# 第三章 语句与表达式

本章要点： 语句；表达式；运算优先级；条件控制；循环控制

本章节将介绍语句（statement）和表达式（expression）。语句和表达式是程序设计语言中两个基本的概念，它们对于语言的设计和程序的构建都具有重要的意义。

## 3.1 引言

一名擅长烹饪家常菜的大厨米斯琪正要做一道番茄炒蛋。大厨切好番茄、调配蛋液、点火热锅、加入食用油，待油热后加入蛋液、翻炒、加入番茄、加入食用盐、待火候合适时出锅装盘。一盘色香味俱全的番茄炒蛋制作完成。这些烹饪的步骤类似程序设计语言中的语句，表示程序的执行流程。在程序设计语言中，语句是一个完整的执行单元，它执行特定操作但不返回值（或返回无意义的值），负责完成某个任务的单个指令或一组指令。

然而，番茄炒蛋之所以如此美味，不仅仅在于正确的操作步骤，更在于食材的处理和组合。大厨需要挑选新鲜的食材、在菜品中添加合适且适量的调味料等等才能烹饪出完美的料理。这就像程序设计中的表达式，表达式负责程序的计算，产生具体的值或修改程序的状态。在程序设计中，语句和表达式的良好结合能够带来清晰、高效、优雅的程序。

## 3.2 表达式

表达式是由运算符（operator）和操作数（operand）组成的一个可求值的代码片段，它的求值结果可以用到其它表达式或语句中。Rust程序设计语言中的表达式可粗略分为原子表达式（Atomic Expression）、算术表达式（Arithmetic Expression）、位运算表达式（Bitwise Expression）、比较表达式（Comparison Expression）、逻辑表达式（Logical Expression）、块表达式（Block Expression）、函数调用表达式（Function Call Expression）、方法调用表达式（Method Call Expression）等。每个表达式都可以嵌入到另一种表达式中，组成表达能力更强的表达式。本节内容将介绍较为基础几种的表达式。

### 3.2.1 基本表达式

**1. 原子表达式。**

由单个变量、常量或字面量构成的表达式被称为原子表达式。原子表达式是构成更为复杂表达式和语句的基本单元。我们可以用下面的式子表示原子表达式的构成。原子表达式产生的值就是组成该表达式的变量或常量在程序状态中对应的值，或者字面量本身所代表的值。

代码片段ch3\_1.rs第4行的“Zero”、“zero”是变量标识符的同时，也组成了原子表达式。

1. const ZERO: i32 = 0;
2. fn main() {
3. let \_zero = ZERO;
4. }

代码片段ch3\_1.rs

**2. 算数表达式。**

Rust程序设计语言中的运算符有加（+）、减（-）、乘（\*）、除（/）、求余（%）、取负（-），基础类型中整型和浮点类型的数据可以进行算数计算。算数表达产生的值是运算符两侧表达式的值执行相应运算后得到的结果。比较特殊的是，取负（-）运算只需要1个子表达式作为操作数。

代码片段ch3\_2.rs第4~10行的“ZERO + one”、“one - 0”、“1 \* 0”、“0 / 1”、“0 % 1”、“0 + 1 + 1”、“-one”均是符合规范的算数表达式。

1. const ZERO: i32 = 0;
2. fn main() {
3. let one = 1;
4. assert\_eq!(1, ZERO + one);
5. assert\_eq!(1, one - 0);
6. assert\_eq!(0, 1 \* 0);
7. assert\_eq!(0, 0 / 1);
8. assert\_eq!(0, 0 % 1)
9. assert\_eq!(2, 0 + 1 + 1);
10. assert\_eq!(-1, -one);
11. }

代码片段ch3\_2.rs

**3. 位运算表达式。**

位运算表达式负责计算两个整型数据的二进制表示按位运算得到的值。基础类型中的整型、布尔型数据可以参与位运算。

Rust提供的位运算表达式支持以下运算：

1. 按位与（）：对操作数的二进制编码的每一位执行与操作，只有在两个操作数的对应位都为‘1’时结果的对应位才为‘1’，否则为‘0’。代码片段ch3\_3.rs第2行的“0x7f & 0xff”计算过程如图3.1所示。

0

1

1

1

1

1

1

1

=0x7f

1

1

1

1

1

1

1

1

=0xff

&

0

1

1

1

1

1

1

1

=0x7f

图3. 1 按位与运算示例

1. 按位或（）：对操作数的二进制编码的每一位执行或操作，两个操作数的对应位中有一个是‘1’时结果的对应位为‘1’；两个操作数对应位都是‘0’时，结果的对应位为‘0’。代码片段ch3\_3.rs第3行的“0x7f | 0xff”计算过程如图3.2所示。

