



OneV's Den

上善若水，人淡如菊

2016-11-29 • 能工巧匠集

面向协议编程与 Cocoa 的邂逅 (上)

本文是笔者在 MDCC 16 (移动开发者大会) 上 iOS 专场中的主题演讲的文字整理。您可以在[这里](#)找到演讲使用的 Keynote，部分示例代码可以在 MDCC 2016 的[官方 repo](#) 中找到。因为全部内容比较长，所以分成了上下两个部分，本文 (上) 主要介绍了一些理论方面的内容，包括面向对象编程存在的问题，面向协议的基本概念和决策模型等，[下半部分](#)主要展示了一些笔者日常使用面向协议思想和 Cocoa 开发结合的示例代码，并对其进行了一些解说。

引子

面向协议编程 (Protocol Oriented Programming, 以下简称 POP) 是 Apple 在 2015 年 WWDC 上提出的 Swift 的一种编程范式。相比与传统的面向对象编程 (OOP)，POP 显得更加灵活。结合 Swift 的值语义特性和 Swift 标准库的实现，这一年来大家发现了很多 POP 的应用场景。本次演讲希望能在介绍 POP 思想的基础上，引入一些日常开发中可以使用 POP 的场景，让与会来宾能够开始在日常工作中尝试 POP，并改善代码设计。

起 · 初识 – 什么是 Swift 协议

Protocol

Swift 标准库中有 50 多个复杂不一的协议，几乎所有的实际类型都是满足若干协议的。protocol 是 Swift 语言的底座，语言的其他部分正是在这个底座上组织和建立起来的。这和我们熟知的面向对象的构建方式很不一样。

一个最简单但是有实际用处的 Swift 协议定义如下：

```
protocol Greetable {
    var name: String { get }
    func greet()
}
```

这几行代码定义了一个名为 `Greetable` 的协议，其中有一个 `name` 属性的定义，以及一个 `greet` 方法的定义。

所谓协议，就是一组属性和/或方法的定义，而如果某个具体类型想要遵守一个协议，那它需要实现这个协议所定义的所有这些内容。协议实际上做的事情不过是“关于实现的约定”。

面向对象

在深入 Swift 协议的概念之前，我想先重新让大家回顾一下面向对象。相信我们不论在教科书或者是博客等各种地方对这个名词都十分熟悉了。那么有一个很有意思，但是其实并不是每个程序员都想过的问题，面向对象的核心思想究竟是什么？

我们先来看一段面向对象的代码：

```
class Animal {
    var leg: Int { return 2 }
    func eat() {
        print("eat food.")
    }
    func run() {
        print("run with \(leg) legs")
    }
}
```

```
class Tiger: Animal {
    override var leg: Int { return 4 }
    override func eat() {
        print("eat meat.")
    }
}

let tiger = Tiger()
tiger.eat() // "eat meat"
tiger.run() // "run with 4 legs"
```

父类 `Animal` 定义了动物的 `leg` (这里应该使用虚类, 但是 Swift 中没有这个概念, 所以先请无视这里的 `return 2`), 以及动物的 `eat` 和 `run` 方法, 并为它们提供了实现。子类的 `Tiger` 根据自身情况重写了 `leg` (4 条腿)和 `eat` (吃肉), 而对于 `run`, 父类的实现已经满足需求, 因此不必重写。

我们看到 `Tiger` 和 `Animal` 共享了一部分代码, 这部分代码被封装到了父类中, 而除了 `Tiger` 的其他的子类也能够使用 `Animal` 的这些代码。这其实就是 OOP 的核心思想 – 使用封装和继承, 将一系列相关的内容放到一起。我们的前辈们为了能够对真实世界的对象进行建模, 发展出了面向对象编程的概念, 但是这套理念有一些缺陷。虽然我们努力用这套抽象和继承的方法进行建模, 但是实际的事物往往是一系列**特质的组合**, 而不单单是以一脉相承并逐渐扩展的方式构建的。所以最近大家越来越发现面向对象很多时候其实不能很好地对事物进行抽象, 我们可能需要寻找另一种更好的方式。

