值和引用初步解释

值和引用是编程中经常需要面对的两个概念，本文将结合主流编程语言的做法，在抽象和实现层面对这两个概念进行初步的解释。

地址的概念

在解释值和引用之前，我们先看一段 RISC 风格的汇编代码，它完成的事情是将变量 y 的值赋给变量 x.

```
// x = y

ld r1, y
st r1, x
```

在这段代码中，x 和 y 表示的都是变量的地址。两行代码的行为分别是：将地址 y 的内存位置保存的数值读取出来，存放在寄存器 r1 中，然后将寄存器 r1 内的值保存到地址 x 的内存位置。所以，x = y 这条语句中，我们关心的左值是变量 x 的地址，右值是变量 y 的值。这两者是有明显不同的。

以一个简单的 C 程序为例：

```
void main() {
  int a;
  int b;
  a = 5;
  b = a;
}
```

它的 RISC 风格的汇编代码是：

```
// a: 0×0100
// b: 0×0104
// r0 恒为零

addi r1, r0, 5
st   r1, 0×0100
ld   r2, 0×0100
st   r2, 0×0104
```

这部分代码是基于 MIPS 架构的，读者不用关心一些不重要的细节（变量的地址、r0 为什么为 0、r1 为什么不重用等），只需要通过这个例子知道对于一个程序而言，地址意味着什么就够了。如果涉及到指针，熟悉 C 语言的朋友应该明白，对指针取内容就是一次内存读取，我依然给出一个简单例子：

```
// int *a;
// int b;
// b = *a;

ld   r1, a(r0) // a(r0) 表示 r0 + a, 由于 r0 是 0, 所以第二个参数也就是 a
ld   r2, r1
st   r2, b
```

在这个实现中，取出一个指针所指向内存地址的内容需要两次 load 指令访存。有了对地址和指针概念的初步了解，我们就可以探究引用了。

变量的值和引用

在上面的例子中，我展示了 C 语言中对于 int 类型变量赋值的一般实现方式，看起来很简单，似乎也没有什么需要特别关注的地方。而熟悉 Python 或者 Java 的程序员应当知道，变量赋值不一定意味着拷贝。例如下面这一段 Java 代码：

```
Duck d1 = new Duck();
Duck d2 = d1;
```

它所做的事情并不是把 d1 的所有内容——所有字段的值——拷贝到了 d2 中。我们暂且不论 JVM 会怎么处理它，仅仅从 Java 的语义，也即它给程序员的抽象层面上看，d1 和 d2 都只是对对象的引用，它的作用等同于 C 中的指针。如果把它改写成 C++ 代码，可以很明显地发现这一点：

```
Duck *d1 = new Duck();
Duck *d2 = d1;
```

Java 是一门严格的只有引用类型和值传递的语言。在很多 Java 教材或者文档中，会有意地将 Java 中的基本类型（int, float, double）等，和引用类型（各种 class）区分开。在讨论这个问题时，其实我们需要明确语言实现和语言标准之间的不同。Java 的基本类型的确可以像 C 语言那样来解释，但如果把它想象成引用类型，也没有什么问题。举个例子：

```
int a = 5;
int b = a;
int c = a + b;
```

同样是这段代码，按照 C 的解释方式，把 int 类型当作值来解释：

```
addi r1, r0, 5  // r1 = 0 + 5 = 5
st   r1, a      // a = 5
ld   r2, a      // r2 = a = 5
st   r2, b      // b = r2 = 5
add  r3, r1, r2 // r3 = r1 + r2 = 10
st   r3, c      // c = r3 = 10
```

如果把 int 类型的变量当作引用呢？可以这样解释：对象 a 是一个整数对象 5 的引用，b 通过赋值语句也得到了整数对象 5 的引用，加法运算实则是对 a 和 b 引用的对象进行相加，结果是一个新的整数对象，c 得到了这个新的整数对象的引用。

同样能解释得通！所以在接下来的讨论中，我们将 Java 作为单纯只有引用类型的语言代表，C/C++ 作为单纯只有值类型的代表。现在抛弃复杂的汇编风格代码，看一些更加复杂的情形。

