee {
   if emp1.age>emp2.age {
      return emp1;
   } else {
      return emp2;
   }
```

```
}

// 显示结构体的所有元素
fn display( emp:Employee) {
   println!("Name is :{} company is {} age is {}",emp.name,emp.company,emp.age);
}

// 定义一个结构体
struct Employee {
   name:String,
   company:String,
   age:u32
}
```

编译运行以上 Rust 代码，输出结果如下

```
elder is:
Name is :Mohtashim company is TutorialsPoint age is 50
```

17.6 结构体中的方法

Rust 中的结构体可以定义方法 (method)。

方法 (method) 是一段代码的逻辑组合，用于完成某项特定的任务或实现某项特定的功能。

方法（method） 和 函数（function) 有什么不同之处呢?

简单的说：

1. 函数（ function） 没有属主，也就是归属于谁，因此可以直接调用。
2. 方法（ method ） 是有属主的，调用的时候必须指定 属主。
3. 函数（ function） 没有属主，同一个程序不可以出现两个相同签名的函数。
4. 方法（ method ） 有属主，不同的属主可以有着相同签名的方法。

定义方法时需要使用 fn 关键字。

> fn 关键字是 function 取头尾两个字母的缩写。

结构体方法的 **作用域** 仅限于 **结构体内部**。

与 C++ 语言中的结构体的方法不同的是，Rust 中的结构体方法只能定义在结构体的外面。

在定义结构体方法时需要使用 impl 关键字，语法格式如下

```
struct My_struct {}

impl My_struct {
   // 属于结构体的所有其它代码
}
```

impl 关键字最重要的作用，就是定义上面我们所说的方法的属主。所有被 impl My_struct 块包含的代码，都只属于 My_struct 这个结构。

impl 关键字是 implement 的前 4 个字母的缩写。意思是 **实现**。

结构体的普通方法（后面我们还会学到其它方法）时，第一个参数永远是 &self 关键字。self 是"自我"的意思，&self 永远表示着当前的结构体的一个实例。

这是不是可以带来其它的结构体方法的解释：结构体的方法就是用来操作当前结构体的一个实例的。

17.6.1 结构体方法的定义语法

定义结构体方法的语法格式如下

```
struct My_struct {}
impl My_struct {

   // 定义一个结构体的普通方法
   fn method_name(&self[,other_parameters]) {
      //方法的具体逻辑代码
   }

}
```

&self 是结构体普通方法固定的第一个参数，其它参数则是可选的。

即使结构体方法不需要传递任何参数，&self 也是固定的，必须存在的。像下面这种定义方法是错误的

```
struct My_struct {}
impl My_struct {

   //定义一个结构体的普通方法
   fn method_name([other_parameters]) {
      //方法的具体逻辑代码
   }
}
```

17.6.2 结构体方法内部访问结构体元素

因为我们在定义方法时固定传递了 &self 关键字。而 &self 关键字又代表了当前方法的属主。

因此我们可以在方法内部使用 self, 来访问结构体的元素。

详细的语法格式如下

```
struct My_struct {
   age: u32
}

impl My_struct {
```

```
    //定义一个结构体的普通方法
    fn method_name([other_parameters]) {

        self.age = 28;

        println!("{}",self.age);

        //其它的具体逻辑代码
    }
}
```

17.6.3 结构体方法的调用语法

因为结构体的方法是有属主的，所以调用的时候必须先指定 **属主**，调用格式为 **属主.方法名(方法参数)**。

详细的调用语法格式为

```
My_struct.method_name([other_parameters])
```

> 注意: 虽然定义方法时需要固定 &self 作为第一个参数，但在调用的时候是 不需要也不能 传递的。这个参数的传递 Rust 编译器会 偷偷的 帮我们完成。

17.6.4 范例

下面的代码，我们首先定义了一个结构体 Rectangle 用于表示一个 长方形，它有宽高 两个元素 width 和 height。

然后我们又为 结构体 Rectangle 定义了一个方法 area 用于计算当前 长方形实例 的面积。

```
// 定义一个长方形结构体
struct Rectangle {
    width:u32, height:u32
}

// 为长方形结构体定义一个方法，用于计算当前长方形的面积
impl Rectangle {
    fn area(&self)->u32 {
        // 在方法内部，可以使用点号 `self.` 来访问当前结构体的元素。use the . operator to
fetch the value of a field via the self keyword
        self.width * self.height
    }
}

fn main() {
    // 创建 Rectangle 结构体的一个实例
    let small = Rectangle {
        width:10,
```

```
        height:20
    };

    //计算并输出结构体的面积
    println!("width is {} height is {} area of Rectangle
    is {}",small.width,small.height,small.area());
}
```

编译运行以上 Rust 代码，输出结果如下

```
width is 10 height is 20 area of Rectangle is 200
```

17.7 结构体的静态方法

Rust 中的结构体还可以有静态方法。

静态方法可以直接通过结构体名调用而无需先实例化。

结构体的静态方法定义方式和普通方法类似，唯一的不同点是 **不需要使用 &self** 作为参数。

17.7.1 定义静态方法的语法

结构体静态方法的定义语法格式如下

```
impl Structure_Name {

    // Structure_Name 结构体的静态方法
    fn method_name(param1: datatype, param2: datatype) -> return_type {
        // 方法内部逻辑
    }

}
```

静态方法和其它普通方法一样，参数是可选的。也就是可以没有参数

17.7.2 调用静态方法的语法

静态方法可以直接通过结构体名调用，而无需先实例化。

结构体的静态方法需要使用 structure_name:: 语法来访问，详细的语法格式如下

```
structure_name::method_name(v1,v2)
```

17.3 范例

下面的范例，我们为结构体 Point 定义了一个静态方法 getInstance()。

```
getInstance() 是一个 工厂方法，它初始化并返回结构体 Point 的实例。
```

```
//声明结构体 Point
struct Point {
   x: i32,
   y: i32,
}

impl Point {
   // 用于创建 Point 实例的静态方法
   fn getInstance(x: i32, y: i32) -> Point {
      Point { x: x, y: y }
   }
   // 用于显示结构体元素的普通方法
   fn display(&self){
      println!("x = {} y = {}",self.x,self.y );
   }
}
fn main(){

   // 调用静态方法
   let p1 = Point::getInstance(10,20);
   p1.display();

}
```

编译运行以上 Rust 代码，输出结果如下

```
x = 10 y = 20
```

十八、Rust 枚举 Enum

长大的时候回去想想小时候，觉得有时候特么的太傻逼了，比如下面这道选择题

下面哪个不是水果?

```
    A. 香蕉   B. 梨   C. 橘子  D. 茄子
```

我特么的能选择 A。 原因是其它三个都吃过，可是香蕉，我家那种野香蕉树，香蕉就只有拇指大小，还特别涩，哪能吃啊。

长大了之后，也会觉得过得很苦逼，比如填表格的时候

请问你的婚姻状况是?

```
    A. 未婚  B 已婚   C 离异
```

都填了二十几年的表了，选择永远是 A 。

上面两个问题有啥共同点么？

都是很傻逼....？

哈哈，不是的，它们的共同点都是提供了 ABCD 让我们选择。

编程时我们要如何保存 A B C D 这种选择题和答案呢？

如果只利用我们之前的所学知识，大概是

```
let option_a = "香蕉";
let option_b = "梨";
let option_c = "橘子";
let option_d = "茄子";
```

大家有没有发现什么问题？ 我们为啥要傻傻的定义四个变量啊，用一个数组就搞定了

```
let option = ["香蕉","梨","橘子","茄子"];
```

看起来不错的样子，可另一个问题出现了，一般做选择的时候我们都会回答**香蕉**或**茄子** ，肯定不会回答**0**或**3**。因为出题的人看不懂啊。

这个说法不恰当，但目前没想到更好的。

18.1 枚举

像这种万里挑一的问题，从众多个选项中选择一个的问题，像这种众多选项，Rust 提供了一个新的数据类型用来表示它们。

这个新的数据类型就是 **枚举** ，英文 **enum**。

也就是说，枚举 用于从众多的可变列表中选择一个。

18.2枚举定义

Rust 语言提供了 enum 关键字用于定义枚举。

定义枚举的语法格式如下

```
enum enum_name {
   variant1,
   variant2,
   variant3
}
```

例如上面我们的 香蕉橘子选项，我们可以定义一个枚举 Fruits

```
enum Fruits {
    Banana,     // 香蕉
    Pear,       // 梨
    Mandarin,   // 橘子
    Eggplant    // 茄子
}
```

18.3使用枚举

枚举定义好了之后我们就要开始用它了，枚举的使用方式很简单，就是 枚举名::枚举值。语法格式如下

```
enum_name::variant
```

例如上面的枚举，比如我选择了 香蕉，那么赋值的语法如下

```
let selected = Fruits::Banana;
```

如果需要明确指定类型，可以如下

```
let selected: Fruits = Fruits::Banana;
```

18.3.1 范例

下面的范例，演示了枚举类型的基本使用方法和案例。

我们首先定义了一个枚举 Fruits ，它有四个枚举值，分别是 Banana、Pear、 Mandarin 和 Eggplant。

println!() 用于输出枚举。

> 注意： 关于枚举前面的 #[derive(Debug)] 我们后面会介绍

```
#[derive(Debug)]
enum Fruits {
    Banana,     // 香蕉
    Pear,       // 梨
    Mandarin,   // 橘子
    Eggplant    // 茄子
}

fn main() {
   let selected = Fruits::Banana;
   println!("{:?}",selected);
}
```

编译运行以上 Rust 代码，输出结果如下

```
Banana
```

18.4 #[derive(Debug)] 注解

想必大家看到了 enum Fruits 前面的 #[derive(Debug)]。

这个 #[derive(Debug)] 语句的作用是啥呢？

我先不解释，我先把 #[derive(Debug)] 去掉看看

```
enum Fruits {
    Banana,     // 香蕉
    Pear,       // 梨
    Mandarin,   // 橘子
    Eggplant   // 茄子
}

fn main() {
   let selected = Fruits::Banana;
   println!("{:?}",selected);
}
```

编译上面的 Rust 代码，或报错，错误信息如下

```
error[E0277]: `Fruits` doesn't implement `std::fmt::Debug`
  --> src/main.rs:11:20
   |
11 |    println!("{:?}",selected);
   |                     ^^^^^^^^ `Fruits` cannot be formatted using `{:?}`
   |
   = help: the trait `std::fmt::Debug` is not implemented for `Fruits`
   = note: add `#[derive(Debug)]` or manually implement `std::fmt::Debug`
   = note: required by `std::fmt::Debug::fmt`
```

这段错误的意思，就是我们的 enum Fruits 枚举并没有实现 std::fmt::Debug 特质 （ trait ） 。

> 关于特质 trait 我们会在后面介绍，这里你只要把特质当作接口 interface 看待就好。

为了让编译能通过，我们需要将我们的枚举派生自或衍生自一个已经实现了 std::fmt::Debug 特质的东西。这个东西比较常见的就是 Debug 。

因此 #[derive(Debug)] 注解的作用，就是让 Fruits 派生自 Debug。

但，其实，即使添加了 #[derive(Debug)] 注解注解仍然会有警告

```
#[derive(Debug)]
enum Fruits {
    Banana,     // 香蕉
    Pear,       // 梨
    Mandarin,   // 橘子
    Eggplant   // 茄子
}

fn main() {
   let selected = Fruits::Banana;
   println!("{:?}",selected);
}
```

编译结果如下

```
warning: variant is never constructed: `Pear`
 --> main.rs:4:5
  |
4 |     Pear,       // 梨
  |     ^^^^
  |
  = note: #[warn(dead_code)] on by default

warning: variant is never constructed: `Mandarin`
 --> main.rs:5:5
  |
5 |     Mandarin,   // 橘子
  |     ^^^^^^^^

warning: variant is never constructed: `Eggplant`
 --> main.rs:6:5
  |
6 |     Eggplant   // 茄子
  |     ^^^^^^^^
```

大概的意思是说，那些我们没用到的枚举都还没有被构造呢。

```
variant is never constructed: Eggplant 具体是啥意思，以后有空再来 YY 吧。
```

18.5 结构类型 struct 和枚举类型 enum

好了，到目前为止，我们已经学习了两个可以自定义类型的东西了，

- 一个是 结构类型 struct
- 另一个是 枚举类型 enum

它们之间有什么关系和关联呢?

但是是它们是八杆子都打不着的东西。

如果说有那么一丁点儿关系，那就是 枚举类型 enum 可以作为结构体的成员变量的数据类型。

18.5.1 范例

下面的代码，我们定义了一个枚举 GenderCategory 用于表示性别，枚举值有 Male 和 Female 分别表示男和女。

> 还有第三种性别？ 这里忽略吧。

同时，我们又定义了一个结构体 Person 用于描述一个人。这个 Person 结构体使用 GenderCategory 枚举作为其成员变量 gender 的数据类型

因此，Person 结构体的 gener 成员变量就只能有两个值: Male 和 Female

```
// 添加 #[derive(Debug)] 省的报错
#[derive(Debug)]
enum GenderCategory {
   Male,Female
}

// 添加 #[derive(Debug)] 省的报错
#[derive(Debug)]
struct Person {
   name:String,
   gender:GenderCategory
}

fn main() {
   let p1 = Person {
      name:String::from("零基础教程"),
      gender:GenderCategory::Female
   };
   let p2 = Person {
      name:String::from("Admin"),
      gender:GenderCategory::Male
   };
   println!("{:?}",p1);
   println!("{:?}",p2);
}
```

编译运行以上 Rust 代码，输出结果如下

```
Person { name: "零基础教程", gender: Female }
Person { name: "Admin", gender: Male }
```

18.6 Option 枚举

Rust 语言核心和标准库内置了很多枚举，其中有一个枚举我们会经常和它打交道，那就是 Option 枚举。

Option 枚举代表了那种 可有可无 的选项。它有两个枚举值 None 和 Some(T)。

- None 表示可有可无中的 无。
- Some(T) 表示可有可无中的 有，既然有，那么就一定有值，也就是一定有数据类型，那个 T 就表示有值时的值数据类型。

18.6.1 Option 枚举的定义代码如下

```
enum Option<T> {
   Some(T),      // 用于返回一个值 used to return a value
   None          // 用于返回 null , 虽然 Rust 并不支持 null
   the null keyword
}
```

Rust 语言并不支持 null 关键字，取而代之的是使用 None 作为没有的意思。

Option 枚举经常用在函数中作为返回值，因为它可以表示有返回且有值，也可以用于表示有返回但没有值。

如果函数有返回值，那么可以返回 Some(data)，如果函数没有返回值，则可以返回 None

如果你还不理解，那么就直接看范例吧

18.6.2 范例

下面的范例，我们定义了一个函数 is_even()，使用 Option 枚举作为它的返回值类型。

如果传递给 is_even() 函数的参数是个偶数，则返回传递的参数，如果是奇数则返回 None。

```
fn main() {
   let result = is_even(3);
   println!("{:?}",result);
   println!("{:?}",is_even(30));
}
fn is_even(no:i32)->Option<bool> {
   if no %2 == 0 {
      Some(true)
   } else {
      None
   }
}
```

编译运行以上 Rust 代码，输出结果如下

```
None
Some(true)
```

18.7 match 语句和枚举

枚举的另一个重要操作就是判断枚举值。判断一个枚举值，== 比较运算符是不起作用的。

准确的说会报错

```
#[derive(Debug)]
enum Fruits {
    Banana,     // 香蕉
    Pear,       // 梨
    Mandarin,   // 橘子
    Eggplant   // 茄子
}

fn main() {
    let selected = Fruits::Banana;
    if selected == Fruits::Banana {
        println!("你选择了香蕉");
    } else {
        println!("你选择了其它");
    }
   println!("{:?}",selected);
}
```

编译错误

