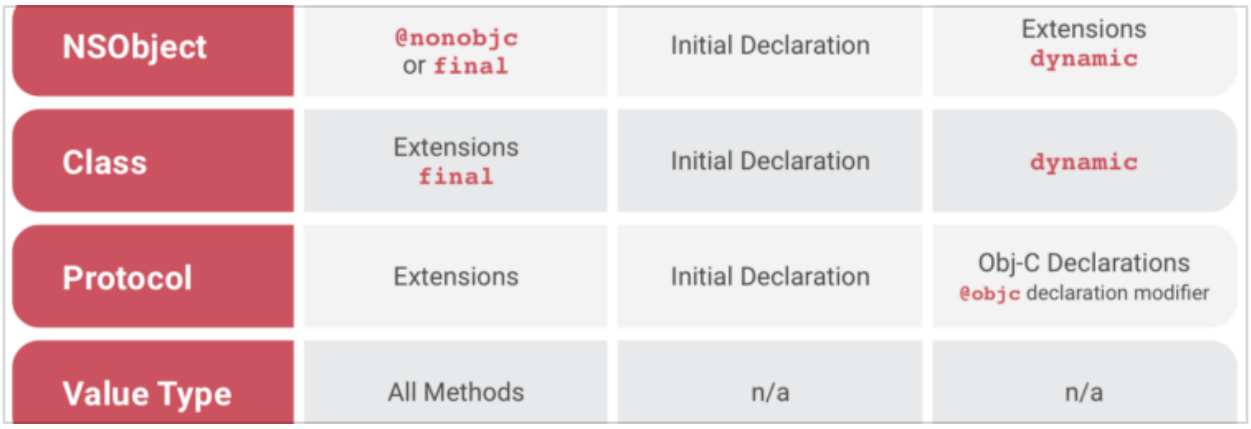
深入理解 Swift 派发机制

#Blog

译者注:
之前看了很多关于 Swift 派发机制的内容, 但感觉没有一篇能够彻底讲清楚这件事情, 看完了这篇文章之后我对 Swift 的派发机制才建立起了初步的认知.

正文

NSObject	`@nonobjc` or `final`	Initial Declaration	Extensions `dynamic`
Class	Extensions `final`	Initial Declaration	`dynamic`
Protocol	Extensions	Initial Declaration	Obj-C Declarations `@objc` declaration modifier
Value Type	All Methods	n/a	n/a



一张表总结引用类型, 修饰符和它们对于 Swift 函数派发方式的影响.

函数派发就是程序判断使用哪种途径去调用一个函数的机制. 每次函数被调用时都会被触发, 但你又不会太留意的一个东西. 了解派发机制对于写出高性能的代码来说很有必要, 而且也能够解释很多 Swift 里"奇怪"的行为.

编译型语言有三种基础的函数派发方式: **直接派发(Direct Dispatch), 函数表派发(Table Dispatch) 和 消息机制派发(Message Dispatch)**, 下面我会仔细讲解这几种方式. 大多数语言都会支持一到两种, Java 默认使用函数表派发, 但你可以通过 `final` 修饰符修改成直接派发. C++ 默认使用直接派发, 但可以通过加上 `virtual` 修饰符来改成函数表派发. 而 Objective-C 则总是使用消息机制派发, 但允许开发者使用 C 直接派发来获取性能的提高. 这样的方式非常好, 但也给很多开发者带来了困扰。

> 译者注: 想要了解 Swift 底层结构的人, 极度推荐这段视频

派发方式 (Types of Dispatch)

程序派发的目的是为了告诉 CPU 需要被调用的函数在哪里, 在我们深入 Swift 派发机制之前, 先来了解一下这三种派发方式, 以及每种方式在动态性和性能之间的取舍.

直接派发 (Direct Dispatch)

直接派发是最快的, 不止是因为需要调用的指令集会更少, 并且编译器还能够有很大的优化空间, 例如函数内联等, 但这不在这篇博客的讨论范围. 直接派发也有人称为静态调用.

然而, 对于编程来说直接调用也是最大的局限, 而且因为缺乏动态性所以没办法支持继承.

