
iOS
成长之路
作者 / SwiftOldDriver
2017
夏

目录

作者：李富强，@李富强Jason，美团外卖 iOS 架构师
博客：http://blog.csdn.net/colorapp

- iOS 组件化 —— 路由设计思路分析

  作者： @一缕殇流化隐半边冰霜，饿了么 iOS 工程师，微博 @halfrost

  博客：http://www.jianshu.com/u/12201cdd5d7a

- iOS App 签名的原理

  作者：bang，iOS 开发者，JSPatch 作者，做过推特中文圈和一些 iOS 独立应用，曾在腾讯工作，开发 QQ 邮箱与微信读书，现就职于蚂蚁金服。

  博客：http://blog.cnbang.net

- 再看关于 Storyboard 的一些争论

  作者：王巍 (@onevcat) iOS 开发者，ObjC 中国发起人

- Swift ABI 稳定性蓝图

  译者：eleven，泊学网(https://boxueio.com)站长

编辑

本书文章由 @故胤道长、@一缕殇流化隐半边冰霜、@没故事的卓同学、@Onetaway 编辑。关于这本书的任何的意见建议可以在微博上私信我们。

序：夏 · 成长

感恩

3月3日，《iOS 成长之路》发布了第一期。之后就收到了大量读者的好评和改进的建议。经过2个多月的沉淀，全新的第二期又与大家见面了。

在这里当然要感谢各位读者大人的支持，如果没有大家的支持，可能也就没有这精美的第二期了。在这一期里，继续秉承着第一期我们的初心，精选出十几篇高质量的文章，为初中级 iOS 开发者的成长之路上献出一份力。

关于成长

卓同学突然要我写这一期的序，我感觉了很大的压力，生怕序写不好会影响这一期的销量。目前我也是一个普通水平的 iOS 开发者，我就简要的谈谈我是如何成长的。2013年我从大学毕业，当时移动互联网的缺口很大，iOS 和 Android 找工作非常容易，听说理发师都要转行过来开发 iOS，我也是从那时起，进入了 iOS 的行业。但是人才市场是残酷的，优胜劣汰，还有不断攀升的房价，这些也算是催我成长的外部原因。今年 iOS 的整个市场应该是冷静下来了，招人的要求也变高了很多。iOS 刚入行也许看上去很简单，轻轻松松就可以拿15-20K的高薪。实际上，想继续再往下深入，甚至成为 iOS 大牛，需要了解的知识是很多的。首先需要熟悉操作系统、计算机网络、数据结构、算法、编译原理，这些都是基础中的基础，都需要深入学习的。接着还需要了解各种设计模式，这块需要有一定的代码量的积淀才行。熟悉了设计模式以后就可以写出更加漂亮的代码了。上面说的这两块我定义为“经济基础”。在武林绝学里面，这部分算是心法，是内功。内功的修炼是非一日之功，需要长久的积累。

当经济基础稳固以后，内功深厚以后，可以继续建造“上层建筑”了，领悟一些招式了。“上层建筑”的花样就很多了，深入学习 Swift，跨平台技术(RN、Weex、H5)，FRP(RxSwift、ReactiveCocoa)……这些技术都值得去学习。这部分算是招式，学会了更多的招式就可以更高效的完成各种需求。

当然可以边建造“上层建筑”，边补自己薄弱的“基础”。边修炼内功，边学招式。在上一期里面有一篇 Peak 老师写的 HTTP 相关的文章，如果认真读完，可以感受到 Peak 老师在计算机网络方面的功底非常深厚。iOS 想优化网络，想优化性能，都要从这些基础知识入手。有读者好奇为啥上一期突然有一篇 HTTP 的文章，这就是原因了。在我的成长之路上，就是一边补着计算机基础，修炼内功，一边建造着上层建筑的过程，学习招式。可能某些阶段无法进步了，也许是内功不够深厚，于是需要去补补基础，加深一下内功，内功修炼到一定层次了，一些招式的领悟就自然而然的会了。

内功有多深厚就决定了能领悟多高超的武功。就像九阳神功，能修炼到第几重就取决于内功有多深厚了。地基有多扎实也就决定了楼房能盖多高。所以对计算机基础知识的掌握也就决定了一个人能在这一行能走多远了。这也许是 BAT 大公司招人和小公司招人最大的区别吧，大公司更加看重一个人的潜力，对计算机基础的考察力度更大，小公司更加偏向一个人进来能不能干活，招式耍的是否够溜。所以从长远的角度来看，内功的修炼是需要长期进行的，公司内用到的框架，这些招式是需要熟练掌握的。

程序员成长的路上是苦涩的，尽管沿路的风景是那么的美丽，但是我们不能驻足不前。请相信自己，我们生来如同璀璨的夏日之花，不凋不败，妖冶如火，承受着一堆需求的负荷，依旧乐此不疲，用代码书写着人生。

成长的路上，与君共勉！

愿景

正如这本书的书名，我们希望这些文章能够在你的 iOS 成长之路上对你有所帮助。

这本书的大部分收入也会以稿费的形式付给这些文章的原创作者。希望能够以此形成一个正向的循环，让更多优秀的文章能够出现。

@halfrost

2017年5月15号 于 上海

Chris Lattner 访谈录(下)

译者：@故胤道长

话题

- **Swift 在 Server 和操作系统方面有着怎样的雄心抱负?**
- **Swift 与 Objective-C 的爱恨情仇?**
- **Swift 之父对于 RxSwift 和 ARC 有什么独到的见解?**
- **随着 Swift 之父的出走，这门语言还会继续高歌猛进吗?**

访谈实录

Swift 在系统、服务器、网页端的发展

16. Swift 在服务器，或者 Linux 上可以说运行得不错。你们是一开始就计划在服务器或者系统端运行 Swift，还是说你们更希望 Swift 专注于 iOS 开发，而不是去与 python 或 Rails 竞争?

你如果去看苹果官方的 Swift 书，里面有这样一句："Swift 的目标是，上能写应用程序，下能写操作系统（Swift was designed to scale from hello world to an entire operating system）"。所以我们一开始，就是要将它创作**成为一门一统天下的语言**。

这也许有点痴人说梦，但是大家等着，过几年就知道了。无论是我还是苹果的其他人，都把 Swift 当成是未来世界的主流语言来看的，它将会超越 Python，甚至有一天取代 C。那么我们是怎么实现这一步的呢?

开源是重要的一环。你不开源，别的平台就不大想用这个语言。当各种各样的开发都采用 Swift，Swift 一统天下的目标也就越来越现实。现在很多学校的计算机基础教育就在教 Swift，它越来越流行了。

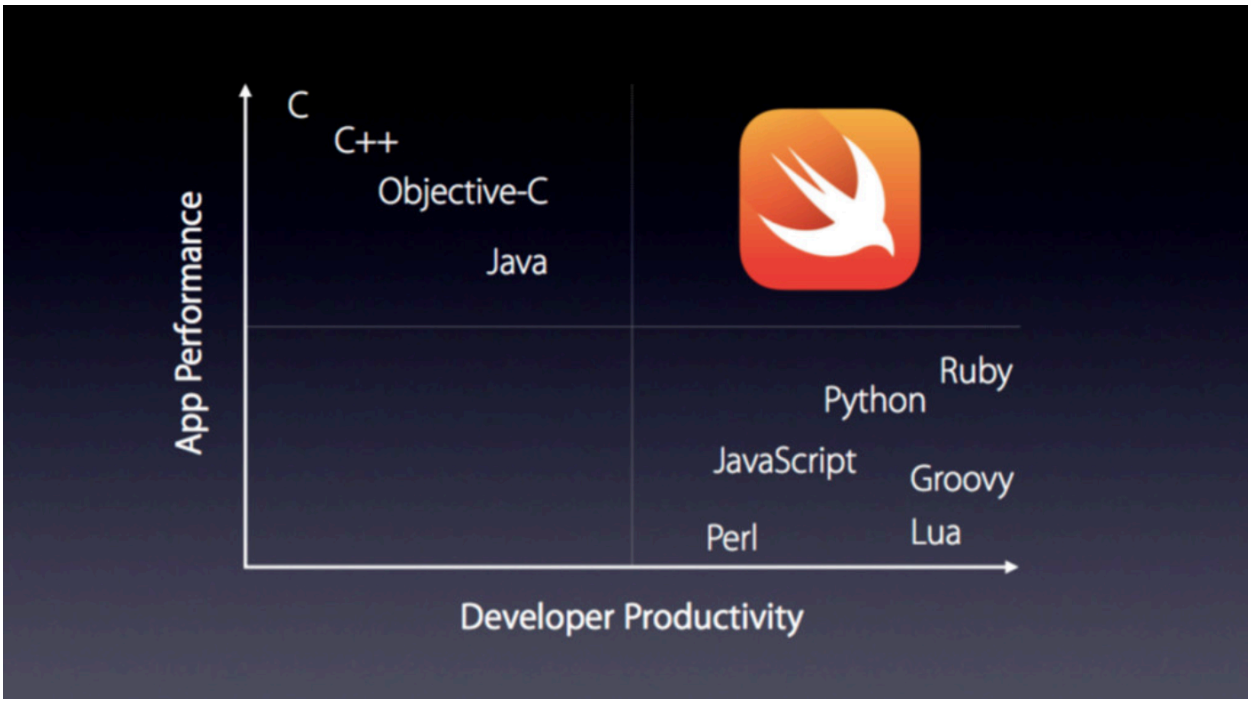
所以嘛，第一步我们就是让这个语言流行起来，让大家使用它。我对"流行"的定义是，Swift 必须要有一个杀手级的产品，这样大家就会知道 Swift 有多好，大家都会使用它。现在 iOS 平台和 Mac OS 平台有很多非常棒的 Swift 应用。这样我们开始第二步，开源。第三步，我们要走得更远。

什么叫走得更远? 我觉得现在我们要做的就是把 Swift 应用到服务器端。其实服务器和移动应用开发颇有类似，比如架构设计和函数库调用上。但是，唯一的麻烦就是我们得让 Swift 能在 Lunix 上流畅运行。同时构建大量服务器端的库函数。现在 Swift.org 上已经有专门的版块讨论服务器端上的开发了，大家集思广益的感觉非常好。

再接下来，**Swift 要取代 Java**，无论是脚本语言还是底层的系统设计，Swift 最终都应该能应付自如。

脚本语言上，开源社区和我们苹果内部都在尝试将正则表达式、多行字符串等脚本语言的特征都加入到 Swift 当中，虽然工作量很大，但我认为它们最终都将成为 Swift 的一个部分。

系统开发方面，我觉得取代 Java 最重要的一点就是 Swift 一定要有自己的特色。我觉得 Rust 是一个不错的语言，虽然现在没多少人用。Swift 在某些顶层开发上要明显优于 Rust。再等过些年，当 Swift 在系统开发上真正流行起来之时，Swift 就离一统天下不远了。



17. 对于 Swift 在服务器上的发展，你觉得交给开源社区去做就足够了吗？苹果自己会不会推出面向服务器端的 Swift 函数库?

首先我觉得若要成为服务器端的流行语言，这几个部分 Swift 必须具备：编码和解码，网络传输协议，HTTP。这些部分我觉得要成为标准函数库，因为它们是最基本的东西，苹果内部自己来做也许更好，因为能确保质量。对于具体的网络应用函数库，我觉得短期内没必要。这是因为业界内部对此就争议很大，如同 Ruby on Rails 那样的王者框架还没有出现。

我觉得对开源社区而言，最重要是两个工作。第一，是 Swift 的包管理器（Package Manager）。这个可以让我们在多个平台、不同函数库之间协同工作，大幅提高兼容性和效率；第二，是并发模型（Concurrency Model）。Go 语言之所以在服务器和云端开发这么受欢迎，就是因为并发模型做得好。**并发模型应该会集成在 Swift 5 中。**

18. 现在 Swift 在服务器端还不是那么成熟。有人说 Swift 不过是写 App 的一门语言。现在已经 3.0 版本了，大家貌似都还只是将 Swift 用来写写 iOS 应用。你怎么看?

我现在根本不担心 Swift 在服务器端最后不会成功。很多人写了几年 Swift，自以为很懂这门语言。当 Swift 具备服务器端特性的时候，苹果一定会跟大家说，你看 Swift 能做这个那个，你用其他语言来写就要麻烦得多。

现在最大的问题是大家还觉得 Swift 只是苹果自己搞出来的东西。他们觉得 Swift 不过是苹果自己的玩具，只能用在苹果自己的 iOS 系统和 Mac OS 系统上。所以我们应该加大开源和构建社区的力度。现在外行对于 Swift 的态度还可以接受，慢慢地 Swift 就会在系统开发领域追上来。

19. 大家似乎都在期待 Swift 能在网页开发上有所建树。现在网页或者网络程序开发方面，一般是多种语言混用，前端和后端可能语言逻辑完全不一样，你对此怎么看?

这可能要花很长时间，要是能取代 Javascript 那就简直了。现在 Dart 在网页开发上做的不错。我个人看好 asm.js 和 WebAssembly，它们都是通过 LLVM 编译的，跟 Swift 一样。如果这两个今后做得足够好，也许就没 Swift 什么事了。未来之事，都很难说。

而且我现在发现，Javascript 已经变成一门基础语言了。我看很多脚本语言现在都直接编译成 Javascript，Javascript 就像比特一样成为一个最基本的表达方式。我觉得五年之后，很有可能 asm.js 会一统网页端。虽然大家说 Javascript 不好 debug，但其实就算你写 C 这么成熟的语言，debug 起来依然很头疼。这也是我们为什么不在 Swift 中加入宏定义，因为那个给编译和 debug 增加了难度。

Swift 语言设计

20. Swift 好像一开始就设计得简单易懂、而同时又有很多高阶的复杂操作。经验丰富的程序员可以写出漂亮的语法糖，对编程一窍不通的小孩也可以玩转 Playground。你认为 Swift 是一门将复杂和简易融为一体的语言吗?

Swift 在这点上目前做得还不错。但我担心开源之后大量的新功能添加进来，使得 Swift 不再简单。我一直致力于让 Swift 成为一门简单易学的语言，同时又足够强大。你想我们为什么不支持内联汇编 (inline assembly support) 这样的功能，就是只有极少数极客会喜欢。以后我们也要秉持这个原则。

一个不会写 Swift 的人。打开 Playground，敲下 “print("Hello World")”，旁边就会显示出来，这点跟 python 很像，你不用去打"\n"这样的换行符号。也就是说 Swift 对于新手来说非常友好，我们可以从 Hello World 开始逐步深入，从简单慢慢过渡到复杂。

对于系统开发而言，Swift 相比 Rust，会更好的自动控制内存分配，因为我们可以借鉴开发 ARC 时的经验。你想内存分配这种底层的东西，也只有少数大牛能精通。那为什么不把 ARC 引入到底层来简化开发呢？我觉得这是 Swift 开发的另一个方向。

21. 有人说 Swift 是大杂烩，一部分借鉴 C#，一部分借鉴 Javascript，一部分借鉴 Objective-C，你是怎么看的?

Swift 确实是大杂烩。但是它并不是简单的模仿其他语言，而是借鉴，然后创造出一个伟大的语言。我们确实参考了大量其他的语言设计。比如 Haskell 很多概念就被引入到 Swift 中。Swift 中的 Protocol，就是从 Haskell 的 construct 中得到启发的。

还有其他部分长得像 Dart，亦或是借鉴了 Go 和 C#。这样做也有另一个好处，开发者拿到 Swift 的时候会有种似曾相识的感觉，这样大家也更愿意用 Swift 开发。

Swift vs. Objective-C

22. 给我个现在就学习 Swift 的理由?

这个其实无所谓。我个人不觉得 Objective-C 会短期内被取代，苹果依然支持 C 和 C++，而且放弃 Objective-C 对苹果来说有百害而无一利。你不必一定要学习 Swift，Swift 只是一门更好的语言。

说到 Swift，我们给它取这个名字就意味着我们希望这门语言非常得高效。它本身设计的目的不是让你短时间内写大量代码，而是用最少的时间、最简洁的代码来完成工作。

编程其实包括方方面面，不仅仅是写代码，还有 debug，给各种系统适配，以及其他各种事情。其实开发的时间短，找 bug 的时间一般都会很长。比如在 Objective-C 中，你会花不少时间修 unrecognized-selector error，但是 Swift 从顶层设计中就排除了这类 bug。

Swift 还有其他一些好处。比如可以对字符串使用 switch...case...语句；可以使用 functional programming；可以用 enum 和 protocol。Swift 其实是一门包罗万象的语言，菜鸟和老手写出来的 Swift 可以完全不一样，这取决于经验。

我最近发现，**很多 iOS 开发者会把 Swift 当 Objective-C 来写**，逻辑结构完全一样，只是换个语法。其实这就意味着他们没有意识到 Swift 的价值 -- 认为 Swift 不过是 Objective-C 的替代品。当开发者深究 Swift 的语法后，他们才会意识到这是一门多么高效的语言。

23. 会不会像 Objective-C 一样，在未来 Swift 添加一些动态特性?

Swift 目前没有加入动态特性的计划。很多人问为什么 Swift 不能有响应，reflection这些特性。甚至有人写博客说，“迟早有一天，苹果要重写 Swift 的所有架构”，我每次在 WWDC 前看到这些博客都会呵呵。很多人不明白什么叫动态性，也不关心我们发布的 Swift 计划表，只是不停的写博客，预测这个吐槽那个。

我个人可以明确表示，Swift 近期内没有加入动态特性的计划。凡事有轻重缓急，我们得先处理其他事情，比如并发模型，比如在系统端上的优化，比如脚本的适配。不过以后如果有时间，Swift 会加入动态特性的，前提是我们计划表里的事情都做完了。

24. 你不担心没有动态特性，很多 Objective-C 的程序员会各种不适应 Swift，然后就放弃用 Swift 了吗?

我不担心啊。Swift 本身支持 Objective-C 上的所有特性，你只需要那部分代码使用 Objective-C 兼容，然后把它们加入到 runtime 中即可。

虽然有很多人说，我就是想写纯粹的 Swift 代码，但其实我不觉得这是一种倒退。你可以使用 reflection 模型，要用这个功能你用就是了，自己设计的代码结构自己负责。在写代码这事上，从来没有非黑即白一说，我们要做最重要的事，而不是天天在推特上开听证会。Swift 核心组做的工作就是把关 Swift 开发，把这门语言导向一个正确的方向。

Swift 编程规范

25. Swift 现在好多语法糖。怎样避免写出奇怪和低效的 Swift 代码？你觉得现在 Swift 可以称得上成熟吗?

现在正是 Swift 成熟之时。Swift 1 和 Swift 2 的时候，确实语言的变化很大，大家很头疼。但是 Swift 3.0 是一个稳定成熟的版本，它真的不错。之后的工作是在 Swift 3.0 的基础上增加新的函数库或者功能，而不是修改现有的架构。

其实 Swift 开发者也在纠结语法糖太多的问题。我听说一些人出了一些 Swift 的书籍，这很好。其实我们在设计 Swift 的时候，就考虑到语法糖的问题了。比如你写代码，把所有变量都用 var，这时候编译器会提醒你对常量使用 let。这说明一点，Swift 是鼓励 immutable 数据类型的，并且 Xcode 也会自动督促你写出更规范的代码。不过目前对于“是该用 class 还是 struct？”这类比较困难的问题，编译器

还没智能到能自动检测并纠正。

26. 有些语言一开始就有设定好的语法糖和规范。为什么 Swift 没有这样，而是让开源社区去讨论？你个人对 Swift 有没有一些编程规范?

作为一个程序员，我骨子里流淌着编程规范的血液。但在 Swift 的开发过程中，我还是改变了一些固有观念。比如说，我认为所有代码代码段都应该是一个地方输入，一个地方输出。但我后来发现这样设计语言很难维护，可读性也不佳。 比如说我们设计的 guard else 语句，你一定要在末尾写上 return 之类的结束语。这就导致了一个函数有多个地方输出：你在 guard else 里 return，在其他地方也 return，不符合我原来的设想。但是如此设计会令安全性提高，因为我们把一些特殊情况给提前处理掉了。

对于空格这种格式问题，我个人倾向于空 2 格。我知道有些人喜欢空 4 格，还有人喜欢 3 格（因为他们觉得文件中不应该有 tab）。这完全是萝卜青菜各有所爱，大家对此争论不休，哪一种都有一定道理。所以我们最后也没有对 Swift 提出固定的格式要求，大家写出自己喜欢的代码就行。但是这也造成了一定程度的混乱 -- 你写的代码格式会与同事的完全不同。但是我觉得这并不会影响语言的多样性。

Go 当年强行推广了一套编程规范，结果到现在仍有争议。我们现在的工作不是做语法上面的规范，而且我们也不希望推出一套规范后大家好不买账。开源的另一个好处是，大家可以自行决定什么是好的语法规范。就算有时间我个人或者 Apple 也不会去写 Swift Style Guide。比起规范我更愿意去回答理论和语言设计上的问题。

有一件趣事我想分享，我一直担心别人会问，为什么 Swift 的函数名叫 func？而不叫 function 或者 fn？这其实颇有争议。不过现在已经是 Swift 3.0 时代了，大家这样用得很顺，我们也不会去更改了，所以争论于此没有意义。

RxSwift 以及响应式编程

27.很多开发者用 RxSwift 或者其他响应式编程。你在开发 Swift 过程中有没有仔细研究过响应式编程这些?

我已经开始关注 RxSwift 了。但是我自己没用响应式编程来开发过产品，所以我对它们的理解来自于博客。RxSwift 看起来很棒，你可以少写很多代码，而且似乎开发效率也会更高。但听说维护和测试起来也很难，有优点也有缺点。

如果我有空写一个 App 的话，我肯定回去试试 RxSwift，然后再过来发表观点。我现在不敢说”强烈推荐”，或者“强烈不推荐”之类的话。

Garbage Collection vs. ARC

28. 我们都知道 Garbage Collection 和 ARC 各有千秋。Objective-C 有 Garbage Collection，后来加入了 ARC 的机制。Swift 则是完全 ARC。你能说说为什么你们那么看好 ARC 吗?

Objective-C 最开始是基于 Libauto 系统开发的，而 Libauto 本身就有诸多限制，所以我们当时采用了 Garbage Collection。我个人觉得 ARC 完全要优于 Garbage Collection，因为后者经常在内存上回收一下我们不想回收的变量。所以我们在 Objective-C 上采用了引用计数和 ARC。

ARC 最重要的一个优势就是，它很好的处理了 final 这类参数。如果你用 Garbage Collection，比如 java 吧，final 参数就是那些不被回收一直在跑的东西，这样展开讲问题是一箩筐。我举个最简单的例子，当有个 final 变量运行在一个错误的线程上时，它会多次重跑，导致实例被不停的创建。ARC 则是从根本上解决了这个问题。

目前反对 ARC 的理由主要有两个，一是人们觉得 ARC 引入了额外的开销，因为你要维护引用计数嘛。另一个是 ARC 容易造成循环引用。

我个人要强调的是，这些毛病 Garbage Collection 也有。除此之外 Garbage Collection 还不能终止所有的线程，或者在特定的一个时间点终止一个线程。这是因为 Garbage Collection 引入了安全指针（safepoint），这同样也是一笔额外的开销。

ARC 中引用计数的开销在实际开发中影响不大。而且我们对对象的整个生命流程都有掌控，而这是 Garbage Collection 不具备的。实际上我觉得 ARC 中有些额外开销是必须的，那些不必须的开销以后也会慢慢改进的。

至于循环引用的问题。相比于你必须在具体的一行说明，retain/malloc 这个变量，然后再在后面某一行说明，release/free这个变量这种麻烦事，你只需要用 strong 或者 weak 表示你对对象的所有权，你省去了大量思考内存分配的担忧和操作，这难道不是一个巨大的进步吗?

身后之事

29.把 Swift 交给 Ted 你放心吗?

完全不用担心。

Ted 这人实力非常强。斯坦佛的博士生毕业，苹果十年工作经验，曾经以一己之力完成了 Clang 的静态分析器。Ted 在管理方面也很优秀。我有时候会突发奇想，让手下一个人或者一个组去做“我认为有意义”的项目。Ted 则是非常稳健的管理者，他总会领导组员去做最重要的事情，这就是我跟他的不同。

另外我们的小组也很强，核心团队的几个人：Doug Gregor, John McCall, Joe Groff, Dave Abrahams。这几个人都是极其优秀的极客。Swift 其他团队的工程师也很给力。有他们在，没有任何理由 Swift 不会成功。

30. 你为什么去做电动车?

首先我个人非常喜欢车。但我又懒得自己老是去加油啊、开车，我更喜欢一种更可靠的方式，最好我自己啥也不用做，车子就可以把我送到目的地。我也不需要担心维护啊什么的。我其实是特斯拉最早的一批客户，我觉得特斯拉驾驶起来很开心。

不过我重来没想过我会去一家汽车公司任职，因为我觉得我是个程序员，这跟汽车有啥关系？不过特斯拉让我去做自动驾驶系统，这个就很对我胃口了。因为这也是世界级的难题，我想尝试挑战一下。

补充

Chris Lattner 提到的语言

- Go

主页：http://golang.org/

Google发布的开源语言。编译速度媲美 C，安全性有过之而无不及。学习曲线也与 Java 类似，比较简单。目前主要用于网络服务器，存储系统，和数据库中。

- Dart

主页：https://www.dartlang.org/

Google 开发的语言。基于类，只能单一继承，风格上偏向 C。目标在于成为下一代网络开发语言。目前 Google 正在尝试用 Dart 开发 Android 应用，达到去 java 化的目标。

- Haskell

主页：https://www.haskell.org/

函数式编程语言，支持惰性求值、模式匹配、列表内包、类型类和类型多态。用户很少，普遍认为难学难用。主要用于金融系统及安全性和性能要求抛高的产品。

参考链接

音频：ACCIDENTAL TECH PODCAST 205

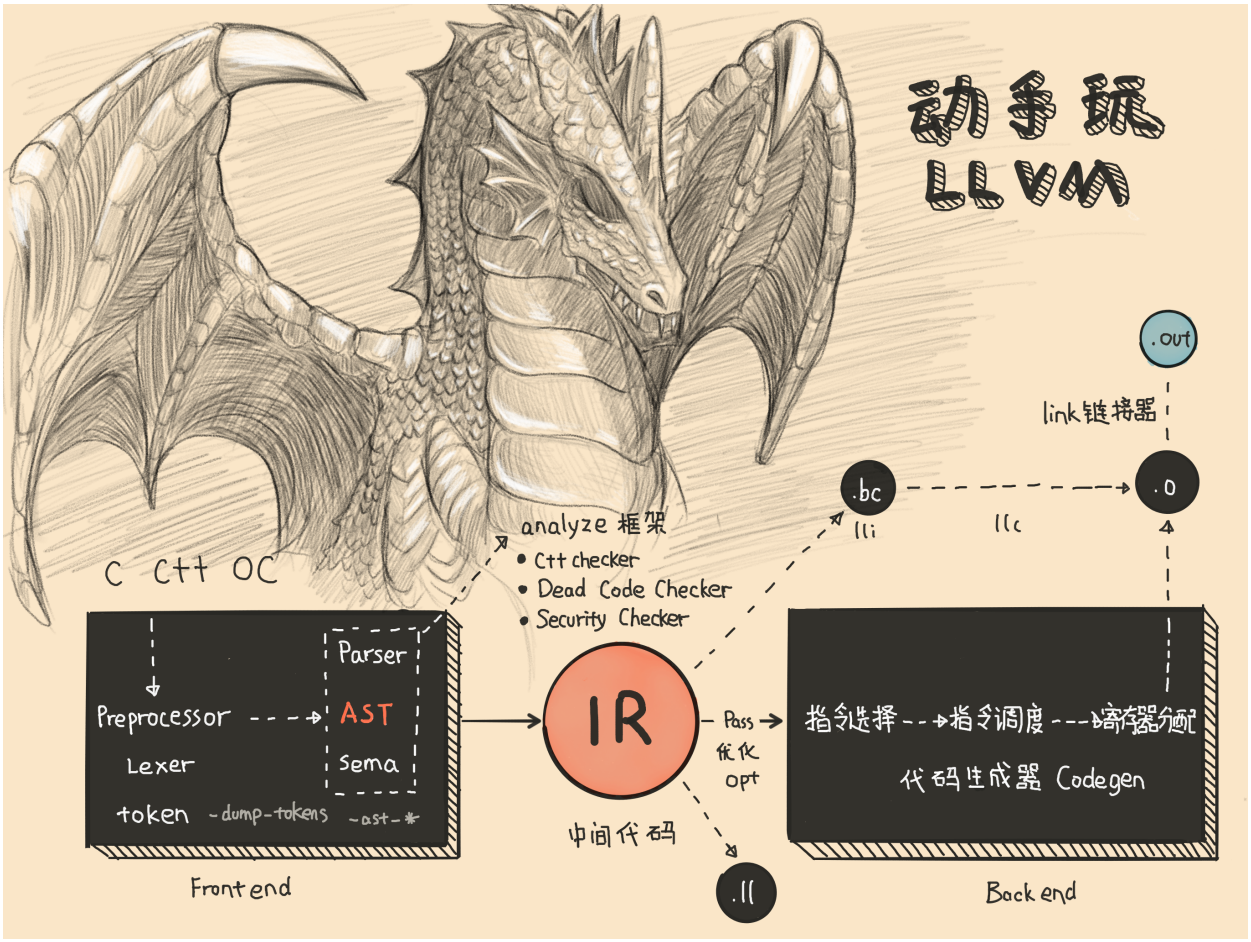
原文：PEOPLE DON'T USE THE WEIRD PARTS

Swift 3 将不包含稳定的ABI

转载 为什么业界很少使用 Haskell?

动手玩 LLVM

作者：戴铭



介绍

LLVM 资料非常少，接触时难免会有种无从下手的感觉，现在很多相关文章都偏向理论，以前我写了篇《 深入剖析 iOS 编译 Clang / LLVM 》可以作为一个最基础深入前的理论入门，最近在 segmentfault 的一个直播（地址： https://segmentfault.com/l/1500000008514518）对先前文章里静态分析和编译器优化部分做了更加详细的解读。光看不练确实没啥用，本文希望能够通过一些具体的比如如何编译 LLVM 的工具链，使用工具链，熟悉 LLVM 各种接口库等来带大家进入 LLVM 的世界，能成为一名能够熟练运用倚天屠龙剑的屠龙战士。

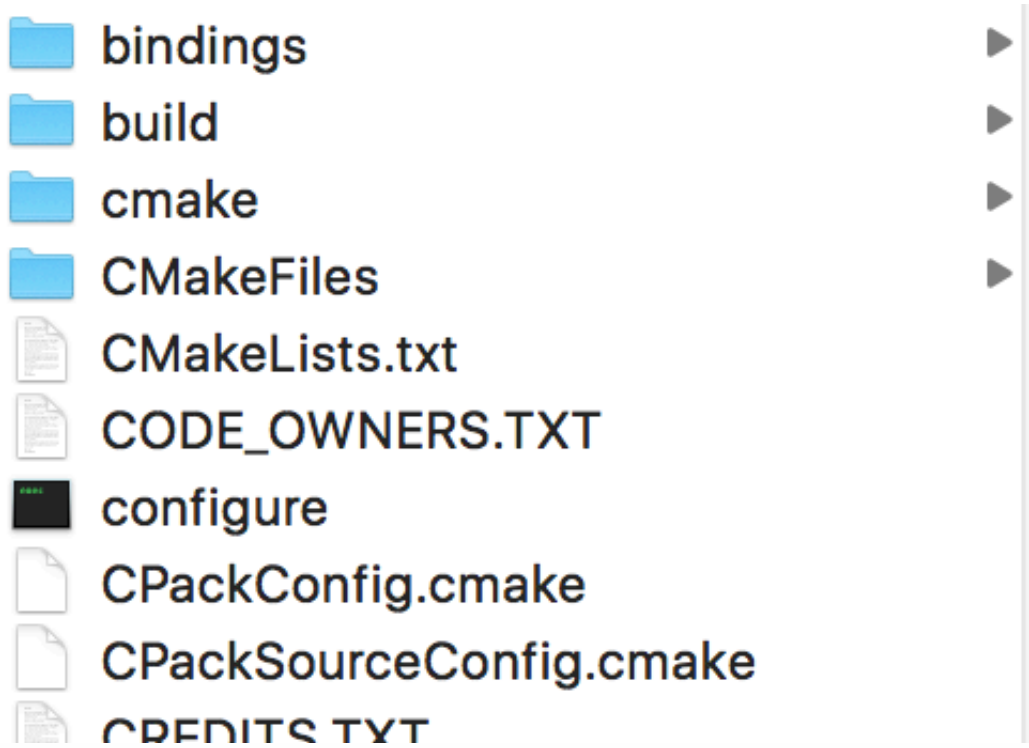
LLVM 工具链

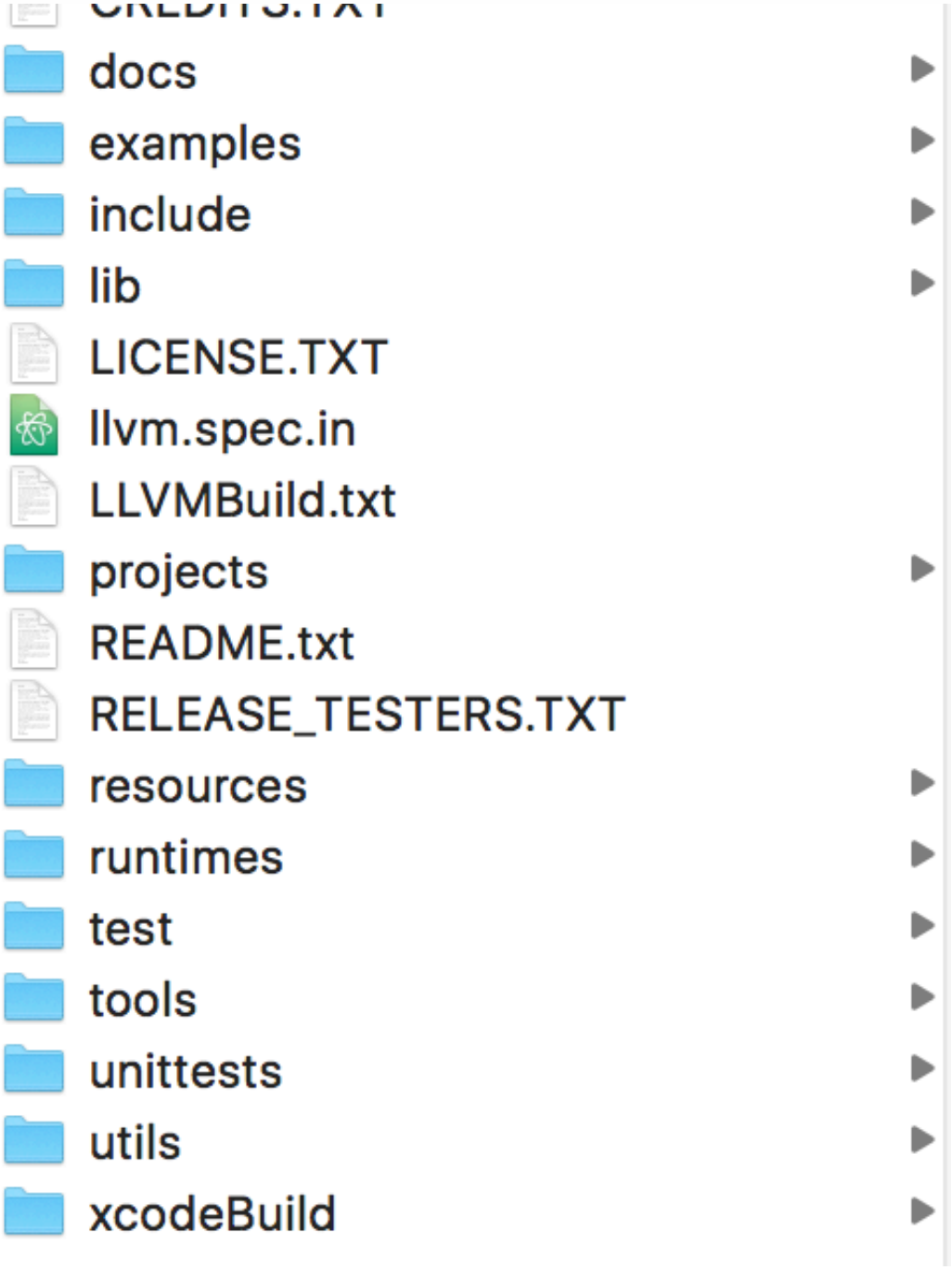
获取 LLVM

```
#先下载 LLVM
svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm

#在 LLVM 的 tools 目录下下载 Clang
cd llvm/tools
svn co http://llvm.org/svn/llvm-project/cfe/trunk clang

#在 LLVM 的 projects 目录下下载 compiler-rt, libcxx, libcxxabi
cd ../projects
svn co http://llvm.org/svn/llvm-project/compiler-rt/trunk compiler-rt
svn co http://llvm.org/svn/llvm-project/libcxx/trunk libcxx
svn co http://llvm.org/svn/llvm-project/libcxxabi/trunk libcxxabi

#在 Clang 的 tools 下安装 extra 工具
cd ../tools/clang/tools
svn co http://llvm.org/svn/llvm-project/clang-tools-extra/trunk extra
```

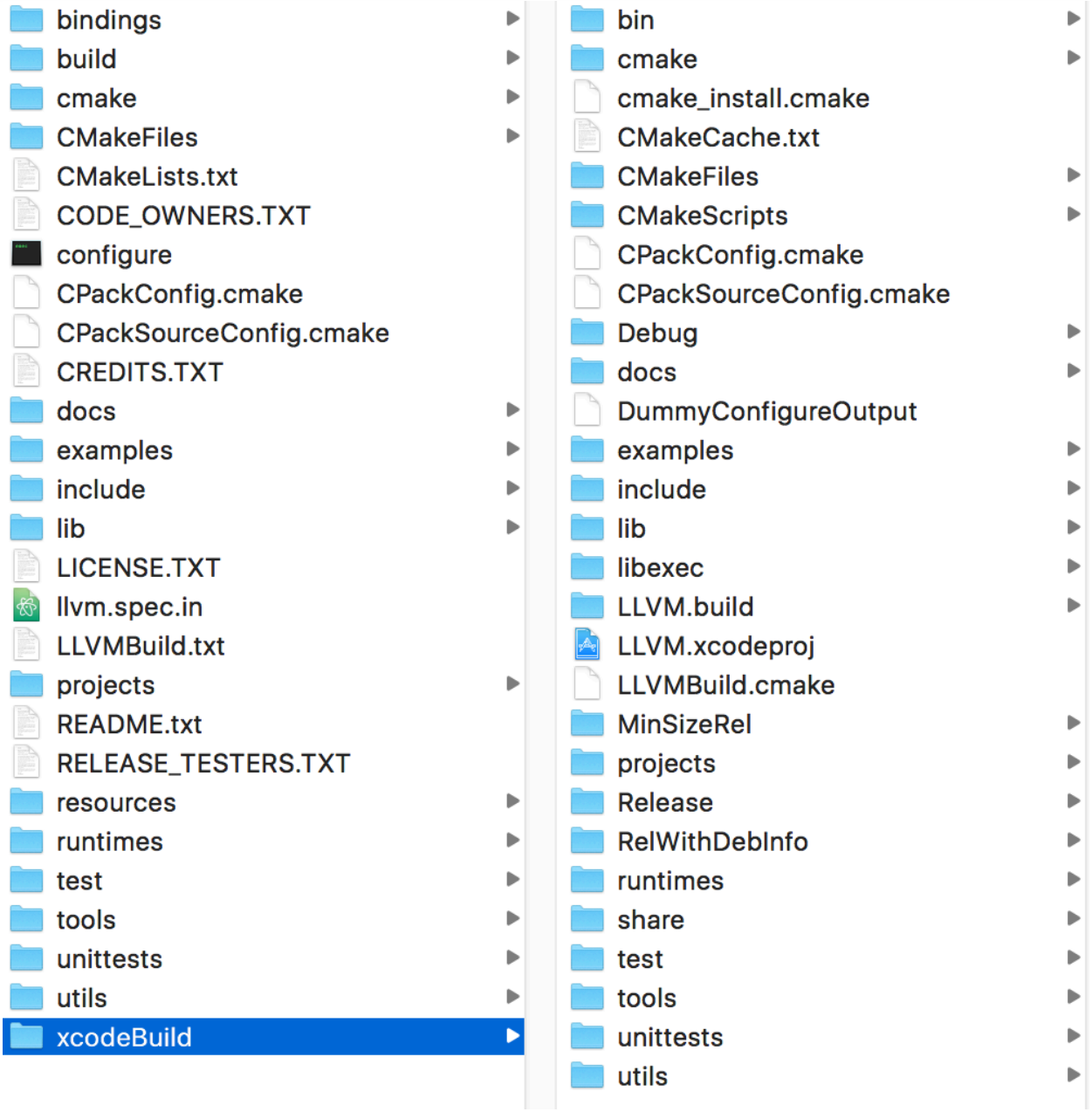
编译 LLVM

```
brew install gcc
brew install cmake
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release -DBUILD_SHARED_LIBS=ON -
DLLVM_TARGETS_TO_BUILD="AArch64;X86" -G "Unix Makefiles" ..
make j8
#安装
make install
#如果找不到标准库，Xcode 需要安装 xcode-select --install
```

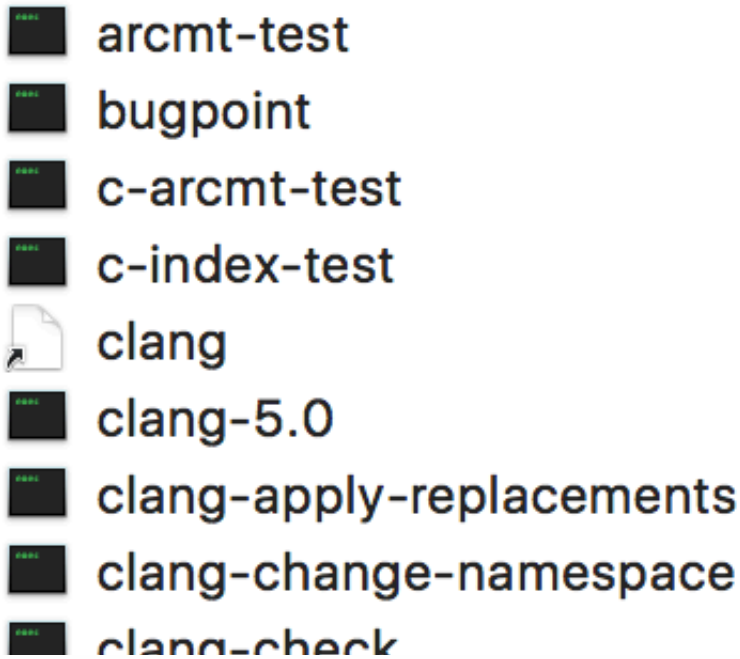




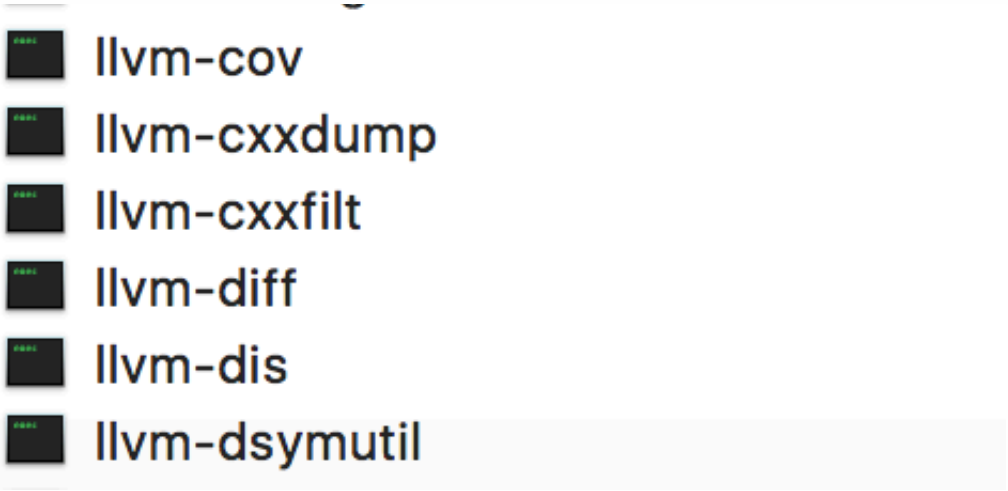

```
#如果希望是 xcodeproject 方式 build 可以使用 -GXcode
mkdir xcodeBuild
cd xcodeBuild
cmake -GXcode /path/to/llvm/source
```



在 bin 下存放着工具链，有了这些工具链就能够完成源码编译了。



clang-check
clang-cl
clang-cpp
clang-format
clang-import-test
clang-include-fixer
clang-move
clang-offload-bundler
clang-query
clang-rename
clang-reorder-fields
clang-tblgen
clang-tidy
clang++
count
diagtool
FileCheck
find-all-symbols
llc
lli
lli-child-target
llvm-ar
llvm-as
llvm-bcanalyzer
llvm-c-test
llvm-cat
llvm-config



LLVM 源码工程目录介绍

- llvm_examples_ - 使用 LLVM IR 和 JIT 的例子。
- llvm_include_ - 导出的头文件。
- llvm_lib_ - 主要源文件都在这里。
- llvm_project_ - 创建自己基于 LLVM 的项目的目录。
- llvm_test_ - 基于 LLVM 的回归测试，健全检察。
- llvm_suite_ - 正确性，性能和基准测试套件。
- llvm_tools_ - 基于 lib 构建的可以执行文件，用户通过这些程序进行交互，-help 可以查看各个工具详细使用。
- llvm_utils_ - LLVM 源代码的实用工具，比如，查找 LLC 和 LLI 生成代码差异工具， Vim 或 Emacs 的语法高亮工具等。

lib 目录介绍

- llvm_lib_IR/ - 核心类比如 Instruction 和 BasicBlock。
- llvm_lib_AsmParser/ - 汇编语言解析器。
- llvm_lib_Bitcode/ - 读取和写入字节码
- llvm_lib_Analysis/ - 各种对程序的分析，比如 Call Graphs，Induction Variables，Natural Loop Identification 等等。
- llvm_lib_Transforms/ - IR-to-IR 程序的变换。
- llvm_lib_Target/ - 对像 X86 这样机器的描述。
- llvm_lib_CodeGen/ - 主要是代码生成，指令选择器，指令调度和寄存器分配。
- llvm_lib_ExecutionEngine/ - 在解释执行和JIT编译场景能够直接在运行时执行字节码的库。

工具链命令介绍

基本命令

- llvm-as - 汇编器，将 .ll 汇编成字节码。
- llvm-dis - 反汇编器，将字节码编成可读的 .ll 文件。
- opt - 字节码优化器。
- llc - 静态编译器，将字节码编译成汇编代码。
- lli - 直接执行 LLVM 字节码。
- llvm-link - 字节码链接器，可以把多个字节码文件链接成一个。
- llvm-ar - 字节码文件打包器。

- llvm-lib - LLVM lib.exe 兼容库工具。
- llvm-nm - 列出字节码和符号表。
- llvm-config - 打印 LLVM 编译选项。
- llvm-diff - 对两个进行比较。
- llvm-cov - 输出 coverage infomation。
- llvm-profdata - Profile 数据工具。
- llvm-stress - 生成随机 .ll 文件。
- llvm-symbolizer - 地址对应源码位置，定位错误。
- llvm-dwarfdump - 打印 DWARF。

调试工具

- bugpoint - 自动测试案例工具
- llvm-extract - 从一个 LLVM 的模块里提取一个函数。
- llvm-bcanalyzer - LLVM 字节码分析器。

开发工具

- FileCheck - 灵活的模式匹配文件验证器。
- tblgen - C++ 代码生成器。
- lit - LLVM 集成测试器。
- llvm-build - LLVM 构建工程时需要的工具。
- llvm-readobj - LLVM Object 结构查看器。

Driver

动手玩的话，特别是想要使用这些工具链之前最好先了解我们和 LLVM 交互的实现。那么这部分就介绍下 LLVM 里的 Driver。

Driver 是 Clang 面对用户的接口，用来解析 Option 设置，判断决定调用的工具链，最终完成整个编译过程。

相关源代码在这里：clang_tools_driver/driver.cpp

整个 Driver 源码的入口函数就是 driver.cpp 里的 main() 函数。从这里可以作为入口看看整个 driver 是如何工作的，这样更利于我们以后轻松动手驾驭 LLVM。

```
int main(int argc_, const char **argv_) {
  llvm::sys::PrintStackTraceOnErrorSignal(argv_[0]);
  llvm::PrettyStackTraceProgram X(argc_, argv_);
  llvm::llvm_shutdown_obj Y; // Call llvm_shutdown() on exit.

  if (llvm::sys::Process::FixupStandardFileDescriptors())
    return 1;

  SmallVector<const char *, 256> argv;
  llvm::SpecificBumpPtrAllocator<char> ArgAllocator;
  std::error_code EC = llvm::sys::Process::GetArgumentVector(
      argv, llvm::makeArrayRef(argv_, argc_), ArgAllocator);
  if (EC) {
    llvm::errs() << "error: couldn't get arguments: " << EC.message() <<
'\n';
    return 1;
  }

  llvm::InitializeAllTargets();
  std::string ProgName = argv[0];
  std::pair<std::string, std::string> TargetAndMode =
      ToolChain::getTargetAndModeFromProgramName(ProgName);

  llvm::BumpPtrAllocator A;
  llvm::StringSaver Saver(A);

    //省略
  ...

  // If we have multiple failing commands, we return the result of the first
  // failing command.
  return Res;
}
```

Driver 的工作流程图

在 driver.cpp 的 main 函数里有 Driver 的初始化。我们来看看和 driver 相关的代码

```
  Driver TheDriver(Path, llvm::sys::getDefaultTargetTriple(), Diags);
  SetInstallDir(argv, TheDriver, CanonicalPrefixes);

  insertTargetAndModeArgs(TargetAndMode.first, TargetAndMode.second, argv,
                          SavedStrings);

  SetBackdoorDriverOutputsFromEnvVars(TheDriver);

  std::unique_ptr<Compilation> C(TheDriver.BuildCompilation(argv));
  int Res = 0;
  SmallVector<std::pair<int, const Command *>, 4> FailingCommands;
  if (C.get())
    Res = TheDriver.ExecuteCompilation(*C, FailingCommands);

  // Force a crash to test the diagnostics.
  if (::getenv("FORCE_CLANG_DIAGNOSTICS_CRASH")) {
    Diags.Report(diag::err_drv_force_crash) <<
"FORCE_CLANG_DIAGNOSTICS_CRASH";

    // Pretend that every command failed.
    FailingCommands.clear();
    for (const auto &J : C->getJobs())
      if (const Command *C = dyn_cast<Command>(&J))
        FailingCommands.push_back(std::make_pair(-1, C));
  }

  for (const auto &P : FailingCommands) {
    int CommandRes = P.first;
    const Command *FailingCommand = P.second;
    if (!Res)
      Res = CommandRes;

    // If result status is < 0, then the driver command signalled an error.
    // If result status is 70, then the driver command reported a fatal
error.
    // On Windows, abort will return an exit code of 3.  In these cases,
    // generate additional diagnostic information if possible.
    bool DiagnoseCrash = CommandRes < 0 || CommandRes == 70;
#ifdef LLVM_ON_WIN32
    DiagnoseCrash |= CommandRes == 3;
#endif
    if (DiagnoseCrash) {
      TheDriver.generateCompilationDiagnostics(*C, *FailingCommand);
      break;
    }
  }
```

可以看到初始化 Driver 后 driver 会调用 BuildCompilation 生成 Compilation。Compilation 字面意思是合集的意思，通过 driver.cpp 的 include 可以看到

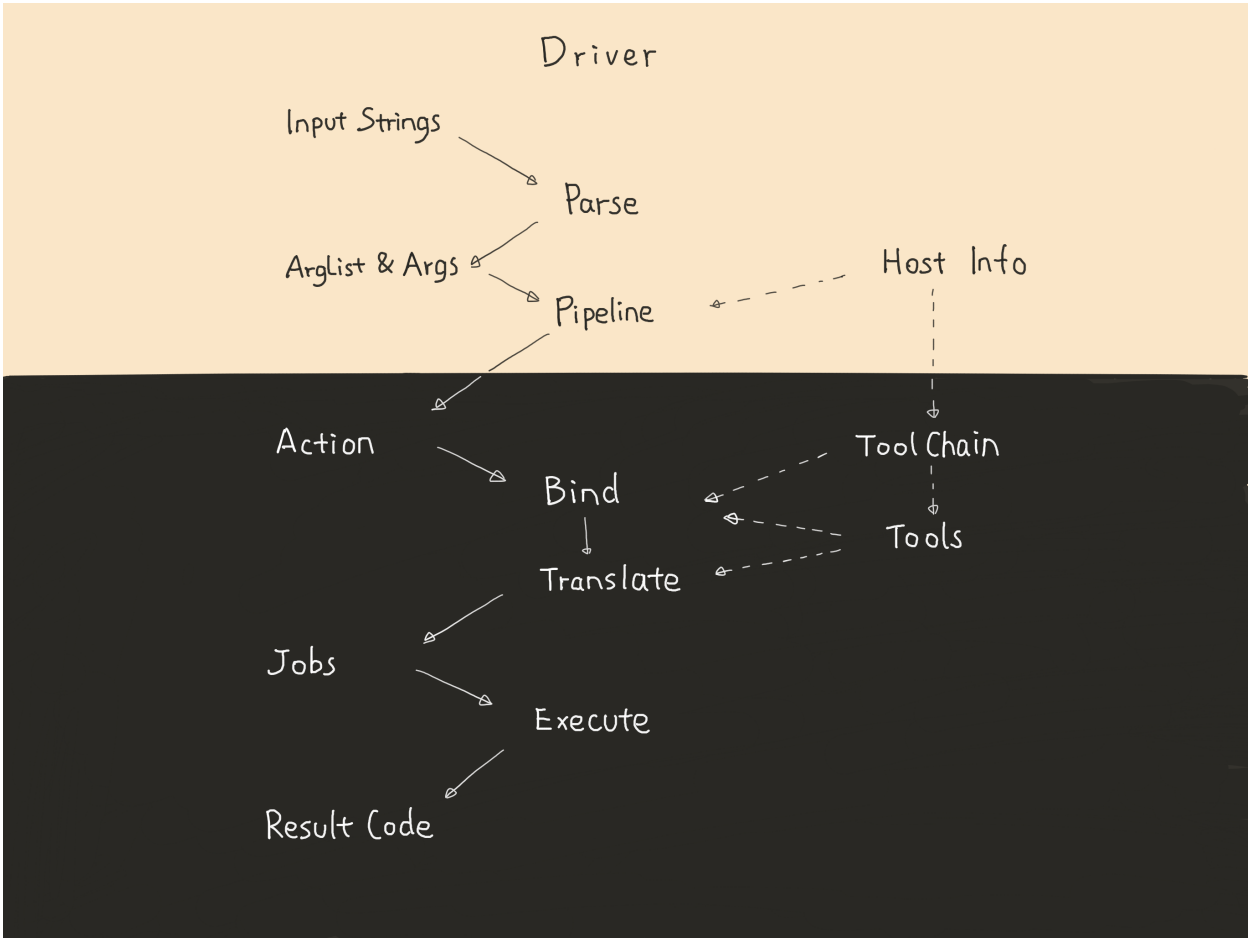
```
#include "clang/Driver/Compilation.h"
```

根据此路径可以细看下 Compilation 这个为了 driver 设置的一组任务的类。通过这个类我们提取里面这个阶段比较关键的几个信息出来

```
class Compilation {
    /// The original (untranslated) input argument list.
  llvm::opt::InputArgList *Args;

  /// The driver translated arguments. Note that toolchains may perform
their
  /// own argument translation.
  llvm::opt::DerivedArgList *TranslatedArgs;
  /// The driver we were created by.
  const Driver &TheDriver;

  /// The default tool chain.
  const ToolChain &DefaultToolChain;
 ...
  /// The list of actions.  This is maintained and modified by consumers,
via
  /// getActions().
  ActionList Actions;

  /// The root list of jobs.
  JobList Jobs;
    ...
public:
    ...
  const Driver &getDriver() const { return TheDriver; }

  const ToolChain &getDefaultToolChain() const { return DefaultToolChain; }
    ...
  ActionList &getActions() { return Actions; }
  const ActionList &getActions() const { return Actions; }
    ...
  JobList &getJobs() { return Jobs; }
  const JobList &getJobs() const { return Jobs; }

  void addCommand(std::unique_ptr<Command> C) { Jobs.addJob(std::move(C)); }
    ...
  /// ExecuteCommand - Execute an actual command.
  ///
  /// \param FailingCommand - For non-zero results, this will be set to the
  /// Command which failed, if any.
  /// \return The result code of the subprocess.
```

```
  int ExecuteCommand(const Command &C, const Command *&FailingCommand)
const;

  /// ExecuteJob - Execute a single job.
  ///
  /// \param FailingCommands - For non-zero results, this will be a vector
of
  /// failing commands and their associated result code.
  void ExecuteJobs(
      const JobList &Jobs,
      SmallVectorImpl<std::pair<int, const Command *>> &FailingCommands)
const;
    ...
};
```

通过这些关键定义再结合 BuildCompilation 函数的实现可以看出这个 Driver 的流程是按照 ArgList - Actions - Jobs 来的，完整的图如下：



Parse

看完完整的 Driver 流程后，我们就先从 Parse 开始说起。

Parse 是解析选项，对应的代码在 ParseArgStrings 这个函数里。

下面通过执行一个试试，比如 clang -### main.c -ITheOptionWeAdd

```
➜  llvmdemo clang -### main.c -ITheOptionWeAdd
clang version 5.0.0 (trunk 294089)
Target: x86_64-apple-darwin16.4.0
Thread model: posix
InstalledDir: /usr/local/bin
 "/usr/local/bin/clang-5.0" "-cc1" "-triple" "x86_64-apple-macosx10.12.0" "-Wdeprecated-ob
jc-isa-usage" "-Werror=deprecated-objc-isa-usage" "-emit-obj" "-mrelax-all" "-disable-free
" "-main-file-name" "main.c" "-mrelocation-model" "pic" "-pic-level" "2" "-mthread-model"
"posix" "-mdisable-fp-elim" "-masm-verbose" "-munwind-tables" "-target-cpu" "penryn" "-tar
get-linker-version" "274.2" "-dwarf-column-info" "-debugger-tuning=lldb" "-resource-dir" "
/usr/local/bin/../lib/clang/5.0.0" "-I" "TheOptionWeAdd" "-fdebug-compilation-dir" "/Users
/didi/Downloads/llvmdemo" "-ferror-limit" "19" "-fmessage-length" "90" "-stack-protector"
"1" "-fblocks" "-fobjc-runtime=macosx-10.12.0" "-fencode-extended-block-signature" "-fmax-
type-align=16" "-fdiagnostics-show-option" "-fcolor-diagnostics" "-o" "/var/folders/r9/35q
9g3d56_d9g0v59w9x2l9w0000gn/T/main-85975b.o" "-x" "c" "main.c"
 "/usr/bin/ld" "-demangle" "-lto_library" "/usr/local/lib/libLTO.dylib" "-no_deduplicate"
"-dynamic" "-arch" "x86_64" "-macosx_version_min" "10.12.0" "-o" "a.out" "/var/folders/r9/
35q9g3d56_d9g0v59w9x2l9w0000gn/T/main-85975b.o" "-lSystem" "/usr/local/bin/../lib/clang/5.
0.0/lib/darwin/libclang_rt.osx.a"
```

这里的 -I 是 Clang 支持的，在 Clang 里是 Option 类，Clang 会对这些 Option 专门的进行解析，使用一种 DSL 语言将其转成 .tb 文件后使用 table-gen 转成 C++ 语言和其它代码一起进行编译。

Driver 层会解析我们传入的 -I Option 参数。

-x 后加个 c 表示是对 c 语言进行编译，Clang Driver 通过文件的后缀 .c 来自动加上这个 参数的。如果是 c++ 语言，仅仅通过在 -x 后添加 cpp 编译还是会出错的。

```
clang -x c++ main.cpp
```

```
➜  llvmdemo clang -x c++ main.cpp
Undefined symbols for architecture x86_64:
  "std::__1::__basic_string_common<true>::__throw_length_error() const", referenced from:
      std::__1::ostreambuf_iterator<char, std::__1::char_traits<char> > std::__1::__pad_an
d_output<char, std::__1::char_traits<char> >(std::__1::ostreambuf_iterator<char, std::__1:
:char_traits<char> >, char const*, char const*, char const*, std::__1::ios_base&, char) in
 main-e1e673.o
  "std::__1::locale::use_facet(std::__1::locale::id&) const", referenced from:
      std::__1::basic_ostream<char, std::__1::char_traits<char> >& std::__1::endl<char, st
d::__1::char_traits<char> >(std::__1::basic_ostream<char, std::__1::char_traits<char> >&)
in main-e1e673.o
      std::__1::basic_ostream<char, std::__1::char_traits<char> >& std::__1::__put_charact
er_sequence<char, std::__1::char_traits<char> >(std::__1::basic_ostream<char, std::__1::ch
ar_traits<char> >&, char const*, unsigned long) in main-e1e673.o
  "std::__1::ios_base::getloc() const", referenced from:
      std::__1::basic_ostream<char, std::__1::char_traits<char> >& std::__1::endl<char, st
d::__1::char_traits<char> >(std::__1::basic_ostream<char, std::__1::char_traits<char> >&)
in main-e1e673.o
      std::__1::basic_ostream<char, std::__1::char_traits<char> >& std::__1::__put_charact
er_sequence<char, std::__1::char_traits<char> >(std::__1::basic_ostream<char, std::__1::ch
ar_traits<char> >&, char const*, unsigned long) in main-e1e673.o
  "std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char> >::
~basic_string()", referenced from:
      std::__1::ostreambuf_iterator<char, std::__1::char_traits<char> > std::__1::__pad_an
d_output<char, std::__1::char_traits<char> >(std::__1::ostreambuf_iterator<char, std::__1:
:char_traits<char> >, char const*, char const*, char const*, std::__1::ios_base&, char) in
 main-e1e673.o
  "std::__1::basic_ostream<char, std::__1::char_traits<char> >::put(char)", referenced fro
m:
      std::__1::basic_ostream<char, std::__1::char_traits<char> >& std::__1::endl<char, st
d::__1::char_traits<char> >(std::__1::basic_ostream<char, std::__1::char_traits<char> >&)
in main-e1e673.o
```

通过报错信息可以看出一些链接错误

因为需要链接 C++ 标准库，所以加上参数 -lc++ 就可以了

```
clang -x c++ -lc++ main.cpp
```

那么 clang++ 和 clang 命令的区别就在于会加载 C++ 库，其实 clang++ 最终还是会调用 Clang，那么手动指定加载库就好了何必还要多个 clang++ 命令呢，这主要是为了能够在这个命令里去加载更多的库，除了标准库以外，还有些非 C++ 标准库，辅助库等等。这样只要是 C++ 的程序用 clang++ 够了。

只有加上 -cc1 这个 option 才能进入到 Clang driver 比如 emit-obj 这个 option 就需要先加上 -cc1。

这点可以通过 driver.cpp 源码来看，在 main() 函数里可以看到在做了些多平台的兼容处理后就开始进行对入参判断第一个是不是 -cc1。

```
if (MarkEOLs && argv.size() > 1 && StringRef(argv[1]).startswith("-cc1"))
    MarkEOLs = false;
  llvm::cl::ExpandResponseFiles(Saver, Tokenizer, argv, MarkEOLs);

  // 处理 -cc1 集成工具
  auto FirstArg = std::find_if(argv.begin() + 1, argv.end(),
                               [](const char *A) { return A != nullptr; });
  if (FirstArg != argv.end() && StringRef(*FirstArg).startswith("-cc1")) {
    // 如果 -cc1 来自 response file, 移除 EOL sentinels
    if (MarkEOLs) {
      auto newEnd = std::remove(argv.begin(), argv.end(), nullptr);
      argv.resize(newEnd - argv.begin());
    }
    return ExecuteCC1Tool(argv, argv[1] + 4);
  }
```

如果是 -cc1 的话会调用 ExecuteCC1Tool 这个函数，先看看这个函数

```
static int ExecuteCC1Tool(ArrayRef<const char *> argv, StringRef Tool) {
  void *GetExecutablePathVP = (void *)(intptr_t) GetExecutablePath;
  if (Tool == "")
    return cc1_main(argv.slice(2), argv[0], GetExecutablePathVP);
  if (Tool == "as")
    return cc1as_main(argv.slice(2), argv[0], GetExecutablePathVP);

  // 拒绝未知工具
  llvm::errs() << "error: unknown integrated tool '" << Tool << "'\n";
  return 1;
}
```

最终的执行会执行 cc1-main 或者 cc1as_main 。这两个函数分别在 driver.cpp 同级目录里的 cc1_main.cpp 和 cc1as_main.cpp 中。

下面看看有哪些解析 Args 的方法

- ParseAnalyzerArgs - 解析出静态分析器 option
- ParseMigratorArgs - 解析 Migrator option
- ParseDependencyOutputArgs - 解析依赖输出 option

- ParseCommentArgs - 解析注释 option
- ParseFileSystemArgs - 解析文件系统 option
- ParseFrontendArgs - 解析前端 option
- ParseTargetArgs - 解析目标 option
- ParseCodeGenArgs - 解析 CodeGen 相关的 option
- ParseHeaderSearchArgs - 解析 HeaderSearch 对象相关初始化相关的 option
- parseSanitizerKinds - 解析 Sanitizer Kinds
- ParsePreprocessorArgs - 解析预处理的 option
- ParsePreprocessorOutputArgs - 解析预处理输出的 option

Pipeline

Pipeline 这里可以添加 -ccc-print-phases 看到进入 Pipeline 以后的事情。

```
➜  llvmdemo clang -ccc-print-phases main.c
0: input, "main.c", c
1: preprocessor, {0}, cpp-output
2: compiler, {1}, ir
3: backend, {2}, assembler
4: assembler, {3}, object
5: linker, {4}, image
6: bind-arch, "x86_64", {5}, image
➜  llvmdemo
```

这些如 -ccc-print-phases 这样的 option 在编译时会生成.inc 这样的 C++ TableGen 文件。在 Options.td 可以看到全部的 option 定义。

在 Clang 的 Pipeline 中很多实际行为都有对应的 Action，比如 preprocessor 时提供文件的 InputAction 和用于绑定机器架构的 BindArchAction。

使用 clang main.c -arch i386 -arch x86_64 -o main 然后 file main 能够看到这时 BindArchAction 这个 Action 起到了作用，编译链接了两次同时创建了一个库既能够支持32位也能够支持64位用 lipo 打包。

```
➜  llvmdemo clang main.c -arch i386 -arch x86_64 -o main
➜  llvmdemo file main
main: Mach-O universal binary with 2 architectures: [i386: Mach-O executable i386] [x86_64
: Mach-O 64-bit executable x86_64]
main (for architecture i386):   Mach-O executable i386
main (for architecture x86_64): Mach-O 64-bit executable x86_64
➜  llvmdemo
```

Action

```
/// BuildActions - Construct the list of actions to perform for the
  /// given arguments, which are only done for a single architecture.
  ///
  /// \param C - The compilation that is being built.
  /// \param Args - The input arguments.
  /// \param Actions - The list to store the resulting actions onto.
  void BuildActions(Compilation &C, llvm::opt::DerivedArgList &Args,
                    const InputList &Inputs, ActionList &Actions) const;

  /// BuildUniversalActions - Construct the list of actions to perform
  /// for the given arguments, which may require a universal build.
  ///
  /// \param C - The compilation that is being built.
  /// \param TC - The default host tool chain.
  void BuildUniversalActions(Compilation &C, const ToolChain &TC,
                             const InputList &BAInputs) const;
```

上面两个方法中 BuildUniversalActions 最后也会走 BuildActions。BuildActions 了，进入这个方法

```
void Driver::BuildActions(Compilation &C, DerivedArgList &Args,
                          const InputList &Inputs, ActionList &Actions)
const {
  llvm::PrettyStackTraceString CrashInfo("Building compilation actions");

  if (!SuppressMissingInputWarning && Inputs.empty()) {
    Diag(clang::diag::err_drv_no_input_files);
    return;
  }

  Arg *FinalPhaseArg;
  phases::ID FinalPhase = getFinalPhase(Args, &FinalPhaseArg);
```

接着跟 getFinalPhase 这个方法。

```
// -{E,EP,P,M,MM} only run the preprocessor.
  if (CCCIsCPP() || (PhaseArg = DAL.getLastArg(options::OPT_E)) ||
      (PhaseArg = DAL.getLastArg(options::OPT__SLASH_EP)) ||
      (PhaseArg = DAL.getLastArg(options::OPT_M, options::OPT_MM)) ||
      (PhaseArg = DAL.getLastArg(options::OPT__SLASH_P))) {
    FinalPhase = phases::Preprocess;

    // -{fsyntax-only,-analyze,emit-ast} only run up to the compiler.
  } else if ((PhaseArg = DAL.getLastArg(options::OPT_fsyntax_only)) ||
             (PhaseArg = DAL.getLastArg(options::OPT_module_file_info)) ||
             (PhaseArg = DAL.getLastArg(options::OPT_verify_pch)) ||
             (PhaseArg = DAL.getLastArg(options::OPT_rewrite_objc)) ||
             (PhaseArg = DAL.getLastArg(options::OPT_rewrite_legacy_objc))
||
             (PhaseArg = DAL.getLastArg(options::OPT__migrate)) ||
             (PhaseArg = DAL.getLastArg(options::OPT__analyze,
                                        options::OPT__analyze_auto)) ||
             (PhaseArg = DAL.getLastArg(options::OPT_emit_ast))) {
    FinalPhase = phases::Compile;

    // -S only runs up to the backend.
  } else if ((PhaseArg = DAL.getLastArg(options::OPT_S))) {
    FinalPhase = phases::Backend;

    // -c compilation only runs up to the assembler.
  } else if ((PhaseArg = DAL.getLastArg(options::OPT_c))) {
    FinalPhase = phases::Assemble;

    // Otherwise do everything.
  } else
    FinalPhase = phases::Link;
```

看完这段代码就会发现其实每次的 option 都会完整的走一遍从预处理，静态分析，backend 再到汇编的过程。

下面列下一些编译器的前端 Action，大家可以一个个用着玩。

- InitOnlyAction - 只做前端初始化，编译器 option 是 -init-only
- PreprocessOnlyAction - 只做预处理，不输出，编译器的 option 是 -Eonly
- PrintPreprocessedAction - 做预处理，子选项还包括-P、-C、-dM、-dD 具体可以查看 PreprocessorOutputOptions 这个类，编译器 option 是 -E
- RewriteIncludesAction - 预处理
- DumpTokensAction - 打印token，option 是 -dump-tokens
- DumpRawTokensAction - 输出原始tokens，包括空格符，option 是 -dump-raw-tokens
- RewriteMacrosAction - 处理并扩展宏定义，对应的 option 是 -rewrite-macros
- HTMLPrintAction - 生成高亮的代码网页，对应的 option 是 -emit-html
- DeclContextPrintAction - 打印声明，option 对应的是 -print-decl-contexts
- ASTDeclListAction - 打印 AST 节点，option 是 -ast-list
- ASTDumpAction - 打印 AST 详细信息，对应 option 是 -ast-dump

- ASTViewAction - 生成 AST dot 文件，能够通过 Graphviz 来查看图形语法树。 option 是 -ast-view
- AnalysisAction - 运行静态分析引擎，option 是 -analyze
- EmitLLVMAction - 生成可读的 IR 中间语言文件，对应的 option 是 -emit-llvm
- EmitBCAction - 生成 IR Bitcode 文件，option 是 -emit-llvm-bc
- MigrateSourceAction - 代码迁移，option 是 -migrate

Bind

Bind 主要是与工具链 ToolChain 交互
根据创建的那些 Action，在 Action 执行时 Bind 来提供使用哪些工具，比如生成汇编时是使用内嵌的还是 GNU 的，还是其它的呢，这个就是由 Bind 来决定的，具体使用的工具有各个架构，平台，系统的 ToolChain 来决定。

通过 clang -ccc-print-bindings main.c -o main 来看看 Bind 的结果

```
➜  llvmdemo clang -ccc-print-bindings main.c -o main
# "x86_64-apple-darwin16.4.0" - "clang", inputs: ["main.c"], output: "/var/folders/r9/35q9g3d56_d9g0v59w9x2l9w0000gn/T/main-3700f1.o"
# "x86_64-apple-darwin16.4.0" - "darwin::Linker", inputs: ["/var/folders/r9/35q9g3d56_d9g0v59w9x2l9w0000gn/T/main-3700f1.o"], output: "main"
➜  llvmdemo
```

可以看到编译选择的是 clang，链接选择的是 darwin::Linker，但是在链接时前没有汇编器的过程，这个就是 Bind 起了作用，它会根据不同的平台来决定选择什么工具，因为是在 Mac 系统里 Bind 就会决定使用 integrated-as 这个内置汇编器。那么如何在不用内置汇编器呢。可以使用 -fno-integrated-as 这个 option。

```
➜  llvmdemo clang -ccc-print-bindings main.c -fno-integrated-as
# "x86_64-apple-darwin16.4.0" - "clang", inputs: ["main.c"], output: "/var/folders/r9/35q9g3d56_d9g0v59w9x2l9w0000gn/T/main-5abd20.s"
# "x86_64-apple-darwin16.4.0" - "darwin::Assembler", inputs: ["/var/folders/r9/35q9g3d56_d9g0v59w9x2l9w0000gn/T/main-5abd20.s"], output: "/var/folders/r9/35q9g3d56_d9g0v59w9x2l9w0000gn/T/main-900eb1.o"
# "x86_64-apple-darwin16.4.0" - "darwin::Linker", inputs: ["/var/folders/r9/35q9g3d56_d9g0v59w9x2l9w0000gn/T/main-900eb1.o"], output: "a.out"
➜  llvmdemo
```

Translate

Translate 就是把相关的参数对应到不同平台上不同的工具。

Jobs

从创建 Jobs 的方法

```
/// BuildJobsForAction - Construct the jobs to perform for the action \p A
and
  /// return an InputInfo for the result of running \p A.  Will only
construct
  /// jobs for a given (Action, ToolChain, BoundArch, DeviceKind) tuple
once.
  InputInfo
  BuildJobsForAction(Compilation &C, const Action *A, const ToolChain *TC,
                     StringRef BoundArch, bool AtTopLevel, bool
MultipleArchs,
                     const char *LinkingOutput,
                     std::map<std::pair<const Action *, std::string>,
InputInfo>
                         &CachedResults,
                     Action::OffloadKind TargetDeviceOffloadKind) const;
```

可以看出 Jobs 需要前面的 Compilation，Action，ToolChain 等，那么 Jobs 就是将前面获取的信息进行组合分组给后面的 Execute 做万全准备。

Execute

在 driver.cpp 的 main 函数里的 ExecuteCompilation 方法里可以看到如下代码：

```
 // Set up response file names for each command, if necessary
  for (auto &Job : C.getJobs())
    setUpResponseFiles(C, Job);

  C.ExecuteJobs(C.getJobs(), FailingCommands);
```

能够看到 Jobs 准备好了后就要开始 Excute 他们。

Execute 就是执行整个的编译过程的 Jobs。过程执行的内容和耗时可以通过添加 -ftime-report 这个 option 来看到。

```
➜ llvmdemo clang main.c -ftime-report
===-------------------------------------------------------------------------===
                         Miscellaneous Ungrouped Timers
===-------------------------------------------------------------------------===

   ---User Time---   --System Time--   --User+System--   ---Wall Time---  --- Name ---
   0.0027 ( 77.4%)   0.0007 ( 67.9%)   0.0034 ( 75.3%)   0.0034 ( 75.1%)  Code Generation Time
   0.0008 ( 22.6%)   0.0003 ( 32.1%)   0.0011 ( 24.7%)   0.0011 ( 24.9%)  LLVM IR Generation Time
   0.0035 (100.0%)   0.0010 (100.0%)   0.0045 (100.0%)   0.0045 (100.0%)  Total

===-------------------------------------------------------------------------===
                                DWARF Emission
===-------------------------------------------------------------------------===
  Total Execution Time: 0.0000 seconds (0.0000 wall clock)

   ---User Time---   --System Time--   --User+System--   ---Wall Time---  --- Name ---
   0.0000 ( 58.3%)   0.0000 ( 50.0%)   0.0000 ( 56.5%)   0.0000 ( 56.2%)  Debug Info Emission
   0.0000 ( 27.8%)   0.0000 ( 40.0%)   0.0000 ( 30.4%)   0.0000 ( 29.7%)  DWARF Exception Writer
   0.0000 ( 13.9%)   0.0000 ( 10.0%)   0.0000 ( 13.0%)   0.0000 ( 14.1%)  DWARF Debug Writer
   0.0000 (100.0%)   0.0000 (100.0%)   0.0000 (100.0%)   0.0000 (100.0%)  Total

===-------------------------------------------------------------------------===
                     ... Pass execution timing report ...
===-------------------------------------------------------------------------===
  Total Execution Time: 0.0017 seconds (0.0017 wall clock)

   ---User Time---   --System Time--   --User+System--   ---Wall Time---  --- Name ---
   0.0005 ( 38.0%)   0.0001 ( 28.2%)   0.0006 ( 35.9%)   0.0006 ( 35.9%)  Expand Atomic instructions
   0.0003 ( 22.9%)   0.0001 ( 22.3%)   0.0004 ( 22.7%)   0.0004 ( 22.7%)  X86 DAG->DAG Instruction Selection
   0.0001 ( 10.3%)   0.0000 (  9.9%)   0.0002 ( 10.3%)   0.0002 ( 10.3%)  X86 Assembly Printer
   0.0001 (  4.0%)   0.0000 (  7.8%)   0.0001 (  4.8%)   0.0001 (  4.8%)  Module Verifier
   0.0001 (  4.5%)   0.0000 (  3.5%)   0.0001 (  4.3%)   0.0001 (  4.2%)  Prologue/Epilogue Insertion & Frame Finalization
   0.0001 (  3.9%)   0.0000 (  3.5%)   0.0001 (  3.8%)   0.0001 (  3.8%)  Fast Register Allocator
   0.0000 (  1.9%)   0.0000 (  1.3%)   0.0000 (  1.7%)   0.0000 (  1.9%)  Insert stack protectors
   0.0000 (  1.6%)   0.0000 (  2.7%)   0.0000 (  1.9%)   0.0000 (  1.8%)  Inliner for always_inline functions
   0.0000 (  1.7%)   0.0000 (  2.2%)   0.0000 (  1.8%)   0.0000 (  1.7%)  Two-Address instruction pass
   0.0000 (  1.3%)   0.0000 (  2.4%)   0.0000 (  1.6%)   0.0000 (  1.5%)  CallGraph Construction
   0.0000 (  1.2%)   0.0000 (  1.6%)   0.0000 (  1.3%)   0.0000 (  1.4%)  Natural Loop Information
   0.0000 (  0.8%)   0.0000 (  1.1%)   0.0000 (  0.9%)   0.0000 (  0.9%)  Scalar Evolution Analysis
   0.0000 (  0.9%)   0.0000 (  0.0%)   0.0000 (  0.7%)   0.0000 (  0.8%)  Module Verifier
   0.0000 (  0.5%)   0.0000 (  1.1%)   0.0000 (  0.6%)   0.0000 (  0.7%)  Remove unreachable blocks from the CFG
   0.0000 (  0.7%)   0.0000 (  0.0%)   0.0000 (  0.6%)   0.0000 (  0.7%)  Module Verifier
   0.0000 (  0.4%)   0.0000 (  1.1%)   0.0000 (  0.5%)   0.0000 (  0.5%)  Post-RA pseudo instruction expansion pass
   0.0000 (  0.4%)   0.0000 (  0.3%)   0.0000 (  0.3%)   0.0000 (  0.5%)  Dominator Tree Construction
   0.0000 (  0.5%)   0.0000 (  0.0%)   0.0000 (  0.4%)   0.0000 (  0.5%)  Unnamed pass: implement Pass::getPassName()
   0.0000 (  0.4%)   0.0000 (  1.3%)   0.0000 (  0.6%)   0.0000 (  0.4%)  Machine Module Information
   0.0000 (  0.3%)   0.0000 (  0.3%)   0.0000 (  0.3%)   0.0000 (  0.4%)  Basic Alias Analysis (stateless AA impl)
   0.0000 (  0.2%)   0.0000 (  0.5%)   0.0000 (  0.3%)   0.0000 (  0.3%)  Eliminate PHI nodes for register allocation
   0.0000 (  0.2%)   0.0000 (  0.8%)   0.0000 (  0.3%)   0.0000 (  0.3%)  Bundle Machine CFG Edges
   0.0000 (  0.1%)   0.0000 (  0.8%)   0.0000 (  0.3%)   0.0000 (  0.3%)  Function Alias Analysis Results
   0.0000 (  0.3%)   0.0000 (  0.0%)   0.0000 (  0.2%)   0.0000 (  0.3%)  Dominator Tree Construction
   0.0000 (  0.1%)   0.0000 (  0.3%)   0.0000 (  0.2%)   0.0000 (  0.2%)  Inserts calls to mcount-like functions
   0.0000 (  0.1%)   0.0000 (  0.5%)   0.0000 (  0.2%)   0.0000 (  0.2%)  Safe Stack instrumentation pass
```

使用工具链

编译 c 程序

```c
#include <stdio.h>
int main() {
    int a,b;
    printf("Please input a:");
    scanf("%d",&a);
    printf("Please input b:");
    scanf("%d",&b);
    printf("a is:%d,b is :%d,count equal:%d",a,b,a+b);
}
```

```
clang main.c -o main
```

```
➜  llvmdemo clang main.c -o main
➜  llvmdemo vi main.c
➜  llvmdemo ./main
Please input a:34
Please input b:3
a is:34,b is :3,count equal:37%
```

生成 Bitcode

编译生成 Bitcode，JIT Compiler 运行字节码文件

```
clang -O3 -emit-llvm main.c -c -o main.bc
lli main.bc
```

生成可视化 Bitcode

```
clang -O3 -emit-llvm main.c -S -o main.ll
```

```
➜  llvmdemo clang -O3 -emit-llvm main.c -S -o main.ll
➜  llvmdemo cat main.ll
; ModuleID = 'main.c'
source_filename = "main.c"
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.12.0"

@.str = private unnamed_addr constant [16 x i8] c"Please input a:\00", align 1
@.str.1 = private unnamed_addr constant [3 x i8] c"%d\00", align 1
@.str.2 = private unnamed_addr constant [16 x i8] c"Please input b:\00", align 1
@.str.3 = private unnamed_addr constant [32 x i8] c"a is:%d,b is :%d,count equal:%d\
00", align 1

; Function Attrs: nounwind ssp uwtable
define i32 @main() local_unnamed_addr #0 {
entry:
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  %0 = bitcast i32* %a to i8*
  call void @llvm.lifetime.start(i64 4, i8* nonnull %0) #3
  %1 = bitcast i32* %b to i8*
```

反汇编字节码

```
llc main.bc -o main.s
```

```
    .section    __TEXT,__text,regular,pure_instructions
    .macosx_version_min 10, 12
    .globl  _main
    .p2align    4, 0x90
```

```
_main:                                  ## @main
    .cfi_startproc
## BB#0:                                ## %entry
    pushq   %rbp
Lcfi0:
    .cfi_def_cfa_offset 16
Lcfi1:
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
Lcfi2:
    .cfi_def_cfa_register %rbp
    pushq   %rbx
    pushq   %rax
Lcfi3:
    .cfi_offset %rbx, -24
    leaq    L_.str(%rip), %rdi
    xorl    %eax, %eax
    callq   _printf
    leaq    L_.str.1(%rip), %rbx
    leaq    -16(%rbp), %rsi
    xorl    %eax, %eax
    movq    %rbx, %rdi
    callq   _scanf
    leaq    L_.str.2(%rip), %rdi
    xorl    %eax, %eax
    callq   _printf
    leaq    -12(%rbp), %rsi
    xorl    %eax, %eax
    movq    %rbx, %rdi
    callq   _scanf
    movl    -16(%rbp), %esi
    movl    -12(%rbp), %edx
    leal    (%rdx,%rsi), %ecx
    leaq    L_.str.3(%rip), %rdi
    xorl    %eax, %eax
                                        ## kill: %ESI<def> %ESI<kill>
%RSI<kill>
                                        ## kill: %EDX<def> %EDX<kill>
%RDX<kill>
    callq   _printf
    xorl    %eax, %eax
    addq    $8, %rsp
    popq    %rbx
    popq    %rbp
    retq
    .cfi_endproc

    .section    __TEXT,__cstring,cstring_literals
L_.str:                                 ## @.str
```

```
    .asciz  "Please input a:"

L_.str.1:                               ## @.str.1
    .asciz  "%d"

L_.str.2:                               ## @.str.2
    .asciz  "Please input b:"

L_.str.3:                               ## @.str.3
    .asciz  "a is:%d,b is :%d,count equal:%d"


.subsections_via_symbols
```

分析可执行文件

前面几行汇编指令

```
    .section    __TEXT,__text,regular,pure_instructions
    .macosx_version_min 10, 12
    .globl  _main
    .p2align    4, 0x90
```

- section - 指令指定接下来是执行哪个段。
- globl - 指令说明 _main 是一个外部符号，main() 对于系统来说是可调用执行文件的。
- align - 指出后面代码的对齐方式，16(2^4) 字节对齐， 0x90 补齐。

main 函数头部部分

```
_main:                                  ## @main
    .cfi_startproc
## BB#0:                                ## %entry
    pushq   %rbp
Lcfi0:
    .cfi_def_cfa_offset 16
Lcfi1:
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
Lcfi2:
    .cfi_def_cfa_register %rbp
    pushq   %rbx
    pushq   %rax
```

- _main - 是函数开始的地址。
- .cfi_startproc 和 .cfi_endproc - 配对出现，前者表示函数的开始，后者表示函数的结束。CFI 是 Call Frame Infomation 的缩写，调用帧信息的意思。
- pushq %rbp - 汇编代码，指把 rbp 的值 push 到栈中。在 ## BB#0 这个 label 里 ABI 会让 rbp

这个寄存器被保护起来，函数返回时让 rbp 寄存器的值跟以前一样。ABI 它指定函数调用是如何在汇编代码层面上工作。

- .cfi_def_cfa_offset 16 和 .cfi_offset %rbp, -16 - 输出堆栈和调试信息。
- movq %rsp, %rbp - 把局部变量放到栈上。

打印部分

```
        leaq    L_.str(%rip), %rdi
        xorl    %eax, %eax
        callq   _printf
```

- leap 会将 L.*str(%rip) 加载到 rax 寄存器里。在 .section* ***TEXT,****cstring,cstring_literals 区域可以看到 L*.str，L.*str.1，L*.str.2 等字符串的定义。
- callq 会调用 print() 函数。

LLVM IR 中间代码

不管编译的语言时 Objective-C 还是 Swift 也不管对应机器是什么，亦或是即时编译，LLVM 里唯一不变的是中间语言 LLVM IR。那么我们就来看看如何玩 LLVM IR。

IR 结构

下面是刚才生成的 main.ll 中间代码文件。

```
; ModuleID = 'main.c'
source_filename = "main.c"
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.12.0"

@.str = private unnamed_addr constant [16 x i8] c"Please input a:\00", align
1
@.str.1 = private unnamed_addr constant [3 x i8] c"%d\00", align 1
@.str.2 = private unnamed_addr constant [16 x i8] c"Please input b:\00",
align 1
@.str.3 = private unnamed_addr constant [32 x i8] c"a is:%d,b is :%d,count
equal:%d\00", align 1

; Function Attrs: nounwind ssp uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = bitcast i32* %1 to i8*
  call void @llvm.lifetime.start(i64 4, i8* %3) #3
  %4 = bitcast i32* %2 to i8*
  call void @llvm.lifetime.start(i64 4, i8* %4) #3
  %5 = tail call i32 (i8*, …) @printf(i8* getelementptr inbounds ([16 x i8],
[16 x i8]* @.str, i64 0, i64 0))
  %6 = call i32 (i8*, …) @scanf(i8* getelementptr inbounds ([3 x i8], [3 x
```

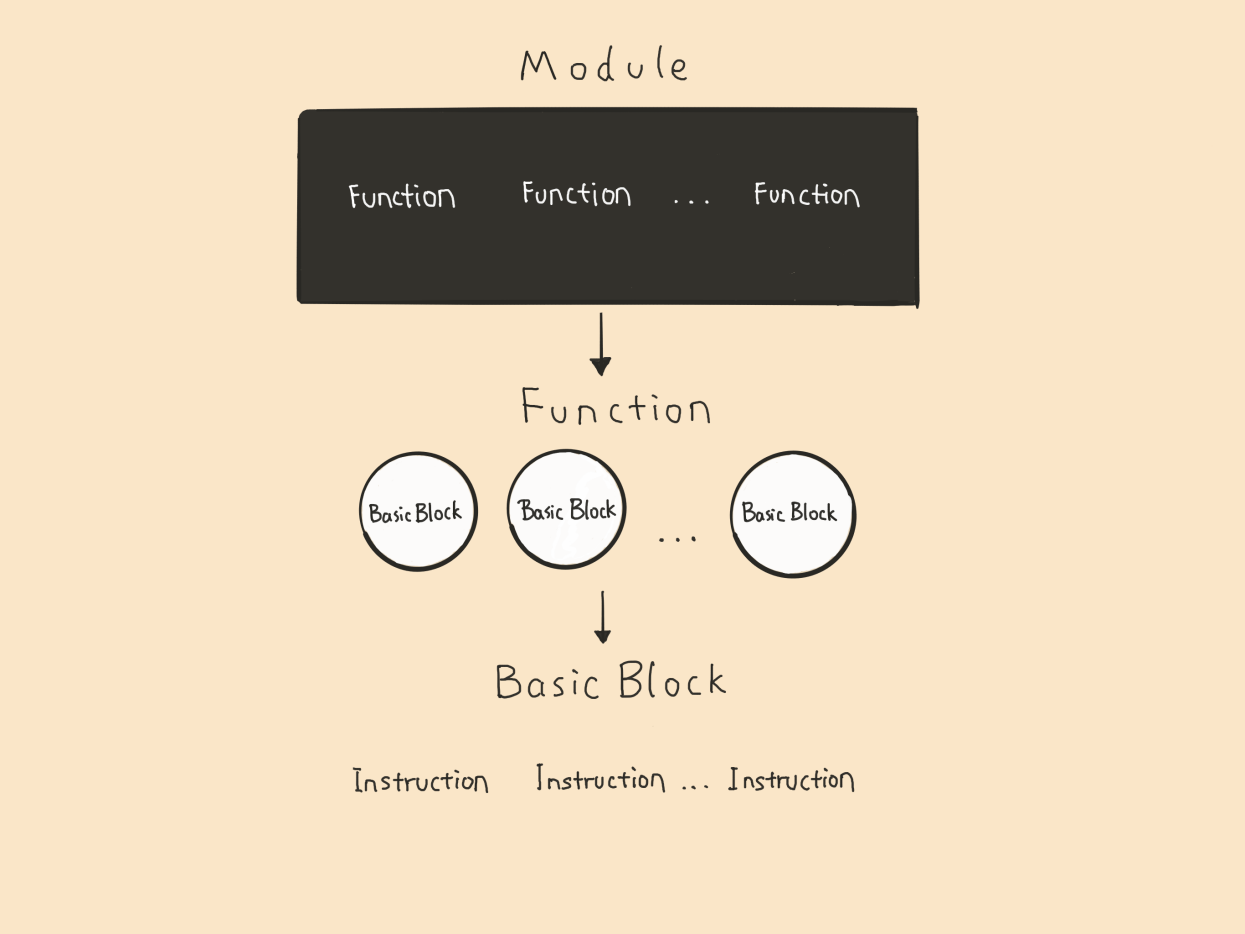
```
i8]* @.str.1, i64 0, i64 0), i32* nonnull %1)
  %7 = call i32 (i8*, …) @printf(i8* getelementptr inbounds ([16 x i8], [16 x
i8]* @.str.2, i64 0, i64 0))
  %8 = call i32 (i8*, …) @scanf(i8* getelementptr inbounds ([3 x i8], [3 x
i8]* @.str.1, i64 0, i64 0), i32* nonnull %2)
  %9 = load i32, i32* %1, align 4, !tbaa !2
  %10 = load i32, i32* %2, align 4, !tbaa !2
  %11 = add nsw i32 %10, %9
  %12 = call i32 (i8*, …) @printf(i8* getelementptr inbounds ([32 x i8], [32
x i8]* @.str.3, i64 0, i64 0), i32 %9, i32 %10, i32 %11)
  call void @llvm.lifetime.end(i64 4, i8* %4) #3
  call void @llvm.lifetime.end(i64 4, i8* %3) #3
  ret i32 0
}

; Function Attrs: argmemonly nounwind
declare void @llvm.lifetime.start(i64, i8* nocapture) #1

; Function Attrs: nounwind
declare i32 @printf(i8* nocapture readonly, …) #2

; Function Attrs: nounwind
declare i32 @scanf(i8* nocapture readonly, …) #2

; Function Attrs: argmemonly nounwind
declare void @llvm.lifetime.end(i64, i8* nocapture) #1

attributes #0 = { nounwind ssp uwtable “disable-tail-calls”=“false” “less-
precise-fpmad”=“false” “no-frame-pointer-elim”=“true” “no-frame-pointer-elim-
non-leaf” “no-infs-fp-math”=“false” “no-nans-fp-math”=“false” “stack-
protector-buffer-size”=“8” “target-cpu”=“penryn” “target-
features”=“+cx16,+fxsr,+mmx,+sse,+sse2,+sse3,+sse4.1,+ssse3” “unsafe-fp-
math”=“false” “use-soft-float”=“false” }
attributes #1 = { argmemonly nounwind }
attributes #2 = { nounwind “disable-tail-calls”=“false” “less-precise-
fpmad”=“false” “no-frame-pointer-elim”=“true” “no-frame-pointer-elim-non-
leaf” “no-infs-fp-math”=“false” “no-nans-fp-math”=“false” “stack-protector-
buffer-size”=“8” “target-cpu”=“penryn” “target-
features”=“+cx16,+fxsr,+mmx,+sse,+sse2,+sse3,+sse4.1,+ssse3” “unsafe-fp-
math”=“false” “use-soft-float”=“false” }
attributes #3 = { nounwind }

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !”PIC Level”, i32 2}
!1 = !{!”Apple LLVM version 8.0.0 (clang-800.0.42.1)”}
!2 = !{!3, !3, i64 0}
!3 = !{!”int”, !4, i64 0}
```

```
!4 = !{!"omnipotent char", !5, i64 0}
!5 = !{!"Simple C/C++ TBAA"}
```

LLVM IR 有三种表示格式，第一种是 bitcode 这样的存储格式，以 .bc 做后缀，第二种是可读的以 .ll，第三种是用于开发时操作 LLVM IR 的内存格式。

一个编译的单元即一个文件在 IR 里就是一个 Module，Module 里有 Global Variable 和 Function，在 Function里有 Basic Block，Basic Block 里有 指令 Instructions。
‘

通过下面的 IR 结构图能够更好的理解 IR 的整体结构。



图中可以看出最大的是 Module，里面包含多个 Function，每个 Function 包含多个 BasicBlock，BasicBlock 里含有 Instruction，代码非常清晰，这样如果想开发一个新语言只需要完成语法解析后通过 LLVM 提供的丰富接口在内存中生成 IR 就可以直接运行在各个不同的平台。

IR 语言满足静态单赋值，可以很好的降低数据流分析和控制流分析的复杂度。及只能在定义时赋值，后面不能更改。但是这样就没法写程序了，输入输出都没法弄，所以函数式编程才会有类似 Monad 这样机制的原因。

LLVM IR 优化

使用 O2，O3 这样的优化会调用对应的 Pass 来进行处理，有比如类似死代码清理，内联化，表达式重组，循环变量移动这样的 Pass。可以通过 llvm-opt 调用 LLVM 优化相关的库。

可能直接这么说不太直观，我们可以更改下原 c 代码举个小例子看看这些 Pass 会做哪些优化。当我们加上

```
int i = 0;
while (i < 10) {
    i++;
    printf("%d",i);
}
```

对应的 IR 代码是

```
  %call4 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8],
[3 x i8]* @.str.1, i64 0, i64 0), i32 1)
  %call4.1 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x
i8], [3 x i8]* @.str.1, i64 0, i64 0), i32 2)
  %call4.2 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x
i8], [3 x i8]* @.str.1, i64 0, i64 0), i32 3)
  %call4.3 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x
i8], [3 x i8]* @.str.1, i64 0, i64 0), i32 4)
  %call4.4 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x
i8], [3 x i8]* @.str.1, i64 0, i64 0), i32 5)
  %call4.5 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x
i8], [3 x i8]* @.str.1, i64 0, i64 0), i32 6)
  %call4.6 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x
i8], [3 x i8]* @.str.1, i64 0, i64 0), i32 7)
  %call4.7 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x
i8], [3 x i8]* @.str.1, i64 0, i64 0), i32 8)
  %call4.8 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x
i8], [3 x i8]* @.str.1, i64 0, i64 0), i32 9)
  %call4.9 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x
i8], [3 x i8]* @.str.1, i64 0, i64 0), i32 10)
```

可以看出来这个 while 在 IR 中就是重复的打印了10次，那要是我把10改成100是不是会变成打印100次呢?

