OneV's Den

上善若水，人淡如菊

2016-02-25 • 能工巧匠集

Swift 性能探索和优化分析

本文首发在 CSDN《程序员》杂志，订阅地址 http://dingyue.programmer.com.cn/。

Apple 在推出 Swift 时就将其冠以先进，安全和高效的新一代编程语言之名。前两点在 Swift 的语法和语言特性中已经表现得淋漓尽致：像是尾随闭包，枚举关联值，可选值和强制的类型安全等都是 Swift 显而易见的优点。但是对于高效一点，就没有那么明显了。在 2014 年 WWDC 大会上 Apple 宣称 Swift 具有超越 Objective-C 的性能，甚至某些情况下可以媲美和超过 C。但是在 Swift 正式发布后，很多开发者发现似乎 Swift 性能并没有像宣传的那样优秀。甚至在 Swift 经过了一年半的演进的今天，稍有不慎就容易掉进语言性能的陷阱中。本文将分析一些使用 Swift 进行 iOS/OS X 开发时性能上的考量和做法，同时，笔者结合自己这一年多来使用 Swift 进行开发的经验，也给出了一些对应办法。

为什么 Swift 的性能值得期待

Swift 具有一门高效语言所需要具备的绝大部分特点。与 Ruby 或者 Python 这样的解释型语言不需要再做什么对比了，相较于其前辈的 Objective-C，Swift 在编译期间就完成了方法的绑定，因此方法调用上不再是类似于 Smalltalk 的消息发送，而是直接获取方法地址并进行调用。虽然 Objective-C 对运行时查找方法的过程进行了缓存和大量的优化，但是不可否认 Swift 的调用方式会更加迅速和高效。

另外，与 Objective-C 不同，Swift 是一门强类型的语言，这意味 Swift 的运行时和代码编译期间的类型是一致的，这样编译器可以得到足够的信息来在生成中间码和机器码时进行优化。虽然都使用 LLVM 工具链进行编译，但是 Swift 的编译过程相比于 Objective-C 要多一个环节 - 生成 Swift 中间代码 (Swift Intermediate Language，SIL)。SIL 中包含有很多根据类型假定的转换，这为之后进一步在更低层级优化提供了良好的基础，分析 SIL 也是我们探索 Swift 性能的有效方法。

最后，Swift 具有良好的内存使用的策略和结构。Swift 标准库中绝大部分类型都是 `struct`，对值类型的使用范围之广，在近期的编程语言中可谓首屈一指。原本值类型不可变性的特点，往往导致对于值的使用和修改意味着创建新的对象，但是 Swift 巧妙地规避了不必要的值类型复制，而仅只在必要时进行内存分配。这使得 Swift 在享受不可变性带来的便利以及避免不必要的共享状态的同时，还能够保持性能上的优秀。

对性能进行测试

《计算机程序设计艺术》和 TeX 的作者高德纳曾经在论文中说过：

和很多人理解的不同，这并不是说我们不应该在项目的早期就开始进行优化，而是指我们需要弄清代码中性能真正的问题和希望达到的目标后再开始进行优化。因此，我们需要知道性能问题到底出在哪儿。对程序性能的测试一定是优化的第一步。

在 Cocoa 开发中，对于性能的测试有几种常见的方式。其中最简单是直接通过输出 log 来监测某一段程序运行所消耗的时间。在 Cocoa 中我们可以使用 `CACurrentMediaTime` 来获取精确的时间。这个方法将会调用 mach 底层的 `mach_absolute_time()`，它的返回是一个基于 Mach absolute time unit 的数字，我们通过在方法调用前后分别获取两次时刻，并计算它们的间隔，就可以了解方法的执行时间：

```
let start = CACurrentMediaTime()

// ...

let end = CACurrentMediaTime()

print("测量时间：\(end - start)")
```

