# 第二章 数据类型与变量

本章要点：数据类型；变量与常量；类型转换。

数据类型与变量在程序设计语言中用于存储和处理数据。程序的效率、可读性和可维护性在很大程度上受到数据类型与变量的选择和使用技巧的影响。

## 2.1 引言

在现实世界中，人们会将所有事物按照特定的含义分门别类。例如，生物学上将所有生命分为真核域、细菌域、古菌域。同属一个类型的事物通常有着一些相同的属性。在计算机世界中，数据以二进制形式存储，这些0和1的序列反映了现实世界中的信息，因此，计算机世界中的数据也应当有类型之分。早在20世纪40年代，类型的概念就被数学家Alonzo Church运用在简单类型λ演算（Simple Typed Lambda Calculus）中，用以证明程序类型系统的安全性，这为后来的函数式编程语言的设计和理论奠定了基础，并对计算理论和编程语言研究产生了深远的影响。

在程序设计语言中引入数据类型，不仅仅是为了让程序中的数据和现实世界中某种类型的事物对应起来，数据类型还能够给我们带来诸多好处。数据类型为我们提供了对0，1序列的抽象表述。程序开发人员如果想要高效地完成软件开发，直接操作0，1序列进行计算显然不是一个明智的选择。数据类型让我们能够在计算数据时，只需要关心进行什么样的操作，而不用关心在内存单元中的0，序列具体需要怎样运算才能得到我们想要的结果；类型安全(type safety)的概念认为，一个类型良好的程序将不会出错；数据类型允许编译器对程序进行类型检查，从而避免程序中出现类型矛盾的错误；同时，类型提供的信息有助于编译器和分析工具对程序进行分析和优化。

在课堂上，老师在进行课堂互动时可能会用“这位同学”、“那位同学”临时指代某位特定的学生；在统计课程成绩时，老师会按名字-成绩的格式登记成绩信息。上面的例子中，我们都用某个特定符号表示一个具体对象或数据。程序中的变量也是如此。程序中的变量(variable)是一个符号，指代程序运行时一个具体对象的值（value）。程序中所有变量及其值的映射关系的集合，构成了程序的状态（state）。下面的式子定义了程序的状态：

变量提供的符号到值的映射关系，使得开发人员在设计程序时对值的操作转为对抽象符号的操作，让开发人员能够更有效地组织和处理数据。

## 2.2 基本数据类型

基本数据类型是程序设计语言的基础组成部分，表示各种基本的数据值。基本数据类型的变量在内存中通常占据固定的空间。对基础类型的操作通常由程序设计语言的语法和语义直接给出。

### 2.2.1 整数类型

Rust提供的整数类型为固定长度的整数类型，包括无符号整数（unsigned integer）类型和有符号整数（signed integer）类型。

无符号整数类型有：

* u8：表示0~28-1范围内的整数，通常情况下占用8 bit空间。
* u16：表示0~216-1范围内的整数，通常情况下占用16 bit空间。
* u32：表示0~232-1范围内的整数，通常情况下占用32 bit空间。
* u64：表示0~264-1范围内的整数，通常情况下占用64 bit空间。
* u128：表示0~2128-1范围内的整数，通常情况下占用128 bit空间。
* usize：表示0~2k-1范围内的整数，占用k bit空间。k的值由编译目标平台的架构确定。k的值通常为32或64. uszie类型通常用于表示一段连续内存空间的大小，或作为访问一段连续内存空间的索引。

有符号整数类型有：

* i8：表示-27~27-1范围内的整数，通常情况下占用8bit空间。
* i16：表示-215~215-1范围内的整数，通常情况下占用16bit空间。
* i32：表示-231~231-1范围内的整数，通常情况下占用32bit空间。
* i64：表示-263~263-1范围内的整数，通常情况下占用64bit空间。
* i128：表示-2127~2127-1范围内的整数，通常情况下占用128bit空间。
* isize：表示-2k~2k-1范围内的整数，占用k bit空间。k的值由编译目标平台的架构确定。k的值通常为32或64. iszie类型的功能与usize类似。

代码片段ch2-1.rs给出了一些整数类型变量的实例。

1. fn main() {
2. let x: u8 = 0;
3. assert\_eq!(x, 0u8);
4. assert\_eq!(0xffff, u16::MAX);
5. assert\_eq!(0x0000, u16::MIN);
6. assert\_eq!(0x7fffffff, i32::MAX);
7. assert\_eq!(0x80000000u32 as i32, i32::MIN);
8. }

代码片段ch2\_1.rs

符号位

k-1个数值位

(b) k bit有符号整数

k个数值位

1. k bit无符号整数

图2.1 长度为k的固定长度整数的内存模型

图2.1给出了固定长度整数内存模型的示意，其中（a）表示长度为k bit的无符号整数；（b）表示长度为k bit的有符号整数。有符号整数需一个bit表示其符号信息，因而对于相同长度的无符号整数和有符号整数，有符号整数能表示的最大整数比无符号整数能表示的最大整数更小，并且有符号数的值域是非对称的。

### 2.2.2 浮点数类型

Rust支持的浮点数遵从IEEE 754-2008标准。Rust提供的浮点数类型包括：

* f32：表示-3.40×1038~3.40×1038范围内、有效位数6~9位的小数，占用32bit空间。
* f64：表示-1.80×10308~1.80×10308范围内、有效位数15~17位的小数，占用64bit空间。

23bit尾数

31

30

23

22

0

63

62

52

51

0

符号位

(b) IEEE 754 标准的64位浮点数

**…**

**…**

8bit阶码

(a) IEEE 754 标准的32位浮点数

符号位

**…**

**…**

11bit阶码

52bit尾数

图2. 1 IEEE 754标准的浮点数内存模型

IEEE 754标准定义的浮点数由3部分组成：符号位、阶码（exponent）、尾数（mantissa）。若将符号位的制值表示为，阶码的十进制值表示为，尾数的十进制值表示为，那么32位非0浮点数的真值经下面的式子计算得出：

64位非0浮点数的真值经下面的式子计算得出：

在IEEE 754标准下，计算浮点数四则运算的方式可以由浮点数真值的计算方式推导得出。例如，有32位浮点数和，假设它们的符号位是和且，阶码是和且，尾数是和，那么的符号位，阶码，尾数满足下面的式子：

由于尾数的长度是有限的，故经过上面式子计算得到的尾数需要舍入小数位，以减少精度损失。 用上面的式子计算32位浮点数3.14+1.14，3.14和1.14的二进制码如图2.3所示。代入上面的式子，可以计算得到3.14+1.14的阶码是1000 00012，尾数是000 1000 1111 0101 1100 00112.