面向对象编程的困境

横切关注点

我们再来看一个例子。这次让我们远离动物世界, 回到 Cocoa, 假设我们有一个 `ViewController`, 它继承自 `UIViewController`, 我们向其中添加一个 `myMethod`:

```
class ViewController: UIViewController
{
    // 继承
    // view, isFirstResponder()...

    // 新加
    func myMethod() {

    }
}
```

如果这时候我们又有一个继承自 `UITableViewController` 的 `AnotherViewController`, 我们也想向其中添加同样的 `myMethod`:

```
class AnotherViewController: UITableViewController
{
    // 继承
    // tableView, isFirstResponder()...

    // 新加
    func myMethod() {

    }
}
```

这时, 我们迎来了 OOP 的第一个大困境, 那就是我们很难在不同继承关系的类里共用代码。这里的问题用“行话”来说叫做“横切关注点” (Cross-Cutting Concerns)。我们的关注点 `myMethod` 位于两条继承链 (`UIViewController -> ViewController` 和 `UIViewController -> UITableViewController -> AnotherViewController`) 的横切面上。面向对象是一种不错的抽象方式, 但是肯定不是最好的方式。它无法描述两个不同事物具有某个相同特性这一点。在这里, 特性的组合要比继承更贴切事物的本质。

想要解决这个问题, 我们有几个方案:

- Copy & Paste

况下。这诚然可以理解，但是这也是坏代码的开头。我们应该尽量避免这种做法。

- 引入 BaseViewController

在一个继承自 `UIViewController` 的 `BaseViewController` 上添加需要共享的代码，或者干脆在 `UIViewController` 上添加 `extension`。看起来这是一个稍微靠谱的做法，但是如果不断这么做，会让所谓的 `Base` 很快变成垃圾堆。职责不明确，任何东西都能扔进 `Base`，你完全不知道哪些类走了 `Base`，而这个“超级类”对代码的影响也会不可预估。

- 依赖注入

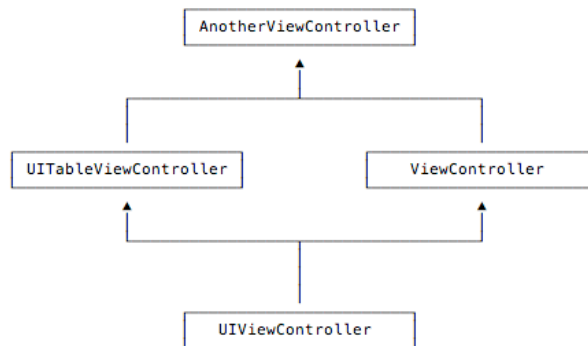
通过外界传入一个带有 `myMethod` 的对象，用新的类型来提供这个功能。这是一个稍好的方式，但是引入额外的依赖关系，可能也是我们不太愿意看到的。

- 多继承

当然，Swift 是不支持多继承的。不过如果有多继承的话，我们确实可以从多个父类进行继承，并将 `myMethod` 添加到合适的地方。有一些语言选择了支持多继承 (比如 C++)，但是它会带来 OOP 中另一个著名的问题：菱形缺陷。

菱形缺陷

上面的例子中，如果我们有多继承，那么 `ViewController` 和 `AnotherViewController` 的关系可能会是这样的：



在上面这种拓扑结构中，我们只需要在 `ViewController` 中实现 `myMethod`，在 `AnotherViewController` 中也可以继承并使用它了。看起来很完美，我们避免了重复。但是多继承有一个无法回避的问题，就是两个父类都实现了同样的方法时，子类该怎么办？我们很难确定应该继承哪一个父类的方法。因为多继承的拓扑结构是一个菱形，所以这个问题又被叫做菱形缺陷 (Diamond Problem)。像是 C++ 这样的语言选择粗暴地将菱形缺陷的问题交给程序员处理，这无疑非常复杂，并且增加了人为错误的可能性。而绝大多数现代语言对多继承这个特性选择避而远之。

