

03 基础概念

在上一次课中，我们一起完成了一个猜数字的小游戏。可能对于很多同学来说，Rust 中的一些概念还比较陌生。不过没有关系，这次课我们将详细讲解 Rust 中的一些基础概念。可能这一节的内容略显枯燥，不过无需担心，也无需着急，毕竟 Rust 是一门挺特殊的语言，我们一起来慢慢的把基础知识夯实，为将来的快速提升打好基础。

变量及其可变性

在 Rust 中，定义一个变量需要使用 `let` 关键字，例如：`let x = 5;`。如果和 JavaScript 类比的话，Rust 中的 `let` 大致对应 JavaScript 中的 `const`。但是 Rust 的情况更复杂，`let x = 5;` 这一句代码的含义有：

1. 创建变量 `x`
2. 创建值 `5`
3. 将值 `5` 绑定到变量 `x` 上
4. 变量 `x` 对应的值不可以修改

实际上在其他的语言中，也是比较类似的过程。但是我们在使用其他语言编写代码的时候，很少强烈的、显式的区分“值”（Value）和“变量”（Variable）。但是在使用 Rust 编写代码的时候，内心一定要明确区分“值”和“变量”，这对于后续理解所有权、引用、借用这些概念会有很大的帮助。

使用 `let` 声明的变量的值是不可变的（immutable。这里“变量”这个词就略显尴尬了），所以前面说到 Rust 中的 `let` 大致对应 JavaScript 中的 `const + Object.freeze()`，后面我们会详细讲解到 Rust 中关于可变性的约束，现在可以简单理解，如果一个“对象”不是可变的，那么对象的属性的值也不可以修改。例如，下面的代码将无法通过编译：

```

1 // 以下代码无法通过编译
2 fn main() {
3     let x = 5;
4     println!("x is: {x}");
5     x = 6;
6     println!("x is: {x}");
7 }
```

如果要声明其值可变的变量，需要使用 `mut` 关键字（`mutable`），例如：

```

1 fn main() {
2     let mut x = 5;
3     println!("x is: {x}");
4     x = 6;
5     println!("x is: {x}");
6 }
```

那么，Rust 中的 `let mut` 大致对应 JavaScript 中的 `let` 了。许多现代编程语言在声明变量时都倾向于明确变量值的可变性，例如：Scala、Swift 等。JavaScript 在 ES6 中也加入了 `let` 和 `const` 关键字来定义变量，给 JavaScript 注入了块级作用域的能力，同时，也鼓励大家在定义变量的时候使用 `let` 和 `const` 来替代 `var`。

在 Rust 中，同样也提供了“常量”的概念，使用 `const` 关键字来声明常量。注意，这里的 `const` 和 JavaScript 中的 `const` 可是很不一样的。在 Rust 中使用 `const` 声明常量：

```
1 const THREE_HOURS_IN_SECONDS: u32 = 60 * 60 * 3;
```

Rust 中的常量有以下限制：

1. 不可以对常量使用 `mut` 关键字
2. 常量必须明确数据类型。定义常量时，不能使用类型推断的能力
3. 常量可以在任意范围声明：全局常量、函数内常量
4. 常量只能绑定到一个常量表达式，也就是说在编译期间就可以知道对应的值的，不可依赖运行期的变量表达式或者值

在 Rust 中，定义变量或者常量的时候，如果需要指定数据类型，其格式是在变量或者常量后跟 `: type`。回顾一下，在上次课中的猜数字小游戏中，有这么一行：`let guess: u32 = guess.trim().parse().expect("...");`，就是采用显示指定类型的。下面还有一些示例：

```

1 const AUTHOR_NAME: &str = "Yuan Yu"; // 全局常量，其类型是 &str
2 fn main() {
3     const PI: f32 = 3.1415926; // 局部常量，类型为 f32
4     let x: i32 = -1;           // 不可变变量，类型为 i32
5     let y: u32 = 100;          // 不可变变量，类型为 u32
6     let mut s: String = String::new(); // 可变变量，类型为 String
7 }
```

数据类型

Rust 是一种静态类型（Static Typing）语言，每个值都有对应的数据类型。从数据类型形态而言，编程语言可以分为静态类型语言和动态类型（Dynamic Typing）语言。

静态类型语言： 在静态类型语言中，变量的类型在编译时就已经确定，并且在整个程序执行过程中不会改变。这种类型的检查可以在编译阶段完成，因此可以在程序运行之前发现类型错误。静态类型语言的例子包括 C、C++、Java、Swift、Rust 等。

静态类型的优点有：

1. 编译时就能检测到类型错误，提高了代码的稳定性。
2. 通常性能更好，因为类型检查已经在编译时完成，无需在运行时进行。
3. 代码更易于维护，因为每个变量的类型都明确指出，有助于理解代码逻辑。

动态类型语言： 在动态类型语言中，变量的类型是在运行时确定的，变量的类型可以在运行时改变。这种语言通常更加灵活，易于快速编程。动态类型语言的例子包括 Python、Ruby、JavaScript 和 PHP。

动态类型语言的优点：

1. 编程灵活性高，容易写出简洁的代码。

2. 更容易进行快速开发和原型设计。
3. 动态类型系统支持一些高级特性，例如运行时类型变更、强大的反射功能等。

总结来说，静态类型语言和动态类型语言各有利弊，适用于不同的开发情景。静态类型语言更适合大型项目和需要高性能的应用，而动态类型语言则更适合快速开发和脚本任务。选择哪种类型的语言取决于项目的具体需求、团队的技术栈和预期的软件质量等因素。

Rust 的类型系统分为两类：标量类型（Scalar Type）和复合类型（Compound Type）。

标量类型

Rust 中的标量类型包含：

1. 整数，integer
2. 浮点数，floating-point
3. 布尔，boolean
4. 字符，character

整数

从符号为分类，Rust 中的整数分为有符号整数（`i`，signed）和无符号整数（`u`，unsigned）。有符号整数的值范围是 $-(2^{n-1})$ 到 $2^{n-1} - 1$ ，无符号整数的值的范围是 0 到 $2^n - 1$ 。JavaScript 中的数值类型不区分有符号的和无符号的，但是 `TypedArray` 中有区分符号的，例如：`Int8Array` 和 `Uint8Array`。

根据数值在内存中占用的字节长度不同又可以分为：

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

arch 长度是指在 32 位硬件和操作系统上是 32-bit，在 64 位硬件和操作系统上是 64-bit。如果不考虑嵌入式开发的话，现在大部分的电脑都是 64-bit 的硬件且安装 64-bit 的操作系统。