我们改成100后，再次生成 IR 可以看到 IR 变成了这样：

```
  br label %while.body

while.body:                                       ; preds = %while.body,
%entry
  %i.010 = phi i32 [ 0, %entry ], [ %inc, %while.body ]
  %inc = add nuw nsw i32 %i.010, 1
  %call4 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8],
[3 x i8]* @.str.1, i64 0, i64 0), i32 %inc)
  %exitcond = icmp eq i32 %inc, 100
  br i1 %exitcond, label %while.end, label %while.body

while.end:                                        ; preds = %while.body
  %2 = load i32, i32* %a, align 4, !tbaa !2
  %3 = load i32, i32* %b, align 4, !tbaa !2
  %add = add nsw i32 %3, %2
  %call5 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([11 x i8],
[11 x i8]* @.str.3, i64 0, i64 0), i32 %add)
  call void @llvm.lifetime.end(i64 4, i8* nonnull %1) #3
  call void @llvm.lifetime.end(i64 4, i8* nonnull %0) #3
  ret i32 0
}
```

这里对不同条件生成的不同都是 Pass 优化器做的事情。解读上面这段 IR 需要先了解下 IR 语法关键字，如下：

- @ - 代表全局变量
- % - 代表局部变量
- alloca - 指令在当前执行的函数的堆栈帧中分配内存，当该函数返回到其调用者时，将自动释放内存。
- i32：- i 是几这个整数就会占几位，i32就是32位4字节
- align - 对齐，比如一个 int,一个 char 和一个 int。单个 int 占4个字节，为了对齐只占一个字节的char需要向4对齐占用4字节空间。
- Load - 读出，store 写入
- icmp - 两个整数值比较，返回布尔值
- br - 选择分支，根据 cond 来转向 label，不根据条件跳转的话类似 goto
- indirectbr - 根据条件间接跳转到一个 label，而这个 label 一般是在一个数组里，所以跳转目标是可变的，由运行时决定的
- label - 代码标签

```
br label %while.body
```

如上面表述，br 会选择跳向 while.body 定义的这个标签。这个标签里可以看到

```
%exitcond = icmp eq i32 %inc, 100
  br i1 %exitcond, label %while.end, label %while.body
```

这段，icmp 会比较当前的 %inc 和定义的临界值 100，根据返回的布尔值来决定 br 跳转到那个代码标签，真就跳转到 while.end 标签，否就在进入 while.body 标签。这就是 while 的逻辑。通过br 跳转和 label 这种标签的概念使得 IR 语言能够成为更低级兼容性更高更方便转向更低级语言的语言。

SSA

LLVM IR 是 SSA 形式的，维护双向 def-use 信息，use-def 是通过普通指针实现信息维护，def-use 是通过内存跳表和链表来实现的，便于 forward dataflow analysis 和 backward dataflow analysis。可以通过 ADCE 这个 Pass 来了解下 backward dataflow，这个pass 的源文件在 lib_Transforms_Scalar/ADCE.cpp 中，ADCE 实现了 Aggressive Dead Code Elimination Pass。这个 Pass 乐观地假设所有 instructions 都是 Dead 直到证明是否定的，允许它消除其他 DCE Pass 的 Dead 计算 catch，特别是涉及循环计算。其它 DCE 相关的 Pass 可以查看同级目录下的 BDCE.cpp 和 DCE.cpp，目录下其它的 Pass 都是和数据流相关的分析包含了各种分析算法和思路。

那么看看加法这个操作的相关的 IR 代码

```
%2 = load i32, i32* %a, align 4, !tbaa !2
%3 = load i32, i32* %b, align 4, !tbaa !2
%add = add nsw i32 %3, %2
```

加法对应的指令是

```
BinaryOperator::CreateAdd(Value *V1, Value *V2, const Twine &Name)
```

两个输入 V1 和 V2 的 def-use 是如何的呢，看看如下代码

```
class Value {
  void addUse(Use &U) { U.addToList(&UseList); }

  // ...
};

class Use {
  Value *Val;
  Use *Next;
  PointerIntPair<Use **, 2, PrevPtrTag> Prev;

  // ...
};

void Use::set(Value *V) {
  if (Val) removeFromList();
  Val = V;
  if (V) V->addUse(*this);
}

Value *Use::operator=(Value *RHS) {
  set(RHS);
```

```
  return RHS;
}

class User : public Value {
  template <int Idx, typename U> static Use &OpFrom(const U *that) {
    return Idx < 0
      ? OperandTraits<U>::op_end(const_cast<U*>(that))[Idx]
      : OperandTraits<U>::op_begin(const_cast<U*>(that))[Idx];
  }
  template <int Idx> Use &Op() {
    return OpFrom<Idx>(this);
  }
  template <int Idx> const Use &Op() const {
    return OpFrom<Idx>(this);
  }

  // ...
};

class Instruction : public User,
                    public ilist_node_with_parent<Instruction, BasicBlock> {
  // ...
};

class BinaryOperator : public Instruction {
  /// Construct a binary instruction, given the opcode and the two
  /// operands.  Optionally (if InstBefore is specified) insert the
instruction
  /// into a BasicBlock right before the specified instruction.  The
specified
  /// Instruction is allowed to be a dereferenced end iterator.
  ///
  static BinaryOperator *Create(BinaryOps Op, Value *S1, Value *S2,
                                const Twine &Name = Twine(),
                                Instruction *InsertBefore = nullptr);

  // ...
};

BinaryOperator::BinaryOperator(BinaryOps iType, Value *S1, Value *S2,
                               Type *Ty, const Twine &Name,
                               Instruction *InsertBefore)
  : Instruction(Ty, iType,
                OperandTraits<BinaryOperator>::op_begin(this),
                OperandTraits<BinaryOperator>::operands(this),
                InsertBefore) {
  Op<0>() = S1;
  Op<1>() = S2;
  init(iType);
```

```
  setName(Name);
}

BinaryOperator *BinaryOperator::Create(BinaryOps Op, Value *S1, Value *S2,
                                       const Twine &Name,
                                       Instruction *InsertBefore) {
  assert(S1->getType() == S2->getType() &&
         "Cannot create binary operator with two operands of differing
type!");
  return new BinaryOperator(Op, S1, S2, S1->getType(), Name, InsertBefore);
}
```

从代码里可以看出是使用了 Use 对象来把 use 和 def 联系起来的。

LLVM IR 通过 mem2reg 这个 Pass 来把局部变量成 SSA 形式。这个 Pass 的代码在 lib_Transforms_Utils_Mem2Reg.cpp 里。LLVM通过 mem2reg Pass 能够识别 alloca 模式，将其设置 SSA value。这时就不在需要 alloca，load和store了。mem2reg 是对 PromoteMemToReg 函数调用的一个简单包装，真正的算法实现是在 PromoteMemToReg 函数里，这个函数在 lib_Transforms_Utils_PromoteMemoryToRegister.cpp 这个文件里。

这个算法会使 alloca 这个仅仅作为 load 和 stores 的用途的指令使用迭代 dominator 边界转换成 PHI 节点，然后通过使用深度优先函数排序重写 loads 和 stores。这种算法叫做 iterated dominance frontier算法，具体实现方法可以参看 PromoteMemToReg 函数的实现。

当然把多个字节码 .bc 合成一个文件，链接时还会优化，IR 结构在优化后会有变化，这样还能够在变化后的 IR 的结构上再进行更多的优化。

这里可以进行 lli 解释执行 LLVM IR。

llc 编译器是专门编译 LLVM IR 的编译器用来生成汇编文件。

调用系统汇编器比如 GNU 的 as 来编译生成 .o Object 文件，接下来就是用链接器链接相关库和 .o 文件一起生成可执行的 .out 或者 exe 文件了。

llvm-mc 还可以直接生成 object 文件。

Clang CFE

动手玩肯定不能少了 Clang 的前端组件及库，熟悉这些库以后就能够自己动手用这些库编写自己的程序了。下面我就对这些库做些介绍，然后再着重说说 libclang 库，以及如何用它来写工具。

- LLVM Support Library - LLVM libSupport 库提供了许多底层库和数据结构，包括命令行 option 处理，各种容器和系统抽象层，用于文件系统访问。
- The Clang “Basic” Library - 提供了跟踪和操纵 source buffers，source buffers 的位置，diagnostics，tokens,抽象目标以及编译语言子集信息的 low-level 实用程序。还有部分可以用在其他的非 c 语言比如 SourceLocation，SourceManager，Diagnositics，FileManager 等。其中 Diagnositics 这个子系统是编译器和普通写代码人交流的主要组成部分，它会诊断当前代码哪些不正确，按照严重程度而产生 WARNING 或 ERROR，每个诊断会有唯一 ID ， SourceLocation 会负责管理。
- The Driver Library - 和 Driver 相关的库，上面已经对其做了详细的介绍。
- Precompiled Headers - Clang 支持预编译 headers 的两个实现。

- The Frontend Library - 这个库里包含了在 Clang 库之上构建的功能，比如输出 diagnositics 的几种方法。
- The Lexer and Preprocessor Library - 词法分析和预处理的库，包含了 Token，Annotation Tokens，TokenLexer，Lexer 等词法类，还有 Parser Library 和 AST 语法树相关的比如 Type，ASTContext，QualType，DeclarationName，DeclContext 以及 CFG 类。
- The Sema Library - 解析器调用此库时，会对输入进行语义分析。 对于有效的程序，Sema 为解析构造一个 AST。
- The CodeGen Library - CodeGen 用 AST 作为输入，并从中生成 LLVM IR 代码。

libclang

libclang 会让你觉得 clang 不仅仅只是一个伟大的编译器。下面从解析源码来说下

先写个 libclang 的程序来解析源码

```
int main(int argc, char *argv[]) {
    CXIndex Index = clang_createIndex(0, 0);
    CXTranslationUnit TU = clang_parseTranslationUnit(Index, 0,
                                                      argv, argc, 0, 0,
CXTranslationUnit_None); for (unsigned I = 0, N =
clang_getNumDiagnostics(TU); I != N; ++I) {
        CXDiagnostic Diag = clang_getDiagnostic(TU, I);
        CXString String =
clang_formatDiagnostic(Diag,clang_defaultDiagnosticDisplayOptions());
        fprintf(stderr, "%s\n", clang_getCString(String));
        clang_disposeString(String);
    }
    clang_disposeTranslationUnit(TU);
    clang_disposeIndex(Index);
    return 0;
}
```

再写个有问题的 c 程序

```
struct List { /**/ };int sum(union List *L) { /* ... */ }
```

运行了语法检查后会出现提示信息

```
list.c:2:9: error: use of 'List' with tag type that does not match
      previous declaration
int sum(union List *Node) {
^~~~~
struct
list.c:1:8: note: previous use is here
struct List {
^
```

下面我们看看诊断过程，显示几个核心诊断方法诊断出问题

- enum CXDiagnosticSeverity clang_getDiagnosticSeverity(CXDiagnostic Diag);
- CXSourceLocation clang_getDiagnosticLocation(CXDiagnostic Diag);
- CXString clang_getDiagnosticSpelling(CXDiagnostic Diag);

接着进行高亮显示，最后提供两个提示修复的方法

- unsigned clang_getDiagnosticNumFixIts(CXDiagnostic Diag);* CXString clang_getDiagnosticFixIt(CXDiagnostic Diag, unsigned FixIt,CXSourceRange *ReplacementRange);

我们先遍历语法树的节点。源 c 程序如下

```
struct List {
    int Data;
    struct List *Next;
};
int sum(struct List *Node) {
    int result = 0;
    for (; Node; Node = Node->Next)
        result = result + Node->Data;
    return result;
}
```

先找出所有的声明，比如 List，Data，Next，sum，Node 以及 result 等。再找出引用，比如 struct List *Next 里的 List。还有声明和表达式，比如 int result = 0; 还有 for 语句等。还有宏定义和实例化等。

CXCursor 会统一 AST 的节点，规范包含的信息

- 代码所在位置和长度
- 名字和符号解析
- 类型
- 子节点

举个 CXCursor 分析例子

```
struct List {
    int Data;
    struct List *Next;
};
```

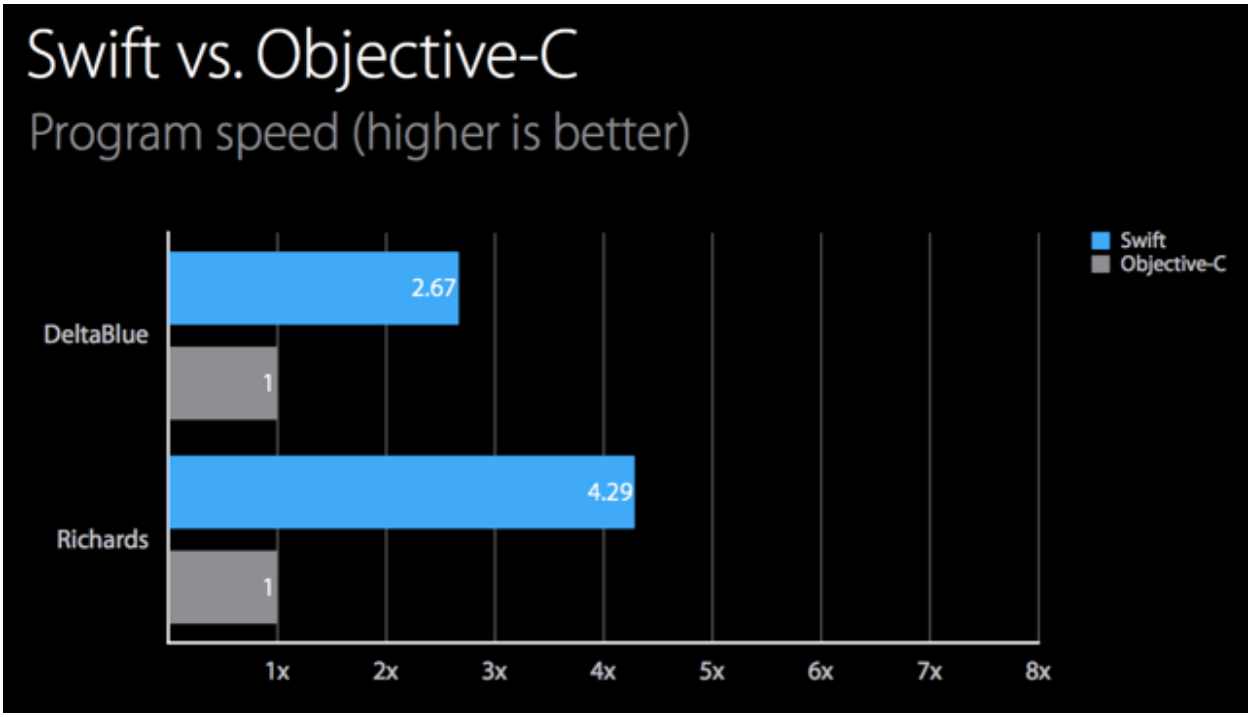
CXCursor 的处理过程如下

```
//Top-level cursor C
clang_getCursorKind(C) == CXCursor_StructDecl
clang_getCursorSpelling(C) == "List" //获取名字字符串
clang_getCursorLocation(C) //位置
clang_getCursorExtent(C) //长度
clang_visitChildren(C, ...); //访问子节点

//Reference cursor R
clang_getCursorKind(R) == CXCursor_TypeRef
clang_getCursorSpelling(R) == "List"
clang_getCursorLocation(R)
clang_getCursorExtent(R)
clang_getCursorReferenced(R) == C //指向C
```

Swift 性能分析

作者：唐巧

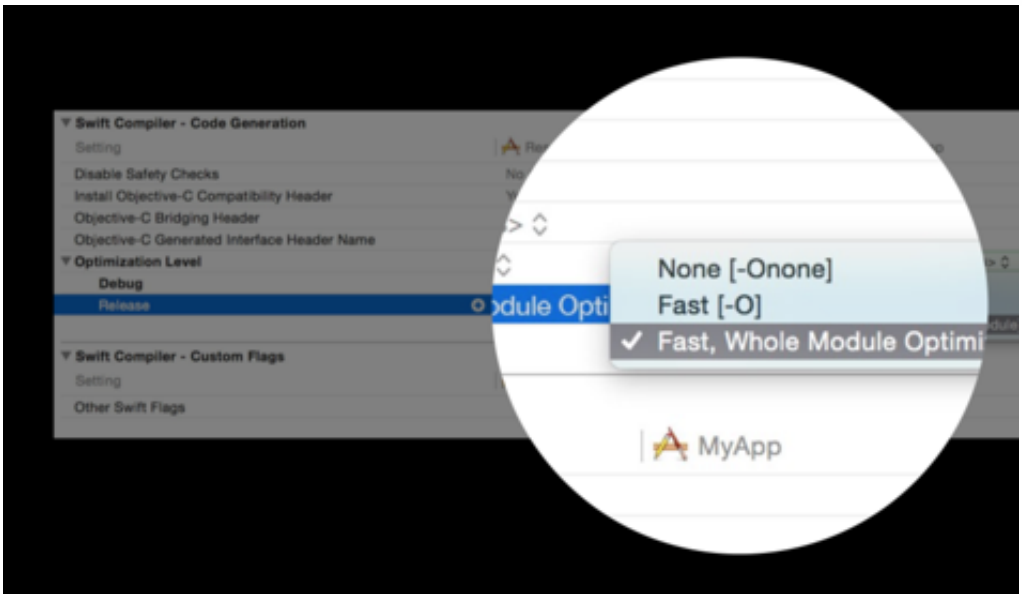
Richards 和 Deltablue 是衡量语言运算速度的两个主流的评测代码。Swift 在这两个评测中，显示出远超 Objective-C 的性能（如下图）。特别是 Richards 评测，Swift 比 Objective-C 快了 4 倍。那么，为什么 Swift 这么快呢?



有些人会觉得 Swift 就应该很快，因为它是一个新的、现代的语言。但是别忘了，Swift 其实非常年轻，从 2014 年推出 1.0 版本以来，它才经过了三年的发展。而 Objective-C 作为苹果在过去 30 年来唯一的在 macOS 和 iOS 平台上的语言，经过了大量 Apple 工程师的优化，如果有什么优化 Swift 可以做，为什么 Objective-C 就不能在过去几十年中做到呢？为了解决我自己的这个疑惑，我查阅了一些相关的资料，并做了一些实验。

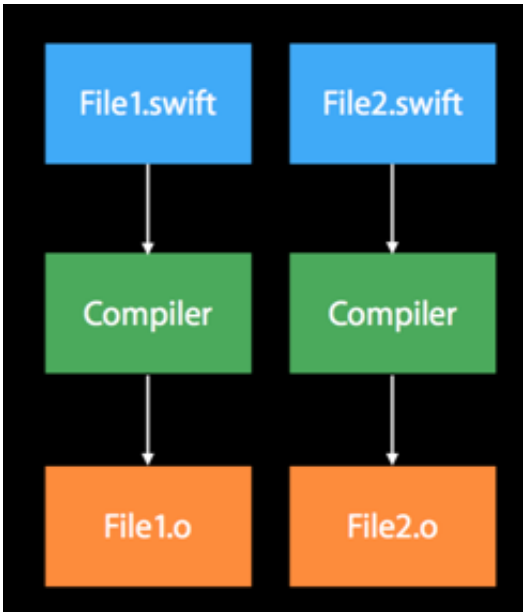
接下来，我将从编译器优化，内存分配优化，引用计数优化，方法调用优化，面向协议编程的实现细节等方面来介绍 Swift 在性能上所做的努力。最后，我们也会一起看看编译器处理后的源码，加深我们对于这些优化的理解。

编译器优化



Swift 在编译器优化方面引入了 `Whole Module Optimizations` 的机制(如上图)。这个机制有什么用呢?

在没有这个机制之前，编译器在编译的时候，针对每个源文件，生成对应的目标文件，然后链接器会将目标文件组合起来，最终生成可执行程序。这个过程就像下图那样。每一个 `.o` 文件就是一个目标文件，目标文件之间无法相互优化。



我们想像这样一个场景，File1.swift 中定义了如下代码：

```
func min<T : Comparable>(x: T, y: T) -> T {
    return y < x ? y : x
}
```

这是一个范型的函数，能够支持各种 Comparable 的类型。然后我们如果在 File2.swift 中用到这个函数，假设我们在 File2.swift 中的代码如下：

```
func test() {
    let x: Int = 1
    let y: Int = 2
    let r = min<Int>(x, y)
}
```

编译器在处理时，会将 File1.swift 中的范型函数编译成类似这样的代码：

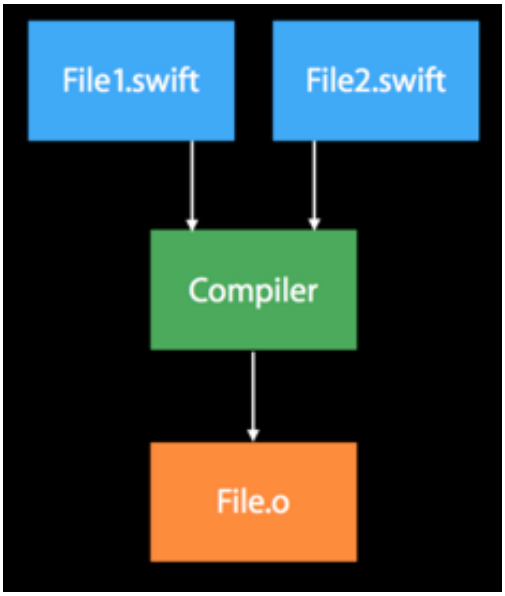
```
func min<T : Comparable>(x: T, y: T, FTable: FunctionTable) -> T {
    let xCopy = FTable.copy(x)
    let yCopy = FTable.copy(y)
    let m = FTable.lessThan(yCopy, xCopy) ? y : x
    FTable.release(x)
    FTable.release(y)
    return m
}
```

这当然能正常工作，但是 Int 其实是一个基本类型，如果整个程序只有这一处使用这个范型函数，编译器完全可以把这个 min 函数定制成一个只支持 Int 的版本，像这样：

```
func min<Int>(x: Int, y: Int) -> Int {
    return y < x ? y : x
}
```

但是，如果没有 `Whole Module Optimizations`，编译器在编译 File1.swift 的时候，无法知晓这个函数在别的文件当中的使用方式，就不敢做这样的优化。而如果开启了 `Whole Module Optimizations`，编译器知道了所有的信息，就可以把这个范型函数优化成只支持 Int 类型的版本。

类似的优化还有很多，下图是开启 `Whole Module Optimizations` 之后编译的过程。应该说，`Whole Module Optimizations` 给予了编译器更多的信息，使得编译器可以做更多的基于全局信息的优化。



内存分配和引用计数优化

在 Objective-C 语言中，所有对象都是引用类型，Objective-C 语言使用引用计数来管理对象的生命期。引用计数是一种古老但是有效的内存管理方式，除了需要注意循环引用问题之外，由于 ARC 的存在，大部分程序员并没有特别感受到引用计数的麻烦之处。

但是不得不说，在 Objective-C 语言中，引用计数代码其实无处不在。虽然你可能没有写一行关于引用计数的代码，但是编译器却为你生成了大量相关的引用计数文件。如果你感兴趣，可以尝试用 IDA 反汇编 Objective-C 语言编写的 App 文件，你就可以看到大量的引用计数代码。

下面一段代码，是我曾经逆向分析过的一个 App 的加密函数，你可以到，在短短的 30 行代码中，除了像 `objc_msgSend` 这种方法调用的代码，就是大量的 `objc_retain`，`objc_retainAutoreleasedReturnValue`，以及 `objc_release` 的代码。

```
v39 = objc_msgSend(CFSTR("v1/user?id="), "stringByAppendingString:", v37);
v40 = objc_retainAutoreleasedReturnValue(v39);
v41 = v40;
v42 = objc_msgSend(v49, "operationWithPath:params:httpMethod:", v40, 0,
CFSTR("POST"));
v43 = v37;
v44 = (void *)objc_retainAutoreleasedReturnValue(v42);
v58 = (int)&_NSConcreteStackBlock;
v59 = -1040187392;
v60 = 0;
v61 = sub_5D478;
v62 = (int)&unk_2E2130;
v63 = objc_retain(v11, sub_5D478);
v51 = (int)&_NSConcreteStackBlock;
v52 = -1040187392;
v53 = 0;
v54 = sub_5D500;
v55 = (int)&unk_2E2150;
v56 = objc_retain(v48, &unk_2E2150);
v57 = v41;
v45 = objc_retain(v41, &selRef_addCompletionHandler_errorHandler_);
objc_msgSend(v44, "addCompletionHandler:errorHandler:", &v58, &v51);
objc_msgSend(v49, "enqueueOperation:", v44);
objc_release(v57);
objc_release(v56);
objc_release(v63);
objc_release(v45);
v38 = (int)v44;
v37 = v43;
```

这么多的引用计数操作对性能有没有影响？当然有影响！实际上，为了线程安全，每一个对象的引用计数操作，都伴随着锁（Lock）的操作，而这些操作其实都是非常费时的。我们来看下面一个例子：

```
var array: [TangQiao] = ...

for t in array {
    // increase RC
    // decrease RC
}
```

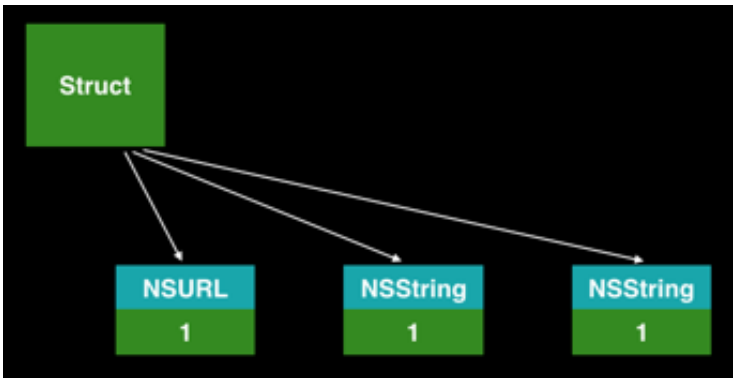
假设在这个例子中，我们的类型 `TangQiao` 是一个 class 类型。那么如果你要遍历这个数组中的每个元素，编译器就会在循环体的内部，对于每一个遍历的元素加上引用计数的增加和减少操作，这其实是特消耗性能的。

所以，大家知道，Swift 引入了值类型，即：struct。struct 有着许多优点，其中一个重要的优点，就是 struct 是不需要引用计数管理的。所以如果你把 `TangQiao` 这个类改成 struct 类型，那么在遍历数组的过程中，所有的引用计数代码就会从编译器中消失。

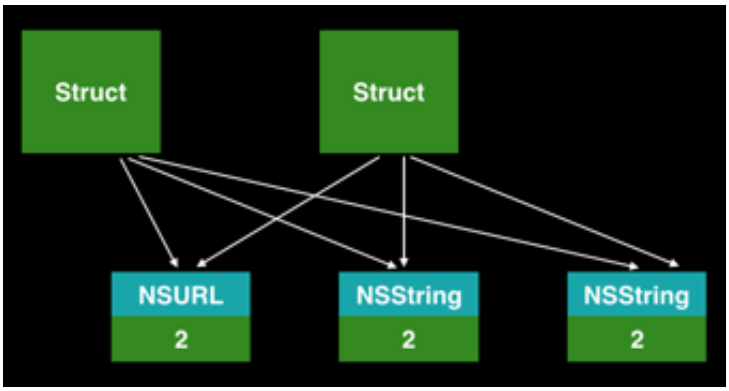
但是 struct 的使用也有要注意的地方，如果我们的 struct 中含有大量的引用类型成员，那么在变量复制时，也可能千万大量的引用计数操作。以下是一个例子：

```
struct TangQiao {
    var website = NSURL(string: "http://blog.devtang.com")
    var name = NSString(string: "tangqiaoboy")
    var addr = NSString(string: "address")
}
var x = TangQiao()
var y = x
```

在这个例子中，因为 `x` 变量的三个成员变量都是引用类型，所以其创建时，内存布局会是如下这样：



而当我们调用 `y = x` 时，由于 x 被复制，所以相关的引用成员的引用计数都需要 `+1`，内存布局会更新成这样：

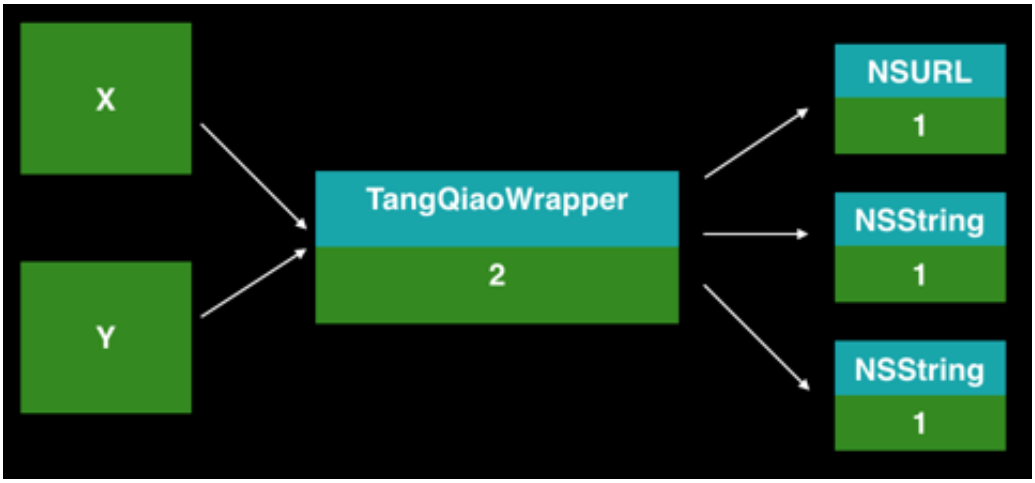


这当然是不太能被接受的事情，如果你的 struct 里面有着大量的引用类型的成员，这将意味着你不但无法避免引用计数，而且每次参数传递或复制时都会带来大量的引用计数操作。

对于这种情况，一种有效但是略显丑陋的办法是：引用一个封装类，把这些引用类型的成员再封装一层，像如下这样：

```
struct TangQiao {
    var member: TangQiaoWrapper = TangQiaoWrapper()
}

class TangQiaoWrapper {
    var website = NSURL(string: "http://blog.devtang.com")
    var name = NSString(string: "tangqiaoboy")
    var addr = NSString(string: "address")
}
var x = TangQiao()
var y = x
```

有了上面这个 `TangQiaoWrapper` 类，当发生对象复制时，内存布局中只有 `TangQiaoWrapper` 的引用计数会变化，其它的引用计数则不会变化，如下图所示：



方法调用优化

从刚刚我们看到的反汇编代码中，我们可以看到，所有的 Objective-C 语言的方法调用，其实都是向相应的对象「发消息」，所以编译之后，都是变成了调用 `objc_msgSend` 函数，`objc_msgSend` 函数接受的第一个参数是接受消息的对象，第二个参数是消息的名字，之后接的是消息所带的参数，参数可能有 0 个或多个。

到现在我仍然认为，将函数调用设计成发消息是一个非常有趣的设计，因为它意味着在这种设计上可以为语言本身增加很多动态性的设计。比如：因为消息的名字其实是字符串，所以 Objective-C 语言就可以用变量来传递这个字符串，进而可以实现一些运行时动态调用，语言提供的 `NSSelectorFromString` 就是协助我们将字符串转换成可以调用的 `Selector`。另外因为消息调用都是最终到了 `objc_msgSend` 而不是具体的函数地址，所以这也使得 Objective-C 可以支持方法的动态替换，这些动态性使得 Objective-C 语言变得异常灵活，所以才会出现像 JSPatch 这样优秀的动态补丁下发框架。

但是，这种调用方式因为不是将函数地址硬编码到代码中，所以在调用时，Objective-C 语言需要查表才能够获得真实的调用地址。在苹果的这篇 官方文档 中，苹果详细介绍了整个方法调用时，函数地址的查询过程。苹果也发现这样调用起来很慢，所以为了加速，它会缓存下来方法调用的查询结果，但是即使这样，相对于直接的调用，性能上肯定还是会有一些折扣。

Swift 语言在设计的时候直接放弃了 Objective-C 语言的这个机制。在这一点上，Swift 算是和其它流行的编程语言保持了一致。可以明确的是，这样肯定会有性能上的提升。不过与此同时，Swift 确实也失掉了极大的动态特性，动态修改类的成员函数实现在语言层面上就被封禁死了。我相信在未来 Swift 语言还是会引入不少动态特性，不过这肯定不是 Swift 语言当前的首要目标。

面向协议编程的实现

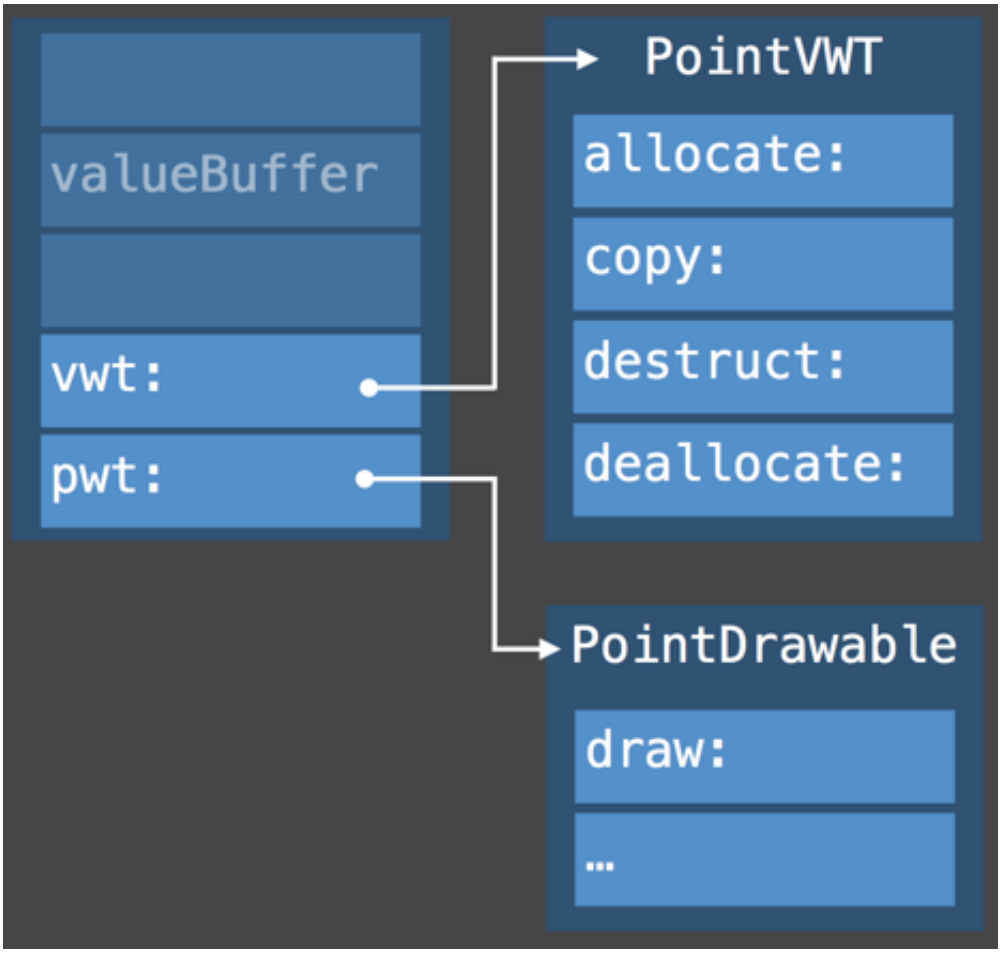
Swift 鼓励大家使用值类型，也鼓励大家使用协议。但是，这里面有一个不得不考虑的问题需要解决：如何将不同的值类型但是实现了同一个协议的实例，放到同一个数组当中？我们来看一个例子：

```
protocol Drawable {
    func draw()
}

struct Point: Drawable {
    var x: Int
    var y: Int
    init() {
        x = 1
        y = 2
    }
    func draw() {
        print("Point draw")
    }
}

struct Line: Drawable {
    var x1: Int
    var y1: Int
    var x2: Int
    var y2: Int
    init() {
        x1 = 5
        y1 = 6
        x2 = 7
        y2 = 8
    }

    func draw() {
        print("Line draw")
    }
}

let a: Drawable = Point()
a.draw()

let b: Drawable = Line()
b.draw()

let arr: [Drawable] = [a, b]
```

在这个例子中，我们定义了一个协议 `Drawable`，然后值类型 `Point` 和 `Line` 都实现了这个协议。在代码的最后，我们申明了一个 `[Drawable]` 的数组，将 `Point` 和 `Line` 的实例都放到了同一个数组中。因为 `Point` 实例的大小是 2 个 word（每个 word 8 字节），`Line` 实例的大小是 4 个 word，所以最后我们定义的 `Drawable` 数组就面临一个挑战：它需要把不同大小的元素放到同一个数组中。

这有什么问题呢？这表示我们无法很方便地定位元素。假如我们的数组真的是把不同大小的元素放到一个数组里面，那就意味着，如果我们想定位到第 i 个元素，我们需要把第 0 ~ i-1 个元素的大小都算出来，这样还可以算出第 i 个元素的内存偏移量。而如果每个元素的大小都是固定的，我们只需要用元素大小乘以 i，就可以算出偏移量来了。

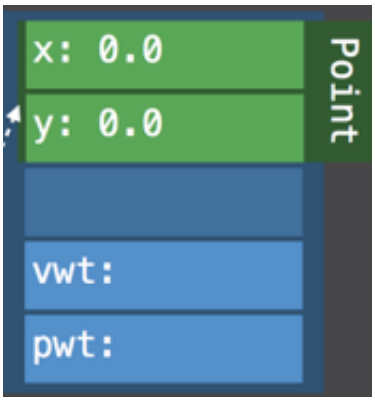
苹果想了一个办法来解决上面提到的问题：使用一个额外的容器（Container）来放每个带有协议的值类型，而数组里面放的，其实是每一个固定大小的容器。下面我就给大家介绍一下详细的实现细节。



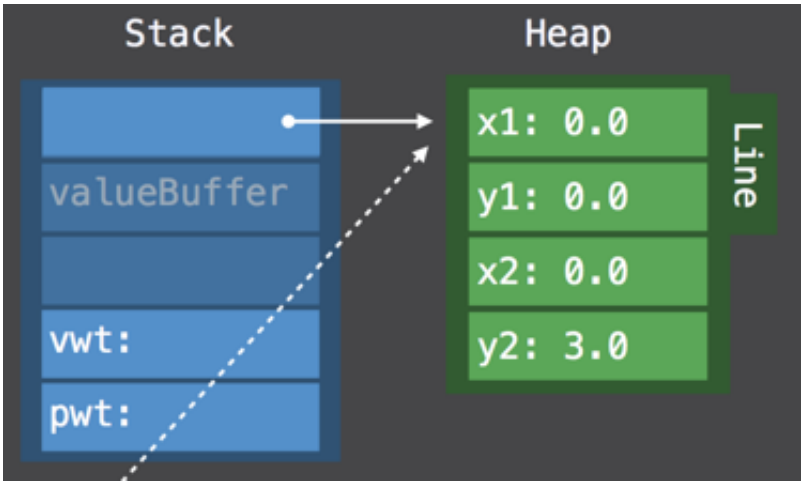
这个额外的容器是一个值类型，大小一共是 5 个 word。上图是该结构示意图，其中：

- 前 3 个 word 叫做 Value Buffer，是用于存放元素的值。
- 第 4 个 word 叫做 Value Witness Table，用于存放该值类型的创建，复制，回收时的函数地址。
- 第 5 个 word 叫做 Protocol Witness Table，用于存放协议（Protocol）对应的函数的实现函数地址。

我们知道这个结构中 Value Buffer 一共是 3 个 word，所以如果数组元素存放的是像 `Point` 这种实例的话，是可以直接存放进去的，就像下图这样。



但是，如果是存放像 `Line` 这种大于 3 个 word 大小的实例的话，Value Buffer 就不够用了，于是，Swift 会另外在堆中申请一块内存，将值复制过去，然后将这块内存的地址保存到 Value Buffer 的第 1 个 word 中，就像下图这样。



最终，这种设计使得：

- 数组中每个元素的大小都是固定的 5 个 word，解决了数组元素下标快速定位的问题。
- 因为有 Value Buffer 的存在，我们可以将不同大小的值类型存放到 Value Buffer 中，小于等于 3 个 word 的值直接存储，更大的则通过保存引用地址的方式存储。
- 通过 Value Witness Table，我们可以找到这个值类型的相关生命周期的管理函数。
- 通过 Protocol Witness Table，我们可以找到协议的具体实现函数的地址。

我们来看一个具体的例子，下图是一段使用值+协议来传递参数给函数的例子。

```
func drawACopy(local : Drawable) {
   local.draw()
}
let val : Drawable = Point()
drawACopy(val)
```

这段代码在编译器看来，会改写成如下这样，首先，编译器会生成一个 Container 的 struct:

```
struct ExistContDrawable {
   var valueBuffer: (Int, Int, Int)
   var vwt: ValueWitnessTable
   var pwt: DrawableProtocolWitnessTable
}
```

正如我们刚刚所说，这个 Container 一共 5 个 word。分别存放 Value Buffer，Value Witness Table 和 Protocol Witness Table。接着，这段代码会被改写成这样：

```
func drawACopy(val: ExistContDrawable) {
   // 创建容器
   var local = ExistContDrawable()
   // 设置 Value Witness Table 和 Protocol Witness Table
   let vwt = val.vwt
   let pwt = val.pwt
   local.vwt = vwt
   local.pwt = pwt
   // 利用 Value Witness Table 中的函数来赋值
   vwt.allocateBufferAndCopyValue(&local, val)
   // 利用 Protocol Witness Table 来调用 draw 函数
   pwt.draw(vwt.projectBuffer(&local))
   // 利用 Value Witness Table 中的函数来释放内存
   vwt.destructAndDeallocateBuffer(&local)
}
```

从注释中可以比较清楚地看到，两个 witness table 很好地协助了值类型进行赋值和函数调用。

用 LLDB 来观察 Container 的内存布局

我们可以用 LLDB 的相关执令，在运行时 dump 出变量的内存布局，以便验证 Container 和 Witness table 的存在。

在介绍过程之前，我先介绍几个实用的 LLDB 命令。限于篇幅和重点，这里只做简单的介绍，详细的可以翻查 LLDB 的官方文档。

- `breakpoint set -a address`，该命令可以在指定位置设置断点，设置之后可以使用 `continue` 来运行到断点处。
- `x -s8 -c5 -fx address`，该命令可以读出 address 所指向地址的内存。
- `di -s address -c 10`，该命令可以反汇编 address 所指向的地址开始的汇编代码。
- `re r rdi rsi rdx rcx rax`，该命令可以输出相关寄存器的值以及值表示的内容。

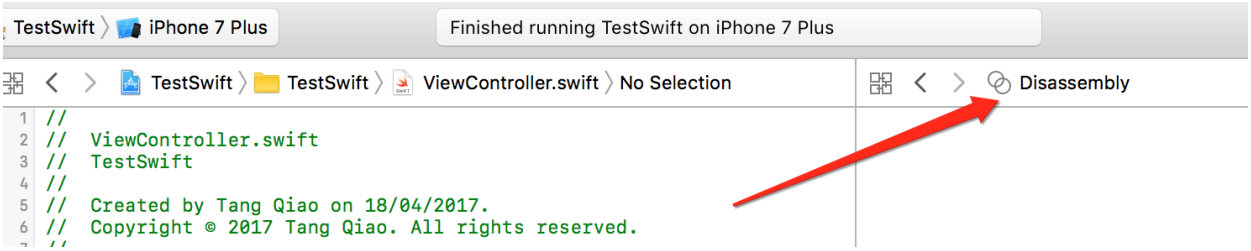
好了，接下来我们看看实验用的代码：

```
protocol Drawable {
    func draw()
}

struct Point: Drawable {
```

```
    var x: Int
    var y: Int
    init() {
        x = 1
        y = 2
    }
    func draw() {
        print("Point draw")
    }
}

struct Line: Drawable {
    var x1: Int
    var y1: Int
    var x2: Int
    var y2: Int
    init() {
        x1 = 5
        y1 = 6
        x2 = 7
        y2 = 8
    }

    func draw() {
        print("Line draw")
    }
}

func outputArray(_ arr: [Drawable]) {
    print("output array");
}

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        let a: Drawable = Point()
        a.draw()

        let b: Drawable = Line()
        b.draw()

        let arr: [Drawable] = [a, b]
        outputArray(arr)
    }
}
```

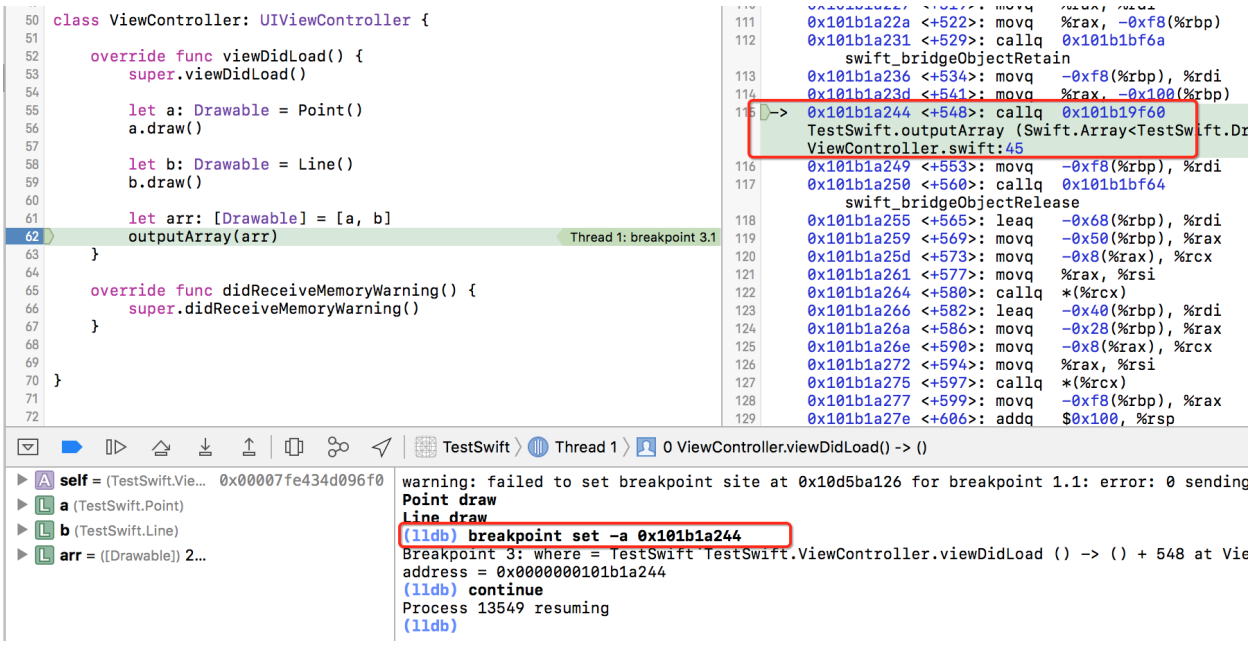
在这段代码中，我们定义了一个名为 Drawable 的 protocol，然后创建了一个 Drawable 的数组实例，然后我们将代码在 `outputArray(arr)` 处设置断点，然后运行我们的代码。

为了方便我们观察反汇编的代码，我们可以将 Assistant Editor 的显示切换成显示汇编码，如下所示：



观察汇编码，我们发现真实的函数调用其实在断点之后，如下图所示：

所以，我们先使用刚刚介绍的 `breakpoint` 命令，执行 `breakpoint set -a 0x101b1a244`，设置断点在那个位置，然后执行 `continue`，如下所示：



需要介绍的是，在函数调用时，$rdi 寄存器会保存参数的值，所以我们可以用 `x` 命令将 $rdi 寄存器地址的 dump 读出来。这样我们得到的结果如下：

```
(lldb) x -s8 -c20 -fx $rdi
0x6000000c9760: 0x00007fe434f01358 0x0000000200000008
0x6000000c9770: 0x0000000000000002 0x0000000000000004
0x6000000c9780: 0x0000000000000001 0x0000000000000002
0x6000000c9790: 0x0000000000000000 0x0000000101b1e2b8
0x6000000c97a0: 0x0000000101b1e200 0x0000600000031100
0x6000000c97b0: 0x0000000000000000 0x0000000000000000
0x6000000c97c0: 0x0000000101b1e388 0x0000000101b1e208
0x6000000c97d0: 0x0000000000000000 0x0000000000000000
0x6000000c97e0: 0x0000000000000000 0x0000000000000000
0x6000000c97f0: 0x0000000000000000 0x0000000000000000
```

这里的前 4 个 word 是数组的结构，接下来从 `0x6000000c9780` 开始的 5 个 word 即为第一个值，它的值为

```
0x6000000c9780: 0x0000000000000001 0x0000000000000002
0x6000000c9790: 0x0000000000000000 0x0000000101b1e2b8
0x6000000c97a0: 0x0000000101b1e200
```

然后紧接着的是第二个值，它的内容是：

```
0x6000000c9780:                    0x0000600000031100
0x6000000c97b0: 0x0000000000000000 0x0000000000000000
0x6000000c97c0: 0x0000000101b1e388 0x0000000101b1e208
```

很有意思的是，我们就可以很清晰地看到，第一个容器的 Value Buffer 中放的1 和 2 刚好是 Point 的值，而第二个容器的 Value Buffer 中只有第一个变量有值，后面为空，我们猜测这就是 Line 在堆中的地址。我们可以继续用 `x` 命令将这个地址的值 dump 出来检查，结果如下：

```
(lldb) x -s8 -c4 -fx 0x0000600000031100
0x600000031100: 0x0000000000000005 0x0000000000000006
0x600000031110: 0x0000000000000007 0x0000000000000008
```

我们顺利从这个地址中读到了 Line 的值：5，6，7，8，印证了我们的想法。

我们还可以用 `di` 命令来反汇编这两个容器的第 5 个 Word，看是否是 Protocol Witness Table 的地址，结果如下：

```
(lldb) di -s 0x0000000101b1e200 -c 2
TestSwift`protocol witness table for TestSwift.Point : TestSwift.Drawable in
TestSwift:
    0x101b1e200 <+0>:  addb   %bl, 0x101(%rcx,%rsi,4)
    0x101b1e207 <+7>:  addb   %dl, (%rax)

(lldb) di -s 0x0000000101b1e208 -c 2
TestSwift`protocol witness table for TestSwift.Line : TestSwift.Drawable in
TestSwift:
    0x101b1e208 <+0>:  adcb   %bl, 0x101b1(%rdi)
    0x101b1e20e <+6>:  addb   %al, (%rax)
```

从输出结果中 LLDB 给我们的提示信息来看，我们顺利找到了相关的函数地址。

总结

Swift 语言内部使用了大量创新的设计，使得其在性能上能够相对 Objective-C 语言产生超越，并且它能够很好地支持值类型和面向协议编程。使用 LLDB 相关的指令，可以很方便地帮助我们理解 Swift 对象的内部内存结构，加深理解。

监控SDK的架构设计

作者：王朝成

0x00 引言

随着柯洁对战AlphaGo的败北，`人工智能`一词进入了越来越多人的视界。一时间似乎人人都在谈论`人工智能`，风口似乎正在向这一块领域进行倾斜。不过今天我们来谈的不是`人工智能`，而是`人工智能`背后的`大数据`。

构建`人工智能`基础的机器学习，需要大量的数据进行训练；即要有足够的数据量来`喂饱`我们的机器学习算法，以让它更加的`智能`。

那么问题来了，如何获取到我们想要的`大数据`？有读者可能第一时间就会想到现在非常火的爬虫技术。没错，爬虫可以帮我们获取到大量的公开数据。然而对于我们自身的业务、性能等信息却无用武之地。因此我们自然会想到，自己构建一个监控或者说埋点的SDK，来获取用户行为或者性能分析的数据。这样便能够最大限度地发挥数据的作用。

本文会结合一些经验来谈谈以下几个方面的内容：

1. 如何架构一个iOS的监控SDK?
2. 如何实现自动采集?

0x01 iOS下的监控设计架构

俗话说得好，脱离业务的实践来谈架构就是一种耍流氓。因此我们先来看看一种监控业务的场景。

其实采集用户数据的场景无外乎两种：APM和UBT。APM(Application Performace Management)倾向于数据性能的采集，而UBT(User Behavior Track)则更倾向于业务数据的分析。这二者之间有一定的共同性，都是从业务层获取到数据，然后持久化在手机本地，再寻找相应的时机发送给服务器端，完成一次数据同步的工作。

基于以上的数据流转过程，我们主要会从以下几点来展开讨论：

1. 数据持久化相关的设计
2. 数据发送传输相关的设计
3. 采集相关的设计

数据持久化相关的设计

先来看看数据持久化方面的内容。在这一小节里，我们主要会面临以下几个问题：

- 为什么需要数据持久化
- 数据持久化方案的选择
- 如何设计数据持久化层

为什么需要数据持久化?

对于移动端设备，特别是iOS设备而言，用户在操作`APP`时，随时都有可能按下Home键回到桌面，或者手动杀死应用。在这一种场景下，倘若采集到的数据一直存在于设备内存中，iOS系统则可能随时清空该应用内存，最终导致我们采集的数据丢失。因此为了减少数据损耗，在数据采集完成后的第一时间进行数据持久化，似乎显得额外的重要。

当然也有读者会疑惑，如果我们在采集完成后第一时间就同步到服务器，是否就可以减少这一个步骤？值得肯定的是，这种场景的确存在，但却仅存在于100%网络正常且服务100%正常情况下的理想国。当然，如果说所有的数据仅仅用于采样分析，并可以忍受部分数据的丢失，那么这种去数据持久化的方案也并非不能接受。

数据持久化方案的选择

接下来再来看看数据持久化的方案。在iOS平台上，目前比较流行的数据持久化方案有以下几种：

- Core Data
- Sqlite
- Realm

选择哪种方案更好？其实是个永远都在争执的话题。`Core Data`是Apple原生推荐和支持的方式，但学习曲线陡峭，很多开发者并不大喜欢。Sqlite也支持，但使用较为复杂；开源社区上`FMDB`是一个不错的封装好的选择。而`Realm`则是最近一个比较火的跨平台的数据持久化中间件，拥有不错的社区支持，也不失为另一种选择。当然也有直接使用文件系统这种方式去处理数据，也是一种尝试。

那么，究竟如何选择？其实更取决于使用它的人。如果团队成员对`Core Data`都比较推崇，那完全可以全部采取这样一种方式；如果团队成员比较喜欢尝鲜，也有不错的探索精神，则`Realm`也会是个不错的选择；而如果中规中矩，又不想花那么多的时间去熟悉`Core Data`的框架，那选择`Sqlite`也并没有太多的过错。

如何设计数据持久化层

对于采集到的监控数据而言，需要注意到的地方是，这部分数据都是要最终同步到远端服务器上。因此，本地的持久化数据相对于服务器端而言，是某种意义上的缓存。所以，我们可以根据这样一个点，来思考如何设计这类数据的输入、输出和存储的问题。

很快会有读者疑惑，不就是根据数据库API来提供接口给上层调用么？哪里来的这么多讲究？

这里其实存在两种截然不同的数据存储方式：

1. 格式化存储
2. 二进制存储

这是什么概念？

格式化存储方式类似服务器端的数据库表结构设计。把整个数据模型在数据库里进行映射存储。这种存储方式的好处在于Debug相对简单，比如选择了Sqlite作为数据库解决方案，那么就可以直接在相应的文件夹下打开这个创建的sqlite数据库文件，查看当前的数据情况，从而发现问题。但是这种方式也存在其先天缺陷，那就是如果一旦数据结构做了变更，那么整个数据库的表结构也要有相应的变更。其二，当表结构变更发生在数据量较多的情况下时候，进行数据升级也会占据较多的时间。其三，其可扩展性也相对较差；如果一旦未来需要新增某种监控数据类型，那表结构也就需要做相应的变更。

二进制存储则是直接将`Blob`的二进制数据存储在本地数据库中。如即将准备发送的`JSON`数据，或者`ProtoBuff`数据等。这种设计的初衷点就在于之前提到的`缓存`二字。既然我们本地的数据仅仅只是远端数据的一次缓存，那么本地就不应该去关注或者消费这些数据。因此，并不需要太过关注其数据本身的数据结构，而是直接保存要发送的二进制即可。这种设计方案好处在于可扩展性强，不论与服务器端通信的协议如何变，数据库都不需要发生变更，都是存储二进制数据；同时由于字段少，所以吞吐率也相对较高。但这种设计也存在一定的风险，那就是`Debug`时的耗时需要增加，需有相应的工具来反序列化数据库里的数据才能知道数据存储正确与否；同时这种设计方案也不能更新数据库内不正确的数据值，就算发现错误也难以修正。

因此建议在一开始时还是采用格式化存储方式来做存储。当方案成熟后，可以考虑切换到二进制存储来提高扩展性和性能。

以上讨论了一下数据的存储方式，那么接下来我们再来讨论一下数据的写和读的问题。

我们都知道，数据库经常会遇到读和写同时进行的问题，这时候就会导致数据的不一致。解决这个问题的方案有许多种，也是我们老生常谈的那些，例如读写锁、互斥锁、信号量、队列等等。个人比较倾向和喜欢用一个串行队列的方式来解决这个问题。一来实现起来较为简便，二来上层调用到了数据库层就将接下来的任务交由子线程队列处理，可以从某种意义上防止主线程的采集卡顿问题。

综合以上的讨论，我们似乎可以看出些端倪。在设计数据持久化层时，可以做两层结构处理；底层是提供数据库数据的`CRUD`功能的层次，上层则提供数据库序列化（如果有）和读写队列的功能。

数据发送传输相关的设计

数据的发送传输层主要目的则是用于完成取数据，装数据，和发数据三个过程。主要需要考虑的是以下几个方面：

- 数据的传输协议
- 数据的序列化方式
- 发送间隔策略

数据的传输协议

在数据的传输协议上，主要考量的是，我们需要长连接还是短连接?

先来说说长连接。我们这里的长连接主要还是指的TCP层的Socket的长连接。由于每个TCP的连接都需要三次握手，这就会有相应的时间消耗。如果每次的发送都是`先连接后发送`，那么处理的速度会降低很多；所以每次的操作后都不断开，多次发送数据时候直接发送数据包会节省很多TCP建立的开销。

但这种方式同时也会带来巨大的服务器资源——维护这些连接所消耗的大量内存。试想一下成千上万甚至上亿的客户端连接，需要多少的服务器资源。当然，如果公司已经存在一条统一的长连接通道，那么一致地向该通道发送数据也不失为一种好的方式。

再说说短连接，狭义上的短连接我们通常指的HTTP协议，发送完数据后就将其连接关闭。这种好处在于代码上客户端和服务器端都易于实现，能达到快速开发的效果。不足之处可能就在于每次开启一次请求时，都需要重新建立连接，从而导致额外的开销。

至于如何选择，还是需要视整体情况。不过如果在公司没有统一的长连接网关的情况下，建议暂时可以采取HTTP/1.1 Keep Alive来进行过渡，当然最新HTTP/2也是个很不错的选择。

数据的序列化方式

谈完了协议，再谈谈数据如何传输的问题。这主要包含了两个方面：数据的序列化方式和数据的压缩方式。

对于数据的序列化算法，当前比较流行的几种方式有`JSON`，`ProtoBuff`，`Thrift`等；而对于压缩方式而言，则存在`Gzip`，`7z`等多种方案。

由于监控的数据，存在发送间隔短、发送次数多的特点，因此如何减少用户的额外消耗流量是数据的序列化主要考虑的问题。在这些协议的测试过程中，`ProtoBuff`+`7z`的组合获得了最大的数据压缩率。不过同时也带来了不小的二进制包开销，如`7z`的算法库本身就有好几M的大小；因此需要在各自范围内的时间复杂度和空间复杂度中根据需求做一个权衡。毕竟适合自己的才是最好的。

发送间隔策略

提到发送间隔，一部分开发者可能会拍脑袋构思出以下几种设计：

1. 若干秒一次发送或者若干条数据满了之后一直尝试发送直到发送成功
2. 若干秒一次发送，如果不成功则发送间隔乘以2以此类推。直到发送成功发送间隔恢复初始

实时上，这两种发送间隔策略在实践的过程中都存在一些问题。首先我们来看看第一条。

第一种发送策略会在平时正常的服务器状态下，风平浪静。然而一旦网络发生一丝抖动，所有活跃客户端都会不断地尝试建立连接和发送数据给我们的服务器。这种情况下相当与发生了一次雪崩，即让我们自己的客户端发起了一次针对我们自己服务器的`DDOS`攻击。

再来看第二种发送策略，相对第一种而言已经有一定的进步,因为发送失败后会有一定的时间后退来进行重试。然而在实践过程中发现，一旦发送网络抖动，客户端还是会在以上约定的偶数时间上来对自己的服务器进行小规模的`DDOS`攻击，问题仍然没有得到有效地解决。

难道无解了么？其实不然，多年前的`CSMA/CD`，即`冲突检测的载波监听多路访问的方法`，给了我们一些启发。它的工作原理如下（摘自Wiki)：

> 发送数据前，先侦听信道是否空闲，若空闲，则立即发送数据。若信道忙碌，则等待一段时间至信道中的信息传输结束后再发送数据；若在上一段信息发送结束后，同时有两个或两个以上的节点都提出发送请求，则判定为冲突。若侦听到冲突,则立即停止发送数据，等待一段随机时间,再重新尝试。

这里我们注意到一段文字`等待一段随机时间`。其实`CSMA/CD`有自己的算法，那么我们是否也能根据这种思想来更深层地做一次优化，将重试的时间分散在一段时间区间内？答案是肯定的，事实证明这也能有效降低服务器在网络抖动时的负载压力。

采集相关的设计

以上基本完成了一款监控SDK的消费者部分的设计，现在来看看生产者部分。

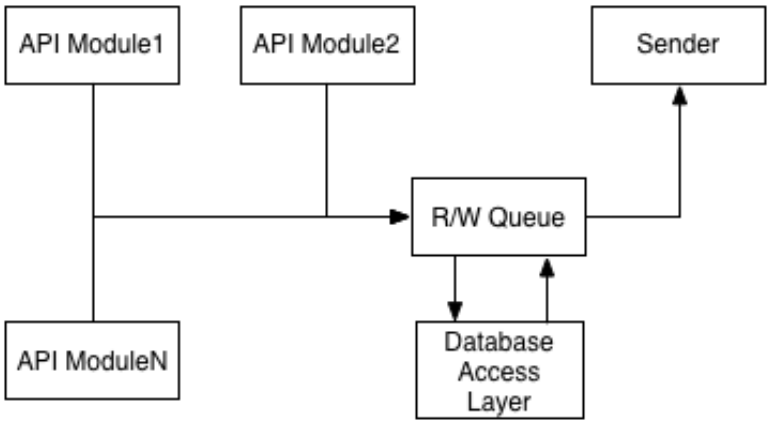
一般来说，采集分为手动和自动两种。这里我们先来谈一下手动这个方面。手动采集一般更多的还是对业务场景的抽象，如一个事件、一次请求、又或者一个页面的记录。因此手动采集的设计更像是一个API接口的设计。不同的业务提供一个API接口，比如以下提供的页面和事件的API接口。

```
- (void)trackPageViewWithName:(NSString * _Nonnull)name
                       params:(NSDictionary * _Nullable)params;

- (void)trackEventWithID:(NSString * _Nonnull)eventID
                    page:(NSString * _Nullable)page
                  params:(NSDictionary * _Nullable)params;
```

通过提供诸如此类的接口，再将业务层传输下来的数据进行存储，基本就能够满足一款普通的监控采集SDK的需求了。

小结

以上一个简单的监控SDK的基本架构也就出来了：



不同的API模块将其自身对应的数据发送给数据库读写队列进行存储，而发送模块则定时来数据库查询是否有需要发送的数据并将其同步至服务器。

0x02 自动采集

以上设计好了手动采集数据的SDK，那么，能否实现自动采集呢？或者说，如何在iOS平台上实现自动的采集呢?

在这里，我们首先要从技术层面来找是否存在可行性。既然名为自动采集，那么肯定是在不惊动原代码的前提下来进行的。既然不能动原代码，那么肯定是要用到一些`AOP`的技术了。因此，让我们先来看看iOS平台上的一些`AOP`技术。

周所周知，iOS下有两大语言Objective-C和Swift。同时我们也都清楚Objective-C是一门动态语言，并且Swift在ABI稳定之前都是通过动态库的形式链接进整个iOS系统的。因此要做到`AOP`主要还是从Objective-C上着手会比较方便。而`Objective-C`的`Method Swizzling`已经被我们用的滚瓜烂熟了，自然成为了我们的`AOP`首选。但对于再底层的一些C语言库，这种方式就不适用了，这时候Facebook大神们出的`Fishhook`可以从MachO的角度来进行一些`AOP`，可以作为部分的补充。

一个普通的切面代码可以如下：

```
+ (void)swizzingClassMethodWithOriginClass:(Class)oriCls
                            originSelector:(SEL)originSelector
                               targetClass:(Class)targetCls
                            targetSelector:(SEL)targetSelector {

    NSParameterAssert(oriCls);
    NSParameterAssert(targetCls);
    NSParameterAssert(originSelector);
    NSParameterAssert(targetSelector);
    if ([oriCls instancesRespondToSelector:targetSelector])
        return;

    Method swizzledMethod = class_getClassMethod(targetCls, targetSelector);

    if (sk_addMethod(object_getClass(oriCls), targetSelector,
swizzledMethod)) {
        sk_swizzleClassSelector(object_getClass(oriCls), originSelector,
targetSelector);
    }
}

+ (void)swizzingInstanceMethodWithOriginClass:(Class)oriCls
                               originSelector:(SEL)originSelector
                                  targetClass:(Class)targetCls
                               targetSelector:(SEL)targetSelector {
    NSParameterAssert(oriCls);
    NSParameterAssert(targetCls);
    NSParameterAssert(originSelector);
    NSParameterAssert(targetSelector);
    if ([oriCls instancesRespondToSelector:targetSelector])
        return;

    Method swizzledMethod = class_getInstanceMethod(targetCls,
targetSelector);

    if (sk_addMethod(oriCls, targetSelector, swizzledMethod)) {
        sk_swizzleInstanceSelector(oriCls, originSelector, targetSelector);
    }
}
```

如果技术上可行性没有问题，那么业务层是否可行呢？这里我们从两块不同的数据类型来分别探讨一下这个问题：

- 性能监控数据
- 业务埋点数据

性能监控数据

由于性能监控数据种类繁多，这里以网络请求为例。

对于网络请求的性能，我们主要关心的是其在TCP/IP整个协议栈上每个阶段的时间，以及HTTP请求的包大小。了解到需求后，我们便可以着手来看看从哪儿入手。

我们都知道，iOS平台的HTTP网络请求主要都集中于于`urlconnection`和`urlsession`两个大库（基于CFNetworking的实现由于篇幅问题暂不在此展开讨论），因此可以从这个源头开始入手，`AOP`掉主要的一些函数，下面代码以`urlsession`为例：

```
+ (void)_swizzleNSURLSession {
    // Swizzle NSURLSession method
    NSArray<NSString *> *selectors = @[
                                       // asnyc selector

NSStringFromSelector(@selector(dataTaskWithRequest:completionHandler:)),

NSStringFromSelector(@selector(downloadTaskWithRequest:completionHandler:)),
                                       // sync selector

NSStringFromSelector(@selector(downloadTaskWithRequest:))

                                       ];
    for (NSString *selector in selectors) {

                ...

        while (class_getInstanceMethod(currentClass, originSelector)) {
            if (realRespondsToSelector(originSelector, currentClass)) {
                [FGRSwizzlingUtils
swizzingInstanceMethodWithOriginClass:currentClass

originSelector:originSelector
                                                                    targetClass:
[FGRURLSessionBlender class]

targetSelector:targetSelector];
            }
            ...
        }

        [FGRSwizzlingUtils swizzingInstanceMethodWithOriginClass:[session
class]

originSelector:originSelector
                                                          targetClass:
[FGRURLSessionBlender class]

targetSelector:targetSelector];
        ...
    }
```

```
        ...

    for (NSString *classSelector in classSelectors) {

        ...

        SEL originSelector = NSSelectorFromString(classSelector);
        SEL targetSelector = NSSelectorFromString(targetSelectorString);

        [FGRSwizzlingUtils swizzingClassMethodWithOriginClass:[NSURLSession
class]

originSelector:originSelector
                                                        targetClass:
[FGRURLSessionBlender class]

targetSelector:targetSelector];
    }
}
```

当然这里其实我们需要注意一个点，就是iOS10及其以上，平台提供给我们一套API回调方便我们获取相关的数据：

```
/*
 * Sent when complete statistics information has been collected for the task.
 */
- (void)URLSession:(NSURLSession *)session task:(NSURLSessionTask *)task
didFinishCollectingMetrics:(NSURLSessionTaskMetrics *)metrics
API_AVAILABLE(macosx(10.12), ios(10.0), watchos(3.0), tvos(10.0));
```

但对于iOS7, 8, 9来说，想获取具体的`DNS时间`，`TCP时间`，`SSL时间`这些详细Metrics并不那么容易，需要通过Fishhook底层的C库来达到目的。但可惜的是，在iOS9下，我们似乎通过这招也没法切到SSL相关的数据。如果读者有什么好方法希望能够不吝赐教。

```
uint32_t c = _dyld_image_count();
    for (uint32_t i = 0; i < c; i++) {
        const char *name = _dyld_get_image_name(i);
        NSString *imageName = [NSString stringWithUTF8String:name];
        if ( [imageName rangeOfString:@"CFNetwork"].length != 0
            || [imageName rangeOfString:@"libsystem_network.dylib"].length !=
0
            || [imageName rangeOfString:@"Security"].length != 0) {

            rebind_symbols_image((void *)_dyld_get_image_header(i),
_dyld_get_image_vmaddr_slide(i), (struct rebinding[7]) {
                {"getpeername", (void *)Fragarach_getpeername, (void
**)&FragarachOriginal_getpeername},
                {"dup", (void *)Fragarach_dup, (void
**)&FragarachOriginal_dup },
                {"write", (void *)Fragarach_write, (void
**)&FragarachOriginal_write },
                {"read", (void *)Fragarach_read, (void
**)&FragarachOriginal_read },
                {"SSLWrite", (void *)Fragarach_SSLWrite, (void
**)&FragarachOriginal_SSLWrite },
                {"SSLHandshake", (void *)Fragarach_SSLHandshake, (void
**)&FragarachOriginal_SSLHandshake },
                {"close", (void *)Fragarach_close, (void
**)&FragarachOriginal_close }
            }, 7);
        }

    }
```

当通过`AOP`方式hook住相关的函数后，我们便能够在每个函数节点通过存储时间戳的方式达到记录协议栈时间的目的。而对于请求的大小，则一般通过计算HTTP请求的头和Body的长度来达到总体长度的目的。

事实上性能数据的自动化采集是个非常大的范畴，由于篇幅原因只列举了基本的网络请求的自动化采集的方式，也希望能起到抛砖引玉的作用。

业务埋点数据

对于业务埋点来说，我们过去经常使用的一般还是手动地埋，这样比较精准而且可控，想要什么就有什么。但对于程序员工程师来说，总是会去想如果能自动化就自动化去处理掉，因此才会去琢磨如何去进行业务的自动埋点。

对于自动化业务埋点，一般需要解决以下几个问题：

1. 位置
2. 操作
3. 参数

对于第一个问题，基本大家都通过`xPath`这种方式来进行定位。对于同一个页面里很多种相同的元素则通过UIButton[1], UIButton[2]这种方式来进行区分。由于有很多的文章都对这种方式进行了介绍，这里就不再浪费篇幅进行介绍。

第二个问题，操作。其实对于iOS平台的用户行为而言，所有的行为都能归为以下几个大类：

- ViewAppear&ViewDisappear
- Action-Target
- ReuseCells
- Gestures
- ApplicationStatus（如AppDidLaunch 等）

所以，通过对这几种主要行为进行Hook，基本能达到捕获绝大多数行为操作的目的。

第三个问题，参数。对于大多数操作，我们其实都能捕获函数触发时带的参数。然而对于一些埋点，大数据团队更加希望能够获取触发该操作时的一些用户状态信息。这时候，对于自动埋点而言可能就并不那么适用了。有一种做法是，SDK提供若干个全局的参数列表，每次的操作记录都带上这些参数的瞬时值，从而解决一部分这样的问题。

然而不幸的是，总是有一部分数据采集的需求是比较难实现或者说自动采集实现起来比较麻烦的，比如曝光率、非全局的瞬时参数值。这时候可能还是需要结合一些手动的埋点才能最终达到目的。

其实对于业务数据的自动采集而言，更多的时候需要在后端大数据的产品进行支持，从多个维度进行分析才能真正获取到更多有价值的信息！

0x03 结语

历史车轮总是在不断地前进，技术、架构也同样在不断地前行。我们总是想着能否再智能一点，再自动化一点。于是有了以上的文字。

希望本文能对读者如何设计一个监控SDK有所启发！

初识 LLVM

作者：@Alone_Monkey

只要你和代码打交道，了解编译器的工作流程和原理定会让你受益无穷，无论是分析程序，还是基于它写自己的插件，甚至学习一门全新的语音。通过本文，将带你了解 LLVM，并使用 LLVM 来完成一些有意思的事情。

一、什么是 LLVM?

> The LLVM Project is a collection of modular and reusable compiler and toolchain technologies.

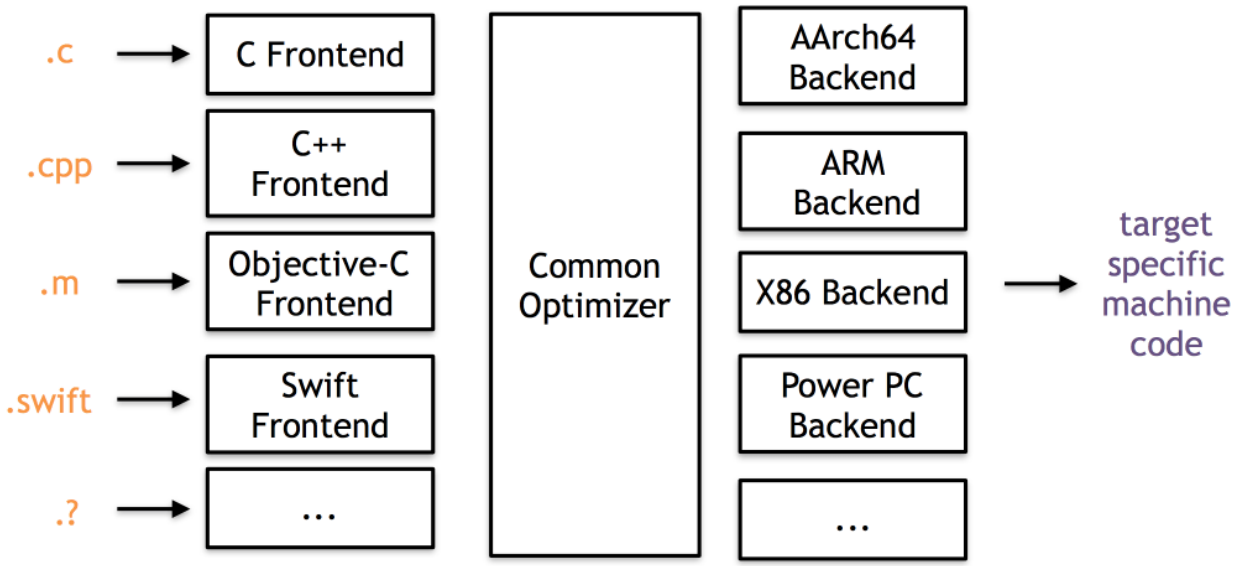
简单来说，LLVM 项目是一系列分模块、可重用的编译工具链。它提供了一种代码编写良好的中间表示 (IR)，可以作为多种语言的后端，还可以提供与变成语言无关的优化和针对多种 cpu 的代码生成功能。

先来看下 LLVM 架构的主要组成部分：

- 前端：前端用来获取源代码然后将它转变为某种中间表示，我们可以选择不同的编译器来作为 LLVM 的前端，如 gcc，clang 。
- Pass (通常翻译为“流程”)：Pass 用来将程序的中间表示之间相互变换。一般情况下，Pass 可以用来优化代码，这部分通常是我们关注的部分。
- 后端：后端用来生成实际的机器码。

虽然如今大多数编译器都采用的是这种架构，但是 LLVM 不同的就是对于不同的语言它都提供了同一种中间表示。传统的编译器的架构如下:

LLVM 的架构如下：



当编译器需要支持多种源代码和目标架构时，基于LLVM的架构，设计一门新的语言只需要去实现一个新的前端就行了，支持新的后端架构也只需要实现一个新的后端就行了。其它部分完成可以复用，就不用再重新设计一次了。

二、安装编译 LLVM

这里使用 clang 作为前端:

1.直接从官网下载:[http://releases.llvm.org/download.html
](http://releases.llvm.org/download.html)

2.svn 获取

```
svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm
cd llvm/tools
svn co http://llvm.org/svn/llvm-project/cfe/trunk clang
cd ../projects
svn co http://llvm.org/svn/llvm-project/compiler-rt/trunk compiler-rt
cd ../tools/clang/tools
svn co http://llvm.org/svn/llvm-project/clang-tools-extra/trunk extra
```

3.git 获取

```
git clone http://llvm.org/git/llvm.git
cd llvm/tools
git clone http://llvm.org/git/clang.git
cd ../projects
git clone http://llvm.org/git/compiler-rt.git
cd ../tools/clang/tools
git clone http://llvm.org/git/clang-tools-extra.git
```

最新的 LLVM 只支持 cmake 来编译了，首先安装 cmake 。

```
brew install cmake
```

编译:

```
mkdir build
cmake /path/to/llvm/source
cmake --build .
```

编译时间比较长，而且编译结果会生成20G左右的文件。

编译完成后，就能在`build/bin/`目录下面找到生成的工具了。

三、从源码到可执行文件

我们在开发的时候的时候，如果想要生成一个可执行文件或应用，我们点击 run 就完事了，那么在点击 run 之后编译器背后又做了哪些事情呢?