23bit尾数

31

30

23

22

0

0

1 0 0 0 0 0 0 0

1 0 0 1 0 0 0 1 1 1 1 0 1 0 1 1 1 0 0 0 0 1 1

8bit阶码

(a) 3.14的32位浮点数二进制码

符号位

23bit尾数

31

30

23

22

0

0

0 1 1 1 1 1 1 1

0 0 1 0 0 0 1 1 1 1 0 1 0 1 1 1 0 0 0 0 1 0 1

8bit阶码

(b) 1.14的32位浮点数二进制码

符号位

图2. 3 IEEE 754标准下32位浮点数3.14和1.14的二进制码

IEEE 754标准中还定义了一些特殊值的浮点数表示：

* 阶码、尾数全0的浮点数表示0；
* 指数位全部为 1，尾数位全部为0表示无穷大；
* 指数位全部为 1，尾数位至少有一位为1表示NaN值，表示无效的操作结果。

代码片段ch2\_2.rs列出了一些浮点数类型的实例。Rust给f32和f64类型的浮点数都提供了IEEE 754标准中定义的特殊常量值，包括MIN（最小常量值）、MAX（最大常量值）、INFINITY（正无穷大）、NEG\_ INFINITY（负无穷大）、NAN（正无穷大）、NAN（无效值）。另外，Rust还给f32和f64类型提供了常量值EPSILON，用以表示浮点数的最小可表示精度。

1. fn main() {
2. let \_f = 0.0f32;
3. let f = 1.1920929e-7;
4. assert\_eq!(f, f32::EPSILON);
5. assert\_eq!(2.220446049250313e-16, f64::EPSILON);
6. assert\_eq!(-3.4028235e38, f32::MIN);
7. assert\_eq!(3.4028235e38, f32::MAX);
8. assert\_eq!(-1.7976931348623157e308, f64::MIN);
9. assert\_eq!(1.7976931348623157e308, f64::MAX);
10. }

代码片段ch2\_2.rs

### 2.1.3 字符类型

Rust提供char类型表示字符类型，char类型表示一个单独的字符。Rust中，char类型的变量的值均为Unicode标量值（Unicode scalar value），每个字符类型的变量占32bit空间。Unicode标量值的范围在0x0000~0xd7ff和0x00 e000~0x10 ffff内。代码片段ch2\_3.rs提供了一些char类型变量的实例。Char类型支持按ASCII十六进制码和Unicode十六进制码定义。‘\’在字符中表示转义符。被定义的字符是Rust的关键字时，需要在被定义的字符值前加上转义符。

1. fn main() {
2. let x: char = 'x';
3. assert\_eq!(x as u32, 0x78u32);
4. *// Use ASCII hex code to define a char*
5. assert\_eq!(x, '\x78');
6. *// Use Unicode hex code to define a char*
7. assert\_eq!(x, '\u{78}');
8. *// will print: '\'' is '*
9. let single\_quotation\_mark: char = '\'';
10. println!("\'\\\'\' is {}", single\_quotation\_mark);
11. *// will print: '\' is \*
12. let backslash: char = '\\';
13. println!("\'\\\' is {}", backslash);
14. assert\_eq!(0x10ffffu32, char::MAX as u32);
15. }

代码片段ch2\_3.rs

### 2.2.4 布尔类型

bool类型的变量用于表示一个逻辑命题的真、假。bool类型变量的值域为{true, false}，占一个字节空间。代码片段ch2\_4.rs提供了一些bool类型变量的实例。

1. fn main() {
2. let t: bool = true;
3. let f: bool = false;
4. assert\_eq!(1 > 0, true);
5. assert\_eq!(1 > 114, false);
6. }

代码片段ch2\_4.rs

### 2.2.5 特殊类型

除了上面的基本类型，Rust还提供了单元类型（unit type）和never类型。这两种特殊类型是Rust类型系统不可或缺的重要组成部分，增强了Rust类型系统的表达能力。

**1. 单元类型。**

Rust的单元类型表示一个没有具体值的类型，不占用内存空间。单元类型写作“()”。比较特别的，“()”既能表示单元类型，也能表示一个单元类型的值。单元类型主要被使用于函数无返回值、复合数据结构的占位符等场景。

1. use std::mem::size\_of;
2. fn main() {
3. let \_unit: () = ();
4. *// The size of unit type is 0.*
5. assert\_eq!(size\_of::<()>(), 0);
6. }

代码片段ch2\_5.rs

**2. Never类型。**

Rust提供的never类型写作“!”，表示不可能有返回值的运算过程，常用于描述发生错误、线程退出或其他不可恢复的程序状态。代码片段ch2\_6.rs使用Rust的panic!宏（抛出程序运行时错误）创建了一个never类型的变量。

1. fn main() {
2. let \_never = panic!("Master Spark~~~");
3. }

代码片段ch2\_6.rs

Rust中Never类型的设计理念源自类型理论中的底类型（bottom type），写作⊥。底类型在强类型系统的语言中，表示永不返回值的函数或表示程序错误的状态。

### ~~2.2.6 Option类型~~

~~Rust的Option<T>类型是一个枚举类型。有关枚举类型的介绍将在后面的章节中涉及。Option<T>类型表示一个值可以是某个具体类型T的值（Some(T)），也可以是空（None）。Option<T>类型主要被用于处理可空值。如代码片段ch2\_7.rs所示，Rust中的str类型提供的get方法允许用户从str类型的字符串中取出一个字串，返回值的类型是Option<&str>。标准库的设计者通过用Option<&str>返回值类型提醒调用get方法的用户，在子串的索引超出字符串的长度范围时是无效的，get方法将返回None值（第6行）。在第10行，使用可能为空的值时，直接取出Option类型中的值是存在风险的。Rust给Option类型设计的接口能够帮助用户规避其他主流程序设计语言中常见的空指针错误和异常。~~

1. ~~fn main() {~~
2. ~~let s = "Rust Programming Language.";~~
3. ~~let c = s.get(0..4);~~
4. ~~assert\_eq!(c, Some("Rust"));~~
5. ~~let c = s.get(0..100);~~
6. ~~assert\_eq!(c, None);~~
7. ~~if c.is\_some() {~~
8. ~~println!("{}", c.unwrap());~~
9. ~~}~~
10. ~~}~~

~~代码片段ch2\_7.rs~~

## 2.3 复合类型

### 2.3.1 数组

**1. 数组的定义与基本概念。**

在Rust中，数组用于存储固定数量、相同类型的元素集合。数组在内存中占据连续的空间，这使得访问数组元素非常高效。Rust数组的类型标注为[T; N]，其中T表示元素的类型，N表示元素的数量。

Rust数组具有强类型和固定长度的特点。数组在声明后，其大小和元素类型不可更改。

1. // 声明一个长度为5，元素类型为i32的数组
2. let numbers: [i32; 5] = [1, 2, 3, 4, 5]

