# rust一些通用编程的概念

官网文档[数据类型 - Rust 程序设计语言 中文版 (rustwiki.org)](https://rustwiki.org/zh-CN/book/ch03-02-data-types.html)

## 变量，数据类型，条件判断，循环

### 变量

#### rust中变量的可变性是值得注意的

* 例如:
* fn main(){
    
   let number = 1;
    
   number = 2;
    
   println!("the number is {}",number);
    
  }

##### let关键字定义的变量默认无法改变，上述的方式会导致运行报错

* 使用cargo run运行时得到以下结果
* PS D:\rust\_project\demo2> cargo run
    
   Compiling demo2 v0.1.0 (D:\rust\_project\demo2)
    
  warning: value assigned to `number` is never read
    
   --> src/main.rs:2:9
    
   |
    
  2 | let number = 1;
    
   | ^^^^^^
    
   |
    
   = help: maybe it is overwritten before being read?
    
   = note: `#[warn(unused\_assignments)]` on by default
    
    
  error[E0384]: cannot assign twice to immutable variable `number`
    
   --> src/main.rs:3:5
    
   |
    
  2 | let number = 1;
    
   | ------ first assignment to `number`
    
  3 | number = 2;
    
   | ^^^^^^^^^^ cannot assign twice to immutable variable
    
   |
    
  help: consider making this binding mutable
    
   |
    
  2 | let mut number = 1;
    
   | +++
    
    
  For more information about this error, try `rustc --explain E0384`.
    
  warning: `demo2` (bin "demo2") generated 1 warning
    
  error: could not compile `demo2` (bin "demo2") due to 1 previous error; 1 warning emitted
* 因为这里let直接定义的属于不可变变量，如果你需要定义一个可变变量需要使用**mut**关键字
* fn main(){
    
   let mut number = 1;
    
   number = 2;
    
   println!("the number is {}",number);
    
  }
* 结果就可以正常运行，不过会有相对应的warnning
* warning: value assigned to `number` is never read
    
   --> src/main.rs:2:13
    
   |
    
  2 | let mut number = 1;
    
   | ^^^^^^
    
   |
    
   = help: maybe it is overwritten before being read?
    
   = note: `#[warn(unused\_assignments)]` on by default
    
    
  warning: `demo2` (bin "demo2") generated 1 warning
    
   Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.47s
    
   Running `target\debug\demo2.exe`
    
  the number is 2
* 再引入一个概念
* **常量**
* 常量和不可变变量类似，用于绑定到一个常量名且不允许更改的值，但是常量和变量之间存在一定差异，常量不允许使用**mut**关键字
* 且自始至终无法改变，使用**const**关键字定义常量，同时必须标注数据类型，常量可以在任意作用域中声明，包括全局作用域，且无法为函数调用结果或只能在运算时得到的值
* fn main(){
    
   const THREE\_HOURS\_IN\_SECONDS: u32 = 60 \* 60 \* 3;
    
   println!("the time is {}",THREE\_HOURS\_IN\_SECONDS);
    
   }
* D:\rust\_project\demo2>cargo run
    
   Compiling demo2 v0.1.0 (D:\rust\_project\demo2)
    
   Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.35s
    
   Running `target\debug\demo2.exe`
    
  the time is 10800
* **遮掩**
* 可以通过声明相同的变量名称来遮盖前面一个变量
* fn main() {
    
   let x = 5;
    
    
   let x = x + 1;
    
    
   {
    
   let x = x \* 2;
    
   println!("The value of x in the inner scope is: {}", x);
    
   }
    
    
   println!("The value of x is: {}", x);
    
  }
* The value of x in the inner scope is: 12
    
  The value of x is: 6
* **遮掩**与**mut**区别在于
  1. 遮掩需要重新使用let定义新变量来遮盖达到修改原先变量的值，而mut不需要
  2. 遮掩是创建一个新变量所以可以改变原先变量的数据类型，但是mut始终一个变量类型
* fn main() {
    
   let spaces = " ";
    
   let spaces = spaces.len();
    