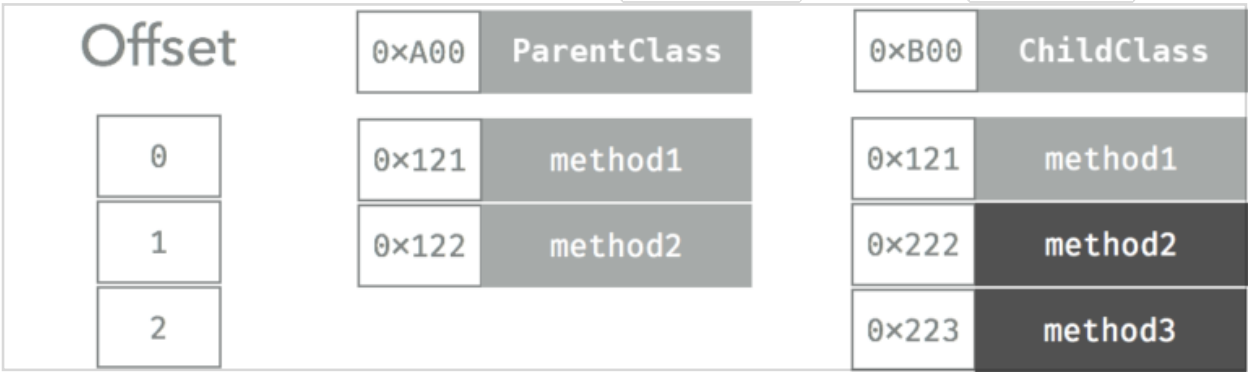
函数表派发 (Table Dispatch)

函数表派发是编译型语言实现动态行为最常见的实现方式. 函数表使用了一个数组来存储类声明的每一个函数的指针. 大部分语言把这个称为 "virtual table"(虚函数表), Swift 里称为 "witness table". 每一个类都会维护一个函数表, 里面记录着类所有的函数, 如果父类函数被 override 的话, 表里面只会保存被 override 之后的函数. 一个子类新添加的函数, 都会被插入到这个数组的最后. 运行时会根据这一个表去决定实际要被调用的函数.

举个例子, 看看下面两个类:

```
class ParentClass {
    func method1() {}
    func method2() {}
}
class ChildClass: ParentClass {
    override func method2() {}
    func method3() {}
}
```

在这个情况下, 编译器会创建两个函数表, 一个是 `ParentClass` 的, 另一个是 `ChildClass` 的:



这张表展示了 ParentClass 和 ChildClass 虚数表里 method1, method2, method3 在内存里的布局.

```
let obj = ChildClass()
```

```
obj.method2()
```

当一个函数被调用时, 会经历下面的几个过程:

1. 读取对象 `0xB00` 的函数表.
2. 读取函数指针的索引. 在这里, `method2` 的索引是1(偏移量), 也就是 `0xB00 + 1`
3. 跳到 `0x222` (函数指针指向 0x222)

查表是一种简单, 易实现, 而且性能可预知的方式. 然而, 这种派发方式比起直接派发还是慢一点. 从字节码角度来看, 多了两次读和一次跳转, 由此带来了性能的损耗. 另一个慢的原因在于编译器可能会由于函数内执行的任务导致无法优化. (如果函数带有副作用的话)

