

1

Rust 语言介绍

Rust 编写的软件

- Rust 语言工具链
- Redox 操作系统
- Linux 6.1 内核加入了用 Rust 语言编写部分代码的支持
- Servo 浏览器引擎
- Typst 排版系统
- lsd、bat、fd 等命令行工具
- rCore 教学操作系统
- MadFS 文件系统
-

2

第一次接触

Hello, Rust!

用 Rust 编写 “Hello, world!” 程序:

```
fn main() {  
    println!("Hello, world!");  
}
```

【在线运行此代码】

编译:

```
$ rustc hello.rs
```

运行：

```
$ ./hello
```

Hello, world!

知识点:

- 函数定义
- **main** 函数
- 输出（宏调用）
- 字符串
- 编译与运行

猜数游戏

通过一个简单的项目演示 Rust 的一些常见功能。项目要求：

- 程序随机产生一个 1-100 之间的秘密整数 n ，让用户来猜。
- 提示用户，输入一个想猜的数 x 。
- 如果 $x = n$ ，则猜数成功，打印祝贺信息并退出程序。
- 否则，根据 x 和 n 的大小关系输出提示信息，继续让用户来猜。

创建新项目

使用 Cargo 创建一个新的项目：

```
$ cargo new guessing_game
```

```
$ cd guessing_game
```

Cargo.toml¹ 的内容:

```
[package]
name = "guessing_game"
version = "0.1.0"
edition = "2024"
```

```
[dependencies]
```

¹TOML (Tom's Obvious, Minimal Language) 是一种配置文件格式，常用于保存程序的配置信息。

运行项目

src/main.rs 的内容与前面那个 “Hello, world!” 程序是一样的。编译并运行：

```
$ cargo run
  Compiling guessing_game v0.1.0 (.../guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 1.50s
    Running `target/debug/guessing_game`
Hello, world!
```

- `cargo run` 命令可以自动进行编译并运行项目，便于快速迭代程序。
- 如果只要进行编译，可以用 `cargo build` 命令。

处理猜测

```
use std::io;

fn main() {
    println!("Guess the number!");
    println!("Please input your guess.");
    let mut guess = String::new();
    io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");
    println!("You guessed: {guess}");
}
```

知识点:

- 可变字符串类型
- 读取标准输入
- 错误处理
- 格式化输出

生成秘密数

在 Cargo.toml 中加入 rand 包:

```
# ...  
[dependencies]  
rand = "0.8.3"
```

编译:

```
$ cargo build
  Updating crates.io index
Downloaded rand v0.8.3
...
Compiling rand v0.8.3
Compiling guessing_game v0.1.0 (.../guessing_game)
  Finished dev [unoptimized + debuginfo] target(s) in 2.53s
```

比较猜测情况

```
use rand::Rng;
use std::cmp::Ordering;

let secret_number = rand::thread_rng().gen_range(1..=100);

let guess: u32 = guess.trim().parse().expect("Please type a number!");
println!("You guessed: {guess}");
match guess.cmp(&secret_number) {
    Ordering::Less => println!("Too small!"),
    Ordering::Greater => println!("Too big!"),
    Ordering::Equal => println!("You win!"),
}
```


结束猜数过程

```
loop {  
  // ...  
  match guess.cmp(&secret_number) {  
    Ordering::Less => println!("Too small!"),  
    Ordering::Greater => println!("Too big!"),  
    Ordering::Equal => {  
      println!("You win!");  
      break;  
    }  
  }  
}
```

3

基本语法

变量绑定 (续)

- 绑定可以被掩盖 (shadowing)，也叫隐藏或重影：

```
let x = 17;  
let y = 53;  
// x is not mutable, but we're able to re-bind it  
let x = "Shadowed!";
```

- 掩盖后的绑定 `x` 会一直存活到当前的作用域 (scope) 结束。
- 此时，第一个绑定已经丢失。
- 可以让同一含义不同类型的东西使用同一个变量名：

```
let mut cost = String::new();
std::io::stdin().read_line(&mut cost).unwrap();
let cost: f64 = cost.trim().parse().unwrap();
```

常量

```
const PI: f64 = 3.14159;
```

```
const MAGIC: i32 = 42;
```

- 常量 (constants) 与不可变变量的区别
 - 编译时常数，在编译时可以确定。
 - 可以出现在任何作用域（包括全局作用域）。
 - 可以提前使用。

表达式 (续)