在 JavaScript ES11 中加入了 `BigInt` 类型，用来安全的处理 $2^{53} - 1$ 以上的整数。在 Rust 标准库中没有对应的数据类型，但是第三方库 `num-bigint` 提供了这种能力。

在实际的项目中，具体应该采用哪种类型的整数，应该结合当前业务情况并充分推演未来 5 年数据的增长变化。另外，如果涉及到网络通信的，还需要考虑通信双方或者多方所使用的编程语言对符号位的支持能力。

整数字面量

整数字面量（literal），支持以下方式定义整数值并绑定变量。可以显示的在变量后面跟上 `: 类型`，也可以在字面量的数值后面跟上类型或者 `_类型`：

```

1 fn main() {
2     let a: u32 = 1_234_567; // 十进制，直接写。支持使用下划线做千分符
3     let a1      = 1_234_567u32;
4     let a2      = 1_234_567_u32;
5     let b: u8  = 0xFF;           // 十六进制，0x 开头
6     let b1      = 0xFFu8;
7     let b2      = 0xFF_u8;
8     let c: i8  = 0o77;          // 八进制，0o 开头
9     let c1      = 0o77i8;
10    let c2     = 0o77_i8;
11    let d: u8  = 0b1111_0000; // 二进制，0b 开头

```

```

12     let d1      = 0b1111_0000u8;
13     let d2      = 0b1111_0000_u8;
14     let e: u8   = b'A';           // 字节, 仅支持 u8 类型
15 }
```

JavaScript 代码

```

1 function test() {
2     const a = 1_234_567; // ECMAScript 2021(ES12) 中支持下划线作为千
3     分符号分隔数值字面量
4     const b = 0xFF;
5     const c = 0o77;
6 }
```

整数类型转换

在实际开发中，经常会遇到从一种整数类型转换到另外一种整数类型的情况，此时我们可以使用关键字 `as` 来进行转换。从一个较小位宽的整数转换到较大位宽的整数总是安全的，例如，从 `u8` 转换到 `u16`，从 `f32` 转换到 `f64`。

```

1 assert_eq!( 10_i8 as u16, 10_u16);
2 assert_eq!( -1_i16 as i32, -1_i32);
```

如果从一个较大位宽的整数向较小位宽的整数转换时，可能会发生截断（truncation）。例如：

```

1 assert_eq!( 1000_i16 as u8, 232_u8);
2 assert_eq!(65535_u32 as i16, -1_i16);
3 assert_eq!( -1_i8 as u8, 255_u8);
4 assert_eq!( 255_u8 as i8, -1_i8);
```

数学运算及溢出处理

Rust 中整数类型除了支持使用 `+`, `-`, `*`, `/` 等运算符号进行数值计算之外，还提供了对应的一些方法来处理整数运算中的溢出问题。

Checked

Checked 操作的返回值是一个 `Option` 类型，如果运算没有溢出，则是 `Some(v)`，如果运算溢出了，则返回 `None`。

```
1 10u8.checked_add(100); // => Some(110)
2 128u8.checked_add(200); // => None
3 128u8.checked_mul(3); // => None
```

Wrapping

Wrapping 操作针对没有发生溢出的结果，返回对应的值；如果发生溢出了，则返回结果对 2^N 取模（余数），其中 N 是预期位宽。

```
1 100_u16.wrapping_mul(200); // => 20000
2 500_u16.wrapping_mul(500); // => 50000
3 900_u16.wrapping_mul(900); // => 23568 810000 % 2^16 => 23568
```

Saturating

Saturating 操作针对溢出的结果，“钳制”到对应位宽可以表达的最大或者最小的值。

```
1 32760_i16.saturating_add(10); // => 32767
2 (-32760_i16).saturating_sub(10); // => -32768
```

Overflowing

Overflowing 操作返回一个 2 元 Tuple 类型，第一个（下标为 0 的）表示 *Wrapping* 后的结果，第二个（下标为 1 的）元素是一个布尔值，表示是否发生了溢出。

```
1 255_u8.overflowing_sub(2); // => (253, false)
2 255_u8.overflowing_add(2); // => (1, true)
```

浮点数

在 Rust 中，浮点数分为 `f32` 单精度浮点数和 `f64` 双精度浮点数。其中 `f64` 可以对应 JavaScript 中的 `Number` 类型。

`f32` 可表示的数值类型约为 -3.4×10^{38} 至 $+3.4 \times 10^{38}$ ，`f64` 可表示的数值类型约为 -1.8×10^{308} 至 $+1.8 \times 10^{308}$ 。

浮点数字面量

```

1 fn main() {
2     let pi = 3.14; // 如果不特殊指定，浮点数字面量对应的是 f64
3     let x = 3.14f32;
4     let y: f32 = 3.14;
5     let x: f64 = 6.28;
6 }
```

数值类型中的常量

类似于 JavaScript 的 `Number` 类型中有 `Number.MAX_VALUE`, `Number.MIN_VALUE`, `Number.NaN` 等常量，Rust 的数值类型中也有对应的常量。

- 整数类型都有 `MAX`, `MIN`, `BITS` 常量。例如类型 `u32` 中：`pub const BITS:u32 = 32u32;`
- 浮点数类型 `f32` 和 `f64` 中有更多的常量。例如类型 `f64` 中有 `NAN`、`INFINITY` 等

数值类型运算符及方法

- `+` 加法运算
- `-` 减法运算
- `*` 乘法运算
- `/` 除法运算。注意，Rust 中两个整数相除，实际上执行的是整除。例如：`1/3` 的结果是 `0`, `4/3` 的结果是 `1`

- `%` 取模（取余数）
- `f32::sqrt()`, `f64::sqrt()`。目前稳定版 Rust 没有提供针对整数的平方根函数
- `pow()` 计算整数的幂
- `f32::powf()`, `f64::powf()` 浮点数的小数次幂
- `f32::powi()`, `f64::powi()` 浮点数的整数次幂
- `abs()` 取绝对值
- `round()` 四舍五入
- `ceil()` 向上取整
- `floor()` 向下取整

布尔类型

布尔类型很简单，和 JavaScript 中的很类似，只有 2 个值：`true` 和 `false`。

```
1 let t: bool = true;
2 let f: bool = false;
```

