从Swift看Objective-C的数组使用

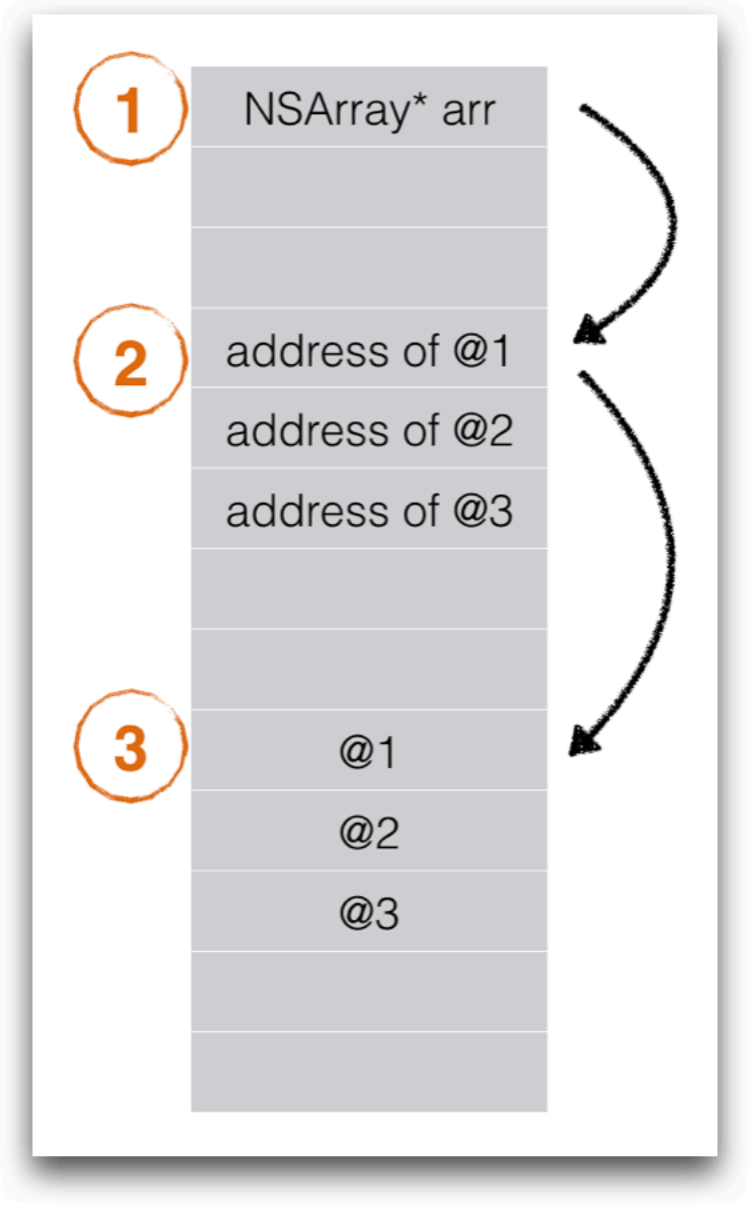
状态维护是个怎么说都不够的话题，毕竟状态的处理是我们整个App最核心的部分，也是最容易出bug的地方。之前写过一篇以函数式编程的角度看状态维护的文章，这次从Swift语言层面的改进，看看Objective C下该如何合理的处理数组的维护。

Objective C数组的内存布局

要了解NSArray，NSSet，NSDictionary这些集合类的使用方法，我们需要先弄明白其对应的内存布局（Memory Layout），以一个NSMutableArray的property为例：

```
//declare
@property (nonatomic, strong) NSMutableArray*                    arr;

//init
self.arr = @[@1, @2, @3].mutableCopy;
```

arr初始化之后，以64位系统为例，其实际的内存布局分为三块：



第一块是指针 NSMutableArray* arr 所处的位置，为8个字节。第二块是数组实际的内存区域所处的位置，为连续3个指针地址，各占8个字节一共24个字节。第三块才是@1，@2，@3这些NSNumber对象真正的内存空间。当我们调用不同的API对arr进行操作的时候，要分清楚实际是在操作哪部分内存。