动态派发安全性

Objective-C 恰如其名，是一门典型的 OOP 语言，同时它继承了 Small Talk 的消息发送机制。这套机制十分灵活，是 OC 的基础思想，但是有时候相对危险。考虑下面的代码：

```
ViewController *v1 = ...
[v1 myMethod];

AnotherViewController *v2 = ...
[v2 myMethod];

NSArray *array = @[v1, v2];
for (id obj in array) {
    [obj myMethod];
}
```

我们如果在 `ViewController` 和 `AnotherViewController` 中都实现了 `myMethod` 的话，这段代码是没有问题的。`myMethod` 将会被动态发送给 `array` 中的 `v1` 和 `v2`。但是，要是我们有一个没有实现 `myMethod` 的类型，会如何呢？

```
// v3 没有实现 `myMethod`

NSArray *array = @[v1, v2, v3];
for (id obj in array) {
    [obj myMethod];
}

// Runtime error:
// unrecognized selector sent to instance blabla
```

编译依然可以通过，但是显然，程序将在运行时崩溃。Objective-C 是不安全的，编译器默认你知道某个方法确实有实现，这是消息发送的灵活性所必须付出的代价。而在 app 开发看来，用可能的崩溃来换取灵活性，显然这个代价太大了。虽然这不是 OOP 范式的问题，但它确实在 Objective-C 时代给我们带来了切肤之痛。

三大困境

我们可以总结一下 OOP 面临的这一个问题。

- 动态派发安全性
- 横切关注点
- 菱形缺陷

首先，在 OC 中动态派发让我们承担了在运行时才发现错误的风险，这很有可能是发生在线上产品中的错误。其次，横切关注点让我们难以对对象进行完美的建模，代码的重用也会更加糟糕。

承·相知 – 协议扩展和面向协议编程

使用协议解决 OOP 困境

协议并不是什么新东西，也不是 Swift 的发明。在 Java 和 C# 里，它叫做 `Interface`。而 Swift 中的 `protocol` 将这个概念继承了下来，并发扬光大。让我们回到一开始定义的那个简单协议，并尝试着实现这个协议：

```
protocol Greetable {
    var name: String { get }
    func greet()
}
```

```
struct Person: Greetable {
    let name: String
    func greet() {
        print("你好 \ \(name)")
    }
}

Person(name: "Wei Wang").greet()
```

实现很简单，`Person` 结构体通过实现 `name` 和 `greet` 来满足 `Greetable`。在调用时，我们就可以使用 `Greetable` 中定义的方法了。

动态派发安全性

除了 `Person`，其他类型也可以实现 `Greetable`，比如 `Cat`：

```
struct Cat: Greetable {
    let name: String
    func greet() {
        print("meow~ \ \(name)")
    }
}
```

现在，我们就可以将协议作为标准类型，来对方法调用进行动态派发了：

```
let array: [Greetable] = [
    Person(name: "Wei Wang"),
    Cat(name: "onevcat")]
```

```

}
// 你好 Wei Wang
// meow~ onevc

```

对于没有实现 `Greetable` 的类型，编译器将返回错误，因此不存在消息误发送的情况：


```

struct Bug: Greetable {
    let name: String
}

// Compiler Error:
// 'Bug' does not conform to protocol 'Greetable'
// protocol requires function 'greet()'

```

这样一来，动态派发安全性的问题迎刃而解。如果你保持在 `Swift` 的世界里，那这个你的所有代码都是安全的。

-  动态派发安全性
- 横切关注点
- 菱形缺陷

横切关注点

使用协议和协议扩展，我们可以很好地共享代码。回到上一节的 `myMethod` 方法，我们来看看如何使用协议来搞定它。首先，我们可以定义一个含有 `myMethod` 的协议：

```

protocol P {
    func myMethod()
}