为了方便使用，我们还可以将这段代码封装到一个方法中，这样我们就能在项目中需要测试性能的地方方便地使用它了：

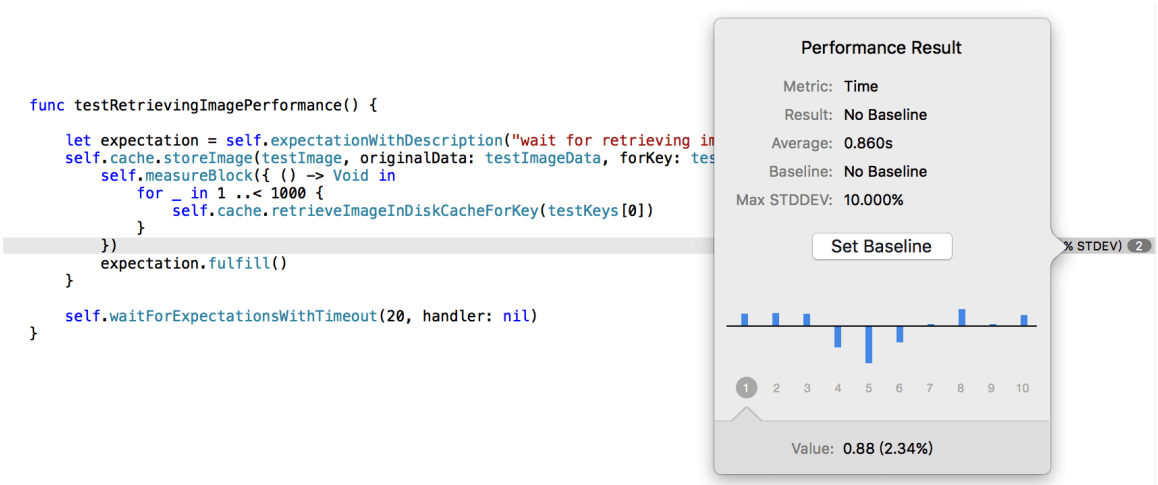
```
func measure(f: ()->()) {
    let start = CACurrentMediaTime()
    f()
    let end = CACurrentMediaTime()
    print("测量时间：\(end - start)")
}

measure {
    doSomeHeavyWork()
}
```

`CACurrentMediaTime` 和 log 的方法适合于我们对既有代码进行探索，另一种有效的方法是使用 Instruments 的 Time Profiler 来在更高层面寻找代码的性能弱点。将程序挂载到 Time Profiler 后，每一个方法调用的耗时都将被记录。

当我们寻找到需要进行优化的代码路径后，为其建立一个单元测试来持续地检测代码的性能是很好的做法。在 Xcode 中默认的测试框架 XCTest 提供了检测并汇报性能的方法：`measureBlock`。通过将测试的代码块放到 `measureBlock` 中，Xcode 在测试时就会多次运行这段代码，并统计平均耗时。更方便的是，你可以设定一个基准，Xcode 会记录每次的耗时并在性能没有达到预期时进行提醒。这保证了随着项目开发，关键的代码路径不会发生性能上的退化。

```
func testPerformance() {
    measureBlock() {
        // 需要性能测试的代码
    }
}
```



多线程、算法及数据结构优化

在确定了需要进行性能改善的代码后，一个最根本的优化方式是在程序设计层面进行改良。在移动客户端，对于影响了 UI 流畅度的代码，我们可以将其放到后台线程进行运行。Grand Central Dispatch (GCD) 或者 `NSOperation` 可以让我们方便地在不同线程中切换，而不太需要去担心线程调度的问题。一个使用 GCD 将繁重工作放到后台线程，然后在完成后回到主线程操作 UI 的典型例子是这样的：

```
let queue = dispatch_get_global_queue(QOS_CLASS_DEFAULT, 0)
    dispatch_async(queue) {

        // 运行时间较长的代码，放到后台线程运行

        dispatch_async(dispatch_get_main_queue()) {
            // 结束后返回主线程操作 UI
        }
}
```

将工作放到其他线程虽然可以避免主线程阻塞，但它并不能减少这些代码实际的执行时间。进一步地，我们可以考虑改进算法和使用的数据结构来提高效率。根据实际项目中遇到的问题的不同，我们会有不同的解决方式，在这篇文章中，我们难以覆盖和深入去分析各种情况，所以这里我们只会提及一些共通的原则。

对于重复的工作，合理地利用缓存的方式可以极大提高效率，这是在优化时可以优先考虑的方式。Cocoa 开发中 `NSCache` 是专门用来管理缓存的一个类，合理地使用和配置 `NSCache` 把开发者中从管理缓存存储和失效的工作中解放出来。关于 `NSCache` 的详细使用方法，可以参看 NSHipster 关于这方面的文章以及 Apple 的相关文档。

在程序开发时，数据结构使用上的选择也是重要的一环。Swift 标准库提供了一些很基本的数据结构，比如 `Array`、`Dictionary` 和 `Set` 等。这些数据结构都是配合泛型的，在保证数据类型安全的同时，一般来说也能为我们提供足够的性能。关于这些数据的容器类型方法所对应的复杂度，Apple 都在标准库的文档或者注释中进行了标记。如果标准库所提供的类型和方法无法满足性能上的要求，或者没有符合业务需求的数据结构的话，那么考虑使用自己实现的数据结构也是可选项。