```
  --> src/main.rs:11:17
   |
11 |     if selected == Fruits::Banana {
   |        -------- ^^ -------------- Fruits
   |        |
   |        Fruits
   |
   = note: an implementation of `std::cmp::PartialEq` might be missing for `Fruits`
```

判断一个枚举变量的值，唯一能用的操作符就是 match 语句。

match 语句我们之间已经学过了，我们就不介绍了。我们直接上范例，看看如何使用 match 语句来判断枚举值

18.7.1 范例

下面的代码，我们定义了一个枚举 CarType，同时定义了一个函数 print_size() ，它接受 CarType 枚举类型的变量，并使用 match 语句来判断枚举变量的值。

```
enum CarType {
   Hatch,
   Sedan,
```

```
   SUV
}
fn print_size(car:CarType) {
   match car {
      CarType::Hatch => {
         println!("Small sized car");
      },
      CarType::Sedan => {
         println!("medium sized car");
      },
      CarType::SUV =>{
         println!("Large sized Sports Utility car");
      }
   }
}
fn main(){
   print_size(CarType::SUV);
   print_size(CarType::Hatch);
   print_size(CarType::Sedan);
}
```

编译运行以上 Rust 代码，输出结果如下

```
Large sized Sports Utility car
Small sized car
medium sized car
```

18.8 match 语句和 Option 类型

既然 match 语句可以用于比较和判断枚举变量的值，那么对于上面提到的 Option 枚举，match 语句也使用。

事实上，match 语句和 Option 类型相结合才能体现出 Option 枚举的强大之处

```
fn main() {
   match is_even(5) {
      Some(data) => {
         if data==true {
            println!("Even no");
         }
      },
      None => {
         println!("not even");
      }
   }
}
fn is_even(no:i32)->Option<bool> {
   if no%2 == 0 {
```

```
      Some(true)
   } else {
      None
   }
}
```

编译运行以上 Rust 代码，输出结果如下

```
not even
```

18.9 match 语句和带数据类型的枚举

Rust 中的枚举值可以有它们自己的数据类型。这种枚举值带数据类型的语法，彻底把 match 语句和枚举推向了史无前例的高度。

更厉害的是，每一个枚举值可以有不同的数据类型。

枚举值带数据类型的语法格式很简单，如下

```
enum enum_name {
   variant1(data_type1),
   variant2(data_type2),
   variant3(data_type1)
}
```

例如下面的枚举

```
enum GenderCategory {
   Name(String),
   Usr_ID(i32)
}
```

枚举 GenderCategory 有两个枚举值 Name 和 User_ID，它们有着不同的数据类型：String 和 i32。

18.9.1 范例

下面的范例，我们定义了一个带数据类型的枚举 GenderCategory。 然后演示了带数据类型枚举值的初始化和 match 语句的判断

```
// 省的报错
#[derive(Debug)]
enum GenderCategory {
   Name(String),Usr_ID(i32)
}
fn main() {
   let p1 = GenderCategory::Name(String::from("Mohtashim"));
   let p2 = GenderCategory::Usr_ID(100);
   println!("{:?}",p1);
```

```
    println!("{:?}",p2);

    match p1 {
       GenderCategory::Name(val)=> {
          println!("{}",val);
       }
       GenderCategory::Usr_ID(val)=> {
          println!("{}",val);
       }
    }
}
```

编译运行以上 Rust 代码，输出结果如下

```
Name("Mohtashim")
Usr_ID(100)
Mohtashim
```

十九、Rust 模块 Modules

模块 **Module** 用于将函数或结构体按照功能分组。我们通常把相似的函数或者实现相同功能的或者共同实现一个功能的函数和结构体划分到一个模块中。

例如 network 模块包含了所有和网络有关的函数和结构体，graphics 模块则包含了所有和图形处理有关的函数和结构体。

Rust 中的模块，类似于其它语言中的模块或者包的概念，例如 C++ 语言中的命名空间，例如 Java 语言中的包。

比模块更高级别的分组是 **crate**，我们可以将多个模块放到一个 **crate** 下面。crate 是 Rust 语言的基本编译单元。Rust 中的可执行二进制文件程序或者一个库就是一个 carate。

可执行二进制文件程序和库的最大区别，就是 **可执行二进制程序** 有一个包含 main() 方法作为程序入口。

而一个库 (library crate) 是一组可以在其他项目中重用的组件。与二进制包不同，库包没有入口函数（main() 方法）。

Rust 内置了 cargo 作为包管理器，类似于 Python 语言中的 pip 。Rust 官方同时提供了 crates.io 用作所有第三方包的中央存储服务器。你可以使用 cargo install 命令从 crates.io 上下载你的程序所需要的 crate。

上面我们提到了很多新的术语，我们将它们罗列于下表中

术语	中文	说明
crate	crate	Rust 中的基本编译单元，可以被编译为可执行文件或库
cargo	cargo	Rust 官方出品的 Rust 包管理器
module	模块	一个 crate 中有相互逻辑的代码
crates.io	crates.io	Rust 第三方扩展包的中央官方仓库

crate 翻译为中文是 **箱|板条** 的意思，个人觉得还不如翻译为 **项目** 来的实在。

比如交流的时候说 创建一个板条箱，估计没人能听懂

19.1 Rust 中模块的定义语法

Rust 提供了 mod 关键字用于定义一个模块，定义模块的语法格式如下

```
mod module_name {
   fn function_name() {
      // 具体的函数逻辑
   }
   fn function_name() {
      // 具体的函数逻辑
   }
}
```

module_name 必须是一个合法的标识符，它的格式和函数名称一样。

19.1.1 公开的模块和公开的函数

Rust 语言默认所有的模块和模块内的函数都是私有的，也就是只能在模块内部使用。

如果一个模块或者模块内的函数需要导出为外部使用，则需要添加 pub 关键字。

定义一个公开的模块和模块内公开的函数的语法如下

```
//公开的模块
pub mod a_public_module {
   pub fn a_public_function() {
      // 公开的方法
   }
   fn a_private_function() {
      // 私有的方法
   }
}
//私有的模块
mod a_private_module {

   // 私有的方法
```

```
   fn a_private_function() {
   }
}
```

模块 Module 可以是公开可访问的 pub，也可以是私有的。

如果一个模块不添加 pub 关键字，那么它就是私有的，私有的模块不能为外部其它模块或程序所调用。

Rust 语言中的模块默认是私有的。

如果一个模块添加了 pub 关键字，那么它就是公开对外可访问的。

不过需要注意的是，私有模块的所有函数都必须是私有的，而公开的模块，则即可以有公开的函数也可以有私有的函数。

模块中的函数默认都是私有的，如果一个函数没有添加 pub 关键字，那么它就是私有的，相反，添加了 pub 关键字的函数则是公开的。

> 简直和绕口令差不多了....

19.1.2 范例： 定义一个模块

我们已经学习了 Rust 中模块的基本知识和定义语法，接下来我们尝试定义一个模块 movies，它包含一个单独的方法 play()。

play() 方法接受一个单独的字符串参数，然后输出这个字符串。

```
pub mod movies {
   pub fn play(name:String) {
      println!("Playing movie {}",name);
   }
}

fn main(){
   movies::play("Herold and Kumar".to_string());
}
```

运行以上 Rust 代码，输出结果如下

```
Playing movie Herold and Kumar
```

19.2 use 关键字

每次调用外部的模块中的函数或结构体都要添加 **模块限定**，这样似乎有点啰嗦了。

我们能不能在文件头部先把需要调用的函数/结构体引用进来，然后调用的时候就可以省去 **模块限定** 呢？

答案是可以的。

Rust 从 C++ 借鉴了 use 关键字。

use 关键字用于文件头部预先引入需要用到的外部模块中的函数或结构体。

19.2.1 use 关键字的使用语法

```
use public_module_name::function_name;
```

19.2.2 范例

有了 use 关键字，我们就可以预先引入外部模块中的函数和结构体而不用在使用时使用 **全限定模块**。

```
pub mod movies {
   pub fn play(name:String) {
      println!("Playing movie {}",name);
   }
}

use movies::play;

fn main(){
   play("Herold and Kumar ".to_string());
}
```

运行以上 Rust 代码，输出结果如下

```
Playing movie Herold and Kumar
```

19.3 嵌套模块 / 多级模块

Rust 允许一个模块中嵌套另一个模块，换种说法，就是允许多层级模块。

嵌套模块的语法格式很简单，如下

```
pub mod movies {
   pub mod english {
      pub mod comedy {
         pub fn play(name:String) {
            println!("Playing comedy movie {}",name);
         }
      }
   }
}
```

上面这段代码中，movies 模块内嵌了 english 模块，english 模块内嵌了 comedy 模块，而 comedy 模块内才含有真正的函数 play()。

调用或使用嵌套模块的方法也很简单，只要使用两个冒号 (::) 从左到右拼接从外到内的模块即可，例如上面的模块，使用方式如下

```
use movies::english::comedy::play;
```

19.3.1 范例

下面的范例是对上面代码的补充

```
pub mod movies {
   pub mod english {
      pub mod comedy {
         pub fn play(name:String) {
            println!("Playing comedy movie {}",name);
         }
      }
   }
}
use movies::english::comedy::play; // 导入公开的模块

fn main() {
   // 短路径语法
   play("Herold and Kumar".to_string());
   play("The Hangover".to_string());

   // 全路径语法
   movies::english::comedy::play("Airplane!".to_string());
}
```

运行以上 Rust 代码，输出结果如下

```
[www.badu.com]$ cargo run
   Compiling guess-game-app v0.1.0 (/Users/Admin/Downloads/guess-game-app)
    Finished dev [unoptimized + debuginfo] target(s) in 1.73s
     Running `target/debug/guess-game-app`
Playing comedy movie Herold and Kumar
Playing comedy movie The Hangover
Playing comedy movie Airplane!
```

19.4 范例：创建一个库 crate 并编写一些用例

上面大篇幅我们一直都在说模块的一些用法，估计把大家绕的一头雾水了。

下面我们来点真格的，从零开始创建一个库并编写一些测试用例。

首先我们来理一理我们要创建的库的一些基本信息。

元数据	值
crate 名	movie_lib
模块名字	movies

Rust 项目一般使用 cargo 作为包管理器，我们也不例外，我们会用它来管理我们的 crate ，权当复习复习。

19.4.1 第一步 - 创建 movie_lib 库 crate

在你的工作目录下创建一个项目目录叫做 movie_app，创建项目的命令如下

```
$ mkdir movie_app
$ ls
movie_app
```

> 如果你是 Windows 电脑，可以直接点击右键新建目录。

我的工作目录是 /Users/Admin/Downloads/rust。

> 你的工作目录不需要和我的一样，因为只要 movie_app 是一样的即可。另一方面，你用的可能是 Windows 电脑，这个和我的 Mac 苹果电脑路径也不一样。如果有问题，请联系。

接下来在 movie_app 目录下新建一个目录 movie_lib。

然后在 movie_lib 目录下新建文件 Cargo.toml 和 src 目录。

最后在 movie_lib/src 目录下新建 lib.rs 文件和 movies.rs 文件。

创建完成后的目录结构如下

```
movie_app
   movie_lib/
      -->Cargo.toml
      -->src/
         lib.rs
         movies.rs
```

Cargo.toml 文件主要用于保存库 crate 的一些基本信息/元数据，比如库 crate 的版本号、库名称、作者信息等等

上面三个步骤可以在 movie-app 目录下运行 cargo new movie_lib --lib 命令一键创建。

```
$ cd movie_app/
$ cargo new movie_lib --lib
     Created library `movie_lib` package
$ ls
movie-lib
$ tree ../movie-app/
../movie_app/
```

```
└── movie_lib
    ├── Cargo.toml
    └── src
        └── lib.rs

2 directories, 2 files

$ touch movie_lib/src/movies.rs
$ tree ../movie_app/
../movie_app/
└── movie_lib
    ├── Cargo.toml
    └── src
        ├── lib.rs
        └── movies.rs

2 directories, 3 files
```

19.4.2 第二步 - 编辑 Cargo.toml 文件修改库 crate 的一些基本信息

修改之前，我们先来看看 Cargo.toml 文件里的原内容是啥

```
$ cat movie_lib/Cargo.toml
```

输出为

```
[package]
name = "movies_lib"
version = "0.1.0"
authors = ["****<noreply@xx.com>"]
edition = "2018"

[dependencies]
```

如果你不是使用 cargo new 命令创建 movie_lib ，那么这个文件的内容可能是空的。

其实也没啥好改的，因为我们创建的时候信息都挺正确的，如果你要将 crate 名字改成其它的，则可以直接修改 name

```
[package]
name = "movies_lib"
version = "0.1.0"
authors = ["****<noreply@xx.com>"]
edition = "2018"

[dependencies]
```

19.4.3 第三步 - 编辑 lib.rs 文件

lib.rs 文件 用于指定 库 crate 有哪些公开的模块可用。

如果你使用 cargo new 创建 movie_lib，那么 lib.rs 里是有一些内容的

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}
```

这些内容以后我们有空再回来讲讲，现在，我们专注于这个 movie_lib。

因为这是一个 库 crate ，因此我们需要在 lib.rs 文件中指定我们的需要导出的模块名字。

语法格式为

```
pub mod [库名字];
```

例如我们这个库的名字是 movies，因此 lib.rs 文件的内容为

```
pub mod movies;
```

19.4.4 第四步 - 编辑 movies.rs 文件

我们的 movie_lib 库只有一个模块 movies，而这个模块只有一个功能，就是在 movies.rs 中放置一个函数 play() 。

play() 函数用于输出传递过来的字符串参数

```
pub fn play(name:String){
   println!("Playing movie {} :movies_app",name);
}
```

因为 play() 函数需要导出给外部使用因此需要添加 pub 关键字。

19.4.5 第五步 - 编译 movie_lib 库 crate

我们可以在 movie_lib 目录下使用 cargo build 命令来构建/编译项目。这个命令除了编译作用外，还会检查我们的 crate 结构是否正确。

如果构建成功，则输出结果类似于

```
$ cargo build
   Compiling movie_lib v0.1.0 (/Users/Admin/Downloads/rust/movie_app/movie_lib)
    Finished dev [unoptimized + debuginfo] target(s) in 0.89s
```

19.4.6 第六步 - 创建测试应用程序

在 movie_app 目录下新建一个目录 movie_lib_test。

然后在 movie_lib_test 目录下新建文件 Cargo.toml 和 src 目录。

最后在 movie_lib_test/src 目录下新建 main.rs 文件。

创建完成后的目录结构如下

```
../movie_app/
├── movie_lib
│   ├── Cargo.lock
│   ├── Cargo.toml
│   └── src
│       ├── lib.rs
│       └── movies.rs
└── movie_lib_test
    ├── Cargo.toml
    └── src
        └── main.rs

4 directories, 6 files
```

上面三个步骤可以在 movie_app 目录下运行 cargo new movie_lib_test --bin 命令一键创建。

movie_lib_test 是一个测试应用程序，它的主要作用就是测试我们刚刚写的 movie_lib。因为是一个可执行二进制项目，因此 movie_lib_test/src/main.rs 中必须包含 main() 作为入口函数。

19.4.7 第七步 - 编辑 Cargo.toml 添加本地依赖

打开 Cargo.toml 文件并在 [dependencies] 节点下添加

```
movies_lib = { path = "../movie_lib" }
```