0

1

1

1

0

0

0

0

=0x70

1

1

1

1

0

0

0

0

=0xf0

|

1

1

1

1

0

0

0

0

=0xf0

图3. 2 按位或运算示例

1. 按位异或（）：对操作数的二进制编码的每一位执行异或操作，两个操作数的对应位若不相同，结果的对应位为‘1’；两个操作数对应位相同，结果的对应位为‘0’。代码片段ch3\_3.rs第4行的“0x7f | 0xff”计算过程如图3.3所示。

0

1

1

1

1

1

1

1

=0x7f

1

1

1

1

1

1

1

1

=0xff

^

1

0

0

0

0

0

0

0

=0x80

图3. 3 按位异或运算示例

1. 逻辑左移（）：对左移符号左侧操作数的二进制编码向左移动右操作数指定的位数，舍弃左侧溢出位，右侧补0. 代码片段ch3\_3.rs第5行的“0x7f 1”计算过程如图3.4所示。

0

1

1

1

1

1

1

1

=0x7f

1

1

1

1

1

1

1

1

0

=0xfe

0

舍弃溢出位

图3. 4 逻辑左移运算示例

1. 逻辑右移（）：对右移符号左侧操作数的二进制编码向右移动右操作数指定的位数，舍弃右侧溢出位，左侧补0. 代码片段ch3\_3.rs第6行的“0x7f 1”计算过程如图3.5所示。

0

1

1

1

1

1

1

1

=0x7f

1

0

0

1

1

1

1

1

1

=0x3f

图3. 5 逻辑右移运算示例

1. fn main() {
2. assert\_eq!(0x7f, 0x7f & 0xff);
3. assert\_eq!(0xff, 0x70 | 0xf0);
4. assert\_eq!(0x80, 0x7f ^ 0xff);
5. assert\_eq!(0xfe, 0x7f << 1);
6. assert\_eq!(0x3f, 0x7f >> 1);
7. }

代码片段ch3\_3.rs

**4. 比较表达式。**

比较表达式计算两个表达式的值的大小关系，产生一个bool类型的值。若产生的值为true，那么比较表达式所表示的大小关系成立；若产生的值为false，则关系的大小关系不成立。

Rust提供的比较表达式支持以下运算：

1. 相等性比较（==）：若“==”两侧的子表达式的值相同，比较表达式将得到true值；若“==”两侧的子表达式的值不相同，则得到false值。如代码片段ch3\_4.rs第5行的“x == y”，x的值为10，y的值为12，故“x == y”的值是false。
2. 不等性比较（!=）：若“!=”两侧的子表达式的值不相同，比较表达式将得到true值；若“==”两侧的子表达式的值相同，则得到false值。如代码片段ch3\_4.rs第6行的“x != y”，x的值为10，y的值为12，故“x != y”的值是true。
3. 小于（<）：若“<”左侧子表达式的值小于“<”右侧子表达式的值，比较表达式将得到true值；否则得到false值。如代码片段ch3\_4.rs第7行的“x < y”，x的值为10，y的值为12，故“x < y”的值是true。
4. 小于或等于（<=）：若“<=”左侧子表达式的值小于等于“<=”右侧子表达式的值，比较表达式将得到true值；否则得到false值。如代码片段ch3\_4.rs第8行的“x <= y”，x的值为10，y的值为12，故“x <= y”的值是true。
5. 大于（>）：若“>”左侧子表达式的值大于“>”右侧子表达式的值，比较表达式将得到true值；否则得到false值。如代码片段ch3\_4.rs第9行的“x > y”，x的值为10，y的值为12，故“x > y”的值是true。
6. 大于或等于（>=）：若“>=”左侧子表达式的值大于等于“>=”右侧子表达式的值，比较表达式将得到true值；否则得到false值。如代码片段ch3\_4.rs第10行的“x >= y”，x的值为10，y的值为12，故“x >= y”的值是true。

当且仅当参与比较运算的表达式产生的值的类型能够进行比较时，比较表达式才能得到有意义的值。具体而言，具备相等关系的类型能够参与“==”和“!=”运算；具备偏序关系（Partial Order）的类型能够参与“<” 、“<=”、“>”、“>=”运算。

1. fn main() {
2. let x = 10;
3. let y = 12;
5. assert\_eq!(x == y, false);
6. assert\_eq!(x != y, true);
7. assert\_eq!(x < y, true);
8. assert\_eq!(x <= y, true);
9. assert\_eq!(x > y, false);
10. assert\_eq!(x >= y, false);
11. }