  }
* 这其中第一个spaces为字符串类型，第二个为数字类型
* fn main() {
    
   let mut spaces = " ";
    
   spaces = spaces.len();
    
  }
* 而这样编译器则会报错
* Compiling demo2 v0.1.0 (D:\rust\_project\demo2)
    
  error: format argument must be a string literal
    
   --> src/main.rs:4:14
    
   |
    
  4 | println!(spaces)
    
   | ^^^^^^
    
   |
    
  help: you might be missing a string literal to format with
    
   |
    
  4 | println!("{}", spaces)
    
   | +++++
    
    
  error: could not compile `demo2` (bin "demo2") due to 1 previous error

#### 数据类型

* rust是一种静态类型语言，因此它在编译期必须知道所有变量的类型，rust每一个值都有确切的数据类型
* 一般可以在变量名称后面使用英文冒号加数据类型的方式进行标注
* eg.
* let guess: u32 = "42".parse().expect("Not a number!");
* : u32表示guess是一个无符号的32位整型数据
* 同时这段代码必须加上对应的:u32数据标注，因为等式的右边采用parse()方法将String类型转为数值类型时，必须显性的声明变量的数据类型
* 否则报错
* error[E0282]: type annotations needed
    
   --> src/main.rs:2:9
    
   |
    
  2 | let guess = "42".parse().expect("Not a number!");
    
   | ^^^^^ consider giving `guess` a type
    
    
  For more information about this error, try `rustc --explain E0282`.
    
  error: could not compile `no\_type\_annotations` due to previous error

#### 标量类型

* 标量类型表示单个值。Rust有4个基本的标量类型：整型、浮点型、布尔型和字符
  + 整数类型

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

* + 有无符号表示是否取负数，每个有符号类型规定的数字范围是 -(2n - 1) ~ 2n - 1 - 1，其中 n 是该定义形式的位长度。所以 i8 可存储数字范围是 -(27) ~ 27 - 1，即 -128 ~ 127。无符号类型可以存储的数字范围是 0 ~ 2n - 1，所以 u8 能够存储的数字为 0 ~ 28 - 1，即 0 ~ 255。
  + 此外，isize 和 usize 类型取决于程序运行的计算机体系结构，在表中表示为“arch”：若使用 64 位架构系统则为 64 位，若使用 32 位架构系统则为 32 位。
  + 可能属于多种数字类型的数字字面量允许使用类型后缀来指定类型，例如 57u8。数字字面量还可以使用 \_ 作为可视分隔符以方便读数，如 1\_000，此值和 1000 相同。

| **数字字面量** | **示例** |
| --- | --- |
| 十进制 | 98\_222 |
| 十六进制 | 0xff |
| 八进制 | 0o77 |
| 二进制 | 0b1111\_0000 |
| 字节 (仅限于 u8) | b'A' |

* + 同时如果不确定整数类型，rust通常会默认i32
  + 浮点数
  + 带有小数的数字
  + 浮点数按照 IEEE-754 标准表示。f32 类型是单精度浮点型，f64 为双精度浮点型。
  + Rust 的所有数字类型都支持基本数学运算：加法、减法、乘法、除法和取模运算。整数除法会向下取整。下面代码演示了各使用一条 let 语句来说明相应数字运算的用法：
  + fn main() {
      
     // addition
      
     let sum = 5 + 10;
      
      
     // subtraction
      
     let difference = 95.5 - 4.3;
      
      
     // multiplication
      
     let product = 4 \* 30;
      
      
     // division
      
     let quotient = 56.7 / 32.2;
      
     let floored = 2 / 3; // Results in 0
      
      
     // remainder
      
     let remainder = 43 % 5;
      