我们先来一个例子：

```
#import <Foundation/Foundation.h>

#define TEN 10

int main(){
    @autoreleasepool {
        int numberOne = TEN;
        int numberTwo = 8;
        NSString* name = [[NSString alloc] initWithUTF8String:"AloneMonkey"];
        int age = numberOne + numberTwo;
        NSLog(@"Hello, %@, Age: %d", name, age);
    }
    return 0;
}
```

上面这个文件，我们可以通过命令行直接编译，然后链接：

```
xcrun -sdk iphoneos clang -arch armv7 -F Foundation -fobjc-arc -c main.m -o
main.o
xcrun -sdk iphoneos clang main.o -arch armv7 -fobjc-arc -framework Foundation
-o main
```

拷贝到手机运行：

```
monkeyde-iPhone:/tmp root# ./main
2016-12-19 17:16:34.654 main[2164:213100] Hello, AloneMonkey, Age: 18
```

大家不会以为就这样就完了吧，当然不是，我们要继续深入剖析。

3.1 预处理（Preprocess）

这部分包括 macro 宏的展开，import/include 头文件的导入，以及 #if 等处理。

可以通过执行以下命令，来告诉 clang 只执行到预处理这一步：

```
clang -E main.m
```

执行完这个命令之后，我们会发现导入了很多的头文件内容。

```
......
# 1
"/System/Library/Frameworks/Foundation.framework/Headers/FoundationLegacySwif
tCompatibility.h" 1 3
# 185 "/System/Library/Frameworks/Foundation.framework/Headers/Foundation.h"
2 3
# 2 "main.m" 2

int main(){
    @autoreleasepool {
        int numberOne = 10;
        int numberTwo = 8;
        NSString* name = [[NSString alloc] initWithUTF8String:"AloneMonkey"];
        int age = numberOne + numberTwo;
        NSLog(@"Hello, %@, Age: %d", name, age);
    }
    return 0;
}
```

可以看到上面的预处理已经把宏替换了，并且导入了头文件。但是这样的话会引入很多不会去改变的系统库比如Foundation，所以有了 pch 预处理文件，可以在这里去引入一些通用的头文件。

后来Xcode新建的项目里面去掉了 pch 文件，引入了 moduels 的概念，把一些通用的库打成 modules 的形式，然后导入，默认会加上`-fmodules`参数。

```
clang -E -fmodules main.m
```

这样的话，只需要`@import`一下就能导入对应库的modules模块了。

```
@import Foundation;
int main(){
    @autoreleasepool {
        int numberOne = 10;
        int numberTwo = 8;
        NSString* name = [[NSString alloc] initWithUTF8String:"AloneMonkey"];
        int age = numberOne + numberTwo;
        NSLog(@"Hello, %@, Age: %d", name, age);
    }
    return 0;
}
```

3.2 词法分析 (Lexical Analysis)

在预处理之后，就要进行词法分析了，将预处理过的代码转化成一个个 Token，比如左括号、右括号、等于、字符串等等。

```
clang -fmodules -fsyntax-only -Xclang -dump-tokens main.m
```

```
annot_module_include '#import <F'               Loc=<main.m:1:1>
int 'int'        [StartOfLine]  Loc=<main.m:5:1>
identifier 'main'        [LeadingSpace] Loc=<main.m:5:5>
l_paren '('             Loc=<main.m:5:9>
r_paren ')'             Loc=<main.m:5:10>
l_brace '{'             Loc=<main.m:5:11>
at '@'   [StartOfLine] [LeadingSpace]   Loc=<main.m:6:5>
identifier 'autoreleasepool'            Loc=<main.m:6:6>
l_brace '{'      [LeadingSpace] Loc=<main.m:6:22>
int 'int'        [StartOfLine] [LeadingSpace]   Loc=<main.m:7:9>
identifier 'numberOne'   [LeadingSpace] Loc=<main.m:7:13>
equal '='        [LeadingSpace] Loc=<main.m:7:23>
numeric_constant '10'    [LeadingSpace] Loc=<main.m:7:25
<Spelling=main.m:3:13>>
semi ';'                Loc=<main.m:7:28>
int 'int'        [StartOfLine] [LeadingSpace]   Loc=<main.m:8:9>
identifier 'numberTwo'   [LeadingSpace] Loc=<main.m:8:13>
equal '='        [LeadingSpace] Loc=<main.m:8:23>
numeric_constant '8'     [LeadingSpace] Loc=<main.m:8:25>
semi ';'                Loc=<main.m:8:26>
identifier 'NSString'    [StartOfLine] [LeadingSpace]   Loc=<main.m:9:9>
star '*'                Loc=<main.m:9:17>
identifier 'name'        [LeadingSpace] Loc=<main.m:9:19>
equal '='        [LeadingSpace] Loc=<main.m:9:24>
l_square '['     [LeadingSpace] Loc=<main.m:9:26>
l_square '['            Loc=<main.m:9:27>
identifier 'NSString'           Loc=<main.m:9:28>
identifier 'alloc'       [LeadingSpace] Loc=<main.m:9:37>
r_square ']'            Loc=<main.m:9:42>
identifier 'initWithUTF8String'  [LeadingSpace] Loc=<main.m:9:44>
colon ':'               Loc=<main.m:9:62>
string_literal '"AloneMonkey"'          Loc=<main.m:9:63>
r_square ']'            Loc=<main.m:9:76>
semi ';'                Loc=<main.m:9:77>
int 'int'        [StartOfLine] [LeadingSpace]   Loc=<main.m:10:9>
identifier 'age'         [LeadingSpace] Loc=<main.m:10:13>
equal '='        [LeadingSpace] Loc=<main.m:10:17>
identifier 'numberOne'   [LeadingSpace] Loc=<main.m:10:19>
plus '+'         [LeadingSpace] Loc=<main.m:10:29>
identifier 'numberTwo'   [LeadingSpace] Loc=<main.m:10:31>
semi ';'                Loc=<main.m:10:40>
identifier 'NSLog'       [StartOfLine] [LeadingSpace]   Loc=<main.m:11:9>
l_paren '('             Loc=<main.m:11:14>
at '@'          Loc=<main.m:11:15>
string_literal '"Hello, %@, Age: %d"'           Loc=<main.m:11:16>
comma ','               Loc=<main.m:11:36>
identifier 'name'        [LeadingSpace] Loc=<main.m:11:38>
comma ','               Loc=<main.m:11:42>
identifier 'age'         [LeadingSpace] Loc=<main.m:11:44>
```

```
r_paren ')'             Loc=<main.m:11:47>
semi ';'                Loc=<main.m:11:48>
r_brace '}'      [StartOfLine] [LeadingSpace]   Loc=<main.m:12:5>
return 'return'  [StartOfLine] [LeadingSpace]   Loc=<main.m:13:5>
numeric_constant '0'     [LeadingSpace] Loc=<main.m:13:12>
semi ';'                Loc=<main.m:13:13>
r_brace '}'      [StartOfLine]  Loc=<main.m:14:1>
eof ''          Loc=<main.m:14:2>
```

3.3 语法分析 (Semantic Analysis)

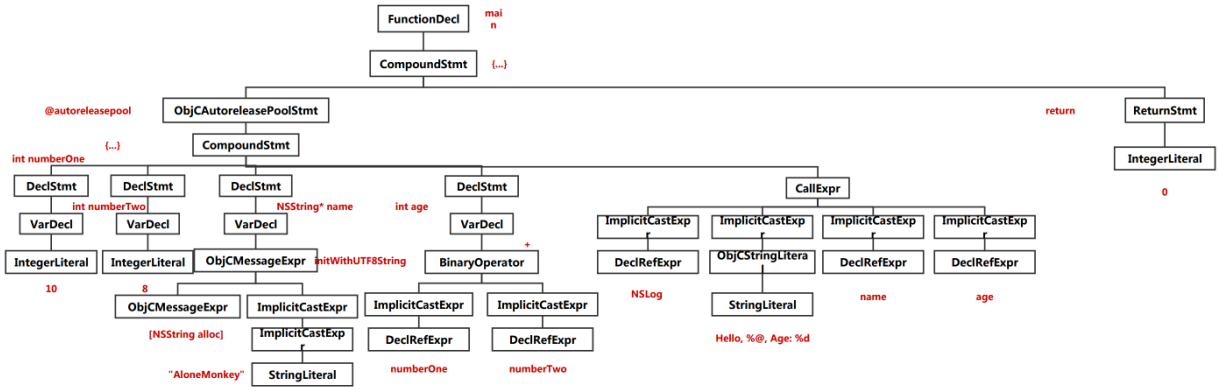
根据当前语言的语法，验证语法是否正确，并将所有节点组合成抽象语法树(AST)

```
clang -fmodules -fsyntax-only -Xclang -ast-dump main.m
```

```
......
`-FunctionDecl 0x7f8661d8a370 <main.m:5:1, line:14:1> line:5:5 main 'int ()'
  `-CompoundStmt 0x7f8661d8aab0 <col:11, line:14:1>
    |-ObjCAutoreleasePoolStmt 0x7f8661d8aa68 <line:6:5, line:12:5>
    | `-CompoundStmt 0x7f8661d8aa28 <line:6:22, line:12:5>
    |   |-DeclStmt 0x7f8661d8a4a0 <line:7:9, col:28>
    |   | `-VarDecl 0x7f8661d8a420 <col:9, line:3:13> line:7:13 used
numberOne 'int' cinit
    |   |   `-IntegerLiteral 0x7f8661d8a480 <line:3:13> 'int' 10
    |   |-DeclStmt 0x7f8661d8a550 <line:8:9, col:26>
    |   | `-VarDecl 0x7f8661d8a4d0 <col:9, col:25> col:13 used numberTwo
'int' cinit
    |   |   `-IntegerLiteral 0x7f8661d8a530 <col:25> 'int' 8
    |   |-DeclStmt 0x7f8661d8a6c0 <line:9:9, col:77>
    |   | `-VarDecl 0x7f8661d8a580 <col:9, col:76> col:19 used name 'NSString
*' cinit
    |   |   `-ObjCMessageExpr 0x7f8661d8a688 <col:26, col:76> 'NSString *
_Nullable':'NSString *' selector=initWithUTF8String:
    |   |     |-ObjCMessageExpr 0x7f8661d8a5f0 <col:27, col:42> 'NSString *'
selector=alloc class='NSString'
    |   |     `-ImplicitCastExpr 0x7f8661d8a670 <col:63> 'const char *
_Nonnull':'const char *' <BitCast>
    |   |       `-ImplicitCastExpr 0x7f8661d8a658 <col:63> 'char *'
<ArrayToPointerDecay>
    |   |         `-StringLiteral 0x7f8661d8a620 <col:63> 'char [12]' lvalue
"AloneMonkey"
    |   |-DeclStmt 0x7f8661d8a7f8 <line:10:9, col:40>
    |   | `-VarDecl 0x7f8661d8a6f0 <col:9, col:31> col:13 used age 'int'
cinit
    |   |   `-BinaryOperator 0x7f8661d8a7d0 <col:19, col:31> 'int' '+'
    |   |     |-ImplicitCastExpr 0x7f8661d8a7a0 <col:19> 'int'
<LValueToRValue>
    |   |     | `-DeclRefExpr 0x7f8661d8a750 <col:19> 'int' lvalue Var
```

```
0x7f8661d8a420 'numberOne' 'int'
    |   |     `-ImplicitCastExpr 0x7f8661d8a7b8 <col:31> 'int'
<LValueToRValue>
    |   |       `-DeclRefExpr 0x7f8661d8a778 <col:31> 'int' lvalue Var
0x7f8661d8a4d0 'numberTwo' 'int'
    |   `-CallExpr 0x7f8661d8a9a0 <line:11:9, col:47> 'void'
    |     |-ImplicitCastExpr 0x7f8661d8a988 <col:9> 'void (*)(id, ...)'
<FunctionToPointerDecay>
    |     | `-DeclRefExpr 0x7f8661d8a810 <col:9> 'void (id, ...)' Function
0x7f86618df0e0 'NSLog' 'void (id, ...)'
    |     |-ImplicitCastExpr 0x7f8661d8a9e0 <col:15, col:16> 'id':'id'
<BitCast>
    |     | `-ObjCStringLiteral 0x7f8661d8a8b8 <col:15, col:16> 'NSString *'
    |     |   `-StringLiteral 0x7f8661d8a878 <col:16> 'char [19]' lvalue
"Hello, %@, Age: %d"
    |     |-ImplicitCastExpr 0x7f8661d8a9f8 <col:38> 'NSString *'
<LValueToRValue>
    |     | `-DeclRefExpr 0x7f8661d8a8d8 <col:38> 'NSString *' lvalue Var
0x7f8661d8a580 'name' 'NSString *'
    |     `-ImplicitCastExpr 0x7f8661d8aa10 <col:44> 'int' <LValueToRValue>
    |       `-DeclRefExpr 0x7f8661d8a900 <col:44> 'int' lvalue Var
0x7f8661d8a6f0 'age' 'int'
    `-ReturnStmt 0x7f8661d8aa98 <line:13:5, col:12>
      `-IntegerLiteral 0x7f8661d8aa78 <col:12> 'int' 0
```

语法树直观图:



3.4 IR 代码生成 (CodeGen)

CodeGen 负责将语法树从顶至下遍历，翻译成 LLVM IR，LLVM IR 是 Frontend 的输出，也是 LLVM Backerend 的输入，桥接前后端。

可以在中间代码层次去做一些优化工作，我们在 Xcode 的编译设置里面也可以设置优化级别 `-O1`，`-O3`，`-Os`。还可以去写一些自己的 Pass，这里需要解释一下什么是 Pass。

Pass 就是 LLVM 系统转化和优化的工作的一个节点，每个节点做一些工作，这些工作加起来就构成了 LLVM 整个系统的优化和转化。

```
clang -S -fobjc-arc -emit-llvm main.m -o main.ll
```

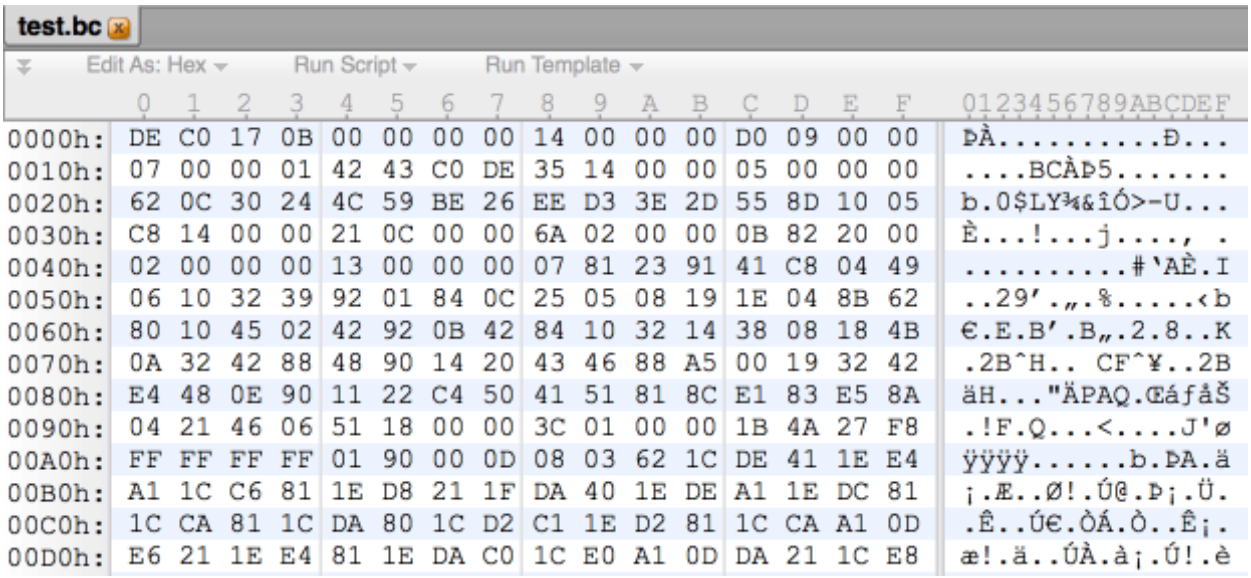
```
......
; Function Attrs: ssp uwtable
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %numberOne = alloca i32, align 4
  %numberTwo = alloca i32, align 4
  %name = alloca %0*, align 8
  %age = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  %0 = call i8* @objc_autoreleasePoolPush() #3
  store i32 10, i32* %numberOne, align 4
  store i32 8, i32* %numberTwo, align 4
  %1 = load %struct._class_t*, %struct._class_t**
@"OBJC_CLASSLIST_REFERENCES_$_", align 8
  %2 = load i8*, i8** @OBJC_SELECTOR_REFERENCES_, align 8, !invariant.load !7
  %3 = bitcast %struct._class_t* %1 to i8*
  %call = call i8* bitcast (i8* (i8*, i8*, ...)* @objc_msgSend to i8* (i8*,
i8*)*)(i8* %3, i8* %2)
  %4 = bitcast i8* %call to %0*
  %5 = load i8*, i8** @OBJC_SELECTOR_REFERENCES_.2, align 8, !invariant.load
!7
  %6 = bitcast %0* %4 to i8*
  %call1 = call i8* bitcast (i8* (i8*, i8*, ...)* @objc_msgSend to i8* (i8*,
i8*, i8*)*)(i8* %6, i8* %5, i8* getelementptr inbounds ([12 x i8], [12 x i8]*
@.str, i32 0, i32 0))
  %7 = bitcast i8* %call1 to %0*
  store %0* %7, %0** %name, align 8
  %8 = load i32, i32* %numberOne, align 4
  %9 = load i32, i32* %numberTwo, align 4
  %10 = sub i32 0, %9
  %11 = sub nsw i32 %8, %10
  %add = add nsw i32 %8, %9
  store i32 %11, i32* %age, align 4
  %12 = load %0*, %0** %name, align 8
  %13 = load i32, i32* %age, align 4
  notail call void (i8*, ...) @NSLog(i8* bitcast
(%struct.__NSConstantString_tag* @_unnamed_cfstring_ to i8*), %0* %12, i32
%13)
  %14 = bitcast %0** %name to i8**
  call void @objc_storeStrong(i8** %14, i8* null) #3
  call void @objc_autoreleasePoolPop(i8* %0)
  ret i32 0
}

declare i8* @objc_autoreleasePoolPush()

; Function Attrs: nonlazybind
declare i8* @objc_msgSend(i8*, i8*, ...) #1
```

```
declare void @NSLog(i8*, ...) #2

declare void @objc_storeStrong(i8**, i8*)

declare void @objc_autoreleasePoolPop(i8*)

......
!6 = !{!"clang version 4.0.0 (trunk 289913) (llvm/trunk 289911)"}
!7 = !{}
```

3.5 生成字节码 (LLVM Bitcode)

我们在 Xcode7 中默认生成bitcode就是这种的中间形式存在， 开启了 bitcode，那么苹果后台拿到的就是这种中间代码，苹果可以对 bitcode 做一个进一步的优化，如果有新的后端架构，仍然可以用这份 bitcode 去生成。

```
clang -emit-llvm -c main.m -o main.bc
```



3.6 生成相关汇编

```
clang -S -fobjc-arc main.m -o main.s
```

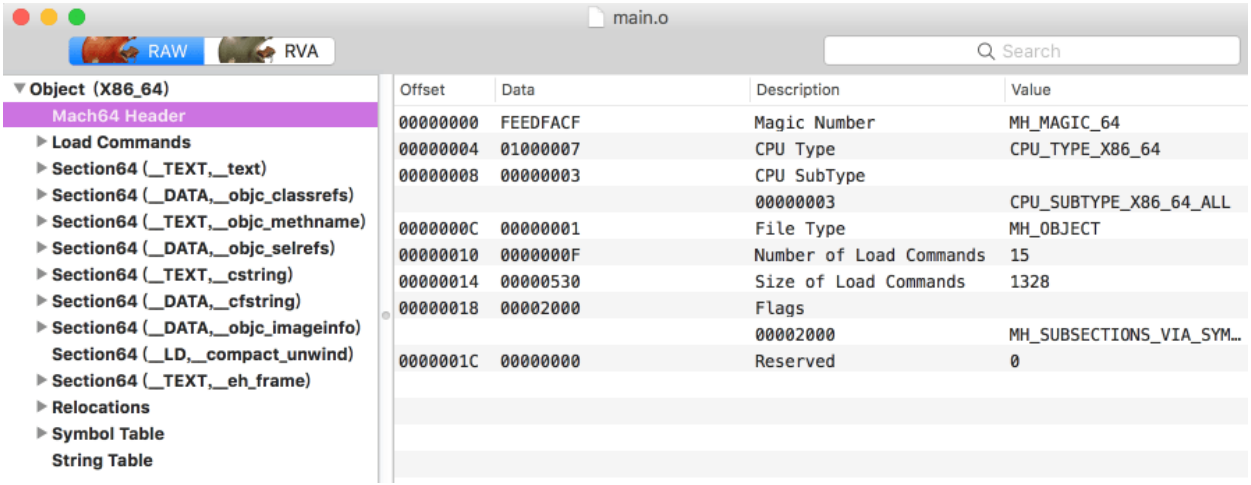
```
        .section        __TEXT,__text,regular,pure_instructions
        .macosx_version_min 10, 12
        .globl  _main
        .p2align        4, 0x90
_main:                                  ## @main
        .cfi_startproc
## BB#0:                                ## %entry
        pushq   %rbp
Lcfi0:
```

```
        .cfi_def_cfa_offset 16
Lcfi1:
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
Lcfi2:
        .cfi_def_cfa_register %rbp
        subq    $48, %rsp
        movl    $0, -4(%rbp)
        callq   _objc_autoreleasePoolPush
        movl    $10, -8(%rbp)
        movl    $8, -12(%rbp)
        movq    L_OBJC_CLASSLIST_REFERENCES_$_(%rip), %rcx
        movq    L_OBJC_SELECTOR_REFERENCES_(%rip), %rsi
        movq    %rcx, %rdi
        movq    %rax, -40(%rbp)         ## 8-byte Spill
        callq   _objc_msgSend
        leaq    L_.str(%rip), %rdx
        movq    L_OBJC_SELECTOR_REFERENCES_.2(%rip), %rsi
        movq    %rax, %rdi
        callq   _objc_msgSend
        leaq    L__unnamed_cfstring_(%rip), %rcx
        xorl    %r8d, %r8d
        movq    %rax, -24(%rbp)
        movl    -8(%rbp), %r9d
        movl    -12(%rbp), %r10d
        subl    %r10d, %r8d
        subl    %r8d, %r9d
        movl    %r9d, -28(%rbp)
        movq    -24(%rbp), %rsi
        movl    -28(%rbp), %edx
        movq    %rcx, %rdi
        movb    $0, %al
        callq   _NSLog
        xorl    %edx, %edx
        movl    %edx, %esi
        leaq    -24(%rbp), %rcx
        movq    %rcx, %rdi
        callq   _objc_storeStrong
        movq    -40(%rbp), %rdi         ## 8-byte Reload
        callq   _objc_autoreleasePoolPop
        xorl    %eax, %eax
        addq    $48, %rsp
        popq    %rbp
        retq
        .cfi_endproc

        .section        __DATA,__objc_classrefs,regular,no_dead_strip
        .p2align        3               ## @"OBJC_CLASSLIST_REFERENCES_$_"
L_OBJC_CLASSLIST_REFERENCES_$_:
```

```
        .quad   _OBJC_CLASS_$_NSString

        .section        __TEXT,__objc_methname,cstring_literals
L_OBJC_METH_VAR_NAME_:                  ## @OBJC_METH_VAR_NAME_
        .asciz  "alloc"

        .section        __DATA,__objc_selrefs,literal_pointers,no_dead_strip
        .p2align        3               ## @OBJC_SELECTOR_REFERENCES_
L_OBJC_SELECTOR_REFERENCES_:
        .quad   L_OBJC_METH_VAR_NAME_

        .section        __TEXT,__cstring,cstring_literals
L_.str:                                 ## @.str
        .asciz  "AloneMonkey"

        .section        __TEXT,__objc_methname,cstring_literals
L_OBJC_METH_VAR_NAME_.1:                ## @OBJC_METH_VAR_NAME_.1
        .asciz  "initWithUTF8String:"

        .section        __DATA,__objc_selrefs,literal_pointers,no_dead_strip
        .p2align        3               ## @OBJC_SELECTOR_REFERENCES_.2
L_OBJC_SELECTOR_REFERENCES_.2:
        .quad   L_OBJC_METH_VAR_NAME_.1

        .section        __TEXT,__cstring,cstring_literals
L_.str.3:                               ## @.str.3
        .asciz  "Hello, %@, Age: %d"

        .section        __DATA,__cfstring
        .p2align        3               ## @_unnamed_cfstring_
L__unnamed_cfstring_:
        .quad   ___CFConstantStringClassReference
        .long   1992                    ## 0x7c8
        .space  4
        .quad   L_.str.3
        .quad   18                      ## 0x12

        .section        __DATA,__objc_imageinfo,regular,no_dead_strip
L_OBJC_IMAGE_INFO:
        .long   0
        .long   64


.subsections_via_symbols
```

3.7 生成目标文件

```
clang -fmodules -c main.m -o main.o
```



3.8 生成可执行文件

```
clang main.o -o main
./main
```

```
2016-12-20 15:25:42.299 main[8941:327306] Hello, AloneMonkey, Age: 18
```

3.9 整体流程

四、可以用Clang做什么?

4.1 libclang进行语法分析

可以使用 libclang 里面提供的方法对源文件进行语法分析，分析它的语法树，遍历语法树上面的每一个节点。可以用于检查拼写错误，或者做字符串加密。

来看一段代码的使用：

```
void *hand =
dlopen("/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xc
toolchain/usr/lib/libclang.dylib",RTLD_LAZY);

//初始化函数指针
initlibfunclist(hand);

CXIndex cxindex = myclang_createIndex(1, 1);

const char *filename = "/path/to/filename";

int index = 0;

const char ** new_command = malloc(10240);

NSMutableString *mus = [NSMutableString
stringWithString:@"/Applications/Xcode.app/Contents/Developer/Toolchains/Xcod
eDefault.xctoolchain/usr/bin/clang -x objective-c -arch armv7 -isysroot
/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Develo
per/SDKs/iPhoneOS.sdk"];

NSArray *arr = [mus componentsSeparatedByString:@" "];

for (NSString *tmp in arr) {
    new_command[index++] = [tmp UTF8String];
}

nameArr = [[NSMutableArray alloc] initWithCapacity:10];

TU = myclang_parseTranslationUnit(cxindex, filename, new_command, index,
NULL, 0, myclang_defaultEditingTranslationUnitOptions());

CXCursor rootCursor = myclang_getTranslationUnitCursor(TU);

myclang_visitChildren(rootCursor, printVisitor, NULL);

myclang_disposeTranslationUnit(TU);
myclang_disposeIndex(cxindex);
free(new_command);

dlclose(hand);
```

然后我们就可以在`printVisitor`这个函数里面去遍历输入文件的语法树了。

```
2016-12-20 16:25:44.006588 ParseClangLib[9525:368452] showString
 int main(){
    @autoreleasepool {
        int numberOne = TEN;
```

```
        int numberTwo = 8;
        NSString* name = [[NSString alloc] initWithUTF8String:"AloneMonkey"];
        int age = numberOne + numberTwo;
        NSLog(@"Hello, %@, Age: %d", name, age);
    }
    return 0;
}
2016-12-20 16:25:44.007101 ParseClangLib[9525:368452] disname is main()
2016-12-20 16:25:44.007142 ParseClangLib[9525:368452] ccurkind is
=>FunctionDecl
2016-12-20 16:25:44.007180 ParseClangLib[9525:368452] 继续遍历孩子节点main()
2016-12-20 16:25:44.007236 ParseClangLib[9525:368452] showString
 {
    @autoreleasepool {
        int numberOne = TEN;
        int numberTwo = 8;
        NSString* name = [[NSString alloc] initWithUTF8String:"AloneMonkey"];
        int age = numberOne + numberTwo;
        NSLog(@"Hello, %@, Age: %d", name, age);
    }
    return 0;
}
2016-12-20 16:25:44.007253 ParseClangLib[9525:368452] disname is
2016-12-20 16:25:44.007263 ParseClangLib[9525:368452] ccurkind is
=>CompoundStmt
2016-12-20 16:25:44.007274 ParseClangLib[9525:368452] 继续遍历孩子节点
2016-12-20 16:25:44.007309 ParseClangLib[9525:368452] showString
 @autoreleasepool {
        int numberOne = TEN;
        int numberTwo = 8;
        NSString* name = [[NSString alloc] initWithUTF8String:"AloneMonkey"];
        int age = numberOne + numberTwo;
        NSLog(@"Hello, %@, Age: %d", name, age);
    }
2016-12-20 16:25:44.007424 ParseClangLib[9525:368452] disname is
2016-12-20 16:25:44.007442 ParseClangLib[9525:368452] ccurkind is
=>ObjCAutoreleasePoolStmt
2016-12-20 16:25:44.007455 ParseClangLib[9525:368452] 继续遍历孩子节点
2016-12-20 16:25:44.007488 ParseClangLib[9525:368452] showString
 {
        int numberOne = TEN;
        int numberTwo = 8;
        NSString* name = [[NSString alloc] initWithUTF8String:"AloneMonkey"];
        int age = numberOne + numberTwo;
        NSLog(@"Hello, %@, Age: %d", name, age);
    }
2016-12-20 16:25:44.007504 ParseClangLib[9525:368452] disname is
2016-12-20 16:25:44.007514 ParseClangLib[9525:368452] ccurkind is
=>CompoundStmt
```

```
2016-12-20 16:25:44.007525 ParseClangLib[9525:368452] 继续遍历孩子节点
2016-12-20 16:25:44.007553 ParseClangLib[9525:368452] showString
 int numberOne = TEN;
2016-12-20 16:25:44.007565 ParseClangLib[9525:368452] disname is
2016-12-20 16:25:44.007574 ParseClangLib[9525:368452] ccurkind is =>DeclStmt
2016-12-20 16:25:44.013133 ParseClangLib[9525:368452] 继续遍历孩子节点
2016-12-20 16:25:44.013206 ParseClangLib[9525:368452] showString
 int numberOne = TEN
.......
2016-12-20 16:25:44.015848 ParseClangLib[9525:368452] ccurkind is
=>ObjCStringLiteral
2016-12-20 16:25:44.015858 ParseClangLib[9525:368452] OC 字符串
2016-12-20 16:25:44.015876 ParseClangLib[9525:368452] showString
 @"Hello, %@, Age: %d"
2016-12-20 16:25:44.015932 ParseClangLib[9525:368452] showString
 name
2016-12-20 16:25:44.015973 ParseClangLib[9525:368452] disname is name
2016-12-20 16:25:44.015997 ParseClangLib[9525:368452] ccurkind is
=>UnexposedExpr
2016-12-20 16:25:44.016013 ParseClangLib[9525:368452] 继续遍历孩子节点name
2016-12-20 16:25:44.016039 ParseClangLib[9525:368452] showString
 name
2016-12-20 16:25:44.016051 ParseClangLib[9525:368452] disname is name
2016-12-20 16:25:44.016060 ParseClangLib[9525:368452] ccurkind is
=>DeclRefExpr
2016-12-20 16:25:44.016071 ParseClangLib[9525:368452] 继续遍历孩子节点
2016-12-20 16:25:44.016137 ParseClangLib[9525:368452] showString
 age
2016-12-20 16:25:44.016160 ParseClangLib[9525:368452] disname is age
2016-12-20 16:25:44.016170 ParseClangLib[9525:368452] ccurkind is
=>UnexposedExpr
2016-12-20 16:25:44.016183 ParseClangLib[9525:368452] 继续遍历孩子节点
2016-12-20 16:25:44.016213 ParseClangLib[9525:368452] showString
 age
2016-12-20 16:25:44.016256 ParseClangLib[9525:368452] disname is age
2016-12-20 16:25:44.016279 ParseClangLib[9525:368452] ccurkind is
=>DeclRefExpr
2016-12-20 16:25:44.016293 ParseClangLib[9525:368452] 继续遍历孩子节点age
2016-12-20 16:25:44.016318 ParseClangLib[9525:368452] showString
 return 0
2016-12-20 16:25:44.016330 ParseClangLib[9525:368452] disname is
2016-12-20 16:25:44.016339 ParseClangLib[9525:368452] ccurkind is
=>ReturnStmt
2016-12-20 16:25:44.016350 ParseClangLib[9525:368452] 继续遍历孩子节点
2016-12-20 16:25:44.016369 ParseClangLib[9525:368452] showString
 0
2016-12-20 16:25:44.016408 ParseClangLib[9525:368452] disname is
2016-12-20 16:25:44.016445 ParseClangLib[9525:368452] ccurkind is
=>IntegerLiteral
```

```
2016-12-20 16:25:44.016461 ParseClangLib[9525:368452] 继续遍历孩子节点
```

我们也通过通过 python 去调用 clang：

```
pip install clang
```

```
#!/usr/bin/python
# vim: set fileencoding=utf-8

import clang.cindex
import asciitree
import sys

def node_children(node):
        return (c for c in node.get_children() if c.location.file ==
sys.argv[1])

def print_node(node):
    text = node.spelling or node.displayname
    kind = str(node.kind)[str(node.kind).index('.')+1:]
    return '{} {}'.format(kind, text)

if len(sys.argv) != 2:
    print("Usage: dump_ast.py [header file name]")
    sys.exit()

clang.cindex.Config.set_library_file('/Applications/Xcode.app/Contents/Develo
per/Toolchains/XcodeDefault.xctoolchain/usr/lib/libclang.dylib')
index = clang.cindex.Index.create()
translation_unit = index.parse(sys.argv[1], ['-x', 'objective-c'])

print asciitree.draw_tree(translation_unit.cursor,
                          lambda n: list(n.get_children()),
                          lambda n: "%s (%s)" % (n.spelling or n.displayname,
str(n.kind).split(".")[1]))
```

```
.......
  +--main (FUNCTION_DECL)
     +-- (COMPOUND_STMT)
        +-- (OBJC_AUTORELEASE_POOL_STMT)
        |  +-- (COMPOUND_STMT)
        |     +-- (DECL_STMT)
        |     |  +--numberOne (VAR_DECL)
        |     |     +-- (INTEGER_LITERAL)
        |     +-- (DECL_STMT)
        |     |  +--numberTwo (VAR_DECL)
        |     |     +-- (INTEGER_LITERAL)
        |     +-- (DECL_STMT)
        |     |  +--name (VAR_DECL)
        |     |     +--NSString (OBJC_CLASS_REF)
        |     |     +--initWithUTF8String: (OBJC_MESSAGE_EXPR)
        |     |        +--alloc (OBJC_MESSAGE_EXPR)
        |     |        |  +--NSString (OBJC_CLASS_REF)
        |     |        +-- (UNEXPOSED_EXPR)
        |     |           +-- (UNEXPOSED_EXPR)
        |     |              +--"AloneMonkey" (STRING_LITERAL)
        |     +-- (DECL_STMT)
        |     |  +--age (VAR_DECL)
        |     |     +-- (BINARY_OPERATOR)
        |     |        +--numberOne (UNEXPOSED_EXPR)
        |     |        |  +--numberOne (DECL_REF_EXPR)
        |     |        +--numberTwo (UNEXPOSED_EXPR)
        |     |           +--numberTwo (DECL_REF_EXPR)
        |     +--NSLog (CALL_EXPR)
        |        +--NSLog (UNEXPOSED_EXPR)
        |        |  +--NSLog (DECL_REF_EXPR)
        |        +-- (UNEXPOSED_EXPR)
        |        |  +--"Hello, %@, Age: %d" (OBJC_STRING_LITERAL)
        |        |     +--"Hello, %@, Age: %d" (STRING_LITERAL)
        |        +--name (UNEXPOSED_EXPR)
        |        |  +--name (DECL_REF_EXPR)
        |        +--age (UNEXPOSED_EXPR)
        |           +--age (DECL_REF_EXPR)
        +-- (RETURN_STMT)
           +-- (INTEGER_LITERAL)
```

那么基于语法树的分析，我们可以针对字符串做加密：

从左上角的明文字符串，处理成右下角的介个样子~

4.2 LibTooling

对语法树有完全的控制权，可以作为一个单独的命令使用，如：`clang-format`

```
clang-format main.m
```

我们也可以自己写一个这样的工具去遍历、访问、甚至修改语法树。 目录:`llvm/tools/clang/tools`

```
#include "clang/Driver/Options.h"
#include "clang/AST/AST.h"
#include "clang/AST/ASTContext.h"
#include "clang/AST/ASTConsumer.h"
#include "clang/AST/RecursiveASTVisitor.h"
#include "clang/Frontend/ASTConsumers.h"
#include "clang/Frontend/FrontendActions.h"
#include "clang/Frontend/CompilerInstance.h"
#include "clang/Tooling/CommonOptionsParser.h"
#include "clang/Tooling/Tooling.h"
#include "clang/Rewrite/Core/Rewriter.h"

using namespace std;
using namespace clang;
using namespace clang::driver;
using namespace clang::tooling;
using namespace llvm;

Rewriter rewriter;
int numFunctions = 0;

static llvm::cl::OptionCategory StatSampleCategory("Stat Sample");
```

```
class ExampleVisitor : public RecursiveASTVisitor<ExampleVisitor> {
private:
    ASTContext *astContext; // used for getting additional AST info

public:
    explicit ExampleVisitor(CompilerInstance *CI)
      : astContext(&(CI->getASTContext())) // initialize private members
    {
        rewriter.setSourceMgr(astContext->getSourceManager(), astContext-
>getLangOpts());
    }

    virtual bool VisitFunctionDecl(FunctionDecl *func) {
        numFunctions++;
        string funcName = func->getNameInfo().getName().getAsString();
        if (funcName == "do_math") {
            rewriter.ReplaceText(func->getLocation(), funcName.length(),
"add5");
            errs() << "** Rewrote function def: " << funcName << "\n";
        }
        return true;
    }

    virtual bool VisitStmt(Stmt *st) {
        if (ReturnStmt *ret = dyn_cast<ReturnStmt>(st)) {
            rewriter.ReplaceText(ret->getRetValue()->getLocStart(), 6,
"val");
            errs() << "** Rewrote ReturnStmt\n";
        }
        if (CallExpr *call = dyn_cast<CallExpr>(st)) {
            rewriter.ReplaceText(call->getLocStart(), 7, "add5");
            errs() << "** Rewrote function call\n";
        }
        return true;
    }
};



class ExampleASTConsumer : public ASTConsumer {
private:
    ExampleVisitor *visitor; // doesn't have to be private

public:
    // override the constructor in order to pass CI
    explicit ExampleASTConsumer(CompilerInstance *CI)
        : visitor(new ExampleVisitor(CI)) // initialize the visitor
    { }
```

```
    // override this to call our ExampleVisitor on the entire source file
    virtual void HandleTranslationUnit(ASTContext &Context) {
        visitor->TraverseDecl(Context.getTranslationUnitDecl());
    }
};



class ExampleFrontendAction : public ASTFrontendAction {
public:
    virtual std::unique_ptr<ASTConsumer> CreateASTConsumer(CompilerInstance
&CI, StringRef file) {
         return llvm::make_unique<ExampleASTConsumer>(&CI); // pass CI
pointer to ASTConsumer
    }
};



int main(int argc, const char **argv) {
    // parse the command-line args passed to your code
    CommonOptionsParser op(argc, argv, StatSampleCategory);
    // create a new Clang Tool instance (a LibTooling environment)
    ClangTool Tool(op.getCompilations(), op.getSourcePathList());

    // run the Clang Tool, creating a new FrontendAction (explained below)
    int result = Tool.run(newFrontendActionFactory<ExampleFrontendAction>
().get());

    errs() << "\nFound " << numFunctions << " functions.\n\n";
    // print out the rewritten source code ("rewriter" is a global var.)

rewriter.getEditBuffer(rewriter.getSourceMgr().getMainFileID()).write(errs())
;
    return result;
}
```

上面的代码通过遍历语法树，去修改里面的方法名和返回变量名：

```
before:
void do_math(int *x) {
    *x += 5;
}

int main(void) {
    int result = -1, val = 4;
    do_math(&val);
    return result;
}

after:
** Rewrote function def: do_math
** Rewrote function call
** Rewrote ReturnStmt

Found 2 functions.

void add5(int *x) {
    *x += 5;
}

int main(void) {
    int result = -1, val = 4;
    add5(&val);
    return val;
}
```

那么，我们看到`LibTooling`对代码的语法树有完全的控制，那么我们可以基于它去检查命名的规范，甚至做一个代码的转换，比如实现OC转Swift。

4.3 ClangPlugin

对语法树有完全的控制权，作为插件注入到编译流程中，可以影响build和决定编译过程。目录:`llvm/tools/clang/examples`

```
#include "clang/Driver/Options.h"
#include "clang/AST/AST.h"
#include "clang/AST/ASTContext.h"
#include "clang/AST/ASTConsumer.h"
#include "clang/AST/RecursiveASTVisitor.h"
#include "clang/Frontend/ASTConsumers.h"
#include "clang/Frontend/FrontendActions.h"
#include "clang/Frontend/CompilerInstance.h"
#include "clang/Frontend/FrontendPluginRegistry.h"
#include "clang/Rewrite/Core/Rewriter.h"

using namespace std;
```

```
using namespace clang;
using namespace llvm;

Rewriter rewriter;
int numFunctions = 0;


class ExampleVisitor : public RecursiveASTVisitor<ExampleVisitor> {
private:
    ASTContext *astContext; // used for getting additional AST info

public:
    explicit ExampleVisitor(CompilerInstance *CI)
      : astContext(&(CI->getASTContext())) // initialize private members
    {
        rewriter.setSourceMgr(astContext->getSourceManager(), astContext-
>getLangOpts());
    }

    virtual bool VisitFunctionDecl(FunctionDecl *func) {
        numFunctions++;
        string funcName = func->getNameInfo().getName().getAsString();
        if (funcName == "do_math") {
            rewriter.ReplaceText(func->getLocation(), funcName.length(),
"add5");
            errs() << "** Rewrote function def: " << funcName << "\n";
        }
        return true;
    }

    virtual bool VisitStmt(Stmt *st) {
        if (ReturnStmt *ret = dyn_cast<ReturnStmt>(st)) {
            rewriter.ReplaceText(ret->getRetValue()->getLocStart(), 6,
"val");
            errs() << "** Rewrote ReturnStmt\n";
        }
        if (CallExpr *call = dyn_cast<CallExpr>(st)) {
            rewriter.ReplaceText(call->getLocStart(), 7, "add5");
            errs() << "** Rewrote function call\n";
        }
        return true;
    }
};



class ExampleASTConsumer : public ASTConsumer {
private:
    ExampleVisitor *visitor; // doesn't have to be private
```

```
public:
    // override the constructor in order to pass CI
    explicit ExampleASTConsumer(CompilerInstance *CI):
        visitor(new ExampleVisitor(CI)) { } // initialize the visitor

    // override this to call our ExampleVisitor on the entire source file
    virtual void HandleTranslationUnit(ASTContext &Context) {
        /* we can use ASTContext to get the TranslationUnitDecl, which is
             a single Decl that collectively represents the entire source
file */
        visitor->TraverseDecl(Context.getTranslationUnitDecl());
    }
};

class PluginExampleAction : public PluginASTAction {
protected:
    // this gets called by Clang when it invokes our Plugin
    // Note that unique pointer is used here.
    std::unique_ptr<ASTConsumer> CreateASTConsumer(CompilerInstance &CI,
StringRef file) {
        return llvm::make_unique<ExampleASTConsumer>(&CI);
    }

    // implement this function if you want to parse custom cmd-line args
    bool ParseArgs(const CompilerInstance &CI, const vector<string> &args) {
        return true;
    }
};


static FrontendPluginRegistry::Add<PluginExampleAction> X("-example-plugin",
"simple Plugin example");
```

```
clang -Xclang -load -Xclang ../build/lib/PluginExample.dylib -Xclang -plugin
-Xclang -example-plugin -c testPlugin.c

** Rewrote function def: do_math
** Rewrote function call
** Rewrote ReturnStmt
```

我们可以基于 ClangPlugin 做些什么事情呢？我们可以用来定义一些编码规范，比如代码风格检查，命名检查等等。下面是我写的判断类名前两个字母是不是大写的例子，如果不是报错。(当然这只是一个例子而已。。。)

```
In file included from /Users/monkey/Desktop/test/test/ViewController.m:9:
/Users/monkey/Desktop/test/test/ViewController.h:11:12: error: 缺少项目类名前缀
@interface ViewController : UIViewController
           ^
1 error generated.
Command /Users/monkey/Documents/llvm/build/bin/clang failed with exit code 1
```

Command /Users/monkey/Documents/llvm/build/bin/clang failed with exit code 1

五、动手写 Pass

5.1 一个简单的 Pass

前面我们说到，Pass 就是 LLVM 系统转化和优化的工作的一个节点，当然我们也可以写一个这样的节点去做一些自己的优化工作或者其它的操作。下面我们来看一下一个简单 Pass 的编写流程：

1.创建头文件

```
cd llvm/include/llvm/Transforms/
mkdir Obfuscation
cd Obfuscation
touch SimplePass.h
```

写入内容：

```
#include "llvm/IR/Function.h"
#include "llvm/Pass.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/IR/Intrinsics.h"
#include "llvm/IR/Instructions.h"
#include "llvm/IR/LegacyPassManager.h"
#include "llvm/Transforms/IPO/PassManagerBuilder.h"

// Namespace
using namespace std;

namespace llvm {
        Pass *createSimplePass(bool flag);
}
```

2.创建源文件

```
cd llvm/lib/Transforms/
mkdir Obfuscation
cd Obfuscation

touch CMakeLists.txt
touch LLVMBuild.txt
touch SimplePass.cpp
```

CMakeLists.txt:

```
add_llvm_loadable_module(LLVMObfuscation
  SimplePass.cpp

  )

  add_dependencies(LLVMObfuscation intrinsics_gen)
```

LLVMBuild.txt:

```
[component_0]
type = Library
name = Obfuscation
parent = Transforms
library_name = Obfuscation
```

SimplePass.cpp:

```
#include "llvm/Transforms/Obfuscation/SimplePass.h"

using namespace llvm;

namespace {
    struct SimplePass : public FunctionPass {
        static char ID; // Pass identification, replacement for typeid
        bool flag;

        SimplePass() : FunctionPass(ID) {}
        SimplePass(bool flag) : FunctionPass(ID) {
                this->flag = flag;
        }

        bool runOnFunction(Function &F) override {
                if(this->flag){
                Function *tmp = &F;
                // 遍历函数中的所有基本块
                for (Function::iterator bb = tmp->begin(); bb != tmp->end();
++bb) {
                    // 遍历基本块中的每条指令
                    for (BasicBlock::iterator inst = bb->begin(); inst != bb-
>end(); ++inst) {
                        // 是否是add指令
                        if (inst->isBinaryOp()) {
                            if (inst->getOpcode() == Instruction::Add) {
                                ob_add(cast<BinaryOperator>(inst));
                            }
                        }
```

```
                    }
                }
            }
            return false;
        }

        // a+b === a-(-b)
        void ob_add(BinaryOperator *bo) {
            BinaryOperator *op = NULL;

            if (bo->getOpcode() == Instruction::Add) {
                // 生成 (-b)
                op = BinaryOperator::CreateNeg(bo->getOperand(1), "", bo);
                // 生成 a-(-b)
                op = BinaryOperator::Create(Instruction::Sub, bo-
>getOperand(0), op, "", bo);

                op->setHasNoSignedWrap(bo->hasNoSignedWrap());
                op->setHasNoUnsignedWrap(bo->hasNoUnsignedWrap());
            }

            // 替换所有出现该指令的地方
            bo->replaceAllUsesWith(op);
        }
    };
}

char SimplePass::ID = 0;

// 注册pass 命令行选项显示为simplepass
static RegisterPass<SimplePass> X("simplepass", "this is a Simple Pass");
Pass *llvm::createSimplePass() { return new SimplePass(); }
```

修改`.../Transforms/LLVMBuild.txt`,加上刚刚写的模块`Obfuscation`

```
subdirectories = Coroutines IPO InstCombine Instrumentation Scalar Utils
Vectorize ObjCARC Obfuscation
```

修改`.../Transforms/CMakeLists.txt`，加上刚刚写的模块`Obfuscation`

```
add_subdirectory(Obfuscation)
```

编译生成：`LLVMSimplePass.dylib`

因为 Pass 是作用于中间代码，所以我们首先要生成一份中间代码：

```
clang -emit-llvm -c test.c -o test.bc
```

然后加载 Pass 优化：

```
../build/bin/opt -load ../build/lib/LLVMSimplePass.dylib -simplepass <
test.bc > after_test.bc
```

对比中间代码：

```
llvm-dis test.bc -o test.ll
llvm-dis after_test.bc -o after_test.ll
```

```
test.ll
......
entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  %c = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  store i32 3, i32* %a, align 4
  store i32 4, i32* %b, align 4
  %0 = load i32, i32* %a, align 4
  %1 = load i32, i32* %b, align 4
  %add = add nsw i32 %0, %1
  store i32 %add, i32* %c, align 4
  %2 = load i32, i32* %c, align 4
  %call = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8],
[4 x i8]* @.str, i32 0, i32 0), i32 %2)
  ret i32 0
}
......
```

```
after_test.ll
......
entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  %c = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  store i32 3, i32* %a, align 4
  store i32 4, i32* %b, align 4
  %0 = load i32, i32* %a, align 4
  %1 = load i32, i32* %b, align 4
  %2 = sub i32 0, %1
  %3 = sub nsw i32 %0, %2
  %add = add nsw i32 %0, %1
  store i32 %3, i32* %c, align 4
  %4 = load i32, i32* %c, align 4
  %call = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8],
[4 x i8]* @.str, i32 0, i32 0), i32 %4)
  ret i32 0
}
......
```

这里写的Pass只是把a+b简单的替换成了a-(-b),只是一个演示，怎么去写自己的Pass，并且作用于代码。

5.2 将 Pass 加入 PassManager 管理

上面我们是单独去加载Pass动态库，这里我们将 Pass 加入 PassManager，这样我们就可以直接通过 clang 的参数去加载我们的 Pass 了。

首先在`llvm/lib/Transforms/IPO/PassManagerBuilder.cpp`添加头文件。

```
#include "llvm/Transforms/Obfuscation/SimplePass.h"
```

然后添加如下语句：

```
static cl::opt<bool> SimplePass("simplepass", cl::init(false),
                           cl::desc("Enable simple pass"));
```

然后在`populateModulePassManager`这个函数中添加如下代码：

```
MPM.add(createSimplePass(SimplePass));
```

最后在 IPO 这个目录的`LLVMBuild.txt`中添加库的支持，否则在编译的时候会提示链接错误。具体内容如下：

```
required_libraries = Analysis Core InstCombine IRReader Linker Object
ProfileData Scalar Support TransformUtils Vectorize Obfuscation
```

修改 Pass 的 CMakeLists.txt 为静态库形式：

```
add_llvm_library(LLVMObfuscation
  SimplePass.cpp
  )

add_dependencies(LLVMObfuscation intrinsics_gen)
```

最后再编译一次。

那么我们可以这么去调用：

```
../build/bin/clang -mllvm -simplepass test.c -o after_test
```

基于 Pass，我们可以做什么？ 我们可以编写自己的 Pass 去混淆代码，以增加他人反编译的难度。

我们可以把代码左上角的样子，变成右下角的样子，甚至更加复杂~

六、总结

上面说了那么说，来总结一下：

1.LLVM 编译一个源文件的过程：

预处理 -> 词法分析 -> Token -> 语法分析 -> AST -> 代码生成 -> LLVM IR -> 优化 -> 生成汇编代码 -> Link -> 目标文件

2.基于 LLVM，我们可以做什么？

1. 做语法树分析，实现语言转换 OC 转 Swift、JS or 其它语言，字符串加密。
2. 编写 ClangPlugin，命名规范，代码规范，扩展功能。

3. 编写 Pass，代码混淆优化。

这篇只是一个简单的入门介绍，有兴趣大家可以自行深入了解。

TCP / IP 漫游

作者：@mrriddler

TCP/IP是互联网的基础协议栈，它包括大大小小几十个协议。本篇文章主要涉及到就是HTTP、TCP、IP协议。我们经常学的网络模型是七层或者五层，实际上一般认为一共只有四层就可以了。

```
    Application layer           ->HTTP
--------------------------
     Transport layer            ->TCP
--------------------------
     Internet layer             ->IP
--------------------------
      Link layer                ->Ethernet
```

应用层、传输层、网际层、连接层，这一眼看上去，很直白嘛，就是分层抽象，层与层互相隔离，这确实没问题。不过，我们可以换个角度来整体理解一下这四层。网络所做的就是传输数据嘛，那这四层就是数据在不同阶段的不同形式。应用层是数据的最终形式，传输层是数据的字节(文本)形式，网际层是数据的二进制形式，连接层是数据的信号形式(这主要指的是连接层的最底层部分，电缆传输的就是电信号嘛)。这样换一个角度去看世界也是蛮有意思的，并且这么看四层就更像是一个整体了。

数据在发送通过这四层的时候，每经过一层会将数据作为Body，并加上这一层的控制信息作为header。而到达目的地后，再这通过四层时，每层会将相应的header剥离，最后给接收方一个与发送方发送的一模一样的数据。

HTTP

HTTP已有很多文章介绍了，这篇文章就不多说了。仅聊一聊对HTTP协议的理解，HTTP协议内部实际上是一个，协议字符串解析器。HTTP协议以字符串形式将数据解析、逆解析成请求行/响应行、请求头、请求体构成的报文。为什么说是字符串解析器呢？实际上，请求行、响应行、header甚至是分隔标识(CRLF)都是字符串。而body就是数据，并且HTTP将数据的最终形式交由协议使用者来决定(text、xml、json等)。

TCP

你需要知道的关于TCP的第一件事就是TCP是个包含非常多知识点的协议(⊙﹏⊙)，并且它还是一个实践性很强的协议，TCP中很多为什么这么做，都是应用于网络中实践出真知得出的“真理”。TCP是个可靠传输协议，在如此动荡(不安)的网络环境里，想要确保这一点可不容易，TCP独有很多机制来做到这一点。先来聊一下可靠传输协议。

可靠传输协议要求，双方的报文段(segment)必须都能无损的到达，并且发送方和接收方最后的报文段顺序需要一致。这代表，要检查报文是否完好无损，完好无损才算个合格的报文。如果报文受损或者发送方报文段根本没有到达接收方，发送方需要重传。如何知道报文根本没有到达接收方？发送方启动一个计时器就可以，计时器超时就认为报文没有到达接收方。

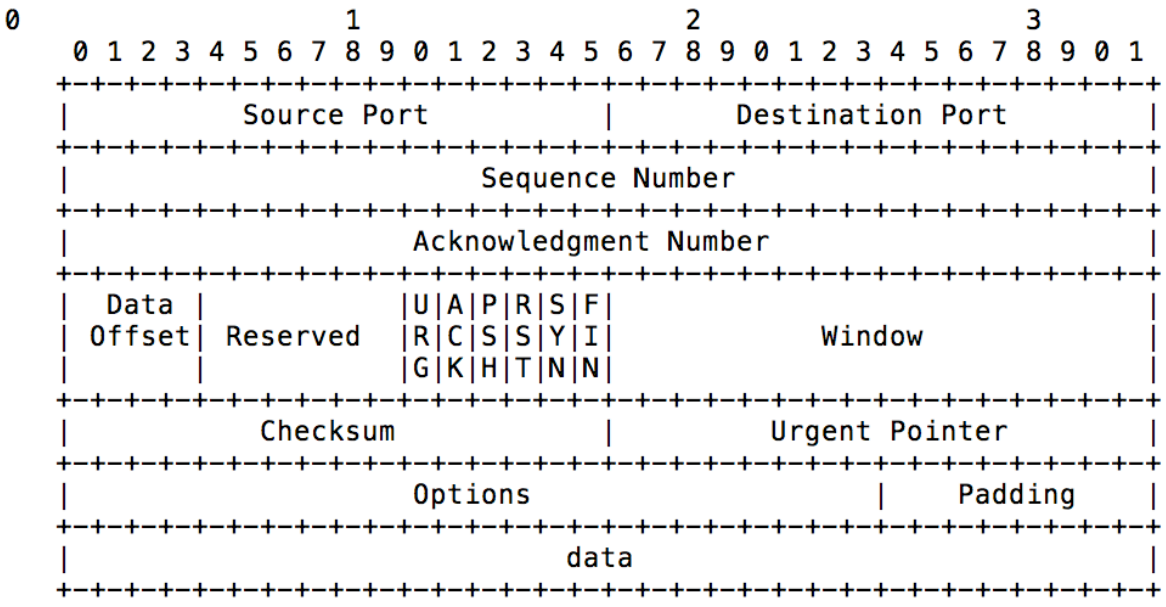
并且既然重传就需要接收方要对发送方确认，确认后发送方才放心。最简单的确认做法就是，一方发送报文段，另一方再发送一个单独做确认的报文段，但这样太低效了。不如，在发送自己的报文段的时候，顺便确认下所接受的报文段。为了最后要确认顺序，报文段还需要有序号来标识顺序。而上面说的这个几点在TCP协议中都有体现，更直接的在TCP header中就有体现。

TCP还是一个面向连接的协议，建立连接是通过大名鼎鼎的三次握手，断开连接通过四次挥手。这个连接肯定不是建立物理上的连接，而是逻辑上建立了连接。连接也就指的是，发送方和接收方都要初始化一些状态会被用来跟踪(track)发送过程，也就说TCP是个有状态的协议，三次握手和四次挥手后面会聊。

TCP提供端到端的服务，它会具体到应用程序的port。不同于IP提供点到点的服务，如果将发送数据比作送快递的话，IP提供的服务就是，快递员准确的送到你家，TCP提供的服务就是，不仅送到你家，还将快递准确的交到接收人的手上。

TCP除了要做到可靠传输，并且还要照顾接收方和网络总体状况，这其中还有流量控制和拥塞控制。上面说的这些机制后文会一个一个聊，现在先来看一下TCP的header。

TCP Header



我们来逐一的过一下这些字段，Source Port和Destination Port代表目的和源端口。

Sequence Number代表报文段的序号来表示顺序。Acknowledgment Number代表发送方作为接收方已接收的报文段，并且期望收到下个报文段的开始序列号。TCP实际上将传送数据看做成数据的字节流，而不是一个个单独的报文段。这点从Sequence Number就可以看出，Sequence Number以传送的字节数作为单位，而不以报文段的数量作为单位。

Data Offset作为对齐的空位，Reserved作为保留位。下面是重点Flag位。

Flag包括Urgent、Ack、Push、Reset、Syn、Fin这6位。Urgent作为报文段的紧急数据标识，但具体如何处理交给接收方决定。Reset作为报文段的连接异常结束或端口号错误的标识。而Ack确认、Syn同步、Fin结束在三次握手和四次挥手中作为关键标识位。Push代表TCP不再等待是否还有其他报文段到达，马上交给上层应用层。

Window位作为流量控制的基础，后面会更具体的聊。

Check Sum作为校验位，校验报文段是否在传输过程中受损。Urgent Pointer在Urgent位为1时，才会出现，指向紧急数据的最后一个字节。

Options常见的标识有nop、TS val(time stamp)、ecr(echo reply)、mss。nop标识就和气泡指令一个意思，就是占位的。而TS val和ecr分别代表发送方的时间戳和接收方的时间戳，基于这两个时间戳来计算出RTT往返时间 (round-trip time) ，当然还要加权平均，具体计算就不多说了，RTT会被用来衡量重传计时器的超时时长。mss(Maximum Segment Size)指的是，连接层每次传输的数据有个最大限制MTU(Maximum Transmission Unit)，一般是1500比特，超过这个量要分成多个报文段，mss则是这个最大限制减去TCP的header，光是要传输的数据的大小，一般为1460比特。

三次握手

三次握手，只能由客户端向服务端发起。第一次客户端发送SYN为1，序列号seq为某序列号，表示客户端想要建立连接。第二次服务端返回ACK、SYN都为1，序列号seq为某序列号，确认号为接收的序列号加1，表示确认服务端也想要建立连接。第三次客户端发送ACK为1，确认号为所接收序列号加1，再次确认，然后连接建立。

为何要进行三次握手？为了防止失效的报文段又到达了服务端产生错误。假设，客户端发送的第一个报文段延时到达了服务端，这个报文应被认为失效。但服务端误认为客户端想要建立一个新连接，就发出了确认，若没有第三次确认再建立起连接。服务端就错误地建立起一个连接。

如果三次握手，第三次失败了会怎么办？此时，服务端并不会马上放弃，服务端还会尝试重新发送确认，默认重试5次，间隔从1秒开始，后来每次是前一次的2倍。5次重试后，未果则放弃连接。

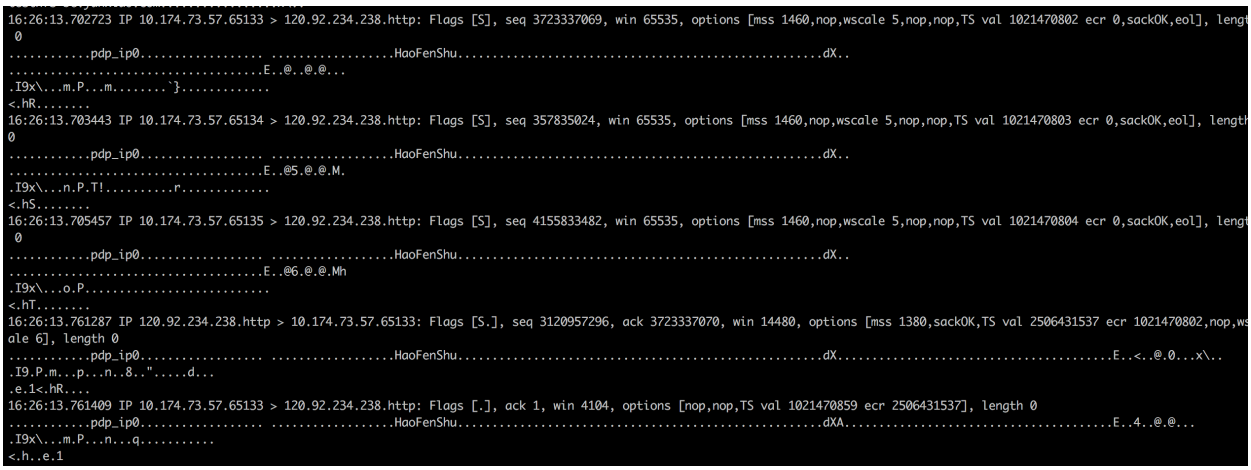
四次挥手

四次挥手，客户端和服务端都可以发起。第一次发送方发送FIN为1、ACK为1，序列号为某序列号，表示发送方想结束连接。第二次接收方发送ACK为1，确认号为接收序列号加1，表示我还没有准备好结束连接。第三次接收方发送FIN为1、ACK为1，序列号为某序列号，表示我已经准备好结束连接了。第四次发送方，发送ACK为1，确认号为所接收序列号加1，表示确认，结束连接。

半关闭

在四次挥手的基础上，发送方可以在接收第二次接收方发送ACK后，可以形成发送方不再发送报文段，但仍然接收接收方发送的报文段的这种现象。这就形成了半关闭。

一开始，我对TCP/IP一点都不感冒，这确实是自己懒。光看书和概念，味如嚼蜡。还是动手实验，才会有更深的体会，推荐tcpdump工具。tcpdump如何使用点我。下面是我用tcpdump截的几个报文段。

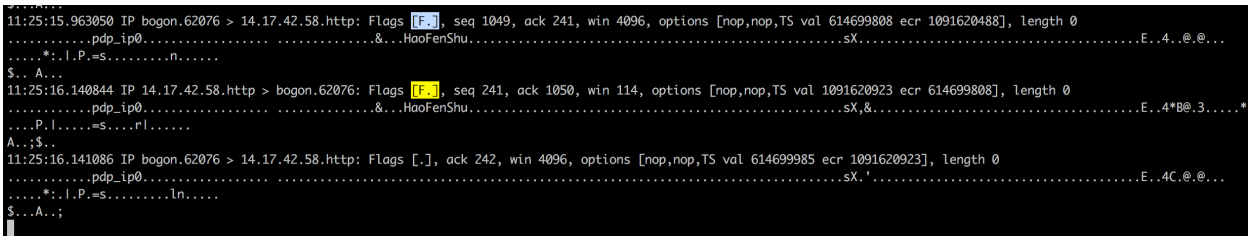


tcpdump第一行截到的信息翻译：

```
16:26:13.702723 IP 10.174.73.57.65133 > 120.92.234.238.http: Flags[S], seq
37233370769, win 65535,
       ↓         ↓          ↓                      ↓        ↓       ↓       ↓
              ↓
    发送时间    IP协议     源IP.port      到    目的IP.port http协议 Flags位 seq确认
号          win大小

options [mss 1460,nop,wscale 5,nop,nop,TS val 1021470802 ecr 0, sackOK,
eol], length 0
                                    ↓
    ↓
                                options位                                  报文
段不包含header的长度
```

实际上，图中的五个报文段，其中的三个报文段正是TCP三次握手的过程。读者可以尝试找一下哪三个是。

这里还有一张TCP四次挥手的截图：



TCP的确认方式

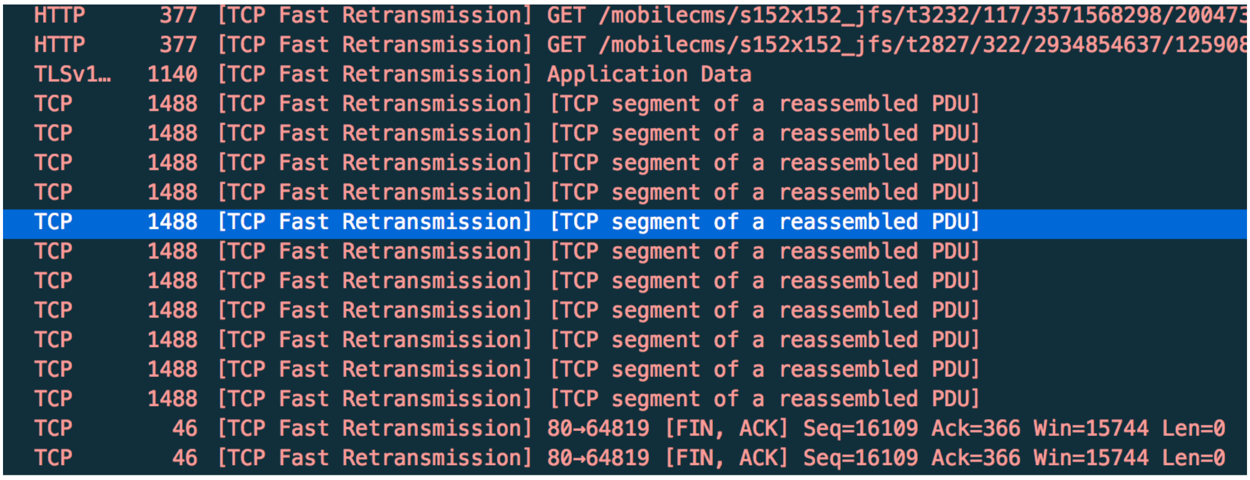
实际上，TCP实现的可靠传输协议还有更多的细节。上文说道可靠协议在发送自己的报文段的时候，可以顺便确认下所接受的报文段。TCP就是这样去确认，以至于TCP确认会延迟，去等待是否有报文段发送，让报文段捎上确认。延迟一般为200毫秒。

比如说TCP使用的确认方式。TCP使用了累计确认。接收方为了交付给应用层正确的顺序，只有顺序正确的报文段会被确认然后交付给上层。发送方如果收到了接收方的某个确认号，即使这个确认号以前的报文段没有收到确认号也会被认为正确接收。

TCP还会使用选择确认(selective acknowledgement)。假设，发送方发送了多个报文段，初始的报文段出现了问题，没有抵达接收方。发送方会仅仅认为这个初始的报文段失效了，而后发送的几个报文段准确到达了接收方。也就是说，后发送的报文段虽然没有接收到直接的确认，而发送方选择性的确认了他们。接收方会将后发送的失序报文段先放入缓冲(buffer)中。这就是TCP实践出真知的最好例子。网络环境动荡一般会影响单个报文段，而不会影响一大片的报文段。

基于上面所说的累计确认和选择确认，若是报文段失效，发送可能会收到多次对于同一个报文段的冗余确认，若是收到了三次冗余确认，就认为这个报文段失效了，TCP不会等待计时器超时再重传，TCP会直接启动快速重传(fast retransmit)，直接重传。这一点，实际上就是以时间和数据量两个指标作为衡量重传的条件。

同样，还是用工具实践一下有意思一点，推荐使用WireShark，WireShark的Filter非常强大，在进行网络诊断的时候非常有用。WireShark会根据TCP header中的Sequence Number，分析出冗余ACK、快速重传等现象，具体点我。WireShark在Filter中输入tcp.analysis.fast_retransmission，就可以找出快速重传的报文段。



流量控制

流量控制实际上是发送方发送和接收方处理速度匹配的过程。

TCP连接发送的报文段，都会放入上文所说的缓冲中等待应用程序取出。如果一端持续发送报文段，另一端一直没有及时处理完并接着取出报文段，就会造成缓冲溢出。这时，就需要进行双方的速度匹配，进行流量控制。接收方会将自己的缓冲剩余空间rwnd告诉发送方，发送方为了控制速度，只能再发送所得到的剩余空间rwnd容量的报文段。由于上文所说TCP采取的确认方式，发送方得到的这个rwnd容量不会限制已发送而未得到确认的报文段，这些报文段很可能已经在接收方的缓冲中了，只限制将要发送的报文段。

将传送数据看做数据流后，上面的这个过程就像在以序号作为基准在数据流上移动窗口一样，所以得名流量窗口。而剩余空间rwnd，也就是TCP header中的window位。

这里还有个问题，如果一方接收到了零剩余空间信息，这方就再也不发送报文了吗？不是，TCP为应对这个情况会有个计时器(persist timer)，出现这种情况就会让计时器记时，当计时器触发，这方会发送个剩余空间探测报文段(window probe)，以检测是否可以重新发送报文段。如果一直没有剩余空间，计时器永远不会终止，仍会做重新记时、超时的循环。

在WireShark中可以通过tcp.analysis.zero_window_probe、tcp.analysis.window_full找出剩余空间探测报文段和通知发送方接收方空间已满的报文段。

拥塞控制(Congestion-Control)

拥塞控制很好理解，TCP如果不照顾网络总体状况，一股脑的传送数据的话，在极差的网络环境下只会恶性循环，可以直接搞瘫网络。而传送数据太小心的话又不能充分利用带宽资源。所以，拥塞控制就是一个动态平衡的策略。拥塞控制实际上就是一个由三个状态组成的有限状态机(FSM)，这三个状态是慢启动(slow start)、拥塞避免(congestion avoid)、快速恢复(fast recovery)。

首先，聊这三个状态之前先聊一下TCP对拥塞的认识。

- TCP如何根据自己的有限信息判定网络拥塞？只要出现超时重传和3次冗余ACK引起的快速重传就认为网络拥塞了。
- 网络拥塞也有程度的区分，TCP如何判定拥塞的程度？超时重传被认为拥塞程度强，快速重传被

认为拥塞程度弱。

- 相对于拥塞，TCP如何判断网络不拥塞?只要收到了非冗余的ACK，TCP就认为一切顺利，没有拥塞。
- TCP如何判断快要拥塞了？TCP会在每次发生拥塞后，记录下导致发生拥塞的报文段的数量的一半，最小不能小于2单位(mss)报文段，这个值被用来衡量下次是否快要拥塞。

```
|   不拥塞    |       快要拥塞          |拥塞程度弱|拥塞程度强|
      ↑                   ↑                 ↑          ↑
|正常收到ACK|到了上次拥塞一半的量|  快速重传  |  超时重传  |
```

TCP有了这四个认识，就可以愉快地照顾网络总体情况了。TCP以cwnd标识能够发送的报文段的量，不用多说，拥塞控制整个过程也像是在数据流上移动窗口，所以也叫拥塞窗口。

首先，慢启动。

在慢启动阶段，TCP以cwnd为1作为初始量，然后每确认一个报文段，都会为cwnd加1。这样，如果TCP一直保持最大限度的发送报文段，每过一个RTT，TCP发送的报文段量就会翻倍。所以，在慢启动阶段，TCP是指数级增长。慢启动的语义是，现在对网络状态不是很清楚，先假设状态不好，一上来少发送点，然后多发点试探网络状况。

其次，拥塞避免。

当cwnd增长到快要拥塞的时候会状态迁移到拥塞避免。上文说到为标志快要拥塞会维护一个值ssthresh(slow start threshold)，当cwnd大于等于ssthresh，慢启动迁移到拥塞避免状态。进入拥塞状态后，每确认一个报文段，都会为cwnd加1/cwnd。这样，如果TCP一直保持最大限度的发送报文段，每过一个RTT，TCP发送的报文段量会加1。所以，在拥塞避免状态，TCP是线性增长。拥塞避免的语义是，快要拥塞，小心一点。

然后，快速恢复。

如果出现了快速重传怎么办？不管是慢启动还是拥塞避免，都迁移到快速恢复。既然拥塞程度弱，那就适当的降低cwnd，将cwnd除2，并且维护ssthresh记录拥塞的量，将cwnd的值赋给ssthresh。发生快速恢复就说明出现了3次冗余ACK，TCP基于选择确认，认为引起3次冗余ACK的报文段顺利到达，将cwnd加上3个单位(mss)的量。如果再收到这个报文段的冗余ACK，为cwnd加1。如果收到了非这个报文段的冗余ACK，表明这个报文段正确到达了，将ssthresh赋给cwnd，并结束快速恢复，迁移到拥塞避免状态。所以，在快速恢复状态，TCP增长的量级在拥塞避免和慢启动之间。快速恢复的语义是，出了点小岔子，没问题稳一稳，回到正轨上后，既然出了点小岔子，那以后就小心一点。

最后，状态变迁。

现在已经有了对快速重传的处理。那超时重传怎么办？如果出现超时重传，无论在哪个状态都迁移到慢启动，将cwnd重置为1。这样，这三个状态都可以两两互相迁移到。TCP的拥塞控制就在迁移状态中度过。

最后的最后，上图。

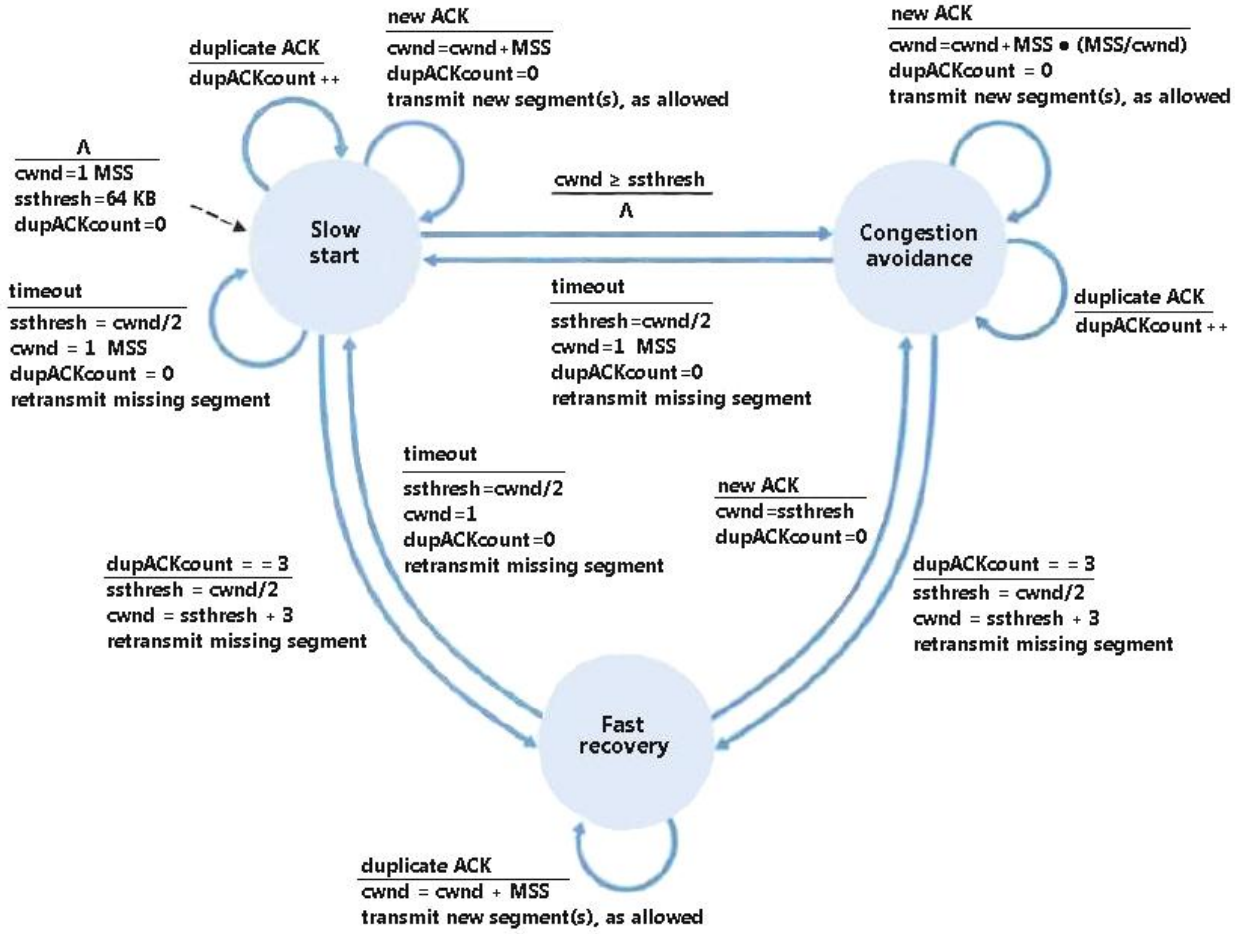


Figure 2. FSM description of TCP congestion control

更多的问题

实际上到这里，TCP的核心已经聊完了。但是。。是的，你没有猜错，TCP还有更多的问题。

糊涂窗口综合症(Silly-Window-Syndrome)与Nagle算法

流量控制很好的照顾了接收方，但是也引来了问题，如果接收方一直告诉发送方的剩余空间rwnd很小。那么发送方将一直发送内容很小的报文段。相对于TCP header20字节，如果每次内容只有个位数字节，那这样网络基本上就都在传输控制信息，网络使用率就太低了。这就出现了糊涂窗口综合症。而出现这种情况发送方和接收方自然也都有自己的应对办法。

对于接收方一般会使用David D Clark's的策略，就是"欺骗"发送方，如果是剩余空间很小的情况，干脆就通告发送方剩余空间是零，这样发送方就不会再发送小内容了。等到剩余空间超过1单位或者剩余空间超过缓冲的一半的时候，再不"欺骗"发送方。

对于发送方会使用Nagle算法。就是对于小内容的停等协议。如果是小内容的话，要查看是否所有已发送的小内容都已被确认，都被确认才能发送，这就形成了对小内容的停止发送等待确认的协议。Nagle认为小内容就是小于1单位(mss)的量。

实际上Nagle算法不光是设计来解决发送方的糊涂窗口综合征，它还减轻了拥塞。它可以将多个等待的小内容合并成一个数据报发送。这样直接的减少数据报的数量，从而减轻了拥塞。

TCP保活(Keep-Alive)

TCP的保活与HTTP的长连接不是一个意思，HTTP的长连接是复用TCP连接，减少连接时延。而TCP的保活是检查连接的对方是否还响应。一般是服务器端对客户端经行保活，如果客户端不响应，服务器端就不浪费资源，断掉连接。TCP自然是使用计时器来实现保活，超时时间默认为两小时。如果对方无反应，每隔75秒需重试9次。有趣的是，如果对方重启了或者说崩溃后又恢复了，对方接到保护探测报文段后会设置RST flag(复位)返回给发送方，然后发送方会断掉连接。

总结

TCP协议是个可靠协议，通过序号、校验和、超时重传、快速重传、确认来做到这点。并且还要照顾接收方和网络总体状况，主要体现在流量控制和拥塞控制。它还是个会建立连接的协议，需要在双方记录一些状态去跟踪传输过程。并提供端到端的服务。

IP

IP协议最大的任务就是寻路，找到发往目的地的路径然后发送过去，也就是说IP协议提供“点到点”的服务。IP协议不是可靠传输协议，只能尽力将数据报(digram)发送到目的地。这也代表着，数据报和数据包之间是独立的，没有状态。相对与TCP协议像是可变数据，IP协议就像是不可变数据。实际上，IP协议无状态流就像是响应式编程，具体点我。

IP协议寻路

先聊一下IP协议如何寻路，IP协议不可能一次性将数据报发送到目的地，必须经过多个中转站。如果要求一次性发送到目的地，要求双方有个独有的连接，然而为网络上所有人都建立这样一个连接是不可能的。并且这个中转站不可能强大到知道整个网络的拓扑结构，它只知道周围的节点的拓扑结构。

这就呈现出了IP寻路模型。路由器充当中转站的角色，主机和路由器都有一个路由表，路由表指示周围路由器的拓扑结构，就像一个地图一样，数据报通过查询路由表的结果寻路到下一个路由器。下一个路由器以同样方式负责寻路到再下一个路由器。这样，每一个路由器只负责到下一跳路由器(next-hop router)。最后IP协议通过多个路由器就到达了目的地。路由表不仅可以通过精确的目的地主机号寻路，还可以以子网的网络号寻路。当然还有保底的默认路径。

子网实际上作为比主机更大粒度的划分网络，以子网寻路可以极大的减少路由表的体积。相当于通过加大划分的粒度，减少了维护整个网络系统的成本。

IP协议寻路还有更多的问题。比如，主机也可以将数据报以发送给自己，当发现IP地址是自己时，数据报会交给以太网环回程序，环回程序将数据报加入本地的IP队列与其他数据报一视同仁。

主机可以被设置成路由器转发数据报，如果主机接收到了不是自己IP地址的数据报，只要被设置可以转发出去。但是如果没有寻路到下一跳怎么办？主机要返回一个ICMP(网络控制消息协议)，代表差错。

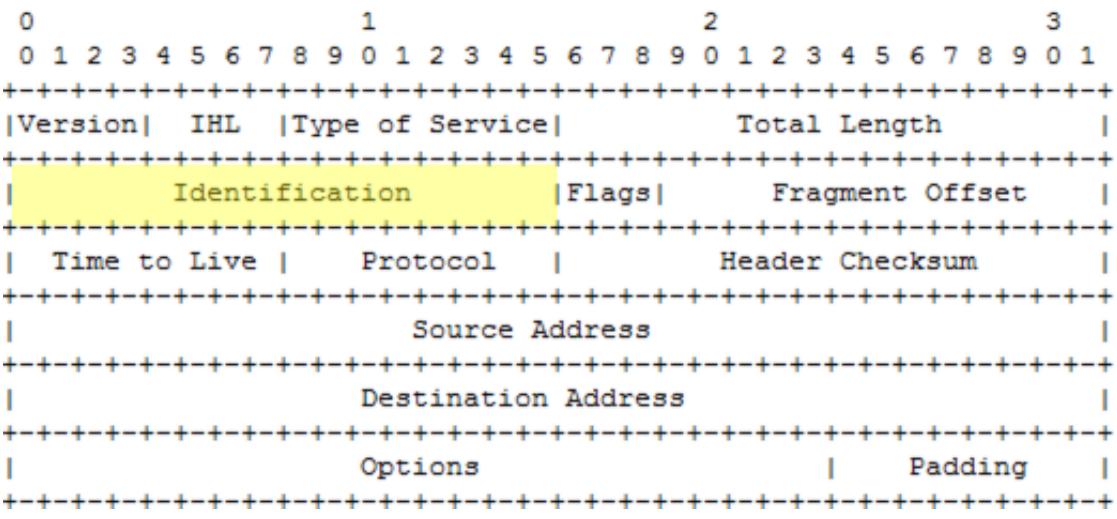
ICMP还可以用来重定向，比如说主机想发送一个数据报到目的地，可以发送给A和B，寻路的结果下一跳为A，主机发送给了A。A寻路的下一跳为B，发送给了B，A可以侦测出这个情况，然后发送给主机一个重定向ICMP，让主机的路由表修改为寻路到B。

数据分片(IP-Fragmentation)

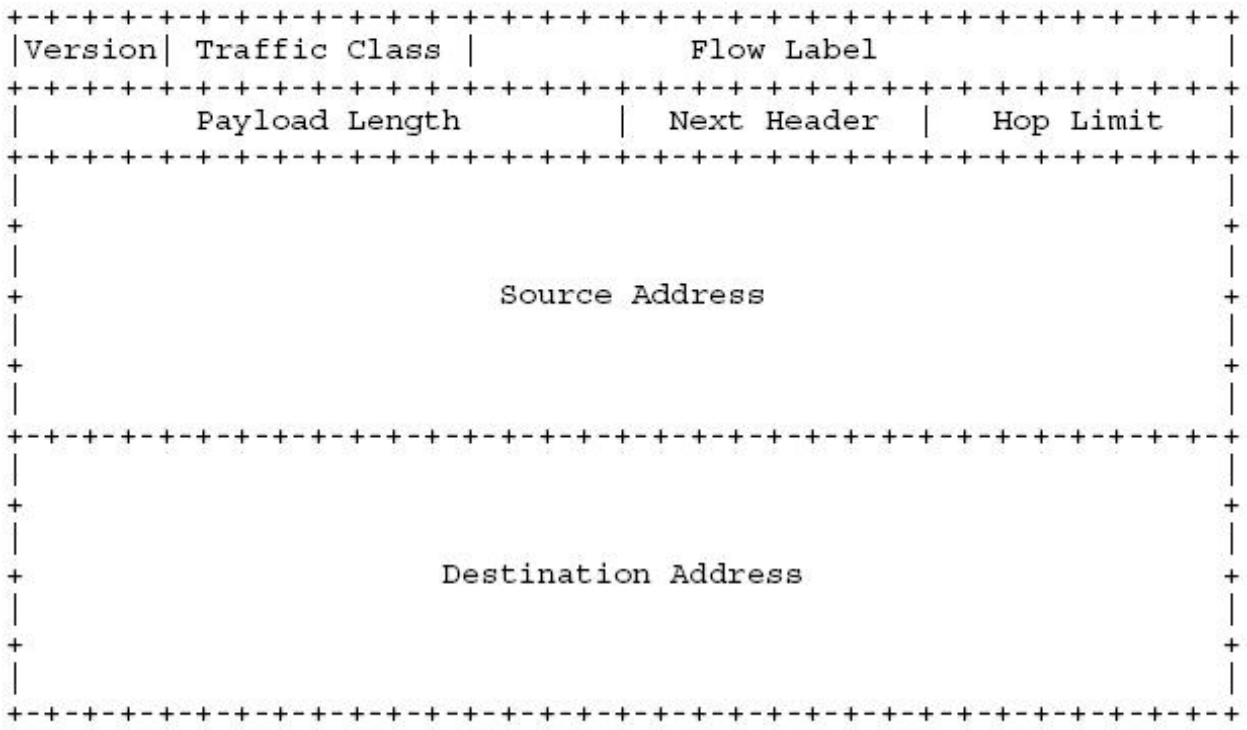
当数据报量超过了MTU怎么办？对比于TCP的分段，IP要分片。然而，这两个步骤互不干扰，是完全隔离开的。IP分片后，接收方接收到数据报后，将分片要重新组合起来。IP分片对于UDP协议比较有用，对TCP没有太大用处，TCP更希望自己来分段，而不靠IP去分片。IP不是个可靠协议，如果分片其中的一片出了问题，TCP也无法重传单个分片，自然TCP就更希望自己来分段，做到重传单个分段。

IP Header

IP协议分为IPv4和IPv6版本，两种版本header不相同，版本在Version区域区分。首先IPv4，



其次，IPv6



这里，就捡几个体现出IP服务的说一下，相比于TCP。IP协议的header没有那么能体现出IP协议的特点。

IPv4 header一般为20字节，IPv6 header一般为40字节。IPv4中address为32位，而IPv6增大到了128位。这样就从address分配紧张到地球上的每一颗砂砾都能有IP address了！

TTL和Hop Limit都是表示IP协议还能跳的路由器数量，如果为零了，则数据报会被丢弃，并返回一个ICMP通知源主机。Traceroute程序就是这样收集数据报被丢弃后发送的ICMP实现的。

IPv4用Identification唯一识别数据报(分片数据报相同)，Fragment offset标识分片的起始位置。而在IPv6中都可以用更加灵活的Next Header表示，Next Header就像链表一样，可以连接多个"Header"，拓展出多个Header。除了分片的起始位置、还可以表示同IPv4 protocol一样能表示的上层协议。

参考链接

TCP

TCP/IP详解 卷1:协议

计算机网络自顶向下方法

复用的精妙 - UITableView 复用技术原理分析

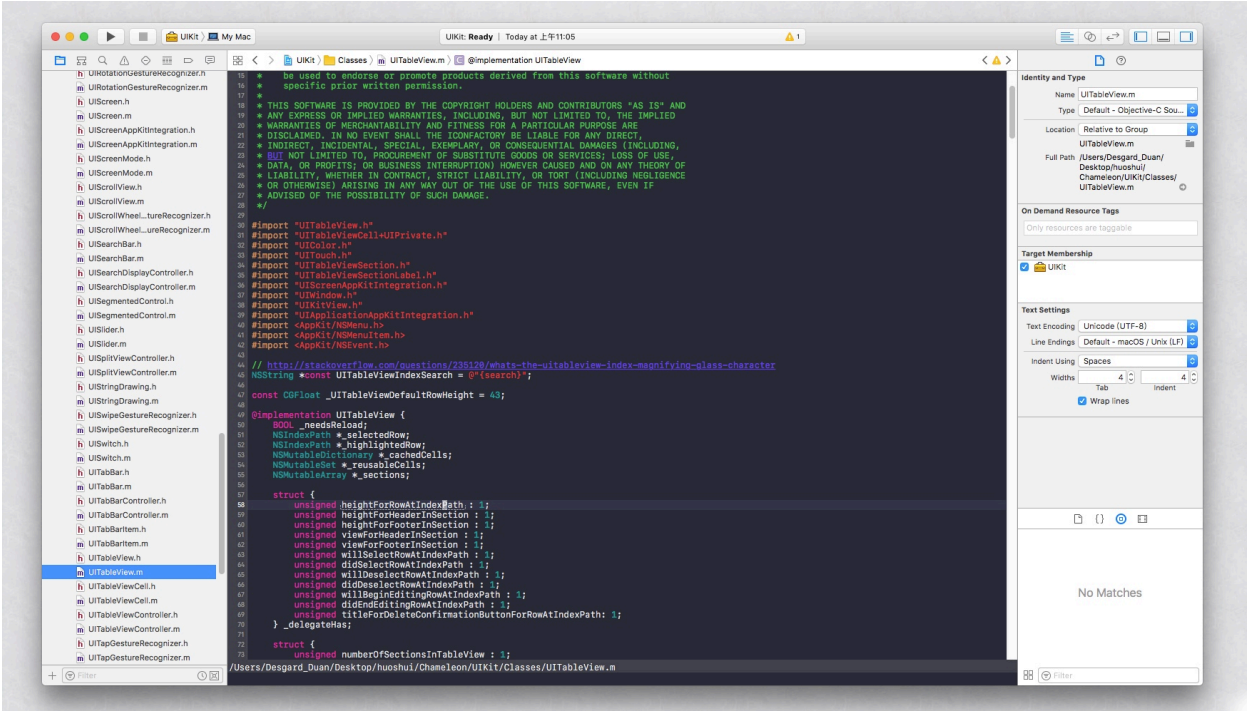
作者：冬瓜

在现在很多公司的 app 中，许多展示页面为了多条数据内容，而采用 `UITableView` 来设计页面。在滑动 `UITableView` 的时候，并不会因为数据量大而产生卡顿的情况，这正是因为其**复用机制**的特点。但是其复用机制是如何实现的？我决定来探索一番。

Chameleon PROJECT

Chameleon 是我长期以来一直关注的一个项目。接触过 macOS 开发的人肯定多少有写了解。（虽然这个项目在三年以前就已经停更，但是在原理上还是有很高的参考价值。）*Chameleon* 用于将 iOS 的功能迁移到 macOS 上，并且在其中为 macOS 实现了一套与 iOS UIKit 同名的框架，并且其代码都为开源。由于 *Chameleon* 属于对苹果早期源码的逆向工程项目，所以我们可以据此来对 iOS 一些闭源库展开学习和思路的借鉴。

> *Chameleon* 所迁移的 iOS 版本为 `3.2`，如今已经没有人使用，所以其代码和思路我们只能用来了解。例如在 iOS 8 之后推出的根据 `autoLayout` 自动计算 `cell` 高度的功能，在其中无法体现。



UITableView 的初始化方法

当我们定义一个 `UITableView` 对象的时候，需要对这个对象进行初始化。最常用的方法莫过于 `-(id)initWithFrame:(CGRect)frame style:(UITableViewStyle)theStyle`。下面跟着这个初始化入口，逐渐来分析代码：

```
- (id)initWithFrame:(CGRect)frame style:(UITableViewStyle)theStyle {
    if ((self=[super initWithFrame:frame])) {
        // 确定 TableView 的 Style
        _style = theStyle;
        // 要点一：Cell 缓存字典
        _cachedCells = [[NSMutableDictionary alloc] init];
        // 要点二：Section 缓存 Mutable Array
        _sections = [[NSMutableArray alloc] init];
        // 要点三：复用 Cell Mutable Set
        _reusableCells = [[NSMutableSet alloc] init];

        // 一些关于 Table View 的属性设置
        self.separatorColor = [UIColor colorWithRed:.88f green:.88f
blue:.88f alpha:1];
        self.separatorStyle = UITableViewCellSeparatorStyleSingleLine;
        self.showsHorizontalScrollIndicator = NO;
        self.allowsSelection = YES;
        self.allowsSelectionDuringEditing = NO;
        self.sectionHeaderHeight = self.sectionFooterHeight = 22;
        self.alwaysBounceVertical = YES;

        if (_style == UITableViewStylePlain) {
            self.backgroundColor = [UIColor whiteColor];
        }
        // 加入 Layout 标记，进行手动触发布局设置
        [self _setNeedsReload];
    }
    return self;
}
```

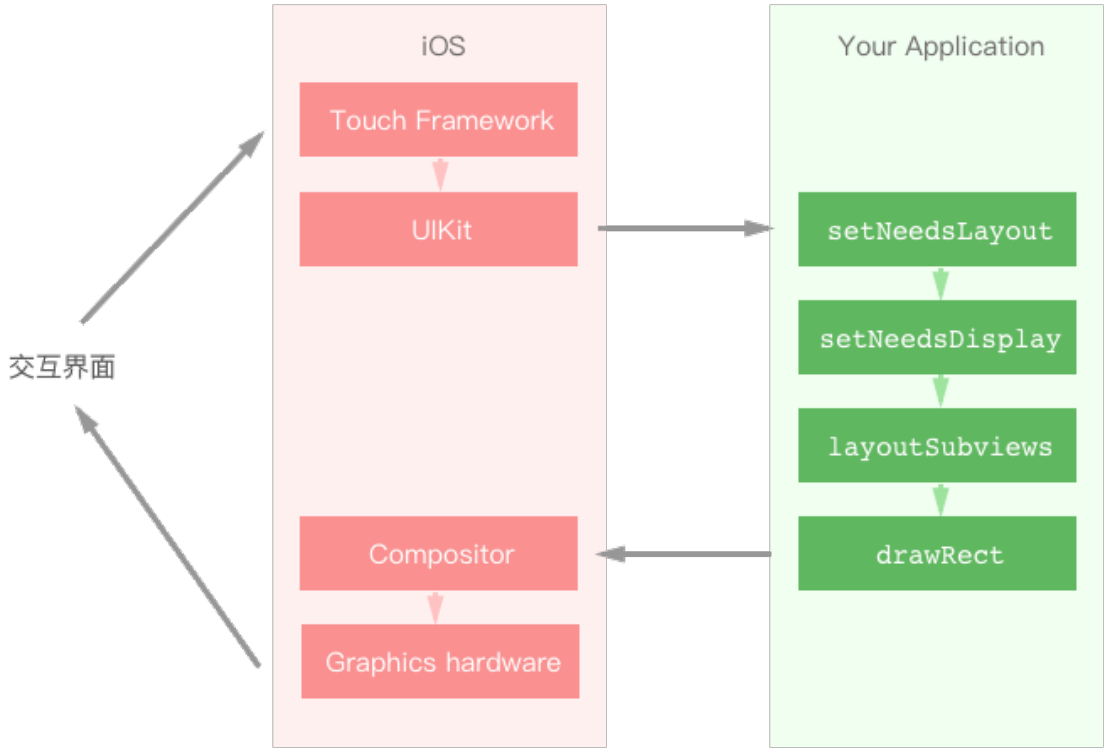
在初始化代码中就看到了重点，`_cachedCells`、`_sections` 和 `_reusableCells` 无疑是复用的核心成员。