布尔值可以通过 `as` 关键字转换成整数，例如：

```
1 fn main() {
2     let t = true;
3     let b = 1 + (t as u8);
4     println!("b = {b}");
5 }
```

字符类型

Rust 中的字符类型是使用一对儿单引号（' '）引起来的单个字符，它代表这个字符的 Unicode 值，对应的是一个 32-bit 的整数值，也就是说，一个 `char` 类型占用 4 个字节，取值范围为：`0x0000 - 0xD7FF` 以及 `0xE000 - 0x10FFFF`。

`char` 类型使用反斜线 (\) 作为转义字符，这一点和很多其他语言是一致的。字符类型的字面量示例如下：

```

1 fn test_char() {
2     let c1 = 'c';
3     let c2 = '\u{1F60A}'; // 玫瑰花字符
4     let c3 = '\\\\';
5     let c4 = '\u{2603}\u{FE0F}'; // 带有变体符的哭脸表情
6     let c5 = '\x2A'; // ASCII 字符可以使用 \xHH 格式的字面量
7     let c6 = '\u{CA0}'; // 也可以使用 \u{HHHHHHH} 字面量表示任意的
8     Unicode 字符
9
10    println!("{}{}, {}{}, {}{}, {}{}, {}{}");
```

聪明的你可能已经猜到了，`char` 类型也可以使用 `as` 关键字转换到整数类型的。那么，请思考一下：使用哪种位宽的整数接收任意的 `char` 类型是安全的？

复合类型

元组

Rust 中的元组 (Tuple, ['tjʊpəl], ['tʌpəl/]) 用来将一系列值组合到一起，这些值可以是不同类型的。元组的长度一旦确定了，就不可以更改了。现在很多语言中都提供了元组支持，在某些场景下可以省去了定义复杂类型的麻烦。

元组使用一对儿圆括号 (()) 括起来，使用逗号 (,) 分隔不同的值。不包含任何值的元组称为单元元组 (`unit`)。使用下标访问元组内的值：`tul.0`。

```

1 let tup = (500, 6.4, 1);
2 let five_hundred = tup.0;
3 let six_point_four = tup.1;
4 let one = tup.2;
```

数组

Rust 中的数组是用来将相同类型的一组值组合放到一起的类型。和 JavaScript 中的 Array 不同之处在于：

1. Rust 中数组的长度是固定的，一旦声明则不可修改。JavaScript 中的数组长度可变
2. Rust 中的数组中的值必须是相同类型。JavaScript 的数组中可以存放任意类型
3. 实际上，JavaScript 中的数组更接近 Rust 中的 Vec（后面课程会讲到）

数组中值的访问方式很容易猜到：`[index]`，并且 `index` 是从 `0` 开始的。

```

1 // 长度为 12, 值类型为 &str 的数组
2 let months = ["January", "Feburary", "March",
3                 "April", "May", "June",
4                 "July", "August", "September",
5                 "October", "November", "December"];
6
7 // 长度为 5, 值类型为 i32 的数组
8 let a: [i32; 5] = [1, 2, 3, 4, 5];
9
10 // 长度为 5, 每个元素的初始值都为 3 的数组
11 let b = [3; 5];

```

其他常用类型

切片

切片（slice）类型是比较少见的，至少在 JavaScript 中没有对应的类型。但是，在 JavaScript `Array.prototype.slice()` 函数和含义和 Rust 的 Slice 是几乎一致的。

JavaScript `Array.prototype.slice()`

The `slice()` method of `Array` instances returns a *shallow copy* of a portion of an array into a new array object selected from `start` to `end` (`end` not included) where `start` and `end` represent the index of items in that array. The original array will not be modified.

熟悉 JavaScript 的同学，可以快速的掌握 Rust 中切片的精要：就是一组值中的某一段的“浅拷贝”，在 Rust 中是“引用”。

`[T]` 表示类型 `T` 切片，和数组很相似，但是它没有长度！切片用来表示数组或者向量中的一段数值，所以他的长度是不固定的，不可以直接将 `[T]` 类型的值和某个变量绑定，也不可以将 `[T]` 类型的变量作为参数来调用函数。看到这里，你一定很迷惑：这个切片简直就是毫无用处嘛！不是这样的，我们只需使用引用（关于引用的相关知识在后续章节会详细讲解）方式使用切片就非常方便了：`&[T]`。

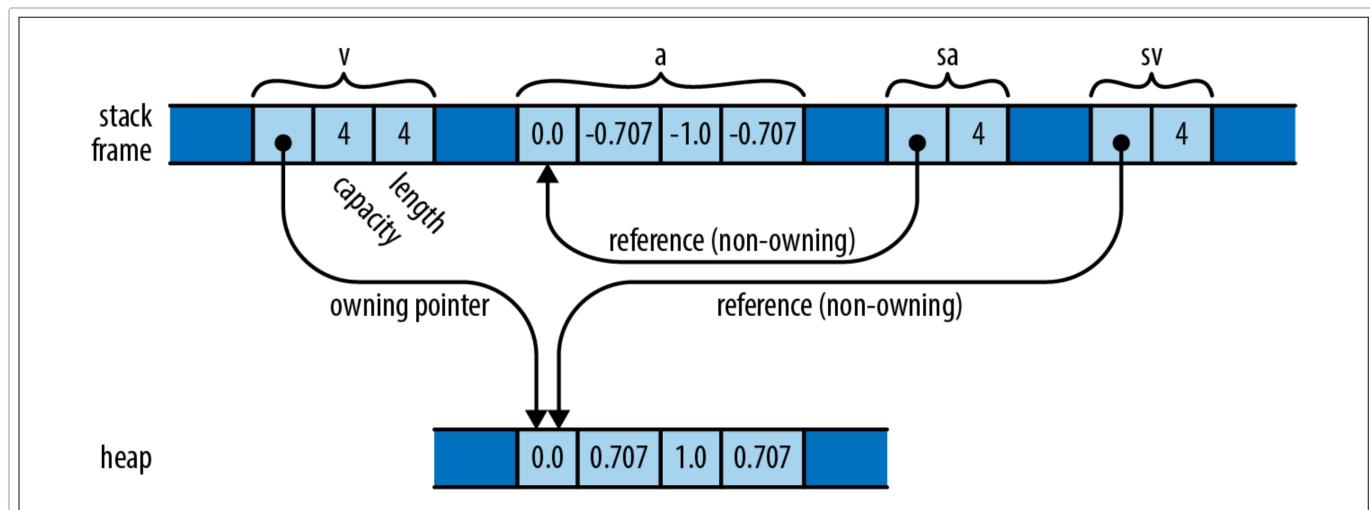
Rust 中的切片属于胖指针（fat pointer），它记录了所指向的第一个元素的位置，以及元素的个数。例如，下面这段代码：

```

1 // v 是一个 Vec 类型的值，后面会讲解到 Vec
2 let v: Vec<f64> = vec![0.0, 0.707, 1.0, 0.707];
3
4 let a: [f64; 4] = [0.0, 0.707, 1.0, 0.707];
5 let sv: &[f64] = &v;
6 let sa: &[f64] = &a;