代码片段ch3\_4.rs

**5. 逻辑表达式。**

逻辑表达式用于执行逻辑运算，产生布尔类型的值。参与逻辑运算的子表达式也必须能够产生布尔类型的值。

Rust提供的逻辑表达式支持以下运算：

1. 逻辑与（&&）：当且仅当参与运算的两个子表达式的值都是true时，逻辑与表达式得到true值；否则逻辑与表达式得到false值。如代码片段ch3\_5.rs第4行的“x>1 && x>2”。
2. 逻辑或（||）：当参与运算的两个子表达式至少有一个的值都是true时，逻辑与表达式得到true值；否则逻辑与表达式得到false值。如代码片段ch3\_5.rs第5行的“x>2 || x>3”。
3. 逻辑非（!）：当参与运算的子表达式的值为true时，逻辑非表达式得到false值；当参与运算的子表达式的值为false时，逻辑非表达式得到true值；如代码片段ch3\_5.rs第6行的“! x > 2”。
4. fn main() {
5. let x = 3;
7. assert\_eq!(x > 1 && x > 2, true);
8. assert\_eq!(x > 2 || x > 3, true);
9. assert\_eq!(!x > 2, false);
10. }

代码片段ch3\_5.rs

在介绍上面的五类表达式（原子表达式、算数表达式、位运算表达式、比较表达式、逻辑表达式）时，我们定义了一系列运算符表示表达式需要进行的运算。需要两个表达式参与运算的运算符，称作双目运算符（binary operator），例如加法（+）、减法（-） 、 乘法（\*） 、 除法（/）；只需要一个表达式参与运算的运算符，称作单目运算符（unary operator），例如取负（-）、逻辑非（！）。

Rust中的双目运算符两侧的表达式必须能够得到相同类型的值。这是为了确保在运算时类型的安全性。代码片段ch3\_6.rs中的第6行中的变量x: i32和变量y: i8在做加法运算时，需要先将其中一者的类型转换为另一者的类型才能够通过编译器的检查；第7行中的y与f的加法运算也需要确保两者类型一致后才能进行运算。

1. fn main() {
2. let x: i32 = 1;
3. let y: i8 = 1;
4. let f: f32 = 4.0;
5. assert\_eq!(x + y as i32, 2);
6. assert\_eq!(y as f32 + f, 5.0);
7. }

代码片段ch3\_6.rs

### 3.2.2 赋值表达式与运算符优先级

**1. 赋值表达式。**

赋值表达式用于给变量赋予新的值，即修改程序的状态。如下面的式子所示，赋值表达式的左侧为被赋予新值的变量，右侧为一个表达式。表达式的值的类型应与变量的类型一致。在Rust中，只有在绑定变量、初始化变量或给可变变量更新值时，才能够使用赋值表达式。Rust中的赋值表达式得到的值为“()”。

使用“=”的赋值表达式是最为常见的赋值形式，代码片段ch3\_7.rs中的第4行“x = 2”给变量x赋予了一个新的初始值2: i32. 除此之外，Rust还提供了以下几种复合赋值运算符：

1. 加法赋值（+=）：形如的表达式相当于。如代码片段ch3\_7.rs中的第7行“x += x + 1”，相当于“x = x + x + 1”。
2. 减法赋值（-=）：形如的表达式相当于。
3. 乘法赋值（\*=）：形如的表达式相当于。
4. 除法赋值（/=）：形如的表达式相当于。
5. 按位与赋值（&=）：形如的表达式相当于。
6. 按位或赋值（|=）：形如的表达式相当于。
7. 按位异或赋值（^=）：形如的表达式相当于。
8. 逻辑左移赋值（<<=）：形如的表达式相当于。
9. 逻辑右移赋值（>>=）：形如的表达式相当于。
10. fn main() {
11. let mut x;
12. x = 2;
13. assert\_eq!(x, 2);
14. x += x + 1;
15. assert\_eq!(x, 5);
16. }

代码片段ch3\_7.rs

**2. 运算符优先级。**

运算符的优先级决定了表达式中各个运算符的计算顺序。当一个表达式由多个子表达式组成时，我们需要谨慎考虑这些子表达式的计算顺序是否与程序设计的目标一致。

表3. 1 运算符优先级

|  |  |
| --- | --- |
| 运算符 | 结合性 |
| 单目运算符（例如：逻辑非（！）、取负（-）） | / |
| 乘法（\*）、除法（/）、取模（%） | 从左至右 |
| 加法（+）、减法（-） | 从左至右 |
| 逻辑左移（<<）、逻辑右移（>>） | 从左至右 |
| 按位与（&） | 从左至右 |
| 按位异或（^） | 从左至右 |
| 按位或（|） | 从左至右 |
| 比较运算符（==、 !=、 <、 >、 <= 、>=） | 需要圆括号指定运算顺序 |
| 逻辑与（&&） | 从左向右 |
| 逻辑或（||） | 从左向右 |
| 赋值运算符（=、+=、 -=、\*=、/=、%=、&=、|=、^=、<<=、>>=） | 从右向左 |