如果项目中有很多数学计算方面的工作导致了效率问题的话，考虑并行计算能极大改善程序性能。iOS 和 OS X 都有针对数学或者图形计算等数字信号处理方面进行了专门优化的框架：Accelerate.framework，利用相关的 API，我们可以轻松快速地完成很多经典的数字或者图像处理问题。因为这个框架只提供一组 C API，所以在 Swift 中直接使用会有一定困难。如果你的项目中要处理的计算相对简单的话，也可以使用 Surge，它是一个基于 Accelerate 框架的 Swift 项目，让我们能在代码里从并行计算中获得难以置信的性能提升。

编译器优化

Swift 编译器十分智能，它能在编译期间帮助我们移除不需要的代码，或者将某些方法进行内联 (inline) 处理。编译器优化的强度可以在编译时通过参数进行控制，Xcode 工程默认情况下有 Debug 和 Release 两种编译配置，在 Debug 模式下，LLVM Code Generation 和 Swift Code Generation 都不开启优化，这能保证编译速度。而在 Release 模式下，LLVM 默认使用 “Fastest, Smallest [-Os]”，Swift Compiler 默认使用 “Fast [-O]”，作为优化级别。我们另外还有几个额外的优化级别可以选择，优化级别越高，编译器对于源码的改动幅度和开启的优化力度也就越大，同时编译期间消耗的时间也就越多。虽然绝大部分情况下没有问题，但是仍然需要当心的是，一些优化等级采用的是激进的优化策略，而禁用了一些检查。这可能在源码很复杂的情况下导致潜在的错误。如果你使用了很高的优化级别，请再三测试 Release 和 Debug 条件下程序运行的逻辑，以防止编译器优化所带来的问题。

值得一提的是，Swift 编译器有一个很有用的优化等级：”Fast, Whole Module Optimization”，也即 `-O -whole-module-optimization`。在这个优化等级下，Swift 编译器将会同时考虑整个 module 中所有源码的情况，并将那些没有被继承和重载的类型和方法标记为 `final`，这将尽可能地避免动态派发的调用，或者甚至将方法进行内联处理以加速运行。开启这个额外的优化将会大幅增加编译时间，所以应该只在应用要发布的时候打开这个选项。

虽然现在编译器在进行优化的时候已经足够智能了，但是在面对编写得非常复杂的情况时，很多本应实施的优化可能失效。因此保持代码的整洁、干净和简单，可以让编译器优化良好工作，以得到高效的机器码。

尽量使用 Swift 类型

`Swift.Array` 与 `NSArray`，`Swift.String` 和 `NSString` 等。虽然我们不需要在语言层面做类型转换，但是这个过程却不是免费的。在转换次数很多的时候，这往往会成为性能的瓶颈。一个常见的 Swift 和 Objective-C 混用的例子是 JSON 解析。考虑以下代码：

```
let jsonData: NSData = //...
let jsonObject = try? NSJSONSerialization
                .JSONObjectWithData(jsonData, options: []) as? [String: AnyObject]
```

这是我们日常开发中很常见的代码，使用 `NSJSONSerialization` 将数据转换为 JSON 对象后，我们得到的是一个 NSObject 对象。在 Swift 中使用时，我们一般会先将其转换为 `[String: AnyObject]`，这个转换在一次性处理成千上万条 JSON 数据时会带来严重的性能退化。Swift 3 中我们可能可以基于 Swift 的 Foundation 框架来解决这个问题，但是现在，如果存在这样的情况，一种处理方式是避免使用 Swift 的字典类型，而使用 `NSDictionary`。另外，适当地使用 lazy 加载的方法，也是避免一次性进行过多的类型转换的好思路。

尽可能避免混合地使用 Swift 类型和 `NSObject` 子类，会对性能的提高有所帮助。

避免无意义的 log，保持好的编码习惯

在调试程序时，很多开发者喜欢用输出 log 的方式对代码的运行进行追踪，帮助理解。Swift 编译器并不会帮我们将 `print` 或者 `debugPrint` 删去，在最终 app 中它们会把内容输出到终端，造成性能的损失。我们当然可以在发布时用查找的方式将所有这些 log 输出语句删除或者注释掉，但是更好的方法是通过添加条件编译来将这些语句排除在 Release 版本外。在 Xcode 的 Build Setting 中，在 **Other Swift flags** 的 Debug 栏中加入 `-D DEBUG` 即可加入一个编译标识。

之后我们就可以通过将 `print` 或者 `debugPrint` 包装一下：

```
func dPrint(item: Any) {
    #if DEBUG
    print(item)
    #endif
}
```

