对 Rust 语言的分析

Rust 是一门最近比较热的语言，有很多人问过我对 Rust 的看法。由于我本人是一个语言专家，实现过几乎所有的语言特性，所以我不认为任何一种语言是新的。任何“新语言”对我来说，不过是把早已存在的语言特性（或者毛病），挑一些出来放在一起。所以一般情况下我都不会去评论别人设计的语言，甚至懒得看一眼，除非它历史悠久（比如像 C 或者 C++），或者它在工作中惹恼了我（像 Go 和 JavaScript 那样）。这就是为什么这些人问我 Rust 的问题，我一般都没有回复，或者一笔带过。

不过最近有点闲，我想既然有人这么热衷于这种新语言，那我还是稍微凑下热闹，顺便分享一下我对某些常见的设计思路的看法。所以这篇文章虽然是在评论 Rust 的设计，它却不只是针对 Rust。它是针对某些语言特性，而不只是针对某一种语言。

由于我这人性格很难闭门造车，所以现在我只是把这篇文章的开头发布出来，边写边更新。所以你要明白，这只是一个开端，我会按自己理解的进度对这篇文章进行更新。你看了之后，可以隔一段时间再回来看新的内容。如果有特别疑惑的问题，也可以发信来问，我会汇总之后把看法发布在这里。

变量声明语法

Rust 的变量声明跟 Scala 和 Swift 的很像。你用

```
let x = 8;
```

这样的构造来声明一个新的变量。大部分时候 Rust 可以推导出变量的类型，所以你不一定需要写明它的类型。如果你真的要指明变量类型，需要这样写：

```
let x: i32 = 8;
```

在我看来这是丑陋的语法。本来语义是把变量 x 绑定到值 8，可是 x 和 8 之间却隔着一个“i32”，看起来像是把 8 赋值给了 i32……

变量缺省都是不可变的，也就是不可赋值。你必须用一种特殊的构造

```
let mut x = 8;
```

来声明可变变量。这跟 Swift/Scala 的 `let` 和 `var` 的区别是一样的，只是形式不大一样。

变量可以重复绑定

Rust 的变量定义有一个比其它语言更奇怪的地方，它可以让你在同一个作用域里面“重复绑定”同一个名字，甚至可以把它绑定到另外一个类型：

```
let mut x: i32 = 1;
x = 7;
let x = x; // 这两个 x 是两个不同的变量

let y = 4;
// 30 lines of code ...
let y = "I can also be bound to text!";
// 30 lines of code ...
println!("y is {}", y);      // 定义在第二个 let y 的地方
```

在 Yin 语言最初的设计里面，我也是允许这样的重复绑定的。第一个 y 和 第二个 y 是两个不同的变量，只不过它们碰巧叫同一个名字而已。你甚至可以在同一行出现两个 x，而它们其实是不同的变量！这难道不是一个很酷，很灵活，其他语言都没有的设计吗？后来我发现，虽然这实现起来没什么难度，可是这样做不但没有带来更大的方便性，反而可能引起程序的混淆不清。在同一个作用域里面，给两个不同的变量起同一个名字，这有什么用处呢？自找麻烦而已。

比如上面的例子，在下面我们看到一个对变量 y 的引用，它是在哪里定义的呢？你需要在头脑中对程序进行“数据流分析”，才能找到它定义的位置。从上面读起，我们看到 `let y = 4`，然而这不一定是正确的定义，因为 y 可以被重新绑定，所以我们必须继续往下看。30 行代码之后，我们看到了第二个对 y 的绑定，可是我们仍然不能确定。继续往下扫，30行代码之后我们到了引用 y 的地方，没有再看到其它对 y 的绑定，所以我们才能确信第二个 let 是 y 的定义位置，它是一个字符串。

这难道不是很费事吗？更糟的是，这种人工扫描不是一次性的工作，每次看到这个变量，你都要疑惑一下它是什么东西，因为它可以被重新绑定，你必须重新确定一下它的定义。如果语言不允许在同一个作用域里面重复绑定同一个名字，你就根本不需要担心这个事情了。你只需要在作用域里面找到唯一的那个 `let y = ...`，那就是它的定义。