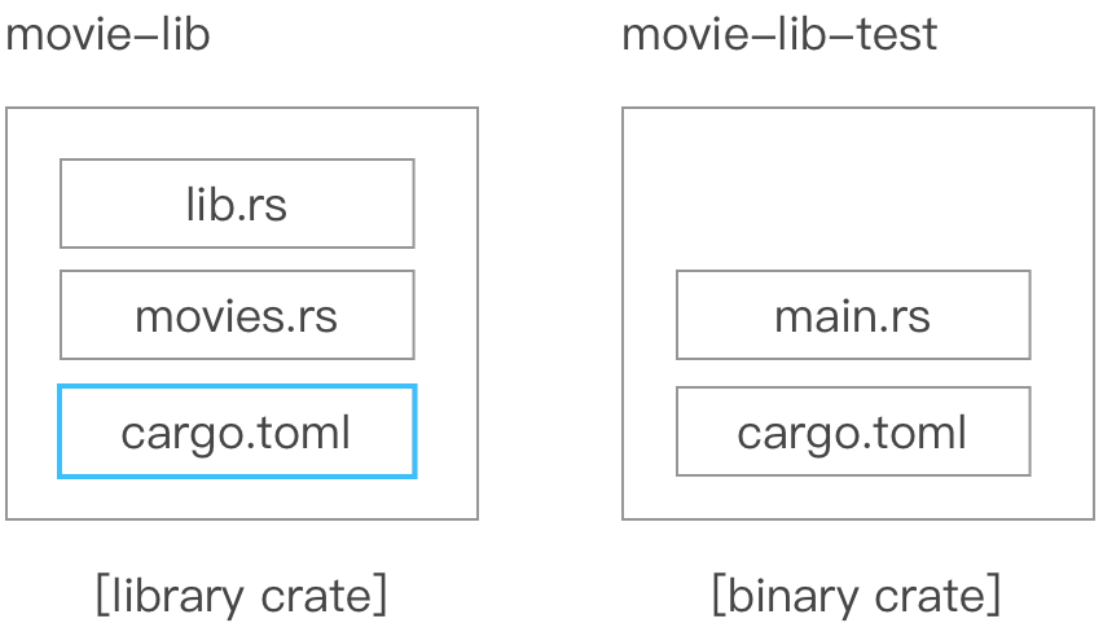
修改完成后的代码如下

```
[package]
name = "movie_lib_test"
version = "0.1.0"
authors = ["*** <noreply@xxx.com>"]
edition = "2018"

[dependencies]
movies_lib = { path = "../movie_lib" }
```

> 注意: 请你特别留意 movies_lib 依赖项的文件位置。

下面的图例标明了 movies_lib 和 测试应用程序 两个项目的目录结构



19.4.8 第八步 - 添加一些使用范例代码到 src/mian.rs 文件中

打开 src/main.rs 文件并复制粘贴一下内容

```
extern crate movies_lib;

use movies_lib::movies::play;

fn main() {
   println!("inside main of test ");
   play("零基础教程".to_string());
}
```

上面的代码中

- extern crate movies_lib; 用于引入我们刚刚创建的模块 movies_lib。这个是必须的
- use movies_lib::movies::play; 用于引入 movies_lib::movies::play。如果没有这句，那么使用

play() 函数就要明确指定模块和模块下的文件。

- play("零基础教程".to_string()); 调用我们 play() 函数并传递字符串参数。

> 注意 1: 如果你不清楚当前项目的 crate 名字，可以打开 Cargo.toml 文件查询。

> 注意 2: 请仔细阅读 use movies_lib::movies::play; 中 :: 隔开的每一部分，它的组成其实是 crate 名字 + 库名字 + 函数名/结构体名/常量名

19.4.9 第九步 - 使用 cargo build 编译或使用 cargo run 运行

终于，激动人心的时刻来临了，我们可以在 movie_lib_test 目录下运行 cargo build 构建项目，当然了，如果仅仅是想看看运行结果，可以直接输入 cargo run 运行，这个命令会先执行 cargo build ，输出结果下:

```
$ cargo run
   Compiling movies_lib v0.1.0 (/Users/Admn/Downloads/rust/movie_app/movie_lib)
   Compiling movie_lib_test v0.1.0
(/Users/Admin/Downloads/rust/movie_app/movie_lib_test)
    Finished dev [unoptimized + debuginfo] target(s) in 1.70s
     Running `target/debug/movie_lib_test`
inside main of test
Playing movie  :movies_app
```

二十、Rust Collections

> collection 翻译成中文是 集合 的意思，set 翻译成中文也是集合的意思。这要如何区分啊？ 在 V2EX 上问了下，马上就有好心人来告诉我，可以把 collection 翻译成 容器，谢谢了

Rust 语言的容器标准库提供了最常见的通用的数据结构的实现。包括 **向量（Vector）**、**哈希表（HashMap）**、**哈希集合（HashSet）** 等等。

Rust 容器库提供的数据结构没有 C++ 或 Java 那么多那么细致，但上面三个也足够使用了。

本章节我们将会详细的介绍上面提到的三个数据结构。

20.1 向量 Vector

前面的Rust数组章节中，我们有提到数组是相同数据类型的值的集合，但数组有一个缺点，就是它的长度是在编译时就确定的，一旦定义就永不可更改。

数组是各个语言所共通的，任何一个语言都不可能为了修复长度不可变这个 BUG 而改变数组长度不可变这个通识。

因此，急需要一个新的数据结构，它的元素布局方式和数组一样，但是长度可以在运行时随意变更。

也就是说，我们需要一个 **长度可变的数组**。于是，向量 Vector 就被提上日程了。

向量 是一个长度可变的数组。它和数组一样，在内存上开辟一段 **连续的内存块** 用于存储元素。

从某些方面说，**向量** 既有数组的特征，又有自己独有的特征：

- 向量的长度是可变的，可以在运行时增长或者缩短。
- 向量也是相同类型元素的集合。
- 向量以特定顺序（添加顺序）将数据存储为元素序列。
- 向量中的每个元素都分配有唯一的索引号。 索引从 0 开始并自增到 n-1，其中 n 是集合的大小。例如集合有 5 个元素，那么第一个元素的下标是 0，最后一个元素的下标是 4。
- 元素添加到向量时会添加到向量的末尾。这个操作类似于 **栈 （ stack)**，因此可以用来实现 **栈** 的功能。
- 向量的内存在 **堆 (heap)** 上存储，因此长度动态可变。

20.1.1 创建向量的语法

Rust 在标准库中定义了结构体 **Vec** 用于表示一个向量。同时提供了 **new()** 静态方法用于创建一个结构体 Vec 的实例。

因此，向量的创建语法格式如下

```
let mut instance_name = Vec::new();
```

除了提供 **new()** 静态方法创建向量之外， Rust 标准库还提供了 **vec!()** 宏来简化向量的创建。

```
let vector_name = vec![val1,val2,val3]
```

结构体 Vec 包含了大量的方法用于操作向量和向量中的元素，我们逻辑几个常见的于下表，并在后面做一个简单的介绍。

方法	签名	说明
new()	pub fn new()->Vec	创建一个空的向量的实例
push()	pub fn push(&mut self, value: T)	将某个值 T 添加到向量的末尾
remove()	pub fn remove(&mut self, index: usize) -> T	删除并返回指定的下标元素。
contains()	pub fn contains(&self, x: &T) -> bool	判断向量是否包含某个值
len()	pub fn len(&self) -> usize	返回向量中的元素个数

20.1.2 使用 Vec::new() 静态方法创建向量

创建向量的一般通过调用 Vec 结构的 **new()** 静态方法来创建。

当有了向量的一个实例后，再通过 **push()** 方法像向量添加元素

```
fn main() {
   let mut v = Vec::new();
   v.push(20);
   v.push(30);
   v.push(40);

   println!("size of vector is :{}",v.len());
   println!("{:?}",v);
}
```

运行以上 Rust 代码，输出结果如下

```
size of vector is :3
[20, 30, 40]
```

上面的代码中，我们使用结构体 **Vec** 提供的静态方法 **new()** 创建向量的一个实例。

有了向量时候之后，使用 **push(val)** 方法像实例添加元素。

len() 方法用于获取向量的元素个数。

20.1.3 使用 vec! 宏创建向量

使用 **Vec::new()** 方法创建一个向量的实例，然后在使用 **push()** 方法添加元素的操作看起来有点复杂。

为了使创建向量看起来像创建数组那么简单，Rust 标准库提供了 **vect!** 用于简化向量的创建。

使用 **vect!** 宏创建向量时，向量的数据类型由第一个元素自动推断出来。

```
fn main() {
   let v = vec![1,2,3];
   println!("{:?}",v);
}
```

运行以上 Rust 代码，输出结果如下

```
[1, 2, 3]
```

向量也是相同类型元素的集合。

因此，如果给向量传递了不同数据类型的值则会引发错误 error[E0308]: mismatched types 。

下面的代码，编译会报错

```
fn main() {
   let v = vec![1,2,3,"hello"];
   println!("{:?}",v);
}
```

错误信息为

```
error[E0308]: mismatched types
 --> src/main.rs:2:23
  |
2 |    let v = vec![1,2,3,"hello"];
  |                       ^^^^^^^ expected integer, found reference
  |
  = note: expected type `{integer}`
             found type `&'static str`

error: aborting due to previous error
```

20.1.4 追加元素到向量中 push()

push() 方法可以将指定的值添加到向量的末尾

```
fn main() {
   let mut v = Vec::new();
   v.push(20);
   v.push(30);
   v.push(40);

   println!("{:?}",v);
}
```

运行以上 Rust 代码，输出结果如下

```
[20, 30, 40]
```

20.1.5 删除向量中的某个元素 remove()

remove() 方法移除并返回向量中指定的下标索引处的元素，将其后面的所有元素移到向左移动一位。

```
fn main() {
   let mut v = vec![10,20,30];
   v.remove(1);
   println!("{:?}",v);
}
```

运行以上 Rust 代码，输出结果如下

```
[10, 30]
```

20.1.6 判断向量是否包含某个元素

contains() 用于判断向量是否包含某个值。

如果值在向量中存在则返回 *true*，否则返回 *false*。

```
fn main() {
   let v = vec![10,20,30];
   if v.contains(&10) {
      println!("found 10");
   }
   println!("{:?}",v);
}
```

运行以上 Rust 代码，输出结果如下

```
found 10
[10, 20, 30]
```

20.1.7 获取向量的长度

len() 方法可以获取向量的长度，也就是向量元素的个数。

```
fn main() {
   let v = vec![1,2,3];
   println!("size of vector is :{}",v.len());
}
```

运行以上 Rust 代码，输出结果如下

```
size of vector is :3
```

20.1.8 访问向量元素的方法

向量既然被称为是可变的数组，那么它的元素当然可以使用 **下标** 语法来访问。

也就是可以使用 **索引号** 来访问向量的每一个元素。

例如下面的代码，我们可以使用 *v[0]* 来访问第一个元素 20，使用 *v[1]* 来访问第二个元素。

```
fn main() {
   let mut v = Vec::new();
   v.push(20);
   v.push(30);

   println!("{:?}",v[0]);
}
```

运行以上 Rust 代码，输出结果如下

```
20
```

20.1.9 迭代/遍历向量

向量本身就实现了迭代器特质，因此可以直接使用 for in 语法来遍历向量

```
fn main() {
   let mut v = Vec::new();
   v.push(20);
   v.push(30);
   v.push(40);
   v.push(500);

   for i in v {
      println!("{}",i);
   }

   // println!("{:?}",v); // 运行出错，因为向量已经不可用
}
```

编译运行以上 Rust 代码，输出结果如下

```
20
30
40
500
```

如果把上面代码中的注释去掉，则会报编译错误

```
fn main() {
   let mut v = Vec::new();
   v.push(20);
   v.push(30);
   v.push(40);
   v.push(500);

   for i in v {
      println!("{}",i);
   }

   println!("{:?}",v); // 运行出错，因为向量已经不可用
}
```

编译出错

```
error[E0382]: borrow of moved value: `v`
  --> src/main.rs:12:20
   |
2  |    let mut v = Vec::new();
   |        ----- move occurs because `v` has type `std::vec::Vec<i32>`, which does
not implement the `Copy` trait
...
8  |    for i in v {
   |             - value moved here
...
12 |    println!("{:?}",v); // 运行出错，因为向量已经不可用
   |                    ^ value borrowed here after move
```

出错原因我们在 Rust 所有权 Ownership 章节已经提到过了，这里就不做详细介绍了。

修复的方式，就是在使用使用 *for in* 来迭代向量的一个引用

```
fn main() {
   let mut v = Vec::new();
   v.push(20);
   v.push(30);
   v.push(40);
   v.push(500);

   for i in &v {
      println!("{}",i);
   }
   println!("{:?}",v);
}
```

编译运行以上 Rust 代码，输出结果如下

```
20
30
40
500
[20, 30, 40, 500]
```

20.2 哈希表 HashMap

哈希表 HashMap 就是 **键值对** 的集合。哈希表中不允许有重复的键，但允许不同的键有相同的值。

从另一方面说，哈希表有点像 **查找表**。键用于查找值。

Rust 语言使用 HashMap 结构体来表示哈希表。

HashMap 结构体在 Rust 语言标准库中的 *std::collections* 模块中定义。

使用 HashMap 结构体之前需要显式导入 *std::collections* 模块。

20.2.1 创建哈希表的语法

Rust 语言标准库 *std::collections* 的结构体 HashMap 提供了 *new()* 静态方法用于创建哈希表的一个实例。

使用 *HashMap::new()* 创建哈希表的语法格式如下

```
let mut instance_name = HashMap::new();
```

new() 方法会创建一个空的哈希表。但这个空的哈希表是不能立即使用的，因为它还没指定数据类型。当我们给哈希表添加了元素之后才能正常使用。

结构体 HashMap 同时提供了大量的方法用于操作哈希表中的元素，我们将常用的几个方法罗列于此

方法	方法签名	说明
insert()	pub fn insert(&mut self, k: K, v: V) -> Option	插入/更新一个键值对到哈希表中，如果数据已经存在则返回旧值，如果不存在则返回 None
len()	pub fn len(&self) -> usize	返回哈希表中键值对的个数
get()	pub fn get<Q: ?Sized>(&lself, k: &Q) -> Option<&V>	根据键从哈希表中获取相应的值
iter()	pub fn iter(&self) -> Iter<K, V>	返回哈希表键值对的无序迭代器，迭代器元素类型为 (&'a K, &'a V)
contains_key()	pub fn contains_key<Q: ?Sized>(&self, k: &Q) -> bool	如果哈希表中存在指定的键则返回 true 否则返回 false
remove()	pub fn remove_entry<Q: ?Sized>(&mut self, k: &Q) -> Option<(K, V)>	从哈希表中删除并返回指定的键值对

20.2.2 插入/更新一个键值对到哈希表中 insert()

insert() 方法用于插入或更新一个键值对到哈希表中。

如果键已经存在，则更新为新的简直对，并则返回旧的值。

如果键不存在则执行插入操作并返回 None。

```
use std::collections::HashMap;
fn main(){
   let mut stateCodes = HashMap::new();
   stateCodes.insert("name","从零蛋开始教程");
   stateCodes.insert("site","https://www.baidu.com");
   println!("{:?}",stateCodes);
}
```

编译运行以上 Rust 代码，输出结果如下

```
{"name": "从零蛋开始教程", "site": "https://www.baidu.com"}
```

20.2.3 获取哈希表中键值对的个数 len()

len() 方法用于获取哈希表的长度，也就是哈希表中键值对的个数。

```
use std::collections::HashMap;
fn main() {
   let mut stateCodes = HashMap::new();
   stateCodes.insert("name","从零蛋开始教程");
   stateCodes.insert("site","https://www.baidu.com");
   println!("size of map is {}",stateCodes.len());
}
```