这样，在 Release 版本中，`dPrint` 将会是一个空方法，所有对这个方法的调用都会被编译器剔除掉。需要注意的是，在这种封装下，如果你传入的 `items` 是一个表达式而不是直接的变量的话，这个表达式还是会被先执行求值的。如果这对性能也产生了可测的影响的话，我们最好用 `@autoclosure` 修饰参数来重新包装 `print`。这可以将求值运行推迟到方法内部，这样在 Release 时这个求值也会被一并去掉：

```
func dPrint(@autoclosure item: () -> Any) {
    #if DEBUG
    print(item())
    #endif
}

dPrint(resultFromHeavyWork())
// Release 版本中 resultFromHeavyWork() 不会被执行
```

小结

Swift 还是一门很新的语言，并且处于高速发展中。因为现在 Swift 只用于 Cocoa 开发，因此它和 Cocoa 框架还有着千丝万缕的联系。很多时候由于这些原因，我们对于 Swift 性能的评估并不公正。这门语言本身设计就是以高性能为考

译器的支持。

最好的优化就是不用优化。在软件开发中，保证书写正确简洁的代码，在项目开始阶段就注意可能存在的性能缺陷，将可扩展性的考虑纳入软件构建中，按照实际需求进行优化，不要陷入为了优化而优化的怪圈，这些往往都可以让我们避免额外的优化时间，让我们的工作得更加愉快。

参考

- Swift Intermediate Language
- NSCache - NSHipster
- NSCache 文档
- Surge

最近的文章
Swift 2 throws 全解析 - 从原理到实践
本文最初于 2015 年 12 月发布在 IBM developerWorks 中国网站发表，其网址是 http://www.ibm.com/developerworks/cn/mobile/mo-cn-swift/index.html。如需转载请保留此行声明。Swift 2 错误处理简介throws 关键字和异常处理机制是 Swift 2 中新加入的重要特性。Apple 希望通过在语言层面对异常处理的流程进行规范和统一，来让代码更加安全，同时让开发者可以更加及时可靠地处理这些错误。Swi……
2016-03-29 • 能工巧匠集 继续阅读

更早的文章
如何打造一个让人愉快的框架
这是我在今年 1 月 10 日 @Swift 开发者大会 上演讲的文字稿。相关的视频还在制作中，没有到现场的朋友可以通过这个文字稿了解到这个 session 的内容。
虽然我的工作是程序员，但是最近半年其实我的主要干的事儿是养了一个小孩。所以这半年来可以说没有积累到什么技术，反而是积累了不少养小孩的心得。当知道了有这么次会议可以分享这半年来的心得的时候，我毫不犹豫地选定了主题。那就是 如何打造一个让人愉快的小孩但考虑到这是一次开发者会议…当我把这个想法和题目提交给大……
2016-01-19 • 能工巧匠集 继续阅读

回复 · 分享 ›

My Soul, Your Beats! · 2年前
喵神，我开发了一个有关导航自定义转场的库: https://github.com/Soul-Bea...
回复 · 分享 ›

My Soul, Your Beats! ➚ My Soul, Your Beats! · 2年前
https://github.com/Soul-Bea...
回复 · 分享 ›

Jet Lee · 2年前
喵神你好，我请教一个问题，assign 修饰基本数据类型，这个都是这样用的，assign 在非 ARC 的情况下，是相当于 unsafe_unretained，也就是一个跟 weak 差不多的东西，这样应该是会导致野指针吧。网上搜到一条相关的说，基础数据类型一般分配在栈上，栈的内存会由系统自己自动处理，不会造成野指针。但是再去搜 stack objects in objective-c 的时候，基本搜不到靠谱的信息了，有一个比较靠谱的是 10 年的文档，说只有一种，那就是 block。问得有点凌乱了，其实问题就是为什么基本数据类型不需要强引用
回复 · 分享 ›

onevcat 管理员 ➚ Jet Lee · 2年前
因为值类型 (或者说你指的基本数据类型) 并不是一个 (指向某个堆地址的) 指针，而是直接含有值的一段内存，自然也就不存在野指针。关于 stack object 的话确实 blcok 是，可以看看 Mike Ash 的这篇 Friday FAQ https://mikeash.com/pyblog/...
回复 · 分享 ›

Akring · 2年前
只是想问一下，为什么要从Ghost转到Jekyll呢？Jekyll有什么明显优势？
回复 · 分享 ›

onevcat 管理员 ➚ Akring · 2年前
其实 Ghost 也还不错的，至少配合个 forever 也没遇到挂掉起不来或者说慢之类的问题。最早我是用 WordPress 的，然后换成 Octopress 静态生成，再然后换到 Ghost，现在又回到静态，纯粹是瞎折腾。不过好处是也算有个机会多接触下 Ruby 什么的，有时候也挺方便的。

