Swift 语言的设计错误

在『编程的智慧』一文中，我分析和肯定了 Swift 语言的 optional type 设计，但这并不等于 Swift 语言的整体设计是完美没有问题的。其实 Swift 1.0 刚出来的时候，我就发现它的 array 可变性设计存在严重的错误。Swift 2.0 修正了这个问题，然而他们的修正方法却没有击中要害，所以导致了其它的问题。这个错误一直延续到今天。

Swift 1.0 试图利用 var 和 let 的区别来指定 array 成员的可变性，然而其实 var 和 let 只能指定 array reference 的可变性，而不能指定 array 成员的可变性。举个例子，Swift 1.0 试图实现这样的语义：

```
var shoppingList = ["Eggs", "Milk"]

// 可以对 array 成员赋值
shoppingList[0] = "Salad"

let shoppingList = ["Eggs", "Milk"]

// 不能对 array 成员赋值，报错
shoppingList[0] = "Salad"
```

这是错误的。在 Swift 1.0 里面，array 像其它的 object 一样，是一种“reference type”。为了理解这个问题，你应该清晰地区分 array reference 和 array 成员的区别。在这个例子里，`shoppingList` 是一个 array reference，而 `shoppingList[0]` 是访问一个 array 成员，这两者有着非常大的不同。

var 和 let 本来是用于指定 `shoppingList` 这个 reference 是否可变，也就是决定 `shoppingList` 是否可以指向另一个 array 对象。正确的用法应该是这样：

```
var shoppingList = ["Eggs", "Milk"]

// 可以对 array reference 赋值
shoppingList = ["Salad", "Noodles"]

// 可以对 array 成员赋值
shoppingList[0] = "Salad"

let shoppingList = ["Eggs", "Milk"]

// 不能对 array reference 赋值，报错
shoppingList = ["Salad", "Noodles"]

// let 不能限制对 array 成员赋值，不报错
shoppingList[0] = "Salad"
```

也就是说你可以用 var 和 let 来限制 `shoppingList` 这个 reference 的可变性，而不能用来限制 `shoppingList[0]` 这样的成员访问的可变性。

var 和 let 一旦被用于指定 array reference 的可变性，就不再能用于指定 array 成员的可变性。实际上 var 和 let 用于局部变量定义的时候，只能指定栈上数据的可变性。如果你理解 reference 是放在栈（stack）上的，而 Swift 1.0 的 array 是放在堆（heap）上的，就会明白array 成员（一种堆数据）可变性，必须用另外的方式来指定，而不能用 var 和 let。

很多古老的语言都已经看清楚了这个问题，它们明确的用两种不同的方式来指定栈和堆数据的可变性。C++ 程序员都知道 `int const *` 和 `int * const` 的区别。Objective C 程序员都知道 `NSArray` 和 `NSMutableArray` 的区别。我不知道为什么 Swift 的设计者看不到这个问题，试图用同样的关键字（var 和 let）来指定栈和堆两种不同位置数据的可变性。实际上，不可变数组和可变数组，应该使用两种不同的类型来表示，就像 Objective C 的 `NSArray` 和 `NSMutableArray` 那样，而不应该使用 var 和 let 来区分。

Swift 2.0 修正了这个问题，然而可惜的是，它的修正方式是错误的。Swift 2.0 做出了一个离谱的改动，它把 array 从 reference type 变成了所谓 value type，也就是说把整个 array 放在栈上，而不是堆上。这貌似解决了以上的问题，由于 array 成了 value type，那么 `shoppingList` 就不是 reference，而代表整个 array 本身。所以在 array 是 value type 的情况下，你确实可以用 var 和 let 来决定它的成员是否可变。

```
let shoppingList = ["Eggs", "Milk"]

// 不能对 array 成员赋值，因为 shoppingList 是 value type
// 它表示整个 array 而不是一个指针
// 这个 array 的任何一部分都不可变
shoppingList[0] = "Salad"
```