    }
  + 布尔类型
  + 表示是否，和大多数编程语言一样
  + fn main() {
      
     let t = true;
      
      
     let f: bool = false; // with explicit type annotation
      
    }
  + 字符类型
  + Rust 的 char（字符）类型是该语言最基本的字母类型，且为4个字节支持Unicode编码下面是一些声明 char 值的例子：
  + fn main() {
      
     let c = 'z';
      
     let z = 'ℤ';
      
     let heart\_eyed\_cat = '😻';
      
    }
  + 复合类型
    - 元组
    - 和python中的元组概念类似
    - 可以将多种类型的值组合到一个复合类型中
    - fn main() {
        
       let tup: (i32, f64, u8) = (500, 6.4, 1);
        
      }
    - 同时可以使用模式解构的方式获取元组中的某个值
    - fn main() {
        
       let tup: (i32,u32,f64,bool) = (-1,1121,3.1415926,true);
        
       let (x,y,z,w) = tup;
        
       println!("The value of x is: {}",x);
        
      }
    - D:\rust\_project\demo2>cargo run
        
       Compiling demo2 v0.1.0 (D:\rust\_project\demo2)
        
      warning: unused variable: `y`
        
       --> src/main.rs:3:12
        
       |
        
      3 | let (x,y,z,w) = tup;
        
       | ^ help: if this is intentional, prefix it with an underscore: `\_y`
        
       |
        
       = note: `#[warn(unused\_variables)]` on by default
        
        
      warning: unused variable: `z`
        
       --> src/main.rs:3:14
        
       |
        
      3 | let (x,y,z,w) = tup;
        
       | ^ help: if this is intentional, prefix it with an underscore: `\_z`
        
        
      warning: unused variable: `w`
        
       --> src/main.rs:3:16
        
       |
        
      3 | let (x,y,z,w) = tup;
        
       | ^ help: if this is intentional, prefix it with an underscore: `\_w`
        
        
      warning: `demo2` (bin "demo2") generated 3 warnings
        
       Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.35s
        
       Running `target\debug\demo2.exe`
        
      The value of x is: -1
    - 也可以利用.运算符来获取元组中的某个值,同时索引从0开始
    - warning: unused variable: `x`
        
       --> src/main.rs:3:10
        
       |
        
      3 | let (x,y,z,w) = tup;
        
       | ^ help: if this is intentional, prefix it with an underscore: `\_x`
        
       |
        
       = note: `#[warn(unused\_variables)]` on by default
        
        
      warning: unused variable: `y`
        
       --> src/main.rs:3:12
        
       |
        
      3 | let (x,y,z,w) = tup;
        
       | ^ help: if this is intentional, prefix it with an underscore: `\_y`
        
        
      warning: unused variable: `z`
        
       --> src/main.rs:3:14
        
       |
        
      3 | let (x,y,z,w) = tup;
        
       | ^ help: if this is intentional, prefix it with an underscore: `\_z`
        
        
      warning: unused variable: `w`
        
       --> src/main.rs:3:16
        
       |
        
      3 | let (x,y,z,w) = tup;
        
       | ^ help: if this is intentional, prefix it with an underscore: `\_w`
        
        
      warning: `demo2` (bin "demo2") generated 4 warnings
        
       Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.35s
        
       Running `target\debug\demo2.exe`
        
      The value of x is: -1
    - 数组类型
    - 写法上等同于python中的列表，但是概念等同于c语言
    - fn main() {
        
       let array = [1,2,3];
        
       println!("{}", array[0])
        
      }
    - D:\rust\_project\demo2>cargo run
        
       Compiling demo2 v0.1.0 (D:\rust\_project\demo2)
        
       Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.35s
        
       Running `target\debug\demo2.exe`
        
      1
    - 但是数组长度**不可变**，数组中的**数据类型必须一致**
    - 显示写法
    - fn main() {
        
      let a: [i32; 5] = [1, 2, 3, 4, 5];
        
