使用func和closure加工数据

函数和Closure真的是不同的类型么?
(https://www.boxueio.com/series/functions-and-closure/ebook/150)

OC中水土不服的运行时特性
(https://www.boxueio.com/series/functions-and-closure/ebook/152)

通过Local function捕获变量共享资源

Back to series (/series/functions-and-closure)

字号
字号
默认主题
金色主题
暗色主题

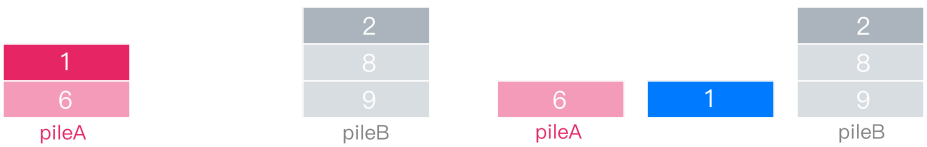
在上一节，了解了Swift中local function可以捕获变量的特性之后，除了用捕获的变量保存状态，我们还可以用它来在多次调用之间共享资源，以提高程序的执行效率。在这一节，我们就通过归并排序的实现，来了解这个用法。

实现原理的简单回顾

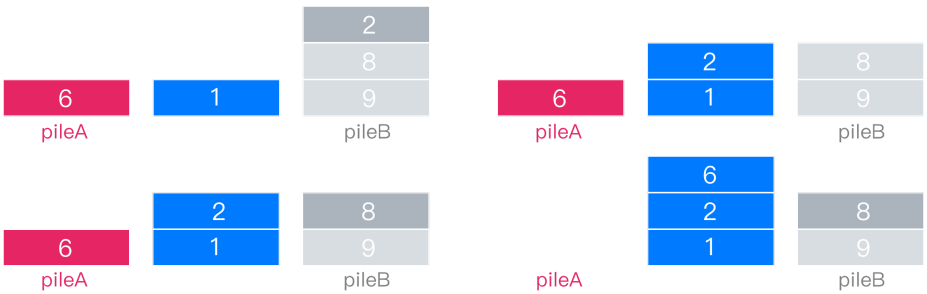
在开始具体的编码前，我们先来回顾下算法的原理。假设，我们有**两摞已经排好序的数字** `pileA` 和 `pileB`，它们各自的值，如图所示：

要把这两摞数字从小到大合并起来，该怎么做呢？

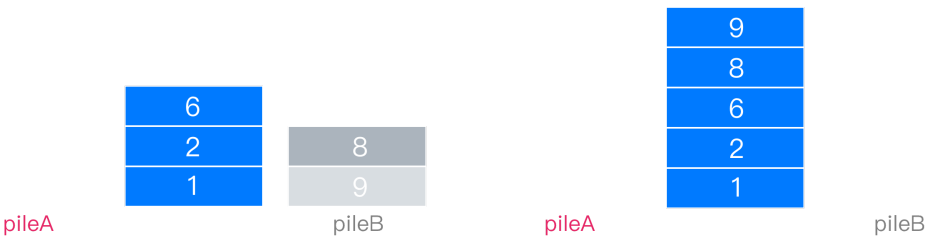
首先，我们找到 `pileA` 和 `pileB` 顶端两个数字中，更小的一个，并把它单独拿出来：