也许你会说，只有当有人滥用这个特性的时候，才会导致问题。然而语言设计的问题往往就在于，一旦你允许某种奇葩的用法，就一定会有人自作聪明去用。因为你无法确信别人是否会那样做，所以你随时都得提高警惕，而不能放松下心情来。

类型推导

另外一个很多人误解的地方是类型推导。在 Rust 和 C# 之类的语言里面，你不需要像 Java 那样写

```
int x = 8;
```

这样显式的指出变量的类型，而是可以让编译器把类型推导出来。比如你写：

```
let x = 8;  // x 的类型推导为 i32
```

编译器的类型推导就可以知道 x 的类型是 i32，而不需要你把“i32”写在那里。这似乎是一个很方便的东西。然而看过很多 C# 代码之后你发现，这看似方便，却让程序变得不好读。在看 C# 代码的时候，我经常看到一堆的变量定义，每一个的前面都是 var。我没法一眼就看出它们表示什么，是整数，bool，还是字符串，还是某个用户定义的类？

```
var correct = ...;
var id = ...;
var slot = ...;
var user = ...;
var passwd = ...;
```

我需要把鼠标移到变量上面，让 Visual Studio 显示出它推导出来的类型，可是鼠标移开之后，我可能又忘了它是什么。有时候发现看同一片代码，都需要反复的做这件事，鼠标移来移去的。而且要是没有 Visual Studio，用其它编辑器，或者在 github 上看代码或者 code review 的时候，你就得不到这种信息了。很多 C# 程序员为了避免这个问题，开始用很长的变量名，把类型的名字加在变量名字里面去，这样一来反而更复杂了，却没有想到直接把类型写出来。所以这种形式的类型推导，看似先进或者方便，其实还不如直接在声明处写下变量的类型，就像 Java 那样。

所以，虽然 Rust 在变量声明上似乎有更灵活的设计，然而我觉得 C 和 Java 之类的语言那样看似死板的方式其实更好。我建议不要使用 Rust 变量的重复绑定，避免使用类型推导，尽量明确的写出类型，以方便读者。如果你真的在乎代码的质量，就会发现大部分时候你的代码的读者是你自己，而不是别人，因为你需要反复的阅读和提炼你的代码。

动作的“返回值”

Rust 的文档说它是一种“大部分基于表达式”的语言，并且给出这样一个例子：

```
let mut y = 5;
let x = (y = 6);  // x has the value `()`, not `6`
```

奇怪的是，这里变量 x 会得到一个值，空的 tuple，()。这种思路不大对，它是从像 OCaml 那样的语言照搬过来的，而 OCaml 本身就有问题。在 OCaml 里面，如果你使用 print_string，那你会得到如下的结果：

```
print_string "hello world!\n";;

hello world!
- : unit = ()
```

这里，print_string 是一个“动作”，它对应过程式语言里面的“statement”。就像 C 语言的 printf。动作通常只产生“副作用”，而不返回值。在 OCaml 里面，为了“理论的优雅”，动作也会返回一个值，这个值叫做 ()。其实 () 相当于 C 语言的 void。C 语言里面有 void 类型，然而它却不允许你声明一个 void 类型的变量。比如你写

```
int main()
{
  void x;
}
```

程序是没法编译通过的（试一试？）。让人惊讶的是，古老的 C 的做法其实是正确的，这里有比较深入的原因。如果你把一个类型看成是一个集合（比如 int 是机器整数的集合），那么 void 所表示的集合是个空集，它里面是不含有任何元素的。声明一个 void 类型的变量是没有任何意义的，因为它不可能有一个值。如果一个函数返回 void，你是没法把它赋值给一个变量的。

可是在 Rust 里面，不但动作（比如 y = 6 ）会返回一个值 ()，你居然可以把这个值赋给一个变量。其实这是错误的作法。原因在于 y = 6 只是一个“动作”，它只是把 6 放进变量 y 里面，这个动作发生了就发生了，它根本不应该返回一个值，它不应该可以出现在 let x = (y = 6); 的右边。就算你牵强附会说 y = 6 的返回值是 ()，这个值是没有任何用处的。更不要说使用空的 tuple 来表示这个值，会引起更大的类型混淆，因为 () 本身有另外的，更有用的含义。