- 由于基本上所有东西都是表达式，因此都可以绑定到变量：

```
let mut x = -5;
let y = if x > 0 {
    "greater"
} else {
    "less"
};
let z = loop {
    x += 10;
    if x > 5 {
        break x;
    }
};
```

基本类型

- 布尔 `bool`: 两个值 `true/false`。
- 字符 `char`: 用单引号, 例如 `'R'`、`'计'`, 是 `Unicode` 的。
- 数值: 分为整数和浮点数, 有不同的大小和符号属性。
 - `i8`、`i16`、`i32`、`i64`、`i128`、`isize`
 - `u8`、`u16`、`u32`、`u64`、`u128`、`usize`
 - `f32`、`f64`
 - 其中 `isize` 和 `usize` 是指针大小的整数, 因此它们的大小与机器架构相关。
 - 字面值 (literals) 写为 `10i8`、`10u16`、`10.0f32`、`10usize` 等。
 - 字面值如果不指定类型, 则默认整数为 `i32`, 浮点数为 `f64`。
- 数组 (arrays)、切片 (slices)、`str` 字符串 (strings)、元组 (tuples)
- 函数

数组

- 数组类型的形式为 `[T; N]`，例如 `[i32; 10]`。
 - `N` 是编译时常数 (compile-time constant)，也就是说数组的长度是固定的。
 - 运行时 (runtime) 访问数组元素会检查是否越界。
- 用 `[]` 来访问数组元素，数组下标从 0 开始。

```
let arr1 = [1, 2, 3]; // (array of 3 elements)
let arr2 = [2; 32];   // (array of 32 `2`s)
```

切片

- 切片类型的形式为 `&[T]`，例如 `&[i32]`。
- 切片表示引用数组中的一部分所形成的视图。
- 切片不能直接创建，需要从别的变量借用 (**borrow**)。
- 切片可以是可变的，也可以是不可变的。
- 如何知道一个切片是否有效？这是下一讲的内容。

```
let arr = [0, 1, 2, 3, 4, 5];
let total_slice = &arr;           // Slice all of `arr`
let total_slice = &arr[..];       // Same, but more explicit
let partial_slice = &arr[2..5];   // [2, 3, 4]
```

字符串

- Rust 有两种字符串: `String` 和 `&str`。
- `String` 是在堆上分配空间、可以增长的字符序列。
- `&str` 是 `String` 的切片类型²。
- 形如 `"foo"` 的字符串字面值都是 `&str` 类型的。

```
let s: &str = "galaxy";  
let s2: String = "galaxy".to_string();  
let s3: String = String::from("galaxy");  
let s4: &str = &s3;
```

²`str` 是没有大小的类型, 编译时不知道大小, 因此无法独立存在。

元组

- 元组是固定大小的、有序的、异构的列表类型。
- 可以通过下标来访问元组的分量，例如 `foo.0`。
- 可以使用 `let` 绑定来解构。

```
let foo: (i32, char, f64) = (72, 'H', 5.1);  
let (x, y, z) = (72, 'H', 5.1);  
let (a, b, c) = foo; // a = 72, b = 'H', c = 5.1
```

向量

Vec<T>

- 标准库提供的类型，可直接使用。
- Vec 是分配在堆上的、可增长的数组。
 - 类似 C++ 中的 `std::vector`、Java 中的 `java.util.ArrayList`。
- <T> 表示泛型，使用时代入实际的类型。
 - 例如，元素是 `i32` 类型的 Vec 写做 `Vec<i32>`。
- 使用 `Vec::new()` 或 `vec!` 宏来创建 Vec。
 - `Vec::new()` 是名字空间的例子，`new` 是定义在 Vec 结构体中的函数。

向量 (续)

```
// Explicit typing
let v0: Vec<i32> = Vec::new();

// v1 and v2 are equal
let mut v1 = Vec::new();
v1.push(1);
v1.push(2);
v1.push(3);

let v2 = vec![1, 2, 3];

// v3 and v4 are equal
let v3 = vec![0; 4];
let v4 = vec![0, 0, 0, 0];
```