比如：

```
self.arr = @[@4];
```

是在对第一块内存区域进行赋值。

```
self.arr[0] = @4;
```

是在对第二块内存区域进行赋值。

```
[self.arr[0] integerValue];
```

是在访问第三块内存区域。

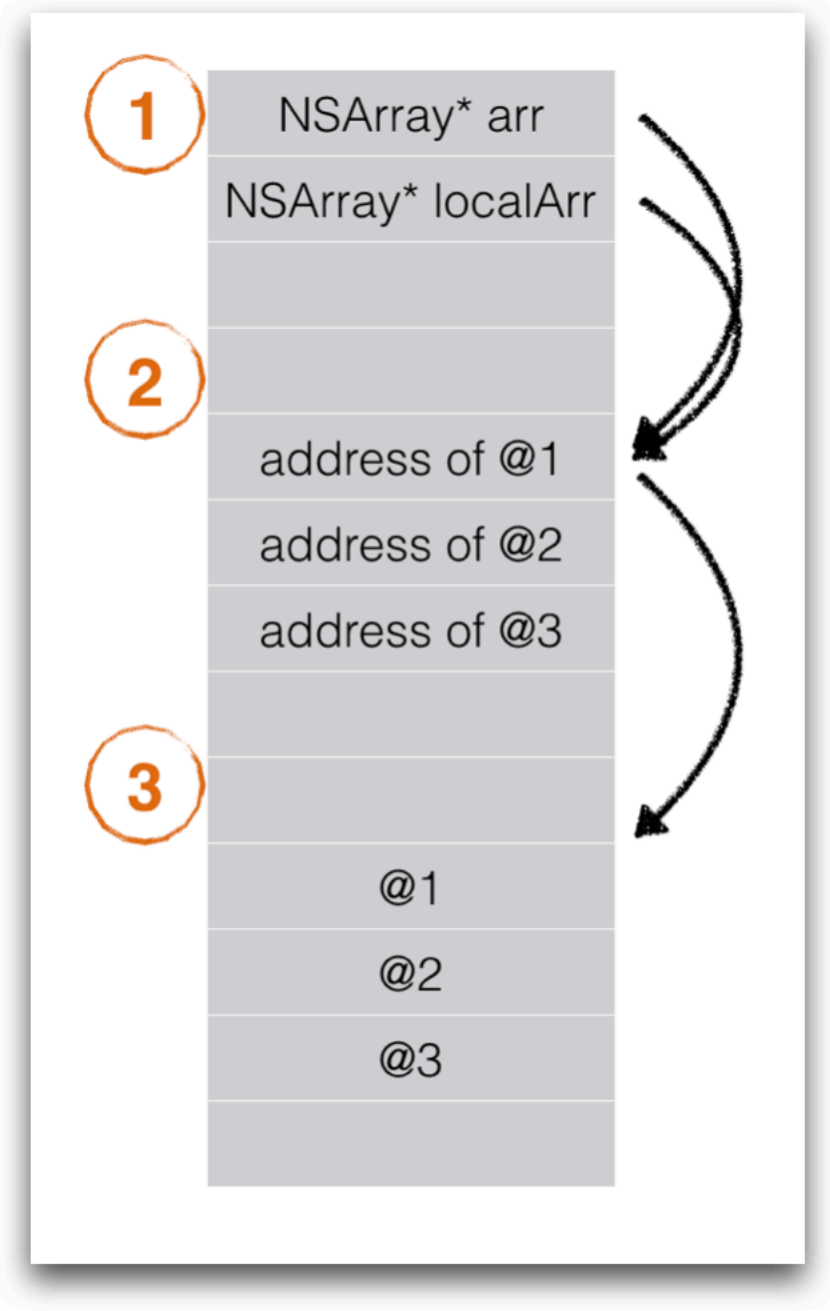
之前写过一篇多线程安全的文章，我们知道即使在多线程的场景下，对第一块内存区域进行读写都是安全的，而第二块和第三块内存区域都是不安全的。

NSMutableArray为什么危险?