要说换回静态的原因，基本上就是相比 Ghost 来说自己想要做扩展的话容易些，可以很方便地自己写 Generator，然后最近的写作模式比较符合静态博客的生成方式。
回复 · 分享 ›

Jet Lee ➚ Akring · 2年前
Ghost 吃内存，同时还依赖一个守护进程的进程，不如纯静态的稳定啊~只要 Nginx 不挂网站就不挂
回复 · 分享 ›

极分享 · 2年前
非常不错哦！劳逸结合
1 · 回复 · 分享 ›

极分享 · 2年前
简直厉害啊 .....
1 · 回复 · 分享 ›

Yan Zhang · 2年前
JSON解析转换Dictionary的那个去年我们项目开发中就遇到了 最后换成了NSDictionary 要不一个request处理起来好几秒。。。
回复 · 分享 ›

John Zuo · 2年前
猫神你好，我用swift开发了一个中等规模的项目，现在发现最后打包出来后，ipa本身有 22m,然后解开包的话，发现里面的unit 执行文件有24m左右，这个是不是有点太大了，尝试了几种编译设置，并没有任何的减少，不知道您有没有发现这个问题?
回复 · 分享 ›

Xin · 2年前
赞一个，学到很多
回复 · 分享 ›

回复 · 分享 ›

Ben · 2年前
最好的优化就是不用优化。说的很有道理啊！
回复 · 分享 ›

lee · 2年前
Dictionary性能很差，Array的使用需要技巧，这些方面可以展开研究
回复 · 分享 ›

WildCat · 2年前
每次看瞄神的文章都受益匪浅，感谢！也祝家里的小朋友天天开心！
回复 · 分享 ›

Sheep · 2年前
一直在关注你，我是一名来自硅谷的程序媛，业余自学写app，从你的书里面学到了很多东西。希望能多看到你的文章：）加油
回复 · 分享 ›

onevcat 管理员 ➚ Sheep · 2年前
感谢关注，多交流 :)
回复 · 分享 ›

axl411 · 2年前
标题和配图简直可以出同人本～
回复 · 分享 ›

Jet Lee · 2年前
Other Swift flags 的 Debug 栏中加入 -D DEBUG，和 Xcode 本身就有的 LLVM - Preprocessing -> DEBUG=1，是不是不一样的用途
回复 · 分享 ›

onevcat 管理员 ➚ Jet Lee · 2年前
不一样。LLVM 的 Preprocessing 是在处理像是 Objective-C 或者 C 代码时预编译阶段进行替换的，其本质就是一个写在配置文件 (build setting) 里的 “define”。
而 Swift 的编译过程是没有预处理这个阶段的，这里的设置对 Swift 代码是无效的。
回复 · 分享 ›

Jet Lee ➚ onevcat · 2年前
受教了 :)
回复 · 分享 ›

nixzhu · 2年前
再修正及自我修正：Knuth的中文名应为“高德纳”
回复 · 分享 ›

onevcat 管理员 ➚ nixzhu · 2年前
你要是刷新一下的话会发现我已经一起偷偷修好了 ^O^
回复 · 分享 ›

codeEagle · 2年前
请教喵神，如果把工程里面的功能做成framework，这样再embed到工程里面，是否会有利于编译?
回复 · 分享 ›

onevcat 管理员 ➚ codeEagle · 2年前
那当然，编译的时候这部分就不需要再编译了，已经是二进制。
回复 · 分享 ›

nixzhu · 2年前
修正：高纳德做的是TeX
回复 · 分享 ›

onevcat 管理员 ➚ nixzhu · 2年前
なるほど
回复 · 分享 ›

crossPQW · 2年前

runnphoenix • 2年前

赞题图

回复 • 分享 ›

在 ONEV'S DEN 上还有

Swift 2 throws 全解析 - 从原理到实践

4条评论 • 2年前

Lincoln — curried function 的写法 已经在swift 2.2被废弃了

OpenCV 在 iOS 开发环境下的编译和配置

1条评论 • 1年前

ming xiao —

写在 2015 的尾巴

68条评论 • 2年前

十年 言 — 榜样

猫都能学会的Unity3D Shader入门指南（一）

1条评论 • 2年前

lieb li — 感谢分享知识，一直有一个问题所以很少用shader，需求就是在ui 上用shader， 问题是mask挡不住shader，请教楼主能解决么?