参数传递中的值和引用

学过《编译原理》的朋友应当知道，编程语言的参数传递有值传递、名传递和引用传递。有些课本难免会把这些概念阐述得很混乱。这里我们不定义名传递的概念，只看看值传递和引用传递。

C 语言只有值传递，学过 C 的朋友都能理解这样的函数是没有办法正常工作的：

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}
```

想要用交换两个整数的值（如果你使用了带参宏，那么你几乎就使用了所谓的“名传递”），在 C 中往往需要这样做：

```
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
```

这就使得你在调用函数时传递两个变量的地址。同样，Java 也只有值传递，只不过 Java 所有的变量都是引用类型，你仍然可以通过参数来修改对象本身：

```
public static void growUp(Person p) {
  p.height += 50;
```

```
}
```

这样，通过调用 growUp(p) 就能改变 p 的字段（height 是 Person 中定义的一个 public 字段），看起来和 C++ 的引用传参很相似：

```
void growUp(Person& p) {
  p.height += 50;
}
```

几乎一样嘛！千万不要跌入陷阱！C++ 版本的 growUp 函数直接对外部的变量 p 的地址进行引用，如果我将代码改成这样：

```
void growUp(Person& p) {
  p.height += 50;
  p = *new Person;
}
```

调用 growUp 函数后，p 可就变成一个新的对象了！而同样的修改方式下：

```
public static void growUp(Person p) {
  p.height += 50;
  p = new Person();
}
```

调用 growUp 函数后，p 只会长高，而不会变成新的对象。在函数体内生成的新对象会被回收，不发挥任何作用。所以说，真正和 Java 版本的 growUp 函数等价的 C++ 函数应当是这个样子的：

```
void growUp(Person* p) {
  p->height += 50;
}
```

这样的函数内部即便对 p 做了手脚，也不会影响外部的变量。值和引用在返回值上也有差别，这里不多加讨论了，以后在讨论 Golang 的时候可以再涉及这个话题。

数据抽象和引用的用途

像 Java 这样单纯只有引用类型的语言不少，Python, Ruby, JavaScript 都可以算进去，尽管它们各自的实现方式不太一样。但像 C 这样只有值类型的语言就不多了，而且这样的语言一定要有指针的支持，否则就意味着每一个变量的声明都会造成大量空间的使用，每一次赋值和传参都会造成大量空间的复制。其核心点在于，“引用”（本文提到的引用都是指 reference, 而非 quote）是

编程过程中很重要的概念和工具，它必不可少。Java 告诉大家，值类型可以消失，但没有引用是不行的。我们不如来从现代语言的设计和使用上看看引用为何如此重要。

在 C++ 中，对象的诞生往往是如下两种语句的结果：

```
Student tom(/*grade*/ 1, /*class*/ 2);
Student* tim = new Student(1, 3);
```

而被 C++ 之父 Stroustrup 认为至关重要的面向对象特征——继承和多态——需要由向上类型转换来体现：

```
Book* book = new EnglishBook;
```

如果一门语言不支持这样的自动向上类型转换，那么它就难以发挥面向对象的优势（不是完全不可以，大家可以尝试用 C 语言写一个类似的功能）。而这转换是靠引用完成的——无论你是用指针还是定义引用变量。所以，至少对于面向对象语言而言，引用所给予的数据抽象能力是必不可少的。

在 C# 和 Swift 等语言中，引用类型和值类型被同时保留了。它们对此的处理有一些共性：引用类型拷贝时拷贝地址（浅拷贝），可以继承；值类型拷贝时拷贝数据（深拷贝），不能继承。从这里也可以明确的看出，一个值类型变量无法被多次引用，所以不能提供引用类型所能提供的抽象能力，进而也就没有继承的必要。在 C# 和 Swift 的横向对比中，就这个问题上，C# 虽然年纪更大，但做得要好得多。Swift 的一个让人不能理解的设计错误就是集合类型是值类型——这是几乎不能容忍的。我猜测这是在早期的 Swift 中对 let（常量声明）的语义处理上出现的失误导致的。

（如果你喜欢本文，请出门扫码献爱心。本文定价1元）