这里，numbers是一个固定大小为5的整型数组。数组中的每个元素都是i32类型。

**2. 数组的创建和初始化。**

数组的创建和初始化可以通过多种方式实现。最常见的方法是直接列出数组中的所有元素。此外，Rust支持使用单一值初始化整个数组，这在需要创建多个重复元素的数组时非常有用。

1. *// 直接列出所有元素*
2. let days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"];
3. *// 使用单一值初始化数组*
4. let zeroes = [0; 10];  *// 一个包含10个0的数组*

在第一个例子中，days数组直接列出了所有元素。在第二个例子中，zeroes数组通过[0; 10]初始化，表示创建了一个包含10个元素的数组，且每个元素都为0。

值得一提的是，Rust遵循RAII（Resource Acquisition Is Initialization）原则，这意味着在Rust中，一旦数组被创建，它就必须被初始化。这个特性确保了内存的安全性和高效性，避免了未初始化的内存引发的问题。

**3. 访问数组的元素**

Rust数组中的元素可以通过索引访问，索引从0开始。访问数组越界的索引将会导致编译错误或者运行时错误(panic)，这是Rust内存安全的一个重要体现。

1. let INDEX: usize = 10;
2. let mut numbers = [1, 2, 3, 4, 5];
3. *// numbers[INDEX] = 10;  // 这会导致编译错误*
4. let index\_read = std::env::args().nth(1).unwrap().parse::<usize>().unwrap();
5. let value = numbers[index\_read]; *//这可能会导致运行时错误*

当使用一个明显超出数组长度的索引时，Rust的编译器能够在编译时捕捉到这个错误。在这个例子中，我们尝试使用常量INDEX（值为10）作为索引来访问数组numbers。由于numbers数组的长度为5，索引10明显超出范围，这将导致编译时错误。

当索引的值在编译时未知，或者其有效性依赖于运行时的条件时，超出范围的索引将会导致运行时错误。在这个例子中，索引index\_read是基于程序运行时传入的命令行参数确定的。如果这个参数对应的值超出了数组numbers的长度，将会导致运行时错误。

**~~迭代数组~~**

~~Rust提供了多种方法来迭代数组中的元素，最常见的是使用for循环。~~

1. ~~for num in numbers.iter() {~~
2. ~~println!("Number: {}", num);~~
3. ~~}~~

~~在上面的代码中，我们使用了iter()方法和for循环来遍历numbers数组的每个元素，并打印它们。iter()方法返回一个迭代器，允许我们以不可变的方式遍历数组。~~

**4. 数组的更新和修改**

Rust数组的长度是不可变的，这意味着Rust数组的长度在声明后无法更改。因此Rust数组不支持添加或删除元素。不可变长度的特性有利于编译时的效率和优化，编译器在编译时就能知道数组将占用多少连续的内存空间，这将使得编译器能够执行内存布局优化、数据预加载等优化。

默认情况下，Rust中的数组是不可变的。但是被声明为可变数组的元素则可以被修改。

1. let numbers = [1, 2, 3, 4, 5];
2. *// numbers[0] = 10;  // 这会导致编译错误*
3. let mut numbers\_mut = [1, 2, 3, 4, 5];
4. numbers\_mut[0] = 10;  *// 修改数组的第一个元素*

在这个例子中，尝试修改numbers数组的任何元素都会导致编译错误，因为它是不可变的。numbers\_mut数组声明中加入mut关键字，使其成为可变的。这样，我们就可以修改numbers\_mut数组中的元素。

1. let mut temperatures = [22.0, 23.5, 19.5];
2. temperatures[0] = 25.0;  *// 更新数组的第一个元素*

在这个例子中，我们更新了temperatures数组的第一个元素。由于temperatures是可变的，这种类型的修改是允许的。

### 2.3.2 元组

**1. 元组的定义与基本概念**

元组是Rust中一种用于存储固定数量多种类型值的复合数据类型。它们允许将不同类型的数据组合成单个复合类型，适合用于返回多个值的场景。与数组不同，元组中的每个元素可以是不同类型的。

1. *// 创建一个包含不同类型元素的元组*
2. let mixed : (i32, f64, &str) = (500, 6.4, "Hello");

在这个例子中，mixed是一个元组，包含了一个整数、一个浮点数和一个字符串。这个元组的类型是(i32, f64, &str)。

**2. 元组的创建和访问**

元组可以通过直接声明其包含的元素来创建。访问元组中的元素需要使用成员访问运算符“.”和对应元素的索引实现。~~与结构体不同，元组中值是没有字段名的。~~

1. *// 创建一个元组*
2. let person = ("Alice", 30);
3. *// 访问元组的元素*
4. let name = person.0;
5. let age = person.1;
6. println!("Name: {}, Age: {}", name, age);

这个例子创建了一个包含两个元素的元组person，并通过“person.0”和“person.1”来分别访问这个元组的第一个和第二个元素。

**3. 元组的解构**

Rust中可以通过模式匹配的方式，将元组中的值分解到单独的变量中。

1. *// 创建一个元组*
2. let tuple = ("Alice", 30);
3. *// 使用模式匹配来解构元组*
4. let (name, age) = tuple;
5. println!("Name: {}, Age: {}", name, age);

在这个例子中，我们通过模式匹配将tuple元组解构成两个独立的变量name和age。这种方式非常适合在需要同时处理元组中多个值的场景。

**~~4. 元组作为函数的多值返回~~**

~~元组在Rust中常用于从函数返回多个值。这使得函数可以同时返回不同类型的值，而无需创建一个专门的结构体。~~

1. ~~fn calculate\_stats(numbers: &[i32]) -> (f64, i32) {~~
2. ~~let sum: i32 = numbers.iter().sum();~~
3. ~~let avg: f64 = sum as f64 / numbers.len() as f64;~~
4. ~~(avg, sum)~~
5. ~~}~~
6. ~~let numbers = [1, 2, 3, 4, 5];~~
7. ~~let (average, total) = calculate\_stats(&numbers);~~
8. ~~println!("Average: {}, Total: {}", average, total);~~

~~在这个例子中，calculate\_stats函数接受一个整数数组的引用，并返回一个元组，其中包含了数组的平均值和总和。这是元组在实际应用中的一个典型用途。~~

### 2.3.3 区间

**1. 区间的概念和创建。**

在Rust中，区间（Range）是一系列连续的值，通常用于循环和迭代。区间通过起始值和结束值定义。Rust提供了闭区间a..=b和半开区间a..b的定义方式。

1. let a\_to\_b = 1..=5; *// 闭区间：包含1到5*
2. let a\_to\_b\_exclusive = 1..5; *// 半开区间：包含1到4*