      }
    - 这里，i32 是每个元素的类型。分号之后，数字 5 表明该数组包含 5 个元素。
    - 如果要为每个元素创建包含相同值的数组，可以指定初始值，后跟分号，然后在方括号中指定数组的长度，如下所示：
    - fn main() {
        
      let a = [3; 5];
        
      }

#### 函数

* **其实很类似与python的显示函数声明**
* 通过关键字**fn**自定义函数
* main函数便是rust程序的启动入口
* fn main() {
    
   println!("Hello, world!");
    
    
   another\_function();
    
  }
    
    
  fn another\_function() {
    
   println!("Another function.");
    
  }
* $ cargo run
    
   Compiling functions v0.1.0 (file:///projects/functions)
    
   Finished dev [unoptimized + debuginfo] target(s) in 0.28s
    
   Running `target/debug/functions`
    
  Hello, world!
    
  Another function.

### [参数](https://rustwiki.org/zh-CN/book/ch03-03-how-functions-work.html#参数)

* 函数也可以被定义为拥有**参数**（*parameter*），参数是特殊变量，是函数签名的一部分。当函数拥有参数（形参）时，可以为这些参数提供具体的值（实参）。技术上讲，这些具体值被称为**实参**（*argument*），但是在日常交流中，人们倾向于不区分使用 *parameter* 和 *argument* 来表示函数定义中的变量或调用函数时传入的具体值。
* 在函数签名中，**必须**声明每个参数的类型。这是一个 Rust 设计中经过慎重考虑的决定：要求在函数定义中提供类型标注，意味着编译器几乎从不需要你在代码的其他地方注明类型来指出你的意图。
* 当一个函数有多个参数时，使用逗号分隔，像这样：
* fn main() {
    
   print\_labeled\_measurement(5, 'h');
    
  }
    
    
  fn print\_labeled\_measurement(value: i32, unit\_label: char) {
    
   println!("The measurement is: {}{}", value, unit\_label);
    
  }
* 这个例子创建了一个有两个参数的名为 print\_labeled\_measurement 的函数。第一个参数名为 value， 类型是 i32。第二个参数是 unit\_label ，类型是 char。接着该函数打印包含 value 和 unit\_label 的文本。

### [语句和表达式](https://rustwiki.org/zh-CN/book/ch03-03-how-functions-work.html#语句和表达式)

* 函数体由一系列语句组成，也可选择以表达式结尾。目前为止，我们介绍的函数还没有包含结尾表达式，不过你已经看到了表达式作为语句的一部分。因为 Rust 是一门基于表达式（expression-based）的语言，所以这是一个需要理解的重要区别。其他语言没有这样的区别，所以让我们看看语句和表达式分别是什么，以及它们的区别如何影响函数体。
* **语句**（*statement*）是执行一些操作但不返回值的指令。表达式（*expression*）计算并产生一个值。让我们看一些例子：
* 实际上，我们已经使用过语句和表达式。使用 let 关键字创建变量并绑定一个值是一个语句。在示例中，let y = 6; 是一个语句。
* fn main() {
    
   let y = 6;
    
  }
* 函数定义也是语句，上面整个例子本身就是一个语句。
* 语句不返回值。因此，不能把 let 语句赋值给另一个变量，就像下面的代码尝试做的那样，会产生一个错误：
* fn main() {
    
   let x = (let y = 6);
    
  }
* $ cargo run
    
   Compiling functions v0.1.0 (file:///projects/functions)
    
  error: expected expression, found statement (`let`)
    
   --> src/main.rs:2:14
    
   |
    
  2 | let x = (let y = 6);
    
   | ^^^^^^^^^
    
   |
    
   = note: variable declaration using `let` is a statement
    
    
  error[E0658]: `let` expressions in this position are experimental
    
   --> src/main.rs:2:14
    
   |
    
  2 | let x = (let y = 6);
    
   | ^^^^^^^^^
    
   |
    
   = note: see issue #53667 <https://github.com/rust-lang/rust/issues/53667> for more information
    