```

切片类型在内存中的布局示意入下图：



在上图中，我们定义了 2 个变量：`v` 和 `a`，分别绑定到向量类型的值和数组类型的值。然后使用变量 `sa` 持有数组类型值的一个切片引用，使用变量 `sv` 持有一个向量类型值的引用。所以，`sa` 和 `sv` 不仅记录了所指向的第一个值的地址，还记录了所指向的这个片段的长度。

上图中提到了栈和堆的概念，如果你对这部分概念不是很熟悉，不用着急，下节课我们将深入讲解栈和堆。

字符串相关类型

Rust 中，字符串相关的类型有 2 个，分别是 `str` ([stə]) 和 `String`。其中，`str` 类似于切片，总是通过引用的方式使用。例如：`let s = "Hello world!"` 中的 `s` 的类型是 `&str`。

`&str` 的字面量表示

`&str` 的字面量表示和 JavaScript 中的字符串非常相似，使用一对儿双引号（`" "`）引起起来的字符串值。

```

1 // 单行文本
2 let s1 = "Hello world!";
3 println!("s1 = {s1}");
4
5 // 使用反斜线 \ 转义字符
6 let s2 = "\\"Ough\\" said the well.\n";
7 println!("s2 = {s2}");
8
9 // 多行文本，实际的字符串中包含换行符和和空格
10 let s3 = "Hello!
11     My name is LiLei";
12 println!("s3 = {s3}");
13
14 // 多行文本，使用反斜线 \ 拼接多行。行尾的换行符和下一行首的空格都不保留
15 let s4 = "Hello! \
16     My name is LiLei";
17 println!("s4 = {s4}");

```

```

18 // r"" 的表示 raw 字符串，其中的字符都将保留，不进行转义
19 let s5 = r"C:\Users\john_smith\Applications";
20 println!("s5 = {s5}");
21
22
23 // 由于 r"" 格式的 raw 字符串不进行转义，
24 // 所以无法在 raw 字符串中使用双引号字面量。
25 // 如果需要在 raw 格式的字符串中包含双引号字符，
26 // 那么可以使用 r###"### 格式的字符串，如下所示：
27 // r###" 开始，直到检测到 "### 结束
28 let s6 = r###"
29     This raw string starts with 'r###'
30     Therefore it does not end until we reach a quote mark ('')
31     followed immediately by three pound signs ('###'):
32 "###;
33 println!("s6 = {s6}");

```

String 类型

Rust 中的 `String` 类型实际上是在内存中一组连续的 Unicode 值，但它并不是采用一系列 `char` 类型值来存储的，它实际上采用的是 UTF-8 编码来存储的。

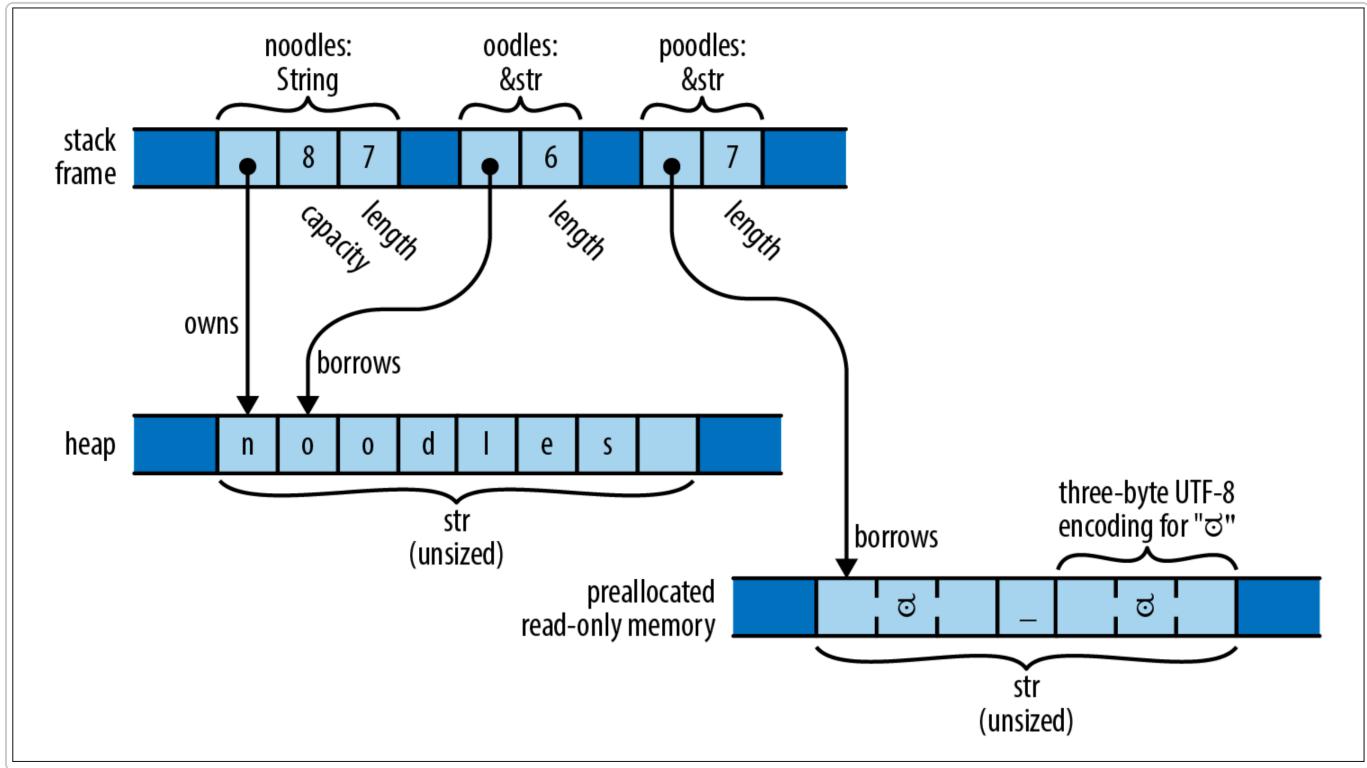
例如，下面一段：

```

1 // 创建一个类型为 String，内容为 "noodles" 的值,
2 // 并绑定到变量 noodles 上
3 let noodles = "noodles".to_string();
4
5 let oodles = &noodles[1..];
6 let poodles = "ಠ_ಠ";