在这个例子中，a\_to\_b定义了一个从1到5的闭区间，包含两端的值；而a\_to\_b\_exclusive则是一个从1到5的半开区间，不包含结束值5。

**2. 区间在循环和迭代中的应用。**

区间在循环控制结构中特别有用，它们常常被用于for循环中，提供了一种简洁的方式来重复执行代码块。

1. for num in 1..=5 {
2. println!("{}", num); *// 这将打印数字1到5*
3. }

在这个例子中，for循环遍历了闭区间1..=5中的每个数字，并打印它们。这是区间在Rust中常见的使用方式之一。

Rust的区间支持步进迭代，即每次迭代跳过一定的步数。这可以通过step\_by方法实现。

1. let range = 0..10;
2. *// 每次迭代跳过2个元素*
3. for num in range.step\_by(2) {
4. println!("{}", num); *// 将打印 0, 2, 4, 6, 8*
5. }

在这个例子中，我们使用step\_by方法指定了每次迭代跳过2个元素。这种方法在需要迭代特定间隔的元素时非常有用。

**~~3. 区间的更多操作。~~**

~~Rust的区间不仅可以用于循环和迭代，还支持一些基本的运算操作。这些操作使得区间更加灵活和强大。~~

**~~包含检查~~**

~~区间可以用来检查某个值是否在其范围内。我们可以通过使用区间的contains方法实现这一功能。~~

1. ~~let range = 1..=5;~~
2. ~~let num = 3;~~
3. *~~// 检查num是否在range内~~*
4. ~~if range.contains(&num) {~~
5. ~~println!("{} is in the range", num);~~
6. ~~}~~

~~在这个例子中，我们使用了contains方法来检查数字3是否在1..=5的范围内。~~

**~~区间长度~~**

~~Rust的区间可以使用 len 方法获取区间的长度。~~

1. ~~let range = 1..=5;~~
2. ~~println!("{}", range.len());~~*~~// 输出 5~~*

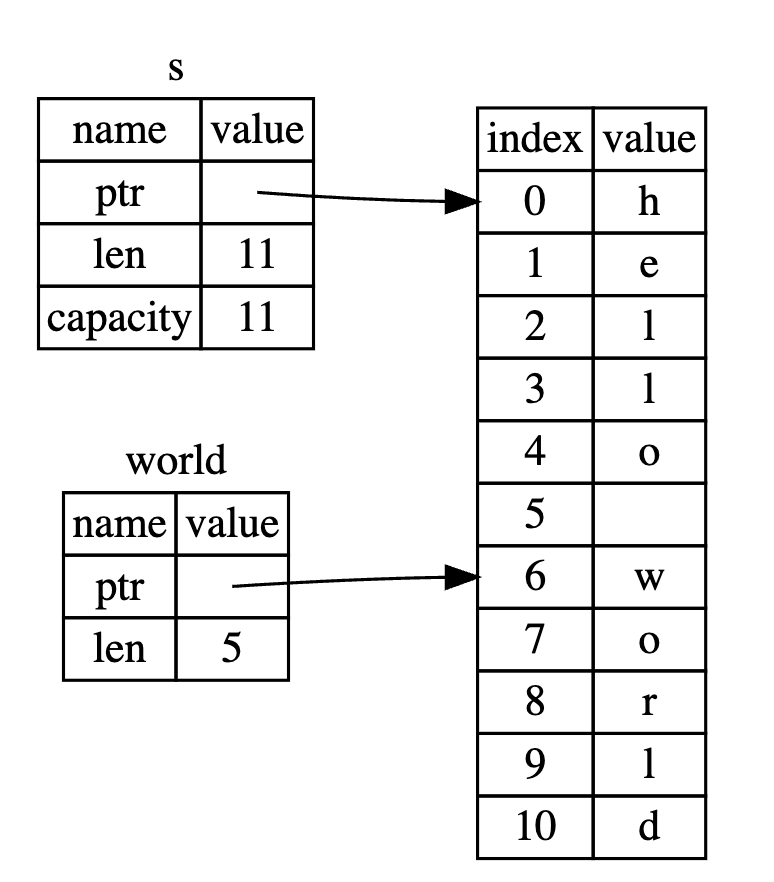
~~在这个例子中我们通过len方法获得了区间[1,5]（左闭右闭）的长度，应为5.~~

### 2.3.4 切片

**1. 切片的定义。**

切片是Rust中一种允许引用集合部分元素的数据类型。切片并不实际拥有它们所包含的数据，它是一个指向数组一部分的引用。数组与切片在内存中的关系如下图所示。

TODO



**2. 切片的创建和使用。**

在Rust中，切片可以通过引用数组或其他切片的一部分来创建。

1. let numbers = [1, 2, 3, 4, 5];
2. *// 创建一个包含前两个元素的切片*
3. let first\_two = &numbers[..2];
4. *// 创建一个包含除了前两个元素之外所有元素的切片*
5. let rest = &numbers[2..];
6. *// 创建一个包含中间三个元素的切片*
7. let middle = &numbers[1..4];

在这个例子中：

first\_two是一个切片，包含了数组numbers的前两个元素。[..2]表示从起始元素到索引2之前的所有元素。

rest是另一个切片，包含了数组numbers从索引2开始的所有元素。[2..]表示从索引2到数组末尾的所有元素。

middle是一个包含中间三个元素的切片，由[1..4]指定，表示从索引1开始到索引4之前的所有元素。

切片在处理字符串、数组或其他集合类型时提供了强大的数据访问能力，同时它具有极高的效率，因为它无需复制原始数据。

**~~3. 切片与所有权~~**

~~切片机制在Rust中的设计体现了语言对内存安全的重视。通过切片，Rust能够在不转移所有权的情况下，安全地提供对数组或集合部分元素的引用。也就是说，切片的创建和使用不会导致数据的移动或复制，这保证了数据访问的安全性。~~

1. ~~fn slice\_sum(slice: &[i32]) -> i32 {~~
2. ~~slice.iter().sum()~~
3. ~~}~~
4. ~~let data = [1, 2, 3, 4, 5];~~
5. ~~let sum = slice\_sum(&data[1..4]);~~
6. ~~println!("Sum: {}", sum);~~*~~// 输出: Sum: 9~~*

~~在这个例子中，slice\_sum函数接收一个指向整数数组一部分的不可变切片，并返回这些元素的和。通过传递切片给函数，我们避免了整个数组的复制，同时函数内部也无法修改原始数据，确保了操作的安全性。~~

**4. 切片的可变与不可变性。**

Rust中的切片分为可变切片和不可变切片。 不可变切片只允许读取切片中的数据。可变切片允许修改切片中的数据，但这要求切片引用的原始数据也必须是可变的。

1. let mut data = [1, 2, 3, 4, 5];
2. *// 不可变切片*
3. let slice\_immutable = &data[1..3];
4. *// 可变切片*
5. let slice\_mutable = &mut data[2..4];
6. slice\_mutable[0] = 6;