其次，我们反复执行第一步的过程，不断从两摞数据的顶端拿走更小的一个放到结果里：



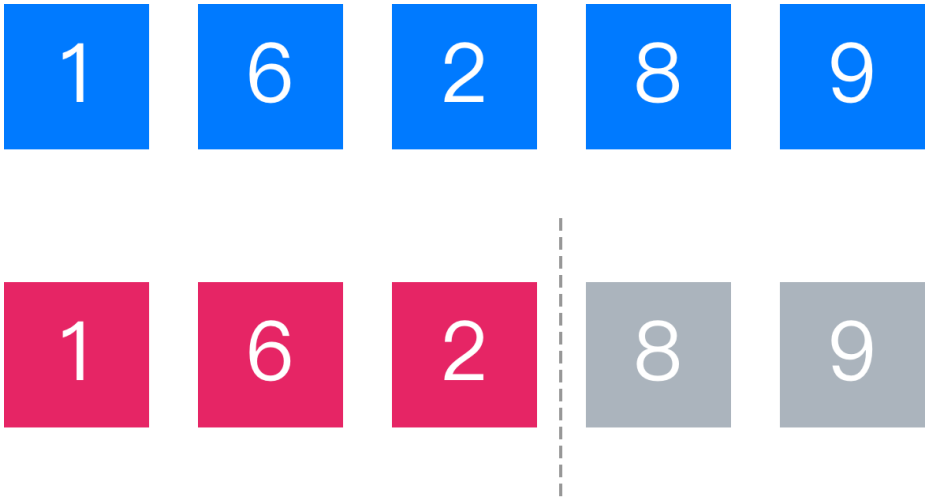
第三，当其中的一摞数据已经拿光之后，由于之前两摞数据各自都已经是排序好的，因此，我们只要把剩余的一摞数据直接添加在结果最后，就好了：



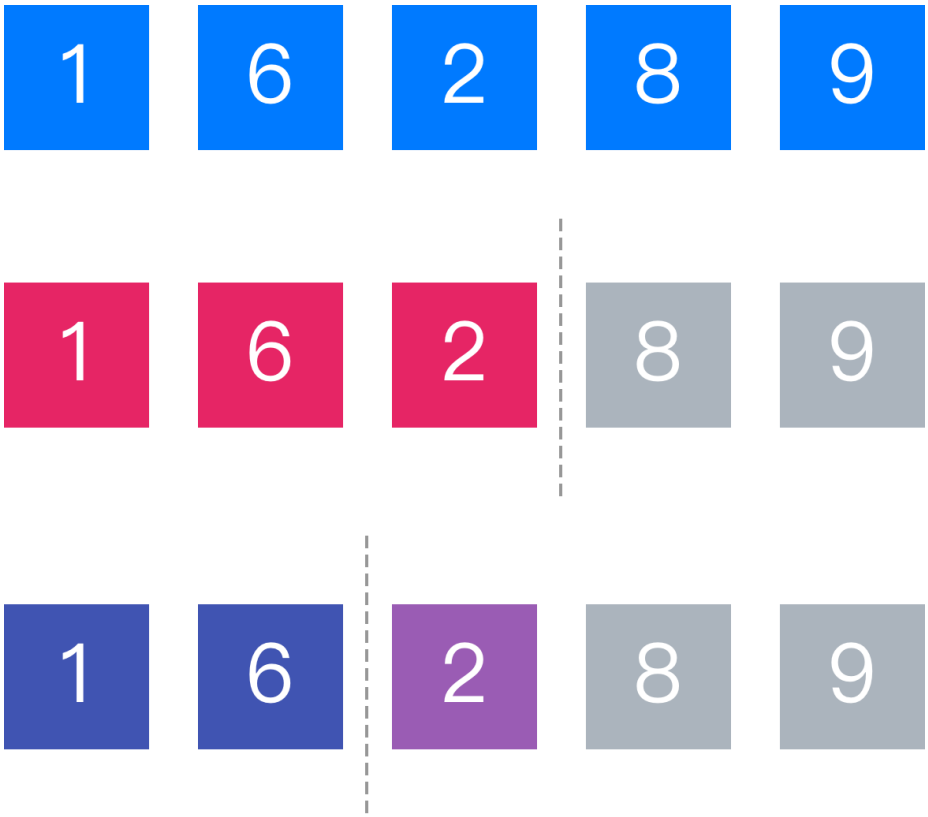
这样，自下而上，我们就得到了期望的结果。这就是merge sort中merge部分的含义。