代码跟踪

我们先来查看一下 `_setNeedsReload` 方法中做了什么：

```
- (void)_setNeedsReload {
    _needsReload = YES;
    [self setNeedsLayout];
}
```

首先先对 `_needsReload` 进行标记，之后调用了 `setNeedsLayout` 方法。对于 `UIView` 的 `setNeedsLayout` 方法，在调用后 *Runloop* 会在即将到来的周期中来检测 `displayIfNeeded` 标记，如果为 `YES` 则会进行 `drawRect` 视图重绘。作为 Apple *UIKit* 层中的基础 Class，在属性变化后都会进行一次视图重绘的过程。这个属性过程的变化即为对象的初始化加载以及手势交互过程。这也就是官方文档中的 *The Runtime Interaction Model*。



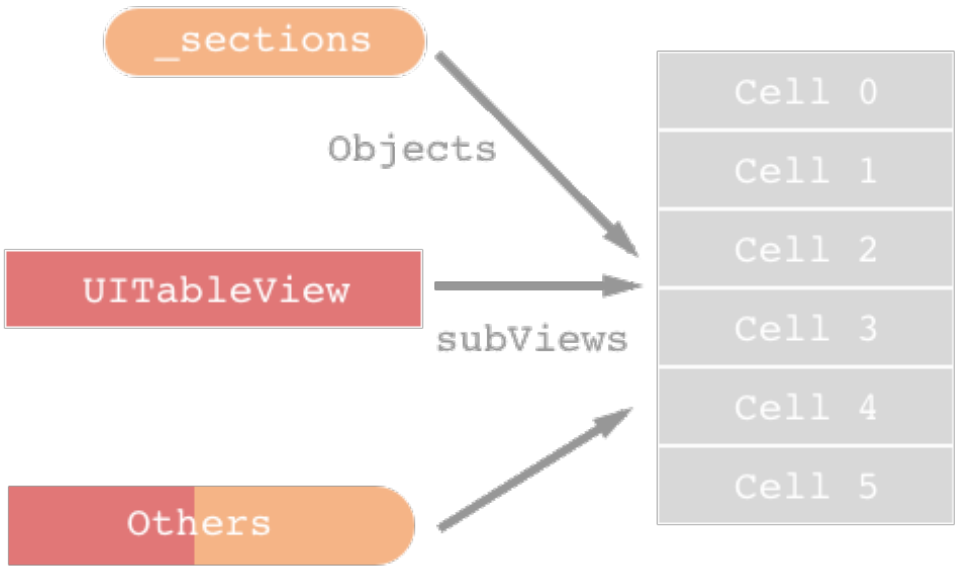
当 *Runloop* 到来时，开始重绘过程即调用 `layoutSubViews` 方法。在 `UITableView` 中这个方法已经被重写过：

```
- (void)layoutSubviews {
    // 会在初始化的末尾手动调用重绘过程
    // 并且 UITableView 是 UIScrollView 的继承，会接受手势
    // 所以在滑动 UITableView 的时候也会调用
    _backgroundView.frame = self.bounds;
    // 根据标记确定是否执行数据更新操作
    [self _reloadDataIfNeeded];
    // 布局入口
    [self _layoutTableView];
    [super layoutSubviews];
}
```

接下来我们开始查看 `_reloadDataIfNeeded` 以及 `reloadData` 方法：

```
- (void)_reloadDataIfNeeded {
    // 查询 _needsReload 标记
    if (_needsReload) {
        [self reloadData];
    }
}

- (void)reloadData {
    // 清除之前的缓存并删除 Cell
    // makeObjectsPerformSelector 方法值都进行调用某个方法
    [[_cachedCells allValues]
makeObjectsPerformSelector:@selector(removeFromSuperview)];
    // 复用 Cell Set 也进行删除操作
    [_reusableCells
makeObjectsPerformSelector:@selector(removeFromSuperview)];
    [_reusableCells removeAllObjects];
    [_cachedCells removeAllObjects];

    // 删除选择的 Cell
    _selectedRow = nil;
    // 删除被高亮的 Cell
    _highlightedRow = nil;

    // 更新缓存中状态
    [self _updateSectionsCache];
    // 设置 Size
    [self _setContentSize];

    _needsReload = NO;
}
```

当 `reloadData` 方法被触发时，`UITableView` 默认为在这个 `UITableView` 中的数据将会全部发生变化。测试之前遗留下的缓存列表以及复用列表全部都丧失了利用性。为了避免出现**悬挂指针**的情况（有可能某个 cell 被其他的视图进行了引用），我们需要对每个 cell 进行 `removeFromSuperview` 处理，这个处理即针对于容器 `UITableView`，又对其他的引用做出保障。然后我们更新当前 tableView 中的两个缓存容器，`_reusableCells` 和 `_cachedCells`，以及其他需要重置的成员属性。



需要解除所有的持有关系

最关键的地方到了，缓存状态的更新方法 `_updateSectionsCache`，其中涉及到数据如何存储、如何复用的操作：

```
- (void)_updateSectionsCache {
    // 使用 dataSource 来创建缓存容器
    // 如果没有 dataSource 则放弃重用操作
    // 在这个逆向工程中并没有对 header 进行缓存操作，但是 Apple 的 UIKit 中一定也做到了
    // 真正的 UIKit 中应该会获取更多的数据进行存储，并实现了 TableView 中所有视图的复用

    // 先移除每个 Section 的 Header 和 Footer 视图
    for (UITableViewSection *previousSectionRecord in _sections) {
        [previousSectionRecord.headerView removeFromSuperview];
        [previousSectionRecord.footerView removeFromSuperview];
    }

    // 清除旧缓存，对容器进行初始化操作
    [_sections removeAllObjects];

    if (_dataSource) {
        // 根据 dataSource 计算高度和偏移量
        const CGFloat defaultRowHeight = _rowHeight ?: _UITableViewDefaultRowHeight;
        // 获取 Section 数目
        const NSInteger numberOfSections = [self numberOfSections];
        for (NSInteger section=0; section<numberOfSections; section++) {
```

```
            const NSInteger numberOfRowsInSection = [self
numberOfRowsInSection:section];

            UITableViewSection *sectionRecord = [[UITableViewSection alloc]
init];
            sectionRecord.headerTitle =
_dataSourceHas.titleForHeaderInSection? [self.dataSource tableView:self
titleForHeaderInSection:section] : nil;
            sectionRecord.footerTitle =
_dataSourceHas.titleForFooterInSection? [self.dataSource tableView:self
titleForFooterInSection:section] : nil;

            sectionRecord.headerHeight =
_delegateHas.heightForHeaderInSection? [self.delegate tableView:self
heightForHeaderInSection:section] : _sectionHeaderHeight;
            sectionRecord.footerHeight =
_delegateHas.heightForFooterInSection ? [self.delegate tableView:self
heightForFooterInSection:section] : _sectionFooterHeight;

            sectionRecord.headerView = (sectionRecord.headerHeight > 0 &&
_delegateHas.viewForHeaderInSection)? [self.delegate tableView:self
viewForHeaderInSection:section] : nil;
            sectionRecord.footerView = (sectionRecord.footerHeight > 0 &&
_delegateHas.viewForFooterInSection)? [self.delegate tableView:self
viewForFooterInSection:section] : nil;

            // 先初始化一个默认的 headerView , 如果没有直接设置 headerView 就直接更
换标题
            if (!sectionRecord.headerView && sectionRecord.headerHeight > 0
&& sectionRecord.headerTitle) {
                sectionRecord.headerView = [UITableViewSectionLabel
sectionLabelWithTitle:sectionRecord.headerTitle];
            }

            // Footer 也做相同的处理
            if (!sectionRecord.footerView && sectionRecord.footerHeight > 0
&& sectionRecord.footerTitle) {
                sectionRecord.footerView = [UITableViewSectionLabel
sectionLabelWithTitle:sectionRecord.footerTitle];
            }

            if (sectionRecord.headerView) {
                [self addSubview:sectionRecord.headerView];
            } else {
                sectionRecord.headerHeight = 0;
            }

            if (sectionRecord.footerView) {
                [self addSubview:sectionRecord.footerView];
```

```
            } else {
                sectionRecord.footerHeight = 0;
            }

            // 为高度数组动态开辟空间
            CGFloat *rowHeights = malloc(numberOfRowsInSection *
sizeof(CGFloat));
            // 初始化总高度
            CGFloat totalRowsHeight = 0;

            for (NSInteger row=0; row<numberOfRowsInSection; row++) {
                // 获取 Cell 高度，未设置则使用默认高度
                const CGFloat rowHeight =
_delegateHas.heightForRowAtIndexPath? [self.delegate tableView:self
heightForRowAtIndexPath:[NSIndexPath indexPathForRow:row inSection:section]]
: defaultRowHeight;
                // 记录高度
                rowHeights[row] = rowHeight;
                // 总高度统计
                totalRowsHeight += rowHeight;
            }

            sectionRecord.rowsHeight = totalRowsHeight;
            [sectionRecord setNumberOfRows:numberOfRowsInSection
withHeights:rowHeights];
            free(rowHeights);

            // 缓存高度记录
            [_sections addObject:sectionRecord];
        }
    }
}
```

我们发现在 `_updateSectionsCache` 更新缓存状态的过程中对 `_sections` 中的数据全部清除。之后缓存了更新后的所有 Section 数据。那么这些数据有什么利用价值呢？继续来看**布局更新**操作。

```
- (void)_layoutTableView {
    // 在需要渲染时放置需要的 Header 和 Cell
    // 缓存所有出现的单元格，并添加至复用容器
    // 之后那些不显示但是已经出现的 Cell 将会被复用

    // 获取容器视图相对于父类视图的尺寸及坐标
    const CGSize boundsSize = self.bounds.size;
    // 获取向下滑动偏移量
    const CGFloat contentOffset = self.contentOffset.y;
    // 获取可视矩形框的尺寸
    const CGRect visibleBounds =
CGRectMake(0,contentOffset,boundsSize.width,boundsSize.height);
    // 表高纪录值
```

```
    CGFloat tableHeight = 0;
    // 如果有 header 则需要额外计算
    if (_tableHeaderView) {
        CGRect tableHeaderFrame = _tableHeaderView.frame;
        tableHeaderFrame.origin = CGPointZero;
        tableHeaderFrame.size.width = boundsSize.width;
        _tableHeaderView.frame = tableHeaderFrame;
        tableHeight += tableHeaderFrame.size.height;
    }

    // availableCell 记录当前正在显示的 Cell
    // 在滑出显示区之后将添加至 _reusableCells
    NSMutableDictionary *availableCells = [_cachedCells mutableCopy];
    const NSInteger numberOfSections = [_sections count];
    [_cachedCells removeAllObjects];

    // 滑动列表，更新当前显示容器
    for (NSInteger section=0; section<numberOfSections; section++) {
        CGRect sectionRect = [self rectForSection:section];
        tableHeight += sectionRect.size.height;
        if (CGRectIntersectsRect(sectionRect, visibleBounds)) {
            const CGRect headerRect = [self rectForHeaderInSection:section];
            const CGRect footerRect = [self rectForFooterInSection:section];
            UITableViewSection *sectionRecord = [_sections
objectAtIndex:section];
            const NSInteger numberOfRows = sectionRecord.numberOfRows;

            if (sectionRecord.headerView) {
                sectionRecord.headerView.frame = headerRect;
            }

            if (sectionRecord.footerView) {
                sectionRecord.footerView.frame = footerRect;
            }

            for (NSInteger row=0; row<numberOfRows; row++) {
                // 构造 indexPath 为代理方法准备
                NSIndexPath *indexPath = [NSIndexPath indexPathForRow:row
inSection:section];
                // 获取第 row 个坐标位置
                CGRect rowRect = [self rectForRowAtIndexPath:indexPath];
                // 判断当前 Cell 是否与显示区域相交
                if (CGRectIntersectsRect(rowRect,visibleBounds) &&
rowRect.size.height > 0) {
                    // 首先查看 availableCells 中是否已经有了当前 Cell 的存储
                    // 如果没有，则请求 tableView 的代理方法获取 Cell
                    UITableViewCell *cell = [availableCells
objectForKey:indexPath] ?: [self.dataSource tableView:self
cellForRowAtIndexPath:indexPath];
```

```
                    // 由于碰撞检测生效，则按照逻辑需要更新 availableCells 字典
                    if (cell) {
                        // 获取到 Cell 后，将其进行缓存操作
                        [_cachedCells setObject:cell forKey:indexPath];
                        [availableCells removeObjectForKey:indexPath];
                        cell.highlighted = [_highlightedRow
isEqual:indexPath];
                        cell.selected = [_selectedRow isEqual:indexPath];
                        cell.frame = rowRect;
                        cell.backgroundColor = self.backgroundColor;
                        [cell _setSeparatorStyle:_separatorStyle
color:_separatorColor];
                        [self addSubview:cell];
                    }
                }
            }
        }
    }

    // 将已经退出屏幕且定义 reuseIdentifier 的 Cell 加入可复用 Cell 容器中
    for (UITableViewCell *cell in [availableCells allValues]) {
        if (cell.reuseIdentifier) {
            [_reusableCells addObject:cell];
        } else {
            [cell removeFromSuperview];
        }
    }

    // 不能复用的 Cell 会直接销毁，可复用的 Cell 会存储在 _reusableCells

    // 确保所有的可用（未出现在屏幕上）的复用单元格在 availableCells 中
    // 这样缓存的目的之一是确保动画的流畅性。在动画的帧上都会对显示部分进行处理，重新计算
可见 Cell。
    // 如果直接删除掉所有未出现在屏幕上的单元格，在视觉上会观察到突然消失的动作
    // 整体动画具有跳跃性而显得不流畅

    // 把在可视区的 Cell（但不在屏幕上）已经被回收为可复用的 Cell 从视图中移除
    NSArray* allCachedCells = [_cachedCells allValues];
    for (UITableViewCell *cell in _reusableCells) {
        if (CGRectIntersectsRect(cell.frame,visibleBounds) && !
[allCachedCells containsObject: cell]) {
            [cell removeFromSuperview];
        }
    }

    if (_tableFooterView) {
        CGRect tableFooterFrame = _tableFooterView.frame;
        tableFooterFrame.origin = CGPointMake(0,tableHeight);
        tableFooterFrame.size.width = boundsSize.width;
```

```
        _tableFooterView.frame = tableFooterFrame;
    }
}
```

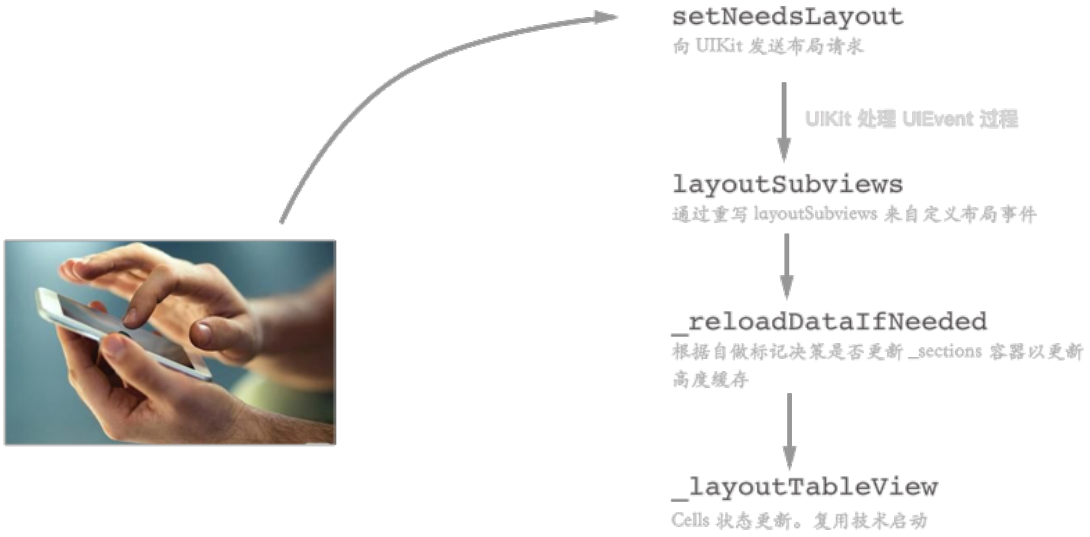
`CGRectIntersectsRect` 方法用于检测两个 Rect 的碰撞情况。如下图所示：

如果你已经对 `UITableView` 的缓存机制有所了解，那么你在阅读完代码之后会对其有更深刻的认识。如果看完代码还是一头雾水，那么请继续看下面的分析。

Cell 复用场景三个阶段

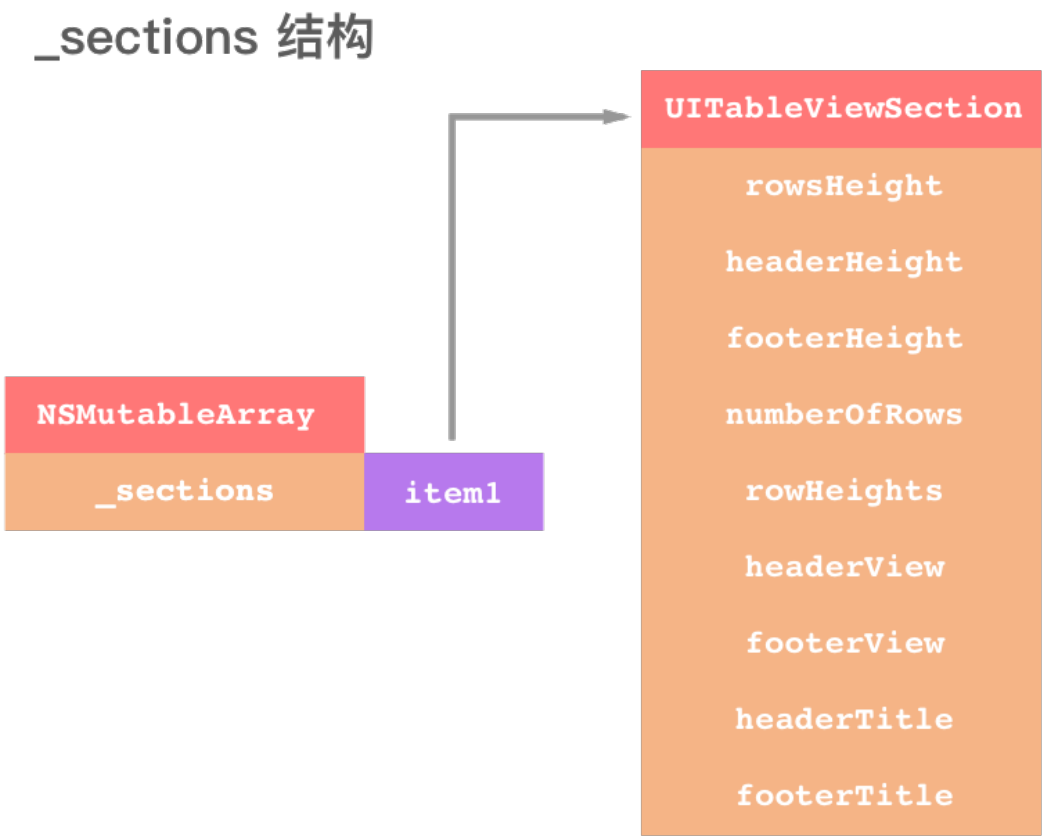
布局方法触发阶段

在用户触摸屏幕后，硬件报告触摸时间传递至 `UIKit` 框架，之后 `UIKit` 将触摸事件打包成 `UIEvent` 对象，分发至指定视图。这时候其视图就会做出相应，并调用 `setNeedsLayout` 方法告诉视图及其子视图需要进行布局更新。此时，`setNeedsLayout` 被调用，也就变为 Cell 复用场景的入口。



缓存 Cell 高度信息阶段

当视图加载后，由 `UIKit` 调用布局方法 `layoutSubviews` 从而进入**缓存 Cell 高度阶段** `_updateSectionsCache`。在这个阶段，通过代理方法 `heightForRowAtIndexPath:` 获取每一个 Cell 的高度，并将高度信息缓存起来。这其中的高度信息由 `UITableViewSection` 的一个实例 `sectionRecord` 进行存储，其中以 section 为单位，存储每个 section 中各个 Cell 的高度、Cell 的数量、以及 section 的总高度、footer 和 header 高度这些信息。这一部分的信息采集是为了在 Cell 复用的核心部分，Cell 的 Rect 尺寸与 tableView 尺寸计算边界情况建立数据基础。



复用 Cell 的核心处理阶段

我们要关注三个存储容器的变化情况：

- `NSMutableDictionary` 类型 **_cachedCells**：用来存储当前屏幕上所有 Cell 与其对应的 indexPath。以键值对的关系进行存储。
- `NSMutableDictionary` 类型 **availableCells**：当列表发生滑动的时候，部分 Cell 从屏幕移出，这个容器会对 `_cachedCells` 进行拷贝，然后将屏幕上此时的 Cell 全部去除。即最终取出所有退出屏幕的 Cell。
- `NSMutableSet` 类型 **_reusableCells**：用来收集曾经出现过此时未出现在屏幕上的 Cell。当再出滑入主屏幕时，则直接使用其中的对象根据 `CGRectIntersectsRect` Rect 碰撞试验进行复用。

在整个核心复用阶段，这三个容器都充当着很重要的角色。我们给出以下的场景实例，例如下图的一个场景，图 ① 为页面刚刚载入的阶段，图 ② 为用户向下滑动一个单元格时的状态：

当到状态 ② 的时候，我们发现 `_reusableCells` 容器中，已经出现了状态 ① 中已经退出屏幕的 Cell 0。而当我们重新将 Cell 0 滑入界面的时候，在系统 `addView` 渲染阶段，会直接将 `_reusableCells` 中的 Cell 0 立即取出进行渲染，从而代替创建新的实例再进行渲染，简化了时间与性能上的开销。

UITableView 的其他细节优化

复用容器数据类型 `NSMutableSet`

在三个重要的容器中，只有 `_reusableCells` 使用了 `NSMutableSet`。这是因为我们在每一次对于 `_cachedCells` 中的 Cell 进行遍历并在屏幕上渲染时，都需要在 `_reusableCells` 进行一次扫描。而且当一个页面反复的上下滑动时，`_reusableCells` 的检索复杂度是相当庞大的。为了确保这一情况下滑动的流畅性，Apple 在设计时不得不将检索复杂度最小化。并且这个复杂度要是非抖动的，不能给体验造成太大的不稳定性。

在 C++ 的 STL 标准库中也有 `multiset` 数据类型，其中实现的方法是通过构建**红黑树**来实现。因为红黑树具有高效检索的性质，这也是 `set` 的一个普遍特点。也许是 `NSMutableSet` 是 *Foundation* 框架的数据结构，构造其主要目的是为了更快的检索。所以 `NSMutableSet` 的实现并没有使用红黑树，而是暴力的使用 **Hash 表**实现。从 *Core Foundation* 中的 CFSet.c 可以清晰的看见其底层实现。在很久之前的 Cocoa Dev 的提问帖中也能发现答案。

高度缓存容器 _sections

在每次布局方法触发阶段，由于 Cell 的状态发生了变化。在对 Cell 复用容器的修改之前，首先要做的一件事是以 Section 为单位对所有的 Cell 进行缓存高度。从这里可以看出 `UITableView` 设计师的细节。 Cell 的高度在 `UITableView` 中充当着十分重要的角色，一下列表是需要使用高度的方法：

- `- (CGFloat)_offsetForSection:(NSInteger)index`：计算指定 Cell 的滑动偏移量。

- `- (CGRect)rectForSection:(NSInteger)section`：返回某个 Section 的整体 Rect。
- `- (CGRect)rectForHeaderInSection:(NSInteger)section`：返回某个 Header 的 Rect。
- `- (CGRect)rectForFooterInSection:(NSInteger)section`：返回某个 Footer 的 Rect。
- `- (CGRect)rectForRowAtIndexPath:(NSIndexPath *)indexPath`：返回某个 Cell 的 Rect。
- `- (NSArray *)indexPathsForRowsInRect:(CGRect)rect`：返回 Rect 列表。
- `- (void)_setContentSize`：根据高度计算 `UITableView` 中实际内容的 Size。

一次有拓展性的源码研究

在阅读完 Chameleon 工程中的 `UITableView` 源码，进一步可以去查看 `FDTemplateLayoutCell` 的优化方案。Apple 的工程师对于细节的处理和方案值得各位开发者细细寻味。多探求、多阅读以写出更优雅的代码。😄

试图取代 TCP 的 QUIC 协议到底是什么

作者：张星宇，@bestswifter

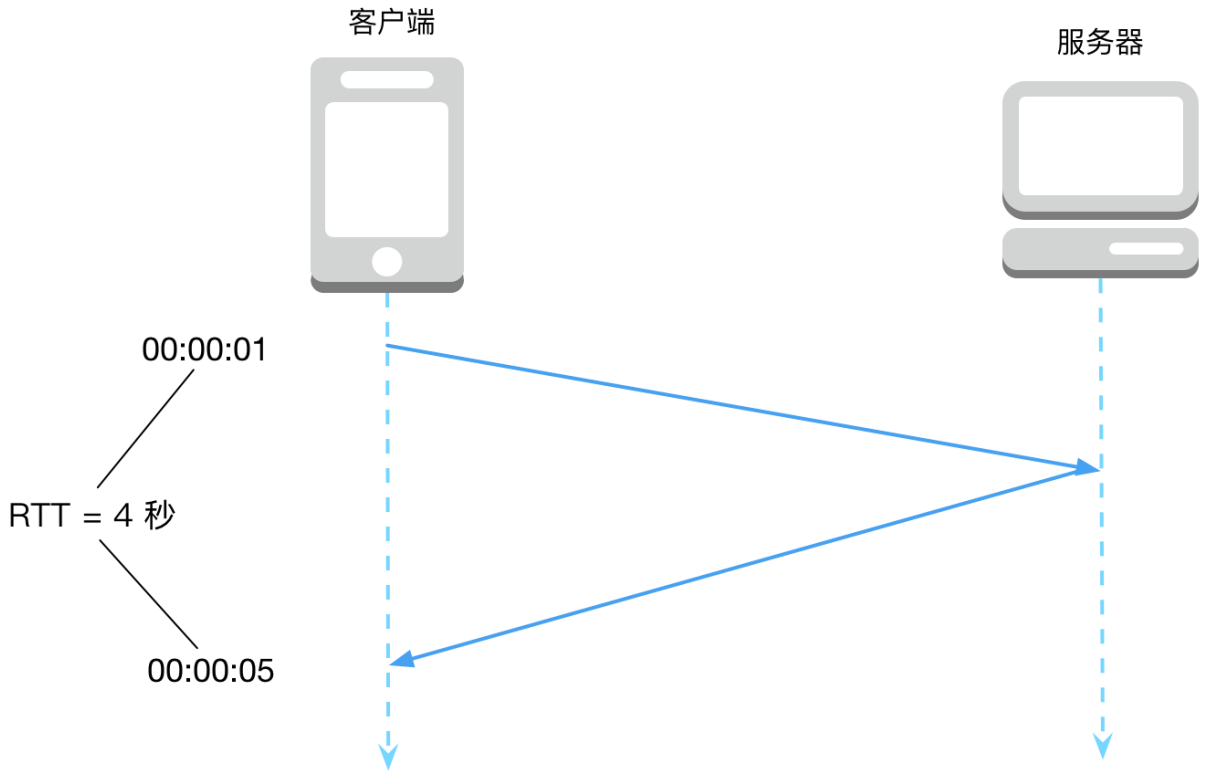
QUIC 协议是 Google 提出的一套开源协议，它基于 UDP 来实现，直接竞争对手是 TCP 协议。QUIC 协议的性能非常好，甚至在某些场景下可以实现 0-RTT 的加密通信。所以这篇文章主要讨论以下几个问题:

1. QUIC 协议有哪些优点，如何实现 0-RTT
2. 传统的 HTTP2 + SSL + TCP 协议栈有哪些缺点
3. 为什么 Google 要另起炉灶，基于 UDP 去做

这三个问题并不独立，所以我不会依次解答他们，我选择从一些常识入手，逐渐深入到底层细节。希望读者能够带着问题阅读并最终获得满意的答案。

RTT

网络请求中一个常见的名词是 RTT(Round Trip Time)，表示客户端从发出一个请求数据，到接收到响应数据之间间隔的时间。



RTT 可以理解成由两部分组成，一部分受到物理条件的限制，比如间隔距离除以信号传递速度，以及包大小除以带宽。另一部分则是客户端、服务器以及沿途各路由器对包的处理解析时间。一般情况下，RTT 大约在几十毫秒左右，网络很好的情况下可以达到个位数，恶劣网络环境下达到几百毫秒也有可能。

根据阮一峰的 SSL延迟有多大？ 一文中做的计算，用户访问支付宝时，一个 RTT 大约需要 22ms，算上 SSL 握手的三个 RTT 则大约消耗了 64ms。

可见网络请求时间绝对不是简单的数据量除以网速这么简单， RTT 是网络请求耗时中不可忽略的一部分，不仅仅是握手阶段需要三个 RTT，在实际网络请求中，还有可能因为丢包等问题而额外增加 RTT。因此任何一个能减少 RTT 的技术都值得认真考虑，因为他们**真的能够显著降低**网络请求耗时。

QUIC 协议概述

下面进入正题，本文的主角，QUIC 协议主要具备以下优点:

1. 多路复用，避免队头阻塞
2. 减少 RTT，请求更快速
3. 快速迭代，广泛支持

多路复用，避免队头阻塞

这句话说起来很容易，但理解起来并不那么显然，要想理解 QUIC 协议到底做了什么以及这么做的必要性，我想还是从最基础的 HTTP 1.0 聊起比较合适。

Pipiline

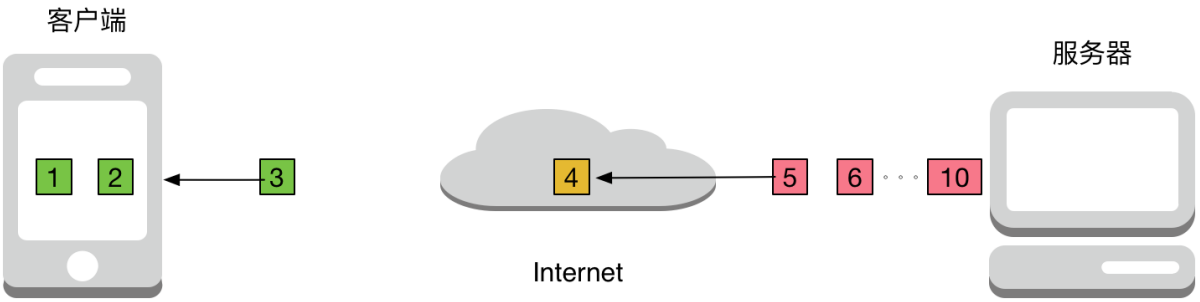
根据谷歌的调查， 现在请求一个网页，平均涉及到 80 个资源，30 多个域名。考虑最原始的情况，每请求一个资源都需要建立一次 TCP 请求，显然不可接受。HTTP 协议规定了一个字段 `Connection`，不过默认的值是 `close`，也就是不开启。

早在 1999 年提出的 HTTP 1.1 协议 中就把 `Connection` 的默认值改成了 `Keep-Alive`，这样同一个域名下的多个 HTTP 请求就可以复用同一个 TCP 连接。这种做法被称为 HTTP Pipeline，优点是显著的减少了建立连接的次数，也就是大幅度减少了 RTT。以上面的数据为例，如果 80 个资源都要走一次 HTTP 1.0，那么需要建立 80 个 TCP 连接，握手 80 次，也就是 80 个 RTT。如果采用了 HTTP 1.1 的 Pipeline，只需要建立 30 个 TCP 连接，也就是 30 个 RTT，提高了 **62.5%** 的效率。

Pipeline 解决了 TCP 连接浪费的问题，但它自己还存在一些不足之处，也就是所有管道模型都难以避免的队头阻塞问题。

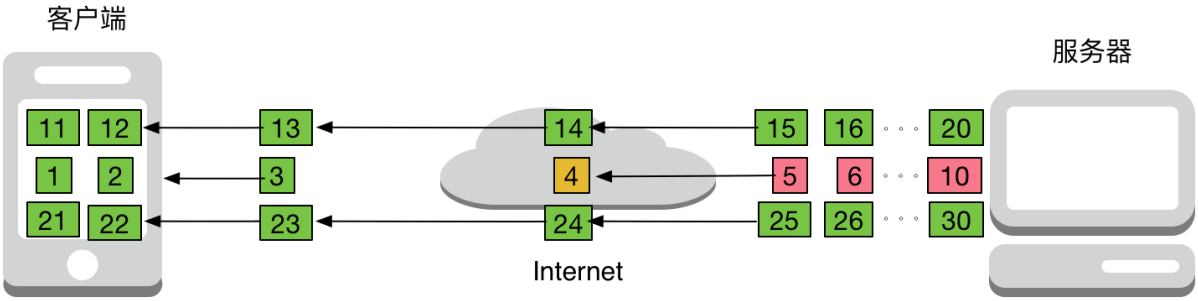
队头阻塞

我们再举个简单而且直观的例子，假设加载一个 HTML 一共要请求 10 个资源，那么请求的总时间是每一个资源请求时间的总和。最直观的体验就是，网速越快请求时间越短。然而如果某一个资源的请求被阻塞了(比如 SQL 语句执行非常慢)。但对于客户端来说所有后续的请求都会因此而被阻塞。



队头阻塞(Head of line blocking，下文简称 HOC)说的是当有多个串行请求执行时，如果第一个请求不执行完，后续的请求也无法执行。比如上图中，如果第四个资源的传输花了很久，后面的资源都得等着，平白浪费了很多时间，带宽资源没有得到充分利用。

因此，HTTP 协议允许客户端发起多个并行请求，比如在笔者的机器上最多支持六个并发请求。并发请求主要是用于解决 HOC 问题，当有三个并发请求时，情况会变成这样:

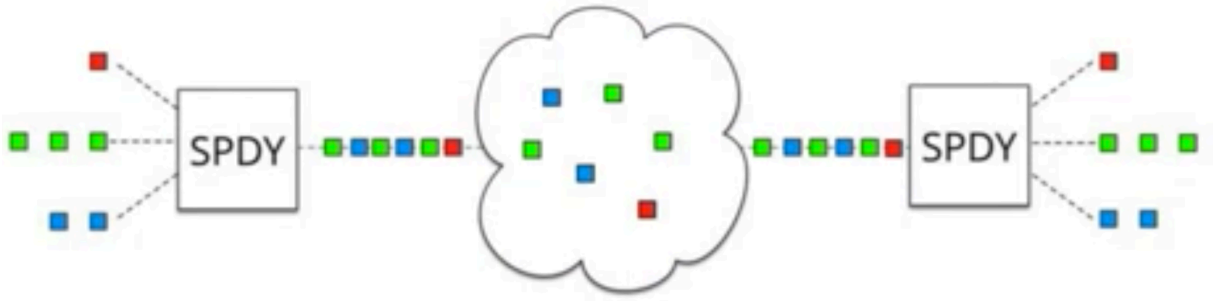


可见虽然第四个资源的请求被阻塞了，但是其他的资源请求并不一定会被阻塞，这样总的来说网络的平均利用率得到了提升。

支持并发请求是解决解决 HOC 问题的一种方案，这句话没有错，但是我们要理解到: “并发请求并非是直接解决了 HOC 的问题，而是尽可能减少 HOC 造成的影响”，以上图为例，HOC 的问题依然存在，只是不会太浪费带宽而已。有读者可能会好奇，为什么不多搞几个并发的 HTTP 请求呢？刚刚说过笔者的电脑最多支持 6 个并发请求，谷歌曾经做过实验，把 6 改成 10，然后尝试访问了三千多个网页，发现平均访问时间竟然还增加了 5% 左右。这是因为一次请求涉及的域名有限，再多的并发 HTTP 请求并不能显著提高带宽利用率，反而会消耗性能。

SPDY

有没有办法解决队头阻塞呢，答案是肯定的。SPDY 协议的做法很值得借鉴，它采用了多路复用 (Multiplexing) 技术，允许多个 HTTP 请求共享同一个 TCP 连接。我们假设每个资源被分为多个包传递，在 HTTP 1.1 中只有前面一个资源的所有数据包传输完毕后后面资源的包才能开始传递(HOC 问题)，而 SPDY 并不这么要求，大家可以一起传输。



这么做的代价是数据会略微有一些冗余，每一个资源的数据包都要带上标记，用来指明自己属于哪个资源，这样客户端最后才能把他们正确的拼接起来。不同的标记可以理解为图中不同的颜色，每一个小方格可以理解为资源的某一个包。

有些读者对 SPDY 协议可能不太了解，其实把它当做 HTTP2 的前身和试验品就好。当然 HTTP2.0 的好处远远不止这些，比如我们可以很容易的基于 HTTP2.0 实现长连接，而以往的选择要么是用更底层的 TCP，要么是使用与 HTTP 同级的 Web Socket 协议。现在 HTTP 协议直接支持了长连接，对开发者而言确实是一大利好。考虑到本篇文章主要是讨论 QUIC 协议相关，就不对 HTTP 2.0 做详细分析了。

茶歇小结

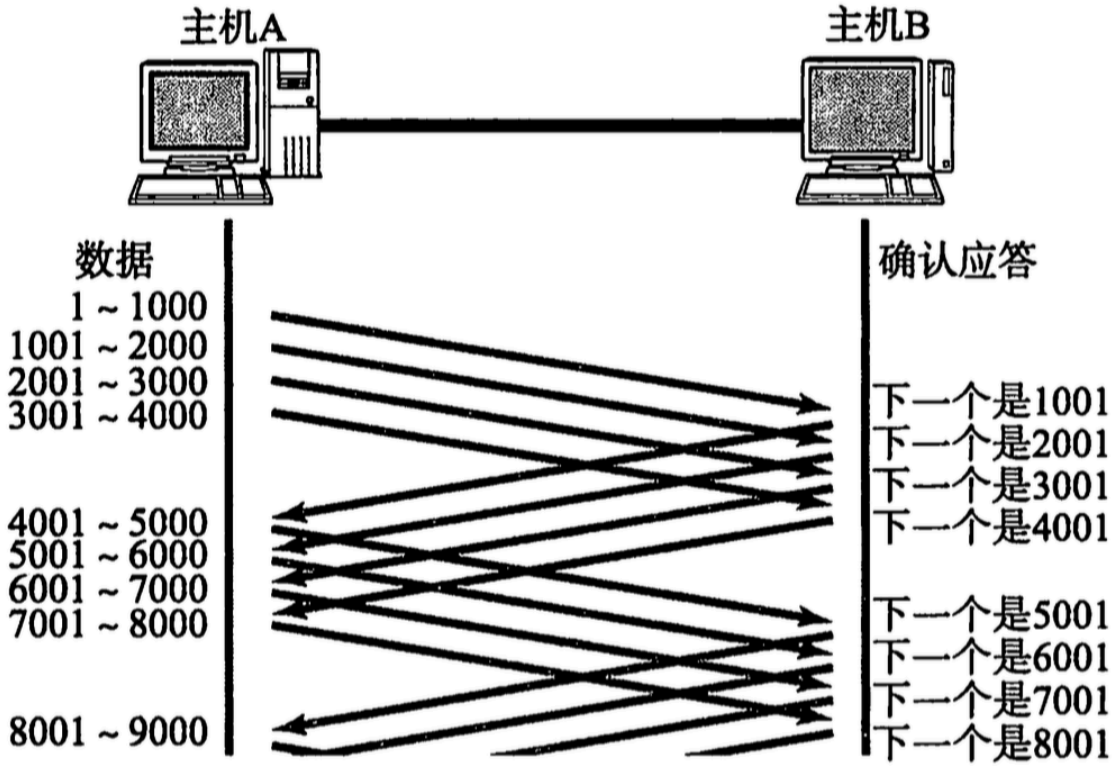
刚刚聊到了三个技术名词，Pipeline、并发请求和多路复用，千万不要被绕晕了。

Pipeline 是为了减少不必要的 TCP 连接，但依然存在队头阻塞(HOC)的缺点，一种解决思路是利用并发连接减少某一个 HOC 的影响，另一个是共享(**注意与复用的区别**) TCP 连接，直接避免 HOC 问题的发生。

TCP 窗口

是不是觉得 SPDY 的多路复用已经够厉害了，解决了队头阻塞问题？很遗憾的是，并没有，而且我可以很肯定的说，只要你还在用 TCP 链接，HOC 就是逃不掉的噩梦，不信我们来看看 TCP 的实现细节。

我们知道 TCP 协议会保证数据的可达性，如果发生了丢包或者错包，数据就会被重传。于是问题来了，如果一个包丢了，那么后面的包就得停下来等这个包重新传输，也就是发生了队头阻塞。当然 TCP 协议的设计者们也不傻，他们发明了滑动窗口的概念:



- **根据窗口为4000字节时返回的确认应答，下一步就发送比这个值还要大4000个序列号为止的数据。这跟前面每个段接收确认应答以后再发送另一个新段的情况相比，即使往返时间变长也不会影响网络的吞吐量。**

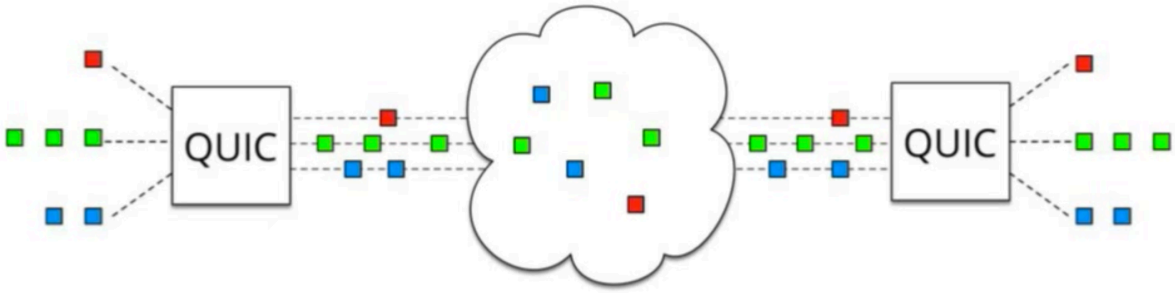
这样的好处是在第一个数据包(1-1000) 发出后，不必等到 ACK 返回就可以立刻发送第二个数据包。可以看出图中的 TCP 窗口大小是 4，所以第四个包发送后就会开始等待，直到第一个包的 ACK 返回。这样窗口可以向后滑动一位，第五个包被发送。

如果第一、二、三个的包都丢失了也没有关系，当发送方收到第四个包时，它可以确信一定是前三个 ACK 丢了而不是数据包丢了，否则不会收到 4001 的 ACK，所以发送方可以大胆的把窗口向后滑动四位。

滑动窗口的概念大幅度提高了 TCP 传输数据时抗干扰的能力，一般丢失一两个 ACK 根本没关系。但如果是发送的包丢失，或者出错，窗口就无法向前滑动，出现了队头阻塞的现象。

QUIC 多路复用与纠错

所以说 HOC 不仅仅在 HTTP 层存在，在 TCP 层也存在，这也正是 QUIC 协议要解决的问题。回顾 SPDY 是如何解决 HOC 的，没错，多路复用(Multiplex)。QUIC 协议也采用了多路复用技术。



QUIC 协议基于 UDP 实现，我们知道 UDP 协议只负责发送数据，并不保证数据可达性。这一方面为 QUIC 的多路复用提供了基础，另一方面也要求 QUIC 协议自己保证数据可达性。

SPDY 为各个数据包做好标记，指明他们属于哪个 HTTP 请求，至于这些包能不能到达客户端，SPDY 并不关心，因为数据可达性由 TCP 协议保证。既然客户端一定能收到包，那就只要排序、拼接就行了。QUIC 协议采用了多路复用度思想，但同时还得自己保证数据的可达性。

TCP 协议的丢包重传并不是一个好想法，因为一旦有了前后顺序，队头阻塞问题将不可避免。而无序的数据发送给接受者以后，如何保证不丢包，不错包呢？这看起来是个不可能完成的任务，不过如果把要求降低成:“最多丢一个包，或者错一个包”，事情就简单多了，操作系统中有一种存储方式叫 RAID 5，采用的是异或运算加上数据冗余的方式来保证前向纠错(FEC: Forward Error Correcting)。

我们知道异或运算的规则是，`0 ^ 1 = 1`、`1 ^ 1 = 0`，也就是相同数字异或成 1，不同数字异或成 0。对两个数字做异或运算，其实就是将他们转成二进制后按位做异或，因此对于任何数字 a，都有:

```
a ^ a = 0
a ^ 0 = a
```

同时很容易证明异或运算满足交换律和结合律，我们假设有下面这个等式:

```
A1 ^ A2 ^ A3 ^ ... ^ An = T
```

如果想让等式的左边只留下一个一个元素，只要在等号两边做 n-1 次异或就可以了:

```
(A1 ^ A1) ^ A2 ^ A3 ^ ... ^ An = T ^ A1
// 所以
A2 ^ A3 ^ ... ^ An = T ^ A1
// 所以
A3 ^ ... ^ An = T ^ A1 ^ A2
// 所以 ......
Ai = T ^ A1 ^ A2 ^ ... Ai-1 ^ Ai+1 ^ Ai+2 ^ ... ^ An
```

换句话说，A1 到 An 和 T 这总共 n+1 个元素中，不管是任何一个元素缺失，都可以从另外 n 个元素推导出来。如果把 A1、A2 一直到 An 想象成要发送的数据，T 想象成冗余数据，那么除了丢包重传，我们还可以采用冗余数据包的形式来保证数据准确性。

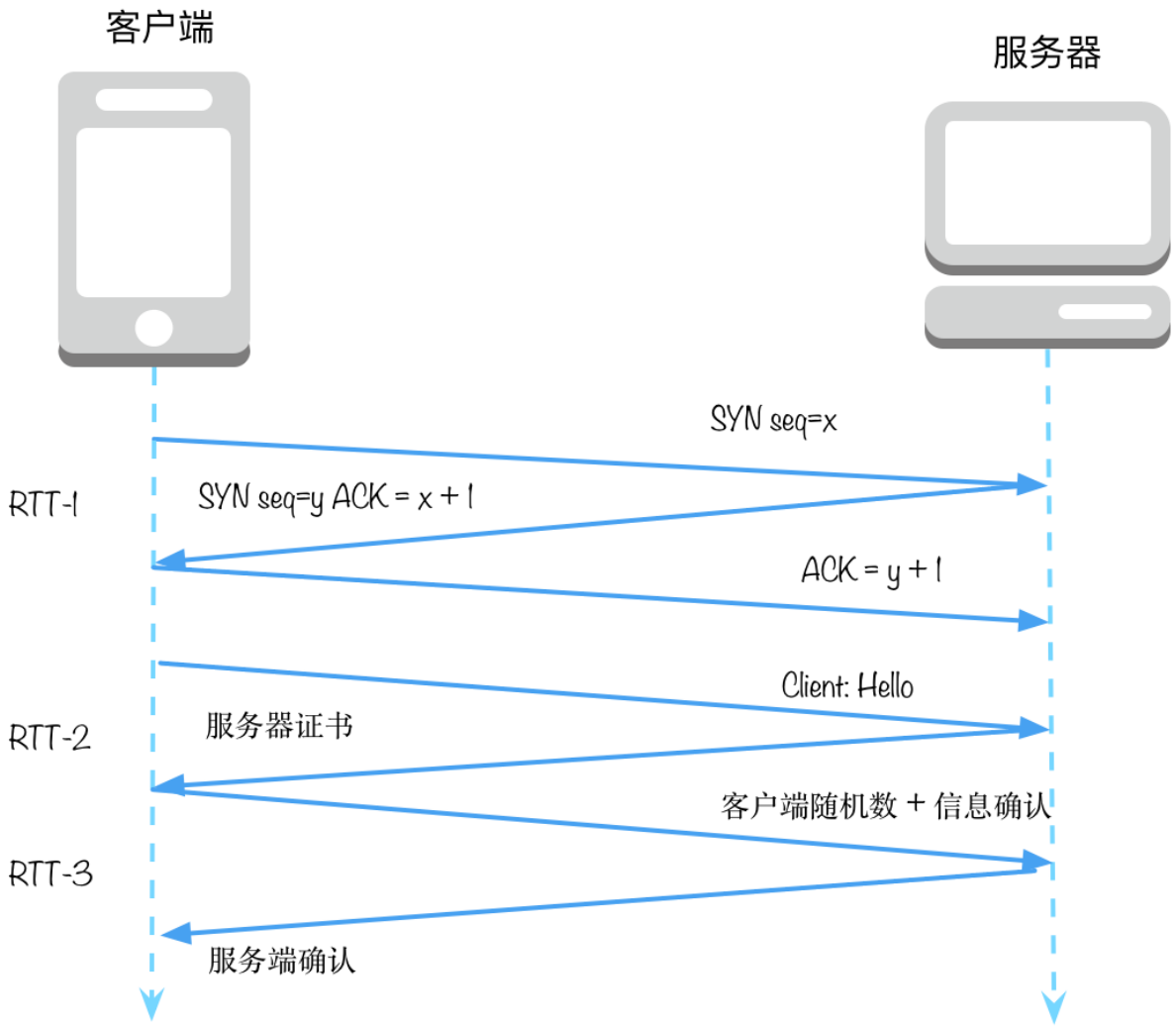
举个例子，假设有 5 个数据包要发送，我可以额外发送一个包(上面例子中的 T)，它的值是前五个包的异或结果。这样不管是前五个包中丢失了任何一个，或者某个包数据有错(可以当成丢包来处理)，都可以用另外四个包和这个冗余的包 T 进行异或运算，从而恢复出来。

当然要注意的是，这种方案仅仅在只发生一个错包或丢包时有效，如果丢失两个包就无能为力了(这也就是为什么只发一个冗余包就够的原因)。因此数据包和冗余包之间的比值需要精心设计，如果比值过高，很容易出现丢两个包的情况，如果比值过低，又会导致冗余度太高，需要设计者根据概率计算结果进行权衡。

利用冗余数据的思想，QUIC 协议基本上避免了重发数据的情况，这种利用已有数据就能进行错误恢复的技术叫做前向恢复(FEC: Fowrard Error Correcting)。当然 QUIC 协议还是支持重传的，比如某些非常重要的数据或者丢失两个包的情况。

更少的 RTT

我们考虑一次 HTTPS 请求，它的基本流程是三次 TCP 握手外加四次 SSL/TLS 握手，从图中可以看到这需要三个 RTT:



对于 HTTP 2.0 来说，本来需要一个额外的 RTT 来进行协商，判断客户端与服务器是不是都支持 HTTP 2.0，不过好在它可以和 SSL 握手的请求合并。这也就是为什么大多数主流浏览器(比如 Chrome、Firefox) 仅支持 HTTPS 2.0 而不单独支持 HTTP 2.0 的原因，毕竟 HTTP 2.0 需要一个额外的 RTT，HTTPS 2.0 需要两个额外的 RTT，仅仅是增加一个 RTT 就能获得数据安全性，还是很划算的。

SSL 握手优化

有关 HTTPS 的详细解释可以参考我之前的文章: 九个问题从入门到熟悉HTTPS，这里我们简单复习一下 SSL 握手的大致流程:

1. 客户端发送第一个握手，包含一个随机数，以及对协议的支持情况(版本、加密方法、压缩方法等)
2. 服务器返回证书，以及服务端生成随机数
3. 客户端校验证书，生成一个新的随机数，用证书中的公钥加密后发给服务端
4. 服务端确认消息，双方根据上述三个随机数生成后续会话的公钥

由于需要确认证书，生成多个随机数来保证安全，握手阶段的两个 RTT 很难节省。不过之前我们见过 HTTP 的 Pipeline 技术可以复用 TCP 连接，那么按照类似的思想，SSL 连接也可以被恢复。思考一下为什么 SSL 要设计这么复杂的握手机制，它本质上是为了保证对称秘钥的安全传输，所以 SSL 会话恢复主要考虑的也是如何恢复对称秘钥。

一个常用的方案是采用 Session Ticket，实现起来很容易: 一旦 SSL 会话建立起来，服务端把会话的基本信息，比如对称秘钥、加密方法等信息加密后发给客户端，客户端可以缓存下来这个 Session Ticket。需要恢复 SSL 会话时直接把它发回给服务端校验即可，这样可以在 SSL 层减少一个 RTT。

TCP 快速打开

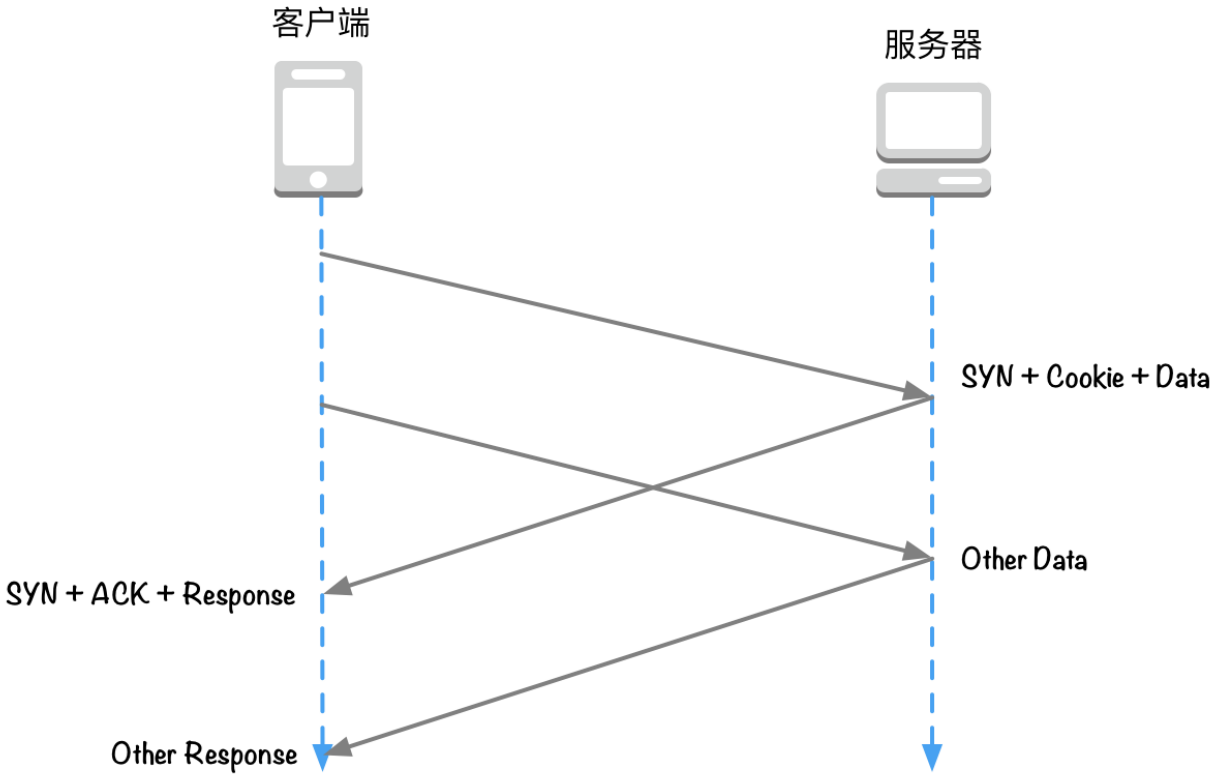
聊完了 SSL 层，下面说说 TCP 的优化方案。我们都知道 TCP 的三次握手需要花费一个 RTT，有没有可能做到 0-RTT 呢？比如我们在握手的时候就带上要传递的数据。

实际上 TCP 协议已经规定了这种情况的处理方式，即客户端可以在发送第一个 SYN 握手包时携带数据，但是 TCP 协议的实现者**绝对不允许**(原文: MUST NOT) 把这个数据包上传给应用层。这主要是为了防止 TCP 泛洪攻击。

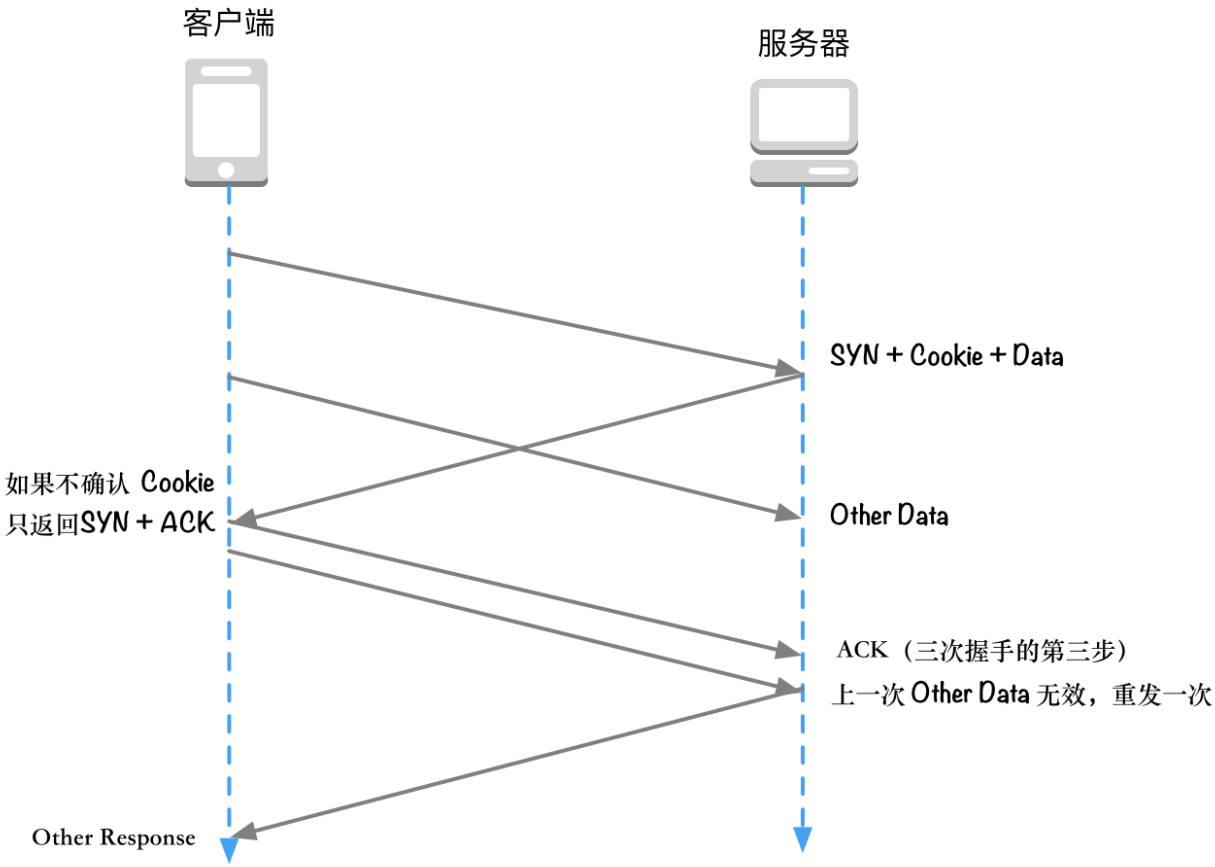
TCP 泛洪攻击是指攻击者利用多台机器发送 SYN 请求从而耗尽服务器的 backlog 队列，backlog 队列维护的是那些接受了 SYN 请求但还没有正式开始会话的连接。这样做的好处是服务器不会过早的分配端口、建立连接。RFC 4987 详细的描述了各种防止 TCP 泛洪攻击的方法，包括尽早释放 SYN，增加队列长度等等。

如果 SYN 握手的包能被传输到应用层，那么现有的防护措施都无法防御泛洪攻击，而且服务端也会因为这些攻击而耗尽内存和 CPU。所以人们设计了 TFO (TCP Fast Open)，这是对 TCP 的拓展，不仅可以在发送 SYN 时携带数据，还可以保证安全性。

TFO 设计了一个 cookie，它在第一次握手时由 server 生成，cookie 主要是用来标识客户端的身份，以及保存上次会话的配置信息。因此在后续重新建立 TCP 连接时，客户端会携带 SYN + Cookie + 请求数据，然后不等 ACK 返回就直接开始发送数据。



服务端收到 SYN 后会验证 cookie 是否有效，如果无效则会退回到三次握手的步骤，如下图所示:



同时，为了安全起见，服务端为每个端口记录了一个值 `PendingFastOpenRequests`，用来表示有多少请求利用了 TFO，如果超过预设上限就不再接受。

关于 TFO 的优化，可以总结出三点内容:

1. TFO 设计的 cookie 思想和 SSL 恢复握手时的 Session Ticket 很像，都是由服务端生成一段 cookie 交给客户端保存，从而避免后续的握手，有利于快速恢复。
2. 第一次请求绝对不会触发 TFO，因为服务器会在接收到 SYN 请求后把 cookie 和 ACK 一起返回。后续客户端如果要重新连接，才有可能使用这个 cookie 进行 TFO
3. TFO 并不考虑在 TCP 层过滤重复请求，以前也有类似的提案想要做过滤，但因为无法保证安全性而被拒绝。所以 TFO 仅仅是避免了泛洪攻击(类似于 backlog)，但客户端接收到的，和 SYN 包一起发来的数据，依然有可能重复。不过也只有可能是 SYN 数据重复，所以 TFO 并不处理这种情况，要求服务端程序自行解决。这也就是说，不仅仅要操作系统的支持，更要求应用程序(比如 MySQL) 也支持 TFO。

0-RTT

TFO 使得 TCP 协议有可能变成 0-RTT，核心思想和 Session Ticket 的概念类似: 将当前会话的上下文缓存在客户端。如果以后需要恢复对话，只需要将缓存发给服务器校验，而不必花费一个 RTT 去等待。

结合 TFO 和 Session Ticket 技术，一个本来需要花费 3 个 RTT 才能完成的请求可以被优化到一个 RTT。如果使用 QUIC 协议，我们甚至可以更进一步，将 Session Ticket 也放到 TFO 中一起发送，这样就实现了 0-RTT 的对话恢复。感兴趣的读者可以阅读: [Facebook App对TLS的魔改造：实现0-RTT]

Why QUIC

从以上分析可以发现，HTTP2 和 SSL 可以说已经进行了大量的优化，可以提升的空间非常小。而 TCP 存在诸多不足之处，一方面它设计较早，而且主要目的是设计一种通用、可靠的传输协议，并非专门为网页或者 App 而设计，另一方面对 TCP 的改进要比对 SSL 和 HTTP 的改进麻烦的多，因为 TCP 是由各个操作系统实现，就以 TFO 为例吧，它在新版本的 Linux 内核中被实现，但想等到它普及开来就不知道要到猴年马月了，有兴趣的读者可以参考参考现在 Windows XP 系统的市场占有率。

反观 HTTP 和 SSL，虽然早期 HTTP 1.0 的问题更多，但是经过 1.1、SPDY、2.0 等版本的更迭，已经非常优秀了。其中的根本原因还是在于 HTTP 和 SSL 位于应用层，优化升级比较容易实现，所以经过长年累月的优化升级，现在大部分瓶颈都集中于 TCP 层。但 TCP 不仅优化点较多，而且还不容易更新。那么能不能在传输层搞一个和 TCP、UDP 类似的协议呢？答案也是否定的，其实曾经有一个 SCTP 协议打算进行一系列优化，但并没有被广泛接受。这是因为数据在传输的过程中需要经过各个路由器，这些设备只能识别并解析 TCP 和 UDP 协议的数据包，无法解析新的协议。所以 SCTP 也只能用于内网的实验环境中。

TCP 要改进，但不方便改，新增一个协议又不被已有的设备支持，看起来唯一的方案就是使用 UDP 了。虽然 UDP 协议不保证数据可达性，但这也是 UDP 的优点所在，它天然支持 0-RTT 的通信，所以一个比较新颖激进的想法就冒出来了:

采用 UDP 作为底层协议，在 UDP 之上实现数据可达性

目前，QUIC 协议内置在 Chrome 浏览器中，每次更新只需要升级浏览器即可，在 2014 年前就已经迭代了 13 个版本。

总结

最后，作为总结，我简单回答一下文章开头的几个问题:

1. QUIC 协议有哪些优点，如何实现 0-RTT?

    1. QUIC 协议在传输层就支持多路复用，避免了队头阻塞问题。
    2. QUIC 协议基于 UDP，更自由更高效
    3. QUIC 协议借鉴了 TFO 的思想，支持会话上下文缓存，方便恢复，具备实现 0-RTT 的可能
2. 传统的 HTTP2 + SSL + TCP 协议栈有哪些缺点?
    1. SSL 的会话恢复依然需要一个 RTT，而且难以合并到 TCP 层
    2. TCP 的滑动窗口存在队头阻塞问题
    3. TCP 的重传纠错会浪费一个 RTT
3. 为什么 Google 要另起炉灶，基于 UDP 去做?
    1. TCP 由操作系统实现，很难更新
    2. UDP 非常高效，几乎没有性能负担
    3. 将 QUIC 嵌入到 Chrome 中可以方便后续的升级迭代

参考资料

图解SSL/TLS协议
SSL/TLS协议运行机制的概述
Facebook App对TLS的魔改造：实现0-RTT
视频: QUIC: next generation multiplexed transport over UDP
官方文档: QUIC Wire Layout Specification

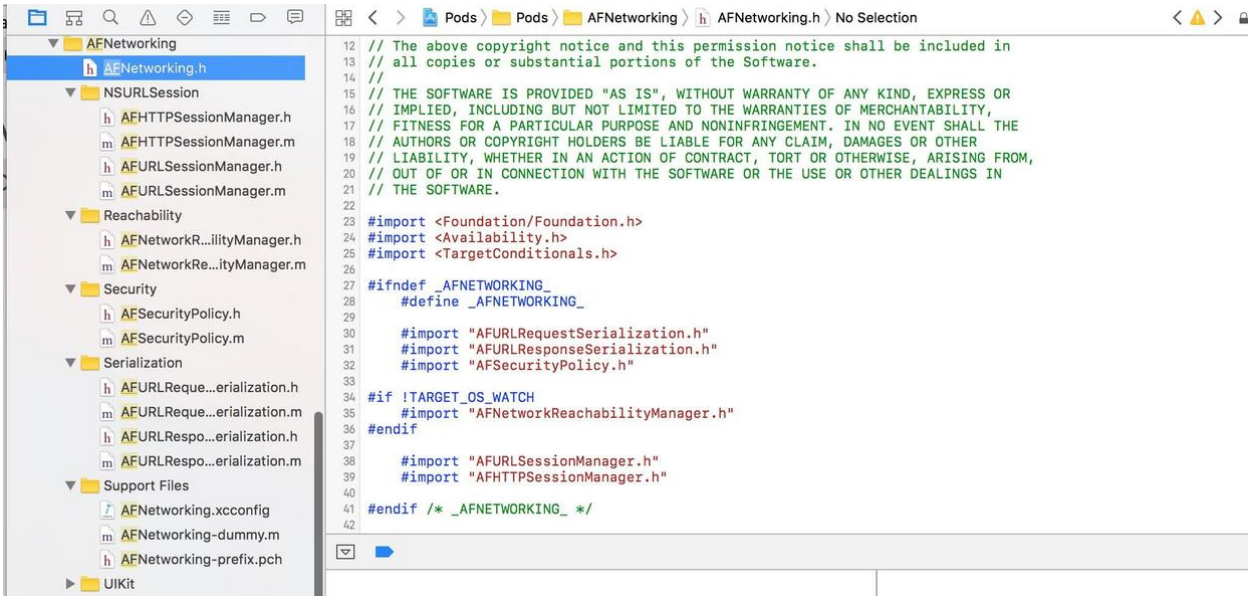
AFNetworking 源码分析

作者：@涂耀辉

- 作为一个iOS开发，也许你不知道`NSUrlRequest`、不知道`NSUrlConnection`、也不知道`NSURLSession`。。但是你一定知道`AFNetworking`。
- 大多数人习惯了只要是请求网络都用AF，但是你真的知道AF做了什么吗？为什么我们不用原生的`NSURLSession`而选择`AFNetworking`？
- 本文将从源码的角度去分析`AF`的实际作用。或许看完这篇文章，你心里会有一个答案。

一. AF3.X的网络请求主流程

首先，我们就一起分析一下该框架的组成。 将AF下载并导入工程后，下图为框架源码整个结构：

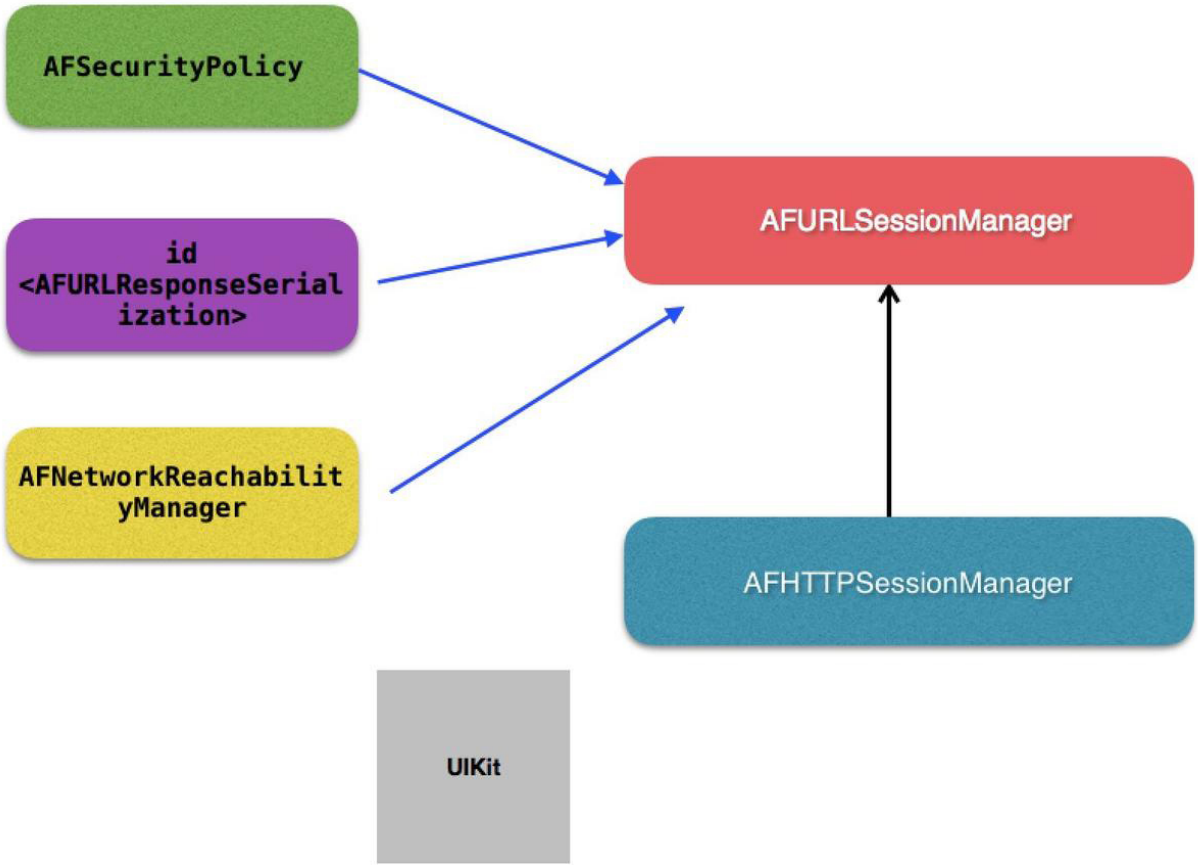


除去Support Files，可以看到AF分为如下5个功能模块：

- 网络通信模块(AFURLSessionManager、AFHTTPSessionManger)
- 网络状态监听模块(Reachability)
- 网络通信安全策略模块(Security)
- 网络通信信息序列化/反序列化模块(Serialization)
- 对于iOS UIKit库的扩展(UIKit)

其核心为网络通信模块AFURLSessionManager。

大家都知道，AF3.x是基于`NSURLSession`来封装的。所以这个类围绕着NSURLSession做了一系列的上层封装。而其余的四个模块，均是为了配合`AFURLSessionManager`类的网络通信做一些必要的处理工作。如结构关系图如下所示：



其中AFHTTPSessionManager是继承于AFURLSessionManager的，我们一般做网络请求都是用这个类，**但是它本身是没有做实事的，只是做了一些简单的封装，把请求逻辑分发给父类AFURLSessionManager去做。**

接着我们从源码的角度一个个模块去解读它的作用：

1.对外接口类：AFHTTPSessionManager

这个类提供了对外调用的接口，当我们初始化AF实例时候，都是使用这个类的这几个方法：

```
+ (instancetype)manager;
- (instancetype)initWithBaseURL:(nullable NSURL *)url;
- (instancetype)initWithBaseURL:(nullable NSURL *)url
           sessionConfiguration:(nullable NSURLSessionConfiguration
*)configuration;
```

而发起网络请求，都是调用该类以下的方法：

```
- (nullable NSURLSessionDataTask *)GET:(NSString *)URLString
                  parameters:(nullable id)parameters
                     success:(nullable void (^)(NSURLSessionDataTask *task,
id _Nullable responseObject))success
                     failure:(nullable void (^)(NSURLSessionDataTask *
_Nullable task, NSError *error))failure DEPRECATED_ATTRIBUTE;
- (nullable NSURLSessionDataTask *)POST:(NSString *)URLString
                   parameters:(nullable id)parameters
                      success:(nullable void (^)(NSURLSessionDataTask *task,
id _Nullable responseObject))success
                      failure:(nullable void (^)(NSURLSessionDataTask *
_Nullable task, NSError *error))failure DEPRECATED_ATTRIBUTE;

//还有其他的好几个方法，不一一举出了...
```

我们之前说过了这个类不做实事，只是提供对外接口，真正的初始化，以及网络请求，都是交给父类`AFURLSessionManager`去做的。

- 类似初始化方法中：

```
 - (instancetype)initWithBaseURL:(NSURL *)url
           sessionConfiguration:(NSURLSessionConfiguration *)configuration
{
    self = [super initWithSessionConfiguration:configuration];
    if (!self) {
        return nil;
    }
    //....
    return self;
}
```

- 类似网络请求方法中，调用父类方法拿到`task`，这个类仅仅是把得到的`task`，`resume`即可：

```
   dataTask = [self dataTaskWithRequest:request
                         uploadProgress:uploadProgress
                       downloadProgress:downloadProgress
                      completionHandler:^(NSURLResponse * __unused response,
id responseObject, NSError *error) {
        if (error) {
            if (failure) {
                failure(dataTask, error);
            }
        } else {
            if (success) {
                success(dataTask, responseObject);
            }
        }
    }];
```

当然这个类还做了一件很重要的事，就是把传过来的参数，编码成我们请求时需要的`request`，并且传给父类去做网络请求：

```
NSError *serializationError = nil;

//把参数，还有各种东西转化为一个request
NSMutableURLRequest *request = [self.requestSerializer
requestWithMethod:method URLString:[[NSURL URLWithString:URLString
relativeToURL:self.baseURL] absoluteString] parameters:parameters
error:&serializationError];
```

接着我们去看看这个`requestSerializer`所属类是如何将参数转换成`request`的。

2.请求参数解析类：AFHTTPRequestSerializer

下面这个方法就是这个类对外的核心方法：

```
- (NSMutableURLRequest *)requestWithMethod:(NSString *)method
                                 URLString:(NSString *)URLString
                                parameters:(id)parameters
                                     error:(NSError *__autoreleasing *)error
{
    //断言，debug模式下，如果缺少改参数，crash
    NSParameterAssert(method);
    NSParameterAssert(URLString);

    NSURL *url = [NSURL URLWithString:URLString];

    NSParameterAssert(url);

    NSMutableURLRequest *mutableRequest = [[NSMutableURLRequest alloc]
initWithURL:url];
    mutableRequest.HTTPMethod = method;

    //将request的各种属性循环遍历
    for (NSString *keyPath in AFHTTPRequestSerializerObservedKeyPaths()) {
        //如果自己观察到的发生变化的属性，在这些方法里
        if ([self.mutableObservedChangedKeyPaths containsObject:keyPath]) {
           //把给自己设置的属性给request设置
            [mutableRequest setValue:[self valueForKeyPath:keyPath]
forKey:keyPath];
        }
    }
    //将传入的parameters进行编码，并添加到request中
    mutableRequest = [[self requestBySerializingRequest:mutableRequest
withParameters:parameters error:error] mutableCopy];

        return mutableRequest;
}
```

可以根据一个Url和参数，去构造需要的`request`，这个类还把自己的一些属性(用户自定义设置)参数，加到其中去了。
其中主要部分是参数的编码，主要经过下面三步变成了一个字符串：

```
@{
 @"name" : @"bang",
 @"phone": @{@"mobile": @"xx", @"home": @"xx"},
 @"families": @[@"father", @"mother"],
 @"nums": [NSSet setWithObjects:@"1", @"2", nil]
}
->
@[
 field: @"name", value: @"bang",
 field: @"phone[mobile]", value: @"xx",
 field: @"phone[home]", value: @"xx",
 field: @"families[]", value: @"father",
 field: @"families[]", value: @"mother",
 field: @"nums", value: @"1",
 field: @"nums", value: @"2",
]
->
name=bang&phone[mobile]=xx&phone[home]=xx&families[]=father&families[]=mother
&nums=1&num=2
```

然后根据网络请求是`GET`、`HEAD`、`DELETE`、`PUT`、`POST`来决定参数字符串是应该放在`Url`后面还是`HTTP`请求体`Body`中。

至此一个完整的`NSURLRequest`就拼接完成了。

接着我们继续顺着请求的线索往下看，我们之前`AFHTTPSessionManager`类调用了父类`AFURLSessionManager`这么一个方法来生成`task`：

```
 dataTask = [self dataTaskWithRequest:request ...]
```

3.AF请求核心类：AFURLSessionManager

我们顺着上述线索来看这个方法的实现之前，先看看这个类的初始化方法，它被我们之前说的`AFHTTPSessionManager`类所继承。所以`AFHTTPSessionManager`的初始化触发了这个类的所有初始化：

```
- (instancetype)initWithSessionConfiguration:(NSURLSessionConfiguration
*)configuration {
    self = [super init];
    if (!self) {
        return nil;
    }

    if (!configuration) {
        configuration = [NSURLSessionConfiguration
defaultSessionConfiguration];
    }
```

```
    self.sessionConfiguration = configuration;

    self.operationQueue = [[NSOperationQueue alloc] init];
    //queue并发线程数为1，这个是代理回调的queue
    self.operationQueue.maxConcurrentOperationCount = 1;


    //注意代理，代理的继承，实际上NSURLSession去判断了，你实现了哪个方法会去调用，包括子代理的方法!
    self.session = [NSURLSession sessionWithConfiguration:self.sessionConfiguration delegate:self delegateQueue:self.operationQueue];


    //各种响应转码
    self.responseSerializer = [AFJSONResponseSerializer serializer];

    //ssl证书，是验证证书，还是公钥，还是不用
    self.securityPolicy = [AFSecurityPolicy defaultPolicy];

#if !TARGET_OS_WATCH
    self.reachabilityManager = [AFNetworkReachabilityManager sharedManager];
#endif


    // 设置存储NSURL task与AFURLSessionManagerTaskDelegate的词典（重点，在AFNet中，每一个task都会被匹配一个AFURLSessionManagerTaskDelegate 来做task的delegate事件处理） ===============
    self.mutableTaskDelegatesKeyedByTaskIdentifier = [[NSMutableDictionary alloc] init];
    // =============== 设置AFURLSessionManagerTaskDelegate 词典的锁，确保词典在多线程访问时的线程安全===============
    self.lock = [[NSLock alloc] init];
    self.lock.name = AFURLSessionManagerLockName;

    // =============== 为所管理的session的所有task设置完成块,此方法为生成session之后就调用
    [self.session getTasksWithCompletionHandler:^(NSArray *dataTasks, NSArray *uploadTasks, NSArray *downloadTasks) {
        //开始的时候应该什么都没有
        for (NSURLSessionDataTask *task in dataTasks) {
            [self addDelegateForDataTask:task uploadProgress:nil downloadProgress:nil completionHandler:nil];
        }

        for (NSURLSessionUploadTask *uploadTask in uploadTasks) {
            [self addDelegateForUploadTask:uploadTask progress:nil completionHandler:nil];
```

```
        }

        for (NSURLSessionDownloadTask *downloadTask in downloadTasks) {
            [self addDelegateForDownloadTask:downloadTask progress:nil
destination:nil completionHandler:nil];
        }
    }];

    return self;
}
```

这个方法初始化了一些我们后续需要用到的属性，其他的都很简单，唯一比较费解的两处可能是：

```
 self.operationQueue.maxConcurrentOperationCount = 1;
```

```
 [self.session getTasksWithCompletionHandler:^(NSArray *dataTasks, NSArray
*uploadTasks, NSArray *downloadTasks) {
       //置空处理
 }];
```

我们首先来讲讲这两个操作的作用：

- 第一是让回调的代理`queue`是串行的，即请求完成的`task`只能一个个被回调。
- 第二是清空了`session`中所有`task`。

之所以让人费解的是，这么做的意义是什么？第一个目的我们暂且不说，我们放到文章结尾再来谈。而第二个，我们知道这里是初始化方法，讲道理`session`中不会有任何`task`。但是这是因为大家只知其一，我们有一种后台`session`，当从后台回来的时候，根据一个ID，就可以重新恢复这个`session`，这时候其中就会有之前未完成的`task`了。
而这里这么做的目的就是防止一些之前的后台请求任务，导致程序的`crash`，见github:https://github.com/AFNetworking/AFNetworking/issues/3499

接着我们回到之前的`dataTaskWithRequest`返回task的方法：

```
- (NSURLSessionDataTask *)dataTaskWithRequest:(NSURLRequest *)request
                                uploadProgress:(nullable void (^)(NSProgress
*uploadProgress)) uploadProgressBlock
                              downloadProgress:(nullable void (^)(NSProgress
*downloadProgress)) downloadProgressBlock
                             completionHandler:(nullable void (^)
(NSURLResponse *response, id _Nullable responseObject,  NSError * _Nullable
error))completionHandler {


    __block NSURLSessionDataTask *dataTask = nil;
    //第一件事，创建NSURLSessionDataTask，里面适配了Ios8以下taskIdentifiers，函数创
建task对象。
    //其实现应该是因为iOS 8.0以下版本中会并发地创建多个task对象，而同步有没有做好，导致
taskIdentifiers 不唯一…这边做了一个串行处理
    url_session_manager_create_task_safely(^{
        dataTask = [self.session dataTaskWithRequest:request];
    });

    [self addDelegateForDataTask:dataTask uploadProgress:uploadProgressBlock
downloadProgress:downloadProgressBlock completionHandler:completionHandler];

    return dataTask;
}
```

当然，这个类有数个类似的方法如下：

```
-dataTaskWithRequest:completionHandler:
-dataTaskWithRequest:uploadProgress:downloadProgress:completionHandler:

-uploadTaskWithRequest:fromFile:progress:completionHandler:
-uploadTaskWithRequest:fromData:progress:completionHandler:
-uploadTaskWithStreamedRequest:progress:completionHandler:

-downloadTaskWithRequest:progress:destination:completionHandler:
-downloadTaskWithResumeData:progress:destination:completionHandler:
```

这些方法做的事基本一样，就是下面这两件：

(1). 调用`session`的方法，传`request`过去去生成`task`。注意这里调用了`url_session_manager_create_task_safely`函数去执行的`Block`,这个函数实现如下：

```
static void url_session_manager_create_task_safely(dispatch_block_t block) {
    if (NSFoundationVersionNumber <
NSFoundationVersionNumber_With_Fixed_5871104061079552_bug) {
        dispatch_sync(url_session_manager_creation_queue(), block);
    } else {
      block();
    }
}
```

简单来讲就是为了适配`iOS8`以下`task`创建，其中`taskIdentifiers`属性不唯一，而这个属性是我们之后添加代理的`key`，它必须是唯一的。
所以这里做了一个判断，如果是`iOS8`以下，则用串行同步的方式去执行这个`Block`，也就是创建`session`。否则直接执行。

(2). 给每个`task`创建并对应一个AF的代理对象，这基本上是这个类的核心所在了，这个代理对象为其对应的`task`做数据拼接及成功回调。

我们来看看这个方法：

```
- (void)addDelegateForDataTask:(NSURLSessionDataTask *)dataTask
                uploadProgress:(nullable void (^)(NSProgress
*uploadProgress)) uploadProgressBlock
              downloadProgress:(nullable void (^)(NSProgress
*downloadProgress)) downloadProgressBlock
             completionHandler:(void (^)(NSURLResponse *response, id
responseObject, NSError *error))completionHandler
{
    AFURLSessionManagerTaskDelegate *delegate =
[[AFURLSessionManagerTaskDelegate alloc] init];

    // AFURLSessionManagerTaskDelegate与AFURLSessionManager建立相互关系
    delegate.manager = self;
    delegate.completionHandler = completionHandler;

    //这个taskDescriptionForSessionTasks用来发送开始和挂起通知的时候会用到,就是用这个
值来Post通知，来两者对应
    dataTask.taskDescription = self.taskDescriptionForSessionTasks;

    // ***** 将AF delegate对象与 dataTask建立关系
    [self setDelegate:delegate forTask:dataTask];

    // 设置AF delegate的上传进度，下载进度块。
    delegate.uploadProgressBlock = uploadProgressBlock;
    delegate.downloadProgressBlock = downloadProgressBlock;
}
```

```
- (void)setDelegate:(AFURLSessionManagerTaskDelegate *)delegate
            forTask:(NSURLSessionTask *)task
{
    //断言，如果没有这个参数，debug下crash在这
    NSParameterAssert(task);
    NSParameterAssert(delegate);

    //加锁保证字典线程安全
    [self.lock lock];

    // 将AF delegate放入以taskIdentifier标记的词典中（同一个NSURLSession中的taskIdentifier是唯一的）
    self.mutableTaskDelegatesKeyedByTaskIdentifier[@(task.taskIdentifier)] = delegate;

    // 为AF delegate 设置task 的progress监听
    [delegate setupProgressForTask:task];

    //添加task开始和暂停的通知
    [self addNotificationObserverForTask:task];
    [self.lock unlock];
}
```

就这么两个方法创建了一个`AFURLSessionManagerTaskDelegate`的代理，把这个代理和`task`的`taskIdentifier`一一对应，放在我们最早初始化的字典里，建立起映射。
除此之外，我们还添加了一些task进度的监听，和task开始和挂起的通知。
当然，这些操作都在我们一开始声明的锁中进行，是线程安全的。

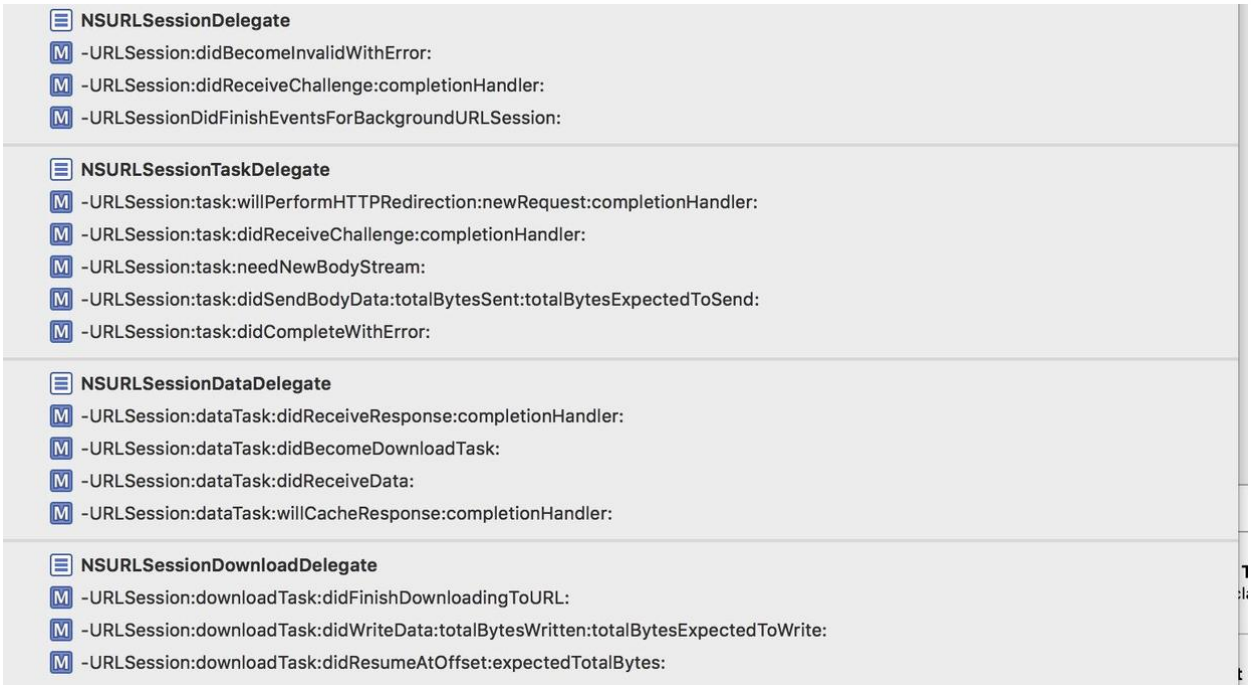
做好这些之后，我们所有的任务开启前的操作就完成了，接着我们还记得
在`AFHTTPSessionManager`中拿到了返回的task，调用了：

```
 [dataTask resume];
```

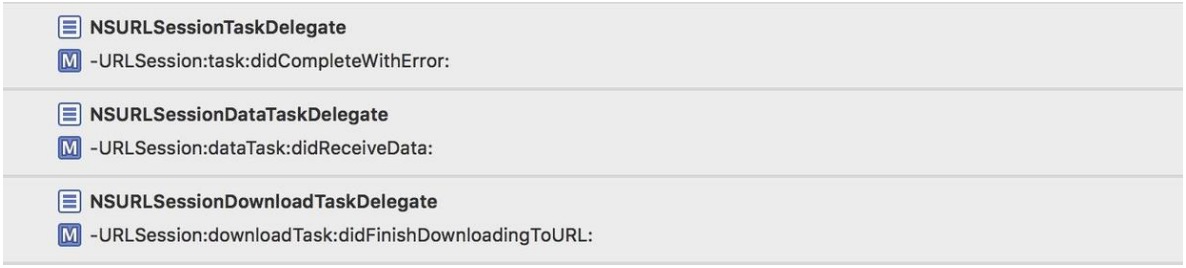
开启了任务，接着task就开始请求网络了，还记得我们初始化方法中：

```
self.session = [NSURLSession
sessionWithConfiguration:self.sessionConfiguration delegate:self
delegateQueue:self.operationQueue];
```