编译运行以上 Rust 代码，输出结果如下

```
size of map is 2
```

20.2.4 根据键从哈希表中获取相应的值 get()

get() 方法用于根据键从哈希表中获取相应的值。

如果值不存在，也就是哈希表不包含参数的键则返回 None。

如果值存在，则返回值的一个引用。

```
use std::collections::HashMap;
fn main() {
   let mut stateCodes = HashMap::new();
   stateCodes.insert("name","从零蛋开始教程");
   stateCodes.insert("site","https://www.baidu.com");
   println!("size of map is {}",stateCodes.len());
   println!("{:?}",stateCodes);

   match stateCodes.get(&"name") {
      Some(value)=> {
         println!("Value for key name is {}",value);
      }
      None => {
         println!("nothing found");
      }
   }
}
```

编译运行以上 Rust 代码，输出结果如下

```
size of map is 2
{"name": "从零蛋开始教程", "site": "https://www.baidu.com"}
Value for key name is 从零蛋开始教程
```

20.2.5 迭代哈希表 iter()

iter() 方法会返回哈希表中 键值对的引用 组成的无序迭代器。

迭代器元素的类型为 (&'a K, &'a V)。

```
use std::collections::HashMap;
fn main() {
   let mut stateCodes = HashMap::new();
    stateCodes.insert("name","从零蛋开始教程");
    stateCodes.insert("site","https://www.baidu.com");

   for (key, val) in stateCodes.iter() {
      println!("key: {} val: {}", key, val);
   }
}
```

编译运行以上 Rust 代码，输出结果如下

```
key: site val: https://www.baidu.com
key: name val: 从零蛋开始教程
```

20.2.6 是否包含指定的键 contains_key()

contains_key() 方法用于判断哈希表中是否包含指定的 **键值对**。

如果包含指定的键，那么会返回相应的值的引用，否则返回 None。

```
use std::collections::HashMap;
fn main() {
    let mut stateCodes = HashMap::new();
    stateCodes.insert("name","从零蛋开始教程");
    stateCodes.insert("site","https://www.baidu.com");
    stateCodes.insert("slogn","从零蛋开始教程，简单编程");

    if stateCodes.contains_key(&"name") {
        println!("found key");
    }
}
```

编译运行以上 Rust 代码，输出结果如下

```
found key
```

20.2.7 从哈希表中删除指定键值对 remove()

remove() 用于从哈希表中删除指定的键值对。

如果键值对存在则返回删除的键值对，返回的数据格式为 (&'a K, &'a V)。

如果键值对不存在则返回 None

```
use std::collections::HashMap;
fn main() {
    let mut stateCodes = HashMap::new();
    stateCodes.insert("name","从零蛋开始教程");
    stateCodes.insert("site","https://www.baidu.com");
    stateCodes.insert("slogn","从零蛋开始教程，简单编程");

    println!("length of the hashmap {}",stateCodes.len());
    stateCodes.remove(&"site");
    println!("length of the hashmap after remove() {}",stateCodes.len());
}
```

编译运行以上 Rust 代码，输出结果如下

```
length of the hashmap 3
length of the hashmap after remove() 2
```

20.3 哈希集合 HashSet

哈希集合 **HashSet**，简称为 **集合 (set)**，是没有重复值的相同数据类型的值的集合。

集合的最大特征就是没有重复值。

Rust 语言标准库 *std::collections* 中定义了结构体 HashSet 用于描述集合。

std::collections 模块中同时包含了大量的方法用于创建、访问和操作集合。

20.3.1 创建集合的语法

Rust 语言标准库 std::collections 的结构体 HashSet 提供了 new() 静态方法用于创建集合的一个实例。

使用 HashSet::new() 创建集合的语法格式如下

```
let mut hash_set_name = HashSet::new();
```

new() 方法会创建一个空的集合。但这个空的集合是不能立即使用的，因为它还没指定数据类型。当我们给集合添加了元素之后才能正常使用。

结构体 HashSet 同时提供了大量的方法用于操作集合中的元素，我们将常用的几个方法罗列于此

方法	方法原型	描述
insert()	pub fn insert(&mut self, value: T) -> bool	插入一个值到集合中, 如果集合已经存在值则插入失败
len()	pub fn len(&self) -> usize	返回集合中的元素个数
get()	pub fn getQ:?Sized(&self, value: &Q) -> Option<&T>	根据指定的值获取集合中相应值的一个引用
iter()	pub fn iter(&self) -> Iter	返回集合中所有元素组成的无序迭代器, 迭代器元素的类型为 &'a T
contains_key()	pub fn contains<Q: ?Sized> (&self, value: &Q) -> bool	判断集合是否包含指定的值
remove()	pub fn remove<Q: ?Sized>(&mut self, value: &Q) -> bool	从结合中删除指定的值

20.3.2 插入一个值到集合中 insert()

insert() 用于插入一个值到集合中。

insert() 方法的函数原型如下

```
pub fn insert(&mut self, value: T) -> bool
```

insert() 用于插入一个值到集合中，如果集合中已经存在指定的值，则返回 false，否则返回 true。

> 注意： 集合中不允许出现重复的值，因此如果集合中已经存在相同的值，则会插入失败。

```
use std::collections::HashSet;

fn main() {
    let mut languages = HashSet::new();
    languages.insert("Python");
    languages.insert("Rust");
    languages.insert("Ruby");
    languages.insert("PHP");

    languages.insert("Rust"); // 插入失败但不会引发异常

    println!("{:?}",languages);
}
```

编译运行以上 Rust 代码，输出结果如下

```
{"Python", "PHP", "Rust", "Ruby"}
```

20.3.3 获取集合的长度 len()

len() 方法用于获取集合的长度，也就是集合中元素的个数。

len() 方法的函数原型如下

```
pub fn len(&self) -> usize
```

注意： usize 是一个指针长度类型，这个由编译时的电脑 CPU 的构架决定。

20.3.4 范例

```
use std::collections::HashSet;
fn main() {
   let mut languages = HashSet::new();
   languages.insert("Python");
   languages.insert("Rust");
   languages.insert("Ruby");
   languages.insert("PHP");
   println!("size of the set is {}",languages.len());
}
```

编译运行以上 Rust 代码，输出结果如下

```
size of the set is 4
```

20.3.5 返回集合所有元素创建的迭代器 iter()

iter() 方法用于返回集合中所有元素组成的无序迭代器。

iter() 方法的函数原型如下

```
pub fn iter(&self) -> Iter
```

> 注意: 迭代器元素的顺序是无序的，毫无规则的。而且每次调用 iter() 返回的元素顺序都可能不一样。

```
use std::collections::HashSet;
fn main() {
   let mut languages = HashSet::new();
   languages.insert("Python");
   languages.insert("Rust");
   languages.insert("Ruby");
   languages.insert("PHP");

   for language in languages.iter() {
      println!("{}",language);
   }
}
```

编译运行以上 Rust 代码，输出结果如下

```
PHP
Python
Rust
Ruby
```

20.3.6 获取集合中指定值的一个引用 get()

get() 方法用于获取集合中指定值的一个引用。

get() 方法的原型如下

```
pub fn get<Q:?Sized>(&self, value: &Q) -> Option<&T>
```

如果值 value 存在于集合中则返回集合中的相应值的一个引用，否则返回 None。

```
use std::collections::HashSet;
fn main() {
   let mut languages = HashSet::new();
   languages.insert("Python");
   languages.insert("Rust");
   languages.insert("Ruby");
   languages.insert("PHP");

   match languages.get(&"Rust"){
      Some(value)=>{
         println!("found {}",value);
      }
      None =>{
         println!("not found");
      }
   }
   println!("{:?}",languages);
}
```

编译运行以上 Rust 代码，输出结果如下

```
found Rust
{"Python", "Ruby", "PHP", "Rust"}
```

20.3.7 判断集合是否包含某个值 contains()

contains() 方法用于判断集合是否包含指定的值。

contains() 方法的函数原型如下

```
pub fn contains<Q: ?Sized>(&self, value: &Q) -> bool
```

如果值 value 存在于集合中则返回 true ，否则返回 false。

```
use std::collections::HashSet;

fn main() {
   let mut languages = HashSet::new();
   languages.insert("Python");
   languages.insert("Rust");
   languages.insert("Ruby");

   if languages.contains(&"Rust") {
      println!("found language");
   }
}
```

编译运行以上 Rust 代码，输出结果如下

```
found language
```

20.3.8 删除集合元素 remove()

remove() 方法用于从集合中删除指定的值。

remove() 方法的原型如下

```
pub fn remove(&mut self, value: &Q) -> bool
```

删除之前如果值 value 存在于集合中则返回 true，如果不存在则返回 false。

```
use std::collections::HashSet;

fn main() {
   let mut languages = HashSet::new();
   languages.insert("Python");
   languages.insert("Rust");
   languages.insert("Ruby");
   println!("length of the Hashset: {}",languages.len());
   languages.remove(&"Kannan");
   println!("length of the Hashset after remove() : {}",languages.len());
}
```

编译运行以上 Rust 代码，输出结果如下

```
length of the Hashset: 3
length of the Hashset after remove() : 3
```

二十一、Rust 错误处理

人人都会犯错，编程业务不例外。

而犯的每个错误，有的可以挽回，有的不可挽回，可挽回的都能轻松处理，不可挽回的就是永久创伤。

编程里也有两种错误，一种错误可以被捕捉到，然后轻松处理，另一种错误，就没办法了，只能导致程序崩溃退出。

Rust 语言也有错误这个概念，而且把错误分为两大类：**可恢复** 和 **不可恢复**，相当于其它语言的 **异常** 和 **错误**。

Name	描述	范例
Recoverable	可以被捕捉，相当于其它语言的异常 Exception	Result 枚举
UnRecoverable	不可捕捉，会导致程序崩溃退出	panic 宏

> 可恢复 （ Recoverable ）

- 可恢复 （ Recoverable ）错误就是那些可被捕捉的错误，因为可以被捕捉，所以可以被矫正，让程序继续运行。 一旦捕捉到了可恢复的错误，程序就可以通过不断尝试之前那个失败的操作或者选择一个备用的操作。
- 可恢复 （ Recoverable ） 错误的一个典型范例就是：读取不存在文件时的 File Not Found 错误。

> 不可恢复 （ UnRecoverable ）

- 不可恢复 （ UnRecoverable ） 错误就是那些致命的会导致程序崩溃的错误。这些错误一旦发生，就会让程序立即停止。
- 不可恢复 （ UnRecoverable ） 错误的一个典型范例是 数组越界。

与其它语言不同，Rust 语言没有 **异常（Exception）** 这个概念，而是 **可恢复 （ Recoverable ）** 错误。

Rust 语言遇到可恢复错误时会返回一个 **Result<T, E>** 的枚举。如果遇到不可恢复的错误，则会自动调用 panic() 宏。

> panic!() 宏或导致程序立即退出。

21.1 panic!() 宏和不可恢复错误

panic!() 会导致程序立即退出，并在退出时向它的调用者反馈退出原因。

panic!() 宏的语法格式如下

```
panic!( string_error_msg )
```

字符串类型的 string_error_msg 用于向调用者传递程序退出的原因。

一般情况下，当遇到不可恢复错误时，程序会自动调用 panic!()。

但我们也可以通过手动调用 panic!() 以达到让程序退出的目的。

> panic!() 不要乱用，除非遇到不可挽救的错误。

21.1.1 范例1

下面的范例，因为 panic!() 会导致程序立即退出，所以后面的 println!() 宏就不会运行。

```
fn main() {
   panic!("Hello");
   println!("End of main"); // 不可能执行的语句
}
```

编译运行以上 Rust 代码，输出结果如下

```
thread 'main' panicked at 'Hello', main.rs:3
```

21.1.2 范例2: 数组越界错误

下面的代码，因为数组的最大下标是 2 ，远远小于 10，因此会触发数组越界错误。

```
fn main() {
   let a = [10,20,30];
   a[10]; // 因为 10 超出了数组的长度，所以会触发不可恢复错误
}
```

编译运行以上 Rust 代码，输出结果如下

```
warning: this expression will panic at run-time
--> main.rs:4:4
  |
4 | a[10];
  | ^^^^^ index out of bounds: the len is 3 but the index is 10

$main
thread 'main' panicked at 'index out of bounds: the len
is 3 but the index is 10', main.rs:4
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

21.1.3 范例 3： 手动出发 panic!() 让程序退出

如果程序执行过程中违反了既定的业务规则，可以手动调用 panic!() 宏让程序退出。

例如下面的代码，因为 13 是奇数，违反了要求偶数的规则。

```
fn main() {
   let no = 13;
   // 测试奇偶
   if no % 2 == 0 {
      println!("Thank you , number is even");
   } else {
      panic!("NOT_AN_EVEN");
   }
   println!("End of main");
}
```

编译运行以上 Rust 代码，输出结果如下

```
thread 'main' panicked at 'NOT_AN_EVEN', main.rs:9
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

21.2 Result 枚举和可恢复错误

一些比较古老的语言，比如 C 通过设置全局变量 errno 来告诉程序发生了什么错误，而其它的语言，比如 Java 在返回类型的基础上还要通过指定可捕捉的异常来达到程序可恢复的目的，而比较现代的语言，比如 Go 则是通过将错误和正常值一起返回来达到可恢复的目的。

Rust 在可恢复错误（ Recoverable ）上更大胆。它使用一个 Result 枚举来封装正常返回的值和错误信息。 带来的好处就是只要一个变量就能接收正常值和错误信息，又不会污染全局空间。

Result<T,E> 枚举被设计用于处理可恢复错误。

Result 枚举的定义如下

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

Result<T,E> 枚举包含了两个值：OK 和 Err。

T 和 E 则是两个范型参数：

- T 用于当 Result 的值为 OK 时作为正常返回的值的数据类型。
- E 用于当 Result 的值为 Err 时作为错误返回的错误的类型。

21.2.1 范例1：Result 枚举的简单使用

下面的范例，我们通过打开一个不存在的文件来演示下 Result 枚举的使用

```
use std::fs::File;
fn main() {
   let f = File::open("main.jpg"); //文件不存在，因此值为 Result.Err
   println!("{:?}",f);
}
```

编译运行以上 Rust 代码，输出结果如下

```
Err(Error { repr: Os { code: 2, message: "No such file or directory" } })
```

上面的代码，如果 main.jpg 存在则结果为 OK(File)。 如果文件不存在则结果为 Err(Error)。

上面的代码仅仅是输出错误信息，这只能是演示目的，正常情况下我们要根据结果类型作出不同的选择。

21.2.2 范例 2: 捕捉错误信息并恢复程序运行

下面的代码，我们使用 match 对 Result 的不同值作出不同的处理。

```
use std::fs::File;
fn main() {
   let f = File::open("main.jpg");   // main.jpg 文件不存在
   match f {
      Ok(f)=> {
         println!("file found {:?}",f);
      },
      Err(e)=> {
         println!("file not found \n{:?}",e);   // 处理错误
      }
   }
   println!("end of main");
}
```

编译运行以上 Rust 代码，输出结果如下

```
file not found
Os { code: 2, kind: NotFound, message: "The system cannot find the file specified." }
end of main
```

> 注意: 上面的代码，不管 f 的值是什么，最后的 println!("end of main"); 都会运行。

21.2.3 范例 3： 实战演练函数返回错误

下面的代码，我们定义了一个函数 is_even()。如果传递的参数不是偶数则会抛出一个可恢复错误。

```
fn main(){
   let result = is_even(13);
   match result {
      Ok(d)=>{
         println!("no is even {}",d);
      },
      Err(msg)=>{
         println!("Error msg is {}",msg);
      }
   }
   println!("end of main");
}
fn is_even(no:i32)->Result<bool,String> {
   if no%2==0 {
      return Ok(true);
   } else {
      return Err("NOT_AN_EVEN".to_string());
   }
}
```