你根本就不应该可以写 `let x = (y = 6);` 这样的代码。只有当你犯错误或者逻辑不清晰的时候，才有可能把 `y = 6` 当成一个值来用。Rust 允许你把这种毫无意义的返回值赋给一个变量，这种错误就没有被及时发现，反而能够通过变量传播到另外一个地方去。有时候这种错误会传播挺远，然后导致问题（运行时错误或者类型检查错误），可是当它出问题的时候，你就不大容易找到错误的起源了。

这是很多语言的通病，特别是像 JavaScript 或者 PHP 之类的语言。它们把毫无意义或者牵强附会的结果（比如 undefined）到处传播，结果使错误很难被发现和追踪。

return 语句

Rust 的设计者似乎很推崇“面向表达式”的语言，所以在 Rust 里面你不需要直接写“return”这个语句。比如，这个例子里面，你可以直接这样写：

```
fn add_one(x: i32) -> i32 {
    x + 1
}
```

返回函数里的最后一个表达式，而不需要写 return 语句，这是函数式语言共有的特征。然而其实我觉得直接写 return 其实是更好的作法，像这个样子：

```
fn foo(x: i32) -> i32 {
    return x + 1;
}
```

编程有一个容易引起问题的作法，叫做“不够明确”，总想让编译器自动去处理一些问题，在这里也是一样的问题。如果你隐性的返回函数里最后一个表达式，那么每一次看见这个函数，你都必须去搞清楚最后一个表达式是什么，这并不是每次都那么明显的。比如下面这段代码：

```
fn main() {
    println!("{}", add_one(7));
}

fn add_one(x: i32) -> i32 {
  if (x < 5) {
      if (x < 10) {
        // 做很多事...
        x * 2
      } else {
        // 做很多事...
        x + 1
      }
  } else {
    // 做很多事...
    x / 2
  }
}
```

由于 if 语句里面有嵌套，每个分支又有好些代码，而且 if 语句又是最后一个语句，所以这个嵌套 if 的三个出口的最后一个表达式都是返回值。如果你写了“return”，那么你可以直接看有几个“return”，或者拿编辑器加亮一下，就知道这个函数有几个出口。然而现在没有了“return”这个关键字，你就必须把最后那个 if 语句自己看清楚了，找到每一个分支的“最后表达式”。很多时候这不是那么明显，你总需要找一下，而且这件事在读代码的时候总是反复做。

所以对于返回值，我的建议是总是明确的写上“return”，就像第二个例子那样。Rust 的文档说这是“poor style”，那不是真的。有一个例外，那就是当函数体里面只有一条语句的时候，那个时候没有任何歧义哪一个是返回表达式。

这个问题类似于重复绑定变量和类型推导的问题，属于一种“用户体验设计”问题。无论如何，编译器都很容易实现，然而不同样式的代码，对于人类阅读的工作量，是很不一样的。很多时候最省人力的做法并不是那种看来最聪明，最酷，打字量最少的办法，而是写得最明确，让读者省事的办法。人们常说，代码读的时候比写的时候多得多，所以要想语言好用省事，我们应该更加重视读的时候，而不是写的时候。

数组的可变性

Rust 的数组可变性标记，跟 Swift 犯了一样的错误。Swift 的问题，我已经在之前的文章有详细叙述，所以这里就不多说了。简言之，同一个标记能表示的可变性，要么针对数组指针，要么针对数组元素，应该只能选择其一。而在 Rust 里面，你只有一个地方可以放“mut”进去，所以要么数组指针和元素全部都可变，要么数组指针和元素都不可变。你没有办法制定一个不可变的数组指针，而它指向的数组的元素却是可变的。

请对比下面两个例子：

```
fn main() {
    let m = [1, 2, 3];      // 指针和元素都不可变
```

```
    m[0] = 10;              // 出错
    m = [4, 5, 6];          // 也出错
}

fn main() {
    let mut m = [1, 2, 3];  // 指针和元素都可变
    m[0] = 10;              // 不出错
    m = [4, 5, 6];          // 也不出错
}
```

内存管理

Rust 号称实现了非常先进的内存管理机制，不需要垃圾回收（GC）或者引用计数（RC）就可以“静态”的管理内存的分配和释放。然而仔细思考之后你就会发现，这很可能是不切实际的梦想（或者广告）。内存的分配和释放（如果要及时释放的话），本身是一个动态的过程，无法用静态分析来实现。现在你说可以通过一些特殊的构造，特殊的指针和传值方式，静态的决定内存的回收时间，真的有可能吗？