表3.1给出了前面两小节中介绍的运算符的优先级和结合性。表格中运算符的优先级从上至下依次由强到弱。对于拥有相同优先级的运算符和，形如的表达式：

1. 若运算符和具有从左至右的结合性，那么先计算的值（记为），然后计算的值；如代码片段ch3\_8.rs第6行的“x + y + z”，由于加法和减法运算符优先级相同，结合性是从左至右，故先计算“x + y”的值（记为），后计算“ - z”的值。
2. 若运算符和具有从右至左的结合性，那么先计算的值（记为），然后计算的值；如代码片段ch3\_8.rs第8行的“x = y = z”，由于赋值运算符的结合性是从右向左，故先计算“y = z”的值得到“()”，同时y的值被修改为z的值，后计算“x = ()”的值，并得到x的值为()。
3. 若运算符和的结合性需要圆括号指明，那么和、相同运算优先级的符号在连续出现时需要用圆括号显示指出运算顺序。即形如的表达式只有在写成或时，该表达式才能够被编译器解析生成无歧义的指令。如代码片段ch3\_8.rs第12行、第13行的“(y > z) == false”、“(y > z) == (z > y)”，“<”、“>”和“==”都具备相同的优先级。如果去掉圆括号，第12行的“y > z == false”按照运算符从左至右结合性依然能够得到一个正确的结果，但对第13行的“y > z == z > y”而言，无论是按照从左至右还是从右至左的结合性都无法得到正确的计算结果。为了确保比较表达式语义的无歧义性，Rust程序设计语言规定比较运算符在连续出现时，必须用圆括号标记子表达式的运算顺序。
4. fn main() {
5. let x = 1;
6. let mut y = 2;
7. let z = 3;
8. assert\_eq!(x + y - z, (x + y) - z);
9. let x = y = z;
10. assert\_eq!(x, ());
11. assert\_eq!(y, 3);
12. assert\_eq!((y > z) == false, true);
13. assert\_eq!((y > z) == (z > y), true);
14. }

代码片段ch3\_8.rs

### 3.1.4 块表达式和语句块

在第二章介绍变量作用域的相关内容中，我们已经了解了块（block）的概念。Rust中的块是由一对花括号“{}”包裹起来的代码块。Rust中的块主要分为两种：块表达式（expression block）和语句块（statement block）。

**1. 块表达式。**

在Rust中，块表达式是一个匿名作用域，其中的代码能够产生一个值。块表达式的最后一个表达式被称为最终操作数（final operand），块的类型是最终操作数的类型，最终操作数的值会成为整个块的值。如果块表达式省略了最终操作数，则块的类型为“ ()”。代码片段ch3\_9.rs的第2~6行的块是一个块表达式。相比于直接书写表达式“(18 \* 24 + 14)\* 60 \* 60)”，第2~6行的块表达式的语义更容易理解。

1. fn main() {
2. let second = {
3. let day = 18;
4. let hour = 14;
5. (day \* 24 + hour) \* 60 \* 60
6. };
7. assert\_eq!(second, (18 \* 24 + 14)\* 60 \* 60);
8. }

代码片段ch3\_9.rs

**2. 语句块。**

[**https://rustwiki.org/zh-CN/reference/expressions/block-expr.html**](https://rustwiki.org/zh-CN/reference/expressions/block-expr.html)

**Rust中似乎没有语句块的概念？块表达式的概念包括了语句块（即没有最终操作数的块表达式）。**

语句块是指一个块中的代码为一系列语句，但整个语句块不产生值。在 Rust 中，语句块通常用于包装多条语句，例如函数体、控制流结构等。代码片段ch3\_9.rs中的第1~9行的代码块是一个语句块，作为main函数的函数体。

### 3.1.5 表达式的副作用

表达式的副作用是指表达式在计算值的同时，产生了对程序状态的影响。例如：赋值表达式在赋值符号右侧子表达式值的同时，也修改了赋值符号左侧变量的值；3.2节的条件控制表达式在执行时会改变程序的控制流；在执行IO操作时，相应的IO调用计算了一个返回值，同时可能也改变了程序外部环境的状态。