在上述例子中，slice\_immutable是一个不可变切片，只能用于读取data数组中索引1到3之前元素的值。slice\_mutable是一个可变切片，允许修改索引2到4之间的元素。

~~在Rust中，可以有多个不可变引用（&T）到某个数据，或者一个可变引用（&mut T），但不能同时拥有。这一规则同样适用于切片。当创建一个可变切片时，实际上是创建了一个对原始数据的可变引用。这意味着，在该可变引用存在期间，不能创建任何其他引用（无论是可变的还是不可变的）到原始数据。这保证了在使用可变切片修改数据时，没有其他引用会意外地读取或修改这些数据，从而防止了数据竞争。~~

1. ~~fn main() {~~
2. ~~let mut data = [10, 20, 30, 40, 50];~~
3. *~~// 创建一个可变切片，引用数组的中间三个元素~~*
4. ~~let slice\_mut = &mut data[1..4];~~
5. *~~// 修改可变切片中的第一个元素~~*
6. ~~slice\_mut[0] = 25;~~*~~// data现在是 [10, 25, 30, 40, 50]~~*
7. ~~println!("Updated data: {:?}", data);~~
8. ~~}~~

~~在这个例子中，我们首先定义了一个可变数组data。然后，我们创建了一个可变切片slice\_mut，它引用了数组data的中间三个元素。通过修改这个切片中的元素，我们实际上修改了原始数组data中的元素。在修改切片的过程中，Rust保证了没有其他引用到data，这遵循了Rust的借用规则，确保了数据的安全性和一致性。~~

### 2.3.5 结构体与枚举类型简介

**1. 结构体类型简介**

结构体是 Rust 中用于创建自定义数据类型的核心元素。它可以将多个具有不同意义的数据项组合在一起，形成一个复合数据类型。结构体通常用于描述一个具有多个属性的对象。Rust 中的结构体有：命名字段结构体、元组结构体、单元结构体。

命名字段结构体的每个字段都有一个名称。如下面定义的Student结构体包含name和age两个字段。

1. struct Student {
2. name: String,
3. age: u32,
4. }

元组是一种特殊的结构体，其字段没有名称，按字段的索引位置访问。

单元结构体是没有字段的结构体，常用作类型标记。

1. struct UnitStruct;
2. **枚举类型简介**

枚举是 Rust 中定义一组相关常量的方式。枚举允许程序中的一个变量可以是多个不同类型之一。下面定义的枚举体Shape包含了Circle、Rectangle、Triangle三种子类型。

1. enum Shape {
2. Circle(f64),        *// 园的半径*
3. Rectangle(f64, f64), *// 长方形的宽度和高度*
4. Triangle(f64, f64, f64), *// 三角形三条边的长度*
5. }

## 2.4 集合类型

Rust提供了多种集合类型来存储具有逻辑或者数据关系的多个值。与数组和切片不同，集合可以动态地增长或缩小其容量。Rust标准库中最常用的集合类型包括向量和散列表。

### 2.4.1 向量

向量（Vector）是一个长度可变的列表，存储多个同类型的值。它允许随时在列表末尾添加或移除元素。

Rust中可以使用Vec::new方法创建一个空向量，或使用vec!宏初始化含有初始值的向量。

1. let v: Vec<i32> = Vec::new(); *// 创建一个空的向量*
2. let v2 = vec![1, 2, 3]; *// 使用宏创建并初始化一个向量*

在这个例子中，我们使用Vec::new 方法创建了一个空向量，使用vec!宏创建了一个初始化为1,2,3三个元素的向量。

我们可以使用索引对向量内的元素进行访问。由于向量的长度可能会在程序运行的过程中动态修改，因此无法在编译阶段确定索引是否越界。Rust在运行时动态检查索引是否越界。

1. *// 创建一个包含整数的向量*
2. let mut numbers = vec![1, 2, 3, 4, 5];
3. *// 使用[]操作符访问向量的元素*
4. let first\_number = numbers[0]; *//first\_number为1*
5. *// 下面这行代码会导致越界panic，因为索引超出了向量的范围*
6. *// let sixth\_number = numbers[5];*
7. *// 使用get方法访问向量的元素*
8. let sixth\_number\_option = numbers.get(5); *//sixth\_number\_option为None*

在上面这个例子中，我们首先创建了一个包含数字1到5的可变整数向量，然后通过索引直接访问向量的第一个和第六个元素，由于向量只有5个元素，因此访问第6个元素会触发panic。

我们还可以通过get方法访问向量的元素（在第10行），这将返回一个Option<T>类型的None，表示该索引下的元素不存在。

~~向量也支持迭代遍历。Rust提供了for语句对向量进行迭代遍历。~~

1. ~~let v = vec![1,2,3];~~
2. ~~for i in &v {~~
3. ~~println!("{}", i);~~
4. ~~}~~

~~在这个例子中，我们使用for循环和迭代器&v，我们可以遍历Vec中的每个元素。这是读取Vec中所有元素的常用方法。~~

### 2.4.2 链表

链表是一种链式数据结构。Rust标准库中提供了一种实现LinkedList。LinkedList是一个双向链表，允许在表的两端插入和删除元素。

1. let mut list = LinkedList::new();
2. *// 添加元素*
3. list.push\_back(1);
4. list.push\_back(2);
5. list.push\_front(0);
6. *// 遍历元素*
7. for value in &list {
8. println!("{}", value);
9. }
10. *// 删除元素*
11. list.pop\_front(); *// 删除第一个元素*
12. list.pop\_back();  *// 删除最后一个元素*

### 2.4.3 散列表和散列集合

**1. 散列表简介**

散列表是一种基于键值对的数据结构，提供了快速的数据查找能力。它能够通过键来访问对应的值。Rust标准库中提供了散列表的实现HashMap。可以通过通过HashMap::new方法创建一个实例，使用insert方法添加键值对。

1. let mut scores = HashMap::new();
2. scores.insert(String::from("Blue"), 10);  *//创建一个Blue-10的键值对*
3. scores.insert(String::from("Yellow"), 50); *// 创建一个Yellow-50的键值对*

在这个例子中，我们首先使用HashMap::new创建了一个空的HashMap，然后使用insert方法添加了两个键值对。insert方法接受一个键和一个值，将它们添加到HashMap中。如果键已经存在，insert会更新键对应的值。

HashMap的值可以通过get方法访问。

1. let mut scores = HashMap::new();
2. scores.insert(String::from("Blue"), 10);  *//创建一个Blue-10的键值对*
3. scores.insert(String::from("Yellow"), 50); *// 创建一个Yellow-50的键值对*
4. if let Some(score) = scores.get("Blue") { *//判断scores中有Blue这个键*
5. println!("Blue's score: {}", score); *//get方法返回Some(score)，并通过if解包Some*
6. }