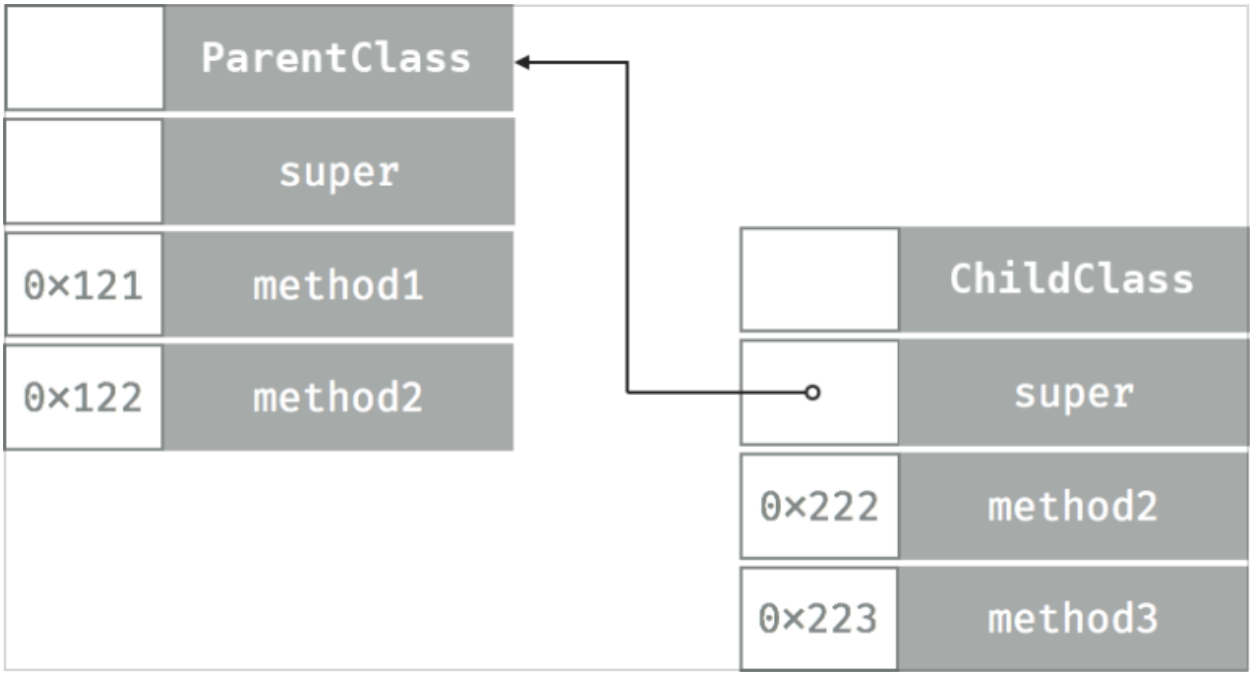
这种基于数组的实现, 缺陷在于函数表无法拓展. 子类会在虚数函数表的最后插入新的函数, 没有位置可以让 extension 安全地插入函数. 这篇提案很详细地描述了这么做的局限.

消息机制派发 (Message Dispatch)

消息机制是调用函数最动态的方式. 也是 Cocoa 的基石, 这样的机制催生了 KVO, UIAppearence 和 CoreData 等功能. 这种运作方式的关键在于开发者可以在运行时改变函数的行为. 不止可以通过 swizzling 来改变, 甚至可以用 isa-swizzling 修改对象的继承关系, 可以在面向对象的基础上实现自定义派发.
举个例子, 看看下面两个类:

```
class ParentClass {
    dynamic func method1() {}
    dynamic func method2() {}
}
class ChildClass: ParentClass {
    override func method2() {}
    dynamic func method3() {}
}
```

Swift 会用树来构建这种继承关系:



这张图很好地展示了 Swift 如何使用树来构建类和子类.

当一个消息被派发, 运行时会顺着类的继承关系向上查找应该被调用的函数. 如果你觉得这样做效率很低, 它确实很低! 然而, 只要缓存建立了起来, 这个查找过程就会通过缓存来把性能提高到和函数表派发一样快. 但这只是消息机制的原理, 这里有一篇文章很深入的讲解了具体的技术细节.

Swift 的派发机制

那么, 到底 Swift 是怎么派发的呢? 我没能找到一个很简明扼要的答案, 但这里有四个选择具体派发方式的因素存在:

1. 声明的位置
2. 引用类型
3. 特定的行为
4. 显式地优化(Visibility Optimizations)

