# 关键字

## 目前正在使用的关键字

如下关键字目前有对应其描述的功能。

* as - 强制类型转换，消除特定包含项的 trait 的歧义，或者对 use 和 extern crate 语句中的项重命名
* break - 立刻退出循环
* const - 定义常量或不变裸指针（constant raw pointer）
* continue - 继续进入下一次循环迭代
* crate - 链接（link）一个外部 crate 或一个代表宏定义的 crate 的宏变量
* dyn - 动态分发 trait 对象
* else - 作为 if 和 if let 控制流结构的 fallback
* enum - 定义一个枚举
* extern - 链接一个外部 crate 、函数或变量
* false - 布尔字面值 false
* fn - 定义一个函数或 函数指针类型 (function pointer type)
* for - 遍历一个迭代器或实现一个 trait 或者指定一个更高级的生命周期
* if - 基于条件表达式的结果分支
* impl - 实现自有或 trait 功能
* in - for 循环语法的一部分
* let - 绑定一个变量
* loop - 无条件循环
* match - 模式匹配
* mod - 定义一个模块
* move - 使闭包获取其所捕获项的所有权
* mut - 表示引用、裸指针或模式绑定的可变性性
* pub - 表示结构体字段、impl 块或模块的公有可见性
* ref - 通过引用绑定
* return - 从函数中返回
* Self - 实现 trait 的类型的类型别名
* self - 表示方法本身或当前模块
* static - 表示全局变量或在整个程序执行期间保持其生命周期
* struct - 定义一个结构体
* super - 表示当前模块的父模块
* trait - 定义一个 trait
* true - 布尔字面值 true
* type - 定义一个类型别名或关联类型
* unsafe - 表示不安全的代码、函数、trait 或实现
* use - 引入外部空间的符号
* where - 表示一个约束类型的从句
* while - 基于一个表达式的结果判断是否进行循环

## 保留做将来使用的关键字

如下关键字没有任何功能，不过由 Rust 保留以备将来的应用。

* abstract
* async
* become
* box
* do
* final
* macro
* override
* priv
* try
* typeof
* unsized
* virtual
* yield

## 原始标识符

原始标识符（Raw identifiers）允许你使用通常不能使用的关键字，其带有 r# 前缀。r# 前缀需同时用于函数名和调用

fn r#match(needle: &str, haystack: &str)

# 基本语法

## 引用

默认情况下，Rust 将 prelude 模块中少量的类型引入到每个程序的作用域中。如果需要的类型不在 prelude 中，你必须使用 use 语句显式地将其引入作用域。

use std::io;

## 入口

main 函数是程序的入口点，fn 语法声明了一个新函数，() 表明没有参数，{ 作为函数体的开始。

fn main() {

## 变量

### 赋值

使用 mut 来使一个变量可变

let foo = 5; // 不可变  
let mut bar = 5; // 可变

不能对不可变变量二次赋值

fn main(){  
 let mut x = 5;  
 println!("The value of x is: {}", x);  
 x = 6;  
 println!("The value of x is: {}", x);  
}

### 隐藏

可以定义一个与之前变量同名的新变量，而新变量会 隐藏 之前的变量。可以用相同变量名称来隐藏一个变量，以及重复使用 let 关键字来多次隐藏，

fn main() {  
 let x = 5;  
 let x = x + 1;  
 let x = x \* 2;  
 println!("The value of x is: {}", x);  
}

隐藏与将变量标记为 mut 是有区别的。当不小心尝试对变量重新赋值时，如果没有使用 let 关键字，就会导致编译时错误。通过使用 let，我们可以用这个值进行一些计算，不过计算完之后变量仍然是不变的。

mut 与隐藏的另一个区别是，当再次使用 let 时，实际上创建了一个新变量，我们可以改变值的类型，但复用这个名字。

## 常量

声明常量使用 const 关键字，并且必须注明值的类型。常量可以在任何作用域中声明，包括全局作用域

Rust 常量的命名规范是使用下划线分隔的大写字母单词，并且可以在数字字面值中插入下划线来提升可读性

const MAX\_POINTS: u32 = 100\_000;

## 数据类型

每一个值都属于某一个 数据类型（data type），这告诉 Rust 它被指定为何种数据，以便明确数据处理方式

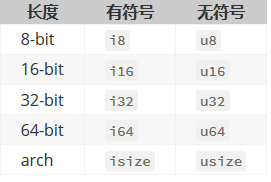
Rust是静态类型（statically typed）语言，在编译时就必须知道所有变量的类型。根据值及其使用方式，编译器通常可以推断出要用的类型。当多种类型均有可能时，必须增加类型注解