除此之外，我们还有另外一部分工作。为了对任意一个数组排序，我们要先把它变成两摞已经排序好的数字。怎么做呢？

首先，我们从中间位置，先对数组进行一次分割。这就就得到了两摞数字，但是它们不一定都是已经排序好的：



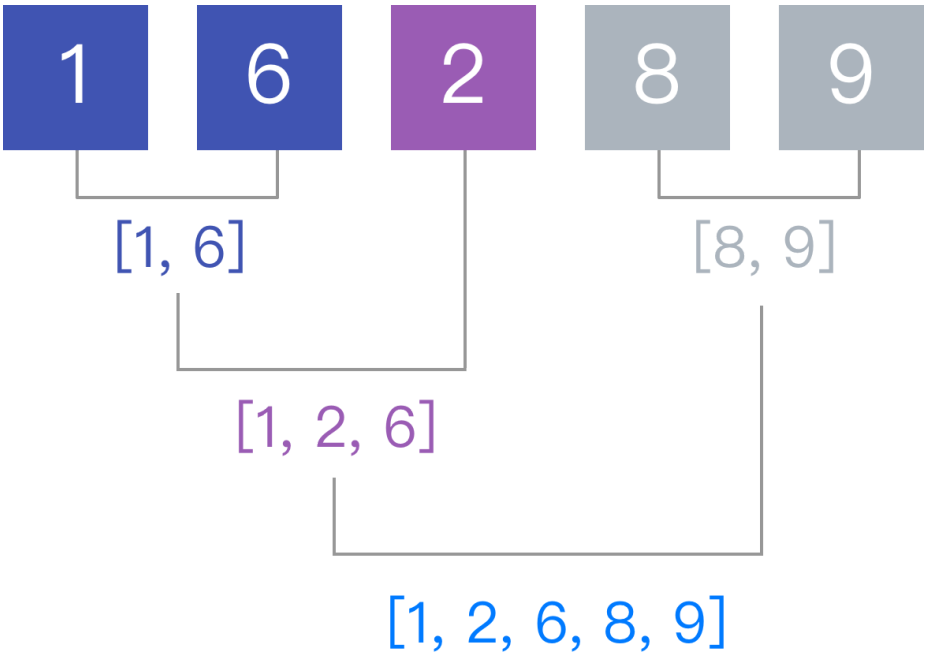
其次，在分割出来的两摞数字中，我们再进行分割：



这样，对于图中的 [1，6] 来说，我们可以把它理解为是两摞已排序好的数字了，因为一个数字是无法排序的（同理 [8，9] 也是这样）。至此，我们就不用再继续分割，可以陆续进行merge了：

1. 对 [1，6] 排序，然后把得到的结果和 [2] 排序；
2. 对 [8，9] 排序；
3. 把步骤1和步骤2的结果进行排序

这样，就完成整个归并排序的过程了。



一个有待改进的实现方法

理解了这个归并排序的设计思想之后，我们就可以动手实现它了。我们先来看一个普通的实现方式。通常归并排序的数组拆分和子数组合并是被分开实现的。我们就通过 `extension Array` 来实现这个算法。

首先，是先拆分后，再归并排序的整体过程：

```
extension Array where Element: Comparable {
    mutating func mergeSort(_ begin: Index, _ end: Index) {
        if (end - begin) > 1 {
            let mid = (begin + end) / 2

            mergeSort(begin, mid)
            mergeSort(mid, end)

            merge(begin, mid, end)
        }
    }
}
```

在上面的代码里，最核心的逻辑就是只要拆分的数组中的元素个数大于2，就计算中间位置后，继续拆分；否则，就以计算的 `mid` 为中线，对 `[begin, mid)` 和 `[mid, end)` 这个数组进行归并排序。