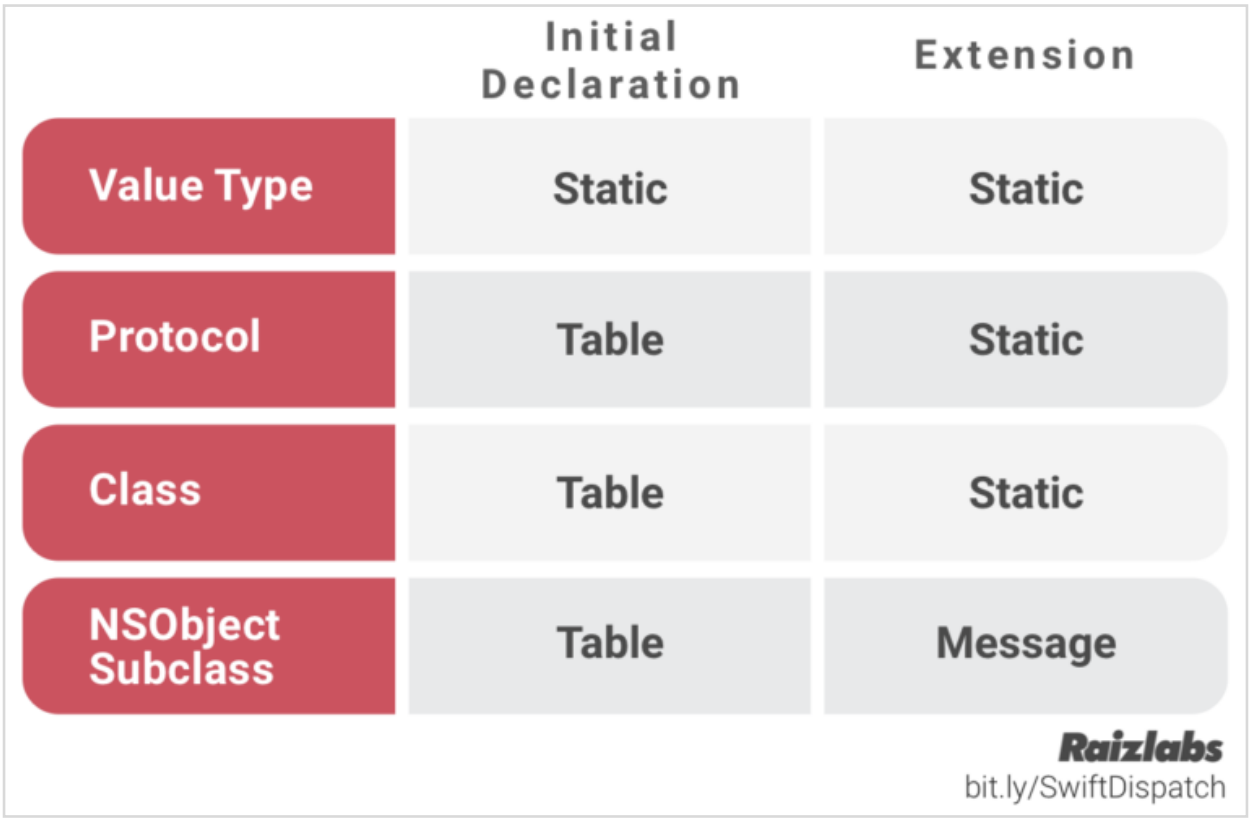
在解释这些因素之前, 我有必要说清楚, Swift 没有在文档里具体写明什么时候会使用函数表什么时候使用消息机制. 唯一的承诺是使用 `dynamic` 修饰的时候会通过 Objective-C 的运行时进行消息机制派发. 下面我写的所有东西, 都只是我在 Swift 3.0 里测试出来的结果, 并且很可能在之后的版本更新里进行修改.

声明的位置 (Location Matters)

在 Swift 里, 一个函数有两个可以声明的位置: 类型声明的作用域, 和 extension. 根据声明类型的不同, 也会有不同的派发方式.

```
class MyClass {
    func mainMethod() {}
}
extension MyClass {
    func extensionMethod() {}
}
```

上面的例子里, mainMethod 会使用函数表派发, 而 extensionMethod 则会使用直接派发. 当我第一次发现这件事情的时候觉得很意外, 直觉上这两个函数的声明方式并没有那么大的差异. 下面是我根据类型, 声明位置总结出来的函数派发方式的表格.



这张表格展示了默认情况下 Swift 使用的派发方式.

总结起来有这么几点:

- 值类型总是会使用直接派发, 简单易懂
- 而协议和类的 extension 都会使用直接派发
- NSObject 的 extension 会使用消息机制进行派发
- NSObject 声明作用域里的函数都会使用函数表进行派发.
- 协议里声明的, 并且带有默认实现的函数会使用函数表进行派发

引用类型 (Reference Type Matters)

引用的类型决定了派发的方式. 这很显而易见, 但也是决定性的差异. 一个比较常见的疑惑, 发生在一个协议拓展和类型拓展同时实现了同一个函数的时候.

```
protocol MyProtocol {
}
struct MyStruct: MyProtocol {
}
extension MyStruct {
    func extensionMethod() {
        print("结构体")
    }
}
extension MyProtocol {
    func extensionMethod() {
        print("协议")
    }
}
let myStruct = MyStruct()
let proto: MyProtocol = myStruct
myStruct.extensionMethod() // -> “结构体”
proto.extensionMethod() // -> “协议”
```

刚接触 Swift 的人可能会认为 `proto.extensionMethod()` 调用的是结构体里的实现. 但是, 引用的类型决定了派发的方式, 协议拓展里的函数会使用直接调用. 如果把 `extensionMethod` 的声明移动到协议的声明位置的话, 则会使用函数表派发, 最终就会调用结构体里的实现. 并且要记得, 如果两种声明方式都使用了直接派发的话, 基于直接派发的运作方式, 我们不可能实现预想的 `override` 行为. 这对于很多从 Objective-C 过渡过来的开发者是反直觉的.