在这个例子中，我们使用get方法来获得Blue这个键对应的值，这个方法返回一个Option类型的值，如果找到值，返回Some(value), 如果没有找到值，返回None。

**2. 散列集合简介**

散列集合是一个存储唯一元素的集合，主要用于快速检查一个值是否存在于集合中。Rust的标准库中提供了散列集合的实现HashSet。HashSet通过HashSet::new方法创建一个实例，使用insert方法添加元素。

1. let mut books = HashSet::new();
2. books.insert("The Odyssey".to\_string());
3. books.insert("The Iliad".to\_string());

上面例子展示了如何创建一个HashSet并使用insert方法进行初始化。如果尝试插入一个已经存在的值，HashSet不会做任何改变，保证了集合中元素的唯一性。

HashSet提供了contains方法检查一个值是否存在于HashSet中。返回True表示这个元素在集合中，返回False表示元素不在这个集合中。

1. use std::collections::HashSet;
2. let mut books = HashSet::new();
3. books.insert("The Odyssey".to\_string());
4. books.insert("The Iliad".to\_string());
5. if books.contains("The Odyssey") {
6. println!("We have The Odyssey");
7. }

~~遍历元素~~

~~我们同样使用for循环来迭代便利HashSet中的值~~

1. ~~use std::collections::HashSet;~~
2. ~~let mut books = HashSet::new();~~
3. ~~books.insert("The Odyssey".to\_string());~~
4. ~~books.insert("The Iliad".to\_string());~~
5. ~~for book in &books {~~
6. ~~println!("{}", book);~~
7. ~~}~~

### 2.4.4 联合

联合类型Union在声明时和Struct类似，但是它的所有字段都存在于同一内存位置中。例如下面的代码中定义的Shape类型可以被同时解释为circle和rectangle。对Union字段的读操作时不安全的，因为不同字段的不同类型之间可能无法转换。对Union中某个字段的修改会影响其他的字段。

1. union Shape {
2. circle: f64,
3. rectangle: (f64, f64)
4. }
6. let mut circ = Shape { circle: 1.0 };
7. *// 读取 union 的字段总是不安全的*
8. assert\_eq!(unsafe { circ .circle }, 1.0);
9. *// 通过任何字段进行更新都会修改所有字段*
10. circ.rectangle = (1.0, 1.0);

## 2.5 字符串类型

### 2.5.1 字符串字面量

字符串字面量是Rust中存储在二进制输出中的固定字符串。它们是不可变的，并且直接嵌入到最终的程序代码中。

1. let greeting = "Hello, world!"; *// 字符串字面量*
2. println!("{}", greeting);

在这个示例中，变量greeting被绑定到一个字符串字面量"Hello, world!"。字符串字面量的类型是&str，这是一个不可变的字符串切片类型。由于字符串字面量在编译时已知，它们是高效且不可变的，适用于作为常量字符串数据的存储方式。

### 2.5.2 字符串类型

Rust中的字符串处理主要涉及两种类型：String和str。String是一个长度和内容均可改变、的UTF-8编码字符串类型，并通常被用于需要改变字符串内容的场景。

1. let mut s = String::from("hello");
2. s.push\_str(", world!"); *// 在String上追加字符串*
3. println!("{}", s);

创建一个String实例通常使用String::from函数或直接调用String的new方法。String::from用于从字面量创建String，而new方法创建一个空的String。在这个例子中，我们首先通过String::from创建了一个String类型的变量s，其内容为"hello"。之后，使用push\_str方法将", world!"追加到s的末尾，使其变成"hello, world!"。

String提供了丰富的方法来修改其内容，让开发者能够高效地处理字符串数据。

* 添加内容：push\_str和push方法可以向String添加内容。push\_str用于添加字符串切片，而push用于添加单个字符。
* 删除内容：pop方法用于移除并返回String的最后一个字符。如果字符串为空，则返回None。
* 替换内容：replace方法用于替换字符串中的某部分内容。
* 清空字符串：clear方法可以清空String的内容，但保留已分配的内存，便于再次使用。
* 拼接字符串：可以使用+运算符或format!宏来拼接字符串。

1. let mut text = String::from("Hello, Rust programmers!");
2. println!("Original text: {}", text);
3. *// 添加内容*
4. text.push\_str(" Welcome to the text editor.");
5. println!("After addition: {}", text);
6. *// 删除内容*
7. text.pop(); *// 移除最后一个字符*
8. println!("After deletion: {}",text);
9. *// 替换内容*
10. text = text.replace("text editor", "Rust workshop");
11. println!("After replacement: {}", text);
12. *// 拼接字符串*
13. let additional\_info = " Let's start coding.";
14. text = format!("{}{}", text, additional\_info);
15. println!("After concatenation: {}", text);
16. *// 清空字符串*
17. text.clear(); *// 清空文本*
18. println!("After clearing: '{}'", text);

这个例子演示了如何使用Rust编写一个简单的文本编辑器功能。我们从创建一个原始字符串开始，分别对这个文本进行了以下操作：

1. 添加内容：通过push\_str方法添加了一段新的欢迎语。

2. 删除内容：使用pop方法移除了文本的最后一个字符（这里是一个句号）。

3. 替换内容：通过replace方法将"text editor"替换成了"Rust workshop"，以更精确地描述我们的活动。

4. 拼接字符串：使用format!宏将text和additional\_info进行了拼接，添加了一些额外信息。

5. 清空字符串：最后，使用clear方法将text的内容完全清空。

### 2.5.3 字符串切片

字符串切片允许引用字符串的一部分，它的类型名是&str。从类型签名中我们可以看出，切片是对原始字符串数据的不可变借用,这保证了切片的内存安全性。这是因为当我们使用字符串切片时，根据Rust可变与不可变性规则，我们就不能对这个被引用的字符串进行修改，从而规避了无效引用、数据竞争等问题。

另一方面，字符串切片提高了对字符串的访问效率，因为切片只是对字符串数据的一个引用，本身并不包含字符串数据，这意味着我们在使用切片时不必对字符串的数据进行复制。

1. let s = String::from("Hello, world!");
2. let slice = &s[0..5]; *// 引用字符串的一部分*
3. println!("{}", slice);
4. }

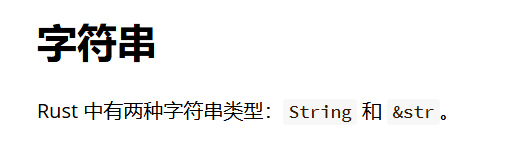
在此示例中，slice是通过对String类型的s变量进行切片操作创建的，它引用了s中的前5个字符"Hello"。这展示了如何使用字符串切片来访问和使用字符串的一部分，而不需要复制或修改原始数据。