在 Rust 中，表达式语句（Expression Statement）是对[表达式](https://rustwiki.org/zh-CN/reference/expressions.html)求值并忽略其结果的语句。如下面的式子所示，表达式语句由以分号结尾的表达式构成。表达式语句中表达式的求值结果将被舍弃，仅保留其副作用，并产生单元类型“()”的值。

在Rust程序设计语言的编程规范中，所有的表达式语句都应当产生副作用。不产生副作用的语句会被Rust编译器视作无效的代码，并给出提示。如代码片段ch3\_10.rs中，第8~10行的语句是由原子表达式组成的语句，它们是不产生任何副作用的语句。

1. const ZERO: i32 = 0;
2. *// no warning on path statements with no effect*
3. #[allow(path\_statements)]
4. fn main() {
5. let zero = ZERO;
6. ZERO;
7. zero;
8. 0;
9. }

代码片段ch3\_10.rs

## 3.2 条件表达式与条件语句

控制流（control flow）是程序执行语句的顺序和方向。程序的执行过程并不总是按照源代码中的顺序一步步执行，而是受到条件、循环和其他控制结构的影响。

在本章开头的例子中，大厨米斯琪为了确保正在烹饪的番茄炒蛋咸淡适宜，需要尝一下锅里汤汁的口味决定是否需要添加更多的盐。在向番茄炒蛋中加盐时，条件为“菜肴太淡”。若“菜肴太淡”成立，则加盐；否则维持原样。这一动作与程序设计语言中流程控制的条件执行相同。程序设计语言中的条件语句通常由条件操作数（condition operand）、真分支（true expression）、假分支（false expression）构成。条件操作数是一个能够得到bool类型值的表达式。真分支、假分支是一段代码。在控制流到达条件语句时，当条件操作数为真，控制流跳转到真分支，执行真分支的代码；当条件操作数为假，控制流跳转到假分支，执行假分支的代码。

### 3.2.1 条件表达式

Rust程序设计语言提供的条件表达式有if表达式，match表达式和if-let表达式。if-let表达式将在第六章中介绍。本章先行介绍Rust中的if表达式和match表达式。

如下面的式子所示，if表达式包含if关键字、一个条件操作数、一个块表达式。或者，if表达式包含一个条件操作数、一个块表达式、else关键字、块表达式或另一个if表达式或另一个if-let表达式。在条件操作数之后的块表达式是if表达式的真分支；在else关键字之后的表达式或块表达式是if表达式的假分支。

代码片段ch3\_11.rs粗略地模拟了向番茄炒蛋中加盐的过程。第7~11行（不包括结尾的分号）是一个if表达式，第7行的“salt < MOEDERATE\_SALT”是条件操作数，第7~9行的块表达式是真分支，表示番茄炒蛋盐不足时加盐的操作，第9~11行的块表达式是假分支，表示番茄炒蛋盐已经足够时不做任何操作。在第7~11行的if表达式执行结束后，番茄炒蛋中的盐至少已经达到了程序中所定义的适量的标准“MOEDERATE\_SALT”。由于第4行将变量salt初值设置为0，故在程序运行时，第7~9行真分支的块代码被执行，第9~11行假分支的块代码没有执行。

1. const MODERATE\_SALT: u8 = 3;
2. fn main() {
3. let mut salt = 0;
4. salt =
5. if salt < MODERATE\_SALT {
6. salt + MODERATE\_SALT
7. } else {
8. salt
9. };
10. assert!(salt >= MODERATE\_SALT);
11. }

代码片段ch3\_11.rs

 if表达式的求值结果是程序运行时if表达式被执行的块表达式的值。如果没有块被求值， if表达式的求值结果是单元类型“()”。这意味着同时具有真分支和假分支的if表达式，两个分支的块表达式的类型必须是一致的。

Rust程序设计语言并没有提供三元运算符（即形如的表达式），这是因为Rust的if语法构成的是一个表达式，可以直接实现三元运算符的功能，作为一个子表达式构成一个更为复杂的表达式，这也有助于开发人员写出可读性更强的代码。

### 3.2.2 if语句

**Rust中似乎只有if表达式，没有if语句块的概念？ 上一节的内容放在这一节更合适？**

### 3.2.3 match与Option

**1. 原子表达式。**

match表达式包含match关键字、检验对象表达式（scrutinee expression）和若干个匹配臂。如下面的式子所示，每个匹配臂都包含一个模式（pattern），匹配臂的模式通常是检验对象表达式的某个可能值或区间。在程序运行时，程序将按照匹配臂的书写顺序，依次将检验对象表达式的值与匹配臂的模式匹配，直到找到一个匹配成功的匹配臂，控制流将跳转到该匹配臂对应的表达式或块表达式继续执行。在匹配臂的表达式或块表达式执行结束后，控制流将跳出match表达式继续执行后续代码。