我们把`AFUrlSessionManager`作为了所有的`task`的`delegate`。当我们请求网络的时候，这些代理开始调用了：



- `AFUrlSessionManager`一共实现了如上图所示这么一大堆`NSUrlSession`相关的代理。（小伙伴们的顺序可能不一样，楼主根据代理隶属重新排序了一下）
- 而只转发了其中3条到AF自定义的delegate中：



这就是我们一开始说的，`AFUrlSessionManager`对这一大堆代理做了一些公共的处理，而转发到AF自定义代理的3条，则负责把每个`task`对应的数据回调出去。

至于这些代理详细作用和实现内容我就不一一细说，感兴趣的可以到作者之前这篇文章中去查看：http://www.jianshu.com/p/856f0e26279d

总结一下，这些代理主要是做了一些额外的处理，并且调用了它的属性`Block`：

```
@property (readwrite, nonatomic, copy) AFURLSessionDidBecomeInvalidBlock
sessionDidBecomeInvalid;
@property (readwrite, nonatomic, copy)
AFURLSessionDidReceiveAuthenticationChallengeBlock
sessionDidReceiveAuthenticationChallenge;
@property (readwrite, nonatomic, copy)
AFURLSessionDidFinishEventsForBackgroundURLSessionBlock
didFinishEventsForBackgroundURLSession;
@property (readwrite, nonatomic, copy)
AFURLSessionTaskWillPerformHTTPRedirectionBlock
taskWillPerformHTTPRedirection;
@property (readwrite, nonatomic, copy)
AFURLSessionTaskDidReceiveAuthenticationChallengeBlock
taskDidReceiveAuthenticationChallenge;
@property (readwrite, nonatomic, copy) AFURLSessionTaskNeedNewBodyStreamBlock
taskNeedNewBodyStream;
@property (readwrite, nonatomic, copy) AFURLSessionTaskDidSendBodyDataBlock
taskDidSendBodyData;
@property (readwrite, nonatomic, copy) AFURLSessionTaskDidCompleteBlock
taskDidComplete;
@property (readwrite, nonatomic, copy)
AFURLSessionDataTaskDidReceiveResponseBlock dataTaskDidReceiveResponse;
@property (readwrite, nonatomic, copy)
AFURLSessionDataTaskDidBecomeDownloadTaskBlock dataTaskDidBecomeDownloadTask;
@property (readwrite, nonatomic, copy)
AFURLSessionDataTaskDidReceiveDataBlock dataTaskDidReceiveData;
@property (readwrite, nonatomic, copy)
AFURLSessionDataTaskWillCacheResponseBlock dataTaskWillCacheResponse;
@property (readwrite, nonatomic, copy)
AFURLSessionDownloadTaskDidFinishDownloadingBlock
downloadTaskDidFinishDownloading;
@property (readwrite, nonatomic, copy)
AFURLSessionDownloadTaskDidWriteDataBlock downloadTaskDidWriteData;
@property (readwrite, nonatomic, copy) AFURLSessionDownloadTaskDidResumeBlock
downloadTaskDidResume;
```

我们可以利用这些`Block`，做一些自定义的处理，`Block`会随着代理调用而被调用，这些代理帮我们做了一些类似数据分片、断电续传、`https`认证等工作。

除此之外，有3个代理方法回调了我们的`task`的AF代理，包括请求完成的代理，收到数据的代理，以及下载完成的代理，以第一个为例：

```
- (void)URLSession:(NSURLSession *)session
              task:(NSURLSessionTask *)task
didCompleteWithError:(NSError *)error
{
    //根据task去取我们一开始创建绑定的delegate
    AFURLSessionManagerTaskDelegate *delegate = [self delegateForTask:task];

    // delegate may be nil when completing a task in the background
    if (delegate) {
        //把代理转发给我们绑定的delegate
        [delegate URLSession:session task:task didCompleteWithError:error];
        //转发完移除delegate
        [self removeDelegateForTask:task];
    }

    //公用Block回调
    if (self.taskDidComplete) {
        self.taskDidComplete(session, task, error);
    }
}
```

通过我们之前设置的`task`和`AF`代理映射，去调用`AF`代理，并且把这个`task`从映射字典中移除。

接着就调用了`AF`的代理：

```
//AF实现的代理！被从urlsession那转发到这

- (void)URLSession:(__unused NSURLSession *)session
              task:(NSURLSessionTask *)task
didCompleteWithError:(NSError *)error
{

#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wgnu"

    //1）强引用self.manager，防止被提前释放；因为self.manager声明为weak,类似Block

    __strong AFURLSessionManager *manager = self.manager;

    __block id responseObject = nil;

    //用来存储一些相关信息，来发送通知用的
    __block NSMutableDictionary *userInfo = [NSMutableDictionary dictionary];
    //存储responseSerializer响应解析对象
    userInfo[AFNetworkingTaskDidCompleteResponseSerializerKey] =
manager.responseSerializer;

    //Performance Improvement from #2672
```

```
    //注意这行代码的用法，感觉写的很Nice...把请求到的数据data传出去，然后就不要这个值了释放内存
    NSData *data = nil;
    if (self.mutableData) {
        data = [self.mutableData copy];
        //We no longer need the reference, so nil it out to gain back some memory.
        self.mutableData = nil;
    }

    //继续给userinfo填数据
    if (self.downloadFileURL) {
        userInfo[AFNetworkingTaskDidCompleteAssetPathKey] = self.downloadFileURL;
    } else if (data) {
        userInfo[AFNetworkingTaskDidCompleteResponseDataKey] = data;
    }
    //错误处理
    if (error) {

        userInfo[AFNetworkingTaskDidCompleteErrorKey] = error;

        //可以自己自定义完成组 和自定义完成queue,完成回调
        dispatch_group_async(manager.completionGroup ?: url_session_manager_completion_group(), manager.completionQueue ?: dispatch_get_main_queue(), ^{
            if (self.completionHandler) {
                self.completionHandler(task.response, responseObject, error);
            }
            //主线程中发送完成通知
            dispatch_async(dispatch_get_main_queue(), ^{
                [[NSNotificationCenter defaultCenter] postNotificationName:AFNetworkingTaskDidCompleteNotification object:task userInfo:userInfo];
            });
        });
    } else {
        //url_session_manager_processing_queue AF的并行队列
        dispatch_async(url_session_manager_processing_queue(), ^{
            NSError *serializationError = nil;

            //解析数据
            responseObject = [manager.responseSerializer responseObjectForResponse:task.response data:data error:&serializationError];

            //如果是下载文件，那么responseObject为下载的路径
            if (self.downloadFileURL) {
                responseObject = self.downloadFileURL;
            }
```

```
            //写入userInfo
            if (responseObject) {
                userInfo[AFNetworkingTaskDidCompleteSerializedResponseKey] =
responseObject;
            }

            //如果解析错误
            if (serializationError) {
                userInfo[AFNetworkingTaskDidCompleteErrorKey] =
serializationError;
            }
            //回调结果
            dispatch_group_async(manager.completionGroup ?:
url_session_manager_completion_group(), manager.completionQueue ?:
dispatch_get_main_queue(), ^{
                if (self.completionHandler) {
                    self.completionHandler(task.response, responseObject,
serializationError);
                }

                dispatch_async(dispatch_get_main_queue(), ^{
                    [[NSNotificationCenter defaultCenter]
postNotificationName:AFNetworkingTaskDidCompleteNotification object:task
userInfo:userInfo];
                });
            });
        });
    }
#pragma clang diagnostic pop
}
```

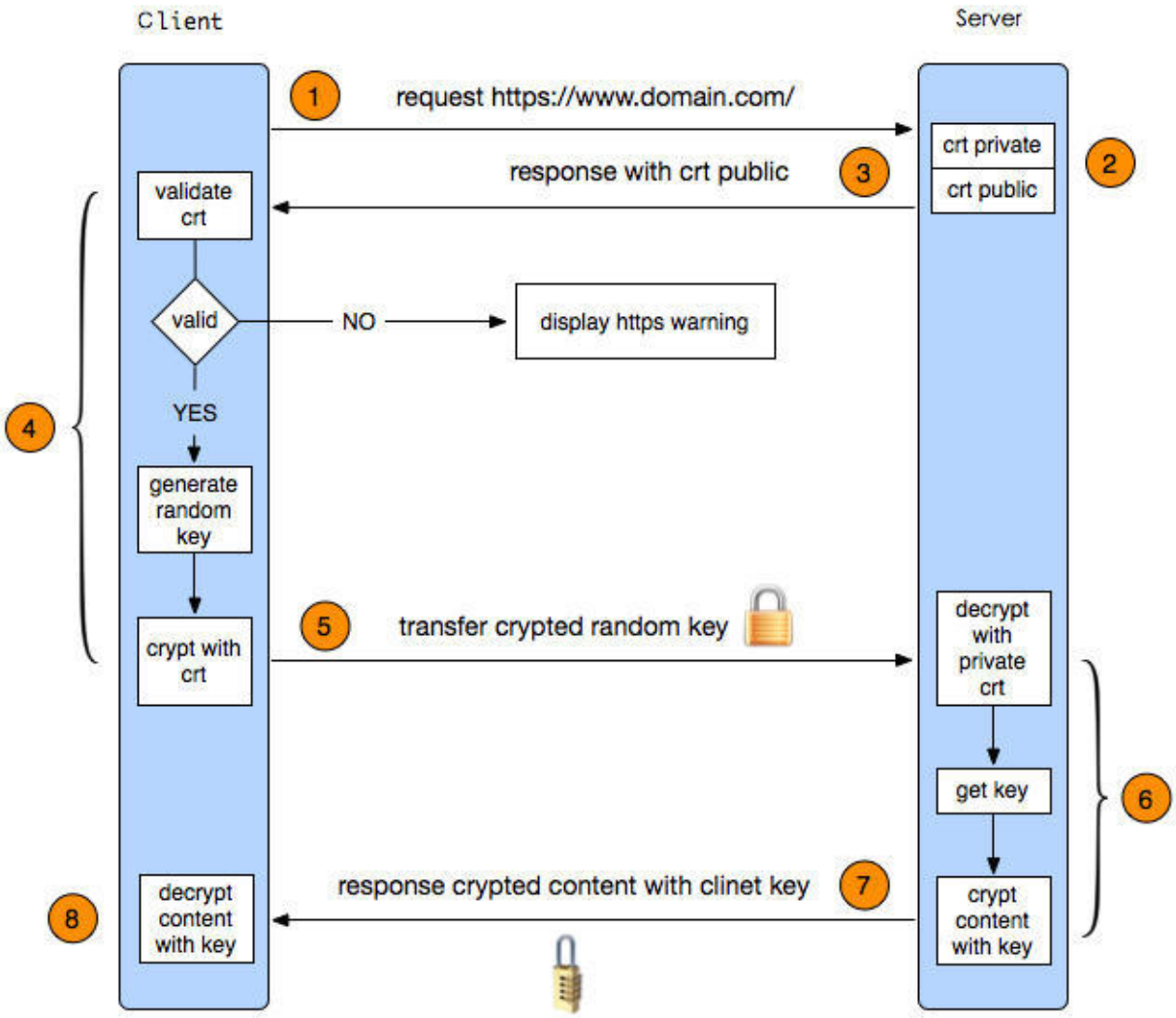
虽然这个方法有点长，但是它主要做了两件事：

1. 调用`responseSerializer`按照我们设置的格式，解析请求到的数据。
2. 用`completionHandler`把数据回调出去，至此数据回到了用户手中。

到这里，`AF`的整个主线流程就完了，当然，我们跳过了很多细节没有讲，比如`responseSerializer`的各种格式的解析过程，还有为了监听`task`的开始和挂起通知，所做的`method swizzling`，这里对`iOS7`的兼容问题的处理，算是相当精彩了。

二. AF3.X的安全策略。

AF的安全策略主要由`https`来保证，那么什么是`https`呢?
这里我们简单的理解下https：**https在http请求的基础上多加了一个证书认证的流程。**认证通过之后，数据传输都是加密进行的。
关于https的更多概念，我就不赘述了，网上有大量的文章，小伙伴们可以自行查阅。在这里大概的讲讲https的认证过程吧，如下图所示：



整个https验证的流程了。简单总结一下：

- 就是用户发起请求，服务器响应后返回一个证书，证书中包含一些基本信息和公钥。
- 用户拿到证书后，去验证这个证书是否合法，不合法，则请求终止。
- 合法则生成一个随机数，作为对称加密的密钥，用服务器返回的公钥对这个随机数加密。然后返回给服务器。
- 服务器拿到加密后的随机数，利用私钥解密，然后再用解密后的随机数（对称密钥），把需要返回的数据加密，加密完成后数据传输给用户。
- 最后用户拿到加密的数据，用一开始的那个随机数（对称密钥），进行数据解密。整个过程完成。

当然这仅仅是一个单向认证，https还会有双向认证，相对于单向认证也很简单。仅仅多了服务端验证客户端这一步，步骤也是一模一样。

我们之前将代理的时候，有关于`https`认证的代理，`AF`做了如下处理：

```
- (void)URLSession:(NSURLSession *)session
didReceiveChallenge:(NSURLAuthenticationChallenge *)challenge
 completionHandler:(void (^)(NSURLSessionAuthChallengeDisposition
disposition, NSURLCredential *credential))completionHandler
{
    //挑战处理类型为 默认
    /*
```

```
     NSURLSessionAuthChallengePerformDefaultHandling：默认方式处理
     NSURLSessionAuthChallengeUseCredential：使用指定的证书
     NSURLSessionAuthChallengeCancelAuthenticationChallenge：取消挑战
     */
    NSURLSessionAuthChallengeDisposition disposition =
NSURLSessionAuthChallengePerformDefaultHandling;
    __block NSURLCredential *credential = nil;

    // sessionDidReceiveAuthenticationChallenge是自定义方法，用来如何应对服务器端的
认证挑战

    if (self.sessionDidReceiveAuthenticationChallenge) {
        disposition = self.sessionDidReceiveAuthenticationChallenge(session,
challenge, &credential);
    } else {
         // 此处服务器要求客户端的接收认证挑战方法是
NSURLAuthenticationMethodServerTrust
        // 也就是说服务器端需要客户端返回一个根据认证挑战的保护空间提供的信任（即
challenge.protectionSpace.serverTrust）产生的挑战证书。

        // 而这个证书就需要使用credentialForTrust:来创建一个NSURLCredential对象
        if ([challenge.protectionSpace.authenticationMethod
isEqualToString:NSURLAuthenticationMethodServerTrust]) {

            // 基于客户端的安全策略来决定是否信任该服务器，不信任的话，也就没必要响应挑战
            if ([self.securityPolicy
evaluateServerTrust:challenge.protectionSpace.serverTrust
forDomain:challenge.protectionSpace.host]) {
                // 创建挑战证书（注：挑战方式为UseCredential和
PerformDefaultHandling都需要新建挑战证书）
                credential = [NSURLCredential
credentialForTrust:challenge.protectionSpace.serverTrust];
                // 确定挑战的方式
                if (credential) {
                    //证书挑战  设计policy,none，则跑到这里
                    disposition = NSURLSessionAuthChallengeUseCredential;
                } else {
                    disposition =
NSURLSessionAuthChallengePerformDefaultHandling;
                }
            } else {
                //取消挑战
                disposition =
NSURLSessionAuthChallengeCancelAuthenticationChallenge;
            }
        } else {
            //默认挑战方式
            disposition = NSURLSessionAuthChallengePerformDefaultHandling;
        }
```

```
    }
    //完成挑战
    if (completionHandler) {
        completionHandler(disposition, credential);
    }
}
```

在这里我们大概的讲讲这个方法做了什么：

- 首先指定了https为默认的认证方式。
- 判断有没有自定义Block:`sessionDidReceiveAuthenticationChallenge`，有的话，使用我们自定义Block,生成一个认证方式，并且可以给`credential`赋值，即我们需要接受认证的证书。然后直接调用`completionHandler`，去根据这两个参数，执行系统的认证。至于这个系统的认证到底做了什么，可以看文章最后，这里暂且略过。
- 如果没有自定义Block，我们判断如果服务端的认证方法要求是`NSURLAuthenticationMethodServerTrust`,**则只需要验证服务端证书是否安全**（即https的单向认证，这是AF默认处理的认证方式，其他的认证方式，只能由我们自定义Block的实现）
- 接着我们就执行了`AFSecurityPolicy`相关的一个方法，做了一个AF内部的一个https认证：

```
[self.securityPolicy
evaluateServerTrust:challenge.protectionSpace.serverTrust
forDomain:challenge.protectionSpace.host])
```

AF默认的处理是，如果这行返回NO、说明AF内部认证失败，则取消https认证，即取消请求。返回YES则进入if块，用服务器返回的一个`serverTrust`去生成了一个认证证书。（注：这个`serverTrust`是服务器传过来的，里面包含了服务器的证书信息，是用来我们本地客户端去验证该证书是否合法用的，后面会更详细的去讲这个参数）然后如果有证书，则用证书认证方式，否则还是用默认的验证方式。最后调用`completionHandler`传递认证方式和要认证的证书，去做系统根证书验证。

总结一下：这里`securityPolicy`存在的作用就是，**使得在系统底层自己去验证之前，AF可以先去验证服务端的证书。**如果通不过，则直接越过系统的验证，取消https的网络请求。否则，继续去走系统根证书的验证。

接下来我们看看`AFSecurityPolicy`内部是如果做https认证的:

如下方式，我们可以创建一个`securityPolicy`：

```
AFSecurityPolicy *policy = [AFSecurityPolicy defaultPolicy];
```

内部创建：

```
+ (instancetype)defaultPolicy {
    AFSecurityPolicy *securityPolicy = [[self alloc] init];
    securityPolicy.SSLPinningMode = AFSSLPinningModeNone;
    return securityPolicy;
}
```

默认指定了一个`SSLPinningMode`模式为`AFSSLPinningModeNone`。
对于AFSecurityPolicy，一共有4个重要的属性：

```
//https验证模式
@property (readonly, nonatomic, assign) AFSSLPinningMode SSLPinningMode;
//可以去匹配服务端证书验证的证书
@property (nonatomic, strong, nullable) NSSet <NSData *> *pinnedCertificates;
//是否支持非法的证书（例如自签名证书）
@property (nonatomic, assign) BOOL allowInvalidCertificates;
//是否去验证证书域名是否匹配
@property (nonatomic, assign) BOOL validatesDomainName;
```

它们的作用我添加在注释里了，第一条就是`AFSSLPinningMode`, 共提供了3种验证方式：

```
typedef NS_ENUM(NSUInteger, AFSSLPinningMode) {
        //不验证
    AFSSLPinningModeNone,
    //只验证公钥
    AFSSLPinningModePublicKey,
    //验证证书
    AFSSLPinningModeCertificate,
};
```

我们接着回到代理https认证的这行代码上：

```
[self.securityPolicy
evaluateServerTrust:challenge.protectionSpace.serverTrust
forDomain:challenge.protectionSpace.host]
```

- 我们传了两个参数进去，一个是`SecTrustRef`类型的serverTrust，这是什么呢？我们看到苹果的文档介绍如下：

> CFType used for performing X.509 certificate trust evaluations.

大概意思是用于执行X。509证书信任评估,再讲简单点，其实就是一个容器，装了服务器端需要验证的证书的基本信息、公钥等等，不仅如此，它还可以装一些评估策略，还有客户端的锚点证书，这个客户端的证书，可以用来和服务端的证书去匹配验证的。

- 除此之外还把服务器域名传了过去。

我们来到这个方法，代码如下：

```
//验证服务端是否值得信任
- (BOOL)evaluateServerTrust:(SecTrustRef)serverTrust
                  forDomain:(NSString *)domain
{
    //判断矛盾的条件
    //判断有域名，且允许自建证书，需要验证域名，
    //因为要验证域名，所以必须不能是后者两种：AFSSLPinningModeNone或者添加到项目里的证
```

```
书为0个。
    if (domain && self.allowInvalidCertificates && self.validatesDomainName
&& (self.SSLPinningMode == AFSSLPinningModeNone || [self.pinnedCertificates
count] == 0)) {
        return NO;
    }

    //用来装验证策略
    NSMutableArray *policies = [NSMutableArray array];
    //要验证域名
    if (self.validatesDomainName) {

        // 如果需要验证domain，那么就使用SecPolicyCreateSSL函数创建验证策略，其中第一
个参数为true表示验证整个SSL证书链，第二个参数传入domain，用于判断整个证书链上叶子节点表示
的那个domain是否和此处传入domain一致
        //添加验证策略
        [policies addObject:(__bridge_transfer id)SecPolicyCreateSSL(true,
(__bridge CFStringRef)domain)];
    } else {
        // 如果不需要验证domain，就使用默认的BasicX509验证策略
        [policies addObject:(__bridge_transfer
id)SecPolicyCreateBasicX509()];
    }

    //serverTrust：X。509服务器的证书信任。
    // 为serverTrust设置验证策略，即告诉客户端如何验证serverTrust
    SecTrustSetPolicies(serverTrust, (__bridge CFArrayRef)policies);


    //有验证策略了，可以去验证了。如果是AFSSLPinningModeNone，是自签名，直接返回可信
任，否则不是自签名的就去系统根证书里去找是否有匹配的证书。
    if (self.SSLPinningMode == AFSSLPinningModeNone) {
        //如果支持自签名，直接返回YES,不允许才去判断第二个条件，判断serverTrust是否有效
        return self.allowInvalidCertificates ||
AFServerTrustIsValid(serverTrust);
    }
    //如果验证无效AFServerTrustIsValid，而且allowInvalidCertificates不允许自签，返
回NO
    else if (!AFServerTrustIsValid(serverTrust) &&
!self.allowInvalidCertificates) {
        return NO;
    }

    //判断SSLPinningMode
    switch (self.SSLPinningMode) {
        // 理论上，上面那个部分已经解决了self.SSLPinningMode)为
AFSSLPinningModeNone)等情况，所以此处再遇到，就直接返回NO
        case AFSSLPinningModeNone:
        default:
```

```
            return NO;

        //验证证书类型
        case AFSSLPinningModeCertificate: {

            NSMutableArray *pinnedCertificates = [NSMutableArray array];

            //把证书data，用系统api转成 SecCertificateRef 类型的数
据,SecCertificateCreateWithData函数对原先的pinnedCertificates做一些处理，保证返回的
证书都是DER编码的X.509证书

            for (NSData *certificateData in self.pinnedCertificates) {
                [pinnedCertificates addObject:(__bridge_transfer
id)SecCertificateCreateWithData(NULL, (__bridge CFDataRef)certificateData)];
            }
            // 将pinnedCertificates设置成需要参与验证的Anchor Certificate（锚点证
书，通过SecTrustSetAnchorCertificates设置了参与校验锚点证书之后，假如验证的数字证书是这
个锚点证书的子节点，即验证的数字证书是由锚点证书对应CA或子CA签发的，或是该证书本身，则信任该
证书），具体就是调用SecTrustEvaluate来验证。
            //serverTrust是服务器来的验证，有需要被验证的证书。
            SecTrustSetAnchorCertificates(serverTrust, (__bridge
CFArrayRef)pinnedCertificates);

            //自签在之前是验证通过不了的，在这一步，把我们自己设置的证书加进去之后，就能验
证成功了。

            //再去调用之前的serverTrust去验证该证书是否有效，有可能：经过这个方法过滤
后，serverTrust里面的pinnedCertificates被筛选到只有信任的那一个证书
            if (!AFServerTrustIsValid(serverTrust)) {
                return NO;
            }

            // obtain the chain after being validated, which *should* contain
the pinned certificate in the last position (if it's the Root CA)
            //注意，这个方法和我们之前的锚点证书没关系了，是去从我们需要被验证的服务端证
书，去拿证书链。
            // 服务器端的证书链，注意此处返回的证书链顺序是从叶节点到根节点
            NSArray *serverCertificates =
AFCertificateTrustChainForServerTrust(serverTrust);

            //reverseObjectEnumerator逆序
            for (NSData *trustChainCertificate in [serverCertificates
reverseObjectEnumerator]) {

                //如果我们的证书中，有一个和它证书链中的证书匹配的，就返回YES
                if ([self.pinnedCertificates
containsObject:trustChainCertificate]) {
                    return YES;
                }
```

```
            }
            //没有匹配的
            return NO;
        }
            //公钥验证 AFSSLPinningModePublicKey模式同样是用证书绑定(SSL Pinning)方式验证，客户端要有服务端的证书拷贝，只是验证时只验证证书里的公钥，不验证证书的有效期等信息。只要公钥是正确的，就能保证通信不会被窃听，因为中间人没有私钥，无法解开通过公钥加密的数据。
        case AFSSLPinningModePublicKey: {

            NSUInteger trustedPublicKeyCount = 0;

            // 从serverTrust中取出服务器端传过来的所有可用的证书，并依次得到相应的公钥
            NSArray *publicKeys = AFPublicKeyTrustChainForServerTrust(serverTrust);

            //遍历服务端公钥
            for (id trustChainPublicKey in publicKeys) {
                //遍历本地公钥
                for (id pinnedPublicKey in self.pinnedPublicKeys) {
                    //判断如果相同 trustedPublicKeyCount+1
                    if (AFSecKeyIsEqualToKey((__bridge SecKeyRef)trustChainPublicKey, (__bridge SecKeyRef)pinnedPublicKey)) {
                        trustedPublicKeyCount += 1;
                    }
                }
            }
            return trustedPublicKeyCount > 0;
        }
    }

    return NO;
}
```

代码的注释很多，这一块确实比枯涩，大家可以参照着源码一起看，加深理解。

这个方法是`AFSecurityPolicy`最核心的方法，其他的都是为了配合这个方法。这个方法完成了服务端的证书的信任评估。我们总结一下这个方法做了什么（细节可以看注释）：

（1）. 根据模式，如果是`AFSSLPinningModeNone`，则肯定是返回YES，不论是自签还是公信机构的证书。

（2）. 如果是`AFSSLPinningModeCertificate`，则从`serverTrust`中去获取证书链，然后和我们一开始初始化设置的证书集合`self.pinnedCertificates`去匹配，**如果有一对能匹配成功的，就返回YES，否则NO。**
看到这可能有小伙伴要问了，什么是证书链？下面这段是我从百科上摘来的:

> 证书链由两个环节组成—信任锚（CA 证书）环节和已签名证书环节。自我签名的证书仅有一个环节的长度—信任锚环节就是已签名证书本身。

简单来说，证书链就是就是根证书，和根据根证书签名派发得到的证书。

（3）. 如果是`AFSSLPinningModePublicKey`公钥验证，则和第二步一样还是从`serverTrust`，获取证书链每一个证书的公钥，放到数组中。和我们的`self.pinnedPublicKeys`，去配对，如果有一个相同的，就返回YES，否则NO。至于这个`self.pinnedPublicKeys`,初始化的地方如下：

```
  //设置证书数组
   - (void)setPinnedCertificates:(NSSet *)pinnedCertificates {

    _pinnedCertificates = pinnedCertificates;

    //获取对应公钥集合
    if (self.pinnedCertificates) {
        //创建公钥集合
        NSMutableSet *mutablePinnedPublicKeys = [NSMutableSet
setWithCapacity:[self.pinnedCertificates count]];
        //从证书中拿到公钥。
        for (NSData *certificate in self.pinnedCertificates) {
            id publicKey = AFPublicKeyForCertificate(certificate);
            if (!publicKey) {
                continue;
            }
            [mutablePinnedPublicKeys addObject:publicKey];
        }
        self.pinnedPublicKeys = [NSSet setWithSet:mutablePinnedPublicKeys];
    } else {
        self.pinnedPublicKeys = nil;
    }
}
```

AF复写了设置证书的set方法，并同时把证书中每个公钥放在了self.pinnedPublicKeys中。

这个方法中关联了一系列的函数，我在这边按照调用顺序一一列出来（有些是系统函数，不在这里列出，会在下文集体描述作用）：

函数一：AFServerTrustIsValid

```
//判断serverTrust是否有效
static BOOL AFServerTrustIsValid(SecTrustRef serverTrust) {

    //默认无效
    BOOL isValid = NO;
    //用来装验证结果，枚举
    SecTrustResultType result;

    //__Require_noErr_Quiet 用来判断前者是0还是非0，如果0则表示没错，就跳到后面的表达式所在位置去执行，否则表示有错就继续往下执行。

    //SecTrustEvaluate系统评估证书的是否可信的函数，去系统根目录找，然后把结果赋值给result。评估结果匹配，返回0，否则出错返回非0
    //do while 0 ,只执行一次，为啥要这样写....
    __Require_noErr_Quiet(SecTrustEvaluate(serverTrust, &result), _out);

    //评估没出错走掉这，只有两种结果能设置为有效，isValid= 1
    //当result为kSecTrustResultUnspecified（此标志表示serverTrust评估成功，此证书也被暗中信任了，但是用户并没有显示地决定信任该证书）。
    //或者当result为kSecTrustResultProceed（此标志表示评估成功，和上面不同的是该评估得到了用户认可），这两者取其一就可以认为对serverTrust评估成功
    isValid = (result == kSecTrustResultUnspecified || result == kSecTrustResultProceed);

    //out函数块,如果为SecTrustEvaluate，返回非0，则评估出错，则isValid为NO
_out:
    return isValid;
}
```

- 这个方法用来验证serverTrust是否有效，其中主要是交由系统API`SecTrustEvaluate`来验证的，它验证完之后会返回一个`SecTrustResultType`枚举类型的result，然后我们根据这个result去判断是否证书是否有效。
- 其中比较有意思的是，它调用了一个系统定义的宏函数`__Require_noErr_Quiet`，函数定义如下：

```
#ifndef __Require_noErr_Quiet
        #define __Require_noErr_Quiet(errorCode, exceptionLabel)
\
          do
\
          {
\
                  if ( __builtin_expect(0 != (errorCode), 0) )
\
                  {
\
                          goto exceptionLabel;
\
                  }
\
          } while ( 0 )
#endif
```

这个函数主要作用就是，判断errorCode是否为0，不为0则，程序用`goto`跳到`exceptionLabel`位置去执行。这个`exceptionLabel`就是一个代码位置标识，类似上面的`_out`。

说它有意思的地方是在于，它用了一个do...while(0)循环，循环条件为0，也就是只执行一次循环就结束。对这么做的原因，楼主百思不得其解...看来系统原生API更是高深莫测...经冰霜大神的提醒，这么做是为了适配早期的 API ?!

函数二、三（两个函数类似，所以放在一起）：获取serverTrust证书链证书，获取serverTrust证书链公钥

```
//获取证书链
static NSArray * AFCertificateTrustChainForServerTrust(SecTrustRef
serverTrust) {
    //使用SecTrustGetCertificateCount函数获取到serverTrust中需要评估的证书链中的证书
数目，并保存到certificateCount中
    CFIndex certificateCount = SecTrustGetCertificateCount(serverTrust);
    //创建数组
    NSMutableArray *trustChain = [NSMutableArray arrayWithCapacity:
(NSUInteger)certificateCount];

    //// 使用SecTrustGetCertificateAtIndex函数获取到证书链中的每个证书，并添加到
trustChain中，最后返回trustChain
    for (CFIndex i = 0; i < certificateCount; i++) {
        SecCertificateRef certificate =
SecTrustGetCertificateAtIndex(serverTrust, i);
        [trustChain addObject:(__bridge_transfer NSData
*)SecCertificateCopyData(certificate)];
    }

    return [NSArray arrayWithArray:trustChain];
}
```

```
// 从serverTrust中取出服务器端传过来的所有可用的证书，并依次得到相应的公钥
static NSArray * AFPublicKeyTrustChainForServerTrust(SecTrustRef serverTrust)
{

    // 接下来的一小段代码和上面AFCertificateTrustChainForServerTrust函数的作用基本一
致，都是为了获取到serverTrust中证书链上的所有证书，并依次遍历，取出公钥。
    //安全策略
    SecPolicyRef policy = SecPolicyCreateBasicX509();
    CFIndex certificateCount = SecTrustGetCertificateCount(serverTrust);
    NSMutableArray *trustChain = [NSMutableArray arrayWithCapacity:
(NSUInteger)certificateCount];
    //遍历serverTrust里证书的证书链。
    for (CFIndex i = 0; i < certificateCount; i++) {
        //从证书链取证书
        SecCertificateRef certificate =
SecTrustGetCertificateAtIndex(serverTrust, i);
        //数组
        SecCertificateRef someCertificates[] = {certificate};
        //CF数组
        CFArrayRef certificates = CFArrayCreate(NULL, (const void
**)someCertificates, 1, NULL);

        SecTrustRef trust;

        // 根据给定的certificates和policy来生成一个trust对象
        //不成功跳到 _out。
```

```
        __Require_noErr_Quiet(SecTrustCreateWithCertificates(certificates,
policy, &trust), _out);

        SecTrustResultType result;

        // 使用SecTrustEvaluate来评估上面构建的trust
        //评估失败跳到 _out
        __Require_noErr_Quiet(SecTrustEvaluate(trust, &result), _out);

        // 如果该trust符合X.509证书格式，那么先使用SecTrustCopyPublicKey获取到trust
的公钥，再将此公钥添加到trustChain中
        [trustChain addObject:(__bridge_transfer
id)SecTrustCopyPublicKey(trust)];

    _out:
        //释放资源
        if (trust) {
            CFRelease(trust);
        }

        if (certificates) {
            CFRelease(certificates);
        }

        continue;
    }
    CFRelease(policy);

    // 返回对应的一组公钥
    return [NSArray arrayWithArray:trustChain];
}
```

两个方法功能类似，都是调用了一些系统的API，利用For循环，获取证书链上每一个证书或者公钥。具体内容看源码很好理解。唯一需要注意的是，这个获取的证书排序，是从证书链的叶节点，到根节点的。

函数四：判断公钥是否相同

```
//判断两个公钥是否相同
static BOOL AFSecKeyIsEqualToKey(SecKeyRef key1, SecKeyRef key2) {

#if TARGET_OS_IOS || TARGET_OS_WATCH || TARGET_OS_TV
    //iOS 判断二者地址
    return [(__bridge id)key1 isEqual:(__bridge id)key2];
#else
    return [AFSecKeyGetData(key1) isEqual:AFSecKeyGetData(key2)];
#endif
}
```

方法适配了各种运行环境，做了匹配的判断。

接下来列出验证过程中调用过得系统原生函数：

```
//1.创建一个验证SSL的策略，两个参数，第一个参数true则表示验证整个证书链
//第二个参数传入domain，用于判断整个证书链上叶子节点表示的那个domain是否和此处传入domain一致
SecPolicyCreateSSL(<#Boolean server#>, <#CFStringRef  _Nullable hostname#>)
SecPolicyCreateBasicX509();
//2.默认的BasicX509验证策略,不验证域名。
SecPolicyCreateBasicX509();
//3.为serverTrust设置验证策略，即告诉客户端如何验证serverTrust
SecTrustSetPolicies(<#SecTrustRef  _Nonnull trust#>, <#CFTypeRef  _Nonnull policies#>)
//4.验证serverTrust,并且把验证结果返回给第二参数 result
SecTrustEvaluate(<#SecTrustRef  _Nonnull trust#>, <#SecTrustResultType * _Nullable result#>)
//5.判断前者errorCode是否为0，为0则跳到exceptionLabel处执行代码
__Require_noErr(<#errorCode#>, <#exceptionLabel#>)
//6.根据证书data,去创建SecCertificateRef类型的数据。
SecCertificateCreateWithData(<#CFAllocatorRef  _Nullable allocator#>, <#CFDataRef  _Nonnull data#>)
//7.给serverTrust设置锚点证书，即如果以后再次去验证serverTrust，会从锚点证书去找是否匹配。
SecTrustSetAnchorCertificates(serverTrust, (__bridge CFArrayRef)pinnedCertificates);
//8.拿到证书链中的证书个数
CFIndex certificateCount = SecTrustGetCertificateCount(serverTrust);
//9.去取得证书链中对应下标的证书。
SecTrustGetCertificateAtIndex(serverTrust, i)
//10.根据证书获取公钥。
SecTrustCopyPublicKey(trust)
```

其功能如注释，大家可以对比着源码，去加以理解~

可能看到这，又有些小伙伴迷糊了，讲了这么多，**那如果做https请求，真正需要我们自己做的到底是什么呢?** 这里来解答一下，分为以下两种情况：

1. 如果你用的是付费的公信机构颁发的证书，标准的https，**那么无论你用的是AF还是NSUrlSession,什么都不用做，代理方法也不用实现。**你的网络请求就能正常完成。
2. 如果你用的是自签名的证书:

- 首先你需要在plist文件中，设置可以返回不安全的请求（关闭该域名的ATS）。
- 其次，如果是`NSUrlSesion`，那么需要在代理方法实现如下：

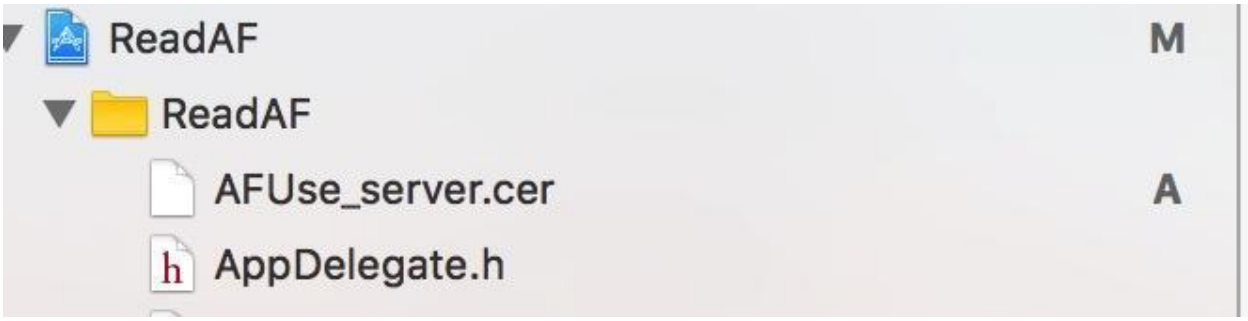
```
    - (void)URLSession:(NSURLSession *)session
didReceiveChallenge:(NSURLAuthenticationChallenge *)challenge
 completionHandler:(void (^)(NSURLSessionAuthChallengeDisposition
disposition, NSURLCredential *credential))completionHandler
 {
          __block NSURLCredential *credential = nil;

            credential = [NSURLCredential
credentialForTrust:challenge.protectionSpace.serverTrust];
            // 确定挑战的方式
            if (credential) {
             //证书挑战 则跑到这里
           disposition = NSURLSessionAuthChallengeUseCredential;
             }
            //完成挑战
         if (completionHandler) {
             completionHandler(disposition, credential);
         }
   }
```

其实上述就是AF的相对于自签证书的实现的简化版。

如果是AF，你则需要设置policy：

```
//允许自签名证书，必须的
policy.allowInvalidCertificates = YES;
//是否验证域名的CN字段
//不是必须的，但是如果写YES，则必须导入证书。
policy.validatesDomainName = NO;
```

当然还可以根据需求，你可以去验证证书或者公钥，前提是，你把自签的服务端证书，或者自签的CA根证书导入到项目中：



并且如下设置证书：

```
NSString *certFilePath = [[NSBundle mainBundle]
pathForResource:@"AFUse_server.cer" ofType:nil];
NSData *certData = [NSData dataWithContentsOfFile:certFilePath];
NSSet *certSet = [NSSet setWithObjects:certData,certData, nil];
policy.pinnedCertificates = certSet;
```

这样你就可以使用AF的不同`AFSSLPinningMode`去验证了。

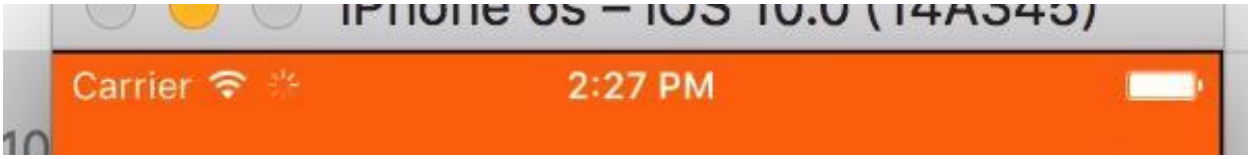
最后总结一下，AF之于https到底做了什么：

- **AF可以让你在系统验证证书之前，就去自主验证。**然后如果自己验证不正确，直接取消网络请求。否则验证通过则继续进行系统验证。
- 讲到这，顺便提一下，系统验证的流程：
    - 系统的验证，首先是去系统的根证书找，看是否有能匹配服务端的证书，如果匹配，则验证成功，返回https的安全数据。
- 如果不匹配则去判断ATS是否关闭，如果关闭，则返回https不安全连接的数据。如果开启ATS，则拒绝这个请求，请求失败。

总之一句话：**AF的验证方式不是必须的，但是对有特殊验证需求的用户确是必要的。**

三. AF3.X的UIKit扩展实现。

这个类的作用相当简单，就是当网络请求的时候，状态栏上的小菊花就会开始转:



需要的代码也很简单，只需在你需要它的位置中（比如AppDelegate）导入类，并加一行代码即可：

```
#import "AFNetworkActivityIndicatorManager.h"
```

```
[[AFNetworkActivityIndicatorManager sharedManager] setEnabled:YES];
```

接下来我们来讲讲这个类的实现：

1. 这个类的实现也非常简单，还记得我们之前讲的AF对`NSURLSessionTask`中做了一个**Method Swizzling**吗？大意是把它的`resume`和`suspend`方法做了一个替换，在原有实现的基础上添加了一个通知的发送。
2. 这个类就是基于这两个通知和task完成的通知来实现的。

首先我们来看看它的初始化方法：

```
+ (instancetype)sharedManager {
    static AFNetworkActivityIndicatorManager *_sharedManager = nil;
    static dispatch_once_t oncePredicate;
    dispatch_once(&oncePredicate, ^{
        _sharedManager = [[self alloc] init];
    });

    return _sharedManager;
}

- (instancetype)init {
    self = [super init];
    if (!self) {
        return nil;
    }
    //设置状态为没有request活跃
    self.currentState = AFNetworkActivityManagerStateNotActive;
    //开始下载通知
    [[NSNotificationCenter defaultCenter] addObserver:self
selector:@selector(networkRequestDidStart:)
name:AFNetworkingTaskDidResumeNotification object:nil];
    //挂起通知
    [[NSNotificationCenter defaultCenter] addObserver:self
selector:@selector(networkRequestDidFinish:)
name:AFNetworkingTaskDidSuspendNotification object:nil];
    //完成通知
    [[NSNotificationCenter defaultCenter] addObserver:self
selector:@selector(networkRequestDidFinish:)
name:AFNetworkingTaskDidCompleteNotification object:nil];
    //开始延迟
    self.activationDelay = kDefaultAFNetworkActivityManagerActivationDelay;
    //结束延迟
    self.completionDelay = kDefaultAFNetworkActivityManagerCompletionDelay;
    return self;
}
```

初始化如上，设置了一个state，这个state是一个枚举：

```
typedef NS_ENUM(NSInteger, AFNetworkActivityManagerState) {
    //没有请求
    AFNetworkActivityManagerStateNotActive,
    //请求延迟开始
    AFNetworkActivityManagerStateDelayingStart,
    //请求进行中
    AFNetworkActivityManagerStateActive,
    //请求延迟结束
    AFNetworkActivityManagerStateDelayingEnd
};
```

这个state一共如上4种状态，其中两种应该很好理解，而延迟开始和延迟结束怎么理解呢?

- 原来这是AF对请求菊花显示做的一个优化处理，试问如果一个请求时间很短，那么菊花很可能闪一下就结束了。如果很多请求过来，那么菊花会不停的闪啊闪，这显然并不是我们想要的效果。
- 所以多了这两个参数：

  1）在一个请求开始的时候，我延迟一会在去转菊花，如果在这延迟时间内，请求结束了，那么我就不需要去转菊花了。

  2）但是一旦转菊花开始，哪怕很短请求就结束了，我们还是会去转一个时间再去结束，这时间就是延迟结束的时间。
- 紧接着我们监听了三个通知，用来监听当前正在进行的网络请求的状态。
- 然后设置了我们前面提到的这个转菊花延迟开始和延迟结束的时间，这两个默认值如下：

```
static NSTimeInterval const kDefaultAFNetworkActivityManagerActivationDelay =
1.0;
static NSTimeInterval const kDefaultAFNetworkActivityManagerCompletionDelay =
0.17;
```

接着我们来看看三个通知触发调用的方法：

```
//请求开始
- (void)networkRequestDidStart:(NSNotification *)notification {

    if ([AFNetworkRequestFromNotification(notification) URL]) {
        //增加请求活跃数
        [self incrementActivityCount];
    }
}
//请求结束
- (void)networkRequestDidFinish:(NSNotification *)notification {
    //AFNetworkRequestFromNotification(notification)返回这个通知的request,用来判
断request是否是有效的
    if ([AFNetworkRequestFromNotification(notification) URL]) {
        //减少请求活跃数
        [self decrementActivityCount];
    }
}
```

方法很简单，就是开始的时候增加了请求活跃数，结束则减少。调用了如下两个方法进行加减：

```
//增加请求活跃数
- (void)incrementActivityCount {

    //活跃的网络数+1，并手动发送KVO
    [self willChangeValueForKey:@"activityCount"];
    @synchronized(self) {
        _activityCount++;
    }
    [self didChangeValueForKey:@"activityCount"];

    //主线程去做
    dispatch_async(dispatch_get_main_queue(), ^{
        [self updateCurrentStateForNetworkActivityChange];
    });
}

//减少请求活跃数
- (void)decrementActivityCount {
    [self willChangeValueForKey:@"activityCount"];
    @synchronized(self) {
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wgnu"
        _activityCount = MAX(_activityCount - 1, 0);
#pragma clang diagnostic pop
    }
    [self didChangeValueForKey:@"activityCount"];

    dispatch_async(dispatch_get_main_queue(), ^{
        [self updateCurrentStateForNetworkActivityChange];
    });
}
```

方法做了什么应该很容易看明白，这里需要注意的是，**task的几个状态的通知，是会在多线程的环境下发送过来的**。所以这里对活跃数的加减，都用了`@synchronized`这种方式的锁，进行了线程保护。然后回到主线程调用了`updateCurrentStateForNetworkActivityChange`

我们接着来看看这个方法：

```
- (void)updateCurrentStateForNetworkActivityChange {
    //如果是允许小菊花
    if (self.enabled) {
        switch (self.currentState) {
            //不活跃
            case AFNetworkActivityManagerStateNotActive:
                //判断活跃数，大于0为YES
                if (self.isNetworkActivityOccurring) {
                    //设置状态为延迟开始
                    [self
setCurrentState:AFNetworkActivityManagerStateDelayingStart];
                }
                break;

            case AFNetworkActivityManagerStateDelayingStart:
                //No op. Let the delay timer finish out.
                break;
            case AFNetworkActivityManagerStateActive:
                if (!self.isNetworkActivityOccurring) {
                    [self
setCurrentState:AFNetworkActivityManagerStateDelayingEnd];
                }
                break;
            case AFNetworkActivityManagerStateDelayingEnd:
                if (self.isNetworkActivityOccurring) {
                    [self
setCurrentState:AFNetworkActivityManagerStateActive];
                }
                break;
        }
    }
}
```

- 这个方法先是判断了我们一开始设置是否需要菊花的`self.enabled`，如果需要，才执行。
- 这里主要是根据当前的状态，来判断下一个状态应该是什么。其中有这么一个属性`self.isNetworkActivityOccurring`：

```
//判断是否活跃
 - (BOOL)isNetworkActivityOccurring {
    @synchronized(self) {
        return self.activityCount > 0;
    }
}
```

那么这个方法应该不难理解了。

这个类复写了currentState的set方法，每当我们改变这个state，就会触发set方法，而怎么该转菊花也在该方法中：

```
//设置当前小菊花状态
- (void)setCurrentState:(AFNetworkActivityManagerState)currentState {
    @synchronized(self) {
        if (_currentState != currentState) {
            //KVO
            [self willChangeValueForKey:@"currentState"];
            _currentState = currentState;
            switch (currentState) {
                //如果为不活跃
                case AFNetworkActivityManagerStateNotActive:
                    //取消两个延迟用的timer
                    [self cancelActivationDelayTimer];
                    [self cancelCompletionDelayTimer];
                    //设置小菊花不可见
                    [self setNetworkActivityIndicatorVisible:NO];
                    break;
                case AFNetworkActivityManagerStateDelayingStart:
                    //开启一个定时器延迟去转菊花
                    [self startActivationDelayTimer];
                    break;
                    //如果是活跃状态
                case AFNetworkActivityManagerStateActive:
                    //取消延迟完成的timer
                    [self cancelCompletionDelayTimer];
                    //开始转菊花
                    [self setNetworkActivityIndicatorVisible:YES];
                    break;
                    //延迟完成状态
                case AFNetworkActivityManagerStateDelayingEnd:
                    //开启延迟完成timer
                    [self startCompletionDelayTimer];
                    break;
            }
        }
        [self didChangeValueForKey:@"currentState"];
    }
}
```

这个set方法就是这个类最核心的方法了。它的作用如下：

- 这里根据当前状态，是否需要开始执行一个延迟开始或者延迟完成，又或者是否需要取消这两个延迟。
- 还判断了，是否需要去转状态栏的菊花，调用了`setNetworkActivityIndicatorVisible:`方法：

```
 - (void)setNetworkActivityIndicatorVisible:
(BOOL)networkActivityIndicatorVisible {
    if (_networkActivityIndicatorVisible != networkActivityIndicatorVisible)
{
        [self willChangeValueForKey:@"networkActivityIndicatorVisible"];
        @synchronized(self) {
             _networkActivityIndicatorVisible =
networkActivityIndicatorVisible;
        }
        [self didChangeValueForKey:@"networkActivityIndicatorVisible"];

        //支持自定义的Block，去自己控制小菊花
        if (self.networkActivityActionBlock) {
            self.networkActivityActionBlock(networkActivityIndicatorVisible);
        } else {
            //否则默认AF根据该Bool，去控制状态栏小菊花是否显示
            [[UIApplication sharedApplication]
setNetworkActivityIndicatorVisible:networkActivityIndicatorVisible];
        }
    }
}
```

- 这个方法就是用来控制菊花是否转。并且支持一个自定义的Block,我们可以自己去拿到这个菊花是否应该转的状态值，去做一些自定义的处理。
- 如果我们没有实现这个Block，则调用:

```
[[UIApplication sharedApplication]
setNetworkActivityIndicatorVisible:networkActivityIndicatorVisible];
```

去转菊花。

回到state的set方法中，我们除了控制菊花去转，还调用了以下4个方法：

```
//开始任务到结束的时间，默认为1秒，如果1秒就结束，那么不转菊花，延迟去开始转
- (void)startActivationDelayTimer {
    //只执行一次
    self.activationDelayTimer = [NSTimer
                                 timerWithTimeInterval:self.activationDelay
target:self selector:@selector(activationDelayTimerFired) userInfo:nil
repeats:NO];
    //添加到主线程runloop去触发
    [[NSRunLoop mainRunLoop] addTimer:self.activationDelayTimer
forMode:NSRunLoopCommonModes];
}

//完成任务到下一个任务开始，默认为0.17秒，如果0.17秒就开始下一个，那么不停  延迟去结束菊花
转
- (void)startCompletionDelayTimer {
    //先取消之前的
    [self.completionDelayTimer invalidate];
    //延迟执行让菊花不在转
    self.completionDelayTimer = [NSTimer
timerWithTimeInterval:self.completionDelay target:self
selector:@selector(completionDelayTimerFired) userInfo:nil repeats:NO];
    [[NSRunLoop mainRunLoop] addTimer:self.completionDelayTimer
forMode:NSRunLoopCommonModes];
}

- (void)cancelActivationDelayTimer {
    [self.activationDelayTimer invalidate];
}

- (void)cancelCompletionDelayTimer {
    [self.completionDelayTimer invalidate];
}
```

这4个方法分别是开始延迟执行一个方法，和结束的时候延迟执行一个方法，和对应这两个方法的取消。其作用，注释应该很容易理解。
我们继续往下看，这两个延迟调用的到底是什么：

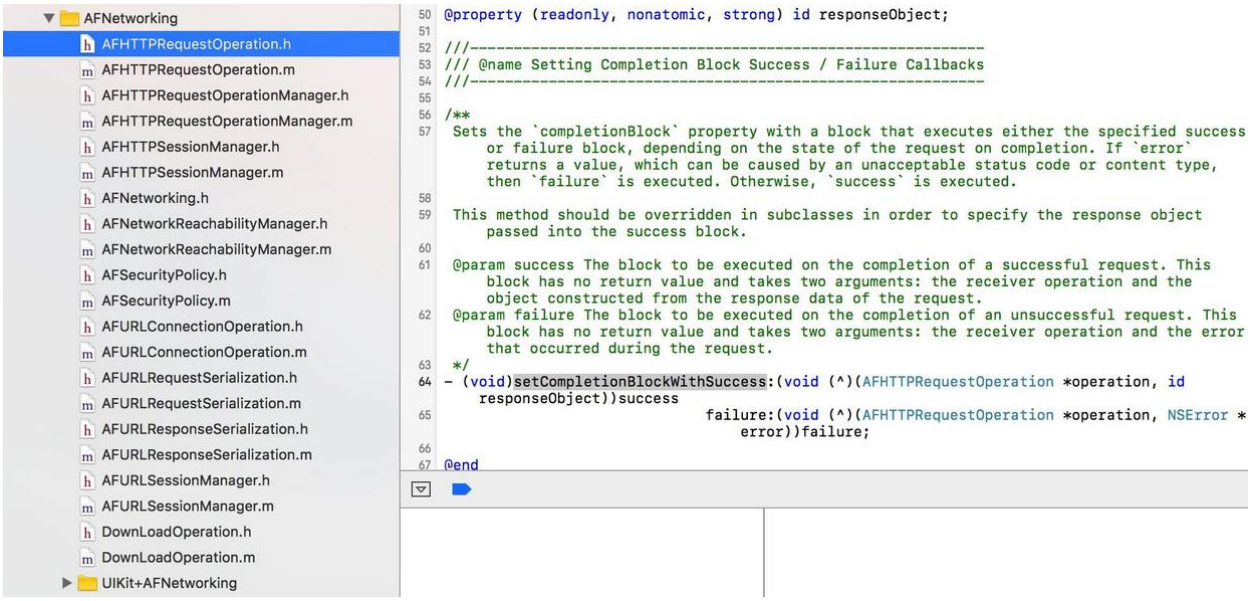
```
- (void)activationDelayTimerFired {
    //活跃状态，即活跃数大于1才转
    if (self.networkActivityOccurring) {
        [self setCurrentState:AFNetworkActivityManagerStateActive];
    } else {
        [self setCurrentState:AFNetworkActivityManagerStateNotActive];
    }
}
- (void)completionDelayTimerFired {
    [self setCurrentState:AFNetworkActivityManagerStateNotActive];
}
```

一个开始，一个完成调用，都设置了不同的currentState的值，又回到之前`state`的`set`方法中了。

至此这个`AFNetworkActivityIndicatorManager`类就讲完了，代码还是相当简单明了的。

四. `AF2.x`的网络请求主流程：

首先我们来看看AF2.x的项目目录：



除了UIKit扩展外，大概就是上述这么多类，其中最重要的有3个类：

1) `AFURLConnectionOperation`

2) `AFHTTPRequestOperation`

3) `AFHTTPRequestOperationManager`

- 大家都知道，AF2.x是基于`NSURLConnection`来封装的，而`NSURLConnection`的创建以及数据请求，就被封装在`AFURLConnectionOperation`这个类中。所以这个类基本上是AF2.x最底层也是最核心的类。
- 而`AFHTTPRequestOperation`是继承自`AFURLConnectionOperation`，对它父类一些方法做了些封装。
- `AFHTTPRequestOperationManager`则是一个管家，去管理这些这些`operation`。

我们接下来按照网络请求的流程去看看AF2.x的实现：
注：本文会涉及一些`NSOperationQueue`、`NSOperation`方面的知识，如果对这方面的内容不了解的话，可以先看看雷纯峰的这篇：
[iOS 并发编程之 Operation Queues
](http://blog.leichunfeng.com/blog/2015/07/29/ios-concurrency-programming-operation-queues/)

首先，我们来写一个get或者post请求：

```
AFHTTPRequestOperationManager *manager = [AFHTTPRequestOperationManager
manager];
[manager GET:url parameters:params
     success:^(AFHTTPRequestOperation *operation, id responseObject) {

     } failure:^(AFHTTPRequestOperation *operation, NSError *error) {

     }];
```

就这么简单的几行代码，完成了一个网络请求。

接着我们来看看`AFHTTPRequestOperationManager`的初始化方法：

```
+ (instancetype)manager {
    return [[self alloc] initWithBaseURL:nil];
}

- (instancetype)init {
    return [self initWithBaseURL:nil];
}
- (instancetype)initWithBaseURL:(NSURL *)url {
    self = [super init];
    if (!self) {
        return nil;
    }
    // Ensure terminal slash for baseURL path, so that NSURL
+URLWithString:relativeToURL: works as expected
    if ([[url path] length] > 0 && ![[url absoluteString] hasSuffix:@"/"]) {
        url = [url URLByAppendingPathComponent:@""];
    }
    self.baseURL = url;
    self.requestSerializer = [AFHTTPRequestSerializer serializer];
    self.responseSerializer = [AFJSONResponseSerializer serializer];
    self.securityPolicy = [AFSecurityPolicy defaultPolicy];
    self.reachabilityManager = [AFNetworkReachabilityManager sharedManager];
    //用来调度所有请求的queue
    self.operationQueue = [[NSOperationQueue alloc] init];
    //是否做证书验证
    self.shouldUseCredentialStorage = YES;
    return self;
}
```

初始化方法很简单，基本和AF3.x类似，除了一下两点：

(1) 设置了一个`operationQueue`，这个队列，用来调度里面所有的`operation`，在AF2.x中，每一个`operation`就是一个网络请求。

(2) 设置`shouldUseCredentialStorage`为YES，这个后面会传给`operation`，`operation`会根据这个值，去返回给代理，系统是否做https的证书验证。

然后我们来看看get方法：

```
- (AFHTTPRequestOperation *)GET:(NSString *)URLString
                     parameters:(id)parameters
                        success:(void (^)(AFHTTPRequestOperation *operation,
id responseObject))success
                        failure:(void (^)(AFHTTPRequestOperation *operation,
NSError *error))failure
{
    //拿到request
    NSMutableURLRequest *request = [self.requestSerializer
requestWithMethod:@"GET" URLString:[[NSURL URLWithString:URLString
relativeToURL:self.baseURL] absoluteString] parameters:parameters error:nil];

    AFHTTPRequestOperation *operation = [self
HTTPRequestOperationWithRequest:request success:success failure:failure];

    [self.operationQueue addOperation:operation];
    return operation;
}
```

方法很简单，如下：

(1）用`self.requestSerializer`生成了一个request，至于如何生成，可以参考之前的文章，这里就不赘述了。

(2）生成了一个`AFHTTPRequestOperation`，然后把这个`operation`加到我们一开始创建的`queue`中。

其中创建`AFHTTPRequestOperation`方法如下：

```
- (AFHTTPRequestOperation *)HTTPRequestOperationWithRequest:(NSURLRequest
*)request
                                                    success:(void (^)
(AFHTTPRequestOperation *operation, id responseObject))success
                                                    failure:(void (^)
(AFHTTPRequestOperation *operation, NSError *error))failure
{
    //创建自定义的AFHTTPRequestOperation
    AFHTTPRequestOperation *operation = [[AFHTTPRequestOperation alloc]
initWithRequest:request];
    operation.responseSerializer = self.responseSerializer;
    operation.shouldUseCredentialStorage = self.shouldUseCredentialStorage;
    operation.credential = self.credential;
    //设置自定义的安全策略
    operation.securityPolicy = self.securityPolicy;

    [operation setCompletionBlockWithSuccess:success failure:failure];
    operation.completionQueue = self.completionQueue;
    operation.completionGroup = self.completionGroup;
    return operation;
}
```

方法创建了一个`AFHTTPRequestOperation`，并把自己的一些参数交给了这个`operation`处理。

接着往里看：

```
- (instancetype)initWithRequest:(NSURLRequest *)urlRequest {
    self = [super initWithRequest:urlRequest];
    if (!self) {
        return nil;
    }

    self.responseSerializer = [AFHTTPResponseSerializer serializer];
    return self;
}
```

除了设置了一个`self.responseSerializer`，实际上是调用了父类，也是我们最核心的类`AFURLConnectionOperation`的初始化方法，首先我们要明确**这个类是继承自NSOperation的**，然后我们接着往下看：

```
//初始化
- (instancetype)initWithRequest:(NSURLRequest *)urlRequest {
    NSParameterAssert(urlRequest);

    self = [super init];
    if (!self) {
        return nil;
    }

    //设置为ready
    _state = AFOperationReadyState;
    //递归锁
    self.lock = [[NSRecursiveLock alloc] init];
    self.lock.name = kAFNetworkingLockName;
    self.runLoopModes = [NSSet setWithObject:NSRunLoopCommonModes];
    self.request = urlRequest;

    //是否应该咨询证书存储连接
    self.shouldUseCredentialStorage = YES;

    //https认证策略
    self.securityPolicy = [AFSecurityPolicy defaultPolicy];

    return self;
}
```

初始化方法中，初始化了一些属性，下面我们来简单的介绍一下这些属性：

1. `_state`设置为`AFOperationReadyState`准备就绪状态

这是个枚举：

```
typedef NS_ENUM(NSInteger, AFOperationState) {
    AFOperationPausedState      = -1,  //停止
    AFOperationReadyState       = 1,   //准备就绪
    AFOperationExecutingState   = 2,  //正在进行中
    AFOperationFinishedState    = 3,  //完成
};
```

这个`_state`标志着这个网络请求的状态，一共如上4种状态。这些状态其实对应着`operation`如下的状态：

```
//映射这个operation的各个状态
static inline NSString * AFKeyPathFromOperationState(AFOperationState state)
{
    switch (state) {
        case AFOperationReadyState:
            return @"isReady";
        case AFOperationExecutingState:
            return @"isExecuting";
        case AFOperationFinishedState:
            return @"isFinished";
        case AFOperationPausedState:
            return @"isPaused";
        default: {
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wunreachable-code"
            return @"state";
#pragma clang diagnostic pop
        }
    }
}
```

并且还复写了这些属性的get方法，用来和自定义的state一一对应：

```
//复写这些方法，与自己的定义的state对应
 - (BOOL)isReady {
    return self.state == AFOperationReadyState && [super isReady];
}
 - (BOOL)isExecuting {
    return self.state == AFOperationExecutingState;
}
 - (BOOL)isFinished {
    return self.state == AFOperationFinishedState;
}
```

2. `self.lock`锁

这个锁是用来提供给本类一些数据操作的线程安全，至于为什么要用递归锁，是因为有些方法可能会存在递归调用的情况，例如有些需要锁的方法可能会在一个大的操作环中，形成递归。**而AF使用了递归锁，避免了这种情况下死锁的发生**。

3. 初始化了`self.runLoopModes`，默认为`NSRunLoopCommonModes`。

4. 生成了一个默认的 `self.securityPolicy`。

这个类为了自定义`operation`的各种状态，而且更好的掌控它的生命周期，复写了`operation`的`start`方法，当这个`operation`在一个新线程被调度执行的时候，首先就调入这个`start`方法中，接下来我们它的实现看看：

```
- (void)start {
    [self.lock lock];

    //如果被取消了就调用取消的方法
    if ([self isCancelled]) {
        //在AF常驻线程中去执行
        [self performSelector:@selector(cancelConnection) onThread:[[self
class] networkRequestThread] withObject:nil waitUntilDone:NO modes:
[self.runLoopModes allObjects]];
    }
    //准备好了，才开始
    else if ([self isReady]) {
        //改变状态，开始执行
        self.state = AFOperationExecutingState;
        [self performSelector:@selector(operationDidStart) onThread:[[self
class] networkRequestThread] withObject:nil waitUntilDone:NO modes:
[self.runLoopModes allObjects]];
    }
    //注意，发起请求和取消请求都是在同一个线程！！包括回调都是在一个线程

    [self.lock unlock];
}
```

这个方法判断了当前的状态，是取消还是准备就绪，然后去调用了各自对应的方法。

- 注意这些方法都是在另外一个线程中去调用的，我们来看看这个线程：

```
 + (void)networkRequestThreadEntryPoint:(id)__unused object {
    @autoreleasepool {
        [[NSThread currentThread] setName:@"AFNetworking"];

        NSRunLoop *runLoop = [NSRunLoop currentRunLoop];
        //添加端口，防止runloop直接退出
        [runLoop addPort:[NSMachPort port] forMode:NSDefaultRunLoopMode];
        [runLoop run];
    }
}
 + (NSThread *)networkRequestThread {
    static NSThread *_networkRequestThread = nil;
    static dispatch_once_t oncePredicate;
    dispatch_once(&oncePredicate, ^{
        _networkRequestThread = [[NSThread alloc] initWithTarget:self
selector:@selector(networkRequestThreadEntryPoint:) object:nil];
        [_networkRequestThread start];
    });

    return _networkRequestThread;
}
```

这两个方法基本上是被许多人举例用过无数次了...

- 这是一个单例，用`NSThread`创建了一个线程，并且为这个线程添加了一个`runloop`，并且加了一个`NSMachPort`，来防止`runloop`直接退出。
- **这条线程就是AF用来发起网络请求，并且接受网络请求回调的线程，仅仅就这一条线程**（到最后我们来讲为什么要这么做）。和我们之前讲的AF3.x发起请求，并且接受请求回调时的处理方式，遥相呼应。

我们接着来看如果准备就绪，start调用的方法：

```
//改变状态，开始执行
self.state = AFOperationExecutingState;
[self performSelector:@selector(operationDidStart) onThread:[[self class]
networkRequestThread] withObject:nil waitUntilDone:NO modes:
[self.runLoopModes allObjects]];
```

接着在常驻线程中,并且不阻塞的方式，在我们`self.runLoopModes`的模式下调用：

```
- (void)operationDidStart {
    [self.lock lock];
    //如果没取消
    if (![self isCancelled]) {
        //设置为startImmediately YES 请求发出，回调会加入到主线程的 Runloop 下，
RunloopMode 会默认为 NSDefaultRunLoopMode
        self.connection = [[NSURLConnection alloc]
initWithRequest:self.request delegate:self startImmediately:NO];

        NSRunLoop *runLoop = [NSRunLoop currentRunLoop];
        for (NSString *runLoopMode in self.runLoopModes) {
            //把connection和outputStream注册到当前线程runloop中去，只有这样，才能在
这个线程中回调
            [self.connection scheduleInRunLoop:runLoop forMode:runLoopMode];
            [self.outputStream scheduleInRunLoop:runLoop
forMode:runLoopMode];
        }
        //打开输出流
        [self.outputStream open];
        //开启请求
        [self.connection start];
    }
    [self.lock unlock];
    dispatch_async(dispatch_get_main_queue(), ^{
        [[NSNotificationCenter defaultCenter]
postNotificationName:AFNetworkingOperationDidStartNotification object:self];
    });
}
```

这个方法做了以下几件事：

(1). 首先这个方法创建了一个`NSURLConnection`，设置代理为自己，startImmediately为NO，至于这个参数干什么用的，我们来看看官方文档：

> startImmediately
> YES if the connection should begin loading data immediately, otherwise NO. If you pass NO, the connection is not scheduled with a run loop. You can then schedule the connection in the run loop and mode of your choice by calling scheduleInRunLoop:forMode: .

大意是，这个值默认为YES，而且任务完成的结果会在主线程的runloop中回调。如果我们设置为NO，则需要调用我们下面看到的：

```
[self.connection scheduleInRunLoop:runLoop forMode:runLoopMode];
```

去注册一个runloop和mode，它会在我们指定的这个runloop所在的线程中回调结果。

(2). 值得一提的是这里调用了：

```
[self.outputStream scheduleInRunLoop:runLoop forMode:runLoopMode];
```

这个`outputStream`在get方法中被初始化了：

```
 - (NSOutputStream *)outputStream {
    if (!_outputStream) {
        //一个写入到内存中的流，可以通过NSStreamDataWrittenToMemoryStreamKey拿到写入后的数据
        self.outputStream = [NSOutputStream outputStreamToMemory];
    }
    return _outputStream;
}
```

这里数据请求和拼接并没有用`NSMutableData`，而是用了`outputStream`，而且把写入的数据，放到内存中。

- 其实讲道理来说`outputStream`的优势在于下载大文件的时候，可以以流的形式，将文件直接保存到本地，**这样可以为我们节省很多的内存**，调用如下方法设置：

```
[NSOutputStream outputStreamToFileAtPath:@"filePath" append:YES];
```

- 但是这里是把流写入内存中，这样其实这个节省内存的意义已经不存在了。那为什么还要用呢？这里我猜测的是就是为了用它这个可以注册在某一个`runloop`的指定`mode`下。 虽然AF使用这个`outputStream`是肯定在这个常驻线程中的，不会有线程安全的问题。但是要注意它是被声明在.h中的：

```
@property (nonatomic, strong) NSOutputStream *outputStream;
```

难保外部不会在其他线程对这个数据做什么操作，所以它相对于`NSMutableData`作用就体现出来了，就算我们在外部其它线程中去操作它，也不会有线程安全的问题。

(3). 这个`connection`开始执行了。

(4). 到主线程发送一个任务开始执行的通知。

接下来网络请求开始执行了，就开始触发`connection`的代理方法了：

```
NSURLConnectionDelegate
-connection:willSendRequestForAuthenticationChallenge:
-connectionShouldUseCredentialStorage:
-connection:willSendRequest:redirectResponse:
-connection:didSendBodyData:totalBytesWritten:totalBytesExpectedToWrite:
-connection:didReceiveResponse:
-connection:didReceiveData:
-connectionDidFinishLoading:
-connection:didFailWithError:
-connection:willCacheResponse:
```

AF2.x一共实现了如上这么多代理方法，这些代理方法，作用大部分和我们之前讲的`NSURLSession`的代理方法类似，我们只挑几个去讲，如果需要了解其他的方法作用，可以参考楼主之前的文章。

重点讲下面这四个代理：

注意，有一点需要说明，我们之前是把connection注册在我们常驻线程的runloop中了，**所以以下所有的代理方法，都是在这仅有的一条常驻线程中回调。**

第一个代理

```
//收到响应，响应头类似相关数据
- (void)connection:(NSURLConnection __unused *)connection
didReceiveResponse:(NSURLResponse *)response
{
    self.response = response;
}
```

没什么好说的，就是收到响应后，把response赋给自己的属性。

第二个代理

```
//拼接获取到的数据
- (void)connection:(NSURLConnection __unused *)connection
    didReceiveData:(NSData *)data
{
    NSUInteger length = [data length];
    while (YES) {
        NSInteger totalNumberOfBytesWritten = 0;
        //如果outputStream 还有空余空间
        if ([self.outputStream hasSpaceAvailable]) {

            //创建一个buffer流缓冲区，大小为data的字节数
            const uint8_t *dataBuffer = (uint8_t *)[data bytes];

            NSInteger numberOfBytesWritten = 0;

            //当写的长度小于数据的长度，在循环里
            while (totalNumberOfBytesWritten < (NSInteger)length) {
                //往outputStream写数据，系统的方法，一次就写一部分，得循环写
                numberOfBytesWritten = [self.outputStream
write:&dataBuffer[(NSUInteger)totalNumberOfBytesWritten] maxLength:(length -
(NSUInteger)totalNumberOfBytesWritten)];
                //如果 numberOfBytesWritten写入失败了。跳出循环
                if (numberOfBytesWritten == -1) {
                    break;
                }
                //加上每次写的长度
                totalNumberOfBytesWritten += numberOfBytesWritten;
            }
```

```
            break;
        }

        //出错
        if (self.outputStream.streamError) {
            //取消connection
            [self.connection cancel];
            //调用失败的方法
            [self performSelector:@selector(connection:didFailWithError:)
withObject:self.connection withObject:self.outputStream.streamError];
            return;
        }
    }

    //主线程回调下载数据大小
    dispatch_async(dispatch_get_main_queue(), ^{
        self.totalBytesRead += (long long)length;

        if (self.downloadProgress) {
            self.downloadProgress(length, self.totalBytesRead,
self.response.expectedContentLength);
        }
    });
}
```

这个方法看起来长，其实容易理解而且简单，它只做了3件事：

1. 给`outputStream`拼接数据，具体如果拼接，大家可以读注释自行理解下。
2. 如果出错则调用：`connection:didFailWithError:`也就是网络请求失败的代理，我们一会下面就会讲。
3. 在主线程中回调下载进度。

第三个代理

```
//完成了调用
- (void)connectionDidFinishLoading:(NSURLConnection __unused *)connection {

    //从outputStream中拿到数据 NSStreamDataWrittenToMemoryStreamKey写入到内存中的
流
    self.responseData = [self.outputStream
propertyForKey:NSStreamDataWrittenToMemoryStreamKey];

    //关闭outputStream
    [self.outputStream close];

    //如果响应数据已经有了，则outputStream置为nil
    if (self.responseData) {
       self.outputStream = nil;
    }
    //清空connection
    self.connection = nil;
    [self finish];
}
```

这个代理是任务完成之后调用。我们从`outputStream`拿到了最后下载数据，然后关闭置空了`outputStream`。并且清空了`connection`。调用了`finish`：

```
 - (void)finish {
    [self.lock lock];
    //修改状态
    self.state = AFOperationFinishedState;
    [self.lock unlock];

    //发送完成的通知
    dispatch_async(dispatch_get_main_queue(), ^{
        [[NSNotificationCenter defaultCenter]
postNotificationName:AFNetworkingOperationDidFinishNotification object:self];
    });
}
```

把当前任务状态改为已完成，并且到主线程发送任务完成的通知。**这里我们设置状态为已完成。其实调用了我们本类复写的set的方法**（前面遗漏了，在这里补充）：

```
 - (void)setState:(AFOperationState)state {

    //判断从当前状态到另一个状态是不是合理，在加上现在是否取消。。大神的框架就是屌啊，这判断严谨的。。一层层
    if (!AFStateTransitionIsValid(self.state, state, [self isCancelled])) {
        return;
    }

    [self.lock lock];

    //拿到对应的父类管理当前线程周期的key
    NSString *oldStateKey = AFKeyPathFromOperationState(self.state);
    NSString *newStateKey = AFKeyPathFromOperationState(state);

    //发出KVO
    [self willChangeValueForKey:newStateKey];
    [self willChangeValueForKey:oldStateKey];
    _state = state;
    [self didChangeValueForKey:oldStateKey];
    [self didChangeValueForKey:newStateKey];
    [self.lock unlock];
}
```

这个方法改变`state`的时候，并且发送了`KVO`。大家了解`NSOperationQueue`就知道，如果对应的operation的属性`finnished`被设置为YES，则代表当前`operation`结束了，会把`operation`从队列中移除，并且调用`operation`的`completionBlock`。**这点很重要，因为我们请求到的数据就是从这个`completionBlock`中传递回去的**（下面接着讲这个完成Block，就能从这里对接上了）。

第四个代理

```
//请求失败的回调，在cancel connection的时候，自己也主动调用了
- (void)connection:(NSURLConnection __unused *)connection
  didFailWithError:(NSError *)error
{
    //拿到error
    self.error = error;
    //关闭outputStream
    [self.outputStream close];
    //如果响应数据已经有了，则outputStream置为nil
    if (self.responseData) {
        self.outputStream = nil;
    }
    self.connection = nil;
    [self finish];
}
```

唯一需要说一下的就是这里给`self.error`赋值，之后完成Block会根据这个error，去判断这次请求是成功还是失败。

至此我们把`AFURLConnectionOperation`的业务主线讲完了。

我们此时数据请求完了，数据在`self.responseData`中，接下来我们来看它是怎么回到我们手里的。
我们回到`AFURLConnectionOperation`子类`AFHTTPRequestOperation`，有这么一个方法：

```
- (void)setCompletionBlockWithSuccess:(void (^)(AFHTTPRequestOperation
*operation, id responseObject))success
                              failure:(void (^)(AFHTTPRequestOperation
*operation, NSError *error))failure
{
    // completionBlock is manually nilled out in AFURLConnectionOperation to
break the retain cycle.
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Warc-retain-cycles"
#pragma clang diagnostic ignored "-Wgnu"
    self.completionBlock = ^{
        if (self.completionGroup) {
            dispatch_group_enter(self.completionGroup);
        }

        dispatch_async(http_request_operation_processing_queue(), ^{
            if (self.error) {
                if (failure) {
                    dispatch_group_async(self.completionGroup ?:
http_request_operation_completion_group(), self.completionQueue ?:
dispatch_get_main_queue(), ^{
                        failure(self, self.error);
                    });
                }
            } else {
                id responseObject = self.responseObject;
                if (self.error) {
                    if (failure) {
                        dispatch_group_async(self.completionGroup ?:
http_request_operation_completion_group(), self.completionQueue ?:
dispatch_get_main_queue(), ^{
                            failure(self, self.error);
                        });
                    }
                } else {
                    if (success) {
                        dispatch_group_async(self.completionGroup ?:
http_request_operation_completion_group(), self.completionQueue ?:
dispatch_get_main_queue(), ^{
                            success(self, responseObject);
                        });
                    }
                }
```

```
            }

            if (self.completionGroup) {
                dispatch_group_leave(self.completionGroup);
            }
        });
    };
#pragma clang diagnostic pop
}
```

一开始我们在`AFHTTPRequestOperationManager`中是调用过这个方法的：

```
[operation setCompletionBlockWithSuccess:success failure:failure];
```

- 我们在把成功和失败的Block传给了这个方法。
- 这个方法也很好理解，就是设置我们之前提到过得`completionBlock`，**当自己数据请求完成，就会调用这个Block。然后我们在这个Block中调用传过来的成功或者失败的Block。**如果error为空，说明请求成功，把数据传出去，否则为失败，把error信息传出。
- 这里也类似AF3.x，可以自定义一个完成组和完成队列。数据可以在我们自定义的完成组和队列中回调出去。
- 除此之外，还有一个有意思的地方：

```
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Warc-retain-cycles"
#pragma clang diagnostic ignored "-Wgnu"
#pragma clang diagnostic pop
```

之前我们说过，这是在忽略编译器的一些警告。

- `-Wgnu`就不说了，是忽略？：。
- 值得提下的是`-Warc-retain-cycles`，这里忽略了循环引用的警告。我们仔细看看就知道`self`持有了`completionBlock`，而`completionBlock`内部持有`self`。这里确实循环引用了。那么AF是如何解决这个循环引用的呢？

我们在回到`AFURLConnectionOperation`，还有一个方法我们之前没讲到，它复写了setCompletionBlock这个方法。

```
//复写setCompletionBlock
- (void)setCompletionBlock:(void (^)(void))block {
    [self.lock lock];
    if (!block) {
        [super setCompletionBlock:nil];
    } else {
        __weak __typeof(self)weakSelf = self;
        [super setCompletionBlock:^ {
            __strong __typeof(weakSelf)strongSelf = weakSelf;

#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wgnu"
            //看有没有自定义的完成组，否则用AF的组
            dispatch_group_t group = strongSelf.completionGroup ?:
url_request_operation_completion_group();
            //看有没有自定义的完成queue，否则用主队列
            dispatch_queue_t queue = strongSelf.completionQueue ?:
dispatch_get_main_queue();
#pragma clang diagnostic pop

            //调用设置的Block,在这个组和队列中
            dispatch_group_async(group, queue, ^{
                block();
            });

            //结束时候置nil，防止循环引用
            dispatch_group_notify(group,
url_request_operation_completion_queue(), ^{
                [strongSelf setCompletionBlock:nil];
            });
        }];
    }
    [self.lock unlock];
}
```

注意，它在我们设置的block调用结束的时候，主动的调用:

```
[strongSelf setCompletionBlock:nil];
```

把Block置空，这样循环引用不复存在了。

好像我们还遗漏了一个东西，就是返回的数据做类型的解析。其实还真不是楼主故意这样东一块西一块的，AF2.x有些代码确实是这样零散。。当然仅仅是相对3.x来说。AFNetworking整体代码质量，以及架构思想已经强过绝大多数开源项目太多了。。这一点毋庸置疑。
我们来接着看看数据解析在什么地方被调用的把：

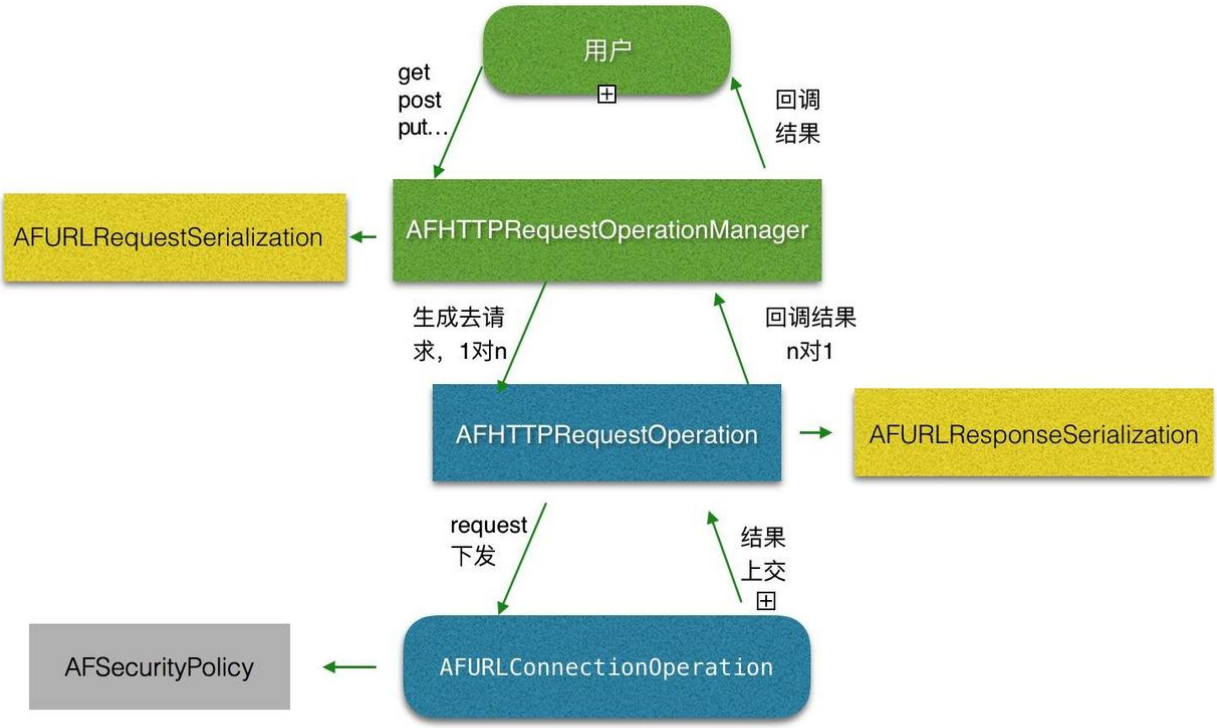
```
- (id)responseObject {
    [self.lock lock];
    if (!_responseObject && [self isFinished] && !self.error) {
        NSError *error = nil;
        //做数据解析
        self.responseObject = [self.responseSerializer
responseObjectForResponse:self.response data:self.responseData error:&error];
        if (error) {
            self.responseSerializationError = error;
        }
    }
    [self.lock unlock];
    return _responseObject;
}
```

`AFHTTPRequestOperation` 复写了 `responseObject` 的get方法，
并且把数据按照我们需要的类型（json、xml等等）进行解析。至于如何解析，可以参考楼主之前AF系列的文章，这里就不赘述了。

有些小伙伴可能会说，楼主你是不是把`AFSecurityPolicy`给忘了啊，其实并没有，它被在`AFURLConnectionOperation`中https认证的代理中被调用，我们之前系列的文章已经讲的非常详细了，感兴趣的朋友可以翻到前面的文章去看看。

至此，AF2.x整个业务流程就结束了。

接下来，我们来总结总结AF2.x整个业务请求的流程：



如上图，我们来梳理一下整个流程：

- 最上层的是`AFHTTPRequestOperationManager`,我们调用它进行get、post等等各种类型的网络请求
- 然后它去调用`AFURLRequestSerialization`做request参数拼装。然后生成了一个`AFHTTPRequestOperation`实例，并把request交给它。然后把`AFHTTPRequestOperation`添加到一个`NSOperationQueue`中。
- 接着`AFHTTPRequestOperation`拿到request后，会去调用它的父类`AFURLConnectionOperation`的初始化方法，并且把相关参数交给它，除此之外，当父类完成数据请求后，它调用了`AFURLResponseSerialization`把数据解析成我们需要的格式（json、XML等等）。
- 最后就是我们AF最底层的类`AFURLConnectionOperation`，它去数据请求，并且如果是https请求，会在请求的相关代理中，调用`AFSecurityPolicy`做https认证。最后请求到的数据返回。

这就是AF2.x整个做网络请求的业务流程。

我们来解决解决之前遗留下来的问题：为什么AF2.x需要一条常驻线程?

首先如果我们用`NSURLConnection`，我们为了获取请求结果有以下三种选择：

1. 在主线程调异步接口
2. 每一个请求用一个线程，对应一个runloop，然后等待结果回调。
3. 只用一条线程，一个runloop，所有结果回调在这个线程上。

很显然AF选择的是第3种方式，创建了一条常驻线程专门处理所有请求的回调事件，这个模型跟`nodejs`有点类似，我们来讨论讨论不选择另外两种方式的原因：

原因一：

试想如果我们所有的请求都在主线程中异步调用，好像没什么不可以？那为什么AF不这么做呢...在这里有两点原因（楼主个人总结的，有不同意见，欢迎讨论）：

第一是，如果我们放到主线程去做，势必要这么写：

```
 [[NSURLConnection alloc] initWithRequest:request delegate:self
startImmediately:YES]
```

这样NSURLConnection的回调会被放在主线程中`NSDefaultRunLoopMode`中，这样我们在其它类似`UITrackingRunLoopMode`模式下，我们是得不到网络请求的结果的，这显然不是我们想要的，那么我们势必需要调用：

```
[connection scheduleInRunLoop:[NSRunLoop currentRunLoop]
forMode:NSRunLoopCommonModes];
```

把它加入`NSRunLoopCommonModes中，试想如果有大量的网络请求，同时回调回来，就会影响我们的UI体验了。

原因二：

另外一点原因是，如果我们请求数据返回，势必要进行数据解析，解析成我们需要的格式，那么这些解析都在主线程中做，给主线程增加额外的负担。
又或者我们回调回来开辟一个新的线程去做数据解析，那么我们有n个请求回来开辟n条线程带来的性能损耗，以及线程间切换带来的损耗，是不是一笔更大的开销。

所以综述两点原因，我们并不适合在主线程中回调。

我们一开始就开辟n条线程去做请求，然后设置runloop保活住线程，等待结果回调。

其实看到这，大家想想都觉得这个方法很傻，为了等待不确定的请求结果，阻塞住线程，白白浪费n条线程的开销。

综上所述，这就是**AF2.x需要一条常驻线程的原因了**。

至此我们把AF2.x核心流程分析完了。

五. 本文总结：AFNetworking到底做了什么?

相信如果从头看到尾的小伙伴，心里都有了一个属于自己的答案。其实在作者心里，并不想去以一言之词总结它，因为`AFNetworking`中凝聚了太多大牛的思想，根本不是看完几遍源码所能去议论的。但是想想也知道，如果我说不总结，估计有些看到这的朋友杀人的心都有...所以我还是赶鸭子上架，来总结总结它。

1. 首先我们需要明确一点的是：

相对于AFNetworking2.x，AFNetworking3.x确实没那么有用了。AFNetworking之前的核心作用就是为了帮我们去调度所有的请求。但是最核心地方却被苹果的`NSURLSession`给借鉴过去了，嗯...是借鉴。这些请求的调度，现在完全由`NSURLSession`给做了，AFNetworking3.x的作用被大大的削弱了。

2. 但是除此之外，其实它还是很有用的：

- **首先它帮我们做了各种请求方式request的拼接。**想想如果我们用`NSURLSession`，我们去做请求，是不是还得自己去考虑各种请求方式下，拼接参数的问题。
- **它还帮我们做了一些公用参数（session级别的），和一些私用参数（task级别的）的分离。**它用Block的形式，支持我们自定义一些代理方法，如果没有实现的话，AF还帮我们做了一些默认的处理。而如果我们用`NSURLSession`的话，还得参照AF这么一套代理转发的架构模式去封装。
- **它帮我们做了自定义的https认证处理。**看了之前讲的AF的安全策略的朋友就知道，如果我们自己用`NSURLSession`实现那几种自定义认证，需要多写多少代码...
- **对于请求到的数据，AF帮我们做了各种格式的数据解析，并且支持我们设置自定义的code范围，自定义的数据方式。**如果不在这些范围中，则直接调用失败block。如果用`NSURLSession`呢？这些都自己去写吧...（你要是做过各种除json外其他的数据解析,就会知道这里面坑有多少...）
- **对于成功和失败的回调处理。**AF帮我们在数据请求到，到回调给用户之间，做了各种错误的判断，保证了成功和失败的回调，界限清晰。在这过程中，AF帮我们做了太多的容错处理，而`NSURLSession`呢？只给了一个完成的回调，我们得多做多少判断，才能拿到一个确定能正常显示的数据?
- ......

- 光是这些网络请求的业务逻辑，AF帮我们做的就太多太多，当然还远不仅于此。它用凝聚着许多大牛的经验方式，帮我在有些处理中做了最优的选择，比如我们之前说到的，回调线程数设置为1的问题...帮我们绕开了很多的坑，比如系统内部并行创建`task`导致id不唯一等等...

3. 而如果我们需要一些UIKit的扩展，AF则提供了最稳定，而且最优化实现方式：

就比如之前说到过得那个状态栏小菊花，如果是我们自己去做，得多写多少代码，而且实现的还没有AF那样质量高。

又或者`AFImageDownloader`，它对于组图片之间的下载协调，以及缓存使用的之间线程调度。对于线程，锁，以及性能各方面权衡，找出最优化的处理方式，试问小伙伴们自己基于`NSURLSession`去写，能到做几分...