Swift JIRA(缺陷跟踪管理系统) 也发现了几个 bugs, Swfit-Evolution 邮件列表里有一大堆讨论, 也有一大堆博客讨论过这个. 但是, 这好像是故意这么做的, 虽然官方文档没有提过这件事情.

指定派发方式 (Specifying Dispatch Behavior)

Swift 有一些修饰符可以指定派发方式.

1.final

`final` 允许类里面的函数使用直接派发. 这个修饰符会让函数失去动态性. 任何函数都可以使用这

个修饰符, 就算是 extension 里本来就是直接派发的函数. 这也会让 Objective-C 的运行时获取不到这个函数, 不会生成相应的 selector.

2.dynamic

`dynamic` 可以让类里面的函数使用消息机制派发. 使用 `dynamic`, 必须导入 `Foundation` 框架, 里面包括了 NSObject 和 Objective-C 的运行时. `dynamic` 可以让声明在 extension 里面的函数能够被 `override`. `dynamic` 可以用在所有 NSObject 的子类和 Swift 的原声类.

3.@objc & @nonobjc

`@objc` 和 `@nonobjc` 显式地声明了一个函数是否能被 Objective-C 的运行时捕获到. 使用 @objc 的典型例子就是给 selector 一个命名空间 `@objc(abc_methodName)`, 让这个函数可以被 Objective-C 的运行时调用. `@nonobjc` 会改变派发的方式, 可以用来禁止消息机制派发这个函数, 不让这个函数注册到 Objective-C 的运行时里. 我不确定这跟 `final` 有什么区别, 因为从使用场景来说也几乎一样. 我个人来说更喜欢 `final`, 因为意图更加明显.

> 译者注: 我个人感觉, 这这主要是为了跟 Objective-C 兼容用的, `final` 等原生关键词, 是让 Swift 写服务端之类的代码的时候可以有原生的关键词可以使用.