```

在内存中的表示如下：



`noodles` 绑定到一个 `String` 类型的值，对应的内容是 `noodles`，Rust 在堆上开辟一块连续的空间，将这几个字符存储进去，然后 `noodles` 变量在栈上分配 24 个字节（64-bit）的空间，其中 8 个字节指向堆内存中起始字符的地址，8 个字节存储字符串的容量（`capacity`），8 个字节表示当前字符串的长度（`length`）。

变量 `oodles` 实际上是指向 `noodles` 的一个切片，所以它是一个 `&str` 类型。记录了指向的第一个字符以及当前切片的长度。

`poodles` 绑定到一个 `&str` 类型的值。由于我们使用的字面量的形式，所以这部分的字符是在编译的时候带入到可执行程序中的，`poodles` 指向首字符地址，以及记录字符串的长度。

len() 方法

`String` 和 `&str` 类型，都有 `len()` 方法。虽然看起来和 JavaScript 中 `String.length` 属性很相似，但是注意，这里的含义是不同的。JavaScript 中是以连续的 UTF-16 代码单元存储字符串的，`String.length` 属性返回的是 UTF-16 的代码单元数量；而 Rust 的 `String` 和 `&str` 是以 UTF-8 编码存储字符串的，`.len()` 方法返回的是 UTF-8 编码之后的字节数量。

比如: `e` 这个字符, 它的 Unicode 编码是 `0x1D452`, 用 UTF-16 表示的时候, 需要 2 个编码单元 (code unit。每个编码单元占 2 个字节), 用 UTF-8 表示需要 4 个字节。所以在 JavaScript 中 `length` 属性和 Rust 中的 `len()` 方法的值是不一样的。

JavaScript 代码

```
apple ~ % node
Welcome to Node.js v18.19.0.
Type ".help" for more information.
> 'e'.length
2
```

JavaScript

Rust 代码

```
apple ~ % evcxr
Welcome to evcxr. For help, type :help
>> println!("{}", "e".len());
4
>> println!("{}", "e".chars().count());
1
>> █
```

Rust

如上图所示, Rust 中还提供了一个 `String.chars()` 方法, 它返回的是 `Chars` 类型, 表示字符串中字符, `Chars` 类型有一个方法叫做 `count()`, 可以获得字符串中字符的数量。

`&str` 和 `String` 类型的对比

1. 存储方式:

- `String`: 是一个可增长的、可变的、有所有权的字符串类型, 通常用于当需要修改或拥有字符串数据时。`String` 在堆上分配内存, 这使得它能够在运行时扩展。
- `&str`: 通常被称为字符串切片 (string slice), 是一个指向 UTF-8 编码字符串数据的不可变引用。`&str` 通常用于指向程序中固定的字符串字面量或其他字符串的一部分。

2. 可变性:

- `String` 可以被修改，例如添加内容、更改或清空等。
- `&str` 是内容不可变的，不能通过 `&str` 来修改它指向的字符串内容。

3. 使用场景:

- 当你需要拥有字符串并对其进行修改时，应使用 `String`。简单说就是：写代码的时候不知道字符串具体内容的，使用 `String`。
- 如果你只需要访问字符串而不需要拥有或修改它，使用 `&str` 更为合适，因为它不涉及额外的内存分配。也就是说，写代码的时候就知道字符串内容的，使用 `&str`。

4. 性能考虑:

- 由于 `String` 涉及到内存分配和可能的重新分配，其操作可能比 `&str` 更消耗资源。
- `&str` 由于其不变性和无需内存分配，通常在性能上更优，尤其是在只需要读取数据的场景中。

5. 类型转换:

- `String` 可以通过调用 `&str.to_string()` 或 `String::from(&str)` 方法从 `&str` 转换得到。
- 要从 `String` 获取一个 `&str`，可以使用 `&` 操作符取得其引用，例如通过 `&s` 其中 `s` 是一个 `String` 变量。
- `&String` 可以自动转换成 `&str`。比如说，一个函数的参数定义为 `&str` 类型的，如果我们传入 `&String` 类型的值，那么这个值会自动解引用为 `&str`。

函数、语句和表达式

函数

在 Rust 中，定义函数使用 `fn` 关键字，例如：

```

1 fn add(x: u32, y: u32) -> u32 {
2     // 这里是函数体
3 }
```

其格式可以写为：`fn function_name(parameters) → ReturnType { // 这里是函数体 }`。其中：

- *function_name* 是函数名，采用全小写下划线分隔的命名风格（蛇形命名法，*snake case*）
- *parameters* 是参数列表，每个参数都是 `param_name: ParamType` 的格式，多个参数之间使用逗号（`,`）分隔
- *ReturnType* 是函数返回值类型。如果函数没有返回值，则可以省略这一段。实际上，Rust 中的函数都是有返回值的，如果不写，那么返回的就是单元元组 `(())`

命名风格

我们在编写代码的时候，应遵循团队的命名风格和编码规范。常见的命名风格有：

- 驼峰命名法（Camel Case）：将多个单词连接在一起，并将首字母小写，后续单词的首字母大写，不使用下划线或其他分隔符。例如：`myVariable`, `firstName`, `getInfo`。
- 帕斯卡命名法（Pascal Case）：和驼峰命名法类似，但首字母大写。也被称为大驼峰命名法。例如：`MyVariable`, `FirstName`, `GetInfo`。
- 下划线命名法（Snake Case）：使用下划线作为单词之间的分隔符，所有字母小写，又称作蛇形命名法。例如：`my_variable`, `first_name`, `get_info`。
- 短横线命名法（Kebab Case）：使用短横线作为单词之间的分隔符。例如：`my-variable`, `first-name`, `get-info`。
- 匈牙利命名法（Hungarian Notation）：在变量名之前加上表示数据类型或其他元数据的前缀。例如：`intCount`, `strName`, `bIsVisible`。