1. let mut message = String::from("Hello, ");
2. let name = "John Doe";
3. message.push\_str(name); *// 添加内容*
4. println!("Greeting: {}", message);
5. *// 尝试修改字符串中的部分内容（不可变引用与可变引用冲突示例）*
6. let part\_of\_message = &message[7..]; *// 创建不可变引用*
8. message.clear(); *// 编译错误*
9. println!("Extracted part: {}", part\_of\_message);
11. println!("Message after clear: '{}'", message);

在上面这个例子中，我们在第8行对创建的字符串使用了一个切片，而在第10行对字符串进行了修改，这将导致第8行的引用发生变化，产生一个不安全的行为。这违反了Rust不可变引用与可变修改不可同时出现的规则，因此会产生编译错误。

### 2.5.4 更多字符串类型

[字符串 String - 通过例子学 Rust 中文版](https://rustwiki.org/zh-CN/rust-by-example/std/str.html)



## 2.6 变量与常量

变量和常量（constant）是程序存储和表示数据的方式。变量与常量的区别在于，变量允许程序在运行时修改其对应的值，而常量的值在程序编译的时候就已经确定下来，并且不能改变。变量为程序引入了灵活性和动态性，而常量则用于表示不可变的、常态下固定的值。

### 2.6.1 变量与可变性

Rust通过let绑定的方式创建变量，如代码片段ch2\_7.rs所示，将值赋给变量标识符（Identifier）的操作称作绑定（binding）。换言之，绑定创建了一个新的变量标识符值的映射关系。变量标识符是以大小写字母或下划线开头，后续以空字符或若干大小写字母、数字、下划线构成的串。按照Rust社区惯例，通常用蛇形命名法（snake case）——变量名中的所有字母小写，单词间用下划线分隔的格式书写变量名。

Rust将变量的可变性分为可变和不可变。如代码片段ch2\_7.rs所示，显示标注“mut”关键字的变量为可变变量（mutable variable）；未标注“mut”关键字的变量为不可变变量（immutable variable）。可变变量的值在运行时可以被改变，而不可变变量的值在运行时不允许被改变。同一个可变变量或不可变变量能够在运行时进行多次绑定。

1. fn main() {
2. *// The value of mutable variables can be changed.*
3. let mut \_x = 1;
4. \_x = 2; *// ok*
5. *// The value of immutable variables cannot be changed.*
6. let \_y = \_x;
7. *// \_y = \_x; // not ok*
8. }

代码片段ch2\_8.rs

在定义变量时，通常使用let语句给变量绑定一个字面量、常量或其他变量作为变量的初始值；或者在定义变量时不进行赋值，但在使用该变量之前的后续代码中必须给未赋值的变量赋值。使用未初始化的变量是未定义行为。出于程序正确性和安全性，Rust禁止使用未经初始化的变量。

### 2.6.2 作用域与变量遮蔽

在Rust语言程序设计课程的第一节课上，你发现授课老师和你的室友重名了。在课堂上，你能够清楚地知道，其他同学在向老师提问时提到的“X老师”不可能指的是你的室友；在宿舍里，你也能够清楚地知道，“X”同学不可能指的是你的Rust语言程序设计课程的授课老师。

在上面的例子中，我们通过不同的生活场景来区分相同名字的人物。在程序设计的过程中，我们也有可能写下名字相同，但语义不同的变量。

类似上面例子，作用域（scope）定义了代码块中变量和其他资源的可见性，并为区分同名变量提供了一种机制。在Rust中，作用域由代码块中的“{}”表示。如代码片段ch2\_8.rs所示，在变量的作用域内，变量可见，即可以对该变量进行操作；在变量的作用域外，变量不可见，无法对该变量进行操作。作用域规则确保了变量在其有效范围内可见，避免变量在无效的地方被访问导致未定义行为。

1. fn main() {
2. {
3. *// In the classroom,*
4. *// the one callled "marisa" is a teacher.*
5. let marisa = "teacher";
6. assert\_eq!(marisa, "teacher");
7. }
8. *// assert\_eq!(marisa, "teacher");*
9. *// Not ok; marisa is not valid here.*
10. {
11. *// In the dormitory,*
12. *// The one callled "marisa" is my roommate*
13. let marisa = "roommate";
14. assert\_eq!(marisa, "roommate");
15. }
16. }

代码片段ch2\_9.rs

如果你的那位和授课老师重名的室友碰巧也在修读Rust语言程序设计课程，并在课堂上碰巧坐在了你的旁边，你在和他讨论课程内容时，“X同学”一定指的是你的室友。在讨论时，你们发现一些有疑问的地方，决定当即向老师提问，在提问时，“X”老师一定指的是授课老师。在提问结束后，你继续和你的室友讨论课程内容。在上述的讨论过程当中，提问部分是被嵌套在讨论中的。类似地，Rust中的作用域也会有嵌套的情形。

当Rust中的作用域发生嵌套，即“{}”内有别的“{}”时，被嵌套的作用域称为内层作用域，内层作用域外的作用域称为外层作用域。外层作用域内的变量在内层作用域中仍然可见，但内层作用域中的变量在外层作用域中不可见。当内层作用域中的变量与外层作用域中的变量重名时，外层作用域的同名变量将在内层作用域中被遮蔽（shadowing），即，在内层作用域中，外层作用域的同名变量不可见，只有在内层定义的同名变量可见，外层作用域的同名变量像是被内层作用域的同名变量遮蔽了一般变得不可见。在同一层作用域内，后定义的同名变量也能将先前定义的同名变量遮蔽，这相当于定义了一个新的变量。代码片段ch2\_9.rs展示了上述情形。添加对代码的说明

1. fn main() {
2. *// In the classroom, your roommate sits beside you.*
3. *// When talking with your roommate, "marisa" refers to your roomate.*
4. let marisa = "roommate";
5. {
6. assert\_eq!(marisa, "roommate");
7. *// When asking questions to the teacher, "marisa" refers to the*
8. *// teacher.*
9. let marisa = "teacher";
10. assert\_eq!(marisa, "teacher");
11. }
12. *// After asking questions, you talk with your roomate, and "marisa"*
13. *// refers to your roommate.*
14. assert\_eq!(marisa, "roommate");
15. *// When your roommate leaves, "marisa" refers to the teacher.*
16. let marisa = "teacher";
17. assert\_eq!(marisa, "teacher");
18. }