所以最后的结论是：**AFNetworking虽然变弱了，但是它还是很有用的。**用它真的不仅仅是习惯，而是因为它确实帮我们做了太多。

SpriteKit 入门与实践

作者：郭鹏

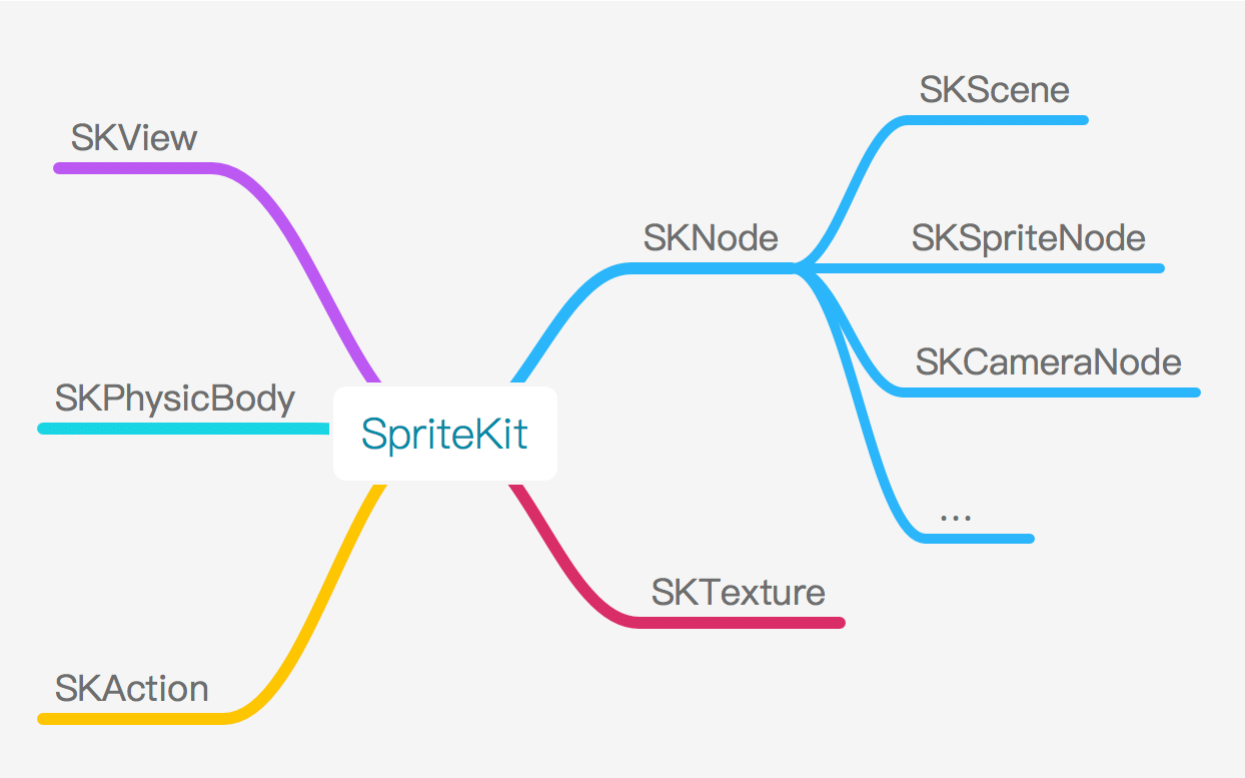
作为一个 iOS 开发，你应该知道一个 app 是怎样从无到有，但对于游戏却不一定。所以本文将带你了解游戏开发方面的知识。

为什么是 SpriteKit

首先，我们需要了解目前手机游戏开发的现状。目前手机游戏引擎最流行的是 Cocos2d-x 和 Unity，它们都是垮平台的引擎，然而这两者对于初学者来说，需要额外学习的东西太多。
SpriteKit 是由 Apple 在 WWDC 2013 发布 2D 游戏引擎，目前可用于 iOS, macOS, tvOS 和 watchOS。它有着良好的设计、简单易用、API 和 Cocos2D 非常像。对于已经熟悉 iOS 开发的人来说，其用法和 UIKit 差不多，而且，他并非只能用来开发游戏。本文后面就会用 SpriteKit 实现类似 iTunes Music 的风格选择交互 UI。

SpriteKit Framework 介绍

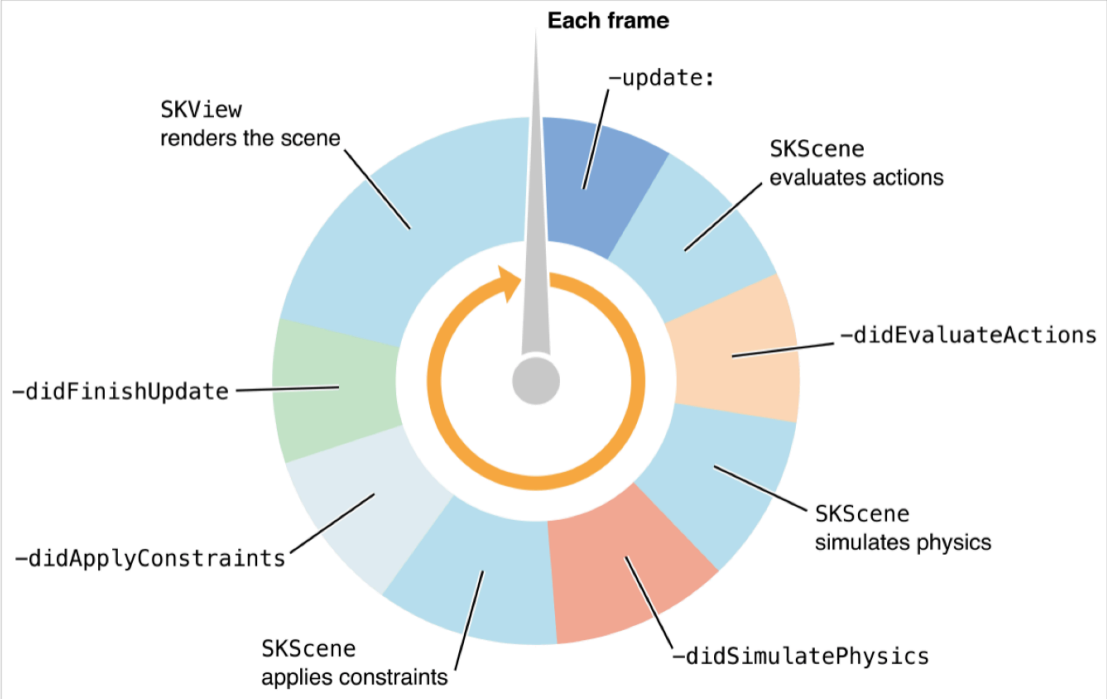
首先我们来看 SpriteKit 主要类结构。



- `SKView` 用来管理和渲染 `SKScene`， 继承自 `UIView`。你甚至可以将 `SKView` 与 `UIKit` 里的其他 `View` 结合使用，比如在其之上加一个 `UIButton`。
- `SKNode` 是基础类，和 `UIKit` 类似， `SpriteKit` 有 `node trees` 的概念，实际中一般和其子类打交道。`SKNode` 定义了一些基础属性和方法，如 `position`, `frame`, `alpha`, `physicBody`, `addChild()`, `runAction()` 等。
- `SKAction` 用来实现位移、缩放等效果，调用 `SKNode.runAction()` 将 `Action` 添加到 `Node`，可以自由组合 `Action` 实现复杂效果。

- `SKPhysicBody` 定义了一个 Node 的物理属性，设置 node.physicBody 后 node 就可以进行物理计算、碰撞检测。SKPhysicBody 包含了质量、速度、弹性、摩擦力等属性。
- `SKScene` 是 `root node`，定义了 SKView 显示的具体内容， physicsWorld 属性可设置重力等全局属性，通过 SKPhysicsContactDelegate 获得碰撞通知。就游戏来说，其内容是动态变化的，所以 SKScene 有一个 rendering loop (见下图)

Figure 1 Frame processing in a scene



每一帧都会执行这个 loop，loop 执行完之后被改变的内容才会重新绘制。子类化 SKScene 时可以重写 `update(_:)` 和 `didXXX` 方法获得回调。

结合 Magnetic 进行代码讲解

上面介绍了 SpriteKit 的一些基本概念，下面我们通过一个示例来介绍如何在普通 app 中结合 SpriteKit 实现一些优雅的交互。此示例受 Github 上的 Magnetic [Magnetic](https://github.com/) 项目启发，用 SpriteKit 实现 iTunes Music 的「个人喜好定制」功能，源码见 https://github.com/iblacksun/Artists， 最后效果如下：


Carrier
8:54 AM
Adele
Bob Dylan
Michael Jackson
David Bowie
Katy Perry
Lady Gaga
Queen
Amy Winehouse
Rihanna
Madonna
Linkin Park
Miley Cyrus
U2
Justin Bieber
Coldplay
Jay-Z
nodes:34 40.7 fps

开始实现

第一步，使用 Xcode 新建一个 Game 类型的项目，Game Technology 选 SpriteKit。简单起见，修改 Deployment Info，使其只支持 iPhone 和 Portrait 方向。Build & Run 之后你会看到我们熟悉的 Hello World。

Hello World 项目中很多模板代码在我们项目中并不需要，删除 GameScene.sks, Actions.sks, GameScene.swift 几个文件；修改 Main.storyboard 将 SKView 的 backgroudColor 修改成白色；修改 GameViewController 替换成如下：

```
class GameViewController: UIViewController {
        override func viewDidLoad() {
                super.viewDidLoad()
                guard let skView = self.view as? SKView else{
                        return
                }
                skView.ignoresSiblingOrder = true
                skView.showsFPS = true
                skView.showsNodeCount = true
        }
}
```

此时如果运行项目的话应该是一个灰色空白界面，底部显示了 nodes 数量和 fps。

构建 Node

现在该主角们登场了，新建 `ArtistsScene.swift` `ArtistNode.swift`，分别继承 `SKScene` 和 `SKShapeNode`。修改 `GameViewController` 的 `viewDidLoad` 方法，在末尾加入代码：

```
let scene = ArtistsScene(size: skView.bounds.size)
skView.presentScene(scene)
```

调用 `presentScene(_:)` 方法呈现 Scene，此时 `ArtistsScene` 就是 root node，因 `SKScene` 的默认背景色是黑色，所以现在运行项目的话会看到一个黑色空白界面。
接下来开始实现功能，重写 `ArtistsScene` 的 `didMove(to:)` 的方法，此方法在 Scene 被 present 时会被调用:

```
override func didMove(to view: SKView) {
        super.didMove(to: view)
        //1.禁用重力
        physicsWorld.gravity = CGVector.zero
        backgroundColor = .white

        let viewWidth = view.frame.size.width
        let viewHeight = view.frame.size.height
        let radius = max(viewWidth, viewHeight)

        //2.添加一个具有向心力的特殊 SKFieldNode.
        let fieldNode = SKFieldNode.radialGravityField()
        fieldNode.region = SKRegion(radius: Float(radius))
        fieldNode.minimumRadius = Float(radius)
        fieldNode.strength = 50
        addChild(fieldNode)

        //3.修改坐标原点
        anchorPoint = CGPoint(x: 0.5, y: 0.5)

        //4. 添加所有 Artist nodes，初始随机分配在左右两侧，受向心力作用，会自动汇聚到
中心点
        for (index, artistName) in artists.enumerated() {
                let x = (index % 2 == 0) ? -viewWidth/2 : viewWidth/2
                let y = CGFloat.random(-viewHeight/2, viewHeight/2)

                let node = ArtistNode(circleOfRadius: 40)
                node.fillColor = .red
                node.position = CGPoint(x: x, y: y)
                addChild(node)
        }
}
```

1. `physicsWorld.gravity = CGVector.zero` 禁用重力作用，否则所有设置过 physicBody 的 node 都会受重力影响自动坠落；
2. 添加一个具有向心力的特殊 `SKFieldNode`，在其 `region` 内的所有 node 都会受影响，自动向中心移动；
3. SpriteKit 的坐标原点在左下角，这点和 UIKit 不一样。设置 `anchorPoint = (0.5, 0.5)`，方便计算后面 `ArtistNode` 的 position；
4. 循环添加 `ArtistNode`，设置其 position，其中 x 平均分配到左右两侧，y 则取顶部和底部间的随机值。
   此时如果运行的话会发现所有 `ArtistNode` 都停留在屏幕两侧并不会想中心靠拢，猜猜原因?
   打开 `ArtistNode`，实现一个 `convenience init` 方法，传入 artistName
   convenience init(artistName :String) {

```
self.init()
self.init(circleOfRadius: 40)
fillColor = .red

physicsBody = SKPhysicsBody(circleOfRadius: frame.size.width / 2)
physicsBody?.allowsRotation = false
physicsBody?.friction = 0
physicsBody?.linearDamping = 3

addMultilineTextNode(artistName, radius: 40)
```

}
在 `init` 中创建一个半径为 40 的圆形 node，填充色是红色；创建一个大小和自身相等的圆形 physicsBody。
接下来的问题是如何将 `artistName` 添加到 ArtistNode？SpriteKit 提供了 `SKLabelNode`，但它不支持文字换行。`addMultilineTextNode(artistName, radius: 40)` 里的代码是将文字转换成图片，然后往 ArtistNode 添加一个 SKSpriteNode。
if let image = UIGraphicsGetImageFromCurrentImageContext(){

```
let texture = SKTexture(image: image)
let spriteNode = SKSpriteNode(texture: texture)
addChild(spriteNode)
```

}
UIGraphicsEndImageContext()

添加交互

完成上面代码并运行之后，所有 ArtistNode 会自动从左右两侧缓慢的向中心移动，但并不能拖拽和点击。
SKNode 继承自 UIResponder (NSResponder)，其事件处理和 UIView 一样：通过 `touchesBegan` `touchesMoved` `touchesEnded` `touchesCancelled` 几个方法处理。
因我们的 root node 是 `ArtistsScene`，所以我们可以重写 `ArtistsScene` 的这几个方法来处理拖拽和点击事件。

```
override func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?) {
        guard let touch = touches.first else {
                return
        }
        let previous = touch.previousLocation(in: self)
        let location = touch.location(in: self)
        if location.distance(from: previous) == 0{
                return
        }
        isMoving = true
        let x = location.x - previous.x
        let y = location.y - previous.y
        for node in children{
                let distance = node.position.distance(from: location)
                let acceleration: CGFloat = 3 * pow(distance, 1/2)
                let direction = CGVector(dx: x * acceleration, dy: y *
acceleration)
                node.physicsBody?.applyForce(direction)
        }
}

override func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent?) {
        guard let touch = touches.first, !isMoving else {
                isMoving = false
                return
        }
        isMoving = false
        let location = touch.location(in: self)
        guard let artistNode = artistNodeAt(location) else{
                return
        }
        artistNode.isSelected = !artistNode.isSelected
}

override func touchesCancelled(_ touches: Set<UITouch>, with event: UIEvent?)
{
        isMoving = false
}
```

1. `touchesMoved` 方法中处理拖拽事件，如果发生拖拽，则向所有 node 施加一个作用力，使之随着手指位置一起移动；
2. `touchesEnded` 中处理点击事件。通过 `isMoving` 区分这个事件是拖拽还是点击，点击的话查找出被点击的 `ArtistNode`，设置其 `isSelected` 属性。
   接下来看看 `ArtistNode.isSelected` 的实现：
   var isSelected = false{

```
didSet{
        guard oldValue != isSelected else {
                return
        }
        removeAction(forKey: "scale")
        let scaleAction = SKAction.scale(to: (isSelected ? 1.5 : 1.0),
duration: 0.2)
        run(scaleAction, withKey: "scale")
}
```

}
`didSet` 获得设值之后的回调，通过 `run(_ action:)`，选中缩放至 1.5，取消选择还原至 1.0。

参考链接

- https://developer.apple.com/spritekit/
- SpriteKit Tutorial: Create an Interactive Children's Book with SpriteKit and Swift 3
- 2D Apple Games by Tutorials

基于 Rx 的网络层实践

作者：李富强

日常开发中，API请求的管理是我们无法回避的一个难题，相对于web，客户端的网络请求处理要更加复杂，例如异步、通用参数、数据模型化、通用错误处理、可取消。更高一些的用户体验要求是，不阻塞用户的操作，随时可以退出以及重试，这也就要求我们对网络请求的各个状态处理都要非常完善，这就需要一个非常完善的网络层提供支撑。

网络框架

Apple 在 iOS SDK 中是为我们提供了 HTTP 网络请求接口的，从 CFNetwork 到 NSURLConnection，以及最近的 NSURLSession，但这些框架我们很少直接去使用。因为官方的 SDK 只提供了最基础的接口，而我们实际业务开发中，网络层需要提供更多的功能，才能提高业务开发效率。下面简单举几个例子说明一下为什么需要框架。

第一，HTTP 参数的处理，例如GET的请求参数在经过 URL Encoding 之后添加 URL 的 query 中，POST 请求的包体组装(multi-part 或者 form)，都是比较繁琐但是相对通用的逻辑，这些都应该在网络层处理之后，让业务层可以无感知地去使用。

第二，返回数据的处理，例如，我们指定服务器返回 JSON 数据，框架会把这些信息处理添加到 HTTP 协议中，我们的业务层逻辑就会很简单，获得可以序列化的数据或者直接报错。

第三，请求状态的管理，例如，限制请求最大并发数、网络请求的取消等。如果使用 NSURLConnection 。还有一个比较经典的示例，就是 NSURLConnection 需要 runloop 机制的支持才能处理回调，如果你启动一个线程，但是没有启动它的 runloop，你在这个线程中发起网络请求的话，这个请求不会产生回调。如果我们把所有的 NSURLConnection 都放到主线程中发起，这会产生一个其他问题，比如主线程的负担过重。

由于这些需求的存在，iOS 开发中诞生了几乎是事实标准的网络框架，早期的 ASIHTTPRequest，这个框架是基于 CFNetwok 直接实现的，逻辑非常复杂，从我开始使用到结束使用，也只是看懂了大概的架构，但是细节一直没有深入探究。ASI 使用底层技术的好处在于性能的确较高。相对于 AFNetworking ，ASI 提供了更多的功能支持，例如大文件下载、同步请求。后续因为作者不再维护，同时暴露出了一些安全问题，以及 AFNetworking 的流行，慢慢大家就不再使用了。Objective-C 时代，AFNetworking 几乎是事实上的网络框架SDK，我个人感觉，最主要的原因是因为简单，AFN 2.x 中 NSURLConnection + runloop、NSOperation、HTTP 参数的处理等，都是非常优雅的代码，建议每个 iOS 开发人员都应该看看。

我们上面提到的一些网络层的需求，AFNetworking 都帮我们进行了处理，但是一般来说，在 AFNetworking 这类网络框架的基础上，大家往往还会再做一层封装，一方面适应自己的业务需求，另外一方面是考虑到后续框架的变迁而隐藏底层网络框架，比如 AFNetworking 从 2.x 到 3.x 的变迁等。我个人的封装方式是定义一个 Request 类，隐藏底层的 NSURLSessionTask 或者 NSOperation。例如：

```
class CusomRequest {
    var task: URLSessionTask?
    let url: URL

    init(method: Method, url: URL, parameters: [String: Any]? = nil,
headers: [String: Any]? = nil) {
        // ...
    }

    func start(callback: CompletionClosure) {
        // ...
    }

    func cancel() {

    }
}
```

Rx

当Swift第一版本发布的时候，我快速学习了一下，但是非常失望，感觉这是一个半成品，很多基本的特性都无法支持。后续断断续续写了一些小东西，Swift 3.0 之后，终于感觉比较成熟。公司内部的项目，由于基础库庞大、包大小的限制等原因，一直无法实践 Swift 。但是我业余的 night job 基本上都在使用 Swift，在做小东西的过程中，也需要设计网络层，当我去探索的时候，发现了 RxAlamofire 这个效率极高的框架，把 RxSwift 和 Alamofire 结合到了一起。

简单介绍一下我对 Reactive Programming 的理解，最开始接触这个概念是从ReactiveCocoa 开始的，后续也经常使用，感觉非常方便。虽然 ReactiveCocoa 也有 Swift 版本，但是当我开始阅读 Rx 官方的文档之后，就果断选择了 RxSwift 替代 ReactiveCocoa，我个人的理解有三方面：

- 首先，我认为 Rx 的文档质量非常优秀，与学习 ReactiveCocoa 时资料匮乏形成鲜明对比，大家只需要看官网的文档就能无障碍理解和使用。
- 其次，我认为 RxSwift 的概念上更好理解，Observable 与 Observer 的关系非常清晰，Obervable 生产 data flow，经过各种 Operators 变化，最终提供给Observer 消费。官网的一段话：`In ReactiveX an observer subscribes to an Observable.`。
- 最后，Rx 支持很多语言，并且概念上都是一致的，让我们可以做到 `Learn Once, Write Everywhere`，我自己尝试过 RxJava，除了一些语法上不一致，其他概念理解起来毫无障碍。甚至 RxJS 在前端领域也越来越流行，这个诱惑很致命的。

我认为对 Rx 设计思想最好的解释是 Rx 官网 (http://reactivex.io/) 的一段话：`ReactiveX is a combination of the best ideas from the Observer pattern, the Iterator pattern, and functional programming`。三种设计模式或者思想，观察者模式，迭代器模式，函数式编程。

针对 Rx，我个人的观点是 `Rx 抹平了调用接口`，无论是进行网络请求，还是定位，甚至 UI 变化，业务层不再需要了解API的具体调用方式，只要服务层封装到位，对业务层而言，只用获取一个 Observable 处理即可，这就是我认为的最大的优势，再加上比较通用化的 Operators，更能简化客户端复杂的数据流操作。举个简单的例子，定位完成之后，发起多个网络请求，请求全部成功之后，页面数据才能显示，中间任何一个步骤出现错误都会中断。如果使用 Rx，你会发现这些问题可以大大简

化，而使用异步回调，会需要非常多的变量来辅助处理。

网络层

解决的问题

选定网络框架之后，就需要搭建自己项目的网络层实现。在网络库的基础上，我们之所以为项目再设定一个网络层，除了我们之前提到的隐藏底层网络框架细节之外，还主要为了解决几个问题。首先，通用参数，例如设备唯一标识、登录用户 token 信息、反爬虫签名、设备的基本信息等。

其次， 统一的错误处理，例如底层网络 HTTP 发生错误、JSON 解析或者数据格式产生错误等，我们可以与 PM 约定较为用户友好的提示，在网络层统一处理。否则业务层在处理网络数据的时都需要再做重复这样的逻辑，有洁癖的人应该都无法这种 copy-paste 模式的重复代码。

最后，统一的业务数据处理。一般来说，API 服务器返回的 JSON 数据都有通用的结构。例如

```
{
   "code": 200,
   "message": "请求已成功提交",
   "data": {
      "id": 111,
      "nick_name": "大强",
      "avatar": "http://xxx.png"
   }
}
```

假设两端服务器与客户端约定，code 等于200表示业务处理成功，其他值表示错误码，那我们就可以在网络层判断 code 值，如果不等于正常值，直接返回 error，业务层的重复逻辑就会再次减少。同时，服务器返回的数据都在 data 字段，在确认 code 等于200，即业务处理成功之后，我们可以把模型化逻辑也在网络层统一处理。经过这两个逻辑的处理，业务层的逻辑进一步简化，失败返回 error，成功返回数据模型。

实现

我们最终的目的是业务层接口尽量简单，处理的逻辑尽可能少，先看看我们最终要达到的效果是什么，个人主页请求用户信息，我会在代码注释中为大家简单解释一下这段代码：

```
/// 网络层
extension Network{
        func userDetail(userid: Int) -> Observable<User> {
                let url = URL(string: baseUrl + "/user/detail.do")!
                let parameters: [String: Any] = ["userid": userid]
                return rx_json(.get, url, parameters: parameters).data()
        }
}

/// 业务层
Network.default.userDetail(userid: self.userId)
        .subscribe(
                onNext:{ [weak self] (user) in
                        /// 处理正常返回的业务数据
                        guard let `self` = self else {return}
                        self.user = user
                        self.refreshUI()
                },
                onError: { [weak self] (error) in
                        /// 处理错误信息
            Toast(text: error.localizedDescription).show()
            self?.navigationController?.popViewController(animated: true)
        }
        )
        /// 假设网络请求耗，而用户此时退出页面，这行代码会把请求cancel，避免网络资源的浪
费
        .addDisposableTo(disposeBag)
```

看完了最终效果，我们开始架构我们的整个网络框架，首先是 Network 对象，用于管理网络请求的单例对象。下面这段代码解决了上面提到的网络层的第一个问题，通用参数的处理。

```
class Network {

    /// 单例
    static let `default`: Network = {
        return Network()
    }()

    // 主机地址
    let baseUrl = "http://example.com"

    /// JSON解析的通用key值
    static let ok = 200
    static let codeKey = "code"
    static let messageKey = "message"
    static let dataKey = "data"
    static let cursorKey = "cursor"
```

```
    /// 添加通用参数
    func commonParameters(parameters: [String: Any]?) -> [String: Any] {
        var newParameters: [String: Any] = [:]
        if parameters != nil {
            newParameters = parameters!
        }

        /// 添加通用参数
        /// 用户token参数
        if let token = AccountCenter.default.user?.token {
            newParameters["token"] = token
        }
        /// App版本信息
        if let version = Bundle.main.object(forInfoDictionaryKey:
"CFBundleShortVersionString") {
                        newParameters["app_version"] = version
        }
        return newParameters
    }

    /// 统一的API请求入口
    func rx_json(_ method: Alamofire.HTTPMethod,
                 _ url: URLConvertible,
                 parameters: [String: Any]? = nil,
                 encoding: ParameterEncoding = URLEncoding.default,
                 headers: [String: String]? = nil)
        -> Observable<JSON> {
        return string(
            method,
            url,
            parameters: commonParameters(parameters: parameters),
            encoding: encoding,
            headers: headers
        )
        .common()
    }
}
```

这就是网络层最基础的代码，不知道大家有没有注意 `rx_json` 最后一行代码。`Observable<String>.common()` 方法，把 `string` 函数返回的 `Observable<String>` 类型转换为了 `Observable<JSON>` 类型，这个是通过为 `Observable` 添加 extension 实现的，这里不仅仅是简单的类型转换，还包含了我们之前所说的通用的网络错误处理。在具体实现之前，我们可以简单定义一下基本的网络错误码和提示语：

```
enum ErrorCode: Int {
    case `default` = -1212
    case json = -1213
    case parameter = -1214
}

enum ErrorMessage: String {
    case `default` = "网络状态不佳，请稍候再试!"
    case json = "服务器数据解析错误!"
    case parameter = "参数错误，请稍候再试!"
}

extension NSError {
    class func network(reason: String, code: Int) -> NSError {
        let userInfo = [NSLocalizedDescriptionKey: reason.localized,
                        NSLocalizedFailureReasonErrorKey: reason.localized]
        return NSError(domain: "com.example.networkerror", code: code,
userInfo: userInfo)
    }
}
```

下面是 common 方法的实现，我将会在代码注释中，为大家详细讲解这段代码：

```
/*
这段代码之所以这么写，是因为Swift中，当我们为某个有泛型的类添加extension时，不支持限定泛型为继承自某个类，例如：
extension Observable where Element == String {}
这段代码会报错，因为我们无法限定Element继承自String或就是String类，但是可以限定Element实现了某个protocol。所以我们可以通过protocol extension的方式绕过去，参考：
http://www.marisibrothers.com/2016/03/extending-swift-generic-types.html。
*/
protocol StringProtocol {}
extension String : StringProtocol {}

extension Observable where Element: StringProtocol {
    func common() -> Observable<JSON> {
        return self
            .catchError({ (error) -> Observable<Element> in
                    /// 这里统一处理更加底层的网络错误，比如服务器404或者500等常见错误。
                    /// 将底层网络框架返回的错误提示，在这里转换为更加友好的用户提示，然后把错误抛给业务层进行处理。
                return Observable<Element>.create({ (observer) -> Disposable
in
                    observer.on(.error(NSError.network(reason:
ErrorMessage.default.rawValue, code: ErrorCode.default.rawValue)))
                    return Disposables.create()
                })
```

```
            })
            .flatMap { (element) -> Observable<JSON> in
                let string = element as! String
                /// 通用的业务数据处理逻辑，将服务器返回的字符串进行解析
                /// 获取JSON数据中的code字段，判断业务请求是否成功，即code是否等于200
                return Observable<JSON>.create({ (observer) -> Disposable in
                        /// 解析JSON
                    let json = JSON.parse(string)
                    if let code = json[Network.codeKey].int, code ==
Network.ok {
                            /// 业务请求成功，把解析后的JSON数据传递下去
                        observer.on(.next(json))
                        observer.on(.completed)
                    } else {
                            /// 如果业务请求失败，则处理错误信息后，把错误抛到业务层处
理
                        var reason = ErrorMessage.default.rawValue
                        if let message = json[Network.messageKey].string {
                            reason = message
                        }
                        var code = ErrorCode.default.rawValue
                        if let c = json[Network.codeKey].int {
                            code = c
                        }
                        observer.on(.error(NSError.network(reason: reason,
code: code)))
                    }
                    return Disposables.create()
                })
            }
            .observeOn(MainScheduler.instance) /// 确定业务层的订阅发生在主线程，
避免在子线程操作UI
    }
}
```

经过上面的处理，我们处理了底层网络错误以及基本的业务数据的处理，这时我们获得了解析之后的 JSON 数据。但是一般来说，我们业务层更倾向于使用模型化之后的数据，在 OC 时代，这个模型化通常会比较麻烦，基本上所有的网络请求都需要把这个逻辑重复一遍。下面是我在 Swift 中的实现，同样，我会通过代码注释的方式为大家解释这段代码：

```
/// 依旧是上面的原因，我们需要为JSON定义一个protocol，来达到限定泛型的类型为JSON类的目的
protocol JSONProtocol {}
extension JSON: JSONProtocol{}

extension Observable where Element: JSONProtocol {
    /// 如果返回数据是列表信息，在这个方法中解析列表数据，并将列表数据模型化为模型数组，这
里使用 T 代表模型类。
    typealias ListType<T> = ([T], String?)
    /// 限定T模型类实现了 ObjectMapper 中的 BaseMappable protocol，通过这个
```

```
protocol就能直接进行模型化
    func list<T: BaseMappable>(callback: ((T) -> Void)? = nil) ->
Observable<ListType<T>> {
        return self.flatMap{ (element) -> Observable<ListType<T>> in
            let json = element as! JSON
            return Observable<ListType<T>>.create { (observer) -> Disposable
in
                    /// 从json数据的data字段中取出列表数据，并使用ObjectMapper的方法
把数据模型化
                if let data = json[Network.dataKey].arrayObject, let array =
Mapper<T>().mapArray(JSONObject: data) {
                        /// 返回模型类数组以及服务器返回的用于分页的cursor
                    observer.on(.next((array,
json[Network.cursorKey].string)))
                    observer.on(.completed)
                } else {
                        /// 解析错误，向业务层抛出通用的json解析错误
                    observer.on(.error(NSError.network(reason:
ErrorMessage.json.rawValue, code: ErrorCode.json.rawValue)))
                }
                return Disposables.create()
            }
        }
    }

    /// 如果服务器返回的数据中，data对应的是某个数据模型，直接调用这个方法即可获得模型化之
后的数据
    func data<T: BaseMappable>(callback: ((T) -> Void)? = nil) ->
Observable<T> {
        return self.flatMap { (element) -> Observable<T> in
            let json = element as! JSON
            return Observable<T>.create { (observer) -> Disposable in
                if let data = json[Network.dataKey].dictionaryObject, let
object = Mapper<T>().map(JSON: data) {
                    if let callback = callback {
                        callback(object)
                    }
                    observer.on(.next(object))
                    observer.on(.completed)
                } else {
                        /// 解析错误，向业务层抛出通用的json解析错误
                    observer.on(.error(NSError.network(reason:
ErrorMessage.json.rawValue, code: ErrorCode.json.rawValue)))
                }
                return Disposables.create()
            }
        }
    }
```

```
    /// 针对只处理状态码的返回数据，使用这个bool()方法处理，返回的bool值只有true，其他情况都是失败
    func bool(callback: (() -> Void)? = nil) -> Observable<Bool> {
        return self.flatMap { (element) -> Observable<Bool> in
            return Observable<Bool>.create { (observer) -> Disposable in
                if let callback = callback {
                    callback()
                }
                observer.on(.next(true))
                observer.on(.completed)
                return Disposables.create()
            }
        }
    }
}
```

上面的代码，我利用图虫的API，简单写了一个示例供大家参考，https://github.com/lihei12345/tuchongapp。

总结

经过上面的步骤，我们介绍了 UIKit 提供的SDK和第三方网络框架，基于他们，最后我们实现了一个网络层，当然这是比较简化的网络层。对于比较复杂和大型的 App 而言，功能可能远远不止这些。通过 Rx，我们不需要再封装 `CustomRequest` 对象，不需要在业务层在写一堆判断逻辑，Rx 把数据流变得异常清晰 -- 业务数据或错误。这篇文章，我主要想为大家演示了 Swift 的语法带来的新的可能性，我们要避免使用 Objective-C 的思维写 Swift，才能真正发挥 Swift 这个现代语言带来的优势。

iOS 组件化 —— 路由设计思路分析

作者：@一缕殇流化隐半边冰霜

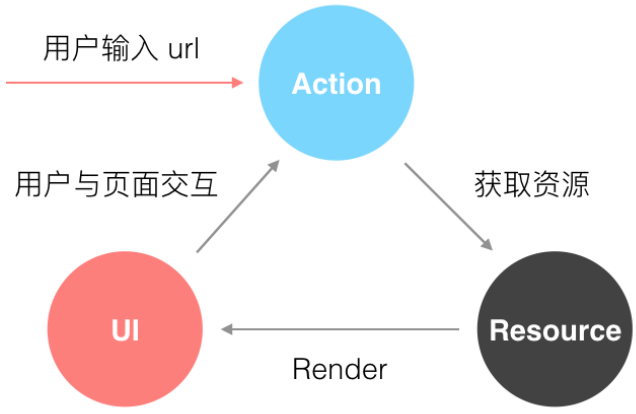
随着用户的需求越来越多，对 App 的用户体验也变的要求越来越高。为了更好的应对各种需求，开发人员从软件工程的角度，将 App 架构由原来简单的 MVC 变成 MVVM， VIPER 等复杂架构。更换适合业务的架构，是为了后期能更好的维护项目。

但是用户依旧不满意，继续对开发人员提出了更多更高的要求，不仅需要高质量的用户体验，还要求快速迭代，最好一天出一个新功能，而且用户还要求不更新就能体验到新功能。为了满足用户需求，于是开发人员就用 H5，ReactNative，Weex 等技术对已有的项目进行改造。项目架构也变得更加的复杂，纵向的会进行分层，网络层，UI 层，数据持久层。每一层横向的也会根据业务进行组件化。尽管这样做了以后会让开发更加有效率，更加好维护，但是如何解耦各层，解耦各个界面和各个组件，降低各个组件之间的耦合度，如何能让整个系统不管多么复杂的情况下都能保持"高内聚，低耦合"的特点？这一系列的问题都摆在开发人员面前，亟待解决。今天就来谈谈解决这个问题的一些思路。

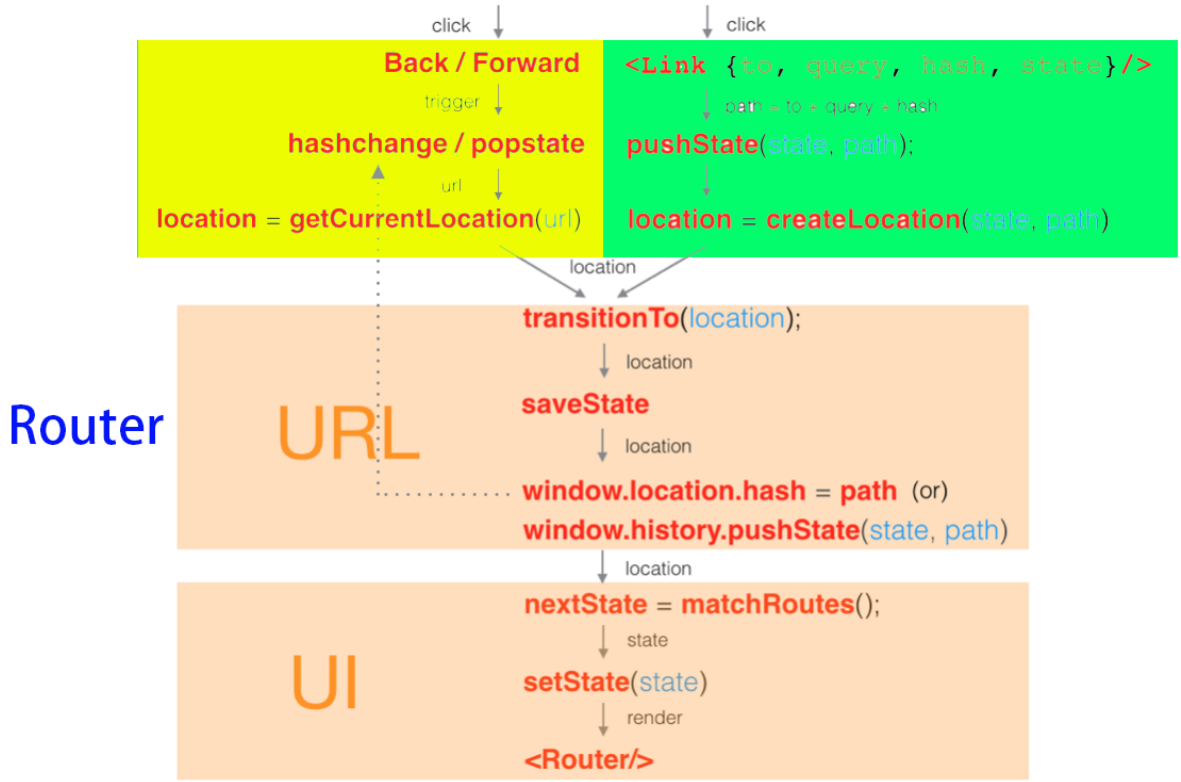
一. 引子

大前端发展这么多年了，相信也一定会遇到相似的问题。近两年 SPA 发展极其迅猛，React 和 Vue 一直处于风口浪尖，那我们就看看他们是如何处理好这一问题的。

在 SPA 单页面应用，路由起到了很关键的作用。路由的作用主要是保证视图和 URL 的同步。在前端的眼里看来，视图是被看成是资源的一种表现。当用户在页面中进行操作时，应用会在若干个交互状态中切换，路由则可以记录下某些重要的状态，比如用户查看一个网站，用户是否登录、在访问网站的哪一个页面。而这些变化同样会被记录在浏览器的历史中，用户可以通过浏览器的前进、后退按钮切换状态。总的来说，用户可以通过手动输入或者与页面进行交互来改变 URL，然后通过同步或者异步的方式向服务端发送请求获取资源，成功后重新绘制 UI，原理如下图所示：



react-router 通过传入的 location 到最终渲染新的 UI，流程如下：



location 的来源有2种，一种是浏览器的回退和前进，另外一种是直接点了一个链接。新的 location 对象后，路由内部的 matchRoutes 方法会匹配出 Route 组件树中与当前 location 对象匹配的一个子集，并且得到了 nextState，在 this.setState(nextState) 时就可以实现重新渲染 Router 组件。

大前端的做法大概是这样的，我们可以把这些思想借鉴到 iOS 这边来。上图中的 Back / Forward 在 iOS这边很多情况下都可以被 UINavgation 所管理。所以 iOS 的 Router 主要处理绿色的那一块。

二. App 路由能解决哪些问题

既然前端能在 SPA 上解决 URL 和 UI 的同步问题，那这种思想可以在 App 上解决哪些问题呢?

思考如下的问题，平时我们开发中是如何优雅的解决的：

1.3D-Touch 功能或者点击推送消息，要求外部跳转到 App 内部一个很深层次的一个界面。

比如微信的 3D-Touch 可以直接跳转到“我的二维码”。“我的二维码”界面在我的里面的第三级界面。或者再极端一点，产品需求给了更加变态的需求，要求跳转到 App 内部第十层的界面，怎么处理?

2.自家的一系列 App 之间如何相互跳转?

如果自己 App 有几个，相互之间还想相互跳转，怎么处理?

3.如何解除 App 组件之间和 App 页面之间的耦合性?

随着项目越来越复杂，各个组件，各个页面之间的跳转逻辑关联性越来越多，如何能优雅的解除各个组件和页面之间的耦合性?

4.如何能统一 iOS 和 Android 两端的页面跳转逻辑？甚至如何能统一三端的请求资源的方式?

项目里面某些模块会混合 ReactNative，Weex，H5 界面，这些界面还会调用 Native 的界面，以及 Native 的组件。那么，如何能统一 Web 端和 Native 端请求资源的方式?

5.如果使用了动态下发配置文件来配置 App 的跳转逻辑，那么如果做到 iOS 和 Android 两边只要共用一套配置文件?

6.如果 App 出现 bug 了，如何不用 JSPatch ，就能做到简单的热修复功能?

比如 App 上线突然遇到了紧急 bug，能否把页面动态降级成 H5，ReactNative，Weex？或者是直接换成一个本地的错误界面?

7.如何在每个组件间调用和页面跳转时都进行埋点统计？每个跳转的地方都手写代码埋点？利用 Runtime AOP ?

8.如何在每个组件间调用的过程中，加入调用的逻辑检查，令牌机制，配合灰度进行风控逻辑?

9.如何在 App 任何界面都可以调用同一个界面或者同一个组件？只能在 AppDelegate 里面注册单例来实现?

比如 App 出现问题了，用户可能在任何界面，如何随时随地的让用户强制登出？或者强制都跳转到同一个本地的 error 界面？或者跳转到相应的 H5，ReactNative，Weex 界面？如何让用户在任何界面，随时随地的弹出一个 View ?

以上这些问题其实都可以通过在 App 端设计一个路由来解决。那么我们怎么设计一个路由呢?

三. App 之间跳转实现

在谈 App 内部的路由之前，先来谈谈在 iOS 系统间，不同 App 之间是怎么实现跳转的。

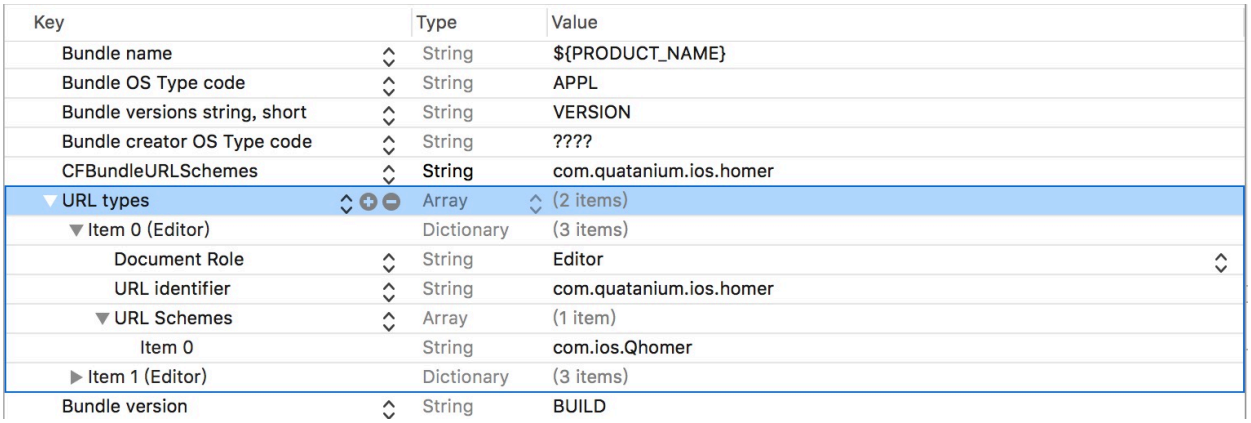
1. URL Scheme方式

iOS 系统是默认支持 URL Scheme 的，具体见官方文档。

比如说，在 iPhone 的 Safari 浏览器上面输入如下的命令，会自动打开一些 App：

```
// 打开邮箱
mailto://

// 给110拨打电话
tel://110
```

在 iOS 9 之前只要在 App 的 info.plist 里面添加 URL types - URL Schemes，如下图：



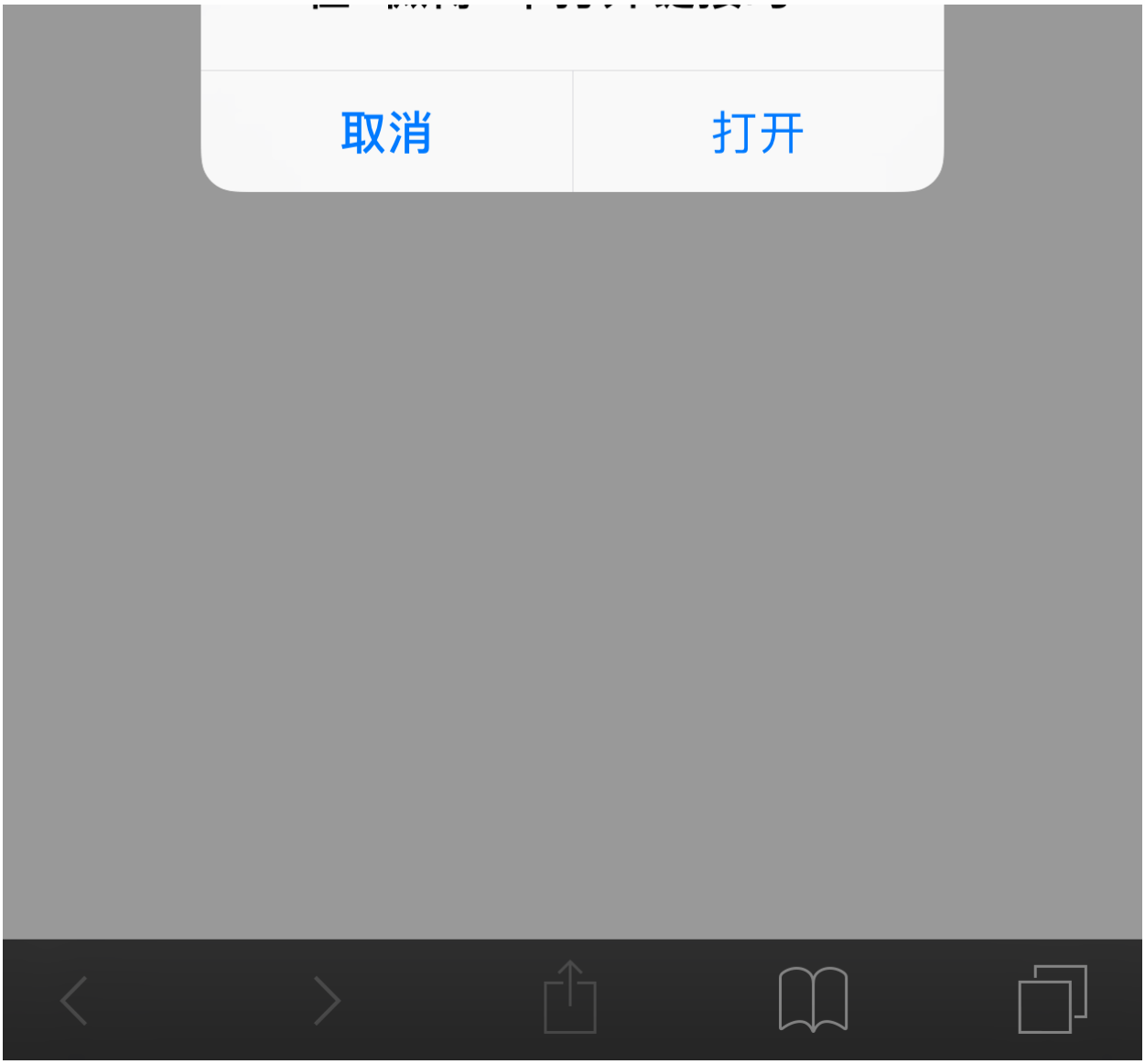
这里就添加了一个 com.ios.Qhomer 的 Scheme。这样就可以在 iPhone 的 Safari 浏览器上面输入：

```
com.ios.Qhomer://
```

就可以直接打开这个 App 了。

关于其他一些常见的 App，可以从 iTunes 里面下载到它的 ipa 文件，解压，显示包内容里面可以找到 info.plist 文件，打开它，在里面就可以相应的 URL Scheme。

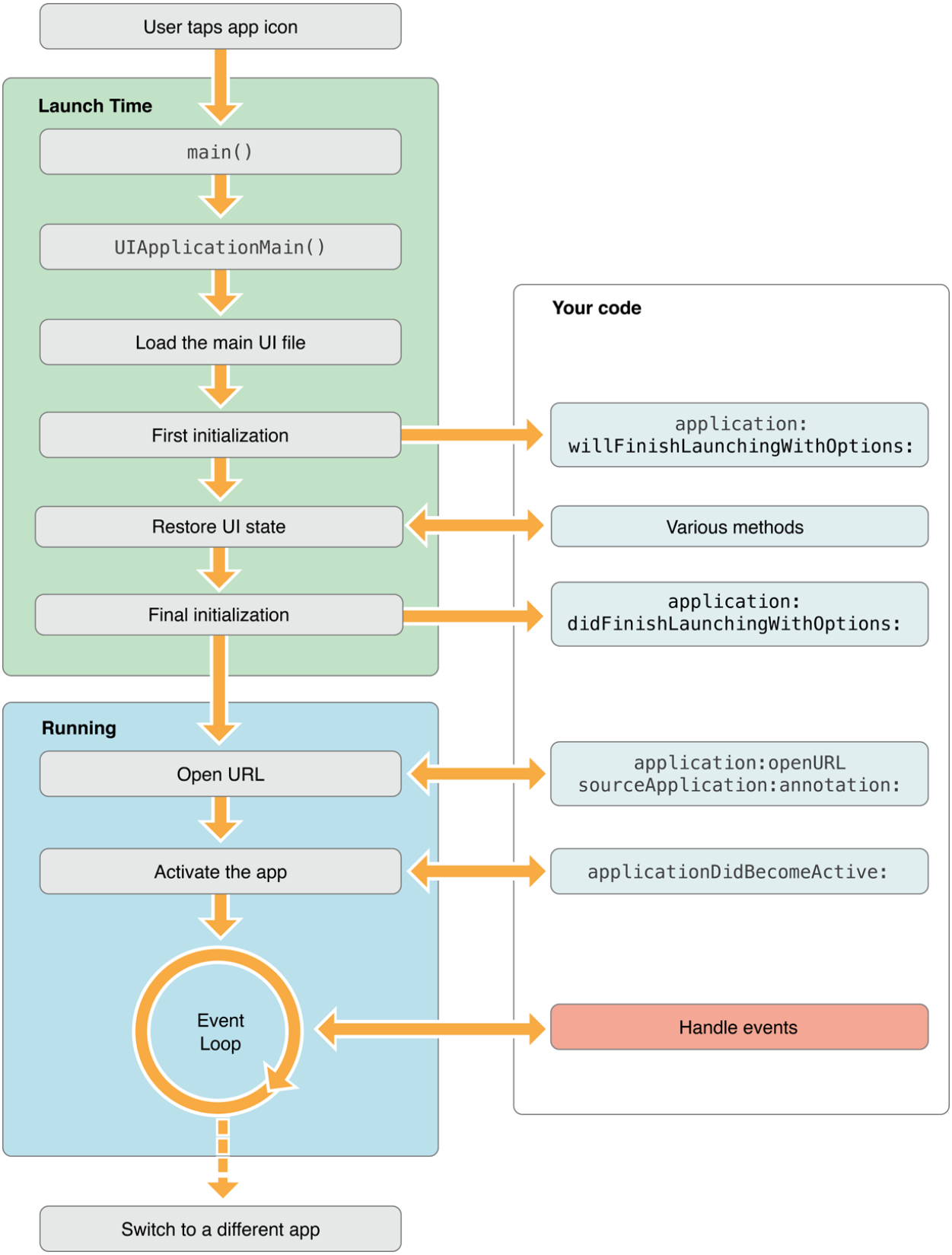
```
// 手机QQ
mqq://

// 微信
weixin://

// 新浪微博
sinaweibo://

// 饿了么
eleme://
```



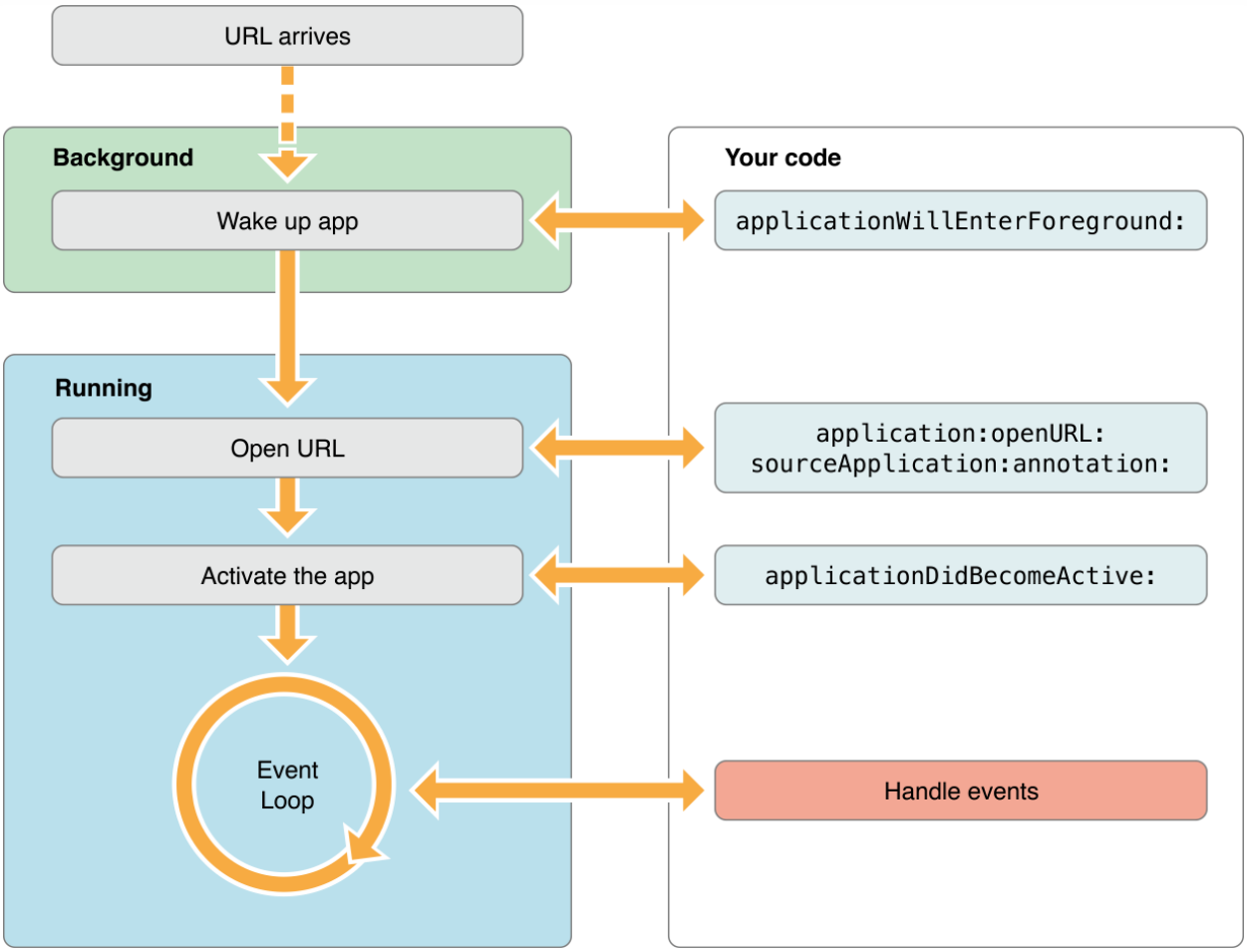
当然了，某些 App 对于调用 URL Scheme 比较敏感，它们不希望其他的 App 随意的就调用自己。

```
- (BOOL)application:(UIApplication *)application
           openURL:(NSURL *)url
  sourceApplication:(NSString *)sourceApplication
         annotation:(id)annotation
{
    NSLog(@"sourceApplication: %@", sourceApplication);
    NSLog(@"URL scheme:%@", [url scheme]);
    NSLog(@"URL query: %@", [url query]);

    if ([sourceApplication isEqualToString:@"com.tencent.weixin"]){
        // 允许打开
        return YES;
    }else{
        return NO;
    }
}
```

如果待调用的 App 已经运行了，那么它的生命周期如下：



如果待调用的 App 在后台，那么它的生命周期如下：



明白了上面的生命周期之后，我们就可以通过调用[application:openURL:sourceApplication:annotation:]这个方法，来阻止一些 App 的随意调用。

如上图，饿了么 App 允许通过 URL Scheme 调用，那么我们可以在 Safari 里面调用到饿了么 App。手机 QQ 不允许调用，我们在 Safari里面也就没法跳转过去。

关于 App 间的跳转问题，感兴趣的可以查看官方文档Inter-App Communication。

App 也是可以直接跳转到系统设置的。比如有些需求要求检测用户有没有开启某些系统权限，如果没有开启就弹框提示，点击弹框的按钮直接跳转到系统设置里面对应的设置界面。

iOS 10 支持通过 URL Scheme 跳转到系统设置

iOS10跳转系统设置的正确姿势

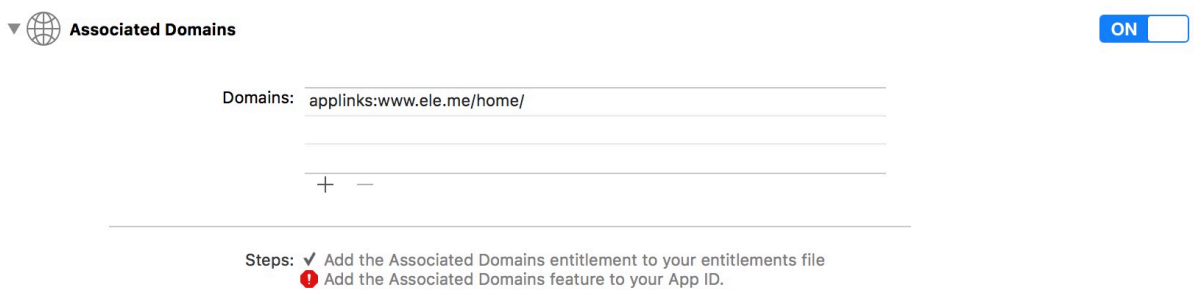
关于 iOS 系统功能的 URL 汇总列表

2. Universal Links 方式

虽然在微信内部开网页会禁止所有的 Scheme，但是 iOS 9.0 新增加了一项功能是 Universal Links，使用这个功能可以使我们的 App 通过 HTTP 链接来启动 App。
1.如果安装过 App，不管在微信里面 http 链接还是在 Safari 浏览器，还是其他第三方浏览器，都可以打开 App。
2.如果没有安装过 App，就会打开网页。

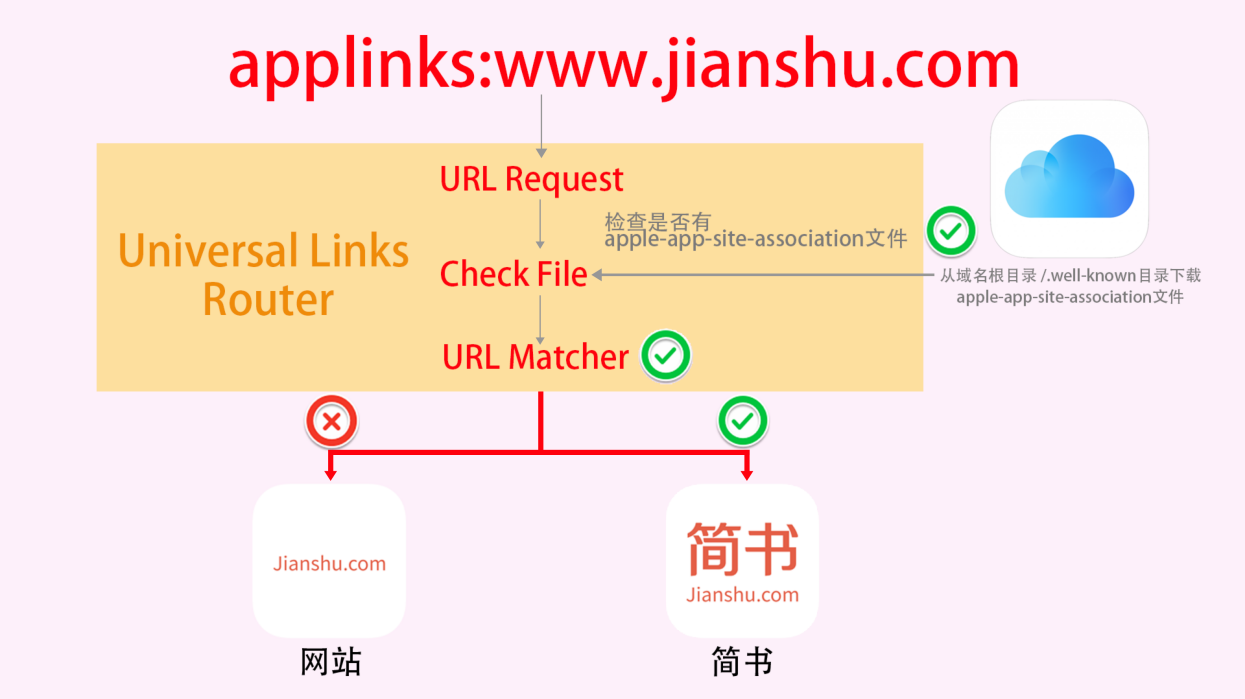
具体设置需要3步：

1.App 需要开启 Associated Domains 服务，并设置 Domains，注意必须要 applinks：开头。



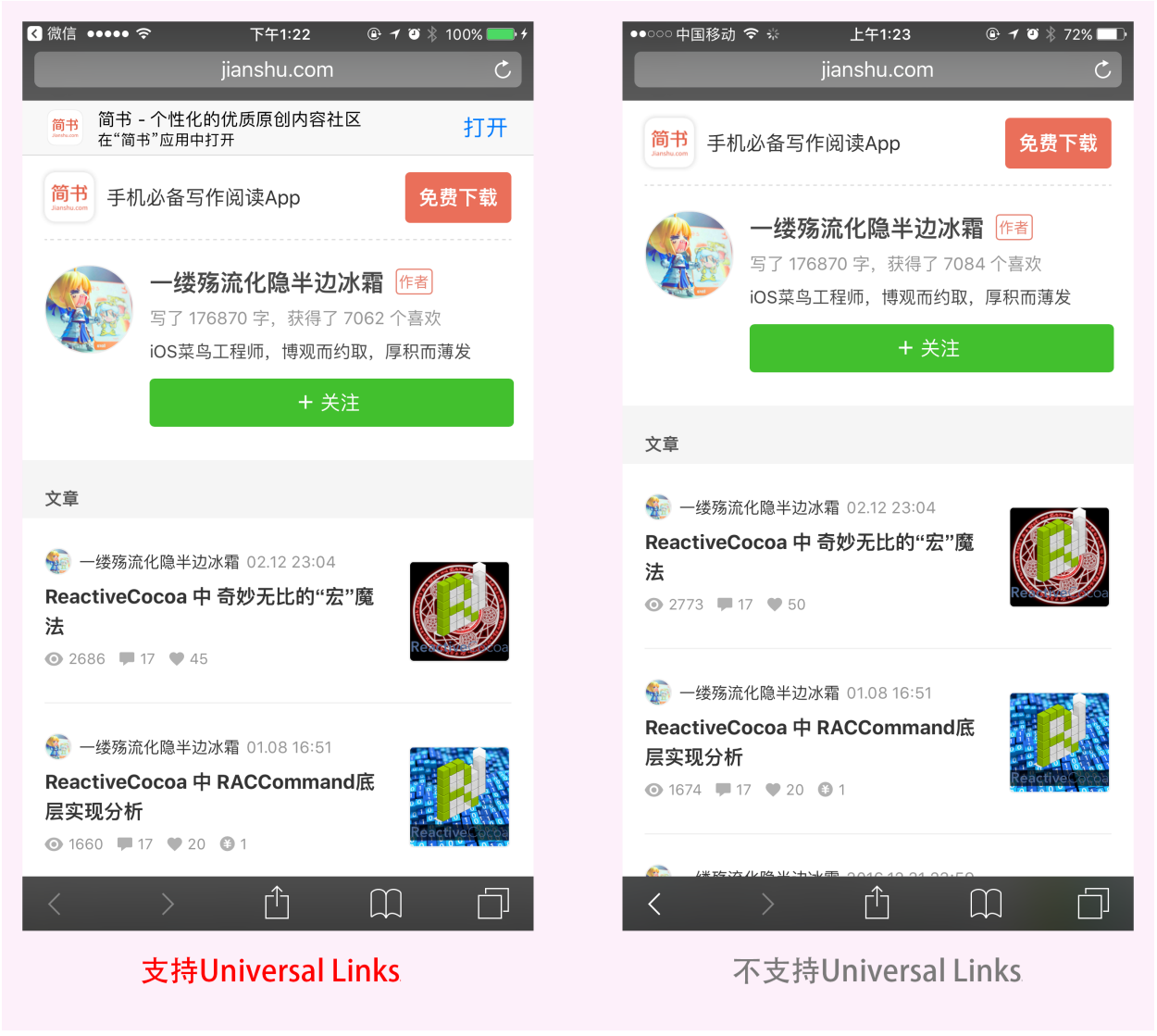
2.域名必须要支持 HTTPS。

3.上传内容是 Json 格式的文件，文件名为 apple-app-site-association 到自己域名的根目录下，或者 .well-known 目录下。iOS 自动会去读取这个文件。具体的文件内容请查看官方文档。



如果 App 支持了 Universal Links 方式，那么可以在其他 App 里面直接跳转到我们自己的 App 里面。如下图，点击链接，由于该链接会 Matcher 到我们设置的链接，所以菜单里面会显示用我们的 App 打开。

在浏览器里面也是一样的效果，如果是支持了 Universal Links 方式，访问相应的 URL，会有不同的效果。如下图：



支持Universal Links　　不支持Universal Links

以上就是iOS系统中 App 间跳转的二种方式。

从 iOS 系统里面支持的 URL Scheme 方式，我们可以看出，对于一个资源的访问，苹果也是用 URI 的方式来访问的。

> **统一资源标识符**（英语：Uniform Resource Identifier，或**URI**)是一个用于标识某一互联网资源名称的字符串。 该种标识允许用户对网络中（一般指万维网）的资源通过特定的协议进行交互操作。URI的最常见的形式是统一资源定位符（URL）。

举个例子：

这是一段 URI，每一段都代表了对应的含义。对方接收到了这样一串字符串，按照规则解析出来，就能获取到所有的有用信息。

这个能给我们设计 App 组件间的路由带来一些思路么？如果我们想要定义一个三端（iOS，Android，H5）的统一访问资源的方式，能用 URI 的这种方式实现么？

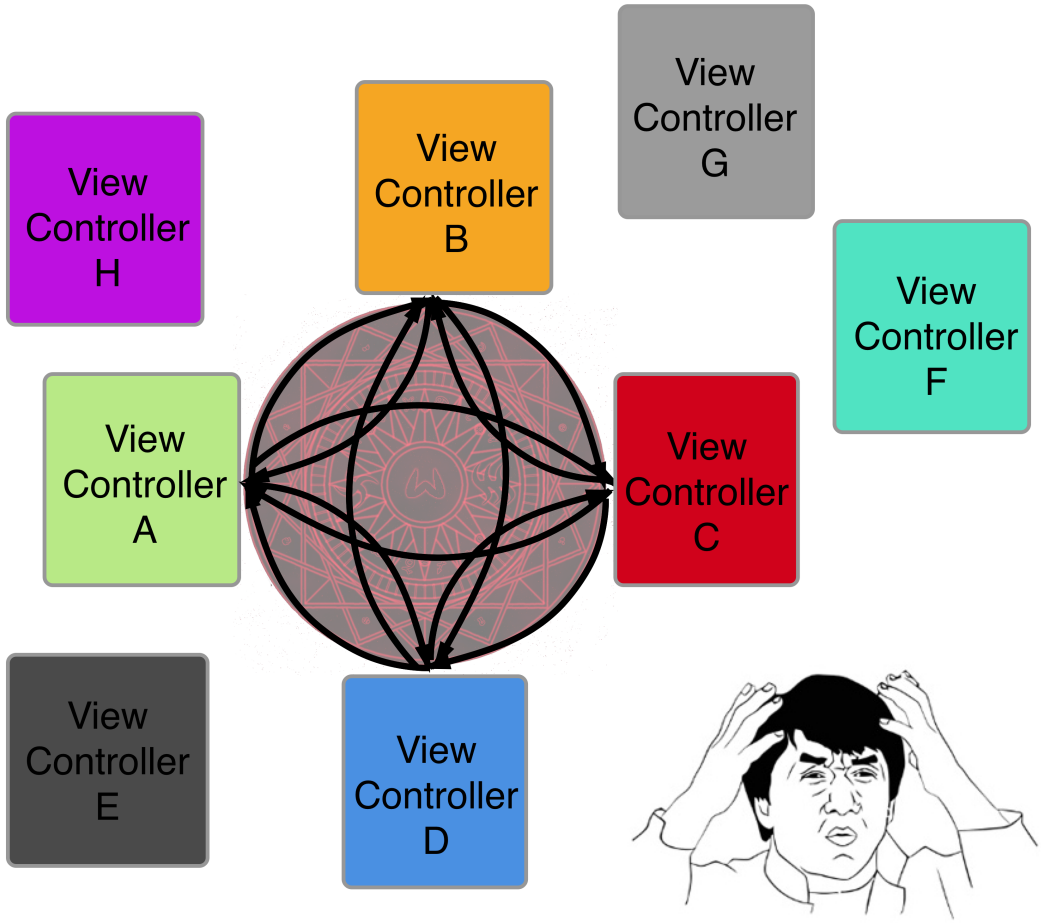
四. App 内组件间路由设计

上一章节中我们介绍了 iOS 系统中，系统是如何帮我们处理 App 间跳转逻辑的。这一章节我们着重讨论一下，App 内部，各个组件之间的路由应该怎么设计。关于 App 内部的路由设计，主要需要解决2个问题：

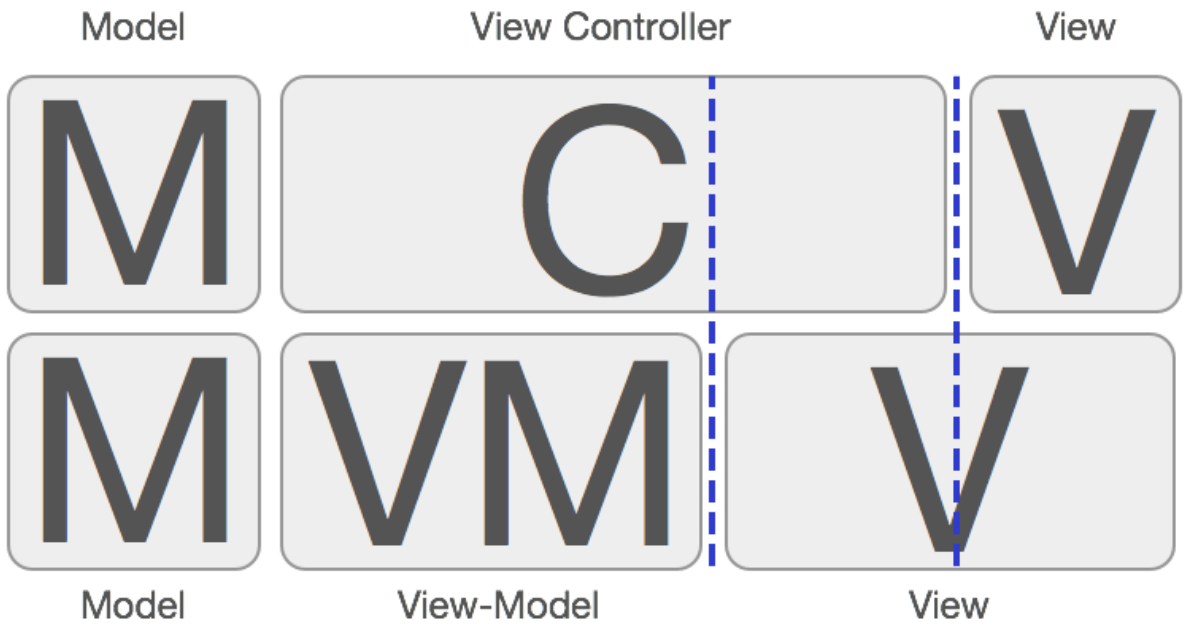
1.各个页面和组件之间的跳转问题。
2.各个组件之间相互调用。

先来分析一下这两个问题。

1. 关于页面跳转



在 iOS 开发的过程中，经常会遇到以下的场景，点击按钮跳转 Push 到另外一个界面，或者点击一个 cell Present 一个新的ViewController。在 MVC 模式中，一般都是新建一个 VC，然后Push / Present 到下一个 VC。但是在 MVVM 中，会有一些不合适的情况。



众所周知，MVVM 把 MVC 拆成了上图演示的样子，原来 View 对应的与数据相关的代码都移到 ViewModel 中，相应的 C 也变瘦了，演变成了 M-VM-C-V 的结构。这里的 C 里面的代码可以只剩下页面跳转相关的逻辑。如果用代码表示就是下面这样子：

假设一个按钮的执行逻辑都封装成了 command。

```
    @weakify(self);
    [[[_viewModel.someCommand executionSignals] flatten] subscribeNext:^(id
x) {
        @strongify(self);
        // 跳转逻辑
        [self.navigationController pushViewController:targetViewController
animated:YES];
  }];
```

上述的代码本身没啥问题，但是可能会弱化 MVVM 框架的一个重要作用。

MVVM 框架的目的除去解耦以外，还有2个很重要的目的：

1. 代码高复用率
2. 方便进行单元测试

如果需要测试一个业务是否正确，我们只要对 ViewModel 进行单元测试即可。前提是假定我们使用 ReactiveCocoa 进行 UI 绑定的过程是准确无误的。目前绑定是正确的。所以我们只需要单元测试到 ViewModel 即可完成业务逻辑的测试。

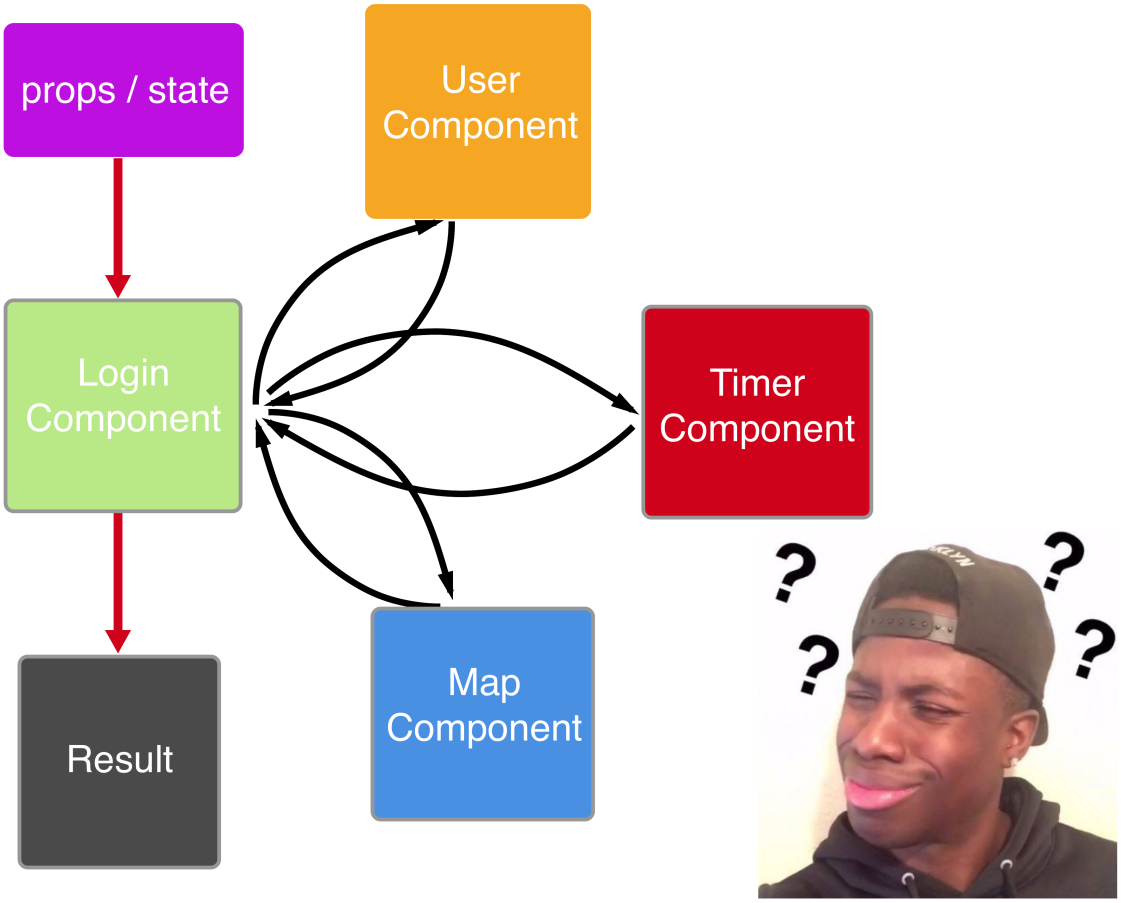
页面跳转也属于业务逻辑，所以应该放在 ViewModel 中一起单元测试，保证业务逻辑测试的覆盖率。

把页面跳转放到 ViewModel 中，有2种做法，第一种就是用路由来实现，第二种由于和路由没有关系，所以这里就不多阐述，有兴趣的可以看lpd-mvvm-kit这个库关于页面跳转的具体实现。

页面跳转相互的耦合性也就体现出来了：

1.由于 pushViewController 或者 presentViewController，后面都需要带一个待操作的 ViewController，那么就必须要引入该类，import 头文件也就引入了耦合性。
2.由于跳转这里写死了跳转操作，如果线上一旦出现了 bug，这里是不受我们控制的。
3.推送消息或者是 3D-Touch 需求，要求直接跳转到内部第10级界面，那么就需要写一个入口跳转到指定界面。

2. 关于组件间调用



关于组件间的调用，也需要解耦。随着业务越来越复杂，我们封装的组件越来越多，要是封装的粒度拿捏不准，就会出现大量组件之间耦合度高的问题。组件的粒度可以随着业务的调整，不断的调整组件职责的划分。但是组件之间的调用依旧不可避免，相互调用对方组件暴露的接口。如何减少各个组件之间的耦合度，是一个设计优秀的路由的职责所在。

3. 如何设计一个路由

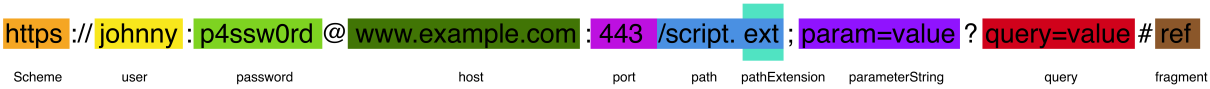
如何设计一个能完美解决上述2个问题的路由，让我们先来看看 GitHub 上优秀开源库的设计思路。以下是我从 Github 上面找的一些路由方案，按照 Star 从高到低排列。依次来分析一下它们各自的设计思路。

（1）JLRoutes Star 3189

JLRoutes 在整个 Github 上面 Star 最多，那就来从它来分析分析它的具体设计思路。

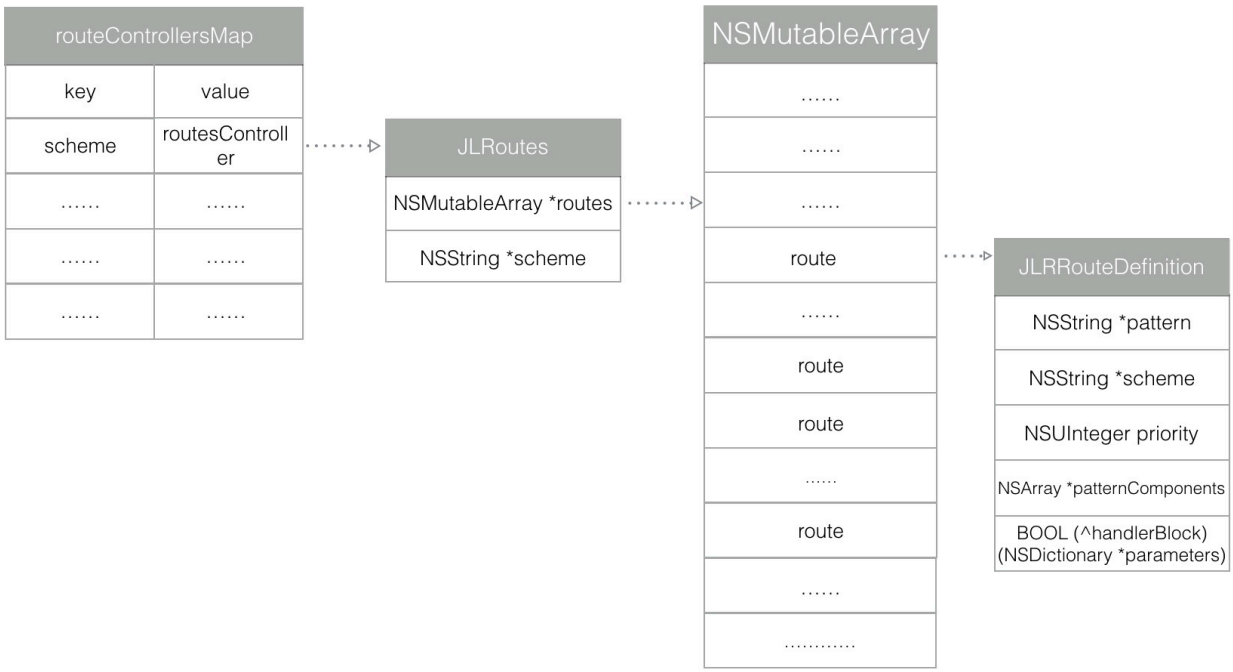
首先 JLRoutes 是受 URL Scheme 思路的影响。它把所有对资源的请求看成是一个 URI。

首先来熟悉一下 NSURLComponent 的各个字段：



> Note
> The URLs employed by the NSURL
> class are described in RFC 1808, RFC 1738, and RFC 2732.