Rust 中的命名风格

在 Rust 中，有一套完整的命名风格，如果我们在使用 Rust 开发的时候，不遵循它的命名规则，检查器会给我们提示。

- 类型名、枚举名：采用帕斯卡命名法，也就是大驼峰命名法。例如：

`StudentCard, CourseInfo`

- 属性名、变量名、函数名，采用下划线命名法，也就是蛇形命名法。例如：

`first_name, get_user_info`

- 常量，采用下划线分隔、全大写的命名法。例如：`FIRST_NAME,`

`DEFAULT_LANGUAGE`

- 项目名称（package），通常使用中划线（-）分隔，全小写，包中的 crate 会被 Rust 自动转换成下划线（_）

语句和表达式

Rust 是一种表达式语言（expression language），它明确区分表达式（Expression）和语句（Statement）。一个表达式是可以对其进行求值并得到一个结果值的，一个语句只是执行某个动作，但是不返回任何值。

比如说：在 Rust 中，定义变量的 `let` 是一个语句（以后的行文中，我们将明确区分语句和表达式），它没有返回值。这一点和 JavaScript 不同。

在 JavaScript 中，一个赋值表达式是有返回值的，例如 JavaScript 代码：`var s = a = b = 1;` 其中 `b=1` 的返回值是 `1`，所以 `a` 被赋值为 `1`，对 `a` 赋值 `1`，又会产生一个返回值 `1` 赋值给变量 `s`。所以上面这句代码执行之后，变量 `s`, `a`, `b` 的值均为 `1`。

```

1 fn foo() -> i32 {
2     let y = { // ----- 一对儿花括号印起来的块是表达式
3         let x = 3; // 字面量 3 是表达式
4         // 块中最后一个表达式如果没有尾部的分号,
5         // 那么它就是这个块表达式的返回值
6         x + 1
7     }; // -----
8     y // 所以, 这里 y 是函数 foo() 的返回值
9 }
10
11 fn main() {
12     let n = foo(); // 函数调用是表达式
13 }
```

所以，一个函数块最后一行不带分号（;）的表达式就是这个函数的返回值。如果函数需要中途返回的，可以使用 `return expression;` 语句。

控制流程

if 表达式

`if` 表达式使用 `if` 关键字，然后跟着 `bool` 类型的表达式，并且这个 `bool` 类型的表达式无需放到括号中。在 Rust 中，没有 Truly 和 Falsy 的概念。比如说，在 JavaScript 中，我们可以这样写：

JavaScript 代码

```

1 function foo() {
2     const a = 1;
3     if (a) {
4         console.log("condition was true");
5     } else {
6         console.log("condition was false");
7     }
8 }
```

但是在 Rust 中，`if` 后面的必须是 `bool` 类型的表达式。所以上面的 JavaScript 代码对应的 Rust 代码如下：

Rust 代码

```

1 fn foo() {
2     let a = 1;
3     if a == 1 {
4         println!("condition was true");
5     } else {
6         println!("condition was false");
7     }
8 }
```

`if-else-if` 表达式很容易想象到：

```

1 fn foo() {
2     let number = 6;
3     if number % 4 == 0 {
4         println!("number is divisible by 4");
5     } else if number % 3 == 0 {
6         println!("number is divisible by 3");
7     } else if number % 2 == 0 {
8         println!("number is divisible by 2");
9     } else {
10         println!("nubmer is not divisible by 4, 3, or 2")
11     }
12 }
```

思考题：上面这段代码最终是哪个分支被执行了？

既然 `if` 是一个表达式，那么它是有返回值的，所以我们可以把 `if` 表达式的返回值绑定给某个变量：

```

1 fn foo() {
2     let condition = true;
3     let number = if condition { 5 } else { 6 };
4     println!("The value of number is: {number}");
5 }
```

需要注意的是，`if` 的分支返中返回值的数据类型必须一致，例如，下面这段代码是无法通过编译的：

```

1 // 以下代码无法通过编译
2 fn foo() {
3     let condition = true;
4     let number = if condition { 5 } else { "six" };
5     println!("The value of number is: {number}");
6 }
```

因为我们在 `if` 分支中返回了 5，对应的数据类型是 `i32`，在 `else` 分支中返回的是 `"six"`，对应的数据类型是 `&str`，所以这段代码无法通过编译。

loop 表达式

`loop` 也是表达式，它是一个死循环，直到遇到 `break` 语句。`break` 语句后面的表达式就是 `loop` 表达式的返回值。

```

1 fn foo() {
2     let mut counter = 0;
3     let result = loop {
4         counter += 1;
5         if counter == 10 {
6             break counter * 2;
7         }
8     };
9
10    println!("The result is {result}");
11 }
```

嵌套的 `loop` 如何中止？在 `loop` 表达式之前可以给他一个 `'label` 来表示这个 `loop` 表达式的名字，然后 `break` 语句后跟随 `'label` 即可。

```

1 fn foo() {
2     let mut count = 0;
3     'counting_up: loop {
4         println!("count = {count}");
5         let mut remaining = 10;
6         let _ = 'inner_loop: loop {
7             println!("remaining = {remaining}");
8
9             if remaining == 9 {
10                 break 'inner_loop 3;
11             }
12
13             if count == 2 {
14                 break 'counting_up;
15             }
16
17             remaining -= 1;
18         };
19
20         count += 1;
21     }
22
23     println!("End count = {count}");
24 }
```

while 语句

`while` 是语句，不是表达式，所以它没有返回值。类似 `if` 表达式，`while` 后面跟随的必须是 `bool` 类型的表达式，并且无需放到括号中。

```

1 fn foo() {
2     let mut number = 3;
3     while number != 0 {
4         println!("{}{}", number);
5         number -= 1;
6     }
7 }
```

for 语句

`for` 是一个语句，不是表达式，所以它没有返回值。

```

1 fn foo() {
2     let items = [10, 20, 30, 40, 50];
3     for item in items {
4         println!("item = {}", item);
5     }
6
7     for i in 0..10 {
8         println!("i = {}", i);
9     }
10 }
```

扩展：`println!` 中的格式

基本语法

Rust 中 `println!` 宏的基本语法为：

```
1 println!(<format-string>, <arguments...>);
```

后面的 `<arguments ...>` 是可以省略的，也就是说，可以直接输出不带任何参数的字符串字面量：

```
1 | println!("Hello, world!");
```