### 标量类型

标量（scalar）类型代表一个单独的值。Rust 有四种基本的标量类型：整型、浮点型、布尔类型和字符类型。

#### 整型

整数是一个没有小数部分的数字。



isize 和 usize 类型依赖运行程序的计算机架构：64 位架构上它们是 64 位的， 32 位架构上它们是 32 位的。

可以使用表格中的任何一种形式编写数字字面值。注意除 byte 以外的所有数字字面值允许使用类型后缀，例如 57u8，同时也允许使用 \_ 做为分隔符以方便读数，例如1\_000



Rust 的默认类型是 i32：它通常是最快的，甚至在 64 位系统上也是。isize 或 usize 主要作为某些集合的索引。

#### 浮点型

Rust 的浮点数类型是 f32 和 f64，分别占 32 位和 64 位。默认类型是 f64，因为在现代 CPU 中，它与 f32 速度几乎一样，不过精度更高。

fn main() {  
 let x = 2.0; // f64  
 let y: f32 = 3.0; // f32  
}

#### 数值运算

fn main() {  
 let sum = 5 + 10; // 加法  
 let difference = 95.5 - 4.3; // 减法  
 let product = 4 \* 30; // 乘法  
 let quotient = 56.7 / 32.2; // 除法  
 let remainder = 43 % 5; // 取余  
}

|  |  |  |  |
| --- | --- | --- | --- |
| 运算符 | 示例 | 解释 | 是否可重载 |
| ! | ident!(...), ident!{...}, ident![...] | 宏展开 |  |
| ! | !expr | 按位非或逻辑非 | Not |
| != | var != expr | 不等比较 | PartialEq |
| % | expr % expr | 算术取模 | Rem |
| %= | var %= expr | 算术取模与赋值 | RemAssign |
| & | &expr, &mut expr | 借用 |  |
| & | &type, &mut type, &'a type, &'a mut type | 借用指针类型 |  |
| & | expr & expr | 按位与 | BitAnd |
| &= | var &= expr | 按位与及赋值 | BitAndAssign |
| && | expr && expr | 逻辑与 |  |
| \* | expr \* expr | 算术乘法 | Mul |
| \*= | var \*= expr | 算术乘法与赋值 | MulAssign |
| \* | \*expr | 解引用 |  |
| \* | \*const type, \*mut type | 裸指针 |  |
| + | trait + trait, 'a + trait | 复合类型限制 |  |
| + | expr + expr | 算术加法 | Add |
| += | var += expr | 算术加法与赋值 | AddAssign |
| , | expr, expr | 参数以及元素分隔符 |  |
| - | - expr | 算术取负 | Neg |
| - | expr - expr | 算术减法 | Sub |
| -= | var -= expr | 算术减法与赋值 | SubAssign |
| -> | fn(...) -> type, |...| -> type | 函数与闭包，返回类型 |  |
| . | expr.ident | 成员访问 |  |
| .. | .., expr.., ..expr, expr..expr | 右排除范围 |  |
| .. | ..expr | 结构体更新语法 |  |
| .. | variant(x, ..), struct\_type { x, .. } | “与剩余部分”的模式绑定 |  |
| ... | expr...expr | 模式: 范围包含模式 |  |
| / | expr / expr | 算术除法 | Div |
| /= | var /= expr | 算术除法与赋值 | DivAssign |
| : | pat: type, ident: type | 约束 |  |
| : | ident: expr | 结构体字段初始化 |  |
| : | 'a: loop {...} | 循环标志 |  |
| ; | expr; | 语句和语句结束符 |  |
| ; | [...; len] | 固定大小数组语法的部分 |  |
| << | expr << expr | 左移 | Shl |
| <<= | var <<= expr | 左移与赋值 | ShlAssign |
| < | expr < expr | 小于比较 | PartialOrd |
| <= | expr <= expr | 小于等于比较 | PartialOrd |
| = | var = expr, ident = type | 赋值/等值 |  |
| == | expr == expr | 等于比较 | PartialEq |
| => | pat => expr | 匹配准备语法的部分 |  |
| > | expr > expr | 大于比较 | PartialOrd |
| >= | expr >= expr | 大于等于比较 | PartialOrd |
| >> | expr >> expr | 右移 | Shr |
| >>= | var >>= expr | 右移与赋值 | ShrAssign |
| @ | ident @ pat | 模式绑定 |  |
| ^ | expr ^ expr | 按位异或 | BitXor |
| ^= | var ^= expr | 按位异或与赋值 | BitXorAssign |
| | | pat | pat | 模式选择 |  |
| | | expr | expr | 按位或 | BitOr |
| |= | var |= expr | 按位或与赋值 | BitOrAssign |
| || | expr || expr | 逻辑或 |  |
| ? | expr? | 错误传播 |  |

