30-如何制定一套适合自己团队的iOS编码规范？

你好，我是戴铭。

如果团队成员的编码规范各不相同，那么你在接收其他人的代码时是不是总会因为无法认同他的代码风格，而想着去重写呢。但是，重写这个事儿不只会增加梳理逻辑和开发成本，而且重写后出现问题的风险也会相应增加。那么，这个问题应该如何解决呢？

在我看来，如果出现这种情况，你的团队急需制定出一套适合自己团队的编码规范。有了统一的编码规范，就能有效避免团队成员由于代码风格不一致而导致的相互认同感缺失问题。

那么，如何制定编码规范呢？在接下来的内容里，我会先跟你说说，我认为的好的编码规范。你在制定编码规范时，也可以按照这个思路去细化出更多、更适合自己的规范，从而制定出团队的编码规范。然后，我会再和你聊聊如何通过 Code Review 的方式将你制定的编码规范进行落地。

好的代码规范

关于好的代码规范，接下来我会从常量、变量、属性、条件语句、循环语句、函数、类，以及分类这8个方面和你一一说明。

常量

在常量的使用上，我建议你要尽量使用类型常量，而不是使用宏定义。比如，你要定义一个字符串常量，可以写成：

```
static NSString * const STMProjectName = @"GCDFetchFeed"
```

变量

对于变量来说，我认为好的编码习惯是：

1. 变量名应该可以明确体现出功能，最好再加上类型做后缀。这样也就明确了每个变量都是做什么的，而不是把一个变量当作不同的值用在不同的地方。
2. 在使用之前，需要先对变量做初始化，并且初始化的地方离使用它的地方越近越好。
3. 不要滥用全局变量，尽量少用它来传递值，通过参数传值可以减少功能模块间的耦合。

比如，下面这段代码中，当名字为字符串时，就可以把字符串类型作为后缀加到变量名后面。

```
let nameString = "Tom"
print("\(nameString)")

nameLabel.text = nameString
```

属性

在iOS开发中，关于属性的编码规范，需要针对开发语言做区分：

- Objective-C 里的属性，要尽量通过 get 方法来进行懒加载，以避免无用的内存占用和多余的计算。
- Swift 的计算属性如果是只读，可以省掉 get 子句。示例代码如下：

```
var rectangleArea: Double {
  return long * wide
}
```

条件语句

在条件语句中，需要考虑到条件语句中可能涉及的所有分支条件，对于每个分支条件都需要考虑到，并进行处理，减少或不使用默认处理。特别是使用 Switch 处理枚举时，不要有 default 分支。

在iOS开发中，你使用 Swift 语言编写 Switch 语句时，如果不加default分支的话，当枚举有新增值时，编译器会提醒你增加分支处理。这样，就可以有效避免分支漏处理的情况。

另外，条件语句的嵌套分支不宜过多，可以充分利用 Swift 中的 guard 语法。比如，这一段处理登录的示例代码：

```
if let userName = login.userNameOK {
    if let password = login.passwordOK {
        // 登录处理
        ...
    } else {
        fatalError("login wrong")
    }
} else {
    fatalError("login wrong")
}
```

上面这段代码表示的是，当用户名和密码都没有问题时再进行登录处理。那么，我们使用 guard 语法时，可以改写如下：

```
guard
  let userName = login.userNameOK,
  let password = login.passwordOK,
  else {
    fatalError("login wrong")
}
// 登录处理
...
```

可以看到，改写后的代码更易读了，异常处理都在一个区域，guard 语句真正起到了守卫的职责。而且你一

旦声明了 guard，编译器就会强制你去处理异常，否则就会报错。异常处理越完善，代码就会越健壮。所以，条件语句的嵌套处理，你可以考虑使用guard语法。

循环语句

在循环语句中，我们应该尽量少地使用 continue 和 break，同样可以使用 guard 语法来解决这个问题。解决方法是：所有需要 continue 和 break 的地方统一使用 guard 去处理，将所有异常都放到一处。这样做的好处是，在维护的时候方便逻辑阅读，使得代码更加易读和易于理解。

函数

对于函数来说，体积不宜过大，最好控制在百行代码以内。如果函数内部逻辑多，我们可以将复杂逻辑分解成多个小逻辑，并将每个小逻辑提取出来作为一个单独的函数。每个函数处理最小单位的逻辑，然后一层一层往上组合。

这样，我们就可以通过函数名明确那段逻辑处理的目的，提高代码的可读性。

拆分成多个逻辑简单的函数后，我们需要注意的是，要对函数的入参进行验证，guard 语法同样适用于检查入参。比如下面的这个函数：

```
func saveRSS(rss: RSS?, store: Store?) {
    guard let rss = rss else {
        return
    }
    guard let store = store else {
        return
    }

    // 保存 RSS
    return
}
```

如上面代码所示，通过 guard语法检查入参 rss 和 store 是否异常，提高函数的健壮性会来得更容易些。

另外，函数内尽量避免使用全局变量来传递数据，使用参数或者局部变量传递数据能够减少函数对外部的依赖，减少耦合，提高函数的独立性，提高单元测试的准确性。

类

在Objective-C 中，类的头文件应该尽可能少地引入其他类的头文件。你可以通过 class 关键字来声明，然后在实现文件里引入需要的其他类的头文件。

对于继承和遵循协议的情况，无法避免引入其他类的头文件，所以你在代码设计时还是要尽量减少继承，特别是继承关系太多时不利于代码的维护和修改，比如说修改父类时还需要考虑对所有子类的影响，如果评估不全，影响就难以控制。