   = help: you can write `matches!(<expr>, <pattern>)` instead of `let <pattern> = <expr>`
    
    
  warning: unnecessary parentheses around assigned value
    
   --> src/main.rs:2:13
    
   |
    
  2 | let x = (let y = 6);
    
   | ^ ^
    
   |
    
   = note: `#[warn(unused\_parens)]` on by default
    
  help: remove these parentheses
    
   |
    
  2 - let x = (let y = 6);
    
  2 + let x = let y = 6;
    
   |
    
    
  For more information about this error, try `rustc --explain E0658`.
    
  warning: `functions` (bin "functions") generated 1 warning
    
  error: could not compile `functions` due to 2 previous errors; 1 warning emitted
* 表达式会计算出一个值,考虑一个数学运算，比如 5 + 6，这是一个表达式并计算出值 11。表达式可以是语句的一部分
* 语句 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。我们用来创建新作用域的大括号（代码块） {} 也是一个表达式，例如
* fn main() {
    
   let y = {
    
   let x = 3;
    
   x + 1
    
   };
    
    
   println!("The value of y is: {}", y);
    
  }
* 中这个就是表达式
* {
    
   let x = 3;
    
   x + 1
    
  }
* 是一个代码块，在这个例子中计算结果是 4。这个值作为 let 语句的一部分被绑定到 y 上。注意，x + 1 行的末尾没有分号，这与你目前见过的大部分代码行不同。表达式的结尾没有分号。如果在表达式的末尾加上分号，那么它就转换为语句，而语句不会返回值。在接下来探讨函数返回值和表达式时

#### 带有返回值的函数

* 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。在 Rust 中，函数的返回值等同于函数体最后一个表达式的值。使用 return 关键字和指定值，可以从函数中提前返回；但大部分函数隐式返回最后一个表达式。这是一个有返回值函数的例子：
* fn five() -> i32 {
    
   5
    
  }
    
    
  fn main() {
    
   let x = five();
    
    
   println!("The value of x is: {}", x);
    
  }
* 在 five 函数中没有函数调用、宏，甚至没有 let 语句——只有数字 5 本身。这在 Rust 中是一个完全有效的函数。注意，函数返回值的类型也被指定好，即 -> i32。尝试运行代码；输出应如下所示：
* $ cargo run
    
   Compiling functions v0.1.0 (file:///projects/functions)
    
   Finished dev [unoptimized + debuginfo] target(s) in 0.30s
    
   Running `target/debug/functions`
    
  The value of x is: 5
* five 函数的返回值是 5，所以返回值类型是 i32。让我们仔细检查一下这段代码。有两个重要的部分：首先，let x = five(); 这一行表明我们使用函数的返回值初始化一个变量。因为 five 函数返回 5，这一行与如下代码相同：
* #![allow(unused)]
    
  fn main() {
    
  let x = 5;
    
  }
* 其次，five 函数没有参数并定义了返回值类型，不过函数体只有单单一个 5 也没有分号，因为这是一个表达式，正是我们想要返回的值。
* 让我们看看另一个例子：
* fn main() {
    
   let x = plus\_one(5);
    
    
   println!("The value of x is: {}", x);
    
  }
    
    
  fn plus\_one(x: i32) -> i32 {
    
   x + 1
    
  }
* 运行代码会打印出 The value of x is: 6。**但如果在包含 x + 1 的行尾加上一个分号，把它从表达式变成语句**，我们将得到一个错误。
* fn main() {
    
   let x = plus\_one(5);
    
    
   println!("The value of x is: {}", x);
    
  }
    
    
  fn plus\_one(x: i32) -> i32 {
    
   x + 1;
    