编译运行以上 Rust 代码，输出结果如下

```
Error msg is NOT_AN_EVEN
end of main
```

21.3 unwrap() 函数和 expect() 函数

上面的 Result<T,E> ，用 match 语句处理起来蛮不错的样子，但写多了就会有 Go 语言那种漫天飞舞 if err != nil 的赶脚。

有的时候我们不想处理或者让程序自己处理 Err， 有时候我们只要 OK 的具体值就可以了。

针对这两种处女座诉求， Rust 语言的开发者们在标准库中定义了两个帮助函数 *unwrap()* 和 *expect()*。

方法	原型	说明
unwrap	unwrap(self):T	如果 self 是 Ok 或 Some 则返回包含的值。否则会调用宏 panic!() 并立即退出程序
expect	expect(self,msg:&str):T	如果 self 是 Ok 或 Some 则返回包含的值。否则调用 panic!() 输出自定义的错误并退出

expect() 函数用于简化不希望事情失败的错误情况。而 unwrap() 函数则在返回 OK 成功的情况下，提取返回的实际结果。

> unwrap() 和 expect() 不仅能够处理 Result <T,E> 枚举，还可以用于处理 Option 枚举。

21.4 unwrap() 函数

unwrap() 函数返回操作成功的实际结果。如果操作失败，它会调用 panic!() 并输出默认的错误消息。

unwrap() 函数的原型如下

```
unwrap(self):T
```

实际上，unwrap() 函数内部的实现就是我们上面见过的 match 语句。

21.4.1 范例1

下面代码，我们使用 unwrap() 函数对判断偶数的那个范例进行改造，看起来是不是舒服多了

```
fn main(){
   let result = is_even(10).unwrap();
   println!("result is {}",result);
   println!("end of main");
}
fn is_even(no:i32)->Result<bool,String> {
   if no%2==0 {
      return Ok(true);
   } else {
      return Err("NOT_AN_EVEN".to_string());
   }
}
```

编译运行以上 Rust 代码，输出结果如下

```
result is true
end of main
```

21.4.2 范例 2

如果我们将 is_even(10) 传递的 10 改成 13 则会退出程序并输出错误消息

```
fn main(){
   let result = is_even(10).unwrap();
   println!("result is {}",result);
   println!("end of main");
}
fn is_even(no:i32)->Result<bool,String> {
   if no%2==0 {
      return Ok(true);
   } else {
      return Err("NOT_AN_EVEN".to_string());
   }
}
```

编译运行以上 Rust 代码，输出结果如下

```
thread 'main' panicked at 'called `Result::unwrap()` on
an `Err` value: "NOT_AN_EVEN"', libcore\result.rs:945:5
note: Run with `RUST_BACKTRACE=1` for a backtrace
```

21.5 函数 expect()

函数 expect() 当 self 是 Ok 或 Some 则返回包含的值。否则调用panic!() 输出自定义的错误并退出程序。

函数 expect() 的原型如下

```
expect(self,msg:&str):T
```

函数 expect() 和 unwrap() 一样，唯一的不同点是 当错误发生时，expect() 会输出自定义的错误消息而不是默认的错误消息。

21.5.1 范例

下面的代码，我们使用 expect() 函数改造下上面那个文件不存在的 match 范例

```
use std::fs::File;
fn main(){
   let f = File::open("pqr.txt").expect("File not able to open"); // 文件不存在
   println!("end of main");
}
```

编译运行以上 Rust 代码，输出结果如下

```
thread 'main' panicked at 'File not able to open: Error { repr: Os
{ code: 2, message: "No such file or directory" } }', src/libcore/result.rs:860
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

二十二、Rust 泛型

泛型 就是可以在运行时指定数据类型的机制。

泛型 最大的好处就是一套代码可以应用于多种类型。比如我们的 **向量**，可以是整型向量，也可以是字符串向量。

泛型 既能保证数据安全和类型安全，同时还能减少代码量。所以，现代语言，没有范型简直就是鸡肋。

嘿，说的就是你，隔壁的 XX 语言。

Rust 语言中的泛型主要包含 **泛型集合**、**泛型结构体**、**泛型函数**、**范型枚举** 和 **特质** 几个方面。

22.1 Rust 语言中的泛型

Rust 使用使用 语法来实现泛型的东西指定数据类型。 其中 *T* 可以是任意数据类型。

例如下面的两个类型

```
age: i32
age: String
```

使用泛型，则可以直接声明为

```
age:T
```

然后在使用过程中指定 *T* 的类型为 *i32* 或 *String* 即可。

22.2 泛型集合

日常使用过程中，我们最常见的泛型应用莫过于 **集合 (collection)** 了。

比如我们一直在使用的向量 Vec。它可以是一个字符串向量，也可以是一个整形向量。

下面的代码，我们声明了一个整型向量

```
fn main(){
   let mut vector_integer: Vec<i32> = vec![20,30];
   vector_integer.push(40);
   println!("{:?}",vector_integer);
}
```

编译运行以上 Rust 范例，输出结果如下

```
[20, 30, 40]
```

如果向整型向量传递一个非整型参数，则会报错

```
fn main() {
   let mut vector_integer: Vec<i32> = vec![20,30];
   vector_integer.push(40);
   vector_integer.push("hello");
   //error[E0308]: 类型不匹配
   println!("{:?}",vector_integer);
}
```

上面的代码中可以看出，整型的向量只能添加整型的参数，如果尝试添加一个字符串参数则会报错。

从某些方面说，**泛型** 让集合更通用，也更安全。

22.3 泛型结构体

我们也可以把某个结构体声明为泛型的，**泛型结构体** 主要是结构体的成员类型可以是泛型。

泛型结构体的定义语法如下

```
struct struct_name<T> {
   field:T,
}
```

例如我们可以定义一个泛型结构体 Data，它只有一个成员 value，类型是一个泛型。

```
struct Data<T> {
   value:T,
}
```

把我们的泛型结构体语法和泛型结构体实例 Data 放在一起对比下，很容易区分泛型结构体和普通结构体的不同。

22.3.1 范例

下面的范例，我们使用了刚刚定义的泛型结构体 Data，演示了如何使用泛型结构体

```
fn main() {
   // 泛型为 i32
   let t:Data<i32> = Data{value:350};
   println!("value is :{} ",t.value);
   // 泛型为 String
   let t2:Data<String> = Data{value:"Tom".to_string()};
   println!("value is :{} ",t2.value);
}

struct Data<T> {
   value:T,
}
```

编译运行以上 Rust 范例，输出结果如下

```
value is :350
value is :Tom
```

22.4 特质 Traits

Rust 语言中没有 **类 (class)** 这个概念，于是顺带把 **接口 (interface)** 这个概念也取消了。

可是，没有了 **接口** 我们要如何证明两个结构体之间的关系呢？

为此，Rust 提供了 **特质 Traits** 这个蛋炒饭的概念。

说蛋炒饭，是因为很多语言都有特质这个概念。

Rust 提供了 **特质 Traits** 这个概念，用于跨多个结构体实现一组标准行为（方法）。

从某些方面来说，虽然 Rust 语言没有接口的概念，但 特质 Traits 实打实的就是接口啊。

22.4.1 特质的定义语法

Rust 语言提供了 trait 关键字用于定义 **特质**。

特质 其实就是一组方法原型。它的语法格式如下

```
trait some_trait {

   // 没有任何实现的虚方法
   fn method1(&self);

   // 有具体实现的普通方法
   fn method2(&self){
      //方法的具体代码
   }
}
```

从语法格式来看，特征可以包含具体方法（带实现的方法）或抽象方法（没有具体实现的方法）

如果想让某个方法的定义在实现了特质的结构体所共享，那么推荐使用具体方法。

如果想让某个方法的定义由实现了特质的结构体自己定义，那么推荐使用抽象方法。

即使某个方法是具体方法，结构体也可以对该方法进行重写。

22.4.2 实现特质 impl for

前面我们说过了，Rust 中的 特质 相当于其它语言的 **接口 （ interface ）** 。

其它语言，比如 Java 实现接口使用的是 implement 关键字。

但 Rust 却不这么做，而是提供了一个更加语义化的词组 **impl for**。

如果要为某个结构体 （ struct ）实现某个特质，需要使用 **impl for** 语句。

impl 是 implement 的缩写。impl for 语义已经很明显了，就是为 xxx 实现 xxx 的意思。

impl for 语句的语法格式如下

```
impl some_trait for structure_name {
   // 实现 method1() 的具体代码
   fn method1(&self ){
   }
}
```

> 注意: 特质的方法是结构体的成员方法，因此第一个参数是 &self。

22.4.3 范例: impl for 的简单使用

下面的范例，我们首先定义了一个结构体 Book 和一个特质 Printable。

然后我们使用 impl for 语句为 Book 实现特质 Printable。

```
fn main(){
   //创建结构体 Book 的实例
   let b1 = Book {
      id:1001,
      name:"Rust in Action"
   };
   b1.print();
}

//声明结构体
struct Book {
   name:&'static str,
   id:u32
}
// 声明特质
trait Printable {
   fn print(&self);
}
```

```
// 实现特质
impl Printable for Book {
   fn print(&self){
      println!("Printing book with id:{} and name {}",self.id,self.name)
   }
}
```

编译运行以上 Rust 范例，输出结果如下

```
Printing book with id:1001 and name Rust in Action
```

22.5 泛型函数

泛型也可以用在函数中，我们称使用了泛型的函数为 泛型函数。

泛型函数 主要是指 函数的参数 是泛型。

> 注意: 泛型函数并不要求所有参数都是泛型，而是某个参数是泛型。

泛型函数的定义语法如下

```
fn function_name<T[:trait_name]>(param1:T, [other_params]) {

   // 函数实现代码
}
```

例如我们定义一个可以打印输出任意类型的泛型函数 print_pro

```
fn print_pro<T:Display>(t:T){
   println!("Inside print_pro generic function:");
   println!("{}",t);
}
```

对比语法和范例，泛型函数的定义就很简单了。

22.5.1 范例

下面的范例我们使用刚刚定义的泛型函数 print_pro()，该函数会打印输出传递给它的参数。

传递的参数可以是实现了 Display 特质的任意类型。

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

fn main(){
   print_pro(10 as u8);
   print_pro(20 as u16);
   print_pro("Hello TutorialsPoint");
}

fn print_pro<T:Display>(t:T){
   println!("Inside print_pro generic function:");
   println!("{}",t);
}
```

编译运行以上 Rust 范例，输出结果如下

```
Inside print_pro generic function:
10
Inside print_pro generic function:
20
Inside print_pro generic function:
Hello TutorialsPoint
```

二十三、Rust IO

本章节我们来讲讲 Rust 中的重要内容 **IO 输入输出**。也就是 Rust 语言如何从标准输入例如键盘中读取数据并将读取的数据显示出来。

本章节我们会简单的介绍 Rust 语言 IO 输入输出的三大块内容：**从标准输入读取数据**、**把数据写入标准输出**、**命令行参数**。

23.1 读取 和 写入 类型

Rust 标准库通过两个特质 (Trait) 来组织 IO 输入输出

- Read 特质用于 读
- Write 特质用于 写

特质	说明	范例
Read	包含了许多方法用于从输入流读取字节数据	Stdin,File
Write	包含了许多方法用于向输出流中写入数据，包含字节数据和 UTF-8 数据两种格式	Stdout,File

23.2 Read Trait 特质 / 标准输入流

Read 特质是一个用于从输入流读取字节的组件。输入流包括 标准输入、键盘、鼠标、命令行、文件等等。

Read 特质中的 *read_line()* 方法用于从输入流中读取一行的字符串数据。它的相关信息如下

Trait 特质	方法	说明
Read	read_line(&mut line)->Result	从输入流中读取一行的字符串数据并存储在 line 参数中。返回 Result 枚举，如果成功则返回读取的字节数。

23.2.1 范例：从命令行/标准输入流 stdin() 中读取数据

Rust 程序可以在运行时接收用户输入的数据。

接收的方式就是从标准输入流 （一般是键盘） 中读取用户输入的内容。

下面的代码，从标准输入流/命令行 中读取用户的输入，并显示出来。

```
fn main(){
   let mut line = String::new();
   println!("请输入你的名字:");
   let b1 = std::io::stdin().read_line(&mut line).unwrap();
   println!("你好 , {}", line);
   println!("读取的字节数为: {}", b1);
}
```

编译运行以上 Rust 代码，输出结果如下

```
请输入你的名字:
从零蛋开始教程
你好 , 从零蛋开始教程

读取的字节数为: 13
```

标准库提供的 std::io::stdin() 会返回返回当前进程的标准输入流 stdin 的句柄。

而 read_line() 则是标准入流 stdin 的句柄上的一个方法，用于从标准输入流读取一行的数据。

read_line() 方法的返回值值一个 Result 枚举，而 unwrap() 则是一个帮助方法，用于简化可恢复错误的处理。它会返回 Result 中存储的实际值。

> 注意: read_line() 方法会自动删除行尾的换行符 \n

23.3 Write Trait 特质 / 标准输出流

Write 特质是一个用于向输出流写入字节的组件。输出流包括 标准输出、命令行、文件 等等。

Write 特质中的 write() 方法完成实际的输出操作。它的相关信息如下

Trait 特质	方法	说明
Write	write(&buf) -> Result	把参数 buf 中的全部或部分字节写入底层的流。返回 Result 枚举，如果成功则返回写入的字节数。

> 注意: write() 方法并不会输出结束时自动追加换行符 \n。如果需要换行符则需要手动添加

23.3.1 范例： 使用 stdout() 向标准输出写入内容

我们之前用过的无数次的 print!() 宏和 println!() 宏都可以向标准输出写入内容。

不过我们这次来一点不一样的，我们使用标准库 std::io 提供的 stdout().write() 方法来输出内容。

```
use std::io::Write;
fn main() {
   let b1 = std::io::stdout().write("从零蛋开始教程 ".as_bytes()).unwrap();
   let b2 =
std::io::stdout().write(String::from("www.baidu.com").as_bytes()).unwrap();
   std::io::stdout().write(format!("\n写入的字节数为：{}\n",
(b1+b2)).as_bytes()).unwrap();
}
```

编译运行以上 Rust 代码，输出结果如下

```
从零蛋开始教程 www.baidu.com
写入的字节数为：24
```

标准库提供的 std::io::stdout() 会返回返回当前进程的标准输出流 stdout 的句柄。

而 write() 则是标准输出流 stdout 的句柄上的一个方法，用于向标准输出流写入字节流内容。

write() 方法的返回值值一个 Result 枚举，而 unwrap() 则是一个帮助方法，用于简化可恢复错误的处理。它会返回 Result 中存储的实际值。

> 注意: 和文件相关的 IO 我们会在下一个章节介绍。

23.4 命令行参数

命令行参数是程序执行前就通过终端或命令行提示符或 Shell 传递给程序的参数。

命令行参数有点类似于传递给函数的实参。

比如我们之前见过的无数次的 main.exe 。之前我们运行它的时候没有传递任何参数

```
./main.exe
```

但实际上我们是可以传递一些参数的，不管程序使用与否，比如我们可以传递 2019 和 从零蛋开始教程 作为参数

./main.exe 2019 从零蛋开始教程 如果要传递多个参数，多个参数之间必须使用 空格（' '） 分隔。如果参数里有空格，则参数必须使用 双引号（"）包起来。

```
./main.exe 2019 "从零蛋开始教程 简单编程"
```