如果匹配臂设置有匹配守卫（match guard），那么只有检验对象表达式的值与匹配臂的模式匹配，并且匹配守卫的条件表达式为真，该匹配臂的表达式或块表达式才能够得到执行。

代码片段ch3\_12.rs用match表达式重写了代码片段ch3\_11.rs的if表达式。第6行“match”关键词后的“salt”构成match语句的检验对象表达式。第7行、第8~11行、第12行各自构成一个匹配臂。第7行匹配臂的语义为：若变量“salt”的值为“0”、“1”、“2”中的一者，则求“salt + MODERATE\_SALT”的值。第8~11行匹配臂的模式使用了通配符“\_”,用于匹配所有可能的情况。第8~11行匹配臂设置了一个匹配守卫“if salt > 2 \* MODERATE\_SALT”，该匹配臂的块代码在变量“salt”满足匹配守卫的条件表达式时才会被执行。第12行的匹配臂用于处理其余的情况，即“salt >= MODERATE\_SALT && salt <= 2 \* MODERATE\_SALT”的情形。

1. const MODERATE\_SALT: u8 = 3;
2. fn main() {
3. let mut salt = 4;
4. salt = match salt {
5. 0 | 1 | 2 => salt + MODERATE\_SALT,
6. \_ if salt > 2 \* MODERATE\_SALT => {
7. println!("Too salty!");
8. salt
9. }
10. \_ => salt,
11. };
12. assert!(salt >= MODERATE\_SALT);
13. }

代码片段ch3\_12.rs

如果同一个match表达式下，有两个匹配臂的模式包含了重复的值，包含了重复值并且处在匹配顺序后侧位置的匹配臂是不可达的，Rust编译器能够检测出这种无用的代码并给出提示。

需要注意的是，match表达式所有匹配臂的模式应覆盖检验对象表达式对应类型的值域。例如代码片段ch3\_12.rs第6~13行的match语句，其两个匹配臂分别对应和范围内的无符号整数，覆盖了u8类型无符号整数的所有可能值。在Rust 2018之后的版本，未能覆盖检验对象表达式对应类型值域的match表达式将无法通过编译。这一强制性规则体现了Rust确保代码安全性和完整性的设计理念。

**2. Option<T>类型。**

Option<T>类型是一个枚举类型。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的情况。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

## 3.3 循环语句

大厨米斯琪成功的背后，离不开对炒锅油温的控制。在炒菜热锅的过程中，对油锅加热是一个持续进行的过程。直到油温达到合适的温度，我们才能向油锅中倒入食材。在程序设计语言中，重复执行一部分代码，直到程序状态满足特定情形时才停止的控制流结构被称为循环（loop）。循环通常由循环条件和循环体构成。程序在运行至循环处时，如果程序状态满足循环条件，会按照循环体内的代码顺序，执行循环体内的代码。在每执行完一次循环体内的代码时，控制流会回到循环开始处，重新判断循环条件是否得到满足。若满足，则重复上述行为；若不满足，控制流将跳转到循环的下一条语句继续执行。

Rust提供的循环表达式有loop表达式、while表达式和for表达式。如下面的式子所示，每种循环表达式的开头可以有一个循环标签（loop label）。Rust还提供了break表达式和continue表达式，以支持更复杂的循环控制流的跳转关系。循环标签、break表达式和continue表达式将在loop表达式的相关内容中做详细介绍。

### 3.3.1 loop语句

loop表达式表示一个无限循环（infinite loop），由“loop”关键字和一个块表达式组成，如下面的式子所示。构成loop表达式的块表达式是循环体，程序将会不断重复执行其块表达式中的代码，直到遇到break表达式或程序退出指令才能结束循环。

break表达式由“break”关键字和一个表达式构成，表达式可省略。当程序执行到break表达式时，将会跳转到与break表达式对应的循环表达式的下一条语句或表达式继续执行。break表达式若含有子表达式，与break表达式对应的循环表达式的返回值为该子表达式的值。若break表达式的子表达式被省略，则与break表达式对应的循环表达式的值为“()”。

代码片段ch3\_13.rs模拟大厨米斯琪加热油锅的过程。第7~12行是一个loop表达式，7~12的块表达式是循环体。循环体中的代码在每次执行时，都会给油锅的温度增加“POWER”对应的值（第8行）；第9~10行的if表达式负责判断程序状态是否满足循环条件——“temperature < MODERATE\_TEMPERATURE”。如果循环条件得到满足，控制流将跳转到块表达式的开始处，开始新的一轮循环。如果循环条件不满足，将会执行第10行的break，控制流跳转至loop循环之后的语句（第14行），继续执行余下的代码。7~12行的loop表达式的值为“()”。