  }
* $ cargo run
    
   Compiling functions v0.1.0 (file:///projects/functions)
    
  error[E0308]: mismatched types
    
   --> src/main.rs:7:24
    
   |
    
  7 | fn plus\_one(x: i32) -> i32 {
    
   | -------- ^^^ expected `i32`, found `()`
    
   | |
    
   | implicitly returns `()` as its body has no tail or `return` expression
    
  8 | x + 1;
    
   | - help: consider removing this semicolon
    
    
  For more information about this error, try `rustc --explain E0308`.
    
  error: could not compile `functions` due to previous error
* 主要的错误信息 “mismatched types”（类型不匹配）揭示了这段代码的核心问题。函数 plus\_one 的定义说明它要返回一个 i32 类型的值，不过语句并不会返回值，此值由单元类型 () 表示，表示不返回值。因为不返回值与函数定义相矛盾，从而出现一个错误。在输出中，Rust 提供了一条信息，可能有助于纠正这个错误：它建议删除分号，这将修复错误。
* 不过使用**return**也可以正常返回
* fn main() {
    
   let res = add(1,2);
    
   println!("{}", res);
    
    
  }
    
  fn add(value\_first: i32, value\_second: i32)->i32{
    
   return value\_first + value\_second;
    
  }
* D:\rust\_project\demo2>cargo run
    
   Compiling demo2 v0.1.0 (D:\rust\_project\demo2)
    
   Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.34s
    
   Running `target\debug\demo2.exe`
    
  3

#### 注释

* 和c语言一样，//和/\*\*/
* 还有一种文档注释后期说明

#### 控制流

* 根据条件是否为真来决定是否执行某些代码，或根据条件是否为真来重复运行一段代码，是大部分编程语言的基本组成部分。Rust 代码中最常见的用来控制执行流的结构是 if 表达式和循环。

### [if 表达式](https://rustwiki.org/zh-CN/book/ch03-05-control-flow.html#if-表达式)

* if 表达式允许根据条件执行不同的代码分支。你提供一个条件并表示 “如果条件满足，运行这段代码；如果条件不满足，不运行这段代码。
* fn main() {
    
   let number = 3;
    
    
   if number < 5 {
    
   println!("condition was true");
    
   } else {
    
   println!("condition was false");
    
   }
    
  }
* $ cargo run
    
   Compiling branches v0.1.0 (file:///projects/branches)
    
   Finished dev [unoptimized + debuginfo] target(s) in 0.31s
    
   Running `target/debug/branches`
    
  condition was true
* **值得注意的是代码中的条件必须是 bool 值。如果条件不是 bool 值，我们将得到一个错误。例如**
* fn main() {
    
   let number = 3;
    
    
   if number {
    
   println!("number was three");
    
   }
    
  }
* $ cargo run
    
   Compiling branches v0.1.0 (file:///projects/branches)
    
  error[E0308]: mismatched types
    
   --> src/main.rs:4:8
    
   |
    
  4 | if number {
    
   | ^^^^^^ expected `bool`, found integer
    
    
  For more information about this error, try `rustc --explain E0308`.
    
  error: could not compile `branches` due to previous error
* **必须自始至终显式地使用布尔值作为 if 的条件**

#### [使用 else if 处理多重条件](https://rustwiki.org/zh-CN/book/ch03-05-control-flow.html#使用-else-if-处理多重条件)

* 可以将 if 和 else 组成的 else if 表达式来实现多重条件。例如：
* fn main() {
    
   let number = 6;
    
    
   if number % 4 == 0 {
    
   println!("number is divisible by 4");
    
   } else if number % 3 == 0 {
    
   println!("number is divisible by 3");
    
   } else if number % 2 == 0 {
    
   println!("number is divisible by 2");
    
   } else {
    
   println!("number is not divisible by 4, 3, or 2");
    