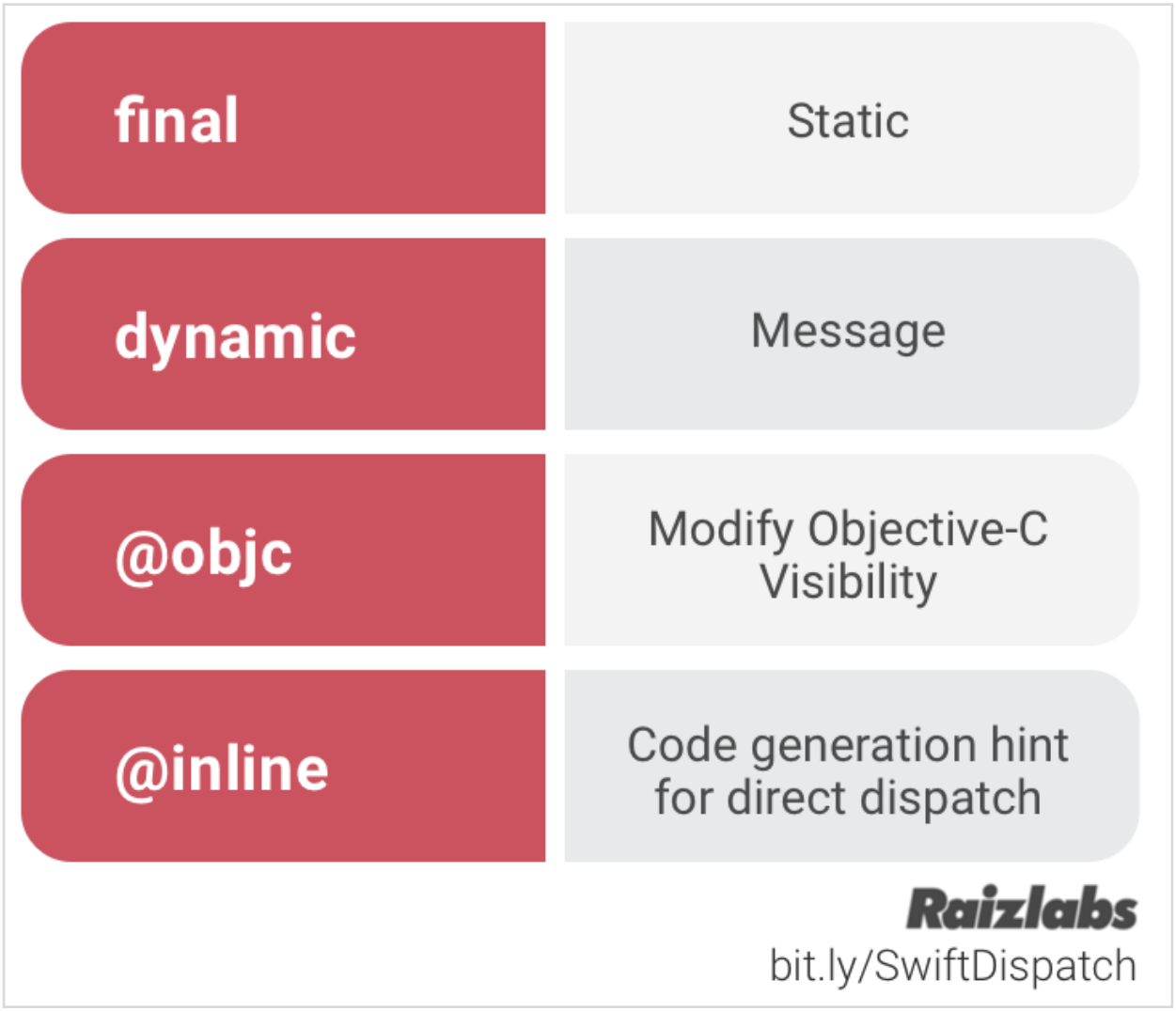
4.final @objc

可以在标记为 `final` 的同时, 也使用 `@objc` 来让函数可以使用消息机制派发. 这么做的结果就是, 调用函数的时候会使用直接派发, 但也会在 Objective-C 的运行时里注册响应的 selector. 函数可以响应 `perform(selector:)` 以及别的 Objective-C 特性, 但在直接调用时又可以有直接派发的性能.

5.@inline

Swift 也支持 `@inline`, 告诉编译器可以使用直接派发. 有趣的是, `dynamic @inline(__always) func dynamicOrDirect() {}` 也可以通过编译! 但这也只是告诉了编译器而已, 实际上这个函数还是会使用消息机制派发. 这样的写法看起来像是一个未定义的行为, 应该避免这么做.

修饰符总结 (Modifier Overview)



这张图总结这些修饰符对于 Swift 派发方式的影响.

如果你想查看上面所有例子的话, 请看这里.

可见的都会被优化 (Visibility Will Optimize)

Swift 会尽最大能力去优化函数派发的方式. 例如, 如果你有一个函数从来没有 override, Swift 就会检车并且在可能的情况下使用直接派发. 这个优化大多数情况下都表现得很好, 但对于使用了 target / action 模式的 Cocoa 开发者就不那么友好了. 例如:

```
override func viewDidLoad() {
    super.viewDidLoad()
    navigationItem.rightBarButtonItem = UIBarButtonItem(
        title: "登录", style: .plain, target: nil,
        action: #selector(ViewController.signInAction)
```

```
    )
}
private func signInAction() {}
```

这里编译器会抛出一个错误: `Argument of '#selector' refers to a method that is not exposed to Objective-C (Objective-C 无法获取 #selector 指定的函数)`. 你如果记得 Swift 会把这个函数优化为直接派发的话, 就能理解这件事情了. 这里修复的方式很简单: 加上 `@objc` 或者 `dynamic` 就可以保证 Objective-C 的运行时可以获取到函数了. 这种类型的错误也会发生在 `UIAppearance` 上, 依赖于 proxy 和 NSInvocation 的代码.

另一个需要注意的是, 如果你没有使用 `dynamic` 修饰的话, 这个优化会默认让 KVO 失效. 如果一个属性绑定了 KVO 的话, 而这个属性的 getter 和 setter 会被优化为直接派发, 代码依旧可以通过编译, 不过动态生成的 KVO 函数就不会被触发.