```

注意这个协议没有提供任何的实现。我们依然需要在实际类型遵守这个协议的时候为它提供具体的实现：

```

// class ViewController: UIViewController
extension ViewController: P {
    func myMethod() {
        doWork()
    }
}

// class AnotherViewController: UITableViewController
extension AnotherViewController: P {
    func myMethod() {
        doWork()
    }
}

```

你可能不禁要问，这和 `Copy & Paste` 的解决方式有何不同？没错，答案就是 – 没有不同。不过稍安勿躁，我们还有其它科技可以解决这个问题，那就是协议扩展。协议本身并不是很强大，只是静态类型语言的编译器保证，在很多静态语言中也有类似的概念。那到底是什么让 `Swift` 成为了一门协议优先的语言？真正使协议发生质变，并让大家如此关注的原因，其实是在 `WWDC 2015` 和 `Swift 2` 发布时，`Apple` 为协议引入了一个新特性，协议扩展，它为 `Swift` 语言带来了一次革命性的变化。

所谓协议扩展，就是我们可以为一个协议提供默认的实现。对于 `P`，可以在 `extension P` 中为 `myMethod` 添加一个实现：

```

protocol P {
    func myMethod()
}

extension P {
    func myMethod() {
        doWork()
    }
}

```

使用 `myMethod` 的实现了：



```
extension ViewController: P { }
extension AnotherViewController: P { }

viewController.myMethod()
anotherViewController.myMethod()
```

不仅如此，除了已经定义过的方法，我们甚至可以在扩展中添加协议里没有定义过的方法。在这些额外的方法中，我们可以依赖协议定义过的方法进行操作。我们之后会看到更多的例子。总结下来：

- 协议定义
 - 提供实现的入口
 - 遵循协议的类型需要对其进行实现
- 协议扩展
 - 为入口提供默认实现
 - 根据入口提供额外实现

这样一来，横切点关注的问题也简单安全地得到了解决。

-  动态派发安全性
-  横切关注点
- 菱形缺陷

菱形缺陷

最后我们看看多继承。多继承中存在的一个重要问题是菱形缺陷，也就是子类无法确定使用哪个父类的方法。在协议的对应方面，这个问题虽然依然存在，但却是可以唯一安全地确定的。我们来看一个多个协议中出现同名元素的例子：

```
protocol Nameable {
    var name: String { get }
}

protocol Identifiable {
    var name: String { get }
    var id: Int { get }
}
```

如果有一个类型，需要同时实现两个协议的话，它**必须**提供一个 `name` 属性，来**同时**满足两个协议的要求：

```
struct Person: Nameable, Identifiable {
    let name: String
    let id: Int
}

// `name` 属性同时满足 Nameable 和 Identifiable 的 name
```

这里比较有意思，又有点让人困惑的是，如果我们为其中的某个协议进行了扩展，在其中提供了默认的 `name` 实现，会如何。考虑下面的代码：

```
extension Nameable {
    var name: String { return "default name" }
}

struct Person: Nameable, Identifiable {
    // let name: String
    let id: Int
}

// Identifiable 也将使用 Nameable extension 中的 name
```

这样的编译是可以通过的，虽然 `Person` 中没有定义 `name`，但是通过 `Nameable` 的 `name`（因为它是静态派发的），`Person` 依然可以遵守 `Identifiable`。不过，当 `Nameable` 和 `Identifiable` 都有 `name` 的协议扩展的

```
extension Nameable {
    var name: String { return "default name" }
}

extension Identifiable {
    var name: String { return "another default name" }
}

struct Person: Nameable, Identifiable {
    // let name: String
    let id: Int
}

// 无法编译, name 属性冲突
```