向量 (续)

```
let v2 = vec![1, 2, 3];  
let x = v2[2]; // 3
```

- 向量可以像数组一样使用 [] 来访问元素。
 - 在 Rust 中不能用 i32/i64 等类型的值作为下标访问元素。
 - 必须使用 `usize` 类型的值，因为 `usize` 保证和指针是一样长度的。
 - 其他类型要显式转换成 `usize`:

```
let i: i8 = 2;  
let y = v2[i as usize];
```
- 标准库中的向量有很多有用的方法，具体可以参见 Rust 官方文档。

类型转换

- 用 `as` 进行类型转换 (cast):

```
let x: i32 = 100;  
let y: u32 = x as u32;
```

- 一般来说，只能在可以安全转换的类型之间进行转换操作。
 - 例如，`[u8; 4]` 不能转换为 `char` 类型。
 - 有不安全的机制可以做这样的事情，代价是编译器就无法再确保安全性。

引用

- 在类型前面写 `&` 表示引用类型: `&i32`。
- 用 `&` 来取引用 (和 C++ 类似)。
- 用 `*` 来解引用 (和 C++ 类似)。
- 在 Rust 中, 引用保证是合法的。
 - 合法性要通过编译时检查。
- 因此, Rust 中引用和一般意义的指针是不一样的。
- 引用的生命周期比较复杂, 后面会详细讨论。

```
let x = 12;  
let ref_x = &x;  
println!("{}", *ref_x); // 12
```

条件语句

```
if x > 0 {  
    10  
} else if x == 0 {  
    0  
} else {  
    println!("Not greater than zero!");  
    -10  
}
```

- 与 C++ 不同，条件部分不需要用小括号括起来。
- 整个条件语句是当做一个表达式来求值的，因此每个分支都必须是相同类型的表达式。
 - 当然，如果作为普通的条件语句来使用的话，可以令类型是 `()`。

```
if x <= 0 {  
    println!("Too small!");  
}
```

循环语句

- Rust 有三种循环：
 - while
 - loop
 - for
- break 和 continue 用于改变循环中的控制流。

while 循环语句

- while 的用法与 C++ 相同 (和 if 一样, 条件部分不需要用小括号括起来):

```
let mut x = 0;
while x < 100 {
    x += 1;
    println!("x: {}", x);
}
```

loop 循环语句

- `loop` 相当于 `while true`, 或者是 C++ 中的 `for (;;)。`
- `loop` 循环中的 `break` 语句可以返回一个值, 作为整个循环的求值结果 (另外两种循环没有这个功能)。

```
let mut x = 0;
let y = loop {
    x += 1;
    if x * x >= 100 {
        break x;
    }
};
```

for 循环语句

- `for` 和 C++ 中的范围循环 `for (auto x : v)` 相似, 使用迭代器 (iterators) 表达式:
 - `n..m` 创建一个从 `n` 到 `m` 半闭半开区间的迭代器。
 - `n..=m` 创建一个从 `n` 到 `m` 闭区间的迭代器。
 - 很多数据结构可以当做迭代器来使用, 比如数组、切片, 还有向量 `Vec` 等等。

```
// Loops from 0 to 9.
for x in 0..10 {
    println!("{}", x);
}
let xs = [0, 1, 2, 3, 4];
// Loop through elements in a slice of `xs`.
for x in &xs {
    println!("{}", x);
}
```

匹配语句

```
let x = 3;
match x {
  1 => println!("one fish"), // <- comma required
  2 => {
    println!("two fish");
    println!("two fish");
  }, // <- comma optional when using braces
  _ => println!("no fish for you"), // "otherwise" case
}
```

- 匹配语句由一个表达式 (x) 和一组 value => expression 的分支语句组成。
- 整个匹配语句被视为一个表达式来求值。
 - 与 if 类似, 所有分支都必须是同样类型的值。
- 下划线 (__) 用于捕捉所有情况。

匹配语句 (续)

```
let x = 3;
let y = -3;
match (x, y) {
    (1, 1) => println!("one"),
    (2, j) => println!("two, {}", j),
    (_, 3) => println!("three"),
    (i, j) if i > 5 && j < 0 => println!("On guard!"),
    (_, _) => println!(":<"),
}
```