实际上我有一个类似的梦。我曾经向我的教授们提出过 N 多种不需 GC 和 RC 就能静态管理内存的办法，结果每一次都被他们给我的小例子给打败了，以至于我很难相信有任何人可以想到比 GC 和 RC 更好的方法。

Rust 那些炫酷的 move semantics, borrowing, lifetime 之类的概念加在一起，不但让语言变得复杂不堪，我感觉并不能从根本上解决内存管理问题。很多人在 blog 里面为这些概念热情洋溢地做宣传，显得自己很懂一样，拿一些玩具代码来演示，可是从没看到任何人说清楚这些东西为什么可以从根本上解决问题，能用到复杂一点的代码里面去。所以我觉得这些东西有“皇帝的新装”之嫌。

连 Rust 自己的文档都说，你可能需要“fight with the borrow checker”。为了通过这些检查，你必须用很怪异的方式来写程序，随着问题复杂度的增加，就要求有更怪异的写法。如果用了 lifetime，很简单一个代码看起来就会是这种样子。真够烦的，我感觉我的眼睛都没法 parse 这段代码了。

```
fn foo<'a, 'b>(x: &'a str, y: &'b str) -> &'a str {
}
```

上一次我看 Rust 文档的时候，没发现有 lifetime 这概念。文档对此的介绍非常粗略，仔细看了也不知道他们在说些什么，更不要说相信这办法真的管用了。对不起，我根本不想去理解这些尖括号里的 `'a` 和 `'b` 是什么，除非你先向我证明这些东西真的能解决内存管理的问题。实际上这个 lifetime 我感觉像是跨过程静态分析时产生的一些标记，要知道静态分析是无法解决内存管理的问题的，我猜想这种 lifetime 在有递归函数的情况下就会遇到麻烦。

实际上我最开头看 Rust 的时候，它号称只用 move semantics 和好几种不同的指针，就可以解决内存管理的问题。可是一旦有了那几种不同的指针，就已经复杂不堪了，比 C 语言还要麻烦，而且显然不能解决问题。Lifetime 恐怕是后来发现有新的问题解决不了才加进去的，可是我不知道他们这次是不是又少考虑了某些情况。

Rust 的设计者显然受了 Linear Logic 一类看似很酷的逻辑的启发和熏陶，想用类似的方式奇迹般的解决内存和资源的回收问题。然而研究过一阵子 Linear Logic 之后我发现，这个逻辑自己都没有解决任何问题，只不过给对象的引用方式施加了一些无端的限制，这样使得对象的引用计数是一个固定的值（1）。内存管理当然容易了，可是这样导致有很多程序你没法表达。

开头让你感觉很有意思，似乎能解决一些小问题。到后来遇到大一点的实际问题的时候，你就发现需要引入越来越复杂的概念，使用越来越奇葩的写法，才能达到目的，而且你总是会在将来某个时候发现它没法解决的问题。因为这个问题很可能从根本上是无法解决的，所以每当遇到有超越现有能力的事情，你就得增加新的“绕过方法”（workaround）。缝缝补补，破败不堪。最后你发现，除了垃圾回收（GC）和引用计数（RC），内存管理还是没有其它更好更简单的办法。

当然我的意见也许不是完全准确，可我真是没有时间去琢磨这么多乱七八糟，不知道管不管用的概念（特别是 lifetime），更不要说真的用它来构建大型的系统程序了。有用来理解这些概念，把程序改成奇葩样子的时间，我可能已经用 C 语言写出很好的手动内存管理代码了。如果你真的看进去理解了，发现这些东西可以用的话，告诉我一声！不过你必须说明原因，不要只告诉我“皇帝是穿了衣服的” :P

完

本来想写一个更详细的评价的，可是到了这个地方，我感觉已经失去兴趣了，困就一个字啊…… Rust 比 C 语言复杂太多，我很难想象用这样的语言来构造大型的操作系统。而构造系统程序，是 Rust 设计的初衷。说真的，写操作系统那样的程序，C 语言真的不算讨厌。用户空间的程序，Java，C# 和 Swift 完全可以胜任。所以我觉得 Rust 的市场空间恐怕非常狭小……

（如果你喜欢这些内容，请付费5美元或者30人民币，谢谢！）