这种情况下, `Person` 无法确定要使用哪个协议扩展中 `name` 的定义。在同时实现两个含有同名元素的协议, 并且它们都提供了默认扩展时, 我们需要在具体的类型中明确地提供实现。这里我们将 `Person` 中的 `name` 进行实现就可以了:




```
extension Nameable {
    var name: String { return "default name" }
}

extension Identifiable {
    var name: String { return "another default name" }
}

struct Person: Nameable, Identifiable {
    let name: String
    let id: Int
}

Person(name: "onevcats", id: 123).name // onevcats
```

这里的行为看起来和菱形问题很像, 但是有一些本质不同。首先, 这个问题出现的前提条件是同名元素以及同时提供了实现, 而协议扩展对于协议本身来说并不是必须的。其次, 我们在具体类型中提供的实现一定是安全和确定的。当然, 菱形缺陷没有被完全解决, Swift 还不能很好地处理多个协议的冲突, 这是 Swift 现在的不足。

-  动态派发安全性
-  横切关注点
-  菱形缺陷

本文的下半部分将展示一些笔者日常使用面向协议思想和 Cocoa 开发结合的示例代码, 并对其进行了一些解说。

最近的文章

面向协议编程与 Cocoa 的邂逅 (下)

本文是笔者在 MDCC 16 (移动开发者大会) 上 iOS 专场中的主题演讲的文字整理。您可以在这里找到演讲使用的 Keynote, 部分示例代码可以在 MDCC 2016 的官方 repo 中找到。在上半部分主要介绍了一些理论方面的内容, 包括面向对象编程存在的问题, 面向协议的基本概念和决策模型等。本文 (下) 主要展示了一些笔者日常使用面向协议思想和 Cocoa 开发结合的示例代码, 并对其进行了一些解说。转·热恋 - 在日常开发中使用协议 WWDC 2015 在 POP 方面有一个非常优秀.....

2016-12-01 • 能工巧匠集

[继续阅读](#)

更早的文章

活久见的重构 - iOS 10 UserNotifications 框架解析

TL;DR iOS 10 中以前杂乱的和通知相关的 API 都被统一了, 现在开发者可以使用独立的 `UserNotifications.framework` 来集中管理和使用 iOS 系统中通知的功能。在此基础上, Apple 还增加了撤回单条通知, 更新已展示通知, 中途修改通知内容, 在通知中展示图片视频, 自定义通知 UI 等一系列新功能, 非常强大。对于开发者来说, 相较于之前版本, iOS 10 提供了一套非常易用的通知处理接口, 是 SDK 的一次重大重构。而之前的绝大部分通知相关 API 都已经被标.....

2016-08-08 • 能工巧匠集

[继续阅读](#)

3条评论

OneV's Den

登录

推荐

分享

最新发布




加入讨论...

通过以下方式登录


或注册一个 DISQUS 帐号

姓名

- 


许赞 · 10个月前

额,本文的下半部分是404,不过我找到了下文

回复 · 分享
- 

sunyazhou · 10个月前

喵神好文章
终于更新了

回复 · 分享
- 

大宝PKU · 10个月前

上述提到的Objective-C动态派发安全性问题, 难道不是编译器的原因? 上升不到语言层面吧, 不应该成为三大困境之一

回复 · 分享

在 ONEV'S DEN 上还有

Swift 并行编程现状和展望 - async/await 和参与者模式

17条评论 · 10个月前

Akring — Swift 的体验确实超过Objective-C, 虽然之前一年一次的重新入门每次都让我想放弃治疗, 但是每次新开项目的时候我都会坚定的选Swift...

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?

所有权宣言 - Swift 官方文章 Ownership Manifesto 译文评注版

19条评论 · 8个月前

Wally — 再次感谢喵大的热心翻译 m(_ _)m, 原文有些不太懂的部分, 还是得看中文会比较能增进理解的說 :)挺期待喵大有时间之余能再次翻译在今年的what's ...

活久见的重构 - iOS 10 UserNotifications 框架解析

39条评论 · 1年前

sanSong zhang —

订阅

在您的网站上使用 Disqus添加 Disqus添加 隐私