Rust 语言在标准库中内置了 std::env::args() 函数返回所有的命令行参数。

std::env::args() 返回的结果包含了程序名。例如上面的命令，std::env::args() 中存储的结果为

```
["./main.exe","2019","从零蛋开始教程 简单编程"]
```

23.4.1 范例：输出命令行传递的所有参数

下面的代码，我们通过迭代 std::env::args() 来输出所有传递给命令行的参数。

```
main.rs
// main.rs
fn main(){
   let cmd_line = std::env::args();
   println!("总共有 {} 个命令行参数",cmd_line.len()); // 传递的参数个数
   for arg in cmd_line {
      println!("[{}]",arg); // 迭代输出命令行传递的参数
   }
}
```

编译运行以上 Rust 代码，输出结果如下

```
$ cargo build
   Compiling guess-game-app v0.1.0 (/Users/Admin/Downloads/guess-game-app)
    Finished dev [unoptimized + debuginfo] target(s) in 0.32s
$ cargo run 1 从零蛋开始教程 3 简单编程
    Finished dev [unoptimized + debuginfo] target(s) in 0.04s
     Running `target/debug/guess-game-app 1 '从零蛋开始教程' 3 '简单编程'`
总共有 5 个命令行参数
[target/debug/guess-game-app]
[1]
[从零蛋开始教程]
[3]
[简单编程]
```

上面的程序，使用 cargo build 命令编译之后，就会生成一个可执行文件 main.exe 或 main。

Rust 语言命令行参数和其它语言的命令行参数一样，都使用 空格（' '） 来分割所有的参数。例如上面我们传递的 1 从零蛋开始教程 3 简单编程 会通过空格分割成 ["1","从零蛋开始教程","3"," "简单编程"] 4 个参数。

但实际上，我们的 std::env::args() 存储者 5 个参数，第一个参数是当前的程序名

程序名是相当于当前的执行目录的。

23.4.2 范例：从命令行读取多个参数

下面的代码从命令行读取多个以空格分开的参数，并统计这些参数的总和。

```
fn main(){
   let cmd_line = std::env::args();
   println!("总共有 {} 个命令行参数",cmd_line.len()); // 传递的参数个数

   let mut sum = 0;
   let mut has_read_first_arg = false;

   //迭代所有参数并计算它们的总和

   for arg in cmd_line {
      if has_read_first_arg { // 跳过第一个参数，因为它的值是程序名
         sum += arg.parse::<i32>().unwrap();
      }
      has_read_first_arg = true; // 设置跳过第一个参数，这样接下来的参数都可以用于计算
   }
   println!("和值为:{}",sum);
}
```

编译运行以上 Rust 代码，输出结果如下

```
$ cargo build
   Compiling guess-game-app v0.1.0 (/Users/Admin/Downloads/guess-game-app)
    Finished dev [unoptimized + debuginfo] target(s) in 1.59s
$ cargo run 1 2 3 4
    Finished dev [unoptimized + debuginfo] target(s) in 0.04s
     Running `target/debug/guess-game-app 1 2 3 4`
总共有 5 个命令行参数
和值为:10
```

二十四、Rust 文件读写

Rust 标准库提供了大量的模块和方法用于读写文件。

Rust 语言使用结构体 **File** 来描述/展现一个文件。

结构体 **File** 有相关的成员变量或函数用于表示程序可以对文件进行的某些操作和可用的操作方法。

所有对结构体 **File** 的操作方法都会返回一个 **Result** 枚举。

下表列出了一些常用的文件读写方法。

模块	方法/方法签名	说明
std::fs::File	open() / pub fn open(path: P) -> Result	静态方法，以 只读 模式打开文件
std::fs::File	create() / pub fn create(path: P) -> Result	静态方法，以 可写 模式打开文件。如果文件存在则清空旧内容。如果文件不存在则新建
std::fs::remove_file	remove_file() / pub fn remove_file(path: P) -> Result<()>	从文件系统中删除某个文件
std::fs::OpenOptions	append() / pub fn append(&mut self, append: bool) -> &mut OpenOptions	设置文件模式为 追加
std::io::Writes	write_all() / fn write_all(&mut self, buf: &[u8]) -> Result<()>	将 buf 中的所有内容写入输出流
std::io::Read	read_to_string() / fn read_to_string(&mut self, buf: &mut String) -> Result	读取所有内容转换为字符串后追加到 buf 中

24.1 Rust 打开文件

Rust 标准库中的 std::fs::File 模块提供了静态方法 open() 用于打开一个文件并返回文件句柄。

open() 函数的原型如下

```
pub fn open(path: P) -> Result
```

open() 函数用于以只读模式打开一个已经存在的文件，如果文件不存在，则会抛出一个错误。如果文件不可读，那么也会抛出一个错误。

24.1.1 范例

下面的范例，我们使用 open() 打开当前目录下的文件，已经文件已经存在，所以不会抛出任何错误

```
fn main() {
    let file = std::fs::File::open("data.txt").unwrap();
    println!("文件打开成功：{:?}",file);
}
```

编译运行以上 Rust 代码，输出结果如下

```
文件打开成功：File { fd: 3, path: "/Users/Admin/Downloads/guess-game-app/src/data.txt", read: true, write: false }
```

如果文件 data.txt 不存在，则会抛出以下错误

```
thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value: Os { code: 2,
kind: NotFound, message: "No such file or directory" }', src/libcore/result.rs:997:5
note: Run with `RUST_BACKTRACE=1` environment variable to display a backtrace.
```

24.2 Rust 创建文件

Rust 标准库中的 std::fs::File 模块提供了静态方法 create() 用于创建一个文件并返回创建的文件句柄。

create() 函数的原型如下

```
pub fn create(path: P) -> Result
```

create() 函数用于创建一个文件并返回创建的文件句柄。如果文件已经存在，则会内部调用 open() 打开文件。如果创建失败，比如目录不可写，则会抛出错误

24.2.1 范例

下面的代码，使用 create() 函数创建文件 data.txt。

```
fn main() {
   let file = std::fs::File::create("data.txt").expect("create failed");
   println!("文件创建成功:{:?}",file);
}
```

编译运行以上 Rust 代码，输出结果如下

```
文件创建成功:File { fd: 3, path: "/Users/Admin/Downloads/guess-game-app/src/data.txt",
read: false, write: true }
```

24.3 Rust 写入文件

Rust 语言标准库 std::io::Writes 提供了函数 write_all() 用于向输出流写入内容。

因为文件流也是输出流的一种，所以该函数也可以用于向文件写入内容。

write_all() 函数在模块 std::io::Writes 中定义，它的函数原型如下

```
fn write_all(&mut self, buf: &[u8]) -> Result<()>
```

write_all() 用于向当前流写入 buf 中的内容。如果写入成功则返回写入的字节数，如果写入失败则抛出错误

24.3.1 范例

下面的代码，我们使用 write_all() 方法向文件 data.txt 写入一些内容

```
use std::io::Write;
fn main() {
   let mut file = std::fs::File::create("data.txt").expect("create failed");
   file.write_all("从零蛋开始教程".as_bytes()).expect("write failed");
   file.write_all("\n简单编程".as_bytes()).expect("write failed");
   println!("data written to file" );
}
```

编译运行以上 Rust 范例，输出结果如下

```
从零蛋开始教程
简单编程
```

> 注意： write_all() 方法并不会在写入结束后自动写入换行符 \n。

24.4 Rust 读取文件

Rust 读取内容的一般步骤为:

1. 使用 open() 函数打开一个文件
2. 然后使用 read_to_string() 函数从文件中读取所有内容并转换为字符串。

open() 函数我们前面已经介绍过了，这次我们主要来讲讲 read_to_string() 函数。

read_to_string() 函数用于从一个文件中读取所有剩余的内容并转换为字符串。

read_to_string() 函数的原型如下

```
fn read_to_string(&mut self, buf: &mut String) -> Result
```

read_to_string() 函数用于读取文件中的所有内容并追加到 buf 中，如果读取成功则返回读取的字节数，如果读取失败则抛出错误。

24.4.1 范例

我们首先在当前目录下新建一个文件 data.txt 并写入以下内容

```
从零蛋开始教程
简单编程
```

然后我们写一个小范例，使用 read_to_string 函数从文件中读取内容。

```
use std::io::Read;

fn main(){
   let mut file = std::fs::File::open("data.txt").unwrap();
   let mut contents = String::new();
   file.read_to_string(&mut contents).unwrap();
   print!("{}", contents);
}
```

编译运行以上 Rust 代码，输出结果如下

```
从零蛋开始教程
简单编程
```

24.5 追加内容到文件末尾

Rust 核心和标准库并没有提供直接的函数用于追加内容到文件的末尾。

但提供了函数 append() 用于将文件的打开模式设置为 追加。

当文件的模式设置为 追加 之后，写入文件的内容就不会代替原先的旧内容而是放在旧内容的后面。

函数 append() 在模块 std::fs::OpenOptions 中定义，它的函数原型为

```
pub fn append(&mut self, append: bool) -> &mut OpenOptions
```

24.5.1 范例

下面的范例，我们使用 append() 方法修改文件的打开模式为 追加。

```
use std::fs::OpenOptions;
use std::io::Write;

fn main() {
   let mut file = OpenOptions::new().append(true).open("data.txt").expect(
      "cannot open file");
   file.write_all("www.baidu.com".as_bytes()).expect("write failed");
   file.write_all("\n从零蛋开始教程".as_bytes()).expect("write failed");
   file.write_all("\n简单编程".as_bytes()).expect("write failed");
   println!("数据追加成功");
}
```

编译运行以上 Rust 代码，输出结果如下

```
数据追加成功
```

打开 data.txt 文件，可以看到内容如下

```
从零蛋开始教程
简单编程www.baidu.com
从零蛋开始教程
简单编程
```

24.6 Rust 删除文件

Rust 标准库 std::fs 提供了函数 remove_file() 用于删除文件。

remove_file() 函数的原型如下

```
pub fn remove_file<P: AsRef>(path: P) -> Result<()>
```

> 注意，删除可能会失败，即使返回结果为 OK，也有可能不会立即就删除。

24.6.1 范例

下面的代码，我们使用 remove_file() 方法删除 data.txt。

expect() 方法当删除出错时用于输出自定义的错误消息。

```
use std::fs;
fn main() {
   fs::remove_file("data.txt").expect("could not remove file");
   println!("file is removed");
}
```

编译运行以上范例，输出结果如下

```
file is removed
```

打开当前目录，我们可以发现文件已经被删除了。

24.7 Rust 复制文件

Rust 标准库没有提供任何函数用于复制一个文件为另一个新文件。

但我们可以使用上面提到的函数和方法来实现文件的复制功能。

下面的代码，我们模仿简单版本的 copy 命令

```
copy old_file_name  new_file_name
```

具体的 Rust 代码如下

```
use std::io::Read;
use std::io::Write;
```

```
fn main() {
   let mut command_line: std::env::Args = std::env::args();
   command_line.next().unwrap();

   // 跳过程序名
   // 原文件
   let source = command_line.next().unwrap();

   // 新文件
   let destination = command_line.next().unwrap();
   let mut file_in = std::fs::File::open(source).unwrap();
   let mut file_out = std::fs::File::create(destination).unwrap();
   let mut buffer = [0u8; 4096];
   loop {
      let nbytes = file_in.read(&mut buffer).unwrap();
      file_out.write(&buffer[..nbytes]).unwrap();
      if nbytes < buffer.len() { break; }
   }
}
```

首先编译上面的代码

```
$ rustc main.rs
```

然后使用下面的命令来运行

windows 电脑

```
$ ./main.exe data.txt data_new.txt
```

linux / mac 电脑

```
$ ./main data.txt data_new.txt
```

其中

data.txt 为我们想要复制的原文件路径 data_new.txt 为我们想要的新文件路径

二十五、Rust Cargo 包管理器

Rust 内置了一个包管理器 **cargo**。它会随着 Rust 的安装而安装。

cargo 类似于 Python 中的 pip 或 Ruby 中的 RubyGems 或 Node.js 中的 NPM。

当然了，cargo 不仅仅是一个包管理器，它还是 Rust 的项目管理利器。

25.1 检查 cargo 是否安装和安装的版本

打开终端或命令行提示符或 Shell，输入下面的命令然后回车

```
cargo --version
```

如果已经安装，则输出结果类似于

```
cargo 1.35.0
```

cargo 包管理器的版本和 Rust 语言是同步的。

25.2 cargo 的帮助信息

如果想要查看 cargo 的帮助信息或查看 cargo 提供了哪些命令和功能，可以在 终端 中输入 cargo 或 cargo -h 然后回车

输出结果类似于

```
Rust's package manager

USAGE:
    cargo [OPTIONS] [SUBCOMMAND]

OPTIONS:
    -V, --version           Print version info and exit
        --list              List installed commands
        --explain <CODE>    Run `rustc --explain CODE`
    -v, --verbose           Use verbose output (-vv very verbose/build.rs output)
    -q, --quiet             No output printed to stdout
        --color <WHEN>      Coloring: auto, always, never
        --frozen            Require Cargo.lock and cache are up to date
        --locked            Require Cargo.lock is up to date
    -Z <FLAG>...            Unstable (nightly-only) flags to Cargo, see 'cargo -Z
help' for details
    -h, --help              Prints help information

Some common cargo commands are (see all commands with --list):
    build       Compile the current package
    check       Analyze the current package and report errors, but don't build object
files
    clean       Remove the target directory
    doc         Build this package's and its dependencies' documentation
    new         Create a new cargo package
    init        Create a new cargo package in an existing directory
    run         Run a binary or example of the local package
    test        Run the tests
```

```
    bench       Run the benchmarks
    update      Update dependencies listed in Cargo.lock
    search      Search registry for crates
    publish     Package and upload this package to the registry
    install     Install a Rust binary. Default location is $HOME/.cargo/bin
    uninstall   Uninstall a Rust binary

See 'cargo help <command>' for more information on a specific command.
```

25.3 cargo 提供的命令

正如上面 cargo 帮助信息中所描述的那样，如果我们想要查看 cargo 提供的所有命令，可以直接在终端里输入

```
cargo --list
```

输出结果如下

```
Installed Commands:
    bench                Execute all benchmarks of a local package
    build                Compile a local package and all of its dependencies
    check                Check a local package and all of its dependencies for errors
    clean                Remove artifacts that cargo has generated in the past
    clippy-preview       Checks a package to catch common mistakes and improve your
Rust code.
    doc                  Build a package's documentation
    fetch                Fetch dependencies of a package from the network
    fix                  Automatically fix lint warnings reported by rustc
    generate-lockfile    Generate the lockfile for a package
    git-checkout         Checkout a copy of a Git repository
    init                 Create a new cargo package in an existing directory
    install              Install a Rust binary. Default location is $HOME/.cargo/bin
    locate-project       Print a JSON representation of a Cargo.toml file's location
    login                Save an api token from the registry locally. If token is not
specified, it will be read from stdin.
    metadata             Output the resolved dependencies of a package, the concrete
used versions including overrides, in machine-readable format
    new                  Create a new cargo package at <path>
    owner                Manage the owners of a crate on the registry
    package              Assemble the local package into a distributable tarball
    pkgid                Print a fully qualified package specification
    publish              Upload a package to the registry
    read-manifest        Print a JSON representation of a Cargo.toml manifest.
    run                  Run a binary or example of the local package
    rustc                Compile a package and all of its dependencies
    rustdoc              Build a package's documentation, using specified custom
flags.
    search               Search packages in crates.io
```

```
    test                Execute all unit and integration tests and build examples of
a local package
    uninstall           Remove a Rust binary
    update              Update dependencies as recorded in the local lock file
    verify-project      Check correctness of crate manifest
    version             Show version information
    yank                Remove a pushed crate from the index
```