在Objective C的世界里，带Mutable的都是危险分子。我们看下面代码：

```
//main thread
self.arr = @[@1, @2, @3].mutableCopy;

for (int i = 0; i < _arr.count; i ++) {
    NSLog(@"element: %@", _arr[i]);
}

//thread 2
NSMutableArray* localArr = self.arr;

//get result from server
NSArray* results = @[@8, @9, @10];

//refresh local arr
[localArr removeAllObjects];
[localArr addObjectsFromArray:results];
```

`NSMutableArray* localArr = self.arr;`执行之后，我们的内存模型是这样的：



这行代码实际上只是新生成了8个字节的第一类内存空间给localArr，localArr实际上还是和arr共享第二块和第三块内存区域，当在thread 2执行 `[localArr removeAllObjects];` 清理第二块内存区域的时候，如果主线程正在同时访问第二块内存区域 `_arr[1]`，就会导致crash了。这类问题的根本原因，还是在对于同一块内存区域的同时读写。

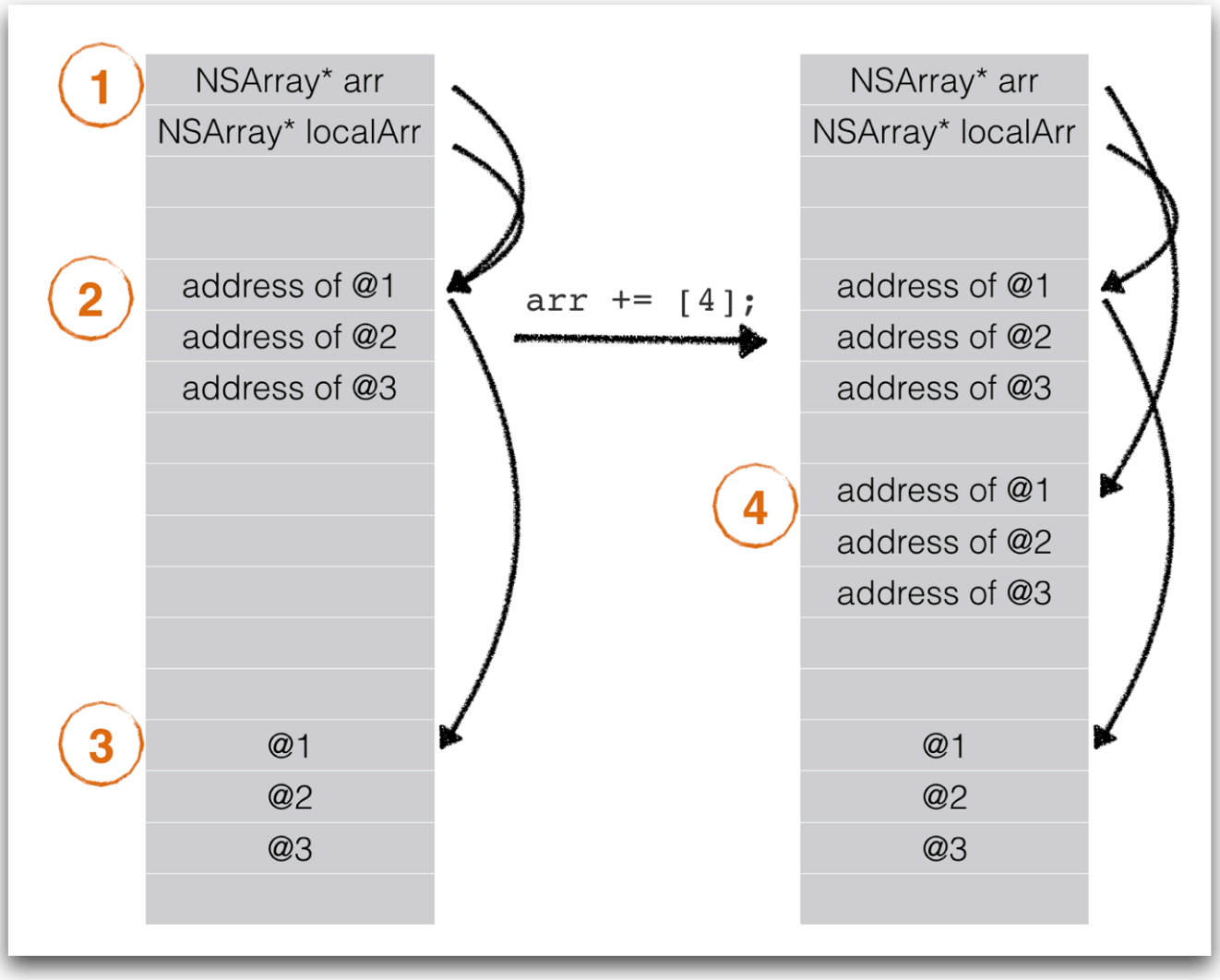
Swift的改变

Swift对于上述的数组赋值操作，从语言层面做了根本性的改变。

Swift当中所有针对集合类的操作，都符合一种叫copy on write(COW)的机制，比如下面的代码：

```
var arr = [1, 2, 3]
var localArr = arr

print("arr: \(arr)")
print("localArr: \(localArr)")

arr += [4];

print("arr: \(arr)")
print("localArr: \(localArr)")
```