Swift 的博客有一篇很赞的文章描述了相关的细节, 和这些优化背后的考虑.

派发总结 (Dispatch Summary)

这里有一大堆规则要记住, 所以我整理了一个表格:

	Direct	Table	Message
NSObject	`@nonobjc` or `final`	Initial Declaration	Extensions `dynamic`
Class	Extensions `final`	Initial Declaration	`dynamic`
Protocol	Extensions	Initial Declaration	Obj-C Declarations `@objc` declaration modifier
Value Type	All Methods	n/a	n/a

Note: Compiler optimizations may upgrade to Direct Dispatch unless **`dynamic`** is specified

Raizlabs bit.ly/SwiftDispatch

这张表总结引用类型, 修饰符和它们对于 Swift 函数派发的影响

NSObject 以及动态性的损失 (NSObject and the Loss of Dynamic Behavior)

不久之前还有一群 Cocoa 开发者讨论动态行为带来的问题. 这段讨论很有趣, 提了一大堆不同的观点. 我希望可以在这里继续探讨一下, 有几个 Swift 的派发方式我觉得损害了动态性, 顺便说一下

我的解决方案.

NSObject 的函数表派发 (Table Dispatch in NSObject)

上面, 我提到 `NSObject` 子类定义里的函数会使用函数表派发. 但我觉得很迷惑, 很难解释清楚, 并且由于下面几个原因, 这也只带来了一点点性能的提升:

- 大部分 `NSObject` 的子类都是在 `obj_msgSend` 的基础上构建的. 我很怀疑这些派发方式的优化, 实际到底会给 Cocoa 的子类带来多大的提升.
- 大多数 Swift 的 `NSObject` 子类都会使用 extension 进行拓展, 都没办法使用这种优化.

最后, 有一些小细节会让派发方式变得很复杂.

派发方式的优化破坏了 NSObject 的功能 (Dispatch Upgrades Breaking NSObject Features)

性能提升很棒, 我很喜欢 Swift 对于派发方式的优化. 但是, `UIView` 子类颜色的属性理论上性能的提升破坏了 UIKit 现有的模式.

> 原文: However, having a theoretical performance boost in my `UIView` subclass color property breaking an established pattern in UIKit is damaging to the language.

NSObject 作为一个选择 (NSObject as a Choice)

使用静态派发的话结构体是个不错的选择, 而使用消息机制派发的话则可以考虑 NSObject. 现在, 如果你想跟一个刚学 Swift 的开发者解释为什么某个东西是一个 NSObject 的子类, 你不得不去介绍 Objective-C 以及这段历史. 现在没有任何理由去继承 NSObject 构建类, 除非你需要使用 Objective-C 构建的框架.

目前, NSObject 在 Swift 里的派发方式, 一句话总结就是复杂, 跟理想还是有差距. 我比较想看到这个修改: 当你继承 NSObject 的时候, 这是一个你想要完全使用动态消息机制的表现.

显式的动态性声明 (Implicit Dynamic Modification)

另一个 Swift 可以改进的地方就是函数动态性的检测. 我觉得在检测到一个函数被 `#selector` 和 `#keypath` 引用时要自动把这些函数标记为 dynamic, 这样的话就会解决大部分 UIAppearance 的动态问题, 但也许有别的编译时的处理方式可以标记这些函数.

Error 以及 Bug (Errors and Bugs)

为了让我们对 Swift 的派发方式有更多了解, 让我们来看一下 Swift 开发者遇到过的 error.

SR-584

这个 Swift bug 是 Swift 函数派发的一个功能. 存在于 NSObject 子类声明的函数(函数表派发), 以

及声明在 extension 的函数(消息机制派发)中. 为了更好地描述这个情况, 我们先来创建一个类:

```
class Person: NSObject {
    func sayHi() {
        print("Hello")
    }
}
func greetings(person: Person) {
    person.sayHi()
}
greetings(person: Person()) // prints 'Hello'
```

`greetings(person:)` 函数使用函数表派发来调用 `sayHi()`. 就像我们看到的, 期望的, "Hello" 会被打印. 没什么好讲的地方, 那现在让我们继承 `Persion`:

```
class MisunderstoodPerson: Person {}
extension MisunderstoodPerson {
    override func sayHi() {
        print("No one gets me.")
    }
}
greetings(person: MisunderstoodPerson()) // prints 'Hello'
```

可以看到, `sayHi()` 函数是在 extension 里声明的, 会使用消息机制进行调用. 当 `greetings(person:)` 被触发时, `sayHi()` 会通过函数表被派发到 `Person` 对象, 而 `misunderstoodPerson` 重写之后会是用消息机制, 而 `MisunderstoodPerson` 的函数表依旧保留了 `Person` 的实现, 紧接着歧义就产生了.