其次，如何来实现这个 `merge` 方法呢？按照一开始我们描述的那样，不断从两摞数据顶端取走最小的一个，取空一个之后，把剩余的另一摞直接加在结果的末尾就好了：

```
extension Array where Element: Comparable {
    private mutating func merge(_ begin: Index, _ mid: Index, _ end: Inde
x) {
        // 1. The result
        var tmp: [Element] = []

        // 2. Fetch the smaller one from the two piles
        var i = begin
        var j = mid

        while i != mid && j != end {
            if self[i] < self[j] {
                tmp.append(self[i])
                i += 1
            }
            else {
                tmp.append(self[j])
                j += 1
            }
        }

        // 3. Append the remaining
        tmp.append(contentsOf: self[i ..< mid])
        tmp.append(contentsOf: self[j ..< end])

        // 4. Update self
        replaceSubrange(begin..<end, with: tmp)
    }
}
```

因为 merge 属于 mergeSort 的执行细节，因此我们把它定义为了 private 方法。如果你理解了整个算法的思想，上面的代码并不会有任何难度。实现了这两个方法之后，我们就可以用下面的代码试一下：

```
var numbers = [1, 2, 6, 9, 8]
numbers.mergeSort(numbers.startIndex, numbers.endIndex)
```

如果一切正常，你就能看到排序后的结果： [1, 2, 6, 8, 9] 了。

通过捕获局部变量共享资源

上面这个实现虽然可以正常工作，但是仍有改进的空间。仔细分析 merge 的执行过程就会发现，每次递归调用时用于保存局部排序结果的 tmp 都是重新在调用栈中申请的，这其实是一个没必要的行为。由于最多情况下， tmp 也就容纳原始数组的所有元素，因此，我们可以利用local function捕获变量的特性，为每一次 merge 的执行提供一个共享的存储空间：

```
mutating func mergeSort(_ begin: Index, _ end: Index) {
    // A shared storage across all recursive calls
    var tmp: [Element] = []
    tmp.reserveCapacity(count)

    func merge(_ begin: Int, _ mid: Int, _ end: Int) {
        // 1. Use the same shared storage
        tmp.removeAll(keepingCapacity: true)

        // The same as before
        var i = begin
        var j = mid
        // ...
        // Omit for simplicity
    }

    if ((end - begin) > 1) {
        let mid = (begin + end) / 2

        mergeSort(begin, mid)
        mergeSort(mid, end)

        merge(begin, mid, end)
    }
}
```

这次，我们在 `mergeSort` 创建了一个可以包含原始数组所有元素的临时变量，并 `merge` 定义为了 `mergeSort` 的local function。在 `merge` 的实现里，我们捕获了临时变量 `tmp` ，并在每次使用前，把它清空。这样，就不用在每次迭代的时候，重新创建和销毁一个 `Array` 了。

并且，由于 `tmp` 仍旧是一个局部变量，当我们对不同的 `Array` 排序时，每一个 `mergeSort` 调用，都会有一个自己的 `tmp` 变量，也不会在多次使用 `mergeSort` 的时候带来冲突。

除此之外，local function还有一个副作用，就是它更深层次的隐藏了 `merge` 方法的实现，我们甚至都不用把它定义为 `private` ，它完全是 `mergeSort` 的私人财产。

What's next?

以上，就是除了共享状态之外，通过local function捕获变量的特性，在多次调用之间共享资源的用法。当然，是否值得这样使用，你还是要在性能，代码简洁性和可维护性上取得一个平衡，无论如何，作为一个有效的提升性能的途径，你应该知道捕获变量的这种用法。在下一节中，我们将讨论一个问题，为什么Objective-C中的运行时特性，在Swift中用起来并不那么得心应手。

◀ 函数和Closure真的是不同的类型么？
(https://www.boxueio.com/series/functions-and-closure/ebook/150)

OC中水土不服的运行时特性 ▶
(https://www.boxueio.com/series/functions-and-closure/ebook/152)