1. const POWER: u8 = 40;
2. const MODERATE\_TEMPERATURE: u8 = 150;
3. fn main() {
4. let mut temperature = 20;
5. loop {
6. temperature += POWER;
7. if temperature >= MODERATE\_TEMPERATURE {
8. break;
9. }
10. }
11. assert!(temperature >= MODERATE\_TEMPERATURE);
12. }

代码片段ch3\_13.rs

如下面的式子所示，break表达式还可以包含一个循环标签。循环标签由一个单引号“’”、一个合法的标识符和一个冒号“:”组成。break表达式通常与包含 break表达式的最内层循环体相关联。但是可以使用[循环标签](https://rustwiki.org/zh-CN/reference/expressions/loop-expr.html?highlight=break#loop-labels)指定与break表达式关联的循环体，前提是此循环体必须包含该break表达式。

代码片段ch3\_14.rs展示了一个两重循环。第7~17行的外层循环具有循环标签“cook”，第10~15行的内层循环包含一个break表达式（第12行）。第12行的break表达式通过循环标签“cook”与外层循环绑定，这意味着程序控制流在进入外层循环后，先重复执行外层循环的代码，在程序状态满足第9行的if跳转条件后进入内层循环，重复执行内层循环的代码，直到遇到第12行的break表达式，然后跳转到外层循环“cook”之后的代码，继续运行（第17行）。

1. const POWER: u8 = 40;
2. const MODERATE\_TEMPERATURE: u8 = 150;
3. fn main() {
4. let mut temperature = 20;
5. 'cook: loop {
6. temperature += POWER;
7. if temperature >= MODERATE\_TEMPERATURE {
8. loop {
9. if temperature == MODERATE\_TEMPERATURE {
10. break 'cook;
11. }
12. temperature -= 1;
13. }
14. }
15. }
16. assert!(temperature == MODERATE\_TEMPERATURE);
17. }

代码片段ch3\_14.rs

continue表达式由continue关键字和一个循环标签组成，如下面的式子所示，循环标签可省略。当程序执行到 continue语句 时，相关循环体的当前迭代将立即结束，并将控制流返回到循环体的开始位置。

代码片段ch3\_15.rs第7~14行的loop循环的循环体含有一个continue表达式（第10行）。程序在运行至第10行continue表达式时，控制流将跳转至loop循环块代码的起始位置继续执行。

1. const POWER: u8 = 40;
2. const MODERATE\_TEMPERATURE: u8 = 150;
3. fn main() {
4. let mut temperature = 20;
5. loop {
6. temperature += POWER;
7. if temperature < MODERATE\_TEMPERATURE - POWER {
8. continue;
9. } else {
10. break;
11. }
12. }
13. assert!(temperature <= MODERATE\_TEMPERATURE);
14. }

代码片段ch3\_15.rs

不含break表达式的loop表达式具有never类型，表明loop表达式不可能计算得到一个值。如代码片段ch3\_16.rs中第6~8行的loop循环，循环体中既没有设置break表达式，也没有指定程序出口，该loop循环构成了一个死循环，程序在进入该循环后一直运行循环体内的代码，直到收到外部中断信号或其他情况才会停止，就像米斯琪在出门旅行时忘记关闭燃气灶一样，只有当燃气耗尽或者锅与环境达成热平衡等其他事件发生，锅的温度才会停止上升。

1. const POWER: u8 = 40;
2. fn main() {
3. let mut \_temperature = 20;
4. loop {
5. \_temperature += POWER;
6. }
7. }

代码片段ch3\_16.rs

### 3.3.2 while语句

while表达式由“while”关键字、循环条件、循环体构成，如下面的式子所示。while表达式表示的循环结构又被称作谓词循环（predicate loop）。

while表达式的循环条件是一个bool类型的表达式。程序在执行while循环时，将检验循环条件是否成立，如果循环条件成立，执行循环体对应的块代码，之后回到while表达式开始处，重新检验循环条件；如果循环条件不成立，退出循环，执行后续的代码。如果while表达式的循环条件始终成立，这样的while表达式则变成了一个无限循环，与loop表达式等价。

代码片段ch3\_17.rs用while表达式实现了与代码片段ch3\_13.rs中loop表达式相同语义的循环。第7行的“temperature < MODERATE\_TEMPERATURE”是while表达式的循环条件，第7~9行的块代码是while循环的循环体。每次执行该while表达式时，都会检验第7行的循环条件，若成立，执行7~9行的块代码，然后回到第7行重复这一过程；若不成立，跳转至第11行继续执行。