JLRoutes 会传入每个字符串，都按照上面的样子进行切分处理，分别根据 RFC 的标准定义，取到各个 NSURLComponent。



JLRoutes 全局会保存一个 Map，这个 Map 会以 scheme 为 Key， JLRoutes 为 Value。所以在 routeControllerMap 里面每个 scheme 都是唯一的。

至于为何有这么多条路由，笔者认为，如果路由按照业务线进行划分的话，每个业务线可能会有不相同的逻辑，即使每个业务里面的组件名字可能相同，但是由于业务线不同，会有不同的路由规则。

举个例子：如果滴滴按照每个城市的打车业务进行组件化拆分，那么每个城市就对应着这里的每个 scheme。每个城市的打车业务都有叫车，付款.....等业务，但是由于每个城市的地方法规不相同，所以这些组件即使名字相同，但是里面的功能也许千差万别。所以这里划分出了多个 route，也可以理解为不同的命名空间。

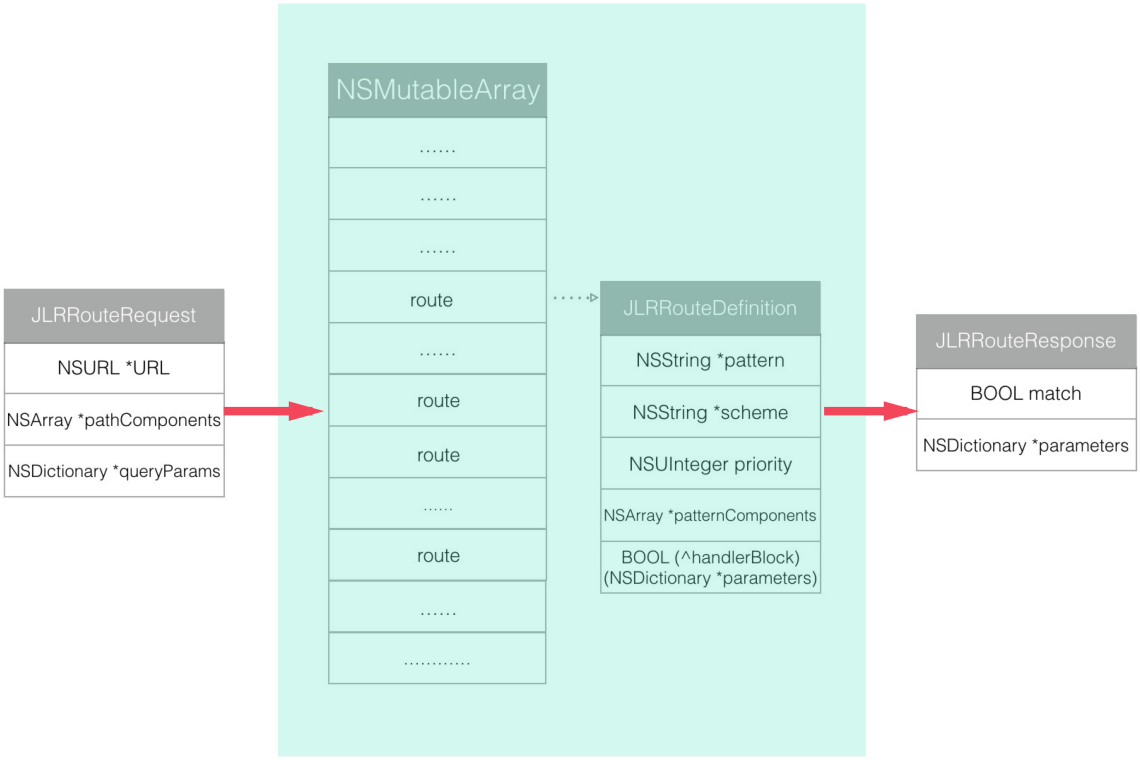
在每个 JLRoutes 里面都保存了一个数组，这个数组里面保存了每个路由规则 JLRRouteDefinition 里面会保存外部传进来的 block 闭包，pattern，和拆分之后的 pattern。

在每个 JLRoutes 的数组里面，会按照路由的优先级进行排列，优先级高的排列在前面。

```
- (void)_registerRoute:(NSString *)routePattern priority:
(NSUInteger)priority handler:(BOOL (^)(NSDictionary
*parameters))handlerBlock
{
    JLRRouteDefinition *route = [[JLRRouteDefinition alloc]
initWithScheme:self.scheme pattern:routePattern priority:priority
handlerBlock:handlerBlock];

    if (priority == 0 || self.routes.count == 0) {
        [self.routes addObject:route];
    } else {
        NSUInteger index = 0;
        BOOL addedRoute = NO;

        // 找到当前已经存在的一条优先级比当前待插入的路由低的路由
        for (JLRRouteDefinition *existingRoute in [self.routes copy]) {
            if (existingRoute.priority < priority) {
                // 如果找到，就插入数组
                [self.routes insertObject:route atIndex:index];
                addedRoute = YES;
                break;
            }
            index++;
        }

        // 如果没有找到任何一条路由比当前待插入的路由低的路由，或者最后一条路由优先级和当
前路由一样，那么就只能插入到最后。
        if (!addedRoute) {
            [self.routes addObject:route];
        }
    }
}
```

由于这个数组里面的路由是一个单调队列，所以查找优先级的时候只用从高往低遍历即可。

具体查找路由的过程如下：



首先根据外部传进来的URL初始化一个 JLRRouteRequest，然后用这个 JLRRouteRequest 在当前的路由数组里面依次 request，每个规则都会生成一个 response，但是只有符合条件的 response 才会 match，最后取出匹配的 JLRRouteResponse 拿出其字典 parameters 里面对应的参数就可以了。查找和匹配过程中重要的代码如下：

```
- (BOOL)_routeURL:(NSURL *)URL withParameters:(NSDictionary *)parameters
executeRouteBlock:(BOOL)executeRouteBlock
{
    if (!URL) {
        return NO;
    }

    [self _verboseLog:@"Trying to route URL %@", URL];

    BOOL didRoute = NO;
    JLRRouteRequest *request = [[JLRRouteRequest alloc] initWithURL:URL];

    for (JLRRouteDefinition *route in [self.routes copy]) {
        // 检查每一个route，生成对应的response
        JLRRouteResponse *response = [route routeResponseForRequest:request
decodePlusSymbols:shouldDecodePlusSymbols];
        if (!response.isMatch) {
            continue;
        }

        [self _verboseLog:@"Successfully matched %@", route];
```

```
        if (!executeRouteBlock) {
            // 如果我们被要求不允许执行，但是又找了匹配的路由response。
            return YES;
        }

        // 装配最后的参数
        NSMutableDictionary *finalParameters = [NSMutableDictionary
dictionary];
        [finalParameters addEntriesFromDictionary:response.parameters];
        [finalParameters addEntriesFromDictionary:parameters];
        [self _verboseLog:@"Final parameters are %@", finalParameters];

        didRoute = [route callHandlerBlockWithParameters:finalParameters];

        if (didRoute) {
            // 调用Handler成功
            break;
        }
    }

    if (!didRoute) {
        [self _verboseLog:@"Could not find a matching route"];
    }

    // 如果在当前路由规则里面没有找到匹配的路由，当前路由不是global 的，并且允许降级到
global里面去查找，那么我们继续在global的路由规则里面去查找。
    if (!didRoute && self.shouldFallbackToGlobalRoutes && ![self
_isGlobalRoutesController]) {
        [self _verboseLog:@"Falling back to global routes..."];
        didRoute = [[JLRoutes globalRoutes] _routeURL:URL
withParameters:parameters executeRouteBlock:executeRouteBlock];
    }

    // 最后，依旧没有找到任何能匹配的，如果有unmatched URL handler，调用这个闭包进行
最后的处理。

if, after everything, we did not route anything and we have an unmatched URL
handler, then call it
    if (!didRoute && executeRouteBlock && self.unmatchedURLHandler) {
        [self _verboseLog:@"Falling back to the unmatched URL handler"];
        self.unmatchedURLHandler(self, URL, parameters);
    }

    return didRoute;
}
```

举个例子：

我们先注册一个 Router，规则如下：

```
[[JLRoutes globalRoutes] addRoute:@"/:object/:action"
handler:^BOOL(NSDictionary *parameters) {
  NSString *object = parameters[@"object"];
  NSString *action = parameters[@"action"];
  // stuff
  return YES;
}];
```

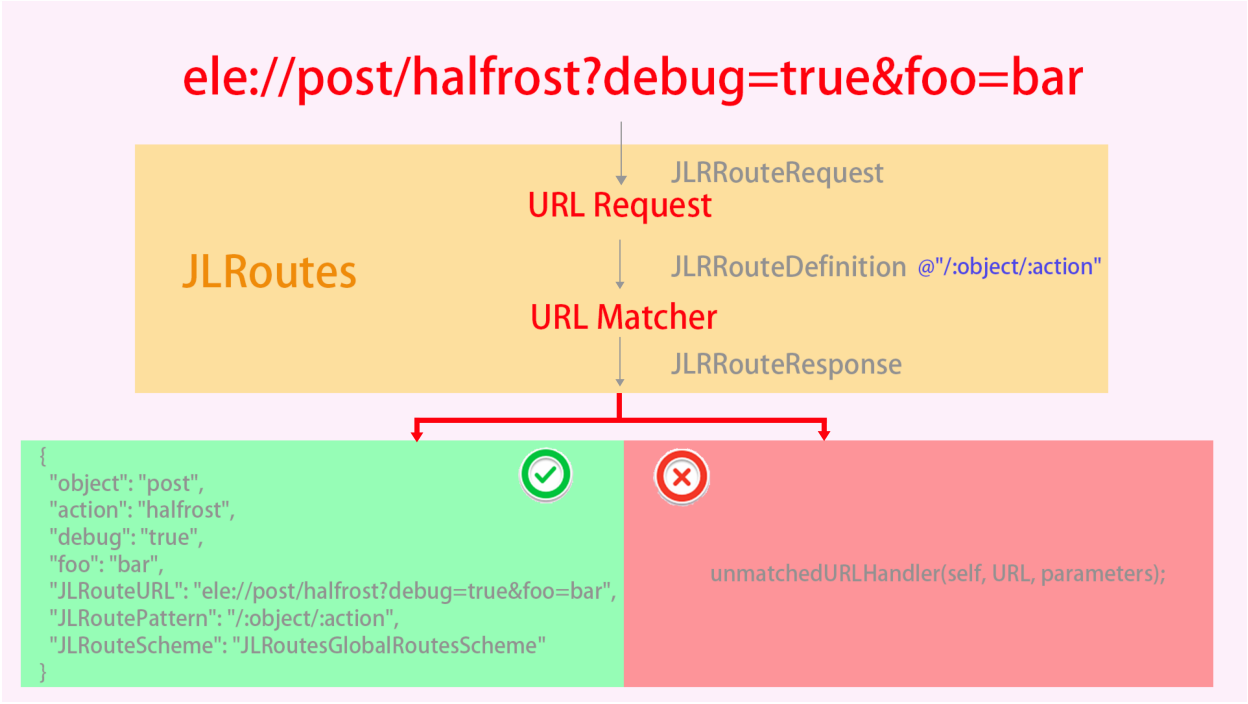
我们传入一个 URL，让 Router 进行处理。

```
NSURL *editPost = [NSURL URLWithString:@"ele://post/halfrost?
debug=true&foo=bar"];
[[UIApplication sharedApplication] openURL:editPost];
```

匹配成功之后，我们会得到下面这样一个字典：

```
{
  "object": "post",
  "action": "halfrost",
  "debug": "true",
  "foo": "bar",
  "JLRouteURL": "ele://post/halfrost?debug=true&foo=bar",
  "JLRoutePattern": "/:object/:action",
  "JLRouteScheme": "JLRoutesGlobalRoutesScheme"
}
```

把上述过程图解出来，见下图：



JLRoutes 还可以支持 Optional 的路由规则，假如定义一条路由规则：

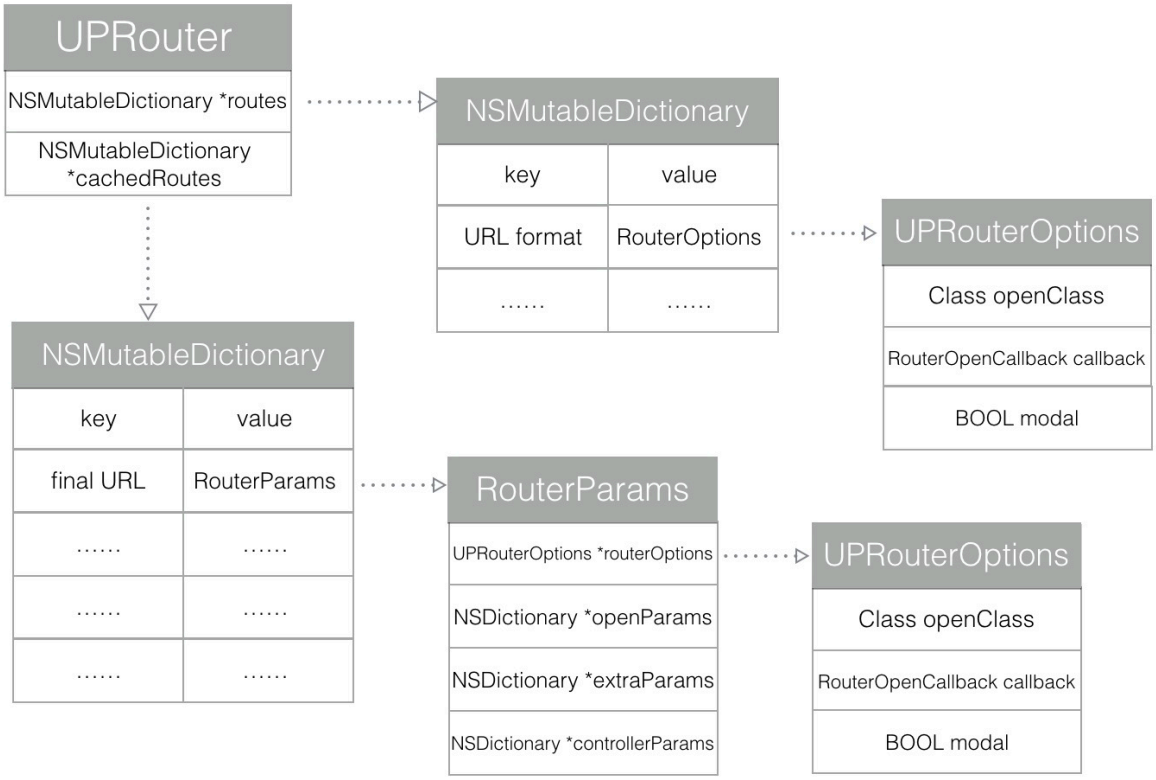
```
/the(/foo/:a)(/bar/:b)
```

JLRoutes 会帮我们默认注册如下4条路由规则：

```
/the/foo/:a/bar/:b
/the/foo/:a
/the/bar/:b
/the
```

（2）routable-ios Star 1415

Routable 路由是用在 in-app native 端的 URL router, 它可以用在 iOS 上也可以用在Android上。



UPRouter 里面保存了2个字典。routes 字典里面存储的 Key 是路由规则，Value 存储的是 UPRouterOptions。cachedRoutes 里面存储的 Key 是最终的 URL，带传参的，Value 存储的是 RouterParams。RouterParams 里面会包含在 routes 匹配的到的 UPRouterOptions，还有额外的打开参数 openParams 和一些额外参数 extraParams。

```
- (RouterParams *)routerParamsForUrl:(NSString *)url extraParams:
(NSDictionary *)extraParams {
    if (!url) {
        //if we wait, caching this as key would throw an exception
        if (_ignoresExceptions) {
            return nil;
        }
        @throw [NSException exceptionWithName:@"RouteNotFoundException"
                                       reason:[NSString
stringWithFormat:ROUTE_NOT_FOUND_FORMAT, url]
                                     userInfo:nil];
    }

    if ([self.cachedRoutes objectForKey:url] && !extraParams) {
        return [self.cachedRoutes objectForKey:url];
    }

   // 比对url通过/分割之后的参数个数和pathComponents的个数是否一样
```

```
    NSArray *givenParts = url.pathComponents;
    NSArray *legacyParts = [url componentsSeparatedByString:@"/"];
    if ([legacyParts count] != [givenParts count]) {
        NSLog(@"Routable Warning - your URL %@ has empty path components -
this will throw an error in an upcoming release", url);
        givenParts = legacyParts;
    }

    __block RouterParams *openParams = nil;
    [self.routes enumerateKeysAndObjectsUsingBlock:
     ^(NSString *routerUrl, UPRouterOptions *routerOptions, BOOL *stop) {

         NSArray *routerParts = [routerUrl pathComponents];
         if ([routerParts count] == [givenParts count]) {

             NSDictionary *givenParams = [self
paramsForUrlComponents:givenParts routerUrlComponents:routerParts];
             if (givenParams) {
                 openParams = [[RouterParams alloc]
initWithRouterOptions:routerOptions openParams:givenParams extraParams:
extraParams];
                 *stop = YES;
             }
         }
     }];

    if (!openParams) {
        if (_ignoresExceptions) {
            return nil;
        }
        @throw [NSException exceptionWithName:@"RouteNotFoundException"
                                       reason:[NSString
stringWithFormat:ROUTE_NOT_FOUND_FORMAT, url]
                                     userInfo:nil];
    }
    [self.cachedRoutes setObject:openParams forKey:url];
    return openParams;
}
```

这一段代码里面重点在干一件事情，遍历 routes 字典，然后找到参数匹配的字符串，封装成 RouterParams 返回。

```
- (NSDictionary *)paramsForUrlComponents:(NSArray *)givenUrlComponents
routerUrlComponents:(NSArray *)routerUrlComponents {

    __block NSMutableDictionary *params = [NSMutableDictionary dictionary];
    [routerUrlComponents enumerateObjectsUsingBlock:
     ^(NSString *routerComponent, NSUInteger idx, BOOL *stop) {

         NSString *givenComponent = givenUrlComponents[idx];
         if ([routerComponent hasPrefix:@":"]) {
             NSString *key = [routerComponent substringFromIndex:1];
             [params setObject:givenComponent forKey:key];
         }
         else if (![routerComponent isEqualToString:givenComponent]) {
             params = nil;
             *stop = YES;
         }
     }];
    return params;
}
```

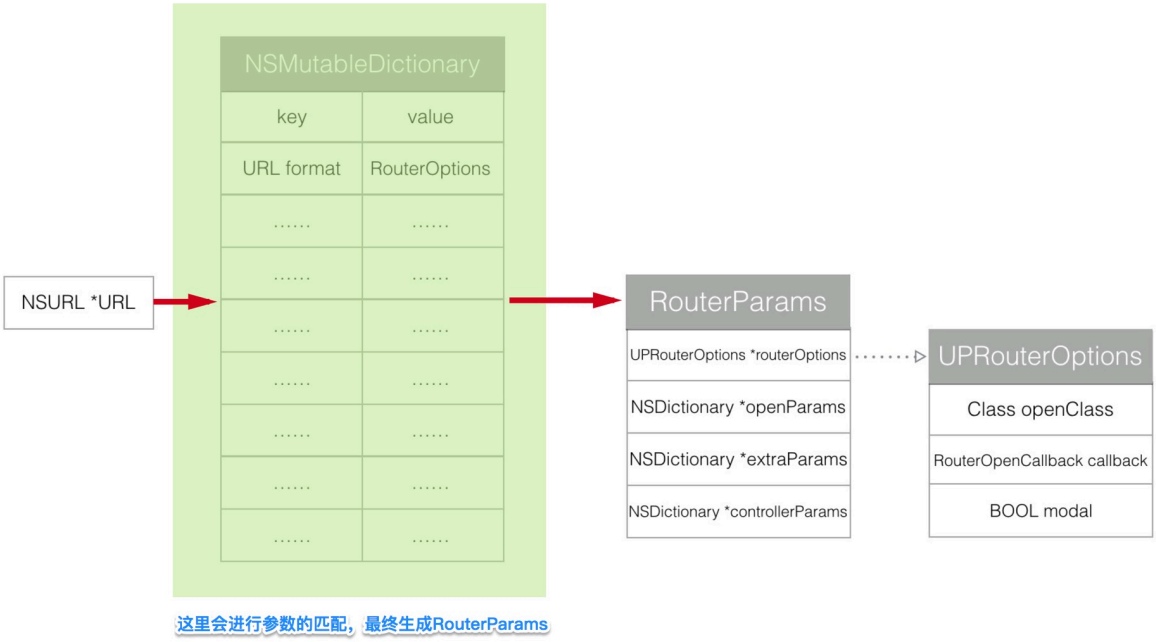
上面这段函数，第一个参数是外部传进来 URL 带有各个入参的分割数组。第二个参数是路由规则分割开的数组。routerComponent 由于规定：号后面才是参数，所以 routerComponent 的第1个位置就是对应的参数名。params 字典里面以参数名为 Key，参数为 Value。

```
 NSDictionary *givenParams = [self paramsForUrlComponents:givenParts
routerUrlComponents:routerParts];
if (givenParams) {
       openParams = [[RouterParams alloc]
initWithRouterOptions:routerOptions openParams:givenParams extraParams:
extraParams];
       *stop = YES;
}
```

最后通过 RouterParams 的初始化方法，把路由规则对应的 UPRouterOptions，上一步封装好的参数字典 givenParams，还有
routerParamsForUrl: extraParams: 方法的第二个入参，这3个参数作为初始化参数，生成了一个 RouterParams。

```
[self.cachedRoutes setObject:openParams forKey:url];
```

最后一步 self.cachedRoutes 的字典里面 Key 为带参数的 URL，Value 是 RouterParams。

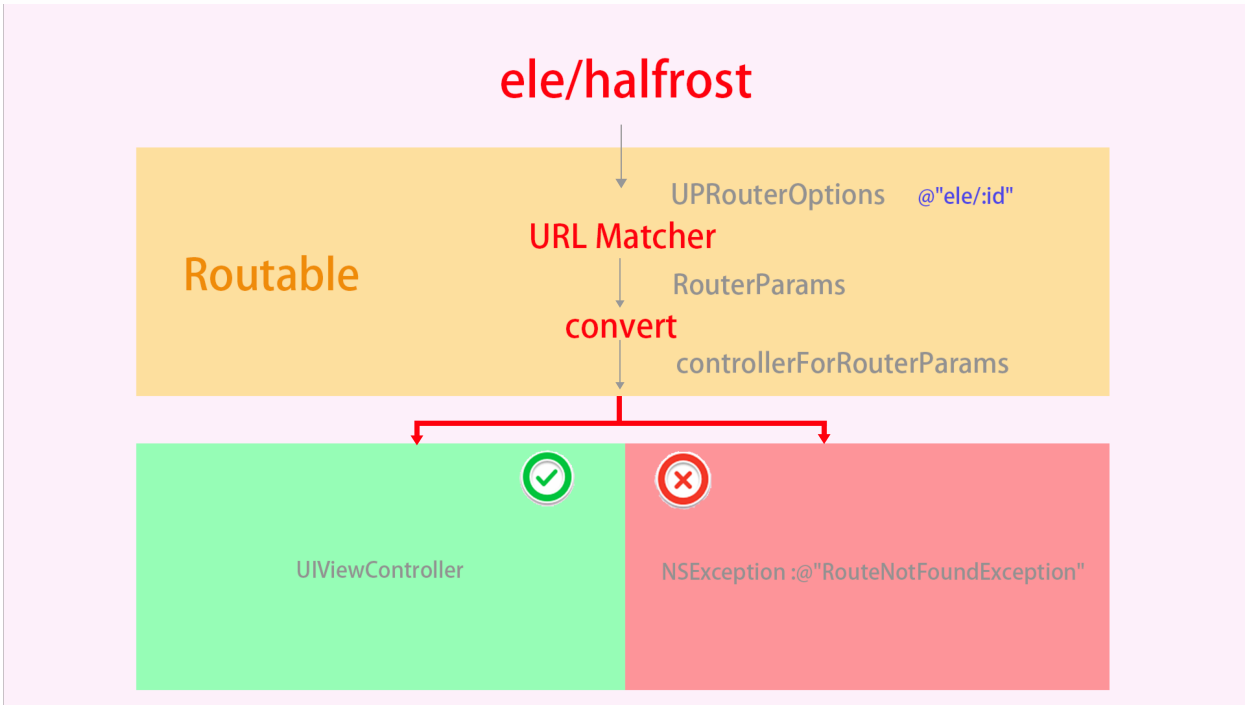


最后将匹配封装出来的 RouterParams 转换成对应的 Controller。

```
- (UIViewController *)controllerForRouterParams:(RouterParams *)params {
    SEL CONTROLLER_CLASS_SELECTOR =
sel_registerName("allocWithRouterParams:");
    SEL CONTROLLER_SELECTOR = sel_registerName("initWithRouterParams:");
    UIViewController *controller = nil;
    Class controllerClass = params.routerOptions.openClass;
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Warc-performSelector-leaks"
    if ([controllerClass respondsToSelector:CONTROLLER_CLASS_SELECTOR]) {
        controller = [controllerClass
performSelector:CONTROLLER_CLASS_SELECTOR withObject:[params
controllerParams]];
    }
    else if ([params.routerOptions.openClass
instancesRespondToSelector:CONTROLLER_SELECTOR]) {
        controller = [[params.routerOptions.openClass alloc]
performSelector:CONTROLLER_SELECTOR withObject:[params controllerParams]];
    }
#pragma clang diagnostic pop
    if (!controller) {
        if (_ignoresExceptions) {
            return controller;
        }
        @throw [NSException exceptionWithName:@"RoutableInitializerNotFound"
                                       reason:[NSString
stringWithFormat:INVALID_CONTROLLER_FORMAT,
NSStringFromClass(controllerClass),
NSStringFromSelector(CONTROLLER_CLASS_SELECTOR),
NSStringFromSelector(CONTROLLER_SELECTOR)]
                                     userInfo:nil];
    }

    controller.modalTransitionStyle = params.routerOptions.transitionStyle;
    controller.modalPresentationStyle =
params.routerOptions.presentationStyle;
    return controller;
}
```

如果 Controller 是一个类，那么就调用 allocWithRouterParams: 方法去初始化。如果 Controller 已经是一个实例了，那么就调用 initWithRouterParams: 方法去初始化。

将 Routable 的大致流程图解如下：



（3）HHRouter Star 1277

这是布丁动画的一个 Router，灵感来自于 ABRouter 和 Routable iOS。

先来看看 HHRouter 的 Api。它提供的方法非常清晰。

ViewController 提供了2个方法。map是用来设置路由规则，matchController 是用来匹配路由规则的，匹配争取之后返回对应的 UIViewController。

```
- (void)map:(NSString *)route toControllerClass:(Class)controllerClass;
- (UIViewController *)matchController:(NSString *)route;
```

block 闭包提供了三个方法，map 也是设置路由规则，matchBlock：是用来匹配路由，找到指定的 block，但是不会调用该 block。callBlock: 是找到指定的 block，找到以后就立即调用。

```
- (void)map:(NSString *)route toBlock:(HHRouterBlock)block;

- (HHRouterBlock)matchBlock:(NSString *)route;
- (id)callBlock:(NSString *)route;
```

matchBlock: 和 callBlock: 的区别就在于前者不会自动调用闭包。所以 matchBlock: 方法找到对应的 block 之后，如果想调用，需要手动调用一次。

除去上面这些方法，HHRouter 还为我们提供了一个特殊的方法。

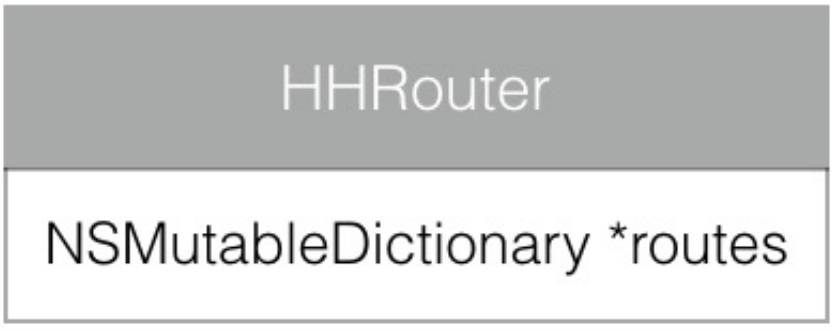
```
- (HHRouteType)canRoute:(NSString *)route;
```

这个方法就是用来找到执行路由规则对应的 RouteType，RouteType 总共就3种:

```
typedef NS_ENUM (NSInteger, HHRouteType) {
    HHRouteTypeNone = 0,
    HHRouteTypeViewController = 1,
    HHRouteTypeBlock = 2
};
```

再来看看 HHRouter 是如何管理路由规则的。整个 HHRouter 就是由一个 NSMutableDictionary *routes 控制的。

```
@interface HHRouter ()
@property (strong, nonatomic) NSMutableDictionary *routes;
@end
```



别看只有这一个看似“简单”的字典数据结构，但是 HHRouter 路由设计的还是很精妙的。

```
- (void)map:(NSString *)route toBlock:(HHRouterBlock)block
{
    NSMutableDictionary *subRoutes = [self subRoutesToRoute:route];
    subRoutes[@"_"] = [block copy];
}

- (void)map:(NSString *)route toControllerClass:(Class)controllerClass
{
    NSMutableDictionary *subRoutes = [self subRoutesToRoute:route];
    subRoutes[@"_"] = controllerClass;
}
```

上面两个方法分别是 block 闭包和 ViewController 设置路由规则调用的方法实体。不管是 ViewController 还是 block 闭包，设置规则的时候都会调用 subRoutesToRoute: 方法。

```
- (NSMutableDictionary *)subRoutesToRoute:(NSString *)route
{
    NSArray *pathComponents = [self pathComponentsFromRoute:route];

    NSInteger index = 0;
    NSMutableDictionary *subRoutes = self.routes;

    while (index < pathComponents.count) {
        NSString *pathComponent = pathComponents[index];
        if (![subRoutes objectForKey:pathComponent]) {
            subRoutes[pathComponent] = [[NSMutableDictionary alloc] init];
        }
        subRoutes = subRoutes[pathComponent];
        index++;
    }

    return subRoutes;
}
```

上面这段函数就是来构造路由匹配规则的字典。

举个例子：

```
[[HHRouter shared] map:@"/user/:userId/"
         toControllerClass:[UserViewController class]];
[[HHRouter shared] map:@"/story/:storyId/"
         toControllerClass:[StoryViewController class]];
[[HHRouter shared] map:@"/user/:userId/story/?a=0"
         toControllerClass:[StoryListViewController class]];
```

设置3条规则以后，按照上面构造路由匹配规则的字典的方法，该路由规则字典就会变成这个样子：

```
{
    story =     {
        ":storyId" =         {
            "_" = StoryViewController;
        };
    };
    user =     {
        ":userId" =         {
            "_" = UserViewController;
            story =             {
                "_" = StoryListViewController;
            };
        };
    };
}
```

路由规则字典生成之后，等到匹配的时候就会遍历这个字典。

假设这时候有一条路由过来：

```
  [[[HHRouter shared] matchController:@"hhrouter20://user/1/"] class],
```

HHRouter 对这条路由的处理方式是先匹配前面的 scheme，如果连 scheme 都不正确的话，会直接导致后面匹配失败。

然后再进行路由匹配，最后生成的参数字典如下：

```
{
    "controller_class" = UserViewController;
    route = "/user/1/";
    userId = 1;
}
```

具体的路由参数匹配的函数在

```
- (NSDictionary *)paramsInRoute:(NSString *)route
```

这个方法里面实现的。这个方法就是按照路由匹配规则，把传进来的URL的参数都一一解析出来，带？号的也都会解析成字典。这个方法没什么难度，就不在赘述了。

ViewController 的字典里面默认还会加上2项：

```
"controller_class" =
route =
```

route 里面都会保存传过来的完整的 URL。

如果传进来的路由后面带访问字符串呢？那我们再来看看：

```
[[HHRouter shared] matchController:@"/user/1/?a=b&c=d"]
```

那么解析出所有的参数字典会是下面的样子：

```
{
    a = b;
    c = d;
    "controller_class" = UserViewController;
    route = "/user/1/?a=b&c=d";
    userId = 1;
}
```

同理，如果是一个 block 闭包的情况呢？

还是先添加一条 block 闭包的路由规则：

```
[[HHRouter shared] map:@"/user/add/"
                   toBlock:^id(NSDictionary* params) {
                   }];
```

这条规则对应的会生成一个路由规则的字典。

```
{
    story =     {
        ":storyId" =         {
            "_" = StoryViewController;
        };
    };
    user =     {
        ":userId" =         {
            "_" = UserViewController;
            story =             {
                "_" = StoryListViewController;
            };
        };
        add =         {
            "_" = "<__NSMallocBlock__: 0x600000240480>";
        };
    };
}
```

注意 "_" 后面跟着是一个 block。

匹配 block 闭包的方式有两种。

```
// 1.第一种方式匹配到对应的block之后，还需要手动调用一次闭包。
    HHRouterBlock block = [[HHRouter shared] matchBlock:@"/user/add/?
a=1&b=2"];
    block(nil);


// 2.第二种方式匹配block之后自动会调用改闭包。
    [[HHRouter shared] callBlock:@"/user/add/?a=1&b=2"];
```

匹配出来的参数字典是如下：

```
{
    a = 1;
    b = 2;
    block = "<__NSMallocBlock__: 0x600000056b90>";
    route = "/user/add/?a=1&b=2";
}
```

block 的字典里面会默认加上下面这2项：

```
block =
route =
```

route 里面都会保存传过来的完整的 URL。

生成的参数字典最终会被绑定到 ViewController 的 Associated Object 关联对象上。

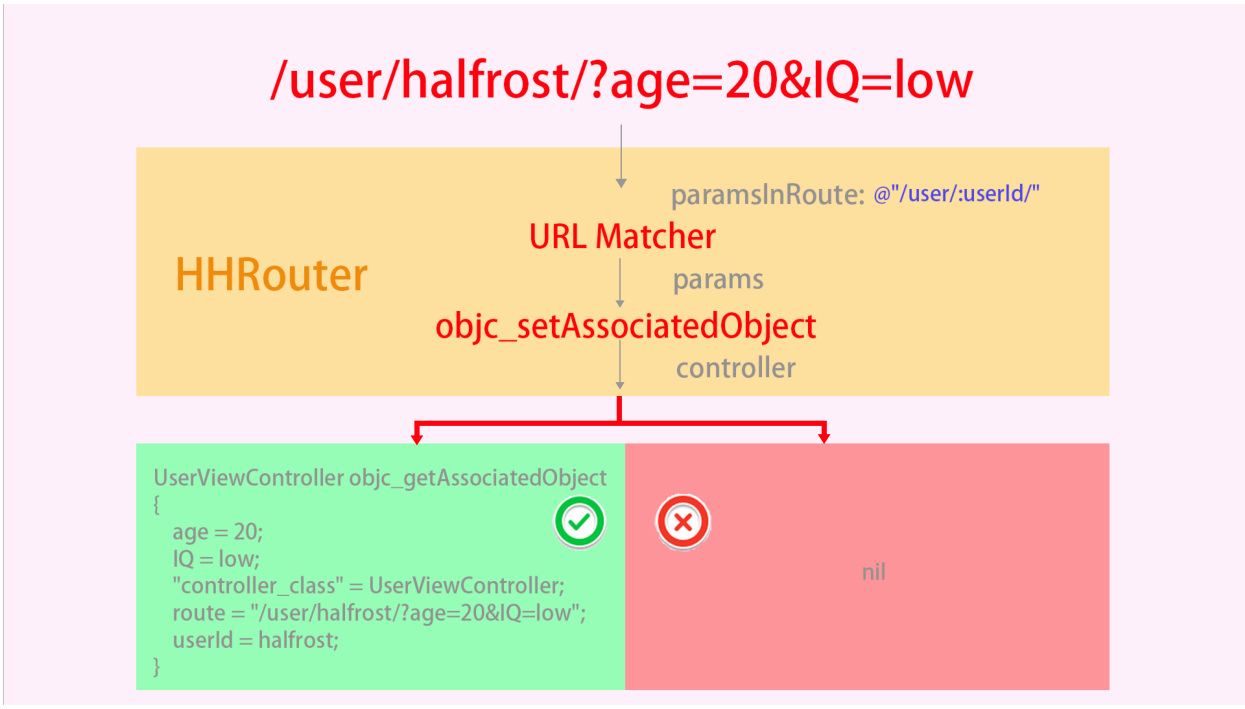
```
- (void)setParams:(NSDictionary *)paramsDictionary
{
    objc_setAssociatedObject(self, &kAssociatedParamsObjectKey,
paramsDictionary, OBJC_ASSOCIATION_RETAIN_NONATOMIC);
}

- (NSDictionary *)params
{
    return objc_getAssociatedObject(self, &kAssociatedParamsObjectKey);
}
```

这个绑定的过程是在 match 匹配完成的时候进行的。

```
- (UIViewController *)matchController:(NSString *)route
{
    NSDictionary *params = [self paramsInRoute:route];
    Class controllerClass = params[@"controller_class"];

    UIViewController *viewController = [[controllerClass alloc] init];

    if ([viewController respondsToSelector:@selector(setParams:)]) {
        [viewController performSelector:@selector(setParams:)
                             withObject:[params copy]];
    }
    return viewController;
}
```

最终得到的 ViewController 也是我们想要的。相应的参数都在它绑定的 params 属性的字典里面。

将上述过程图解出来，如下：



（4）MGJRouter Star 633

这是蘑菇街的一个路由的方法。

这个库的由来：

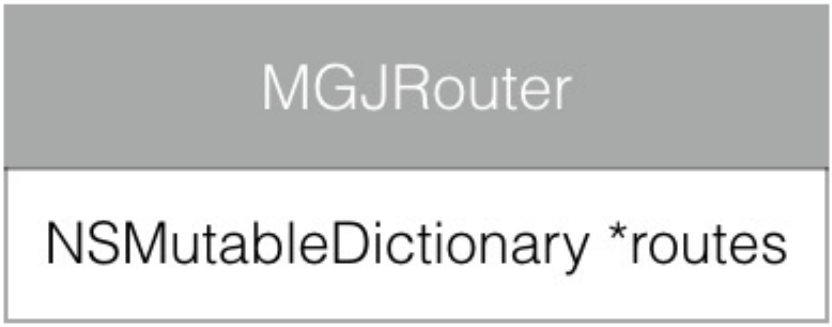
JLRoutes 的问题主要在于查找 URL 的实现不够高效，通过遍历而不是匹配。还有就是功能偏多。

HHRouter 的 URL 查找是基于匹配，所以会更高效，MGJRouter 也是采用的这种方法，但它跟 ViewController 绑定地过于紧密，一定程度上降低了灵活性。

于是就有了 MGJRouter。

从数据结构来看，MGJRouter 还是和 HHRouter 一模一样的。

```
@interface MGJRouter ()
@property (nonatomic) NSMutableDictionary *routes;
@end
```



那么我们就来看看它对 HHRouter 做了哪些优化改进。

1.MGJRouter 支持 openURL 时，可以传一些 userinfo 过去

```
[MGJRouter openURL:@"mgj://category/travel" withUserInfo:@{@"user_id":
@1900} completion:nil];
```

这个对比 HHRouter，仅仅只是写法上的一个语法糖，在 HHRouter 中虽然不支持带字典的参数，但是在URL后面可以用 URL Query Parameter 来弥补。

```
    if (parameters) {
        MGJRouterHandler handler = parameters[@"block"];
        if (completion) {
            parameters[MGJRouterParameterCompletion] = completion;
        }
        if (userInfo) {
            parameters[MGJRouterParameterUserInfo] = userInfo;
        }
        if (handler) {
            [parameters removeObjectForKey:@"block"];
            handler(parameters);
        }
    }
```

MGJRouter 对 userInfo 的处理是直接把它封装到 Key = MGJRouterParameterUserInfo 对应的 Value 里面。

2.支持中文的URL

```
    [parameters enumerateKeysAndObjectsUsingBlock:^(id key, NSString *obj,
BOOL *stop) {
        if ([obj isKindOfClass:[NSString class]]) {
            parameters[key] = [obj
stringByReplacingPercentEscapesUsingEncoding:NSUTF8StringEncoding];
        }
    }];
```

这里就是需要注意一下编码。

3.定义一个全局的 URL Pattern 作为 Fallback

这一点是模仿的 JLRoutes 的匹配不到会自动降级到 global 的思想。

```
    if (parameters) {
        MGJRouterHandler handler = parameters[@"block"];
        if (handler) {
            [parameters removeObjectForKey:@"block"];
            handler(parameters);
        }
    }
```

parameters 字典里面会先存储下一个路由规则，存在 block 闭包中，在匹配的时候会取出这个 handler，降级匹配到这个闭包中，进行最终的处理。

4.当 OpenURL 结束时，可以执行 Completion Block

在 MGJRouter 里面，作者对原来的 HHRouter 字典里面存储的路由规则的结构进行了改造。

```
NSString *const MGJRouterParameterURL = @"MGJRouterParameterURL";
NSString *const MGJRouterParameterCompletion =
@"MGJRouterParameterCompletion";
NSString *const MGJRouterParameterUserInfo = @"MGJRouterParameterUserInfo";
```

这3个 key 会分别保存一些信息：

MGJRouterParameterURL 保存的传进来的完整的 URL 信息。
MGJRouterParameterCompletion 保存的是 completion 闭包。
MGJRouterParameterUserInfo 保存的是 UserInfo 字典。

举个例子：

```
    [MGJRouter registerURLPattern:@"ele://name/:name"
toHandler:^(NSDictionary *routerParameters) {
        void (^completion)(NSString *) =
routerParameters[MGJRouterParameterCompletion];
        if (completion) {
            completion(@"完成了");
        }
    }];

    [MGJRouter openURL:@"ele://name/halfrost/?age=20"
withUserInfo:@{@"user_id": @1900} completion:^(id result) {
        NSLog(@"result = %@",result);
    }];
```

上面的 URL 会匹配成功，那么生成的参数字典结构如下：

```
{
    MGJRouterParameterCompletion = "<__NSGlobalBlock__: 0x107ffe680>";
    MGJRouterParameterURL = "ele://name/halfrost/?age=20";
    MGJRouterParameterUserInfo =     {
        "user_id" = 1900;
    };
    age = 20;
    block = "<__NSMallocBlock__: 0x608000252120>";
    name = halfrost;
}
```

5.可以统一管理 URL

这个功能非常有用。

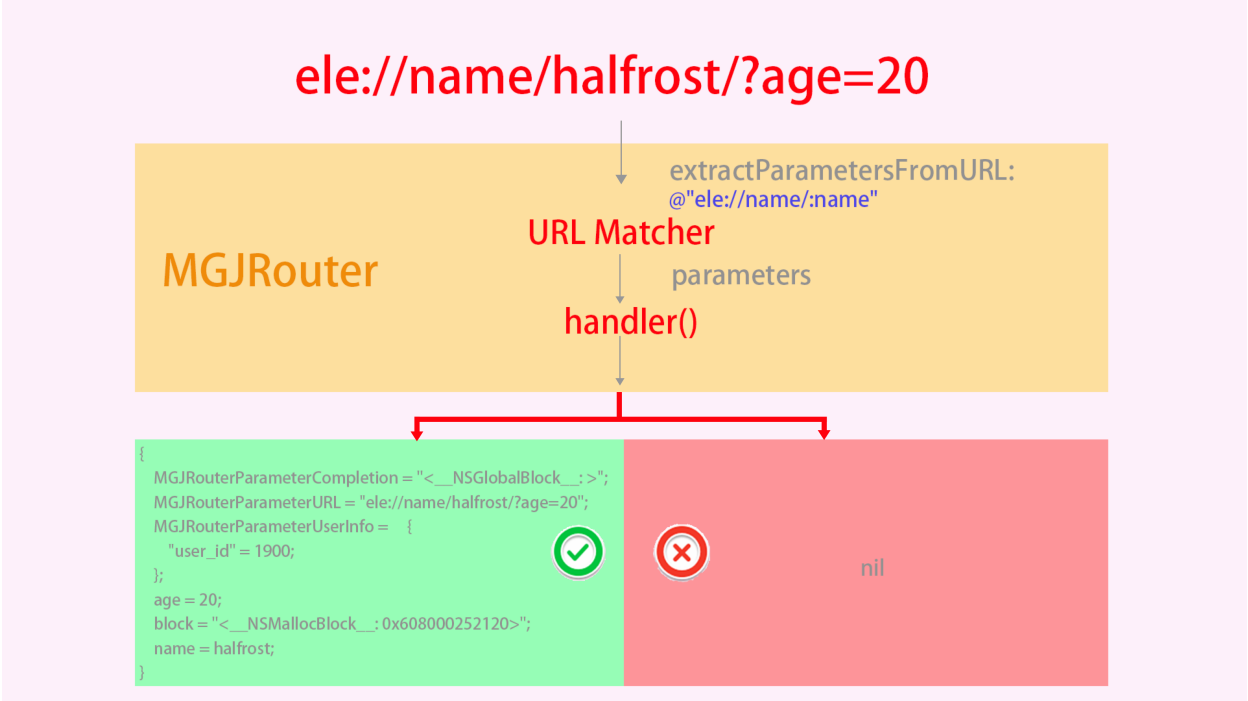
URL 的处理一不小心，就容易散落在项目的各个角落，不容易管理。比如注册时的 pattern 是 mgj://beauty/:id，然后 open 时就是 mgj://beauty/123，这样到时候 url 有改动，处理起来就会很麻烦，不好统一管理。

所以 MGJRouter 提供了一个类方法来处理这个问题。

```
#define TEMPLATE_URL @"qq://name/:name"

[MGJRouter registerURLPattern:TEMPLATE_URL  toHandler:^(NSDictionary
*routerParameters) {
    NSLog(@"routerParameters[name]:%@", routerParameters[@"name"]); //
halfrost
}];

[MGJRouter openURL:[MGJRouter generateURLWithPattern:TEMPLATE_URL
parameters:@[@"halfrost"]]];
}
```

generateURLWithPattern: 函数会对我们定义的宏里面的所有的:进行替换，替换成后面的字符串数组，依次赋值。

将上述过程图解出来，如下：

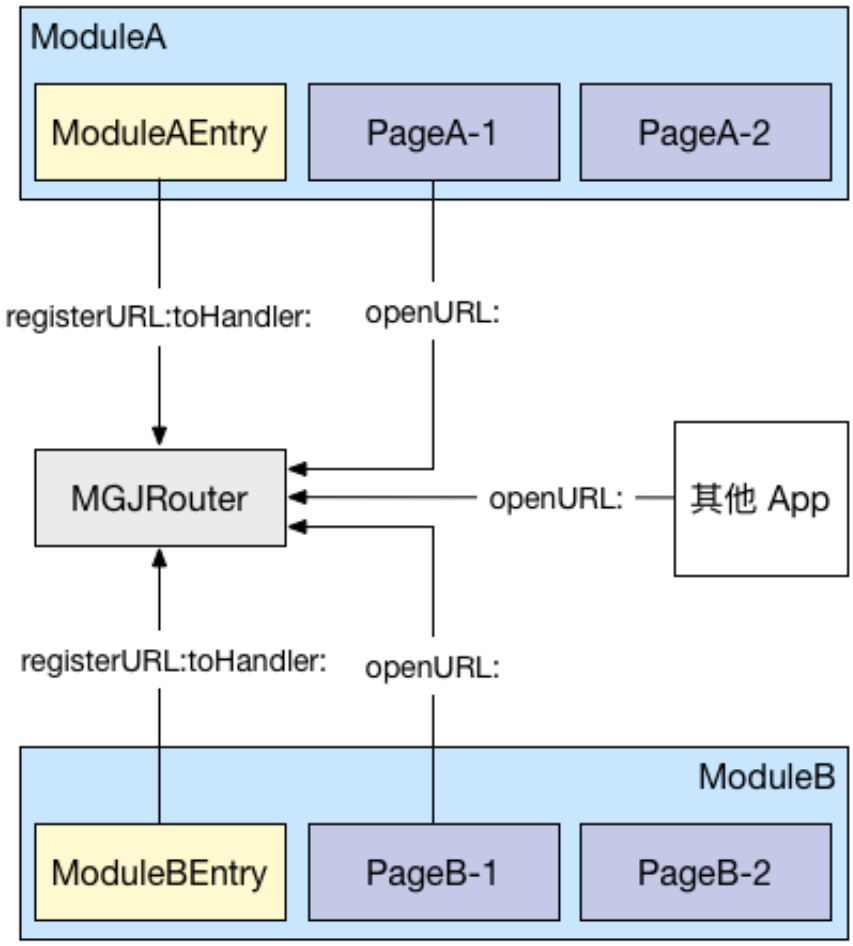


蘑菇街为了区分开页面间调用和组件间调用，于是想出了一种新的方法。用 Protocol 的方法来进行组件间的调用。

每个组件之间都有一个 Entry，这个 Entry，主要做了三件事：

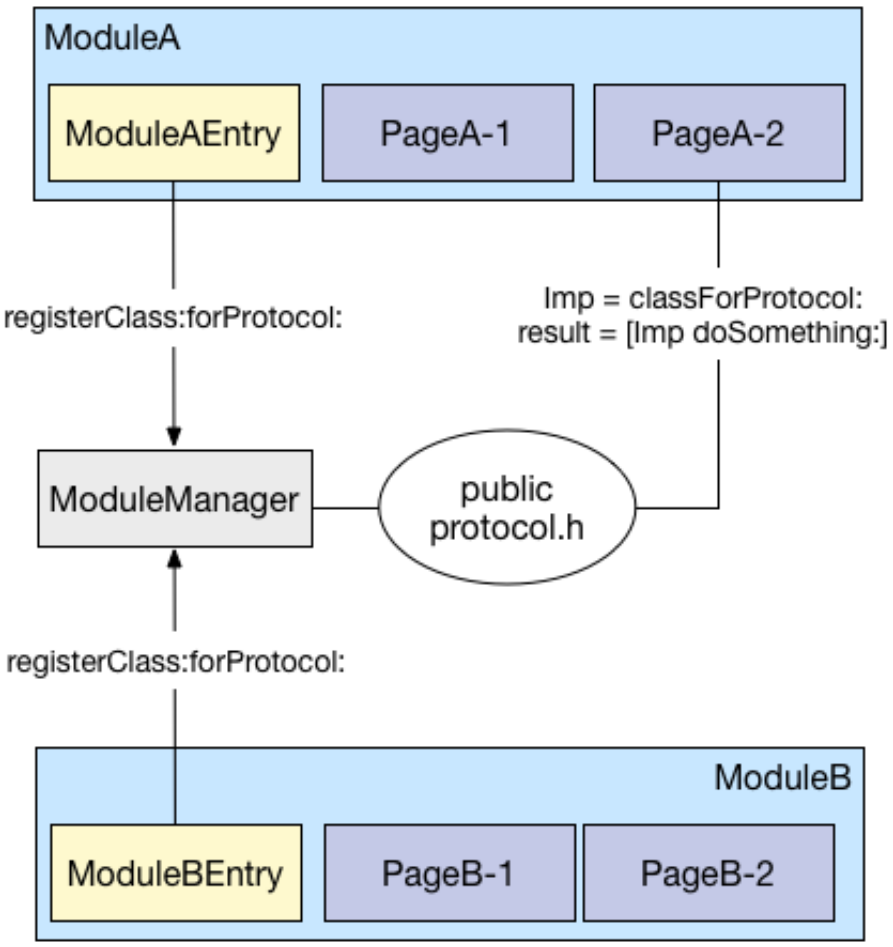
1. 注册这个组件关心的 URL
2. 注册这个组件能够被调用的方法/属性
3. 在 App 生命周期的不同阶段做不同的响应

页面间的 openURL 调用就是如下的样子：



每个组件间都会向 MGJRouter 注册，组件间相互调用或者是其他的 App 都可以通过 openURL: 方法打开一个界面或者调用一个组件。

在组件间的调用，蘑菇街采用了 Protocol 的方式。



[ModuleManager registerClass:ClassA forProtocol:ProtocolA] 的结果就是在 MM 内部维护的 dict 里新加了一个映射关系。

[ModuleManager classForProtocol:ProtocolA] 的返回结果就是之前在 MM 内部 dict 里 protocol 对应的 class，使用方不需要关心这个 class 是个什么东东，反正实现了 ProtocolA 协议，拿来用就行。

这里需要有一个公共的地方来容纳这些 public protocl，也就是图中的 PublicProtocl.h。

我猜测，大概实现可能是下面的样子：

```
@interface ModuleProtocolManager : NSObject

+ (void)registServiceProvide:(id)provide forProtocol:(Protocol*)protocol;
+ (id)serviceProvideForProtocol:(Protocol *)protocol;

@end
```

然后这个是一个单例，在里面注册各个协议：

```
@interface ModuleProtocolManager ()

@property (nonatomic, strong) NSMutableDictionary *serviceProvideSource;
@end

@implementation ModuleProtocolManager

+ (ModuleProtocolManager *)sharedInstance
{
    static ModuleProtocolManager * instance;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        instance = [[self alloc] init];
    });
    return instance;
}

- (instancetype)init
{
    self = [super init];
    if (self) {
        _serviceProvideSource = [[NSMutableDictionary alloc] init];
    }
    return self;
}

+ (void)registServiceProvide:(id)provide forProtocol:(Protocol*)protocol
{
    if (provide == nil || protocol == nil)
        return;
    [[self sharedInstance].serviceProvideSource setObject:provide
forKey:NSStringFromProtocol(protocol)];
}

+ (id)serviceProvideForProtocol:(Protocol *)protocol
{
    return [[self sharedInstance].serviceProvideSource
objectForKey:NSStringFromProtocol(protocol)];
}
```

在 ModuleProtocolManager 中用一个字典保存每个注册的 protocol。现在再来猜猜 ModuleEntry 的实现。

```
#import <Foundation/Foundation.h>
#import <UIKit/UIKit.h>

@protocol DetailModuleEntryProtocol <NSObject>

@required;
- (UIViewController *)detailViewControllerWithId:(NSString*)Id Name:
(NSString *)name;
@end
```

然后每个模块内都有一个和暴露到外面的协议相连接的“接头”。

```
#import <Foundation/Foundation.h>

@interface DetailModuleEntry : NSObject
@end
```

在它的实现中，需要引入3个外部文件，一个是 ModuleProtocolManager，一个是 DetailModuleEntryProtocol，最后一个是所在模块需要跳转或者调用的组件或者页面。

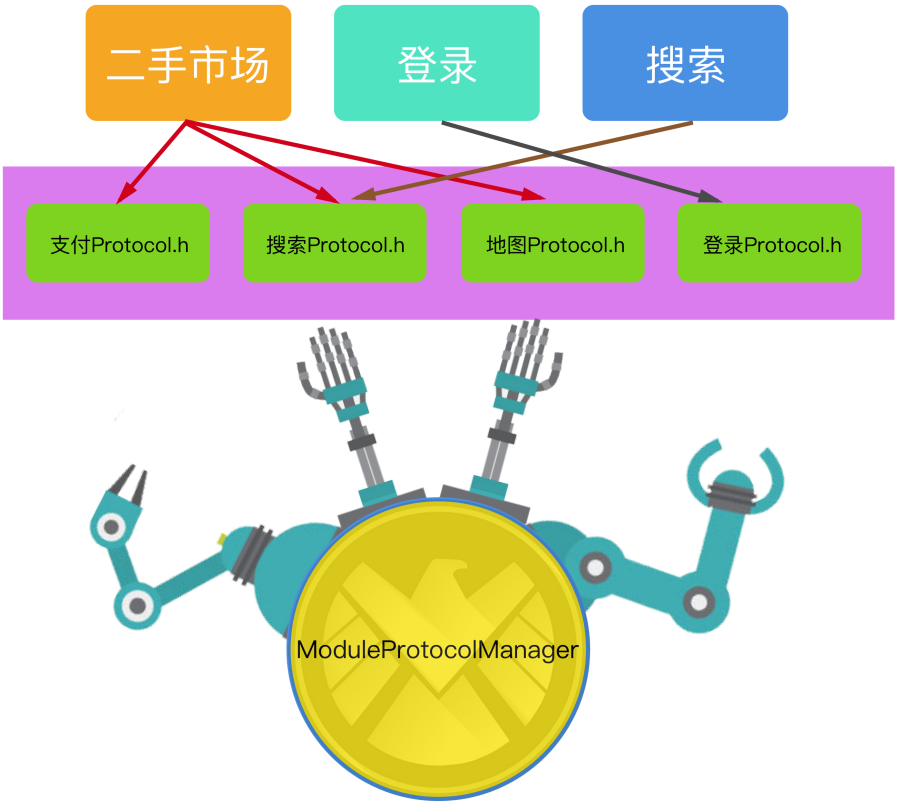
```
#import "DetailModuleEntry.h"

#import <DetailModuleEntryProtocol/DetailModuleEntryProtocol.h>
#import <ModuleProtocolManager/ModuleProtocolManager.h>
#import "DetailViewController.h"

@interface DetailModuleEntry()<DetailModuleEntryProtocol>

@end

@implementation DetailModuleEntry

+ (void)load
{
    [ModuleProtocolManager registServiceProvide:[[self alloc] init]
forProtocol:@protocol(DetailModuleEntryProtocol)];
}

- (UIViewController *)detailViewControllerWithId:(NSString*)Id Name:
(NSString *)name
{
    DetailViewController *detailVC = [[DetailViewController alloc]
initWithId:id Name:name];
    return detailVC;
}

@end
```

至此基于 Protocol 的方案就完成了。如果需要调用某个组件或者跳转某个页面，只要先从 ModuleProtocolManager 的字典里面根据对应的 ModuleEntryProtocol 找到对应的 DetailModuleEntry，找到了 DetailModuleEntry 就是找到了组件或者页面的“入口”了。再把参数传进去即可。

```
- (void)didClickDetailButton:(UIButton *)button
{
    id< DetailModuleEntryProtocol > DetailModuleEntry =
[ModuleProtocolManager
serviceProvideForProtocol:@protocol(DetailModuleEntryProtocol)];
    UIViewController *detailVC = [DetailModuleEntry
detailViewControllerWithId:@“详情界面” Name:@“我的购物车”];
    [self.navigationController pushViewController:detailVC animated:YES];

}
```

这样就可以调用到组件或者界面了。

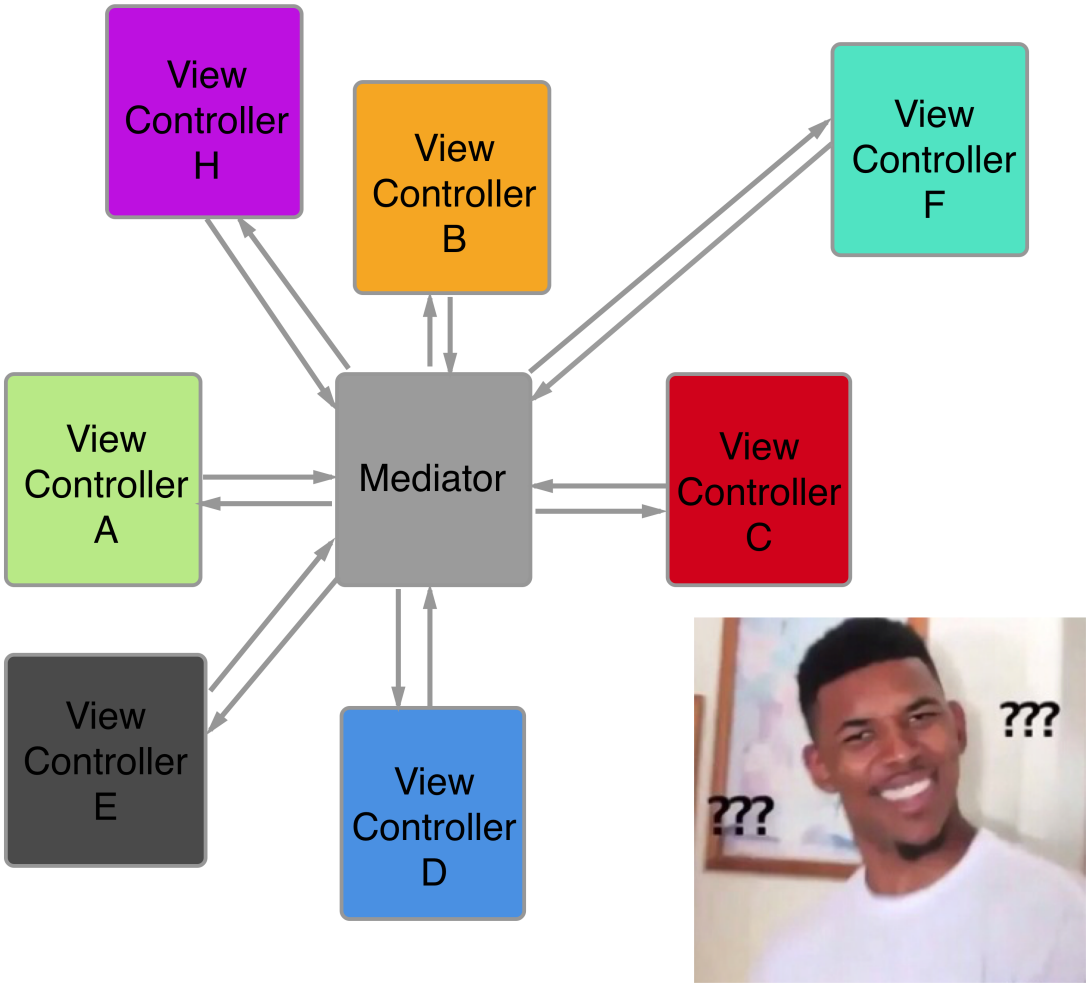
如果组件之间有相同的接口，那么还可以进一步的把这些接口都抽离出来。这些抽离出来的接口变成“元接口”，它们是可以足够支撑起整个组件一层的。



（5）CTMediator Star 803

再来说说 @casatwy 的方案，这方案是基于 Mediator 的。

传统的中间人 Mediator 的模式是这样的：



这种模式每个页面或者组件都会依赖中间者，各个组件之间互相不再依赖，组件间调用只依赖中间者 Mediator，Mediator 还是会依赖其他组件。那么这是最终方案了么?

看看 @casatwy 是怎么继续优化的。

主要思想是利用了 Target-Action 简单粗暴的思想，利用 Runtime 解决解耦的问题。

```
- (id)performTarget:(NSString *)targetName action:(NSString *)actionName
params:(NSDictionary *)params shouldCacheTarget:(BOOL)shouldCacheTarget
{

    NSString *targetClassString = [NSString stringWithFormat:@"Target_%@",
targetName];
    NSString *actionString = [NSString stringWithFormat:@"Action_%@:",
actionName];
    Class targetClass;

    NSObject *target = self.cachedTarget[targetClassString];
    if (target == nil) {
        targetClass = NSClassFromString(targetClassString);
        target = [[targetClass alloc] init];
    }
```

```
    SEL action = NSSelectorFromString(actionString);

    if (target == nil) {
        // 这里是处理无响应请求的地方之一，这个demo做得比较简单，如果没有可以响应的target，就直接return了。实际开发过程中是可以事先给一个固定的target专门用于在这个时候顶上，然后处理这种请求的
        return nil;
    }

    if (shouldCacheTarget) {
        self.cachedTarget[targetClassString] = target;
    }

    if ([target respondsToSelector:action]) {
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Warc-performSelector-leaks"
        return [target performSelector:action withObject:params];
#pragma clang diagnostic pop
    } else {
        // 有可能target是Swift对象
        actionString = [NSString stringWithFormat:@"Action_%@WithParams:", actionName];
        action = NSSelectorFromString(actionString);
        if ([target respondsToSelector:action]) {
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Warc-performSelector-leaks"
            return [target performSelector:action withObject:params];
#pragma clang diagnostic pop
        } else {
            // 这里是处理无响应请求的地方，如果无响应，则尝试调用对应target的notFound方法统一处理
            SEL action = NSSelectorFromString(@"notFound:");
            if ([target respondsToSelector:action]) {
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Warc-performSelector-leaks"
                return [target performSelector:action withObject:params];
#pragma clang diagnostic pop
            } else {
                // 这里也是处理无响应请求的地方，在notFound都没有的时候，这个demo是直接return了。实际开发过程中，可以用前面提到的固定的target顶上的。
                [self.cachedTarget removeObjectForKey:targetClassString];
                return nil;
            }
        }
    }
}
```

targetName 就是调用接口的 Object，actionName 就是调用方法的 SEL，params 是参数，shouldCacheTarget 代表是否需要缓存，如果需要缓存就把 target 存起来，Key 是 targetClassString，Value 是 target。

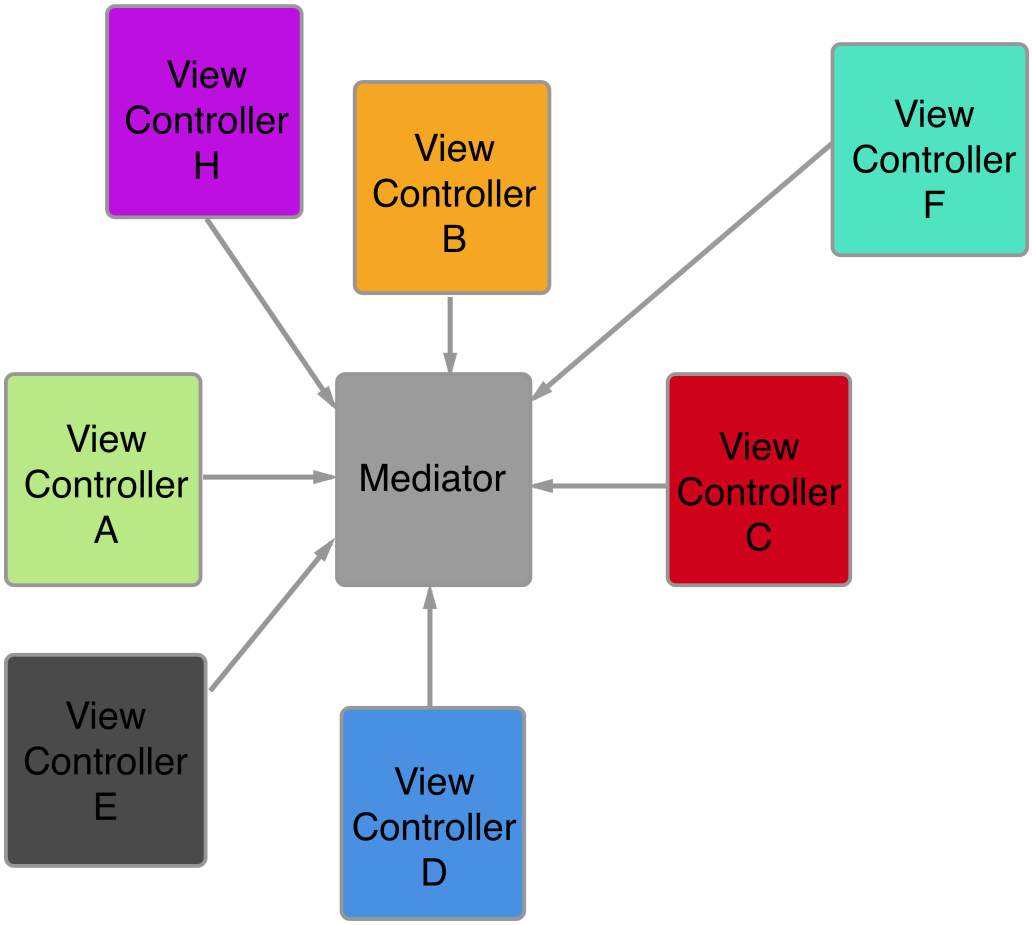
通过这种方式进行改造的，外面调用的方法都很统一，都是调用 performTarget: action: params: shouldCacheTarget:。第三个参数是一个字典，这个字典里面可以传很多参数，只要 Key-Value 写好就可以了。处理错误的方式也统一在一个地方了，target 没有，或者是 target 无法响应相应的方法，都可以在 Mediator 这里进行统一出错处理。

但是在实际开发过程中，不管是界面调用，组件间调用，在 Mediator 中需要定义很多方法。于是作者又想出了建议我们用 Category 的方法，对 Mediator 的所有方法进行拆分，这样就就可以不会导致 Mediator 这个类过于庞大了。

```
- (UIViewController *)CTMediator_viewControllerForDetail
{
    UIViewController *viewController = [self
performTarget:kCTMediatorTargetA

action:kCTMediatorActionNativFetchDetailViewController

params:@{@"key":@"value"}
                                          shouldCacheTarget:NO
                                         ];
    if ([viewController isKindOfClass:[UIViewController class]]) {
        // view controller 交付出去之后，可以由外界选择是push还是present
        return viewController;
    } else {
        // 这里处理异常场景，具体如何处理取决于产品
        return [[UIViewController alloc] init];
    }
}



- (void)CTMediator_presentImage:(UIImage *)image
{
    if (image) {
        [self performTarget:kCTMediatorTargetA
                     action:kCTMediatorActionNativePresentImage
                     params:@{@"image":image}
          shouldCacheTarget:NO];
    } else {
        // 这里处理image为nil的场景，如何处理取决于产品
        [self performTarget:kCTMediatorTargetA
                     action:kCTMediatorActionNativeNoImage
                     params:@{@"image":[UIImage imageNamed:@"noImage"]}
          shouldCacheTarget:NO];
    }
}
```

把这些具体的方法一个个的都写在 Category 里面就好了，调用的方式都非常的一致，都是调用 performTarget: action: params: shouldCacheTarget: 方法。

最终去掉了中间者 Mediator 对组件的依赖，各个组件之间互相不再依赖，组件间调用只依赖中间者 Mediator，Mediator 不依赖其他任何组件。



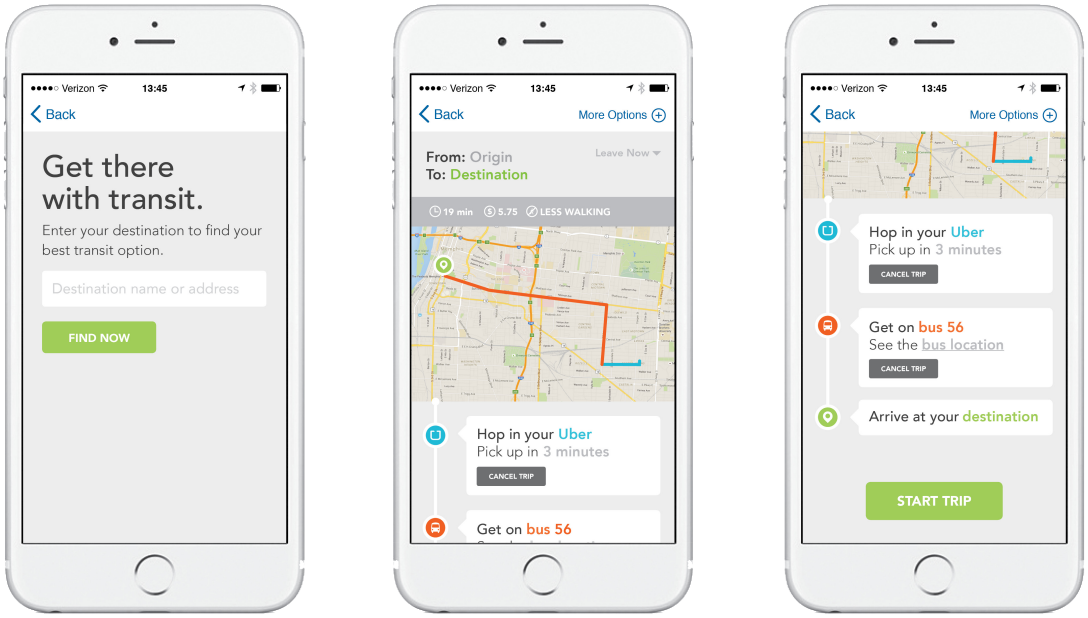
（6）一些并没有开源的方案

除了上面开源的路由方案，还有一些并没有开源的设计精美的方案。这里可以和大家一起分析交流一下。

这个方案是 Uber 骑手App 的一个方案。

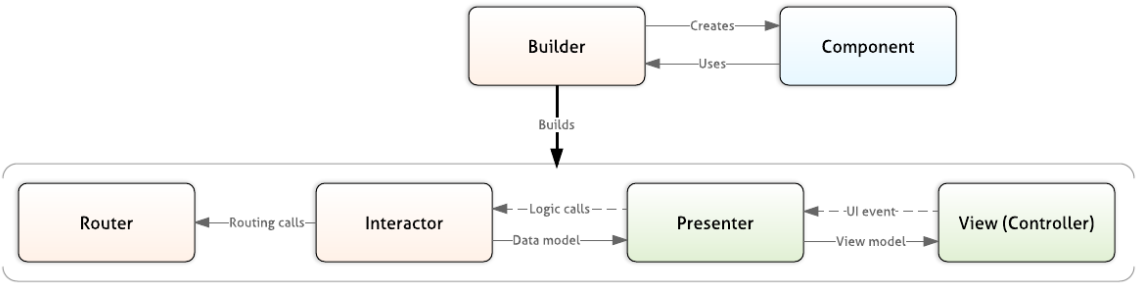
Uber 在发现 MVC 的一些弊端之后：比如动辄上万行巨胖无比的 VC，无法进行单元测试等缺点后，于是考虑把架构换成 VIPER。但是 VIPER 也有一定的弊端。因为它的 iOS 特定的结构，意味着 iOS 必须为 Android 做出一些妥协的权衡。以视图为驱动的应用程序逻辑，代表应用程序状态由视图驱动，整个应用程序都锁定在视图树上。由操作应用程序状态所关联的业务逻辑的改变，就必须经过 Presenter。因此会暴露业务逻辑。最终导致了视图树和业务树进行了紧紧的耦合。这样想实现一个紧紧只有业务逻辑的 Node 节点或者紧紧只有视图逻辑的Node节点就非常的困难了。

通过改进 VIPER 架构，吸收其优秀的特点，改进其缺点，就形成了 Uber 骑手App 的全新架构——Riblets(肋骨)。



在这个新的架构中，即使是相似的逻辑也会被区分成很小很小，相互独立，可以单独进行测试的组件。每个组件都有非常明确的用途。使用这些一小块一小块的 Riblets(肋骨)，最终把整个 App 拼接成一颗 Riblets(肋骨)树。

通过抽象，一个 Riblets(肋骨) 被定义成一下6个更小的组件，这些组件各自有各自的职责。通过一个 Riblets(肋骨) 进一步的抽象业务逻辑和视图逻辑。



一个 Riblets(肋骨) 被设计成这样，那和之前的 VIPER 和 MVC 有什么区别呢？最大的区别在路由上面。

Riblets(肋骨) 内的 Router 不再是视图逻辑驱动的，现在变成了业务逻辑驱动。这一重大改变就导致了整个 App 不再是由表现形式驱动，现在变成了由数据流驱动。

每一个 Riblet 都是由一个路由 Router，一个关联器 Interactor，一个构造器 Builder 和它们相关的组件构成的。所以它的命名（Router - Interactor - Builder，Rib）也由此得来。当然还可以有可选的展示器 Presenter 和视图 View。路由 Router 和关联器 Interactor 处理业务逻辑，展示器 Presenter 和视图 View 处理视图逻辑。

重点分析一下 Riblet 里面路由的职责。

1.路由的职责

在整个 App 的结构树中，路由的职责是用来关联和取消关联其他子 Riblet 的。至于决定是由关联器 Interactor 传递过来的。在状态转换过程中，关联和取消关联子 Riblet 的时候，路由也会影响到关联器 Interactor 的生命周期。路由只包含2个业务逻辑：

1.提供关联和取消关联其他路由的方法。
2.在多个孩子之间决定最终状态的状态转换逻辑。

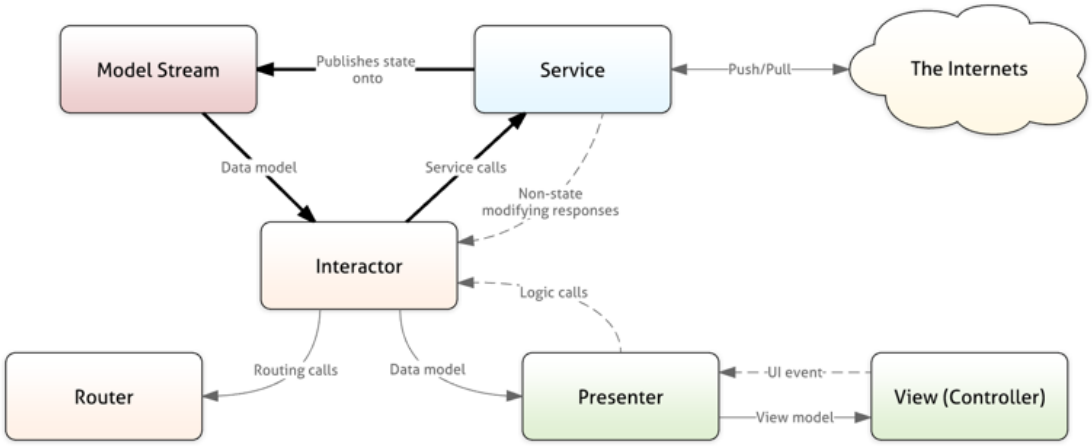
2.拼装

每一个 Riblets 只有一对 Router 路由和 Interactor 关联器。但是它们可以有多对视图。Riblets 只处理业务逻辑，不处理视图相关的部分。Riblets 可以拥有单一的视图（一个 Presenter 展示器和一个 View 视图），也可以拥有多个视图（一个 Presenter 展示器和多个 View 视图，或者多个 Presenter 展示器和多个 View 视图），甚至也可以能没有视图（没有 Presenter 展示器也没有 View 视图）。这种设计可以有助于业务逻辑树的构建，也可以和视图树做到很好的分离。

举个例子，骑手的 Riblet 是一个没有视图的 Riblet，它用来检查当前用户是否有一个激活的路线。如果骑手确定了路线，那么这个 Riblet 就会关联到路线的 Riblet 上面。路线的 Riblet 会在地图上显示出路线图。如果没有确定路线，骑手的 Riblet 就会被关联到请求的 Riblet 上。请求的 Riblet 会在屏幕上显示等待被呼叫。像骑手的 Riblet 这样没有任何视图逻辑的 Riblet，它分开了业务逻辑，在驱动 App 和支撑模块化架构起了重大作用。

3.Riblets 是如何工作的

Riblet 中的数据流



在这个新的架构中，数据流动是单向的。Data 数据流从 service 服务流到 Model Stream 生成 Model 流。Model 流再从 Model Stream 流动到 Interactor 关联器。Interactor 关联器，scheduler 调度器，远程推送都可以想 Service 触发变化来引起 Model Stream 的改动。Model Stream 生成不可改动的 models。这个强制的要求就导致关联器只能通过 Service 层改变 App 的状态。

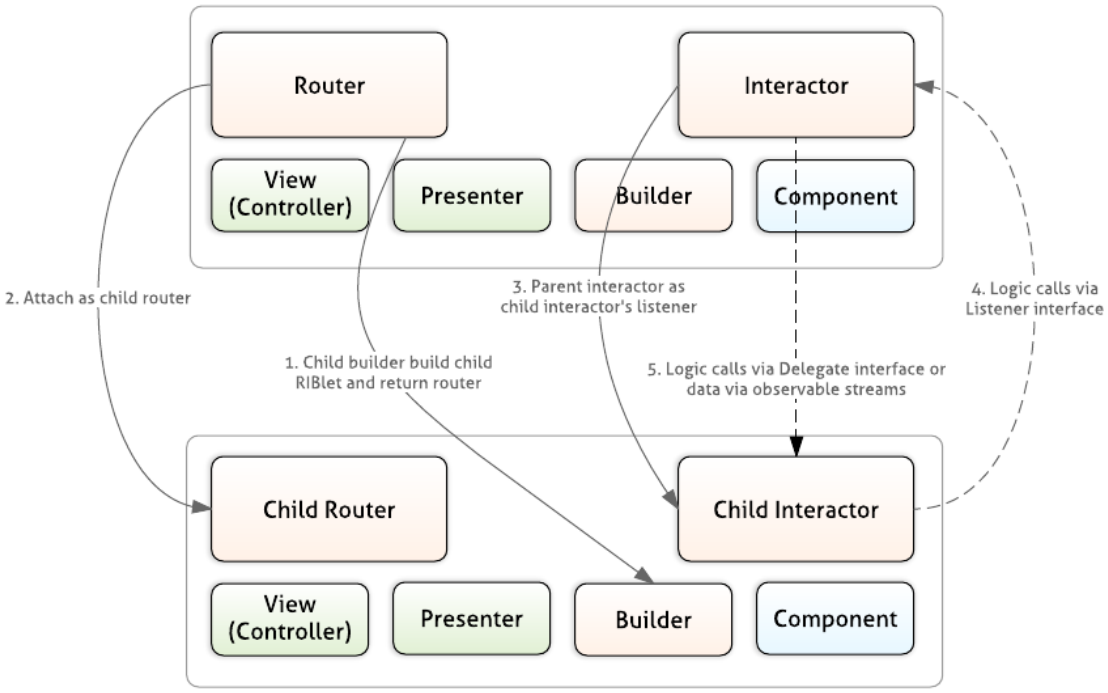
举两个例子：

1. 数据从后台到视图 View 上

   一个状态的改变，引起服务器后台触发推送到 App。数据就被 Push 到 App，然后生成不可变的数据流。关联器收到 model 之后，把它传递给展示器 Presenter。展示器 Presenter 把 model 转换成 view model 传递给视图 View。
2. 数据从视图到服务器后台

   当用户点击了一个按钮，比如登录按钮。视图 View 就会触发 UI 事件传递给展示器 Presenter。展示器 Presenter 调用关联器 Interactor 登录方法。关联器 Interactor 又会调用 Service call 的实际登录方法。请求网络之后会把数据 pull 到后台服务器。

Riblet 间的数据流



当一个关联器 Interactor 在处理业务逻辑的工程中，需要调用其他 Riblet 的事件的时候，关联器 Interactor 需要和子关联器 Interactor 进行关联。见上图5个步骤。

如果调用方法是从子调用父类，父类的 Interactor 的接口通常被定义成监听者 listener。如果调用方法是从父类调用到子类，那么子类的接口通常是一个 delegate，实现父类的一些 Protocol。

在 Riblet 的方案中，路由 Router 仅仅只是用来维护一个树型关系，而关联器 Interactor 才担当的是用来决定触发组件间的逻辑跳转的角色。

五. 各个方案优缺点

经过上面的分析，可以发现，路由的设计思路是从 URLRoute ->Protocol-class ->Target-Action 一步步的深入的过程。这也是逐渐深入本质的过程。

1. URLRoute 注册方案的优缺点

首先 URLRoute 也许是借鉴前端 Router 和系统 App 内跳转的方式想出来的方法。它通过 URL 来请求资源。不管是 H5，RN，Weex，iOS 界面或者组件请求资源的方式就都统一了。URL 里面也会带上参数，这样调用什么界面或者组件都可以。所以这种方式是最容易，也是最先可以想到的。

URLRoute 的优点很多，最大的优点就是服务器可以动态的控制页面跳转，可以统一处理页面出问题之后的错误处理，可以统一三端，iOS，Android，H5 / RN / Weex 的请求方式。

但是这种方式也需要看不同公司的需求。如果公司里面已经完成了服务器端动态下发的脚手架工具，前端也完成了 Native 端如果出现错误了，可以随时替换相同业务界面的需求，那么这个时候可能选择 URLRoute 的几率会更大。

但是如果公司里面 H5 没有做相关出现问题后能替换的界面，H5 开发人员觉得这是给他们增添负担。如果公司也没有完成服务器动态下发路由规则的那套系统，那么公司可能就不会采用 URLRoute 的方式。因为 URLRoute 带来的少量动态性，公司是可以用 JSPatch 来做到。线上出现 bug 了，可以立即用 JSPatch 修掉，而不采用 URLRoute 去做。

所以选择 URLRoute 这种方案，也要看公司的发展情况和人员分配，技术选型方面。

URLRoute 方案也是存在一些缺点的，首先 URL 的 map 规则是需要注册的，它们会在 load 方法里面写。写在 load 方法里面是会影响 App 启动速度的。

其次是大量的硬编码。URL 链接里面关于组件和页面的名字都是硬编码，参数也都是硬编码。而且每个 URL 参数字段都必须要一个文档进行维护，这个对于业务开发人员也是一个负担。而且 URL 短连接散落在整个 App 四处，维护起来实在有点麻烦，虽然蘑菇街想到了用宏统一管理这些链接，但是还是解决不了硬编码的问题。

真正一个好的路由是在无形当中服务整个 App 的，是一个无感知的过程，从这一点来说，略有点缺失。

最后一个缺点是，对于传递 NSObject 的参数，URL是不够友好的，它最多是传递一个字典。

2. Protocol-Class 注册方案的优缺点

Protocol-Class 方案的优点，这个方案没有硬编码。

Protocol-Class 方案也是存在一些缺点的，每个 Protocol 都要向 ModuleManager 进行注册。

这种方案 ModuleEntry 是同时需要依赖 ModuleManager 和组件里面的页面或者组件两者的。当然 ModuleEntry 也是会依赖 ModuleEntryProtocol 的，但是这个依赖是可以去掉的，比如用 Runtime 的方法 NSProtocolFromString，加上硬编码是可以去掉对 Protocol 的依赖的。但是考虑到硬编码的方式对出现 bug，后期维护都是不友好的，所以对 Protocol 的依赖还是不要去除。

最后一个缺点是组件方法的调用是分散在各处的，没有统一的入口，也就没法做组件不存在时或者出现错误时的统一处理。

3. Target-Action 方案的优缺点

Target-Action 方案的优点，充分的利用 Runtime 的特性，无需注册这一步。Target-Action 方案只有存在组件依赖 Mediator 这一层依赖关系。在 Mediator 中维护针对 Mediator 的 Category，每个 category 对应一个 Target，Categroy 中的方法对应 Action 场景。Target-Action 方案也统一了所有组件间调用入口。

Target-Action 方案也能有一定的安全保证，它对 url 中进行 Native 前缀进行验证。

Target-Action 方案的缺点，Target-Action 在 Category 中将常规参数打包成字典，在 Target 处再把字典拆包成常规参数，这就造成了一部分的硬编码。

4. 组件如何拆分?

这个问题其实应该是在打算实施组件化之前就应该考虑的问题。为何还要放在这里说呢？因为组件的拆分每个公司都有属于自己的拆分方案，按照业务线拆？按照最细小的业务功能模块拆？还是按照一个完成的功能进行拆分？这个就牵扯到了拆分粗细度的问题了。组件拆分的粗细度就会直接关系到未来路由需要解耦的程度。

假设，把登录的所有流程封装成一个组件，由于登录里面会涉及到多个页面，那么这些页面都会打包在一个组件里面。那么其他模块需要调用登录状态的时候，这时候就需要用到登录组件暴露在外面可以获取登录状态的接口。那么这个时候就可以考虑把这些接口写到 Protocol 里面，暴露给外面使用。或者用 Target-Action 的方法。这种把一个功能全部都划分成登录组件的话，划分粒度就稍微粗一点。

如果仅仅把登录状态的细小功能划分成一个元组件，那么外面想获取登录状态就直接调用这个组件就好。这种划分的粒度就非常细了。这样就会导致组件个数巨多。

所以在进行拆分组件的时候，也许当时业务并不复杂的时候，拆分成组件，相互耦合也不大。但是随着业务不管变化，之前划分的组件间耦合性越来越大，于是就会考虑继续把之前的组件再进行拆分。也许有些业务砍掉了，之前一些小的组件也许还会被组合到一起。总之，在业务没有完全固定下来之前，组件的划分可能一直进行时。

六. 最好的方案

关于架构，我觉得抛开业务谈架构是没有意义的。因为架构是为了业务服务的，空谈架构只是一种理想的状态。所以没有最好的方案，只有最适合的方案。

最适合自己公司业务的方案才是最好的方案。分而治之，针对不同业务选择不同的方案才是最优的解决方案。如果非要笼统的采用一种方案，不同业务之间需要同一种方案，需要妥协牺牲的东西太多就不好了。

希望本文能抛砖引玉，帮助大家选择出最适合自家业务的路由方案。当然肯定会有更加优秀的方案，希望大家能多多指点我。

参考资料

在现有工程中实施基于CTMediator的组件化方案

iOS应用架构谈 组件化方案

蘑菇街 App 的组件化之路

蘑菇街 App 的组件化之路·续

ENGINEERING THE ARCHITECTURE BEHIND UBER'S NEW RIDER APP

iOS App 签名的原理

作者：bang

iOS 签名机制挺复杂，各种证书，Provisioning Profile，entitlements，CertificateSigningRequest，p12，AppID，概念一堆，也很容易出错，本文尝试从原理出发，一步步推出为什么会有这么多概念，希望能有助于理解 iOS App 签名的原理和流程。

目的

先来看看苹果的签名机制是为了做什么。在 iOS 出来之前，在主流操作系统(Mac/Windows/Linux)上开发和运行软件是不需要签名的，软件随便从哪里下载都能运行，导致平台对第三方软件难以控制，盗版流行。苹果希望解决这样的问题，在 iOS 平台对第三方 APP 有绝对的控制权，一定要保证每一个安装到 iOS 上的 APP 都是经过苹果官方允许的，怎样保证呢？就是通过签名机制。

非对称加密

通常我们说的签名就是数字签名，它是基于非对称加密算法实现的。对称加密是通过同一份密钥加密和解密数据，而非对称加密则有两份密钥，分别是公钥和私钥，用公钥加密的数据，要用私钥才能解密，用私钥加密的数据，要用公钥才能解密。

简单说一下常用的非对称加密算法 RSA 的数学原理，理解简单的数学原理，就可以理解非对称加密是怎么做到的，为什么会是安全的：

1. 选两个质数 `p` 和 `q`，相乘得出一个大整数 `n`，例如 p=61，q=53，n=pq=3233
2. 选 1-n 间的随便一个质数 `e`，例如 e = 17
3. 经过一系列数学公式，算出一个数字 `d`，满足：
   a. 通过 `n` 和 `e` 这两个数据一组数据进行数学运算后，可以通过 n 和 d 去反解运算，反过来也可以。
   b. 如果只知道 `n` 和 `e`，要推导出 `d`，需要知道 `p` 和 `q`，也就是要需要把 n 因数分解。

上述的 `(n,e)` 这两个数据在一起就是公钥，`(n,d)` 这两个数据就是私钥，满足用公钥加密，私钥解密，或反过来公钥加密，私钥解密，也满足在只暴露公钥（只知道 `n` 和 e）的情况下，要推导出私钥 `(n,d)`，需要把大整数 `n` 因数分解。目前因数分解只能靠暴力穷举，而n数字越大，越难以用穷举计算出因数 `p` 和 `q`，也就越安全，当 `n` 大到二进制 1024 位或 2048 位时，以目前技术要破解几乎不可能，所以非常安全。

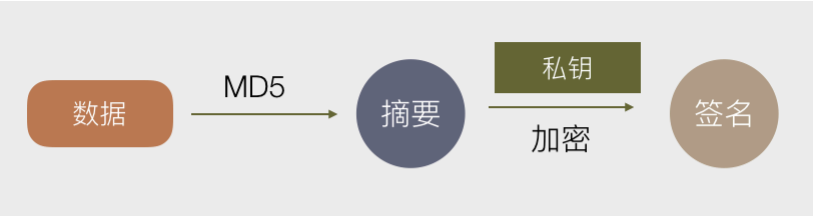
若对数字 `d` 是怎样计算出来的感兴趣，可以详读这两篇文章：RSA 算法原理（一） （二）

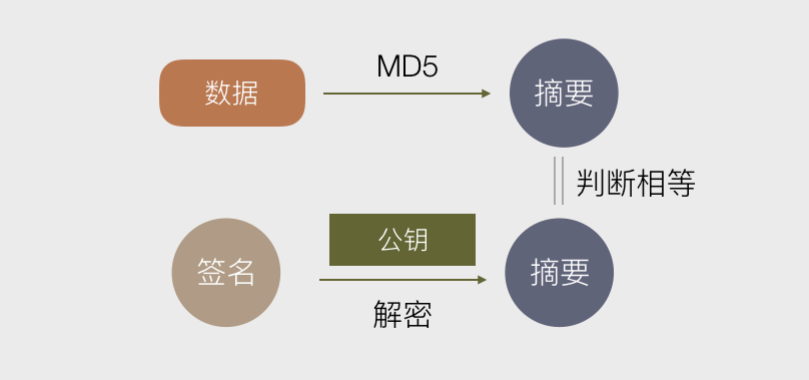
数字签名

现在知道了有非对称加密这东西，那数字签名是怎么回事呢?