- 匹配的表达式可以是任意表达式，包括元组和函数调用。
 - 构成模式 (patterns)。
 - 匹配可以绑定变量，_ 用来忽略不需要的部分。
- 为了通过编译，必须写穷尽的匹配模式。
- 可以用 if 来限制匹配的条件。

模式绑定

- 模式可以非常复杂，后面还会讲解到。
- 模式 (patterns) 也可以用来声明并绑定变量：

```
let (a, b) = ("foo", 12);
```

函数定义

```
fn foo(x: T, y: U, z: V) -> T {  
    // ...  
}
```

- `foo` 是一个函数 (function)，有三个参数：
 - `T` 类型的参数 `x`
 - `U` 类型的参数 `y`
 - `V` 类型的参数 `z`
- 返回值的类型是 `T`。
- `Rust` 必须显式定义函数的参数和返回值的类型。
 - 实际上编译器是可以推断函数的参数和返回值的类型的，但是 `Rust` 的设计者认为显式指定是一种更好的实践。

函数的返回

- 函数的最后一个表达式是它的返回值。
 - 可以用 `return` 来提前返回。

```
fn square(n: i32) -> i32 {  
    n * n  
}  
  
fn squareish(n: i32) -> i32 {  
    if n < 5 { return n; }  
    n * n  
}  
  
fn square_bad(n: i32) -> i32 {  
    n * n;  
}
```

- 最后一个无法通过编译，为什么？

宏

- 宏看起来像函数，但是名字以 `!` 结尾。
- 可以做很多有用的事情。
 - 宏的原理是在编译时产生代码。
- 宏调用的方式看起来和函数类似。
- 用户可以自己来定义宏。
- 很多常用工具是用宏来实现的。

print! 和 println!

- 用于输出文字信息。
- 使用 {} 来做字符串插入，{:?} 做调试输出。
 - 有些类型，例如数组和向量，只能用调试输出的方式来打印。
- {} 里可以加数字，表示第几个参数，新版本还可以把变量名写在 {} 里。

```
let x = "foo";  
print!("{}", {}, {}, x, 3, true);  
// => foo, 3, true  
println!("{:?}", {:?})", x, [1, 2, 3]);  
// => "foo", [1, 2, 3]  
let y = 1;  
println!("{0}, {y}, {0}", x);  
// => foo, 1, foo
```

format!

- 使用与 `print!/println!` 相同的用法来创建 `String` 字符串。

```
let fnted = format!("{}", {:x}, {:?})", 12, 155, Some("Hello"));  
// fnted == "12, 9b, Some("Hello")"
```

panic!

- 恐慌：显示提示信息，并退出当前任务。
- 是一种处理错误的方式，然而并不优雅。
- 如果有更好的处理方式，建议不要使用 `panic!`。

```
if x < 0 {  
    panic!("Kaboom!");  
}
```

assert! 和 assert_eq!

- 如果条件 `condition` 不成立, `assert!(condition)` 会导致恐慌。
- 如果 `left != right`, `assert_eq!(left, right)` 会导致恐慌。
- 非常有用, 用于测试和捕捉非法条件。

```
#[test]
fn test_something() {
    let actual = 1 + 2;
    assert!(actual == 3);
    assert_eq!(3, actual);
}
```

unreachable!

- 用于表示不会达到的代码分支。
- 如果运行到就会导致恐慌。
- 可用来追踪意料之外的问题。

```
if false {  
    unreachable!();  
}
```

unimplemented!

- `panic!("not yet implemented")` 的简写
- 可用于标注还没有实现的功能（例如作业里要补全的地方）。

```
fn sum(x: Vec<i32>) -> i32 {  
    // TODO  
    unimplemented!();  
}
```


- 名字
 - 骆驼形式 (CamelCase): 类型名
 - 蛇形形式 (snake_case): 变量名、函数名
- 缩进与空白
 - Rust 的缩进用 4 个空格字符。
 - 符号前后的空格
 - 操作符前后各有一个空格, 例如 `x + 1`。
 - 分隔符后面有一个空格, 例如 `f(x, 1)`。
- 注释之道
 - 写注释
 - 写有意义的注释
 - 尽量用英文写注释

4

小结

本讲小结

- Rust 语言介绍
- 简单程序的编译和运行
- Cargo 的初步使用
- Rust 的基本语法

下一讲：所有权与结构化数据