# 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

## 2.3 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。这两种特殊类型和基础类型一样简单，同样也是Rust类型系统中不可或缺的一部分。单元类型和never类型增强了Rust类型系统的表达能力。

### 2.3.1 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

### 2.3.2 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。

### 2.3.2 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

## 2.4 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。

### 2.4.1 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。

### 2.4.2 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。

### 2.4.3 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

### 2.4.4 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

### 2.4.5 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

## 2.5 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。

### 2.5.1 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

### 2.5.2 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 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.