1. const POWER: u8 = 40;
2. const MODERATE\_TEMPERATURE: u8 = 150;
3. fn main() {
4. let mut temperature = 20;
5. while temperature < MODERATE\_TEMPERATURE {
6. temperature += POWER;
7. }
8. assert!(temperature >= MODERATE\_TEMPERATURE);
9. }

代码片段ch3\_17.rs

在while表达式中使用break表达式的规则和效果与loop表达式中使用break表达式的规则和效果相同。在 while表达式中，使用continue表达式将使控制流跳转到循环条件处重新判断循环条件，若成立，则执行循环体；不成立，则退出循环。

### 3.3.3 for语句

Rust的for表达式所代表的循环结构被称为迭代器循环（iterator loop），这是因为Rust提供的for表达式只适用于迭代器提供的元素上进行遍历的语法结构。 迭代器的相关内容将在第五章中详细介绍。第二章中介绍的数组类型可用作迭代器的元素遍历。for表达式由“for”关键字、循环变量（由模式表示）、“in”关键字、能够组成迭代器的表达式、循环体（由块表达式组成），如下面的式子所示。在每次执行for表达式的循环体之前，会从迭代器中取出下一个元素，作为循环变量供循环体使用。当迭代器中的元素都被取出时，循环结束。故for表达式的循环条件是迭代器中尚有可用的元素。

代码片段ch3\_18.rs模拟了米斯琪大厨在烹饪番茄炒蛋时加入食材的场景：代码片段的第2行，首先准备好番茄和鸡蛋（记得不要把蛋壳放入锅中），存放在数组“ingredients”中；第4~6行的for表达式以数组“ingredients”创建迭代器，在每次循环执行时，按照数组下标顺序取出数组元素，作为循环变量“ingredient”，用于第4~6行的循环体中。循环体每次将取出的元素与字符串“put some ”拼接，输出到程序标准输出流。

1. fn main() {
2. let ingredients = ["tomato", "egg"];
3. for ingredient in ingredients {
4. println!("put some {}", ingredient);
5. }
6. }

代码片段ch3\_18.rs

在for表达式中使用break表达式的规则和效果与loop表达式中使用break表达式的规则和效果相同。在 for表达式中，使用continue表达式将使控制流跳转到for循环开始处，若迭代器中仍有可用的元素，取出下一个元素作为循环变量，执行循环体；若迭代器无可用元素，退出循环。

### 3.3.4 三种循环的比较

本节内容介绍的loop表达式、while表达式、for表达式各有所长：

* loop表达式是一个无限循环，需要在循环体内使用break表达式中断循环。在一些需要使用无限循环，或者根据某些特定条件需要在循环体内部使用 break 终止循环的情况，使用loop表达式更为合适。
* while 表达式根据指定的循环条件重复执行循环体，循环条件在每次循环执行前进行检查。对于循环条件较为简洁、在循环外部就可以求出的情况，使用while表达式更为合适。得益于while表达式的语法结构，while表达式的循环条件在循环开始处给出，这使得相同语义的循环结构，while表达式比loop表达式的可读性更强。
* for 循环用于遍历迭代器中的元素，语法简洁明了。对于需要遍历一些特定容器中的元素的循环结构，使用for循环较为合适，可读性比loop表达式、while表达式更强。

### 3.3.5 迭代器

迭代器是Rust提供的一种强大工具，用于逐个处理数据集合中的元素。如代码片段ch3\_18.rs的第4行通过for循环创建了数组ingredients的一个迭代器，遍历ingredients中的元素。

Rust中几乎所有的集合类型都能够用迭代器遍历其中元素，例如数组[T; N，链表LinkedList<T> ，区间Range<T>，向量Vec<T>，散列表HashTable<K, V>，散列集合HashSet<T>类型，都能够用迭代器访问其中的元素。

## 3.4 小结

程序设计语言中的语句（Statements）和表达式（Expressions）是构建程序逻辑的关键元素。语句是执行操作但不返回值的代码单元；表达式负责计算并产生一个值，并且可以通过嵌套、组合，形成更复杂的逻辑结构。在本章节中，我们介绍了Rust程序设计语言中语句和表达式的相关概念，以及一部分基础的表达式和控制流结构。

理解Rust中表达式的语法和语义对正确的代码编写至关重要。Rust是一门以表达式为主的语言，几乎所有的代码都离不开表达式，大部分的代码都是可求值的。熟练掌握Rust中常用的语句和表达式形式，开发人员可以更好地利用Rust的特性，写出更为简洁、高效、可读性强的代码。