在这里的解决方法是保证函数使用相同的消息派发机制. 你可以给函数加上 `dynamic` 修饰符, 或者是把函数的实现从 extension 移动到类最初声明的作用域里.

理解了 Swift 的派发方式, 就能够理解这个行为产生的原因了, 虽然 Swift 不应该让我们遇到这个问题.

SR-103

这个 Swift bug 触发了定义在协议拓展的默认实现, 即使是子类已经实现这个函数的情况下. 为了说明这个问题, 我们先定义一个协议, 并且给里面的函数一个默认实现:

```
protocol Greetable {
```

```
    func sayHi()
}
extension Greetable {
    func sayHi() {
        print("Hello")
    }
}
func greetings(greeter: Greetable) {
    greeter.sayHi()
}
```

现在, 让我们定义一个遵守了这个协议的类. 先定义一个 `Person` 类, 遵守 `Greetable` 协议, 然后定义一个子类 `LoudPerson`, 重写 `sayHi()` 方法.

```
class Person: Greetable {
}
class LoudPerson: Person {
    func sayHi() {
        print("HELLO")
    }
}
```

你们发现 `LoudPerson` 实现的函数前面没有 `override` 修饰, 这是一个提示, 也许代码不会像我们设想的那样运行. 在这个例子里, `LoudPerson` 没有在 `Greetable` 的协议记录表(Protocol Witness Table)里成功注册, 当 `sayHi()` 通过 `Greetable` 协议派发时, 默认的实现就会被调用.

解决的方法就是, 在类声明的作用域里就要提供所有协议里定义的函数, 即使已经有默认实现. 或者, 你可以在类的前面加上一个 `final` 修饰符, 保证这个类不会被继承.

Doug Gregor 在 Swift-Evolution 邮件列表里提到, 通过显式地重新把函数声明为类的函数, 就可以解决这个问题, 并且不会偏离我们的设想.

其它 bug (Other bugs)

Another bug that I thought I'd mention is `SR-435`. It involves two protocol extensions, where one extension is more specific than the other. The example in the bug shows one un-constrained extension, and one extension that is constrained to `Equatable` types. When the method is invoked inside a protocol, the more specific method is not called. I'm not sure if this always occurs or not, but seems important to keep an eye on.

另外一个 bug 我在 `SR-435` 里已经提过了. 当有两个协议拓展, 而其中一个更加具体时就会触发. 例如, 有一个不受约束的 extension, 而另一个被 `Equatable` 约束, 当这个方法通过协议派发, 约束比较多的那个 extension 的实现则不会被调用. 我不太确定这是不是百分之百能复现, 但有必要留个心眼.

If you are aware of any other Swift dispatch bugs, drop me a line and I'll update this blog post.

如果你发现了其它 Swift 派发的 bug 的话, @一下我 我就会更新到这篇博客里.

有趣的 Error (Interesting Error)

有一个很好玩的编译错误, 可以窥见到 Swift 的计划. 就像之前说的, 类拓展使用直接派发, 所以你试图 override 一个声明在 extension 里的函数的时候会发生什么?

```
class MyClass {
}
extension MyClass {
    func extensionMethod() {}
}
class SubClass: MyClass {
    override func extensionMethod() {}
}
```

上面的代码会触发一个编译错误 `Declarations in extensions can not be overridden yet` (声明在 extension 里的方法不可以被重写). 这可能是 Swift 团队打算加强函数表派发的一个征兆. 又或者这只是我过度解读, 觉得这门语言可以优化的地方.

致谢 Thanks

我希望了解函数派发机制的过程中你感受到了乐趣, 并且可以帮助你更好的理解 Swift. 虽然我抱怨了 `NSObject` 相关的一些东西, 但我还是觉得 Swift 提供了高性能的可能性, 我只是希望可以有足够简单的方式, 让这篇博客没有存在的必要.