Rust基础语法概述

2019-08-19

Rust是复杂度和应用场景都对标C++的语言，一起学习吧！

最近，我开始思考像本文这样类型的内容算什么，编程语言的教程？内容不全面；对语言的评价？够不着；学习笔记？如果是，那绝非我本意。我倾向于认为这是一个探索的过程，无论对于我自己还是对于别人，我希望可以表现出来的是，你看，新的编程语言没什么神秘的，它如此简单！有的程序员终其一生，都将某种语言作为自己职业头衔的前缀，“Java程序员”或是“后端开发”，我们该跳出这种怪圈。

语句

Rust必须以;结尾。

常量和变量

Rust使用`let`定义常量，使用`let mut`定义变量。这样的写法可能稍微有点奇怪：

```
fn main() {
  let x = 1;
  println!("{}", x);

  let mut y = 2;
  println!("{}", y);

  y = 3;
  println!("{}", y);
}
```

不同于其他语言的是，Rust允许在同一作用域中多次声明同一常量。也就是说，Rust里的常量虽然不可以被第二次赋值，但是同一常量名可以被多次定义。我们虽然能在系统层面明白常量和变量的区别，但是写法上稍微有点容易引起混淆。我多次给同一组符号赋值，这个符号不就是变量吗？

```
fn main() {
  let x = 1;
  println!("{}", x);

  let x = 2;
  println!("{}", x);
}
```

另一个有点奇怪的地方是，Rust的变量不允许重复定义。我们无法推测语言设计者的初衷，这明显不是为了允许重复定义而允许。也许，Rust中只存在常量，`mut`关键字的作用就是给常量一个可以被多次赋值的接口。没有mut，常量就是个常量，有了mut，常量就有了获得新值的“入口”。至于变量重复定义的问题，要啥自行车？

```
fn main() {
  let mut x = 1;
  let mut x = 2;
}
// warning: variable does not need to be mutable
```

控制流

Rust的条件部分不需要写小括号，和Go语言一样。谁先谁后呢？

```
fn main() {
  let number = 2;
  if number == 1 {
    println!("1")
  } else if number == 2 {
    println!("2")
  } else {
    println!("3")
  }
}
```

由于if语句本身是一个表达式，所以也可以嵌套进赋值语句中，实现类似其他语言三目运算符的功能。（Rust是强类型的语言，所以赋值类型必须一致。）

```
fn main() {
  let number = if true {
    3
  } else {
    4
  };
  println!("{}", number);
}
```

与Go语言简洁的多功能for循环相比，Rust支持多种类型的循环：

```
fn main() {
  loop {
    // ...
  }

  while true {
    // ...
  }

  let a = [1, 2, 3];
  for item in a.iter() {
    println!("{}", item);
  }
}
```

函数与值的传递

Rust似乎不存在值传递与引用传递的区别，因为Rust中全都是引用传递，或者分类为常量的传递与变量的传递。对比Java中字符串的创建，Rust中创建字符串也可以使用“声明对象”的方式：

```
fn main() {
  // 常量传递
  let a = String::from("a");
```

```
  testa(&a);

  // 变量传递
  let mut b = String::from("b");
  testb(&mut b);
  println!("{}", b);
}

fn testa(a: &String) {
  println!("{}", a);
}

fn testb(b: &mut String) {
  b.push_str(" b");
}
```

函数当然也是可以有返回值的，Rust中函数的返回值用->定义类型，默认将函数最后一行的值作为返回值，也可以手动return提前结束函数流程。需要注意的是，在最后一行用来作为返回值的表达式，记得不要加封号……

```
fn main() {
  let mut a = test();
  println!("{}", a);

  a = test2();
  println!("{}", a);
}

fn test() -> u32 {
  1
}

fn test2() -> u32 {
  return 2;
}
```

结构体

结构体的基本用法比较常规，没有new关键字，直接“实例化”就可以使用：

```
struct Foo {
  a: String,
  b: i32
}

fn main() {
  let t = Foo {
    a: String::from("a"),
    b: 1,
  };

  println!("{}, {}", t.a, t.b);
}
```

同样可以给结构体添加方法：

```
struct Foo {
  a: String,
  b: i32
}

impl Foo {
  fn test(&self) -> i32 {
    self.b + 1
  }
}

fn main() {
  let t = Foo {
    a: String::from("a"),
    b: 1,
  };

  println!("{}, {}, {}", t.a, t.b, t.test());
}

// a, 1, 2
```

列表与模式匹配

下面的例子创建了包含3个元素的向量，然后将第0个元素赋值给常量one。之后使用模式匹配判断列表的第0个元素是否等于one的值，如果相等则输出字符串”one”、否则为”none”。Rust的模式匹配中、Some()和None都是内置的关键字：

```
fn main() {
  let v = vec![1, 2, 3];

  let one = &v[0];
  println!("{}", one);

  match v.get(0) {
    Some(one) => println!("one"),
    Some(2) => println!("two"),
    None => println!("none"),
  }
}
```

错误处理

panic函数用于抛出异常：

```
fn main() {
  panic!("new Exception");
}
// thread 'main' panicked at 'new Exception', test.rs:4:3
// note: Run with `RUST_BACKTRACE=1` environment variable to display a backtrace.
```

针对错误处理，Rust提供了两个简写的方法，用于便捷的处理错误信息。unwrap()函数会自动抛出panic，如果不使用unwrap()，程序则会跳过发生panic的代

码。这在某种程度上与Java的异常处理逻辑相反，因为Java如果不对异常进行处理，程序就无法继续运行。而Rust如果使用unwrap()对panic进行处理，程序将不再继续执行，同时打印出错误信息。

```
use std::fs::File;

fn main() {
  let f = File::open("hello.txt");
  println!("a");

  let f2 = File::open("hello.txt").unwrap();
  println!("b");
}

// a
// thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value: Os { code: 2, kind: NotFound, message: "系统找不到指定的文件。" }', s
// ...
```

另一个简写的方法是expect()，可用于替代unwrap()。它与unwrap()的区别在于，unwrap()使用系统内置的panic信息，而expect()可以传入参数作为panic的错误信息。仅此而已。

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt").expect("Failed to open hello.txt");
}

// thread 'main' panicked at 'Failed to open hello.txt: ...
// ...
```

Lambda表达式

Rust中的Lambda表达式使用`|`作为入参的界定符，即使用`||`来代替`()`。此外Lambda的公用和其它语言是相同的：

```
fn main() {
  let test = |num| {
    num == 1
  };

  println!("{}, {}", test(1), test(2));
}
// true, false
```

其他

Rust的语言特性远不止此，尤其是Rust与众不同的内存管理机制，以及让Rust新手得其门不得其道的概念”ownership”，都需要我们不断前行。