命令很多，我们就不逐一介绍了，挑几个常用的命令介绍下

命令	说明
cargo build	编译当前项目
cargo check	分析当前项目并报告项目中的错误，但不会编译任何项目文件
cargo run	编译并运行文件 src/main.rs
cargo clean	移除当前项目下的 target 目录及目录中的所有子目录和文件
cargo update	更新当前项目中的 Cargo.lock 文件列出的所有依赖
cargo new	在当前目录下新建一个 cargo 项目

如果你对某个命令不甚熟悉，可以直接使用 cargo help 语法显示命令的帮助信息。

比如要详细了解 new 命令，可以直接输入 cargo help new，输出结果如下

```
cargo-new
Create a new cargo package at <path>

USAGE:
    cargo new [OPTIONS] <path>

OPTIONS:
    -q, --quiet                 No output printed to stdout
        --registry <REGISTRY>   Registry to use
        --vcs <VCS>             Initialize a new repository for the given version
control system (git, hg, pijul, or
                                fossil) or do not initialize any version control at
all (none), overriding a global
                                configuration. [possible values: git, hg, pijul,
fossil, none]
        --bin                   Use a binary (application) template [default]
        --lib                   Use a library template
        --edition <YEAR>        Edition to set for the crate generated [possible
values: 2015, 2018]
        --name <NAME>           Set the resulting package name, defaults to the
directory name
    -v, --verbose               Use verbose output (-vv very verbose/build.rs output)
```

```
        --color <WHEN>           Coloring: auto, always, never
        --frozen                 Require Cargo.lock and cache are up to date
        --locked                 Require Cargo.lock is up to date
    -Z <FLAG>...                 Unstable (nightly-only) flags to Cargo, see 'cargo -Z
help' for details
    -h, --help                   Prints help information

ARGS:
    <path>
```

25.4 cargo 创建 Rust 项目

作为包管理器，cargo 可以帮助我们下载第三方库。但这仅仅是 cargo 功能的冰山一角。

我们还可以使用 cargo 来构建自己的 库，然后发布到 Cargo 的官方仓库中。

cargo 可以创建两种类型的项目：**可执行的二进制程序** 和 **库**。

如果要创建一个 可执行的二进制程序，可以使用下面的 cargo new 命令创建项目

```
cargo new project_name --bin
```

如果要创建一个 库，则可以使用下面的 cargo new 命令。

```
cargo new project_name --lib
```

25.5 范例： 使用 cargo 创建并构建一个完整的二进制可执行程序项目

创建项目、安装依赖、编译项目是 cargo 作为包管理器的三个最重要的功能。

下面我们就以一个简单的游戏：猜数字 来演示下如何使用 cargo 创建并编译项目。

一、 创建项目目录

打开终端或命令行提示符或 Shell 并进入到你想放项目的目录下。

然后输入命令 cargo new guess-game-app --bin，如果不出意外就会输出下面的提示

```
$ cargo new guess-game-app --bin
     Created binary (application) `guess-game-app` package
```

出现上面的提示说明项目创建成功，同时该命令会在当前目录下创建一个新的目录 guess-game-app 并创建一些必要的文件

```
$ tree guess-game-app/
guess-game-app/
├── Cargo.toml
└── src
    └── main.rs
```

通过上面的篇幅可知：cargo new 命令用于创建一个新项目，--bin 选项用于指示当前项目是一个 可执行的二进制程序 项目。

> 在 Cargo 中，通常把一个项目称之为 crates，but 我也不理解为啥...

Rust 官方为 cargo 包管理器提供了中央托管网站。网站地址为 https://crates.io/。 我们可以从该网站上找到一些我们需要的第三方库或者一些有用的第三方项目。

二、添加项目需要的第三方依赖库

我们的 **猜数字** 游戏需要生成 随机数。 但是 Rust 语言核心和 Rust 标准库都没有提供生成随机数的的方法或结构体。

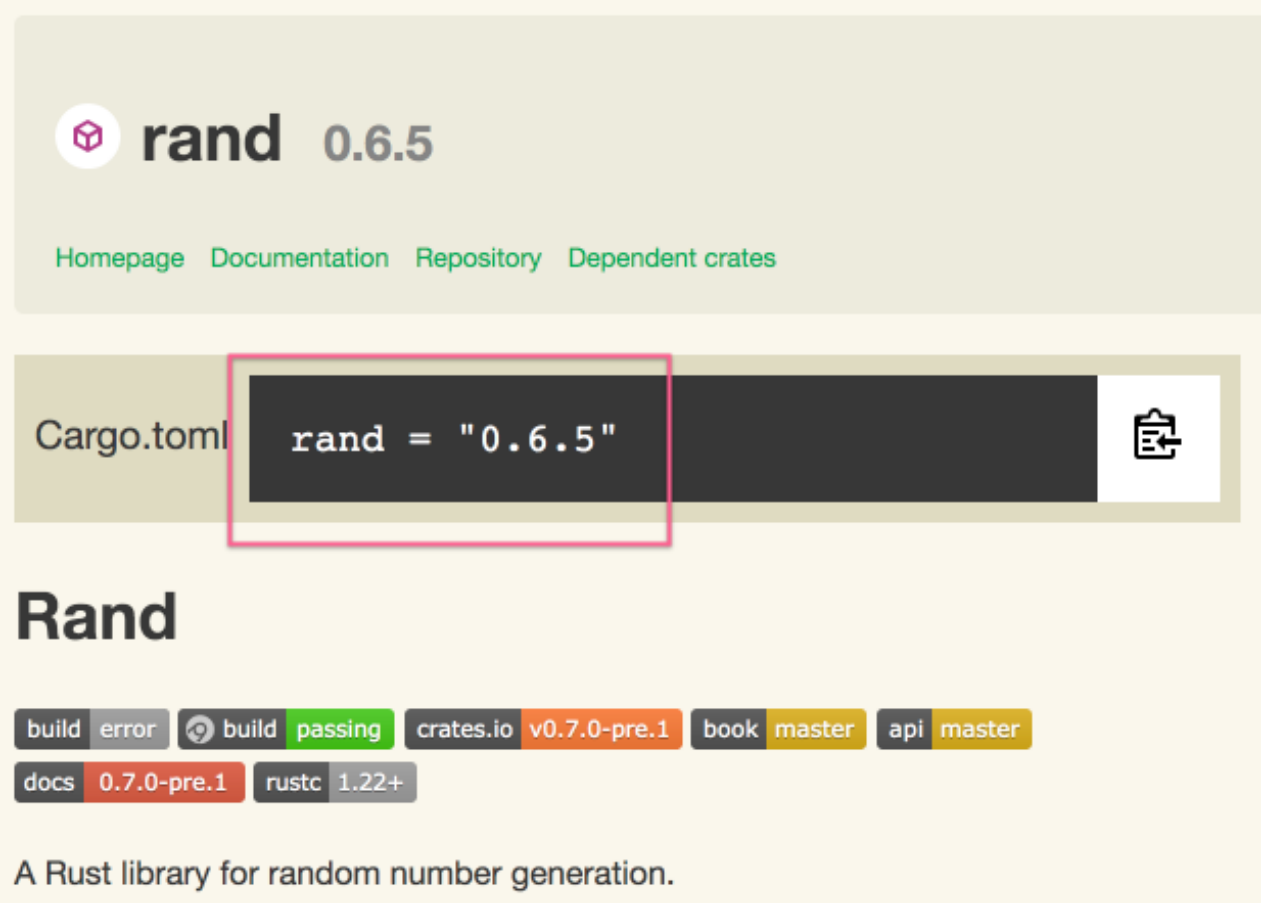
我们只能借助于 https://crates.io/ 上其它开发者提供的第三方库或 crates。

我们可以打开网站 crates.io 并在顶部的搜索中输入 rand 然后回车来查找第三方随机数生成库。

从搜索的结果来看，由很多和随机数生成的相关库，不过我们只使用 rand。

点击搜索结果中的 rand 会跳转到 https://crates.io/crates/rand。

下图是我们需要的 **随机数生成库** rand 的基本信息截图。



了解了 rand 的基本信息后，我们就可以修改 Cargo.toml 添加 rand 依赖了。

准确的说就是在 [dependencies] 节中添加 rand = "0.6.5"。

编辑完成后的文件内容如下

```
[package]
name = "guess-game-app"
version = "0.1.0"
authors = ["** <noreply@xxx.com>"]
edition = "2018"

[dependencies]
rand = "0.6.5"
```

三、 先预编译项目

对于 Rust / C / C++ 这种静态语言，先预编译一下应该是一种习惯。

因为预编译的时候会告诉我们项目配置是否正确，同时会下载第三方库，这样就可以 自动提示 了。

使用 cargo 创建的项目，我们可以在终端里输入 cargo build 来预编译下项目。

输出一般如下

```
    Updating crates.io index
  Downloaded rand v0.6.5
  Downloaded libc v0.2.58
  Downloaded rand_core v0.4.0
  Downloaded rand_hc v0.1.0
  Downloaded rand_chacha v0.1.1
  Downloaded rand_isaac v0.1.1
  Downloaded rand_jitter v0.1.4
  Downloaded rand_os v0.1.3
  Downloaded rand_xorshift v0.1.1
  Downloaded rand_pcg v0.1.2
  Downloaded autocfg v0.1.4
  Downloaded rand_core v0.3.1
   Compiling libc v0.2.58
   Compiling autocfg v0.1.4
   Compiling rand_core v0.4.0
   Compiling rand_core v0.3.1
   Compiling rand_isaac v0.1.1
   Compiling rand_xorshift v0.1.1
   Compiling rand_hc v0.1.0
   Compiling rand_pcg v0.1.2
   Compiling rand_chacha v0.1.1
   Compiling rand v0.6.5
   Compiling rand_os v0.1.3
   Compiling rand_jitter v0.1.4
   Compiling guess-game-app v0.1.0 (/Users/Admin/Downloads/guess-game-app)
    Finished dev [unoptimized + debuginfo] target(s) in 45.77s
```

从预编译输出的信息中我们可以看到：rand 第三方库和相关的依赖都会自动被下载。

四、 理解 猜数字 游戏逻辑

动手码代码之前，我们先来分析下 猜数字 游戏的游戏逻辑。

> 这是一种良好的编程习惯。

1. 游戏开始时先 随机 生成一个 数字 N。
2. 然后输出一些信息提示用户，并让用户猜并输入生成的数字 N 是多少。
3. 如果用户输入的数字小于随机数，则提示用户 Too low 然后让用户继续猜。
4. 如果用户输入的数字大于随机数，则提示用于 Too high 然后让用户继续猜。
5. 如果用户输入的数字正好是随机数，则游戏结束并输出 You go it ..

五、修改 main.rs 文件完善游戏逻辑

猜数字 游戏逻辑很简单，前面我们已经详细的分析过了，接下来我们只要实现上面的逻辑即可。

代码我们就不做做介绍了，直接复制粘贴就好

```
use std::io;
extern crate rand; // 导入当前项目下的 rand 第三方库

use rand::random;
fn get_guess() -> u8 {
   loop {
      println!("Input guess") ;
      let mut guess = String::new();
      io::stdin().read_line(&mut guess)
         .expect("could not read from stdin");
      match guess.trim().parse::<u8>(){ // 需要去除输入首尾的空白
         Ok(v) => return v,
         Err(e) => println!("could not understand input {}",e)
      }
   }
}
fn handle_guess(guess:u8,correct:u8)-> bool {
   if guess < correct {
      println!("Too low");
      false

   } else if guess> correct {
      println!("Too high");
      false
   } else {
      println!("You go it ..");
      true
   }
}
fn main() {
   println!("Welcome to no guessing game");
```

```
    let correct:u8 = random();
    println!("correct value is {}",correct);
    loop {
        let guess = get_guess();
        if handle_guess(guess,correct){
            break;
        }
    }
}
```

六、编译项目并运行可执行文件

我们可以在 终端 中输入命令 cargo run 编译并运行我们的猜数字游戏。

如果没有出现任何错误，那么我们就可以简单的来玩几盘，哈哈

```
    Compiling guess-game-app v0.1.0 (/Users/Admin/Downloads/guess-game-app)
     Finished dev [unoptimized + debuginfo] target(s) in 0.67s
      Running `target/debug/guess-game-app`
Welcome to no guessing game
correct value is 145
Input guess
20
Too low
Input guess
100
Too low
Input guess
97
Too low
Input guess
145
You go it ..
```

哈哈，其实我们是有点作弊嫌疑，因为不提示结果，要猜中得很长的逻辑

二十六、Rust 迭代器 Iterator

迭代器 主要用来 **遍历** 集合。

迭代器 就是把集合中的所有元素按照顺序一个接一个的传递给处理逻辑。

如果把集合比喻为一大缸水，那么迭代器就是水瓢。

26.1 Rust 中的迭代器

Rust 语言中的集合包括 **数组（ array ）**、**向量（ Vect! ）**、**哈希表（ map ）**等。

Rust 语言中的迭代器都要实现标准库中定义的 Iterator 特质。

Iterator 特质有两个函数必须实现：

- 一个是 iter()，用于返回一个 迭代器 对象。迭代器中存储的值，我们称之为 **项** (items) 。
- 另一个是 next()，用于返回迭代器中的下一个元素。如果已经迭代到集合的末尾（最后一个项后面）则返回 None。

Rust 语言中所有的集合都实现了 Iterator 特质。我们可以简单的使用 iter() 和 next() 方法来完成迭代

```
fn main() {

   //创建一个数组
   let a = [10,20,30];

   let mut iter = a.iter(); // 从一个数组中返回迭代器
   println!("{:?}",iter);

   //使用 next() 方法返回迭代器中的下一个元素
   println!("{:?}",iter.next());
   println!("{:?}",iter.next());
   println!("{:?}",iter.next());
   println!("{:?}",iter.next());
}
```

编译运行以上 Rust 代码，输出结果如下

```
Iter([10, 20, 30])
Some(10)
Some(20)
Some(30)
None
```

26.2 for 循环和迭代器

上面的范例中我们了解了迭代器的使用，不过有点累的是，每次都需要手动调用 next() 方法才可以获得下一个迭代项。

为了解决这种手动迭代的代码冗余，Rust 允许我们使用 for 循环来使用迭代器。

26.2.1 for 循环迭代器的语法如下

```
for iterator_item in iterator {
   // 使用迭代项的具体逻辑
}
```

26.2.2 范例

有了 for ... in 语句，我们遍历结合就比较轻松了，短短三行代码就搞定了。

```
fn main() {
   let a = [10,20,30];
   let iter = a.iter();
   for data in iter{
      print!("{}\t",data);
   }
}
```

编译运行以上 Rust 代码，输出结果如下

```
10 20 30
```

26.3 Rust 迭代器类型

Rust 中有三种类型的迭代器。

具体介绍每种迭代器之前，我们先来说说都有哪几种迭代器：

1. 既然迭代器用于遍历集合，那么在一次遍历后，集合是否还能用？这里就分两种情况。
2. 迭代器遍历集合的同时，能够修改集合中的元素？ 这里又分为两种情况了。

也就是说，遍历的迭代器有 4 种：

1. 只读遍历但不可重新遍历
2. 只读遍历但可以重新遍历
3. 可修改遍历但不可重新遍历
4. 可修改遍历但不可重入遍历

最后一种 可修改遍历但不可重入遍历感觉没啥大作用。都已经修改元素了但限制遍历，那要怎么访问啊。

剩下三种，Rust 提供了三个方法来返回。我们都罗列在下表中。

> T 表示集合中的元素。

方法	描述
iter()	返回一个只读可重入迭代器，迭代器元素的类型为 &T
into_iter()	返回一个只读不可重入迭代器，迭代器元素的类型为 T
iter_mut()	返回一个可修改可重入迭代器，迭代器元素的类型为 &mut T