这看似一个可行的解决方案，然而它却没有击中要害。这是一种削足适履的做法，它带来了另外的问题。把 array 作为 value type，使得每一次对 array 变量的赋值或者参数传递，都必须进行拷贝。你没法让两个变量指向同一个

array，也就是说 array 不再能被共享。比如：

```
var a = [1, 2, 3]

// a 的内容被拷贝给 b
// a 和 b 是两个不同的 array，有相同的内容
var b = a
```

这违反了程序员对于数组这种大型结构的心理模型，他们不再能清晰方便的对 array 进行思考。由于 array 会被不经意的自动拷贝，很容易犯错误。数组拷贝需要大量时间，就算接收者不修改它也必须拷贝，所以效率上有很大影响。不能共享同一个 array，在里面读写数据，是一个很大的功能缺失。由于这个原因，没有任何其它现代语言（Java，C#，……）把 array 作为 value type。

如果你看透了 value type 的实质，就会发现这整个概念的存在，在具有垃圾回收（GC）的现代语言里，几乎是没有意义的。有些新语言比如 Swift 和 Rust，试图利用 value type 来解决内存管理的效率问题，然而它带来的性能提升其实是微乎其微的，给程序员带来的麻烦和困扰却是有目共睹的。完全使用 reference type 的语言（比如 Java，Scheme，Python），程序员不需要思考 value type 和 reference type 的区别，大大简化和加速了编程的思维过程。Java 不但有非常高效的 GC，还可以利用 escape analysis 自动把某些堆数据放在栈上，程序员不需要思考就可以达到 value type 带来的那么一点点性能提升。相比之下，Swift，Rust 和 C# 的 value type 制造的更多是麻烦，而没有带来实在的性能优势。

Swift 1.0 犯下这种我一眼就看出来的低级错误，你也许从中发现了一个道理：编译器专家并不等于程序语言专家。很多经验老到的程序语言专家一看到 Swift 最初的 array 设计，就知道那是错的。只要团队里有一个语言专家指出了这个问题，就不需要这样反复的修改折腾。为什么 Swift 直到 1.0 发布都没有发现这个问题，到了 2.0 修正却仍然是错的？我猜这是因为 Apple 并没有聘请到合格的程序语言专家来进行 Swift 的设计，或者有合格的人，然而他们的建议却没有被领导采纳。Swift 的首席设计师是 Chris Lattner，也就是 LLVM 的设计者。他是不错的编译器专家，然而在程序语言设计方面，恐怕只能算业余水平。编译器和程序语言，真的是两个非常不同的领域。Apple 的领导们以为好的编译器作者就能设计出好的程序语言，以至于让 Chris Lattner 做了总设计师。

Swift 团队不像 Go 语言团队完全是一知半解的外行，他们在语言方面确实有一定的基础，所以 Swift 在大体上不会有特别严重的问题。然而可以看出来这些人功力还不够深厚，略带年轻人的自负，浮躁，盲目的创新和借鉴精神。有些设计并不是出自自己深入的见解，而只是“借鉴”其它语言的做法，所以可能犯下经验丰富的语言专家根本不会犯的错误。第一次就应该做对的事情，却需要经过多次返工。以至于每出一个新的版本，就出现一些“不兼容改动”，导致老版本语言写出来的代码不再能用。这个趋势在 Swift 3.0 还要继续。由于 Apple 的统治地位，这种情况对于 Swift 语言也许不是世界末日，然而它确实犯了语言设计的大忌。一个好的语言可以缺少一些特性，但它绝不应该加入错误的设计，导致日后出现不兼容的改变。我希望 Apple 能够早日招募到资深一些的语言设计专家，虚心采纳他们的建议。BTW，如果 Apple 支付足够多的费用，我倒可以考虑兼职做他们的语言设计顾问 ;-)

Java 有 value type 吗？

有人看了以上的内容，问我：“你说 Java 只有 reference type，但是根据 Java 的官方文档，Java 也有 value type 和 reference type 的区别的。” 由于这个问题相当的有趣，我另外写了一篇文章来回答这个问题。