分类

在写分类时，分类里增加的方法名要尽量加上前缀，而如果是系统自带类的分类的话，方法名就一定要加上

前缀，来避免方法名重复的问题。

分类的作用如其名，就是对类做分类用的，所以我建议你，能够把一个类里的公共方法放到不同的分类里，便于管理维护。分类特别适合多人负责同一个类时，根据不同分类来进行各自不同功能的代码维护。

Code Review

上面的内容，就是在我看来比较好的iOS编码规范了。除此之外，你还可以参考其他公司对 iOS 开发制定的编码规范来完善自己团队的编码规范，比如 Spotify 的 Objective-C 编码规范、纽约时报的 Objective-C 的编码规范、Raywenderlich 的 Objective-C 编码规范、Raywenderlich 的 Swift 编码规范。

在我看来，好的代码规范首先要保证代码逻辑清晰，然后再考虑简洁、扩展、重用等问题。逻辑清晰的代码几乎不需要注释来说明，通过命名和清晰地编写逻辑就能够让其他人快速读懂。

不需要注释就能轻松读懂的代码，使用的语言特性也必然是通用和经典的，过新的语言特性和黑魔法不利于代码逻辑的阅读，应该减少使用，即使使用也需要多加注释，避免他人无法理解。

当你制定出好的代码规范后，就需要考虑如何将代码规范落地执行了。代码规范落地最好的方式就是 Code Review 。通过 Code Review ，你可以去检查代码规范是否被团队成员执行，同时还可以在 Code Review 时，及时指导代码编写不规范的同学。

那么，**怎么做Code Review 会比较好呢？**

首先，我觉得要利用好 Code Review 这个卡点，先使用静态检查工具对提交的代码进行一次全面检查。

如果是 Swift 语言的话，你可以使用 SwiftLint工具来检查代码规范。Swift 通过 Hook Clang 和 SourceKit 中 AST 的回调来检查源代码，如何使用SourceKit 开发工具可以参看这篇文章“Uncovering SourceKit”。

SwiftLint 检查的默认规则，你可以参考它的规则说明。SwiftLint 也支持自定义检查规则，支持你添加自己制定的代码规范。你可以在 SwiftLint 目录下添加一个 .swiftlint.yml 配置文件来自定义基于正则表达式的自定义规则。具体方法，你可以参看官方定义自定义规则的说明。

如果你是使用 Objective-C 语言开发的话，可以使用 OCLint 来做代码规范检查。关于 OCLint 如何定制自己的代码规范检查，你可以参看杨萧玉的这篇博文“使用 OCLint 自定义 MVVM 规则”。

然后，进行人工检查。

人工检查，就是使用类似 Phabricator 这样的Code Review工具平台，来分配人员审核提交代码，审核完代码后，审核人可以进行通过、打回、评论等操作。这里需要注意的是，人工检查最容易沦为形式主义，因此为了避免团队成员人工检查成为形式，在开始阶段最好能让团队中编码习惯好、喜欢交流的人来做审核人，以起到良好的示范作用，并以此作为后续的执行标准。

你可能会有疑问，既然工具可以检查代码规范，为什么还需要人工再检查一遍？我想说的是，工具确实可以通过不断完善，甚至引入 AI 分析来提高检查结果的准确性，但是，**我认为 Code Review 之所以最终还是需要人工检查的原因是，通过团队成员之间互相检查代码的方式，希望能够达到相互沟通交流，甚至相互学习的效果。**

试想一下，如果你经过了大量的思考，花费了很多心思写出来一段自认为完美的代码，这时候可以再得到团队其他成员的鼓励，是不是会干劲儿十足呢。相反地，如果你马虎大意，或者经验不足而写出了不好的代码，通过 Code Review 而得到了团队其他成员的建议和指导，是不是能够让你的编码水平快速提高，同时还能够吸纳更多人的经验呢。

Code Review 的过程也能够对代码规范进行迭代改进，最后形成一份能体现出团队整体智慧的代码规范。以后再有新成员加入时，他们也能够快速达到团队整体的编码水平，这就好比一锅老汤，新食材放进来涮涮，很快就有了相同的味道。

小结

在今天这篇文章中，我和你分享了什么是好的代码规范，以及如何通过 Code Review 将编码规范落实到团队中。

对于编码规范来说，我认为不用过于复杂，只要坚持能够让代码逻辑清晰这个原则就可以了，剩下的所有规则都围绕着这个原则来。代码逻辑清晰是高质量的代码工程最基本、最必要的条件。如果代码不清晰的话，那么其他的扩展、重用、简洁优雅都免谈。

写代码的首要任务是能让其他人看得懂，千万不要优先过度工程化。难懂的代码无论工程化做得多好，到最后都会被其他人弃用、重构掉。这是一种资源浪费，损己又损人。

课后作业

你的团队是如何做Code Review 的？如果你的团队还没有 Code Review，那原因是什么呢？

感谢你的收听，欢迎你在评论区给我留言分享你的观点，也欢迎把它分享给更多的朋友一起阅读。

精选留言：

- clownfish 2019-05-18 08:41:53
  前辈,我看了你说的大部分都是Effective 2.0里面的内容,有没有一些您在工作实践中的代码规范要求呢,例

如我想到的使用#pragma 做代码分区等一些, [8赞]

作者回复2019-05-20 17:37:53
15年我整理过一篇，可以参看下。https://github.com/ming1016/study/wiki/%E6%9E%84%E5%BB%BAiOS%E7%A8%B3%E5%AE%9A%E5%BA%94%E7%94%A8%E6%9E%B6%E6%9E%84%E6%97%B6%E6%96%B9%E6%A1%88%E9%80%89%E6%8B%A9%E7%9A%84%E6%80%9D%E8%80%83

- 大头 2019-05-20 15:32:00
同意一楼的建议

- 郑杰 2019-05-18 09:39:47
静态类型需要 变量名加类感觉没有必要吧