26.4 范例： 只读可重入迭代器 iter()

iter() 充分体现了 Rust 中 借用 的概念。它返回的迭代器只是一个指向集合元素的引用。

因为只是引用，所以集合保持不变，并且迭代器在遍历之后还可以继续使用。

```
fn main() {
   let names = vec!["从零蛋开始教程", "简明教程", "简单编程"];
   for name in names.iter() {
      match name {
         &"简明教程" => println!("我们当中有一个异类!"),
         _ => println!("Hello {}", name),
      }
   }
   println!("{:?}",names); // 迭代之后可以重用集合
}
```

编译运行以上 Rust 代码，输出结果如下

```
Hello 从零蛋开始教程
我们当中有一个异类!
Hello 简单编程
["从零蛋开始教程", "简明教程", "简单编程"]
```

26.4.1 范例： 自动拆箱迭代 into_iter()

into_iter() 方法会返回一个自动拆箱迭代。

是不是有点拗口?

into_iter() 同 iter() 一样返回的是只读迭代，但还是有些不同的，into_iter() 充分运用了 所有权 ownership 的概念。它会把所有迭代的值从集合中移动到一个迭代器对象中。

这样，我们的迭代变量就是一个普通对象而不是对集合元素的引用。在 match 匹配时就不需要引用 & 了。

> iter_into() 之后的集合不可重用。

```
fn main(){
   let names = vec!["从零蛋开始教程", "简明教程", "简单编程"];
   for name in names.into_iter() {
      match name {
         "简明教程" => println!("我们当中有一个异类!"),
         _ => println!("Hello {}", name),
      }
   }
   // 迭代器之后集合不可再重复使用，因为元素都被拷贝走了
   //println!("{:?}",names);
   //Error:Cannot access after ownership move
}
```

编译运行以上 Rust 代码，输出结果如下

```
Hello 从零蛋开始教程
我们当中有一个异类!
Hello 简单编程.
```

26.4.2 范例：可变更集合迭代 iter_mut()

集合的 iter() 方法返回的是一个只读迭代，我们不能通过迭代器来修改集合。

如果在迭代集合的同时修改集合的元素，则需要使用 iter_mut() 方法代替 iter() 方法。

iter_mut() 方法返回的迭代元素是一个 引用类型 或者说是智能指针。我们可以通过对迭代变量 解引用的方式来重新赋值。

这种重新赋值会修改集合的原元素。

> iter_mut() 之后的集合是可以重复使用的。

```
fn main() {
   let mut names = vec!["从零蛋开始教程", "简明教程", "简单编程"];
   for name in names.iter_mut() {
      match name {
         &mut "简明教程" => { *name = "从零蛋开始教程";println!("我们中间有一个异类!")},
         _ => println!("Hello {}", name),
      }
   }

   // 集合还可以重复使用
   println!("{:?}",names);
}
```

编译运行以上 Rust 代码，输出结果如下

```
Hello 从零蛋开始教程
我们中间有一个异类!
Hello 简单编程
["从零蛋开始教程", "从零蛋开始教程", "简单编程"]
```

二十七、Rust 闭包 Closure

闭包（ Closure ） 的出现其实是程序员偷懒的结果，也可以说是语言开发者为程序员送的福利。

为什么这么说呢？

我们来看看几个 **闭包** 的解释：

- 闭包就是在一个函数内创建立即调用的另一个函数。
- 闭包是一个匿名函数。也就是没有函数名称。

- 闭包虽然没有函数名，但可以把整个闭包赋值一个变量，通过调用该变量来完成闭包的调用。从某些方面说，这个变量就是函数名的作用。
- 闭包不用声明返回值，但它却可以有返回值。并且使用最后一条语句的执行结果作为返回值。闭包的返回值可以赋值给变量。
- 闭包有时候有些地方又称之为 **内联函数**。这种特性使得闭包可以访问外层函数里的变量。

从上面的描述中可以看出，闭包就是函数内部的一个没有函数名的内联函数。对于那些只使用一次的函数，使用闭包是最佳的代替方案。

27.1 定义闭包的语法

在介绍如何定义闭包前，我们先来看看 如何定义一个普通的函数。

一个普通的函数的定义语法格式如下

```
fn function_name(parameters) -> return_type {
   // 函数的具体逻辑
}
```

从上面的描述中我们知道，闭包是一个没有函数名的内联函数。它的定义语法如下

```
|parameter| {
   // 闭包的具体逻辑
}
```

从语法格式上来看，闭包就是普通函数去掉 fn 关键字，去掉函数名，去掉返回值声明，并把一对小括号改成一对 竖线 ||。

闭包的参数是可选的，如果一个闭包没有参数，那么它的定义语法格式如下

```
||{
   // 闭包的具体逻辑
}
```

闭包虽然没有名称，但我们可以将闭包赋值给一个变量，然后就可以通过调用这个变量来完成闭包的调用。

```
let closure_function = |parameter| {
   // 闭包的具体逻辑
}
```

因为调用闭包的语法实现了 Fn 特质，因此我们可以像调用普通函数那样，使用小括号 () 来调用闭包

```
closure_function(parameter);    //invoking
```

27.1.1 范例1：普通的闭包

下面的代码我们定义了一个闭包用于判断传递的参数是否偶数，并把闭包赋值给变量 is_even。

然后我们通过调用 is_even() 来完成闭包的调用。

```
fn main(){
   let is_even = |x| {
      x%2==0
   };
   let no = 13;
   println!("{} is even ? {}",no,is_even(no));
}
```

编译运行以上 Rust 代码，输出结果如下

```
13 is even ? false
```

27.1.2 范例2：闭包使用外部函数可以访问的变量

闭包还可以访问它所在的外部函数可以访问的所有变量。

```
fn main(){
   let val = 10;
   // 访问外层作用域变量 val
   let closure2 = |x| {
      x + val // 内联函数访问外层作用域变量
   };
   println!("{}",closure2(2));
}
```

编译运行以上 Rust 代码，输出结果如下

```
12
```

二十八、Rust 智能指针

Rust 是一门系统级的语言，至少，它自己是这么定位的，它所对标的语言是 C++。

作为系统级别的语言，抛弃指针是完全不现实的，提供有限的指针功能还是能够做得到的。

Rust 语言是一门现代的语言。一门现代语言会尽可能的抛弃指针，也就是默认会把所有的数据都存储在栈 上。

如果要把数据存储在 **堆** 上，就要在 **堆** 上开辟内存，这时候就要使用到 **指针**。

作为系统级的语言， Rust 提供了在 堆 上存储数据的能力。只不过它把这些能力弱化并封装到了 **Box** 中。

这种把 **栈** 上数据搬到 **堆** 上的能力，我们称之为 **装箱**。

Rust 语言中的某些类型，如 **向量 Vector** 和 **字符串对象 String** 默认就是把数据存储在 **堆** 上的。

Rust 语言把指针封装为以下两大 **特质 trait**。当一个结构体实现了下面的接口后，它们就不再是普通的结构体了。

特质名	包	Description
Deref	std::ops::Deref	用于创建一个只读智能指针，例如 *v
Drop	std::ops::Drop	智能指针超出它的作用域范围时会回调该特质的 **drop()** 方法，类似于其它语言的 析构函数

本章节，我们就来学习 Rust 中那功能少的可怜的智能指针，准确的说是学习 **Box** 这个智能指针装箱器。

28.1 Box 指针

Box 指针也称之 **装箱（ box ）**，允许我们将数组存储在 **堆 （ heap ）** 上而不是 **栈（ stack ）** 上。

但即使把数据存储在 **堆 （ heap ）** 上，**栈（ stack ）** 仍然包含了指向**堆数据的指针**。

Box 指针没有任何额外的其它开销，因为它仅仅只是把数据存储在 **堆 （ heap ）** 而已。

说起来很拗口，我们直接就看代码

```
fn main() {

   let var_i32 = 5;           // 默认数组保存在 栈 上
   let b = Box::new(var_i32); // 使用 Box 后数据会存储在堆上
   println!("b = {}", b);

}
```

编译运行上面的 Rust 代码，输出结果如下

```
b = 5
```

28.1.1 访问 Box 指针存储的数据

当我们使用 Box::new() 把一个数据存储在堆上之后，为了访问存储的具体数据，我们必须 **解引用**。

解引用 需要使用操作符 **星号**，因此 **星号** 也称之为 **解引用操作符**。

这一点和 C++ 一样的。

下面这段代码，为了访问数据 y，我们需要使用 *y 。

```
fn main() {
   let x = 5;            // 值类型数据
   let y = Box::new(x); // y 是一个智能指针，指向堆上存储的数据 5

   println!("{}",5==x);
   println!("{}",5==*y); // 为了访问 y 存储的具体数据，需要解引用
}
```

编译运行上面的 Rust 代码，输出结果如下

```
true
true
```

上面的代码中，因为 5 是一个基础数据类型，所以当使用 5 == x 的时候会返回 true，因为基础类型只会比较值相同与否。

而另一个变量 y，它是一个智能指针，是一个引用类型，直接使用 5 == y 会返回 false。为了访问 y 指向的具体的值，我们需要对 y 解引用。

28.1.2 Deref Trait

Deref 是由 Rust 标准库提供的一个 **特质 (trait)**。

实现 Deref 特质需要我们实现 deref() 方法。

deref() 方法从某些方面说用于借用 self 对象并返回一个指向内部数据的指针。

也就是说 deref() 方法返回一个指向结构体内部数据的指针。

28.1.3 范例

下面的代码有点长，我们的范型结构体 **MyBox** 实现了 **Deref** 特质。

我们可以通过 **dedef()** 方法返回的结构体实例的引用来访问 **堆 heap** 上的数据。

```
use std::ops::Deref;

struct MyBox<T>(T);

impl<T> MyBox<T> {
   // 范型方法
   fn new(x:T)-> MyBox<T> {
      MyBox(x)
   }
}

impl<T> Deref for MyBox<T> {
   type Target = T;
   fn deref(&self) -> &T {
      &self.0 // 返回数据
```

```
    }
}

fn main() {
    let x = 5;
    let y = MyBox::new(x);  // 调用静态方法 new() 返回创建一个结构体实例

    println!("5==x is {}",5==x);
    println!("5==*y is {}",5==*y);  // 解引用 y
    println!("x==*y is {}",x==*y);  // 解引用 y
}
```

编译运行上面的 Rust 代码，输出结果如下

```
5==x is true
5==*y is true
x==*y is true
```

28.1.4 删除特质 Drop Trait

> Drop Trait 我将它翻译为 删除特质 ， 但总感觉怪怪的。

> Drop Trait 翻译的有点坑爹，因为我不知道要如何翻译才能确切的表达那个意思。

Drop Trait 只有一个方法 **drop()**。

当实现了 Drop Trait 的结构体在离开了它的作用域范围时会触发调用 drop() 方法。

一些其它语言中，比如 C++，智能指针每次使用完了之后都必须手动释放相关内存或资源。

而在 Rust 语言中，我们可以把释放内存和资源的操作交给 Drop trait。

具体的，我们直接看代码就好

```
use std::ops::Deref;

struct MyBox<T>(T);

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

impl<T> Deref for MyBox<T> {
    type Target = T;
        fn deref(&self) -< &T {
        &self.0
    }
}
```

```
impl<T> Drop for MyBox<T>{
   fn drop(&mut self){
      println!("dropping MyBox object from memory ");
   }
}
fn main() {
   let x = 50;
   MyBox::new(x);
   MyBox::new("Hello");
}
```

编译运行上面的 Rust 代码，输出结果如下

```
dropping MyBox object from memory
dropping MyBox object from memory
```

输出两次结果是因为我们在 **堆（ heap ）** 上创建了两个对象。

二十九、Rust 多线程并发编程

随着电脑等电子产品全面进入多核时代，并发编程已经是程序不可或缺的功能之一。

并发编程就是同时运行两个或多个任务，就像宅男宅女的我们，一边吃零食还能一边吃饭，顺带还能调侃下男女朋友。

并发编程的一个重要思想就是 **程序不同的部分可以同时独立运行互不干扰**。

29.1 多线程

现代的操作系统，是一个多任务操作系统，系统可以管理多个程序的运行，一个程序往往有一个或多个进程，而一个进程则有一个或多个线程。

让一个进程可以运行多个线程的机制叫做多线程编程。

一个进程一定有一个主线程，主线程之外创建出来的线程称之为 **子线程**

多线程编程，其实就是在主线程之外创建子线程，让子线程和主线程同时运行，完成各自的任务。

Rust 语言支持多线程并发编程。

29.2 创建线程

Rust 语言标准库中的 std::thread 模块用于支持多线程编程。

std::thread 提供很很多方法用于创建线程、管理线程和结束线程。

创建一个新线程，可以使用 std::thread::spawn() 方法。

spawn() 函数的原型如下

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
```

参数 f 是一个闭包（closure ） 是线程要执行的代码。

29.2.1 范例

下面的范例，我们使用 spawn() 函数创建了一个线程，用于输出数字 1 到 10

```
use std::thread;           // 导入线程模块
use std::time::Duration;   // 导入时间模块

fn main() {
   //创建一个新线程
   thread::spawn(|| {
      for i in 1..10 {
         println!("hi number {} from the spawned thread!", i);
         thread::sleep(Duration::from_millis(1));
      }
   });
   // 主线程要执行的代码
   for i in 1..5 {
      println!("hi number {} from the main thread!", i);
      thread::sleep(Duration::from_millis(1));
   }
}
```

编译运行以上 Rust 范例，输出结果如下

```
hi number 1 from the main thread!
hi number 1 from the spawned thread!
hi number 2 from the main thread!
hi number 2 from the spawned thread!
hi number 3 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 4 from the main thread!
```

咦，执行结果好像出错了？ 是吗?

嗯，好像是错了，但好像又不是。

> 注意: 当主线程执行结束，那么就会自动关闭创建出来的衍生子线程。

上面的代码，我们调用 **thread::sleep()** 函数强制线程休眠一段时间，这就允许不同的线程交替执行。

虽然某个线程休眠时会自动让出 cpu，但并不保证其它线程会执行。这取决于操作系统如何调度线程。

这个范例的输出结果是随机的，主线程一旦执行完成程序就会自动退出，不会继续等待子线程。这就是子线程的输出结果为什么不全的原因。

29.3 加入线程句柄 join()

默认情况下，主线程并不会等待子线程执行完毕。如果主线程创建完衍生线程就立即退出的话，那么子线程可能根本没机会开始运行或者执行完毕，

为了避免这种情况，我们可以让主线程等待子线程执行完毕然后再继续执行。

Rust 标准库提供了 **join()** 方法用于把子线程加入主线程等待队列。

join() 方法的原型如下

```
spawn<F, T>(f: F) -> JoinHandle<T>
```

29.3.1 范例

下面的范例，我们使用 **join()** 方法把衍生线程加入主线程等待队列，这样主线程会等待子线程执行完毕才能退出。

```
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();
}
```

编译运行以上 Rust 范例，输出结果如下

```
hi number 1 from the main thread!
hi number 1 from the spawned thread!
hi number 2 from the spawned thread!
hi number 2 from the main thread!
hi number 3 from the spawned thread!
hi number 3 from the main thread!
hi number 4 from the main thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!
```

从输出结果来看，主线程和子线程交替执行。

> 注意: 主线程会等待子线程执行完毕是因为调用了 join() 方法。