数字签名的作用是我对某一份数据打个标记，表示我认可了这份数据（签了个名），然后我发送给其他人，其他人可以知道这份数据是经过我认证的，数据没有被篡改过。

有了上述非对称加密算法，就可以实现这个需求：





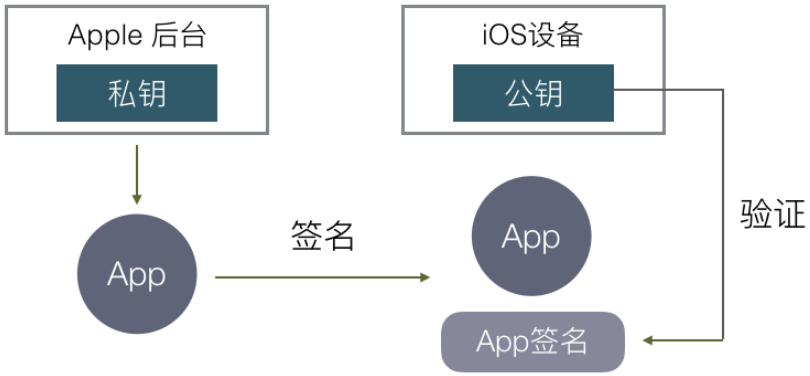
1. 首先用一种算法，算出原始数据的摘要。需满足 a.若原始数据有任何变化，计算出来的摘要值都会变化。 b.摘要要够短。这里最常用的算法是MD5。
2. 生成一份非对称加密的公钥和私钥，私钥我自己拿着，公钥公布出去。
3. 对一份数据，算出摘要后，用私钥加密这个摘要，得到一份加密后的数据，称为原始数据的签名。把它跟原始数据一起发送给用户。
4. 用户收到数据和签名后，用公钥解密得到摘要。同时用户用同样的算法计算原始数据的摘要，对比这里计算出来的摘要和用公钥解密签名得到的摘要是否相等，若相等则表示这份数据中途没有被篡改过，因为如果篡改过，摘要会变化。

之所以要有第一步计算摘要，是因为非对称加密的原理限制可加密的内容不能太大（不能大于上述 n 的位数，也就是一般不能大于 1024 位/ 2048 位），于是若要对任意大的数据签名，就需要改成对它的特征值签名，效果是一样的。

好了，有了非对称加密的基础，知道了数字签名是什么，怎样可以保证一份数据是经过某个地方认证的，来看看怎样通过数字签名的机制保证每一个安装到 iOS 上的 APP 都是经过苹果认证允许的。

最简单的签名

要实现这个需求很简单，最直接的方式，苹果官方生成一对公私钥，在 iOS 里内置一个公钥，私钥由苹果后台保存，我们传 App 上 AppStore 时，苹果后台用私钥对 APP 数据进行签名，iOS 系统下载这个 APP 后，用公钥验证这个签名，若签名正确，这个 APP 肯定是由苹果后台认证的，并且没有被修改过，也就达到了苹果的需求：保证安装的每一个 APP 都是经过苹果官方允许的。



如果我们 iOS 设备安装 APP 只有从 AppStore 下载这一种方式的话，这件事就结束了，没有任何复杂的东西，只有一个数字签名，非常简单地解决问题。

但实际上因为除了从 AppStore 下载，我们还可以有三种方式安装一个 App：

1. 开发 App 时可以直接把开发中的应用安装进手机进行调试。
2. In-House 企业内部分发，可以直接安装企业证书签名后的 APP。
3. AD-Hoc 相当于企业分发的限制版，限制安装设备数量，较少用。

苹果要对用这三种方式安装的 App 进行控制，就有了新的需求，无法像上面这样简单了。

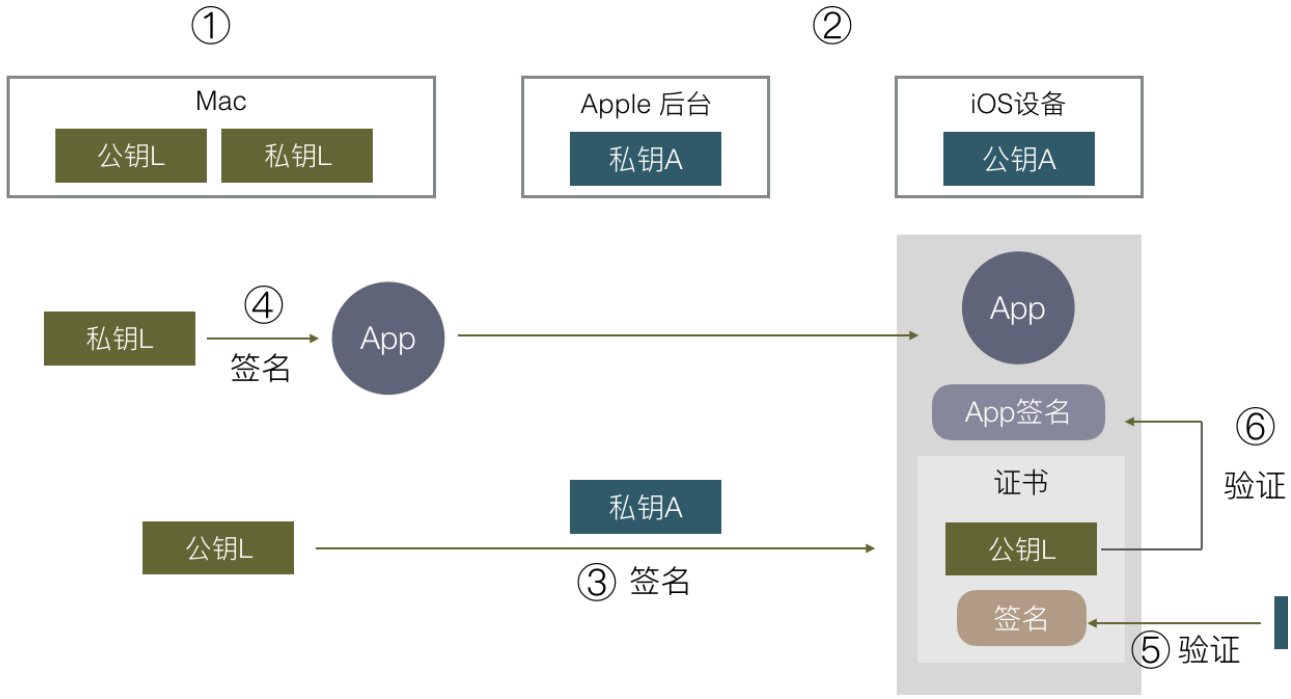
新的需求

我们先来看第一个，开发时安装APP，它有两个个需求：

1. 安装包不需要传到苹果服务器，可以直接安装到手机上。如果你编译一个 APP 到手机前要先传到苹果服务器签名，这显然是不能接受的。
2. 苹果必须对这里的安装有控制权，包括
   a.经过苹果允许才可以这样安装。
   b.不能被滥用导致非开发app也能被安装。

为了实现这些需求，iOS 签名的复杂度也就开始增加了。

苹果这里给出的方案是使用了双层签名，会比较绕，流程大概是这样的：

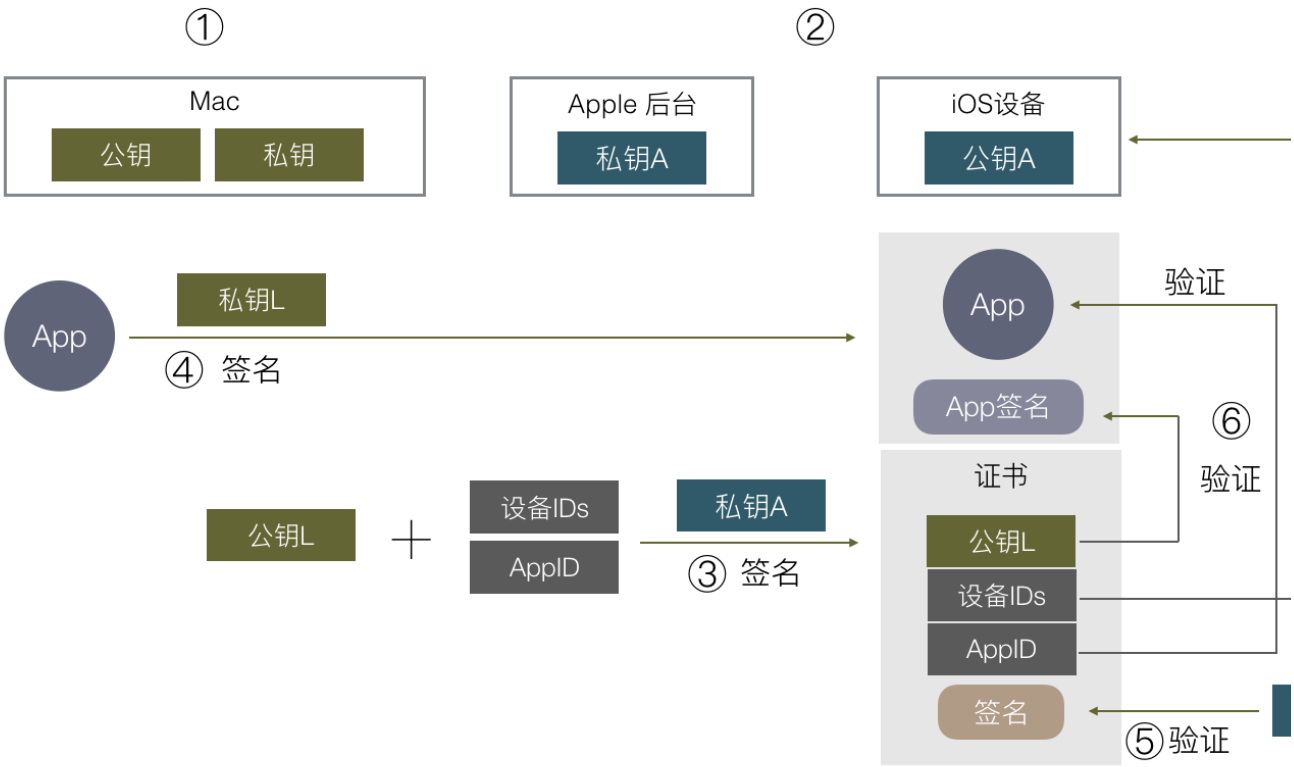


1. 在你的 Mac 开发机器生成一对公私钥，这里称为公钥L，私钥L。L:Local
2. 苹果自己有固定的一对公私钥，跟上面 AppStore 例子一样，私钥在苹果后台，公钥在每个 iOS 设备上。这里称为公钥A，私钥A。A:Apple
3. 把公钥 L 传到苹果后台，用苹果后台里的私钥 A 去签名公钥 L。得到一份数据包含了公钥 L 以及其签名，把这份数据称为证书。
4. 在开发时，编译完一个 APP 后，用本地的私钥 L 对这个 APP 进行签名，同时把第三步得到的证书一起打包进 APP 里，安装到手机上。
5. 在安装时，iOS 系统取得证书，通过系统内置的公钥 A，去验证证书的数字签名是否正确。
6. 验证证书后确保了公钥 L 是苹果认证过的，再用公钥 L 去验证 APP 的签名，这里就间接验证了这个 APP 安装行为是否经过苹果官方允许。（这里只验证安装行为，不验证APP 是否被改动，因为开发阶段 APP 内容总是不断变化的，苹果不需要管。）

加点东西

上述流程只解决了上面第一个需求，也就是需要经过苹果允许才可以安装，还未解决第二个避免被滥用的问题。怎么解决呢？苹果再加了两个限制，一是限制在苹果后台注册过的设备才可以安装，二是限制签名只能针对某一个具体的 APP。

怎么加的？在上述第三步，苹果用私钥 A 签名我们本地公钥 L 时，实际上除了签名公钥 L，还可以加上无限多数据，这些数据都可以保证是经过苹果官方认证的，不会有被篡改的可能。



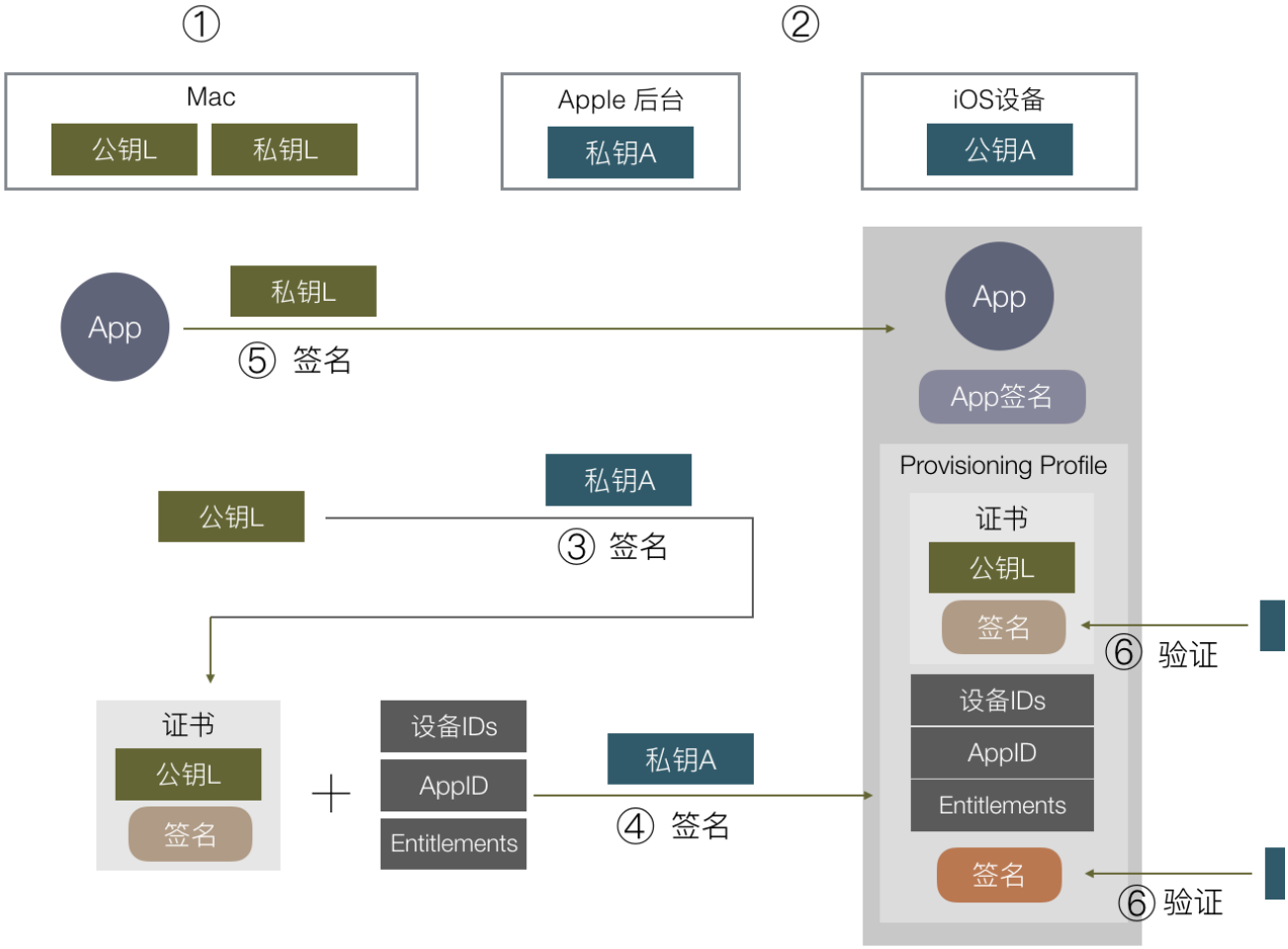
可以想到把 允许安装的设备 ID 列表 和 App对应的 AppID 等数据，都在第三步这里跟公钥L一起组成证书，再用苹果私钥 A 对这个证书签名。在最后第 5 步验证时就可以拿到设备 ID 列表，判断当前设备是否符合要求。根据数字签名的原理，只要数字签名通过验证，第 5 步这里的设备 IDs / AppID / 公钥 L 就都是经过苹果认证的，无法被修改，苹果就可以限制可安装的设备和 APP，避免滥用。

最终流程

到这里这个证书已经变得很复杂了，有很多额外信息，实际上除了 设备 ID / AppID，还有其他信息也需要在这里用苹果签名，像这个 APP 里 iCloud / push / 后台运行 等权限苹果都想控制，苹果把这些权限开关统一称为 Entitlements，它也需要通过签名去授权。

实际上一个“证书”本来就有规定的格式规范，上面我们把各种额外信息塞入证书里是不合适的，于是苹果另外搞了个东西，叫 Provisioning Profile，一个 Provisioning Profile 里就包含了证书以及上述提到的所有额外信息，以及所有信息的签名。

所以整个流程稍微变一下，就变成这样了：



因为步骤有小变动，这里我们不辞啰嗦重新再列一遍整个流程：

1. 在你的 Mac 开发机器生成一对公私钥，这里称为公钥L，私钥L。L:Local
2. 苹果自己有固定的一对公私钥，跟上面 AppStore 例子一样，私钥在苹果后台，公钥在每个 iOS 设备上。这里称为公钥A，私钥A。A:Apple
3. 把公钥 L 传到苹果后台，用苹果后台里的私钥 A 去签名公钥 L。得到一份数据包含了公钥 L 以及其签名，把这份数据称为证书。
4. 在苹果后台申请 AppID，配置好设备 ID 列表和 APP 可使用的权限，再加上第③步的证书，组成的数据用私钥 A 签名，把数据和签名一起组成一个 Provisioning Profile 文件，下载到本地 Mac 开发机。
5. 在开发时，编译完一个 APP 后，用本地的私钥 L 对这个 APP 进行签名，同时把第④步得到的 Provisioning Profile 文件打包进 APP 里，文件名为 `embedded.mobileprovision`，把 APP 安装到手机上。
6. 在安装时，iOS 系统取得证书，通过系统内置的公钥 A，去验证 `embedded.mobileprovision` 的数字签名是否正确，里面的证书签名也会再验一遍。
7. 确保了 `embedded.mobileprovision` 里的数据都是苹果授权以后，就可以取出里面的数据，做各种验证，包括用公钥 L 验证APP签名，验证设备 ID 是否在 ID 列表上，AppID 是否对应得上，权限开关是否跟 APP 里的 Entitlements 对应等。

开发者证书从签名到认证最终苹果采用的流程大致是这样，还有一些细节像证书有效期/证书类型等就不细说了。

概念和操作

上面的步骤对应到我们平常具体的操作和概念是这样的：

1. 第 1 步对应的是 keychain 里的 “从证书颁发机构请求证书”，这里就本地生成了一堆公私钥，保存的 `CertificateSigningRequest` 就是公钥，私钥保存在本地电脑里。
2. 第 2 步苹果处理，不用管。
3. 第 3 步对应把 `CertificateSigningRequest` 传到苹果后台生成证书，并下载到本地。这时本地有两个证书，一个是第 1 步生成的，一个是这里下载回来的，keychain 会把这两个证书关联起来，因为他们公私钥是对应的，在XCode选择下载回来的证书时，实际上会找到 keychain 里对应的私钥去签名。这里私钥只有生成它的这台 Mac 有，如果别的 Mac 也要编译签名这个 App 怎么办？答案是把私钥导出给其他 Mac 用，在 keychain 里导出私钥，就会存成 `.p12` 文件，其他 Mac 打开后就导入了这个私钥。
4. 第 4 步都是在苹果网站上操作，配置 AppID / 权限 / 设备等，最后下载 Provisioning Profile 文件。
5. 第 5 步 XCode 会通过第 3 步下载回来的证书（存着公钥），在本地找到对应的私钥（第一步生成的），用本地私钥去签名 App，并把 Provisioning Profile 文件命名为 `embedded.mobileprovision` 一起打包进去。这里对 App 的签名数据保存分两部分，Mach-O 可执行文件会把签名直接写入这个文件里，其他资源文件则会保存在 `_CodeSignature` 目录下。

第 6 - 7 步的打包和验证都是 Xcode 和 iOS 系统自动做的事。

这里再总结一下这些概念：

1. **证书**：内容是公钥或私钥，由其他机构对其签名组成的数据包。
2. **Entitlements**：包含了 App 权限开关列表。
3. **CertificateSigningRequest**：本地公钥。
4. **p12**：本地私钥，可以导入到其他电脑。
5. **Provisioning Profile**：包含了 证书 / Entitlements 等数据，并由苹果后台私钥签名的数据包。

其他发布方式

前面以开发包为例子说了签名和验证的流程，另外两种方式 In-House 企业签名和 AD-Hoc 流程也是差不多的，只是企业签名不限制安装的设备数，另外需要用户在 iOS 系统设置上手动点击信任这个企业才能通过验证。

而 AppStore 的签名验证方式有些不一样，前面我们说到最简单的签名方式，苹果在后台直接用私钥签名 App 就可以了，实际上苹果确实是这样做的，如果去下载一个 AppStore 的安装包，会发现它里面是没有 `embedded.mobileprovision` 文件的，也就是它安装和启动的流程是不依赖这个文件，验证流程也就跟上述几种类型不一样了。

据猜测，因为上传到 AppStore 的包苹果会重新对内容加密，原来的本地私钥签名就没有用了，需要重新签名，从 AppStore 下载的包苹果也并不打算控制它的有效期，不需要内置一个 `embedded.mobileprovision` 去做校验，直接在苹果用后台的私钥重新签名，iOS 安装时用本地公钥验证 App 签名就可以了。

那为什么发布 AppStore 的包还是要跟开发版一样搞各种证书和 Provisioning Profile？猜测因为苹果想做统一管理，Provisioning Profile 里包含一些权限控制，AppID 的检验等，苹果不想在上传 AppStore 包时重新用另一种协议做一遍这些验证，就不如统一把这部分放在 Provisioning Profile 里，上传 AppStore 时只要用同样的流程验证这个 Provisioning Profile 是否合法就可以了。

所以 App 上传到 AppStore 后，就跟你的 证书 / Provisioning Profile 都没有关系了，无论他们是否过期或被废除，都不会影响 AppStore 上的安装包。

到这里 iOS 签名机制的原理和主流程大致说完了，希望能对理解苹果签名和排查日常签名问题有所帮助。

P.S.一些疑问

最后这里再提一下我关于签名流程的一些的疑问。

企业证书

企业证书签名因为限制少，在国内被广泛用于测试和盗版，fir.im / 蒲公英等测试平台都是通过企业证书分发，国内一些市场像 PP 助手，爱思助手，一部分安装手段也是通过企业证书重签名。通过企业证书签名安装的 App，启动时都会验证证书的有效期，并且不定期请求苹果服务器看证书是否被吊销，若已过期或被吊销，就会无法启动 App。对于这种助手的盗版安装手段，苹果想打击只能一个个吊销企业证书，并没有太好的办法。

这里我的疑问是，苹果做了那么多签名和验证机制去限制在 iOS 安装 App，为什么又要出这样一个限制很少的方式让盗版钻空子呢？若真的是企业用途不适合上 AppStore，也完全可以在 AppStore 开辟一个小的私密版块，还是通过 AppStore 去安装，就不会有这个问题了。

AppStore 加密

另一个问题是我们把 App 传上 AppStore 后，苹果会对 App 进行加密，导致 App 体积增大不少，这个加密实际上是没卵用的，只是让破解的人要多做一个步骤，运行 App 去内存 dump 出可执行文件而已，无论怎样加密，都可以用这种方式拿出加密前的可执行文件。所以为什么要做这样的加密呢？想不到有什么好处。

本地私钥

我们看到前面说的签名流程很绕很复杂，经常出现各种问题，像有 Provisioning Profile 文件但证书又不对，本地有公钥证书没对应私钥等情况，不理解原理的情况下会被绕晕，我的疑问是，这里为什么不能简化呢？还是以开发证书为例，为什么一定要用本地 Mac 生成的私钥去签名？苹果要的只是本地签名，私钥不一定是要本地生成的，苹果也可以自己生成一对公私钥给我们，放在 Provisioning Profile 里，我们用里面的私钥去加密就行了，这样就不会有 `CertificateSigningRequest` 和 `p12` 的概念，跟本地 keychain 没有关系，不需要关心证书，只要有 Provisioning Profile 就能签名，流程会减少，易用性会提高很多，同时苹果想要的控制一点都不会少，也没有什么安全问题，为什么不这样设计呢?

能想到的一个原因是 Provisioning Profile 在非 AppStore 安装时会打包进安装包，第三方拿到这个 Provisioning Profile 文件就能直接用起来给他自己的 App 签名了。但这种问题也挺好解决，只需要打包时去掉文件里的私钥就行了，所以仍不明白为什么这样设计。

再看关于 Storyboard 的一些争论

作者：王巍

从 iOS 5 的时代 Apple 推出 Storyboard (以下简称 SB) 后，关于使用这种方式构建 UI 的争论就在 Cocoa 开发者社区里一直发生着。我在 2013 年写过一篇关于代码手写 UI，xib 和 SB 之间的取舍的文章。在四五年后的今天，SB 得到了多次进化，大家也积攒了很多关于使用 SB 进行开发的经验，我们不妨再回头看看当初的忧虑，并结合 SB 开发的现状，来提取一些现阶段被认为比较好的实践。

这篇文章缘起为对使用 SB 的方式一文 (及其英文原文) 的回应，我对其中部分意见有一些不同的看法。不过正如原文作者在最后一段所说，你应该选择最适合自己的使用方式。所以我的意见或者说所谓的「好的实践」，也只是从我自己的观点出发所得到的结论。本文将首先对原文提出的几个论点逐个分析，然后介绍一些我自己在日常使用 SB 时的经验和方式。

(反正关于 Storyboard 或者 Interface Builder 已经吵了那么多年了，也不在乎多这么一篇-。-)

原文分析

Storyboard 冲突风险和加载

原文中有一个非常激进的观点，那就是：

> 每个 SB 里只放一个 UIViewController

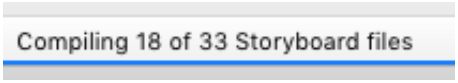
我无法赞同这个观点。如果在 iOS 3 或者 4 时代有 xib 使用经验的开发者会知道，这基本就是将 SB 倒退到 xib 的用法。原文中提到这么做的原因主要有三点：

> - 减少两个开发者同时开发一个 View Controller 时的 git 冲突
> - 加速 storyboard 加载，因为只需要加载一个 UIViewController
> - 只用 initial view controller 就可以从 SB 中加载想要的 View Controller

在 Xcode 7 引入了 SB reference 以后，「SB 容易冲突」已经彻底变成假命题了。通过合理地划分功能模块和每个开发者负责的部分，我们可以完全避免 SB 的修改冲突。最近两三年以来我们在实际项目中完全没有出现过 SB 冲突的情况。

另外，即使 SB 划分出现问题，影响也是可控的。在单个的 SB 文件中，每个 View Controller 有各自的范围域，因此即使存在不同开发者同时着手一个 SB 文件的情况，只要他们不同时修改同一个 View Controller 的内容，也并不会在 View Controller 上产生冲突。在 SB 文件中确实存在一些共用的部分，比如 IB 的版本，系统的版本等，但它们并不影响实质的 UI，而且可以通过统一开发成员的环境来避免冲突。因此，一个 SB 中多个 VC 和一个 SB 中一个 VC，其实所带来的冲突风险几乎是一样的。

关于 SB 的加载，可以看出原作者可能并没有搞清 UI 加载的整个流程，不求甚解地认为 SB 文件中 View Controller 越多加载时间越长，但事实并非如此。细心的同学 (或者项目中有很多 SB 文件的同学) 会发现，在编译的时候 Xcode 有一个 Compiling Storyboard files 的过程：



编译过程中，项目里用到的 SB 文件也会被编译，并以 `storyboardc` 为扩展名保存在最终的 app 包内。这个文件和 `.bundle` 或者 `.framework` 类似，实际上是一个文件夹，里面存储了一个描述该编译后的 SB 信息的 `Info.plist` 文件，以及一系列 `.nib` 文件。原来的 SB 中的每个对象 (或者说，一般就是每个 View Controller) 将会被编译为一个单独的 `.nib`，而 `.nib` 中包含了编码后的对应的对象层级。在加载一个 SB，并从中读取单个 View Controller 时，首先系统会找到编译后的 `.storyboardc` 文件，从 `Info.plist` 中获取所需的 View Controller 类型和 nib 的关系，来完成 `UIStoryboard` 的初始化。接下来读取对应的某个 nib，并使用 `UINibDecoder` 进行解码，将 nib 二进制还原为实际的对象，最后调用该对象的 `initWithCoder:` 完成各个属性的解码。在完成这些工作后，`awakeFromNib` 被调用，来通知开发者从 nib 的加载已经完毕。

如果你理解这个过程，就可以看出，从只有单个 View Controller 的 SB 中加载这个 VC，与从多个 View Controller 中加载一个的情况，在速度上并不会有什么区别。硬要说的话，如果使用太多 SB 文件，反而会在初始化 `UIStoryboard` 时需要读取更多的 `Info.plist`，反而造成性能下降 (相对地我们可以使用 View Controller 的 `storyboard` 属性来获取当前 VC 所属的 `UIStoryboard`，从而避免多次初始化同一个 Storyboard，不过这点性能损失其实无关紧要)。

关于第三点，原作者使用了一段代码来展示如何通过类似这样的方法来创建类型安全的对象：

```
let feed = FeedViewController.instance()
// `feed` is of type `FeedViewController`
```

这么做有几个前提，首先它需要按照 View Controller 类型名字来创建 SB 文件，其次还需要为 `UIViewController` 添加按照类型名字寻找 SB 文件的辅助方法。这并不是一个很明显的优点，它肯定会引入 `NSStringFromClass` 这种动态的东西，而且其实我们有很多更好的方式来创建类型安全的 View Controller。我会在第二部分介绍一些相关的内容。

Segue 的使用

原文中第二个主要观点是：

> 不要使用 Segue

Segue 的基本作用是串联不同的 View Controller，完成各 VC 的迁移或者组织。在第一个观点 (一个 SB 文件只含有一个 VC) 的前提下，不使用 Segue 是自然而然的推论，因为同一个 SB 中没有多个 VC 的关系需要组织，segue 的作用被大大降低。但是作者使用了一个不是很好的例子想要强行说明使用 segue 以及 `prepare(for:sender:)` 的坏处。下面是原文中的一段示例代码：

```
class UsersViewController: UIViewController, UITableViewDelegate {

  private enum SegueIdentifier {
    static let showUserDetails = "showUserDetails"
  }

  var usernames: [String] = ["Marin"]

  func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {
    usernameToSend = usernames[indexPath.row]
    performSegue(withIdentifier: SegueIdentifier.showUserDetails, sender:
nil)
  }


  private var usernameToSend: String?

  override func prepare(for segue: UIStoryboardSegue, sender: Any?) {

    switch segue.identifier {
      case SegueIdentifier.showUserDetails?:

        guard let usernameToSend = usernameToSend else {
          assertionFailure("No username provided!")
          return
        }

        let destination = segue.destination as! UserDetailViewController
        destination.username = usernameToSend

      default:
        break
    }
  }
}
```

简单说，这段代码做的就是在用户点击 table view 中某个 cell 的时候，将点击的内容保存到 View Controller 的一个成员变量 `usernameToSend` 中，然后调用 `performSegue(withIdentifier:sender:)`。接下来，在 `prepare(for:sender:)` 中获取保存的这个成员变量，并且设置给目标 View Controller。对于 table view 来说，这是一个不太必要的做法。我们完全可以直接将 cell 通过 segue 连接到目标 View Controller 上，然后在 `prepare(for:sender:)` 中使用 table view 的 `indexPathForSelectedRow` 获取需要的数据，并对目标 View Controller 进行设置。可能原作者不太清楚 `UITableView` 有这么一个 API，所以用了不太好的例子。

那么 segue 有问题吗？我的回答是有，但是问题不大。实际开发中确实存在不少类似原作者说到的情形，需要将数据在 `prepare(for:sender:)` 中传递给目标 View Controller，不过这种情况的数据很多时候已经存在于当前 View Controller 中 (比如需要传递文本框中输入的文字，或者当前 VC 的 model 的某个属性等)。相比于变量的问题，segue 带来的更大的挑战在于 View Controller 之间迁移的管理。现在我们可以通过代码进行转场 (`pushViewController(:animated)` 或者 `present(:animated:completion)`)，也可以使用 SB 里点击控件的 segue，甚至还可以从代码中调用 `performSegue`，在不同的地方进行管理让代码变得复杂和难以理解，所以我们可能需要考虑如何以集中的方式进行管理。objc.io 的 Swift Talk 的第五期视频 - Connecting View Controllers (而且是免费的) 对这个问题进行了一些探讨，并给出了一种集中管理 View Controller 之间迁移的方式。其中使用回调的方法可以借鉴，但是我个人对整个思路运用在实际项目里存有疑虑，大家也不妨作为参考了解。

除了管理转场外，segue 还能够提供方便的 Container View 的 embed 关系，也可以在使用像是 `UIPageViewController` 这样的多个 VC 关系的时候，用来提供一些初始化时运行的代码，又或者是用 unwind 来方便地实现 dismiss。这些「附加」的功能都让我们少写很多代码，开发效率得到提升，不去尝试使用的话可以说是相当可惜。

要爱，不要拒绝 GUI

原文作者的最后一个主要观点是：

> 所有的属性都在代码中设置

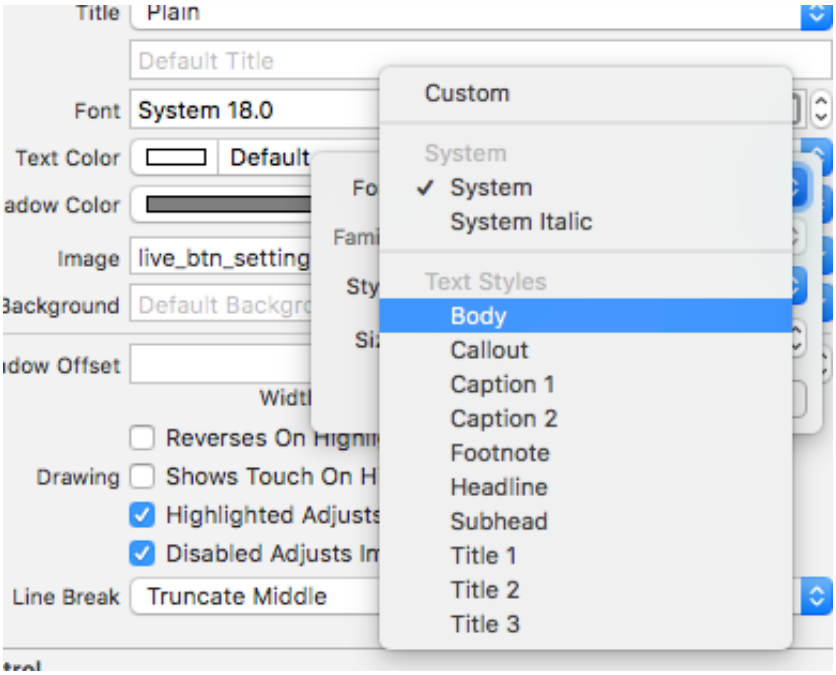
作者在原文一开始就提到，人都是视觉动物，使用 SB 的一大目标就是直观地理解界面。通过 SB 画布我们可以迅速获得要进行开发的 View Controller 的信息，这比阅读代码要快得多。但是，如果所有属性都在代码中进行设置的话，这一优势还剩多少呢？

作者提议在 SB 中对添加的 View 或者 ViewController 保留所有默认设置 (甚至是 view 的背景颜色，或者 label 文字等)，然后使用代码对它们进行设置。在这一点上，原文作者的顾虑是对于 UI 元素样式的更改。作者希望通过使用一些常量来保存像是字体，颜色等，并在代码中将它们分别赋值给 UI 元素，这样能做到设计改变时只在一处进行更改就可以对应。

这种做法带来的缺点相当明显，那就是为了设置这些属性，你需要很多的 IBOutlet，以及很多额外的工作量。我的建议是，对于那些不会随着程序状态改变的内容，最好尽量使用 SB 直接进行设置。比如一个 label 上的文字，除非这些文字确实需要改变 (比如显示的是用户名，或者当前评论数之类)，否则完全没有必要添加 `@IBOutlet`，直接设置 text 会简单得多。其他像是 `UIScrollView` 的 Cancellable Content Touches 等属性，如果不需要在程序中根据程序状态进行改变，也最好直接在 IB 里设置。作者在原文里提到，“通过扫描代码来寻找 view 的属性要比在 storyboard 中寻找一个勾号来的容易”，关于这一点，我认为其实两者并没有什么不同。举例来说，通过 IB 将 `UIScrollView` 的 Cancellable Content Touches 设置为 `false`，在对应的 SB 文件中的 scroll view 里会加上 `canCancelContentTouches="NO"` 这样的属性。通过全局搜索的方式找到这个属性也是轻而易举的。甚至你可以直接修改 SB 的源码达到目的，而根本不需要打开 Xcode 或者 IB。基于查找的可能性，批量的替换和更新与使用代码来设置也并无异。并不存在说在代码里更容易被找到这种情况。

> 不过要注意的是，SB 中的属性在 Xcode 的查找结果中是被过滤，不会出现的，所以可能需要使用其他的文本编辑器来全局查找。

关于像是字体或者颜色这样的 view 样式，作者的顾虑可以理解。IB 现在缺乏良好的做样式的方法，这也是大家诟病已久的问题。在 Font 选择中存在 style 的选项，让我们可以从 Body，Headline 之类的项目中进行选择，看起来很好：



但是这仅仅只是为了支持 Dynamic Type，设置这些值和调用 `UIFont` 的 `preferredFont(forTextStyle:)` 获取特定字体是一样的。我们并不能自行定义这些字体样式，也不能进行添加。颜色也一样，Xcode 并没有提供一个类似可以在 IB 里使用的项目颜色版或者颜色变量的概念。

关于 view 样式，最常见也是最简单的解决方案大概有两种。

第一种是使用自定义的子类，来统一设置字体或者颜色这些属性。比如说你的项目里可以会有 `HeaderLabel`，或者 `BodyLable` 这样的 `UILabel` 的子类，然后在子类里相应的方法中设置字体。这种方式来得比较直接，你可以通过更改 IB 里的 label 类型来适用字体。但是缺点在于当项目变大以后，可能 label 的类型会变得很多。另外，对于非全局性的修改，比如只针对某一个特定 label 调整的时候会比较麻烦，很可能你会想只针对个例做个别调整，而不是专门为这种情况建立新的子类，而这个决定往往会让你之前为了统一样式所做的努力付之一炬。

另外一种方式是为目标 view 的类型添加像是 `style` 属性，然后使用 runtime attribute 来设置。简单的想法大概是这样的，比如针对字体：

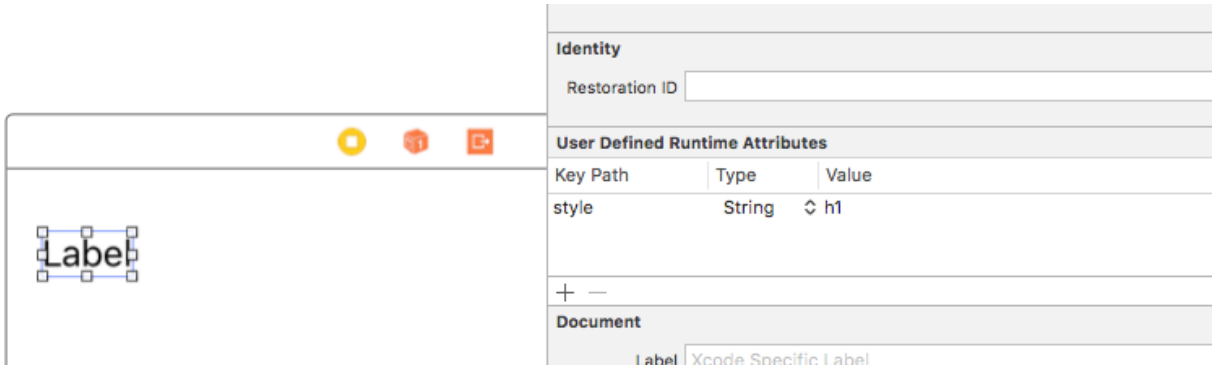
```
extension UIFont {

    enum Style: String {
        case p = "p"
        case h1 = "h1"
        case defalt = ""
    }

    static func font(forStyle string: String?) -> UIFont {
        guard let fontStyle = Style(rawValue: string ?? "") else {
            fatalError("Unrecognized font style.")
        }

        switch fontStyle {
        case .p: return .systemFont(ofSize: 14)
        case .h1: return .boldSystemFont(ofSize: 22)
        case .defalt: return .systemFont(ofSize: 17)
        }
    }
}
```

这段代码为 `UIFont` 添加了一个静态方法，通过输入的字符串获取不同样式的字体。

然后，我们为需要字体样式支持的类型添加设置 `style` 的扩展，比如对 `UILabel`：

```
extension UILabel {
    var style: String {
        get { fatalError("Getting the label style is not permitted.") }
        set { font = UIFont.font(forStyle: newValue) }
    }
}
```

在使用的时候，我们在 IB 里想要适用样式的 `UILabel` 添加 runtime attribute 就可以了：



不过不论哪种做法，缺点都是我们无法在 IB 中直观地看到 label 的变化。当然，可以通过为自定义的 `UILabel` 子类实现 `@IBDesignable` 来克服这个缺点，不过这也需要额外的工作量。还是希望 Xcode 和 IB 能够进步，原生支持类似的样式组织方式吧。不过就因此放弃简单明了的 UI 构建方式，未免有些过于武断。

基本上我对原文的每个观点已经提出了我的想法，不过正如原文作者最后说的那样，你应该选择你自己的使用风格，并决定要如何使用 Storyboard。

> It's not all or nothing.

原文作者就只将 IB 和 Storyboard 作为一个设置 view 层次和添加 layout 约束的工具，这确实是 SB 的强项所在，但是我认为它的功能要远比这强大的多。正确地理解 SB 的设计思想和哲学，正确地在可控范围内使用 SB，对于发掘这个工具的潜力，对于进一步提高开发效率，都会带来好处。

本文下一部分将会简单介绍几个使用 SB 的实践。

实践经验

以类型安全的方式使用 Storyboard

原文作者提到使用单个 VC 的 Storyboard 可以以类型安全的方式进行创建。其实这并不是必要条件，甚至我们通过别的方式可以做得更好。在 Cocoa 框架中，为了灵活性，确实有很多基于字符串的 API，这导致了一定程度的不安全。Apple 自己为了 API 的通用性和兼容性，不太可能对现有的类型不安全的 API 进行大幅修改，不过通过一些合适的封装，我们依然可以让 API 更加安全。不管是我个人的项目还是公司的项目，其实都在使用像是 R.swift 这样的工具。这个项目通过扫描你的各种基于字符串命名的资源 (比如图片名，View Controller 和 segue 的 identifier 等)，创建一个使用类型来获取资源的方式。相比与原作者的类型安全的手法，这显然是一种更成熟和完善的方式。

比如原来我们可以要用这样的代码来从 SB 里获取 View Controller：

```
let myImage = UIImage(names: "myImage")
let viewController = UIStoryboard(name: "Main", bundle:
nil).instantiateViewController(withIdentifier: "myViewController") as!
MyViewController
```

在 R.swift 的帮助下，我们将可以使用下面的代码：

```
let myImage = R.image.myImage()
// myImage: UIImage?

let viewController = R.storyboard.main.myViewController()
// viewController: MyViewController?
```

这种做法在保证类型安全的同时，还可以在编译时就确认相应资源的存在。要是你修改了 SB 中 View Controller 的 identifier，但是没有修改相应代码的话，你会得到一个编译错误。

R.swift 除了可以针对图片和 View Controller 外，也可以用在本地化字符串、Segue、nib 文件或者 cell 等一系列含有字符串 identifier 的地方。通过在项目中引入 R.swift 进行管理，我们在开发中避免了很多可能的资源使用上的危险和 bug，也在自动补全的帮助下节省了无数时间，而像是使用 Storyboard 并从中创建 View Controller 这样的工作也变得完全不值一提了。

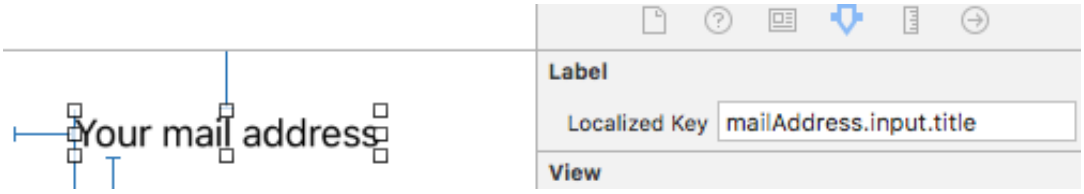
利用 @IBInspectable 减少代码设置

通过 IB 设置 view 的属性有一个局限，那就是有一些属性没有暴露在 IB 的设置面板中，或者是设置的时候有可能要“转个弯”。虽然在 IB 面板中已经包含了八九成经常使用的属性，但是难免会有「漏网之鱼」。我们在工程实践中最常遇到的情形有两种：为一个显示文字的 view 设置本地化字符串，以及为一个 image view 设置圆角。

这两个课题我们都使用在对应的 view 中添加 `@IBInspectable` 的 extension 方法来解决。比如对于本地化字符串的问题，我们会有类似这样的 extension：

```
extension UILabel {
    @IBInspectable var localizedKey: String? {
        set {
            guard let newValue = newValue else { return }
            text = NSLocalizedString(newValue, comment: "")
        }
        get { return text }
    }
}

extension UIButton {
    @IBInspectable var localizedKey: String? {
        set {
            guard let newValue = newValue else { return }
            setTitle(NSLocalizedString(newValue, comment: ""), for: .normal)
        }
        get { return titleLabel?.text }
    }
}

extension UITextField {
    @IBInspectable var localizedKey: String? {
        set {
            guard let newValue = newValue else { return }
            placeholder = NSLocalizedString(newValue, comment: "")
        }
        get { return placeholder }
    }
}
```

这样，在 IB 中我们就可以利用对应类型的 Localized Key 来直接设置本地化字符串了：



设置圆角也类似，为 `UIImageView` (或者甚至是 `UIView`) 引入这样的扩展，并直接在 IB 中进行设置，可以避免很多模板代码：

```
@IBInspectable var cornerRadius: CGFloat {
    get {
        return layer.cornerRadius
    }

    set {
        layer.cornerRadius = newValue
        layer.masksToBounds = newValue > 0
    }
}
```

`@IBInspectable` 实际上和上面提到的 `UILabel` 的 style 方法一样，它们都使用了 runtime attribute。显然，你也可以把 `UILabel` style 写成一个 `@IBInspectable`，来方便在 IB 中直接设置样式。

@IBOutlet 的 didSet

虽然这个小技巧并不会对 IB 或者 SB 的使用带来实质性的改善，但是我觉得还是值得一提。如果我们由于某种原因，确实需要在代码中设置一些 view 的属性，在连接 `@IBOutlet` 后，不少开发者会选择在 `viewDidLoad` 中进行设置。其实个人认为一个更合适的地方是在该 `@IBoutlet` 的 `didSet` 中进行。`@IBoutlet` 所修饰的也是一个属性，这个关键词所做的仅只是将属性暴露给 IB，所以它的各种属性观察方法 (`willSet`， `didSet` 等) 也会被正常调用。比如，下面我们实际项目中的一段代码：

```
@IBOutlet var myTextField: UITextField! {
    didSet {
        // Workaround for https://openradar.appspot.com/28751703
        myTextField.layer.borderWidth = 1.0
        myTextField.layer.borderColor = UIColor.lineGreen.cgColor
    }
}
```

这么做可以让设置 view 的代码和 view 本身相对集中，也可以使 `viewDidLoad` 更加干净。

继承和重用的问题

夸了 Storyboard 这么多，当然不是说它没有缺点。事实不仅如此，SB 还有很多很多可以改善的地方，其中，使用 SB 来实现继承和重用是最困难的地方。

Storyboard 不允许放置单独的 view，所以如果想要通过 IB 来实现 view 的重用的话，我们需要回退到 xib 文件。即使如此，想要在 SB 的 View Controller 中初始化一个通过 xib 加载的 view 也并不是一件很容易的事情。一般对于这种需求，我们会选择在 `init(coder:)` 中加载目标 nib 然后将它作为 subview 添加到目标 view 中。整个过程需要开发者对 nib 加载 view 和 View Controller 的过程有比较清楚的了解，但不幸的是 Apple 把这个过程藏得有些深，所以绝大多数开发者并不关心、也不是很清楚这个过程，就认为这是不可能的。

对于 view 的继承的话更困难一些。依然是由于二进制 nib 将通过解码的方式进行还原，所以在设置父类的属性时需要特别注意。另外，子类的 UI 是否应该通过创建新的 xib 进行构建，还是应该通过代码将父类的 UI 加到子类上，也会是艰难的选择。相比起来，使用代码进行 view 的继承和重用就要容易得多，方法也明确得多。

不光是单独的 view，SB 中 View Controller 的继承和重用也面临着同样的问题。View Controller 的重用相对简单，通过 storyboard 初始化对应的 View Controller，或者通过 segue 就可以了。继承则更麻烦，不过好在相比起 view 的继承，View Controller 的继承关系并不会特别复杂，在 UIKit 中对于 `UIViewController` 的继承最常用的基本也就 `UITableViewController`，`UICollectionViewController`，而作为最终展示给用户的 view 的管理代码来说，也很少有需要继承一个已经高度专用，并使用 IB 构建的 View Controller。如果你在项目中出现这种继承的需求，首先对继承的必要性进行考虑会是不错的选择。如果可以通过不同的配置重用已有的 View Controller，那么说明「继承」可能只是一个伪需求。

不管如何，不能否认，因为构建 UI 的方式是对 xml 文件的编码和解码，由此带来了继承和重用的困难，这是 IB 或者说 SB 的最大的短板。

总结

本文旨在介绍一些我自己对 Storyboard 的看法，和我日常开发中的使用方式。并不是说什么「你应该这样使用」或者「最佳实践就应当如此这般」。你可以选择使用纯代码构建 UI，但同时 Apple 也为我们提供了更快捷的 IB 和 Storyboard 的方式。在我这么几年的使用经验来看，SB 的设计并没有这么不堪，而相比于以前使用代码或者 xib 的方式，现在的开发方式确实让效率得到了提高。开发者根据自己的需求和理解对工具进行选择，每个人的选择和使用的方式都是值得尊重。只要愿意拥抱变化，勇于尝试新的事物，并从中找到合适自己的东西，那么使用什么样的方式本身其实便没有那么重要了。

最后，愿你的技术历久弥新，愿你的生活光芒万丈。

Swift ABI稳定性蓝图

- Authors: Michael Ilseman (compiled through conversations with many others)
- 作者：Michael Ilseman (其中包含了很许多人交流的结果)

蓝图

当前，Swift最重要的工作之一，就是在多个版本之间保持兼容性。而所谓的兼容性，则是要在源码以及二进制层次上，分别达到它们各自的目标：

1. 在源代码层次实现兼容，也就是让新版本的编译器可以编译老版本的Swift代码。这可以降低开发者在迁移到Swift新版本时的痛苦。没有源代码层次的兼容，用Swift编写的项目就很难得以传承，项目中所有的源代码和使用的程序库必须使用同一个版本的Swift语言。而提供了源代码层次的兼容性，程序库的作者就可以基于某个Swift版本只维护一套代码，并且让程序库的使用者使用更新版本的Swift；
2. 而程序库和运行时在二进制上的兼容性则让（用某个版本编译的）程序库在多个不同的Swift版本上进行分发成为可能。二进制的程序库包含了一个Swift module file，它提供了这个程序库API源代码级别的信息；以及一个共享的程序库文件，它包含了在运行时加载的编译之后的代码。因此，实现二进制的兼容性需要完成两个目标：

- 首先，Swift module格式的稳定性可以让编译器用固定的方式表达程序库的公共接口，这也就是Swift module file的稳定性。这包括了API的声明以及代码是否可以inline的表达方式。编译器需要使用module file完成诸多必要的工作，例如，当编译程序库的客户端调用时，对其进行类型检查以及生成调用代码。
- 其次，ABI的稳定性则为使用不同版本Swift编译的程序库和应用程序提供了兼容性。这是这份文档接下来的部分关注的内容。这份文档探索并解释了什么是Swift ABI，并罗列了宣称Swift ABI已稳定前要进行的调查以及完成的目标。作为Swift社区的一份资源，这份文档也为Swift ABI的发展指明了方向。

这份文档中，会用"SR-xxxx"的形式，引用Swift issue tracking system中的内容。这些内容记录了和Swift ABI相关的工程和设计任务。

什么是ABI?

在运行时，通过Swift编写的二进制程序通过ABI和其他的程序库或组件进行交互。ABI是Application Binary Interface的缩写，它是一个规范，通过这个规范，所有被独立编译的二进制实体才能被链接在一起并执行。这些二进制实体必须在一些很低层的细节上达成一致，例如：如何调用函数，如何在内存中表示数据甚至是如何存储以及访问metadata。

ABI是平台相关的，因为它关注的这些底层细节会受到不同的硬件架构以及操作系统的的影响。大部分的平台都定义了一份可以用于C以及C语言家族代码的“标准ABI”。但Swift是一门和C截然不同的语言，它需要为每一种平台定义自己的ABI。而这份文档中中的绝大部分内容都是和平台无关的，和具体平台相关的考量会影响到Swift ABI的设计和实现细节。如果你要了解不同平台上的“标准ABI”，可以参考附录中的内容。

什么是ABI稳定?

ABI稳定是指把ABI锁定在某种形式，以至于未来的编译器都可以生成遵从这种形式的二进制实体（可以是程序库，也可以是应用程序）。一旦ABI稳定了，就意味着它会伴着这个平台的一生一世，直至日益增长的依赖关系让它走向灭亡。

ABI的稳定性仅会影响到外部可见的公共接口和符号的不变性。而内部使用的符号、调用约定以及内存格局仍旧可以修改而不会破坏ABI约定。例如，未来版本编译器完全可以在保留外部接口函数调用约定的同时，改变内部函数的调用约定。

对ABI做的每一个决定都会对编程语言产生长远的影响，甚至会制约一门编程语言在未来可以发展和进化的空间。Swift未来的版本可能会在更多垂直领域为ABI添加特性，但只要声明了某个平台的ABI已稳定，那么任何在效率以及灵活性上曾经不妥的设计都将永远伴着这个平台存在。

为ABI在垂直领域添加新的特性叫做ABI积累性更新（ABI-additive changes）。每当支持这个特性的Swift版本达到最小目标值的时候，就可以把这个特性纳入积累性更新。这种方式允许我们逐步扩展或锁定ABI中的内容。例如，让ABI支持更多特性，或使用更有效率的数据访问方式等。在这份文档后面，我们会看到很多这样的例子。

ABI稳定了会怎样?

ABI稳定之后，OS发行商就可以把Swift标准库和运行时作为操作系统的一部分嵌入，由于这些标准库和运行时可以支持用更老或更新版本Swift构建的应用程序，这样，开发者就无需在分发应用的同时，还要带上一份自己构建应用时使用的标准库和运行时拷贝。这使得工具和操作系统可以更好的进行集成。

就像之前提到的一样，对二进制程序库来说，ABI稳定是必要但不充分的。模块文件格式的稳定性也是必须的，但这超出了这份文档讨论的范畴。

程序库的进化

通常，如果一门编程语言在表现力和性能上均表现良好，并且还提供了二进制接口，它就很容易表现出二进制接口很脆弱的问题。对于用这种语言编写的程序库或组件来说，要让用户在不重新编译源码的条件下使用新版本的程序库是很难的。当前，Swift主要的前进方向是标准库的进化，自然就要给开发者在维护二进制兼容性这个问题上更多进退的空间。很多实现上的考量的确会对ABI的设计有所影响。

因此，迟迟没有声明ABI稳定的一个主要目的，就是为程序库的进化保留足够的灵活性，而不希望开发者由于ABI的稳定性而限制设计空间。程序库在进化上的考量将会在下面每一个单独的章节中描述，但它们阐述的一个公共的观点则是有些设计细节仍旧没有确定下来。

Swift ABI的构成

在实践中，ABI关注的内容是紧密耦合在一起的。但是，作为一个概念模型。我更愿意把它分成6个独立的分类：

1. 和类型相关的，例如：所有的结构和类对象应该有确定的内存布局。为了达成二进制层次上的交互（这里应该指的是不同版本Swift编译器生成的结果在二进制上兼容），它们必须共享相同的布局协议。这部分内容会在数据布局的章节进行讨论。
2. Swift可执行程序，运行时、反射机制、调试器以及可视化工具都和类型的metadata息息相关。因此，metadata应该有一种更稳定的读取方式，要不为类型的metadata设计确定的内存布局，

要不为访问类型的metadata提供一套稳定的APIs。这部分内容会在类型的metadata章节继续讨论。
3. 程序库中每一个被导出或来自外部的符号都需要一个唯一的名称，这个名称的识别应该在所有的二进制实体之间达成一致。由于Swift提供了函数重载以及上下文相关的名字空间（这里应该指的是通过module引入的名字空间），因此，在代码中出现的任何名称可能都不是全局唯一的。为了把它们表达成一个全局唯一的名字，Swift使用了一种叫做name mangling的技术。具体的name mangling方案会在Mangling章节中讨论。
4. 函数必须知道如何相互调用，因此我们需要约定调用栈在内存中是如何布局的，哪些寄存器会在调用间被保留。这些内容统称为函数的调用约定。这部分内容会在Calling Convention章节中讨论。
5. Swift发布的时候自带了一个运行时库，用来处理诸如动态类型转换、引用计数、类型反射等相关的工作。Swift程序在编译的时候会调用这些运行时中的API。因此，Swift运行时API也是Swift ABI的一部分。运行时API的稳定性会在运行时的章节中讨论。
6. 除此之外，Swift还自带了一个标准库，其中定义了很多公共的类型、结构和基于这些结构的方法。为了让这份自带的标准库可以被用不同版本Swift编写的程序调用，标准库也需要对外暴露一份稳定的API。因此，和标准库中定义的类型需要有确定的内存布局一样，Swift标准库的API也是Swift ABI的一部分。关于标准库ABI的稳定性会在标准库的章节中进行讨论。

数据布局

背景

首先，我们来定义一些术语。

- **对象（object）**是指某个类型的存储实体，它可以存储在内存中的某个位置，或存储在寄存器里。对象可以是`struct`/`enum`类型的值、`class`的实例、`class`实例的引用、`protocol`类型的值，甚至是closures。而在那些视`class`为全部的面向对象编程语言中，对象则就是指的`class`的一个实例，这是Swift有别于它们的地方；
- 对象的**数据成员（data member）**是指任意需要存放在类对象内存布局内的值。数据成员包括了一个对象所有的stored properties和associated values；
- **闲置位（spare bit）**是某种类型对象（的内存布局）中，没有被使用的部分。这些部分通常是为了对齐内存地址而填充的地址空间。稍后，会更深入讨论这个话题；
- 在对象的布局中，除了那些表示对象值的bit之外，还有**一类bit并没有实际的意义**。例如，对于一个包含3个`case`的传统C风格`enum`来说，它的值用两个bit就可以表示了（数字0，1，2分别对应三个case，它们对应二进制的00，01和10）。这时，这两个bit可以表示的第4个值3，它的二进制表示中的第一个1就是无意义的；

数据布局，也被称作类型布局，定义了一个对象的数据在内存中的布局。这包括了对象在内存中的大小，对象的对齐（稍后会定义）以及如何在对象中找到每一个数据成员。

如果编译器可以在编译期确定一个对象的布局，这个对象的布局就是静态的。如果对象的布局只有在运行时在可以确定，这类对象的布局就是不透明的（opaque layout）。我们会在opaque layout章节中深入讨论这类对象。

布局和类型的属性

在Swift里，对于每一个静态布局的类型T，ABI指定了计算以下内容的方式：

- 关于类型的对齐：对于`x: T`来说，对象`x`的起始地址对当前硬件平台的内存对齐值取模一定是

0（也就是说总在内存地址对齐的位置开始）；
- 关于类型的尺寸：一个类型对象的大小，按对象占用的字节数计算（可以是0），但是不包含填充在对象结尾的字节；
- 关于每个数据成员的偏移（如果可行）：每一个数据成员的地址，都是从对象起始地址开始计算的；

把计算对齐和对象大小的方式结合在一起，就是这个类型的对象占用内存时的步进计算方式，它等于把对象的尺寸按照内存对齐的大小向上取整一个单位（最小是1单位。例如，对象的大小是7，内存要求8字节对齐，那么对象占用的内存就向上调整到8）。这种计算方式对于在连续内存地址空间中排列对象（例如：数组）时很有帮助。

一些类型有以下两种有趣的属性：

- 如果一个类型仅仅用来存储数据（注：很多`struct`就是如此，但不是全部），在拷贝、移动或销毁这类对象时，就不会有额外的复杂语义。我们管这种类型叫做POD（Plain of Data），也叫做trivial type。这种类型的对象在拷贝时，直接复制它的值即可，销毁时，可以直接回收分配给它的存储资源。只有在一个类型的所有数据成员都是trivial type时，这个类型才是一个trivial type。
- 如果一个对象的地址没有被其它辅助设计的表结构（注：这里应该是指为了实现对象布局而引入的额外数据结构）引用，这种类型的对象就是可以按位移动的（bitwise ovable）。当一个对象要从一个地址拷贝到另外一个地址，并且原地址的对象已经不再需要的时候，就可以把原地址的对象按位拷贝到新地址，然后把原地址对象标记为不可用的状态。如果一个类型所有数据成员都是可以按位移动的，则这个类型的对象也是可以按位移动的。并且，所有的trivial type的对象都是可以按位移动的。

例如，一个`struct Point`，它有两个`Double`类型的属性`x`和`y`，表示平面上X轴和Y轴的坐标。此时，`Point`就是一个trivial type。复制`Point`对象的时候，我们只要按位拷贝对象的内容就可以，销毁的时候我们也无需做任何额外的工作。

再来看一个可以按位移动的非POD类型的例子，就是包含类对象引用的`struct`。在拷贝这类对象时，我们不能只是简单拷贝`struct`对象的值，还要`retain`其包含的类对象。而在销毁这类`struct`对象的时候，我们也要`release`其引用的类对象。但是，这类对象却是可以在不同的内存地址间移动的，只要每次移动后，我们都把原地址的对象标记为不可用，保持其包含的类对象总体引用计数不变就好了。

最后，来看一个不可按位移动的非POD类型的例子，就是一个包含`weak`引用的`struct`。所有的`weak`引用都是通过一个辅助表格维护的。因此，当它们引用的对象被销毁的时候，这些`weak` references才可以被设置成`nil`。当移动这类对象的时候，必须要更新表格中的`weak` reference，让它引用到新的对象地址。

不透明布局

当一个对象的布局只有在运行时才可以确定时，这类对象的布局就叫做不透明的。例如，一个泛型对象，我们就无法在编译期确定对象的布局。再有，就是一种更具适应性的类型（resilient type），我们会在下一节描述这个概念。

对于一个具有不透明布局的对象来说，它的大小、对齐，是否是一个POD类型或者是否可以按位移动都是通过查询它的value witness table来确定的。我们会在value witness table这一节中深入讨论这个话题。数据成员的偏移是通过查询类型的metadata得到的，我们会在value metadata的章节中讨论。拥有不透明布局的对象必须通过间接的方式传递，我们会在函数底层签名（Function Signature

Lowering）的章节中讨论。Swift运行时，通过一些指针和拥有不透明布局的对象进行交互，因此，这类对象必须是可以取地址的。我们会在抽象级别的章节中进一步进行描述。

在实践中，编译器可以在编译期对布局有部分的了解。例如，对于下面这样的`struct`：

```
struct Type<T> {
    var number: Int
    var object: T
}
```

在这种情况下，根据特定的布局算法，整数`number`的布局以及它在`struct`对象中的位置都是可以确定的。但是，泛型属性的存储却是不透明布局，因此，整个结构的大小和对齐都是不确定的。我们正在调研如何用更高效的方式布局这种"半透明"形式的组合SR-3722。这很可能会导致把不透明的部分放到布局的末尾以保证所有静态布局的部分可以正常计算偏移。

程序库的进化

程序库的进化，为所有公共类型的对象默认引入了更具适应性的布局方式。并提供了新的可注解方式，可以为了性能而放弃掉布局的灵活性。一个更具适应性的布局方式通过让对象的布局不透明化避免了很多二进制接口容易被破坏的缺点。更具适应性的布局方式可以更自由的进行修改和进化而不会破坏二进制的兼容性。我们可以重新安排公共数据成员在布局中的位置、或添加 / 删除公共的数据成员（用一个getter/setter计算属性来替代它）。而新的注解方式提供了放弃掉这些灵活性的能力，这可以为对象的布局带来更为严格的约束，以提到代码的编译以及对象的访问效率。

为了在一起分发的modules之间使用跨module的优化，这里有一个适应域（resilience domain）的概念。一个适应域就是一组相互锁定版本的modules，因此，这些modules的多个不同版本之间，是没有二进制兼容性的。在Resilience Domain这一节详细讨论了这个概念。

当一个具备适应性布局方式的类型离开它的适应域之后，这种类型对象的布局方式就应该是不透明的。而在一个适应域内部，这个要求就不存在了，对象的布局方式可以由类型自行指定，静态的或不透明的皆可（点此了解不透明布局）。

新的）注解方式可能会用在未来Swift标准库的类型上，届时，将会对这些注解引入版本化管理，但生成的程序库将会是二进制兼容的。这种行为将对ABI有什么影响仍旧在调查中SR-3911。

抽象级别

Swift中所有类型概念上都存在多个抽象级别。例如，一个`Int`的值是一个具象的类型，并且可以通过寄存器传递给函数。但是，同样的值也可以传递给一个泛型函数，此时参数`T`的布局就是不透明的。对于这种情况，参数必须通过间接的方式传递给函数，于是这个整数值就必须在调用函数的栈空间中，被提升成一个临时变量。因此，当一个值的语义是类型`T`的时候，就比它是一个整数值时有了更高的抽象级别。在多个抽象级别之间移动，是通过一个叫做reabstraction的过程完成的。

对于Swift中的很多类型来说，reabstraction仅仅是把值拷贝到内存中以便让它有址可寻。但对于诸如tuples和high-order function来说，reabstraction就复杂的多。我们会在tuples布局和函数低阶签名节中讨论分别解释这两种情况。

类型一览

接下来的内容，是Swift中各种类型还存在的问题，需要被进一步明确。

Structs

`struct`类型的布局算法应该充分利用存储空间，为此甚至可以不按照其数据成员被声明时的顺序来布局对象SR-3727。我们可能需要一个与声明顺序完全无关的算法，在不破坏二进制兼容性的前提下，重新布局源代码中的数据成员SR-3724。我们还需要考虑，默认情况下，我们是否一定需要让`struct`中的数据成员是可寻址的（也就是按字节对齐的），还是我们可以可以把多个数据成员在可能的情况下进行按位合并以节省空间更好SR-3725。

只要一个`struct`的所有数据成员和其包含的sub-struct成员都可以从`struct`对象中剥离出来，那么，一个0字节的`struct`对象无需占据任何空间。我们可能希望探索一下让`struct`类型的对齐限制在某一个固定的数值上（例如：绝大多数平台上，`struct`对象都按16字节对齐），是否可以在Swift的实现中，带来更多好处SR-3912。

Tuples

Tuple和匿名的`struct`是类似的，但是tuple可以呈现出更加结构化的类型适配特性：例如，一个类型是`(Bool, Bool)`的tuple可以在任何地方适配`(T, U)`这种形式约定的泛型类型。但是`(T, U)`比`(Bool, Bool)`有更高的抽象级别，因此，如果tuple的布局采用了更为激进的布局方式，就会面临更加昂贵的reabstraction成本。这包括把每一个成员切割出来并独立提升成可单独寻址的成员。

关于tuple的布局，还有另外一种声音，就是更简单的，按声明顺序排列的，并且以字节为单位对齐的布局算法。Tuples通常都用于管理小型的局部变量，几乎不需要以性能关键为理由而进行激进的优化。并且，在Swift里，C语言中固定大小的数组也是以tuples的形式引入的，这种简单的布局方式也使得各种tuples的实现方式更加一致。

因此，以牺牲reabstraction的成本为代价，把tuple仿照`struct`的方式更加激进的按位打包（bit-pack）起来是否值得，是我们应该进一步调研的问题SR-3726。

对于具有相同类型和结构的tuple来说，成员是否带有label不应该影响tuple对象的二进制兼容性。

Enums

一个`enum`类型的值可以有多种不同的形式，也就是我们定义的`case`。决定具体是哪个`case`是通过一个标记实现的，通常这个标记是一个整数，用来表示应该如何理解当前`enum`中存储的值。为了节约空间，这个标记可以存放在闲置位，或通过extra inhabitants来表示。

`@closed enums`表示那些不能稍后再添加case的枚举，具体可以分成以下几类：

- Degenerate - 至多有一个`case`，并且没有关联值的`enum`；
- Trivial - 没有关联值的`enum`；
- Single payload - 只有一个`case`有关联值的`enum`；
- Multi-payload - 有多个`case`有关联值的`enum`；

其中，degenerate enums不占据存储空间。Trivial enums只占据它们的标记占用的空间。

对于single payload enums来说，它有两部分内容，一部分是带有关联值的`case`，`enum`会有一个专门的区域存储关联值，叫做payload；而对于其他没有关联值的`case`，Swift会先尝试把区分这些`case`的标记存在payload的extra inhabitants中，如果不可行，就把标记补充在payload后面，但后面的这些标记并不会包含使用了payload的`case`。这时，payload的布局确保和那些不在payload中使用extra inhabitants的`enum`是兼容的。并且，在payloud后面存储标记还可能由于对齐的原因，使得当`enum`被聚合在其他类型中时，形成更高效的布局方式。

而multi-payload enums的布局算法要复杂的多，并且仍旧需要进一步开发SR-3727。这个算法可能需要尝试重新安排payload，以融合多个`case`、节约存储空间、提高代码执行效率或降低代码体积。例如，如果payload中，所有被ARC托管的部分都放在同一个区域，诸如拷贝这样的操作就可以直接完成，而无需在托管和非托管环境之间进行切换。

Enum raw values并不属于ABI的一部分，因为它们是通过getter和setter这两个计算属性实现的。`@objc` enums是和C兼容的，他们都是trivial enum。

标准库的不断进化还引入的`@open` enums （它也采用了更具适应性的布局方式）,它允许程序库的作者在不破坏二进制兼容性的条件下，给`enum`添加新的`case`或调整已有`case`的顺序。至于具体的实现方式则尚未确定。

Classes

当我们讨论类对象布局的时候，会涉及到两个概念：一个是类对象，它分配在系统堆内存中，表示对象的值；一个是类对象的引用，它是通过引用计数的方式管理的指针。

类对象

类对象的布局大部分是不透明的。这主要是用来解决二进制接口容易被破坏的恼人问题，也叫做“脆弱的基类问题（fragile base class problem）”。指的是对基类看似不大的改动都会破坏派生类的二进制兼容性。

一个non-final类对象或者class existential的运行时类型是无法在编译期确定的。为了便于执行动态类型转换，每个对象都必须存储一个指向它类型信息的指针，叫做`isa`指针。`isa`指针总是存储在类对象地址偏移0的位置。至于类型是如何表示的，以及应该包含哪些类型信息则由class metadata决定，我们会在metadata的章节进一步讨论。类似的，一个non-final class方法的调用也是无法在编译期确定的，这需要在根据运行时的类型信息进行动态派发。方法的派发会在后面的章节中专门讨论。

类对象的结构，作为ABI稳定性的一部分，会保证在`isa`字段后面，有一个机器字大小（word-size）的不透明数据，供运行时进行引用计数。但是，为了给Swift语言自身提供更大的灵活性以及为实现预留更多余地，这个不透明数据的格式和使用方式暂时还不会是ABI的一部分。相反的，会有一些运行时函数专门处理引用计数的操作。在未来，这个不透明数据的具体格式和用法可能会锁定在一个更有效率的访问方式上，届时，它将作为ABI的一项积累性更新（ABI-additive change）。

引用

`class`是引用类型。这意味着Swift中所有和类对象打交道的代码都是通过引用完成的，而引用这个概念，在二进制这个层面来看，就是指针。并且，这些指针统一由ARC进行管理。

Objective-C兼容的类对象（例如：那些Objective-C类的派生类或者从Objective-C引入的类）的引用必须保证和Objective-C运行时二进制兼容。因此，这样的引用是不透明的，它们除了保证用0表示`nil`以及不会使用extra inhabitants之外，没有任何其它约定。

而原生的，非Objective-C兼容的Swift类对象的引用并没有这个限制。Swift原声类对象的对齐方式是ABI的一部分，规定了闲置位一定是在引用的lower bits中。如果地址空间有限，平台也可以为引用在upper bits中定义闲置位，或在低端地址处定义extra inhabitants。

为了更有效的执行一些ARC操作，我们可能要研究一下在引用的闲置位中存储本地引用计数（local reference counts）SR-3728。当引用发生逃逸或本地的引用计数为0时，这些改变都应该写回对象。如果这些本地引用计数的实现会跨越ABI边界，这样的改动就必须以ABI累积更新的方式来实现，并对支持该特性的目标版本进行检查。

Existential Containers

如果你搞不清一些专门的术语，任何关于existential的讨论都会变得毫无进展。所以，让我们先围绕existential values，extential containers和witness建立一些背景知识。

在类型理论中，一个existential type描述了一个抽象类型的接口。Existential type的值，叫做existential value。当Swift中，对象的类型是`protocol`时，就会用到这个概念。存储或传递一个`protocol`类型的对象意味着对象在运行时的真实类型是不透明的（也就是编译期不可知的，因此我们也无法确定这类对象的布局）。但是，这类不透明的对象都有确定的接口，因为它们都遵从了特定的`protocol`。

一个遵从了特定`protocol`的类型一定包含其约定的所有方法，但是这些方法的地址是无法在编译期确定的，因为我们只有在运行时，才能确定这个`protocol`对应的真实类型。这种情况和non-final class引用是类似的，因此，也使用了类似的技术手段来解决。Protocol中约定的每一个被实现的方法的地址，都被保存在*witness table*中，我们会在后面的章节中专门讨论。

Existential containers存储单个`protocol`或多个`protocol`组合在一起的值，以及为了遵从每一个`protocol`而引入的*witness table*。如果`protocol`没有约定必须由`class`实现，containers中应该存储以下内容：

- 对象的值本身，这个值可以存放在containers内连的value buffer里，也可能存放在额外分配的内容中；
- 指向类型metadata的指针；
- 指向每一个witness table的指针；

对于那些限定了只能是`class`实现的`protocol`，containers中则会忽略metadata指针（因为对象自身包含指向自己类型信息的指针）以及多余的内连buffer。并且，这里还有一个特例`Any`，由于它没有遵从任何`protocol`，因此Any对象的containers中没有witness table指针。

在声明ABI稳定前，我们正在重新评估existential containers中内连的buffer大小SR-3729。与此同时，我们还在考虑为那些无法在buffer中存储而额外分配的内存空间添加copy-on-write支持SR-xxxx。我们还应该考虑existential参数“爆炸”的问题，例如：把一个existential参数转换成一个遵从了某个协议的泛型参数SR-xxxx。

声明稳定性

声明ABI稳定意味着锁定类型对象的布局方式，并且对如何处理程序库进化过程中需要解决的问题作出决定。最终落地的形态会是一份技术规范，其中定义了未来版本的编译器使用的布局算法，通过这些算法可以确保编译结果的二进制兼容性SR-3730。

对于上面讨论的所有领域，采用更为激进的方式对布局进行改进可能会放在ABI稳定之后。例如，我们会探索重新安排内嵌类型数据成员的位置，并把它和外部类型打包在一起。这样的改进会以ABI累积性更新的方式完成，以满足部署时的最少版本数量要求。这也就意味着module file需要跟踪每个类型的ABI版本信息。

可以在类型布局文档中找到Swift当前使用的类型布局，而这份文档中的描述可能在未来进行调整。

类型元数据

和布局描述了特定类型对象的布局方式类似，类型元数据包含了关于类型自身的信息，其中，可以读取哪些信息以及如何访问这些信息都属于Swift ABI的一部分。

Swift为每一个*具象类型（concrete type）*定义了元数据记录。具象类型包含了所有非泛型类型以及包含具象类型参数的泛型类型。这些元数据记录既可以由编译器生成，也可以推迟到运行时生成（例如：在泛型类型被实例化的时候）。下面章节中的每一部分，讨论了一种元数据中存储的类型信息。

为了稳定访问元数据的方式，一种潜在的方式是提供读写函数，这样可以给元数据的底层数据结构留有一些修改的余地。这样可以让很大一部分元数据的格式变得不透明。但对于那些性能关键部分的访问（例如：动态类型转换，调用witness table中的方法），这种通过中间途径读取信息的方式带来的性能损耗是不太能接受的。因此，我们将会固定这部分性能关键内容的二进制格式，并通过函数访问其他的类型元数据[SR-3923]。

元数据的表达方式中，总有很多我们希望清理掉的历史性人为因素[SR-3924]。我们还希望可以对元数据进行细微的调整以让它提供更多语义信息，以便在未来提供更多语言工具和语言特性，例如反射（reflection）[SR-3925]。这些工作中，一部分需要在声明ABI稳定之前完成，另一部分则可以作为积累型更新。

声明ABI稳定

声明ABI稳定意味着要为所有语言要素的metadata中，需要明确布局的部分提供一份明确的技术规格，这样，未来版本的编译器和工具就可以用相同的方法来读写它们。单独为这些布局明确的部分提供说明文档并不是必要的，但它可能帮助我们更好的理解这些设计。我们还可能在metadata中为未来会添加的功能预留更多空间[SR-3731]。

关于类型metadata的更多内容，可以参考Type Metadata docs，但其中的内容有可能会过时。

泛型参数

Swift有一套在编译期和运行时均实力不凡的泛型系统。由于Swift中的类型可以通过泛型类型参数化，每一个类型的metadata都应该描述它是否包含泛型类型，如果包含，还应该提供这些泛型类型的信息。

但在运行时，对象一定有一个具象类型的。如果在源代码中，对象是个泛型类型，运行时的具象类型就叫做对应泛型类型的一个实例化结果。实例化类型的metadata则提供了它的每一个泛型参数的类型信息。如果泛型类型带有`protocol`约束，metadata中还应该包含每一个作为约束的`protocol`的witness tables。

Value Metadata

带有名字的值类型会在metadata中存储类型的名称（现在，保存的名称是经过mangle处理的，并且，我们正在研究是否可以存储un-mangled的名称[SR-3926]）。如果值类型是一个内嵌类型，还会保存一个指向其父类型metadata的指针。

值类型的metadata还包含一些类型特定的条目。例如，`struct`的metadata中存储了它包含的字段、字段偏移，字段名称以及每一个字段的metadata。`enum`的metadata中则包含了它的每一个`case`，payload的尺寸，以及payload的metadata。Tuple的metadata则包含了它的每个成员以及成员的标签。

Value Witness Tables

每一个具象类型都有一个*value witness table*，它提供了布局这个类型对象以及访问这个类型值的方式。当一个值类型格式不透明的时候（opaque layout），这个类型对象的布局和值在编译期就是不可知的，这时就需要借助value witness table获取这些信息。

Value witness table可以表示一个类型是否是复杂类型，是否可以按位移动，是否包含额外的存储数据以及如何访问它们。例如，对`enums`来说，value witness table就还包含一个用于区分各种`case`的discriminator。我们也正在研究是否可以通过更简单高效的方式完成这个工作[SR-4332]。

类型的value witness table可以静态创建，对于一些泛型类型，也可以动态创建。Swift中每一个不同的类型都有一个元数据指针。只要元数据提供的信息相同（例如：类型布局完全相同），value witness table可以在不同的类型之间共享。Value witness tables总是以最高的抽象级别表达一个类型。为了声明ABI稳定，value witness table中包含的字段和结构必须被锁定[SR-3927]。

Class Metadata

在Apple的平台上，Swift中class的元数据和Objective-C class的元数据是布局兼容的，这就对class元数据中最开始部分的内容提出了一些要求。在这部分内容里，诸如父类指针、对象大小，对象对齐，一些特定的标志位以及Objective-C运行时需要的不透明数据都应该包含在内。

在以下章节中，分别描述了基类成员、基类元数据、泛型参数元数据、类成员以及*vtables*。标准库的进化过程可能会对以下描述的内容进行很多修改，为了配合这些修改，以下描述的对应内容也会随之变得不透明[SR-4343]。

Method Dispatch

调用一个类对象的non-final方法意味着调用一个在编译期不可知的方法，具体调用的方法必须在运行时才可以确定，这个过程是借助一个叫做*vtable*的表完成的，它也叫virtual method table（因为在派生类中可以改写的方法也叫做virtual method）。*Vtable*中的每一项都是一个函数指针，指向了一个instance method（注：这个method可以是类自己定义的，也可以是重写基类的）的实现。如果vatable确定成为ABI的一部分，它也需要通过布局算法的形式为标准库的进化保留可修改的灵活性。

或者，我们还可能通过一个不透明的*chunk*，或者由编译器生成的中间函数（intermediary functions），完成在module之间的方法调用。这样，我们就可以根据情况对方法的调用使用direct或vtable dispach方法[SR-3928]。这种方式为程序库的进化保留了更大的修改空间，一方面，这不会导致二进制兼容性问题；另一方面，还允许我们对类的内部结构进行修改。这种方式还在open和non-open类之间，统一了non-final方法的派发方式。与此同时，还允许我们执行激进的编译器优化，例如：为non-open classes中的方法调用采取de-virutalization措施。这种方法不需要vtable是ABI的一部分，因为这部分信息对其他module不透明是一种更有效的做法。

Protocol和Existential Metadata

Protocol Witness Tables

Protocol witness table是一个函数表，其中包含了一个类型遵从的protocol中约束的方法。如果protocol中还有关联类型（associated type），witness table中还会保存这个类型的元数据。当一个类型只有在运行时才能确定时，Protocol witness table就会搭配existential containers一同使用。

Protocol witness tables既可以在运行时动态创建，也可以通过编译器静态创建。Protocol witness table的布局是ABI的一部分。我们既要确定一个布局算法，又要兼顾标准库的进化需要，它可以允许我们为protocol的发展添加内容，又可以回退到默认的状态[SR-3732]。

Existential Metadata

Existential type metadata包含了以下内容：

- 类型持有的witness table的数量；
- 类型遵从的protocol是否只能由class实现；
- 和每一个遵从的protocol对应的*protocol descriptor*；

每一个protocol descriptor都和一个protocol对应，用于表示这个protocol是否只能由class实现、与这个protocol对应的witness table的大小，并包含了它改进的其它protocol的protocol descriptor。

在Apple的平台上，protocol descriptor的布局和Objective-C运行时的protocol记录是兼容的。而类型元数据的格式作为ABI定义的一部分，还需要进一步讨论[SR-4341]。

Function Metadata

除了一般的元数据之外，函数的元数据还包含了和函数签名相关的信息，例如：参数和返回值类型的元数据，调用约定（call convention），每个参数拥有权的约定以及函数是否抛出异常等。函数的元数据总是以最高抽象级别表示一个函数，我们会在function signature lowering section中解释这个话题。当前，函数参数是通过一个基于tuple的方式表达的，但这可能会根据Swift的发展作出修改[SR-4333]。只要我们对函数参数添加更多语义，就需要保存每一个参数的更多信息。

Mangling

Mangling用于产生唯一的符号。无论是外部符号，还是内部或隐藏的符号，都要经过mangling处理。但只有对外部符号的mangling处理才是ABI的一部分。

ABI稳定意味着一套完整稳定的mangling方法，未来的编译器和语言工具都可以基于这套方案来实现。对于当前使用的mangling方案，可以参考Name Mangling docs，但这份文档中的描述有可能在未来做出调整。

在声明ABI稳定前，当前的mangling方案中还有一些边缘情况需要处理。例如，我们需要规范化泛型和protocol类型的需求，让它们支持order-agnostic mangling [SR-3733]。我们还需更小心地处理可变函数参数的mangling方案，例如：[SR-3734]。但更多需要讨论和改进的，还是降低mangling生成符号的大小。

Mangling方案的设计集中在保持mangling结果简短高效的同时，还要保持符号的唯一性、并可以和已有工具和格式的兼容。由于程序库和框架中普遍存在着public symbol、应用程序中存在着调试符号，这些符号名称对二进制结果的大小有着重要的影响。因此，降低符号大小对二进制结果的影响是当前稳定mangling方案的一个主要考量。在声明ABI稳定之后，任何新的mangling方案和技术，都必须以累积型更新的方式提供，并支持已有的mangling方案。

有很多方法可以在改进mangling方案的同时，不会对当前工具造成大的影响。针对这些方法，我们将通过科学的测试方法测量和追踪生成符号的大小和它对二进制结果大小的影响。因此，mangling中，和ABI相关的部分主要是生成*compact manglings*以及使用*suffix differentiation*。

Compact Manglings

为了缩短mangling结果进行的局部调整对所有Swift编译出来的二进制结果都有好处。这些调整应该进一步压缩现有的mangling结果，又能够和之前的结果有唯一的映射关系。其中一个例子就是在mangling出来的结果中，不在区分struct和enum，这将会为标准库的进化带来更多的自由[SR-3930]。我们正在考虑丢掉witness table中的一些符号，它们并不会为调试提供更多有意义的支持[SR-3931]。最近，我们正在全面修订mangling过程中的字符替换方案，目的就是尽一切可能降低结果中的名称冗余[SR-4344]。

我们还在调查其它一些激进的方向，例如，基于已知的重载函数集合进行mangling。这种方式的缺点是当引入新的重载方法时，会导致mangling结果的不稳定，所以必须小心评估这种方法带来的好处[SR-3933]。

还有一些关于如何存储符号更有意义的思考，例如，对程序库中所有的名称进行整体压缩，但这种方式和先有的底层工具耦合性很高。不幸的是，在声明ABI稳定之后，这可能会让一些更有意义的选项变得不稳定。因此，这些思考可能会在确定了部署目标之后，在未来，以ABI积累型更新的方式发布。

Suffix Differentiation

现如今，已经存在很多底层工具和格式在存储和使用符号信息了，它们使用了一些有效率的存储技术，例如tries。Suffix differentiation是一种利用这些技术调整mangling结果的方法：通过后缀来区分mangling的结果，让符号共享一个公共的前缀。

调用约定

在这份文档里，“标准调用约定”指的是在某个平台上的C语言函数调用约定（可以参考appendix），而“Swift调用约定”则是指Swift函数之间调用时使用的约定。在这个领域里，实现ABI稳定的第一步，就是让Swift接受Swift调用约定[SR-4346]。现在，Swift运行时使用的是标准调用约定，但有可能在未来进行调整（参考Runtime calling convention）。

调用约定的稳定性只和public接口有关。Swift编译器可以自由选择内部方法的调用约定（无论是module内部的，还是module之间的）。

关于调用约定的详细内容，可以参考Swift Calling Convention Whitepaper，但其中描述的内容可能会过时。作为最终确定的调用约定的一部分，这份文档可能会更新成最终确定的规范，或者移动到一份更简洁明确的规范中。

寄存器约定

这一节，将会使用*callee-saved*和*scratch*这两个术语来分类寄存器，这种分类方式是寄存器约定的一部分。

- *Callee-saved寄存器*是指在函数的调用过程中必须被保留的寄存器。如果调用的函数（这个函数叫做*calee*）需要改变存储在calee-saved寄存器中的内容，它必须在返回前恢复寄存器中对应的内容；
- *Scratch寄存器*，也叫做caller-saved寄存器，它们的值无须在函数调用过程中保留。如果这些寄存器的值必须保留，就需要在对应函数调用指令前后，由调用者负责保存和恢复；

对于以上两类寄存器，Swift调用约定和标准调用约定的分类方式大体相同。但是，在一些平台上，Swift调用约定为一些额外的场景添加了calee-saved寄存器，它们是：*call context*寄存器和*error*寄存器。

Call Context Register

*Call context*寄存器中的值取决于调用的函数：

- 类实例方法：指向self的指针；
- 类方法：指向类型，或者其派生类的元数据指针；
- 值类型的mutating方法：指向值的指针（有些情况，值是需要间接传递的）；
- 值类型的non-mutating方法：整个值对象可以通过几个寄存器传递，如果过大就只能间接传递；
- *Thick closures*：调用closure时的上下文环境；

让call context寄存器由调用的函数负责保存是有好处的。这可以调用的上下文环境在多个调用之间保持稳定，以便于在接下来的多个调用或嵌套调用中被反复使用。另外，这种方式还可以让我们把thin closures转换为thick closures。

Error Register

在一些平台上，可以抛出异常的函数通过*error*寄存器向函数的调用者发送错误值。当发生错误时，error寄存器中包含一个指向错误值的指针，否则，它的值就是0。函数的调用者通过检查error寄存器是否为0来决定是否继续执行，或者跳转到对应的错误处理代码。让error寄存器是一种callee-saved寄存器，可以让我们把一个不会抛出异常的函数转换成一个会抛出异常的函数。

函数签名的底层转化

函数签名的底层转化（function signature lowering）是指从函数源代码到底层物理实现的一种映射，这个映射反应了函数参数和返回值的处理方法。例如，哪些值应该保存在寄存器里，哪些值应该保存在栈里。

ABI稳定要求对这些内容进行明确的约定，以便未来版本的Swift可以使用相同的物理签名方式[SR-4349]。更详细的描述和函数底层转化方式可以参考function signature lowering docs。

首先处理的，是函数的返回值，如果可以，就通过一组寄存器保存，否则，就把结果放在栈上。在特定的架构上，需要使用一个良好的启发式方法来解决某些限制问题（例如，一些流行的64位架构只有4个寄存器）[SR-3946]。

接下来要处理的是参数，我们只是想当然的尝试把参数从左到右依次放到寄存器里。当然我们也可能重新调整参数的顺序，例如，closure类型的参数最好放在所有参数后面。这样，可以利用thick closure和thin closure的ABI兼容性。

有些值是必须以间接方式传递和返回的，因为它们只是一个地址。这些值包括：non-bitwise-copyable values，values with opaque layout以及non-class-constrained existential values。即便运行时类型可以通过寄存器传递，或者可以通过调用语句在编译期确定，只要被调用的函数接收或返回布局不透明的类型，这些类型的值就必须采用间接访问的方式处理。

我们应该调查一下是否应该把一个半透明的布局分开，让其中透明的部分通过寄存器来传递[SR-3947]。

参数的所有权并不会通过物理调用约定体现，但它会在mangling的结果中被说明。参数的默认值表达式不是ABI的一部分，因为它可能被函数的调用者忽略。这意味着程序库的作者可以添加、修改或者删除参数的默认值，这并不会带来二进制不兼容性（但可能会带来源代码不兼容）。

高阶函数的底层转换

传递或返回高阶函数可能会导致reabstraction的发生，这就要求编译器必须生成在实际调用约定和期望调用约定之间映射的代码。

来看个例子，假设有下面这两个函数：

```
func add1(_ i: Int) -> Int { return i+1 }
func apply<T,U>(_ f: (T) -> U, _ x: T) -> U { return f(x) }
```

`apply`的函数类型参数`f`必须通过间接方式接收它的参数并返回值，因为`T`和`U`的布局都是不透明的。如果`add1`传递给`apply`，编译器就要为`apply`生成以间接方式接受`add1`参数的代码，但是在调用`add1`时，它的参数又是通过寄存器传递的。最后，编译器还要生成通过寄存器获取`add1`返回值的代码，这个返回值要以间接访问的方式返回给`apply`。

栈的不变性

调用约定还包括了调用栈的不变性，例如栈的对齐。除非有非常特别的原因，Swift应该总是遵循标准调用约定中栈不变性约定的。这是因为Swift有可能会和非Swift代码共享它们的调用栈。例如，一个Swift函数会调用Objective-C函数，而这个函数又调用了其它Swift函数。为了让这些调用正常工作，我们应该保持函数栈有相同的对齐方式（和其它栈不变性的要求）。因此，向标准调用约定看齐，会让这个工作简单很多。

运行时调用约定

Swift运行时使用标准调用约定，但它可能也会逐步进化以保留更多的不变性。例如，为一些运行时函数添加一个或多个scratch寄存器很可能是有好处的。调用运行时函数的Swift代码假定一些寄存器是无需保存的，它们可以被运行时函数修改。但也有些运行时函数并不需要很多scratch寄存器，它们可以假定更多寄存器属于calee-saved这一类。因此，在调用运行时函数的时候，每一个可以从scratch分类划归到callee-saved分类的寄存器，都可以缓解生成调用代码时的寄存器压力。

运行时函数的这些改变可以在未来逐步更新，只要这些函数没有修改当前最新版本中需要保存的寄存器，这些函数和之前的历史版本就一直是保持兼容的。但是，这样一个个函数的改变太过于细致，并且每一个在运行时不再被修改的寄存器都无法在不破坏二进制兼容性的条件下再重新变成可修改的。因此，如果减少可修改寄存器的数量会招致二进制兼容性问题，那么这个行为就是没意义甚至更多是有害的。因此，在做任何调整之前，必须仔细测试，以确保函数在未来绝不会需要更多的可修改寄存器。

Runtime

Swift为编译过的代码提供了很多运行时API。一些调用Swift运行时方法的代码，是由编译器生成的，例如：内存管理相关的以及获取运行时类型信息的。另外，运行时还暴露了一些底层反射API，这些API对一些特定的开发者很有用（例如标准库的作者）。

我们要审计每一个已存在的运行时函数的目标和行为[SR-3735]。对每一个函数来说，我们需要评估它是否如我们需要的一样：

- 如果是，我们就需要进一步精确指明和保障API的语义；
- 如果不是，我们就需要修改、删除或替换这个API，然后再精确指明新的语义；

Runtime还负责在运行时创建新的类型元数据信息（例如在泛型实例化的时候）。程序库的进化经常会导致类型的数据和元数据变得更加不透明，这就为运行时带来了一个新的需求，可以通过运行时API访问类型的数据和元数据。另外，所有权语义也需要新的运行时API，或者对已有API进行修改。具体需要哪些运行时功能，目前仍在进一步讨论[SR-4352]。

未来，还有很多潜在的方向可能会进一步扩大ABI的范围，并直接访问更加透明的数据，例如一个叫做call-site caching的技术。这些都会是ABI积累型更新，它们都是未来很有趣的研究方向。

关于运行时符号信息以及一些相关细节，可以查看Runtime docs，但其中描述的内容又可能已经过期。

标准库

在声明ABI稳定之后，任何一个标准库API必须在未来提供二进制兼容性。标准库还会利用resilience annotations和*可内连*代码。可内连代码和客户端调用代码是绑定在一起的。为了确保二进制兼容性，标准库需要面临下面的挑战（并没有详尽列出所有的问题）：

- 删除或修改任何已经发布的public函数和类型不能带来二进制兼容性问题；
- 内连代码会影响性能和灵活性；
- 可内连代码调用的internal函数是变成ABI的一部分，对这些函数的二进制兼容性要求和public函数是完全一样的；
- Non-resilient类型不能改变它们的布局；
- Protocols不能再添加新的需求；

可内连性

调用了internal函数的可内连代码让这些internal函数成为了ABI的一部分，因为客户端会在外部调用这些可内连代码。因此，标准库中的很多internal接口都需要被锁定。是否把代码标记为可内连需要仔细评估性能需求以及未来可修改的灵活性。

这种在性能和灵活性上的交换还会影响bug补丁以及性能提升代码的部署能力。哪些使用了标准库内连代码的用户将无法通过更新操作系统获得bug修复或性能提升，它们只能把自己的代码和标准库一起重新编译。对这部分内容的详细讨论，可以参考Inlineable Functions。

接下来的改变

由于标准库已经确认了源代码级别的稳定，今年，主要的目标将是底层实现方式的改进。当ABI稳定后，标准库对已经存在的API和non-resilient类型的修改空间将会非常有限。因此，摆正标准库的位置是一项极其重要的工作。

String类型的编程模型仍旧在重新设计中[SR-4354]，很多其它类型（例如Int）也在进行一些实现方式的调整[SR-3196]。与此同时，标准库也在切换新的编译器特性上，例如，通过conditional conformance清理代码并提供最好的APIs [SR-3458]。

另外一个目标是提高使用Swift进行系统编程的能力。所有权的语义可能会有比较大的影响，包括，经过改进的`inout`语义允许更高效和安全的进行数组分片。我们还在调查高效访问连续内存区域的正确抽象方式[SR-4355]。

接下来的步骤

所有的进展和issue跟踪都会通过bugs.swift.org上的JIRA完成，并加上"AffectsABI"的标签。我们将会提供一个看板来更简单的监控ABI稳定性问题的细节和进展。下一步，就是完成每一个需要修改的issue，并探索每一个我们期望探索的方向。

这份文档将会在ABI稳定之前保持更新，添加在JIRA上的新发现。当声明ABI稳定之后，这份文档就应该是完整的Swift ABI规范了。

但是，只是追踪issue并不能有效的沟通ABI达成稳定的整个过程。其中一些issue要比其它issue花费更多时间，但没有一个好的表达方式表明完成issue需要花费多少时间，或者还有多少未知的issue需要被记录下来。因此，我们需要提供一个更高级别的视图来展示ABI稳定的整体进展，这应该会发布在swift.org上。

附录

标准ABIs

Apple ARM64 iOS platform ABI是ARM's AAPCS64的一份供应商定制规格。

Apple ARM32 iOS platform ABI是和ARM's AAPCS类似的一个变体。

Apple x86-64 MacOS platform ABI是基于通用的System V ABI修改的，也用于BSD和Linux。

Apple i386 MacOS platform ABI是基于通用的i386 System V ABI修改的版本。