#### 布尔型

布尔类型有两个可能的值：true 和 false。Rust 中的布尔类型使用 bool 表示

fn main() {  
 let t = true;  
 let f: bool = false; // 显式指定类型注解  
}

#### 字符类型

Rust 的 char 类型代表了一个 Unicode 标量值（Unicode Scalar Value），这意味着它可以比 ASCII 表示更多内容。

在 Rust 中，拼音字母（Accented letters），中文、日文、韩文等字符，emoji（绘文字）以及零长度的空白字符都是有效的 char 值。Unicode 标量值包含从 U+0000 到 U+D7FF 和 U+E000 到 U+10FFFF 在内的值。不过，“字符” 并不是一个 Unicode 中的概念，所以人直觉上的 “字符” 可能与 Rust 中的 char 并不符合。

### 复合类型

复合类型（Compound types）可以将多个值组合成一个类型。Rust 有两个原生的复合类型：元组（tuple）和数组（array）

#### 元组

使用包含在圆括号中的逗号分隔的值列表来创建一个元组。元组中的每一个位置都有一个类型，而且这些不同值的类型也不必是相同的。

fn main() {  
 let tup= (500, 6.4, 1);  
 // 解构  
 let (x, y, z) = tup;  
 println!("The value of y is: {}", y);  
 //直接访问  
 let one=tup.0;  
 println!("The value of one is: {}", one);  
}

#### 数组类型

Rust 中的数组是固定长度的：一旦声明，它们的长度不能增长或缩小。

数组中的每个元素的类型必须相同，值位于中括号内的逗号分隔的列表中：

数组是一整块分配在栈上的内存。可以使用索引来访问数组的元素

fn main() {  
 let a = [1, 2, 3, 4, 5];  
 let one=a[0];  
 println!("one is {}",one);  
}

## 函数

Rust 代码中的函数和变量名使用 snake case 规范风格。在 snake case 中，所有字母都是小写并使用下划线分隔单词。

Rust 不关心函数定义于何处，只要定义了就行。

在函数签名中，必须 声明每个参数的类型。

fn another\_function(*x*: i32, *y*: i32) {  
 println!("The value of x is: {}", *x*);  
 println!("The value of y is: {}", *y*);  
}

#### 函数体

函数体由一系列的语句和一个可选的结尾表达式构成。因为 Rust 是一门基于表达式（expression-based）的语言，这是一个需要理解的（不同于其他语言）重要区别。

语句（Statements）是执行一些操作但不返回值的指令。

* 使用 let 关键字创建变量并绑定一个值是一个语句

let y = 6;

* 函数定义也是语句

fn main() {

let y = 6;

}

语句不返回值。因此，不能把 let 语句赋值给另一个变量

* 表达式可以是语句的一部分

let y = 6; 中的 6 是一个表达式，它计算出的值是 6

* 函数调用是一个表达式。宏调用是一个表达式，用来创建新作用域的大括号（代码块），{}，也是一个表达式

{

let x = 3;

x + 1

}

是一个代码块，它的值是 4，表达式的结尾没有分号。如果在表达式的结尾加上分号，它就变成了语句，而语句不会返回值。

#### 带返回值

函数可以向调用它的代码返回值。我们并不对返回值命名，但要在箭头（->）后声明它的类型。使用 return 关键字和指定值，可从函数中提前返回；但大部分函数隐式的返回最后的表达式。

fn five() -> i32 {  
 5  
}  
fn main() {  
 let x = five();  
 println!("The value of x is: {}", x);  
}

，Rust 并不会尝试自动地将非布尔值转换为布尔值。必须总是显式地使用布尔值作为 if 的条件

## 循环

### loop

fn main() {  
 let mut counter = 0;  
 let result = loop {  
 counter += 1;  
 println!("num:{}",counter);  
 if counter == 10 {  
 break counter \* 2;  
 }  
 };  
 assert\_eq!(result, 20);  
}

### while

fn main() {  
 let mut number = 3;  
 while number != 0 {  
 println!("{}!", number);  
 number = number - 1;  
 }  
 println!("LIFTOFF!!!");  
}

### for

fn main() {  
 let a = [10, 20, 30, 40, 50];  
 for element in a.iter() {  
 println!("the value is: {}", element);  
 }  
}