代码片段ch2\_10.rs

遮蔽机制允许在同一作用域或嵌套作用域内多次使用相同的变量名，使开发人员能更灵活的命名变量和组织代码，使逻辑表述更加清晰。

### 2.6.3 常量

常量（Constants）是在编译时就能确定的值，并且值不能更改。Rust中，用“const”表示常量，如代码片段ch2\_10.rs所示，在定义常量时，需要显示标注常量的类型，并且只能用具有常量值的表达式赋初值。常量的标识符遵从大写蛇形命名法（upper snake case），标识符中的单词都由大写字母表示，单词之间使用下划线‘\_’分隔。常量的标识符常量在定义时，必须给出常量的值。常量在定义后，不能重新赋值或绑定。

1. const C: i32 = 1;
2. fn main() {
3. let x = 1;
4. const X: i32 = C;
5. *// const Y: i32 = x;  // Not ok.*
6. *// const Z: i32;  // Not ok.*
7. *// let C = 1; // Not ok.*
8. *// C = 1;  // Not ok.*
9. assert\_eq!(C + X + x, 3);
10. }

代码片段ch2\_11.rs

### 2.6.4 字面量

字面量（literal）是程序设计语言中表示固定值的符号标注。Rust中常用的字面量包括整数值、浮点数值、布尔值、字符和字符串等。如代码片段ch2\_11.rs，Rust中的整数字面量可以由二进制（第2行）、八进制（第3行）、十进制（第4行）、十六进制（第5行）表示。整数和浮点数字面量的类型可以在其尾部标注（第4行、第5行、第7行）。一般地，整数字面量的默认类型是i32，浮点字面量默认的类型是f64。Rust中的字符字面量（第9行）用单引号表示，字符串字面量（第10行）用双引号表示。

1. fn main() {
2. let \_binary\_literal = 0b1010;
3. let \_octal\_literal = 0o12;
4. let \_decimal\_literal = 10i32;
5. let \_hex\_literal = 0xA\_i64;
6. let \_float64\_literal = 3.14;
7. let \_float32\_litera = 3.14f32;
8. let \_bool\_literal = true;
9. let \_char\_literal = 'c';
10. let \_str\_literal = "str";
11. }

代码片段ch2\_12.rs

### 2.6.5 静态变量

静态变量（static variable）指唯一存在于整个程序执行期间，并在程序启动时即被初始化，即只会被初始化一次的变量。如代码片段ch2\_11.rs所示，Rust中的静态变量在定义时使用“static”关键字 ，并需要显示指定类型和初始的常量值。Rust中的静态变量分为不可变静态变量和可变静态变量。在定义静态变量时，标注有“mut”关键字的静态变量是可变静态变量。由于静态变量唯一存在于整个程序执行期间，多线程环境下对可变变量的访问和修改可能存在竞争条件。Rust中对可变静态变量的访问和修改都需要在unsafe代码块中完成。

1. static S: i32 = 1;
2. fn main() {
3. let x = 2;
4. static mut Y: i32 = S;
5. unsafe { Y = x };
6. assert\_eq!(S + unsafe { Y }, 3);
7. }

代码片段ch2\_13.rs

利用静态变量唯一存在于整个程序执行期间且仅会被初始化一次的性质，可以用可变静态变量实现普通变量无法实现的功能。Rust语言程序设计课程的授课老师想要知道有多少学生会出勤今天的课程，于是他实现了一个代码片段ch2\_12.rs中的计时器，只要有一位新的同学进入教室，就执行一次counter函数。counter函数中的COUNT静态变量在counter函数返回后依然存在，并在重新进入counter函数时COUNT的值不会被再次初始化，故counter函数能够借助COUNT变量成功实现计数器的功能。

1. fn main() {
2. assert\_eq!(1, counter());
3. assert\_eq!(2, counter());
4. assert\_eq!(counter() + 1, counter());
5. }
6. fn counter() -> i32 {
7. static mut COUNT: i32 = 0;
8. unsafe {
9. COUNT += 1;
10. COUNT
11. }
12. }

代码片段ch2\_14.rs

## 2.6 类型转换

有时候，变量或表达式的类型不匹配程序的需求。类型转换（Type Cast）可以将变量转换为需要的类型。本小节将介绍Rust基本类型之间的类型转换操作。

### 2.6.1 转换表达式

Rust基本类型之间的类型转换通常通过as操作符实现。如代码片段ch2\_13.rs所示，不同的整数类型、浮点数类型之间的变量能够相互转换类型。

1. fn main() {
2. *// cast x: i32 as y: i8*
3. let x = 0i32;
4. let y = x as i8;
5. assert\_eq!(y, 0);
6. *// cast x: i32 as y: u8, will be cut short*
7. let x = 0x7fffff61i32;
8. let y = x as u8;
9. assert\_eq!(y, 0x61u8);
10. *// cast y: u8 as char; only u8 can be cast as char*
11. let u = y as char;
12. assert\_eq!('a', u);
14. *// cast bool as integer;*
15. assert\_eq!(true as i32, 0x1);
16. assert\_eq!(false as i32, 0x0);
17. }

代码片段ch2\_15.rs

### 2.6.2 类型转换处理

在使用as操作符进行基本类型的类型转换时，我们需要尤其注意类型转换可能带来的副作用及Rust对as操作符的限制：

1. 精度更高、位数更长的基本类型变量在转换为精度更低、位数更短的类型时将会被截断，导致精度降低或高位数据丢失；
2. 只允许u8类型的变量转换为char类型。其他整数类型的变量的值可能落在char类型变量值域之外，导致转换出错；
3. 数字类型不能转换为bool类型。这会使程序语义不明确。

### 2.6.2 类型转换中的安全问题

## 2.6 小结

本章主要介绍了Rust程序设计语言中的基本数据类型及变量的相关概念。程序设计语言通过变量建立标识符值的映射关系。变量对值的抽象机制让开发人员对值的处理从具体的数据表示变为对抽象符号的操作，使得开发人员能够更灵活地表示和处理数据。Rust程序设计语言的基本类型提供了和其他常见程序设计语言类似的基本类型，包括整数类型、浮点数类型、字符类型和布尔类型。与其他常见程序设计语言不同的是，Rust程序设计语言还提供了单元类型、Never类型作为特殊类型。在这些类型上Rust定义了一套相对严格的类型转换机制，构成了Rust安全类型系统的基石。Ralf Jung等[1]已经形式化地证明Rust的主要部分构成的子集是安全的。

掌握程序设计语言中变量和类型的概念对于学习程序设计语言至关重要，这些概念是大部分程序设计语言中通用的基本入门知识，也是成为成熟Rust开发人员首先需要学会的内容。希望通过本章内容的学习，读者能够有效地创建、修改和管理程序中的数据，理解变量和类型对于程序的功能和意义，了解程序设计中广为接受的设计规范和Rust特有的安全限制。

[1] Jung R, Jourdan J H, Krebbers R, et al. RustBelt: Securing the foundations of the Rust programming language[J]. Proceedings of the ACM on Programming Languages, 2017, 2(POPL): 1-34.