当执行到 `var localArr = arr` 的时候，arr和localArr的内存布局还是和Objective C一致，arr和localArr都共享第二第三块内存区域，但是一旦出现写操作(write)，比如 `arr += [4];` 的时候，Swift就会针对原先arr的第二块内存区域，生成一份新的拷贝(copy)，也就是所谓的copy on write，执行cow之后，arr和localArr就指向不同的第二块内存区域了，如下图所示：



一旦出现针对arr写操作，系统就会将内存区域2拷贝至一块新的内存区域4，并将arr的指针指向新开辟的区域4，之后再发生数组的改变，arr和localArr就指向不同的区域，即使在多线程的环境下同时发生读写，也不会导致访问同一内存区域的crash了。

上面的代码，最后打印的结果中，arr和localArr中所包含的元素也不一致了，毕竟他们已经指向各自的第二类内存区域了。

这也是为什么说Swift是一种更加安全的语言，通过语言层面的修改，帮助开发者避免一些难以调试的bug，而这一切都是对开发者透明的，免费的，开发者并不需要做特意的适配。还是一个简单的=操作，只不过背后发生的事情不一样了。

Objective C的领悟

Objective C还没有退出历史舞台，依然在很多项目中发挥着余热。明白了Swift背后所做的事情，Objective C可以学以致用，只不过要多写点代码。

Objective C既然没有COW，我们可以自己copy。

比如需要对数组进行遍历操作的时候，在遍历之前先Copy：

```
NSArray* iterateArr = [self.arr copy];
for (int i = 0; i < iterateArr.count; i ++) {
    NSLog(@"element: %@", iterateArr[i]);
}
```

比如当我们需要修改数组中的元素的时候，在开始修改之前先Copy：

```
self.arr = @[@1, @2, @3].mutableCopy;

NSMutableArray* modifyArr = [self.arr mutableCopy];
[modifyArr removeAllObjects];
[modifyArr addObjectsFromArray:@[@4, @5, @6]];

self.arr = modifyArr;
```

比如当我们需要返回一个可变数组的时候，返回一个数组的Copy：

```
- (NSMutableArray*)createSamples
{
    [_samples addObject:@1];
    [_samples addObject:@2];

    return [_samples mutableCopy];
}
```

只要是针对共享数组的操作，时刻记得copy一份新的内存区域，就可以实现手动COW的效果，这样Objective C也能在维护状态的时候，是多线程安全的。

Copy更健康

除了NSArray之外，还有其他集合类NSSet，NSDictionary等，NSString本质上也是个集合，对于这些状态的处理，copy可以让他们更加安全。

宗旨是避免共享状态，这不仅仅是出于多线程场景的考虑，即使是在UI线程中维护状态，在一个较长的时间跨度内状态也可能出现意料之外的变化，而copy能隔绝这种变化带来的副作用。

当然copy也不是没有代价的，最明显的代价是内存方面的额外开销，一个含有100个元素的array，如果copy一份的话，在64位系统下，会多出800个字节的空间。这也是为什么Swift只有在write的时候才copy，如果只是读操作，就不会产生copy额外的内存开销。但综合来看，这点内存开销和我们程序的稳定性比起来，几乎可以忽略不计。在维护状态的时候多使用copy，让我们的函数符合Functional Programming当中的纯函数标准，会让我们的代码更加稳定。

总结

学习Swift的时候，如果细心观察，可以发现其他很多地方，也有Swift避免共享同一块内存区域的语法特性。要能真正理解这些语言背后的机制，说到底还是在于我们对于memory layout的理解。

欢迎关注公众号：MrPeakTech

上一篇
2016年iOS技术圈回顾 **(/blog/ios-2016/)**

下一篇
iOS代码耦合的处理 **(/blog/ios-coupling/)**