可以使用本节所讲格式的宏还有：

- `format!` 区别就是 `println!` 将内容直接输出到标准输出，而 `format!` 宏是返回一个 `String` 类型的值
- `write!` 第一个参数是一个 `&mut io::Write` 或 `&mut fmt::Write`
- `writeln!` 同上，且在最后增加换行符
- `eprint!` 输出到标准错误（`stderr`）中
- `eprintln!` 同上，且在最后增加换行符

基本占位符

使用 `{}` 作为参数的基本占位符。这种用法是最常见的：

```
1 | let x = 5;
2 | let y = 10;
3 | println!("x = {}, y = {}", x, y);
```

下标占位符

可以通过 `{index}` 来使用下标对应到参数。参数列表中的参数下标从 `0` 开始，例如：

```
1 | let x = 5;
2 | let y = 10;
3 | println!("x = {1}, y = {0}", y, x);
```

命名参数

`{}` 中可以填入参数名称。参数的名称可以是上下文中的变量名称：

```

1 let x = 5;
2 let y = 10;
3 println!("x = {x}, y = {y}");

```

也可以是参数列表中的命名参数：

```

1 println!("x = {x}, y = {y}", x = 5, y = 10);
2
3 println!("{subject} {verb} {object}",
4         object="the lazy dog",
5         subject="The quick brown fox",
6         verb="jumps over");

```

还可以是函数的输入参数：

```

1 fn make_string(a: u32, b: &str) -> String {
2     format!("{} {}", b, a)
3 }

```

宽度

可以指定输出时的宽度。

```

1 // 以下语句均可输出: "Hello x      !"
2 println!("Hello {:5}!", "x");
3 println!("Hello {:1$}!", "x", 5);
4 println!("Hello {1:0$}!", 5, "x");
5 println!("Hello {:width$}!", "x", width = 5);
6 let width = 5;
7 println!("Hello {:width$}!", "x");

```

对齐与补充字符

使用 < 左对齐， ^ 居中对齐， > 右对齐：

```

1 println!("Hello {:<5}!", "x"); // 输出: "Hello x      !"
2 println!("Hello {:-<5}!", "x"); // 输出: "Hello x----!"
3 println!("Hello {:^5}!", "x"); // 输出: "Hello   x  !"
4 println!("Hello {:>5}!", "x"); // 输出: "Hello      x!"

```

数字的符号及进制

对于数值类型的，可以设置输出时候的符号以及进制：

```

1 println!("Hello {:+}!", 5); // 输出: "Hello +5!"
2 println!("{:#x}!", 27); // 输出: "0x1b!"
3 println!("{:#X}!", 27); // 输出: "0x1B!"
4 println!("{:#b}!", 27); // 输出: "0b11011!"
5 println!("{:#010b}!", 27); // 输出: "0b00011011!"
6 println!("Hello {:05}!", 5); // 输出: "Hello 00005!"
7 println!("Hello {:05}!", 5); // 输出: "Hello -0005!"

```

精度

使用 `.N` 格式设置输出的精度。

- 非数值类型的值，之类的精度可以理解为宽度，超出宽度的部分将会被截断
- 整数，此格式参数将被忽略
- 浮点数，表示小数点后输出的位数

使用 `.N$` 格式的：

- `N` 是整数的：取对应下标的参数值作为宽度格式或精度格式
- `N` 是参数名的：取对应参数的值作为宽度格式或精度格式

转义

格式字符串中包含 `{` 或 `}` 的，可以通过 `{}{}` 转义，例如：

```

1 println!("Hello {}"); // 输出: "Hello {}"
2 println!("{{ Hello"); // 输出: "{ Hello"

```

其他

? 可以输出实现了 Debug trait 的值, #? 可以对值做更漂亮的格式化。

```

1 println!("{:?}" , Some(1)); // 输出: Some(1)
2 println!("{:#?}" , Some(1)); // 输出:
3                                 // Some(
4                                 //      1,
5                                 // )

```

完整语法

`println!` 宏的完整语法如下:

```

1 format_string := text [ maybe_format text ] *
2 maybe_format := '{' '{' | '}' '}' | format
3 format := '{' [ argument ] [ ':' format_spec ] [ ws ] * '}'
4 argument := integer | identifier
5
6 format_spec := [[fill]align][sign]['#']['0'][width]['.'
precision]type
7 fill := character
8 align := '<' | '^' | '>'
9 sign := '+' | '-'
10 width := count
11 precision := count | '*' | '?' | 'x?' | 'x?' | identifier
12 type := '' | '?' | 'x?' | 'x?' | identifier
13 count := parameter | integer
14 parameter := argument '$'

```

实战

摄氏度华氏度转换

摄氏度转华氏度公式: $F = C \times 1.8 + 32$, 华氏度转摄氏度公式: $C = (F - 32) \div 1.8$ 。这个代码的实现不难, `main()` 函数中读取用户的输入, 然后调用对应的函数进行转换:

```

1 fn main() {
2     loop {
3         println!("摄氏度 --> 华氏度请输入 1");
4         println!("华氏度 --> 摄氏度请输入 2");
5         println!("退出请输入 0");
6         println!("\n你的选择是: ");
7
8
9         let mut choice = String::new();
10        io::stdin().read_line(&mut choice).expect("读取输入失败");
11
12        let choice = choice.trim();
13
14        if choice == "0" {
15            break;
16        }
17
18        if choice == "1" {
19            println!("\n请输入摄氏度: ");
20            let mut c = String::new();
21            io::stdin().read_line(&mut c).expect("读取输入失败");
22            let c: f32 = c.trim().parse().expect("请输入有效的数
值");
23            let f = c_to_f(c);
24            println!(" {:.2} 摄氏度 --> {:.2} 华氏度\n", c, f);
25        }
26
27        // 华氏度到摄氏度的转换请大家自行完成
28    }
29 }
30
31 fn c_to_f(c: f32) -> f32 {
32     c * 1.8 + 32.0
33 }
```

数组累加

此实战要求对一个 `[u32; 5]` 的数组元素求和。要求有：

- 让用户输入一些列整数，使用空格分隔
- 如果输入的数值不足 5 个，不足的部分以 0 计
- 如果输入的数值超过 5 个，那么只取前 5 个
- 如果输入的不是有效的整数，则中止
- 计算输入的数值的和并把输入的数值和最终累加的结果打印到屏幕
- 计算数组项之和的要求单独一个函数

```

1 fn main() {
2     println!("请输入 5 个整数，空格分隔");
3
4     let mut s = String::new();
5     io::stdin().read_line(&mut s).expect("读取输入失败");
6
7     let mut numbers = [0u32; 5];
8
9     let mut index = 0;
10
11    for n in s.trim().split(" ") {
12        let nn: u32 = n.trim().parse().expect("请输入有效的整数");
13        numbers[index] = nn;
14        index += 1;
15
16        if index == 5 {
17            break;
18        }
19    }
20
21    let total = arr_add(&numbers[..]);
22    println!("sum {:?} = {}", numbers, total);
23 }
24
25 fn arr_add(items: &[u32]) -> u32 {