   }
    
  }

#### [在 let 语句中使用 if](https://rustwiki.org/zh-CN/book/ch03-05-control-flow.html#在-let-语句中使用-if)

* 因为 if 是一个表达式，我们可以在 let 语句的右侧使用它来将结果赋值给一个变量，例如：
* fn main() {
    
   let condition = true;
    
   let number = if condition { 5 } else { 6 };
    
    
   println!("The value of number is: {}", number);
    
  }
* 其实就是三元运算符
* **但是记住，代码块的值是其最后一个表达式的值，而数字本身就是一个表达式。在这个例子中，整个 if 表达式的值取决于哪个代码块被执行。这意味着 if 的每个分支的可能的返回值都必须是相同类型；在示例中，if 分支和 else 分支的结果都是 i32 整型。如果它们的类型不匹配，如下面这个例子，则会产生一个错误：**
* fn main() {
    
   let condition = true;
    
    
   let number = if condition { 5 } else { "six" };
    
    
   println!("The value of number is: {}", number);
    
  }
* $ cargo run
    
   Compiling branches v0.1.0 (file:///projects/branches)
    
  error[E0308]: `if` and `else` have incompatible types
    
   --> src/main.rs:4:44
    
   |
    
  4 | let number = if condition { 5 } else { "six" };
    
   | - ^^^^^ expected integer, found `&str`
    
   | |
    
   | expected because of this
    
    
  For more information about this error, try `rustc --explain E0308`.
    
  error: could not compile `branches` due to previous error

#### 循环控制流

* 在 Rust 中，循环控制流主要包括三种类型的循环：loop、while 和 for。它们的区别在于使用场景、语法以及控制流的特点。以下是对这三种循环的详细论述：

### 1. loop 循环

* loop 是 Rust 中的无限循环语句。它会无限执行其内部的代码块，直到明确使用 break 退出循环。适用于需要无限循环的场景，比如服务的事件循环。
* **示例：**
* let mut count = 0;  
  loop {  
   count += 1;  
   if count == 10 {  
   break;  
   }  
   println!("Count: {}", count);  
  }
  + **特点：**
    - 无条件循环，需要手动使用 break 来退出。
    - 可以通过 continue 跳过当前循环并进入下一次循环。
    - 适用于无法确定循环次数或者需要手动控制的循环场景。
  + **返回值：**
    - loop 循环可以返回值，通常与 break 结合使用。
  + let result = loop {  
     count += 1;  
     if count == 10 {  
     break count \* 2; // 返回值为 20  
     }  
    };

### 2. while 循环

* while 是基于条件判断的循环，在条件为 true 时重复执行代码块，条件为 false 时退出循环。
* **示例：**
* let mut number = 3;  
  while number != 0 {  
   println!("{}!", number);  
   number -= 1;  
  }  
  println!("Liftoff!");
  + **特点：**
    - 依赖布尔条件的判断，当条件为 false 时自动结束。
    - 适用于条件驱动的循环场景。
  + **优点：**
    - 比 loop 更灵活和安全，因为它不需要显式地使用 break。

### 3. for 循环

* for 循环用于遍历集合（如数组、迭代器、范围等）。它是 Rust 中最常用的循环形式。
* **示例：**
* let a = [10, 20, 30, 40, 50];  
  for element in a.iter() {  
   println!("The value is: {}", element);  
  }
  + **特点：**
    - 适合遍历固定范围或者集合类型（如数组、切片等）。
    - 语法简洁，自动处理索引范围，不容易出现越界错误。
  + **使用范围语法：**
  + for number in 1..4 {  
     println!("{}!", number); // 输出 1, 2, 3  
    }
  + **迭代器结合：**  
    for 循环与迭代器结合非常方便，可以利用 Rust 提供的强大迭代器机制来简化循环操作。

### 小结：

* + **loop**：无限循环，适用于需要手动控制退出的场景。
  + **while**：条件驱动的循环，适合在满足某个条件时执行的操作。
  + **for**：集合或范围遍历，最常用的循环形式，安全且简洁。