```

```

26     let mut total = 0u32;
27     for n in items {
28         total += n;
29     }
30     total
31 }
```

斐波那契数列第 n 项

斐波那契数列：1, 1, 2, 3, 5, 8, 13, 21……，规律就是每一项都是前两项之和。为了方便处理，我们把第 0 项设置为 0，第 1 项为 1，依此类推。

斐波那契数列求和有两种方式实现：递归方式和循环方式。递归方式：

```

1 fn fibo_recursive(n: u32) -> u32 {
2     if n == 0 {
3         return 0;
4     }
5
6     if n == 1 || n == 2 {
7         return 1;
8     }
9
10    fibo1(n - 1) + fibo1(n - 2)
11 }
```

为什么我们把第 0 项定义为 0 呢？因为这里的参数类型是 `u32`，无符号整数包含了 `0` 和正整数，为了代码的健壮性，我们要正确处理边界数据。使用递归方式计算的时候，很容易出现栈溢出的情况，大家可以试试在自己的电脑上大概能够成功计算到多少项。

使用循环方式：

```

1 fn fibo_loop(n: u32) -> u32 {
2     if n == 0 {
3         return 0;
4     }
```

```

5      if n == 1 || n == 2 {
6          return 1;
7      }
8
9
10     let mut a = 1u32;
11     let mut b = 1u32;
12
13     for _ in 3..n {
14         let m = a + b;
15         a = b;
16         b = m;
17     }
18
19     a + b
20 }
```

使用循环方式很容易出现数值计算溢出的情况。请大家对上面的代码进行修改，正确处理数值计算溢出的情况。

字符统计

此练习是让用户输入一些字符，然后统计 `a-z` 每个字符的出现次数。的要求如下：

- 仅统计 `a-z` 26 个字母，其他的字符舍弃
- 如果输入的文本中有大写的 A-Z，计数到对应的小写字母上。也就是说，大小写不敏感的
- 最终输出的时候，一行一个结果，总共输出 26 行。例如：`a => 1` 这样的

这个练习主要是通过使用 `as` 关键字进行数值类型的数据转换。

```

1 fn main() {
2     println!("请输入一些字符");
3     let mut s = String::new();
4     io::stdin().read_line(&mut s).expect("读取输入失败");
```

```

5
6     let mut results = [0u32; 26];
7
8     for c in s.trim().to_lowercase().chars() {
9         let code = c as u8;
10        if 97 <= code && code <= 122 {
11            results[(code - 97) as usize] = results[(code - 97)
12                as usize] + 1;
13        }
14    }
15
16    for i in 0..26 {
17        println!("{} => {}", ((i as u8) + 97) as char,
18        results[i]);
19    }
20 }
```

判断一个数值是否是质数

质数是指除了 1 和他自身之外，没有其他因子的数。例如：2, 3, 5, 7, 11...。0 和 1 都不是质数。让用户输入一个 $2^{31} - 1$ 之内的整数（2,147,483,647），判断是否是质数。所以我们可以通过使用 `2..= 输入数值的平方根` 的整数对输入的整数求余数，如果余数为 0，则表示这个数字不是质数（有除了 1 和自身之外的其他因子）。

```

1 fn is_prime(n: u64) -> bool {
2     let mut i = 2u64;
3     while i * i <= n {
4         if n % i == 0 {
5             return false;
6         }
7         i += 1;
8     }
9
10    true
11 }
```

这里练习的是数值计算中的 `%` 取模操作。另外，目前稳定版的 Rust 中还没有提供计算整数平方根的方法，所以行 3 的代码做了一个反向操作，检测 i^2 是否大于 `n` 来判断 `i` 是否大于 `n` 的平方根。

冒泡排序

随机生成 10 个 1 到 100 之间的整数，存储到数组中，并对数组进行排序。既然是排序，那么就涉及到数据交换位置的操作，Rust 中的切片类型提供了一个叫做 `swap()` 的方法来交换数组中两个元素的位置。

冒泡排序（Bubble Sort）是一种简单的排序算法，它通过重复地遍历待排序的列表、比较相邻的元素并在必要时交换它们来实现排序。它的名字源于排序过程中较小元素会“浮”到列表顶端，而较大元素会“沉”到列表底端，就像气泡在液体中一样。

冒泡排序的工作原理如下：

1. 从列表的开头开始，逐一比较相邻的两个元素。
2. 如果它们的顺序错误（即第一个元素大于第二个元素），则交换它们的位置。
3. 对每对相邻元素重复这个过程，直到到达列表的末尾。这时，最大的元素已经“冒泡”到列表的最后。
4. 将未排序部分的最后一个元素排除在外，因为它已经在正确的位置上。
5. 对剩下的未排序部分重复上述步骤，直到所有元素都排好位置。

```

1 fn main() {
2     println!("随机生成 10 个数值");
3
4     let mut numbers = [0u32; 10];
5
6     for i in 0..10 {
7         numbers[i] = rand::thread_rng().gen_range(1..100);
8     }
9
10    println!("BEFORE: {:?}", numbers);
11
12    let len = numbers.len();

```

```

13     for i in 0..len {
14         for j in 0..len - i - 1 {
15             if numbers[j] > numbers[j + 1] {
16                 numbers.swap(j, j + 1);
17             }
18         }
19     }
20
21     println!("AFTER : {:?}", numbers);
22 }
```

回顾

1. `let` 语句用来定义变量。在 Rust 中定义变量的过程称为将值绑定到变量
2. `let mut` 语句定义其值可变的变量
3. 标量类型有：整数、浮点数、布尔值、字符
4. 数值运算溢出处理 4 种类型：Checked, Wrapping, Saturation, Overflowing
5. 复合类型有：元组、数组
6. 其他类型：切片、字符串（`String` 和 `str`）
7. 表达式和语句的区别
8. 定义函数
9. `if` 表达式，如果要返回值，要求每个分支返回值的数据类型一致
10. `loop` 表达式，以及如何在嵌套 `loop` 中 `break`
11. `while` 语句
12. `for` 语句