



iOS成长之路

作者 / SwiftOldDriver

2017
春

目录

- 序：这是一本什么书
- Swift 之父 Chris Lattner 访谈录（上）
作者：@故胤道长，亚马逊 iOS 工程师，曾就职于 Uber
博客：<http://www.jianshu.com/u/8d5b91490ca5>
- HTTP 2.0 的那些事
作者：MrPeak，公众号：MrPeakTech，曾就职于阿里巴巴
- Auto Layout：Snapkit 源码剖析
作者：@没故事的卓同学，丁香园 iOS 工程师
博客：<http://www.jianshu.com/u/88a056103c02>
- 在服务端写 Swift 是一种什么样的体验
作者：郑宇琦(Enum)，就职于中国科学院宁波工业技术研究院，微博：@提拉拉拉就是技术宅
博客：<http://enumsblog.com>
- Fastlane 实战演练
作者：@一缕殇流化隐半边冰霜，曾就职于平安付，微博@halfrost
博客：<http://www.jianshu.com/u/12201cdd5d7a>
- Fastlane 的神秘花园
作者：@Onetaway，NGS iOS工程师，曾就职于暴走漫画
- CFArray 的历史渊源及实现原理
作者：冬瓜，微博：@冬瓜争做全栈瓜，爱奇艺 iOS 工程师
博客：<http://www.desgard.com>
- load 方法全程跟踪
作者：冬瓜

这是一本什么书

iOS 的入门资料在互联网上可以用泛滥来形容。如果你有计算机基础，想快速学习 iOS，做几个界面很轻易就能做到。然而移动创业热潮退去后，对 iOS 开发者的能力要求也越来越高。当一个已经入门的开发者，想要成为一个更好的 iOS 开发者的时候，就会发现互联网的资料太琐碎，而且资料的好坏也难辨。常常都会困惑我要如何提高自己，哪里有好的学习资料。

这就是这本书尝试解决的问题。我们从一线资深工程师的角度出发，找出我们认为高质量的、对于提高 iOS 开发者技术水平有帮助的文章收录进来。

面向读者

笼统的说就是面向初中级 iOS 开发者。你可能是一个刚刚入门能够熟练使用 UITableView 的新手，也可能是做了一两年对于 UIKit 和 Foundation 的 API 都已经很熟悉了，想要提高自己的开发者。

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。

鸣谢

感谢猫叔设计的封面。苟富贵勿相忘。

Chris Lattner 访谈录（上）

话题

- Chris Lattner 是谁？
- Xcode 的编译器 LLVM 背后有怎样的故事？
- Swift 诞生的前世今生，封闭的苹果为何要拥抱开源？
- 说好的 ABI 稳定性何时能推出？

Chris Lattner 是谁



教育背景

- 伊利诺伊大学 PHD

工作经历

- 2005年 - 2017年供职苹果，前开发部高级总监，架构师
- 2017年开始，担任特斯拉副总裁，负责自动驾驶

主要成就

- Swift 之父，主要作者

- LLVM 之父，主要作者
- Clang 主要贡献者

荣誉

- 2016年被评为“创造未来的25位当世天才”
- 2013年获得 ACM 系统设计大奖

访谈实录

自我介绍

1. 你怎么看待自己？

我是个程序员。我喜欢写代码。我编程有很长时间了。

我在读博的时候就开始写 LLVM 了。当时 LLVM 是我的博士研究项目，我想把它做成工业界中颠覆性的产品。当时我异想天开，尝试了各种架构设计，想解决以往编译器所有的弊端 -- 结果当然没有如愿。我毕业后，就希望能接着搞 LLVM，当时只有苹果允许我入职之后继续设计并实现 LLVM。我想都没想就加入了苹果。

LLVM

2. 说说 LLVM (Low Level Virtual Machine) 到底是什么吧

- 先说编译器：编译器是把程序员的代码翻译成机器可以理解的语言的工具；
- 再谈LLVM：一个模块化和可重用的编译器和工具链技术的集合，Clang 是 LLVM 的子项目，是 C，C++ 和 Objective-C 编译器，因为多模块的复用，所以提供了惊人的快速编译，比 GCC 快3倍。

3. LLVM 是一开始就作为一个完整的编译工具来使用的吗？还是有什么其他故事

LLVM 当时是为了解决一个小问题而开发的：当使用OpenGL 函数库的时候（Mac OS 10.4 和 10.5环境下），比如你要调用这个函数，glVertex3f()，编译器必须将其转化为特定的GPU可以理解的数据。但是这带来一个问题：市面上有海量的GPU，每个GPU的性能和参数也不尽相同，所要求的数据格式也不同。这时 LLVM 可以产生很小的一部分代码去解决这个问题，这是 LLVM 诞生的初衷。

4. LLVM 的 bytecode 和 Apple 现在的 bitcode 有什么不同？

这是历史遗留问题。一开始 LLVM 是开源的，所有代码在转成二进制时就叫做 bytecode -- 因为 java 当年就是这么叫的。当时这一部分有很多问题：比如不能扩展，无法兼容，非常脆弱。

然后就到了 LLVM 2.0，当时我重新设计了架构，采用的就是 Bitcode 机制。LLVM 2.0 将所有代码以比特流(bit stream)而不是字节流(byte stream)的形式来编码。这就是 bitcode 这一术语的由来。

主要的工作流程就是现将代码转成比特流，然后相应处理。处理完后再将编码传到其他地方去。

5. Bitcode 这个机制比直接传输二进制有什么好处

好处那是多了去了。首先 编译器工作起来会越来越好。因为通过Bitcode机制，它可以通过编译不同代码来存储各种优化方法，这样下次碰到类似代码，它就会自动启动相关优化机制，使得效率提升。还有个好处是 LLVM 可以让芯片的兼容性变得很好。因为 Apple 每年都在芯片上推陈出新，它们转化为二进制的规则都不尽相同，LLVM 只要每次重新编码并传输成比特流就好了。

当然 Bitcode 也不是万能的。比如它不能解决 32位的 APP 在64位机器上的兼容问题。这个问题其实应该依靠代码逻辑。

谈管理

6. 在职业生涯中，你在 LLVM 上鞠躬尽瘁，但我们发现这几年你更多的工作是在管理上，你自己怎么看这种转变的？

我虽然做管理了，但是我依然喜欢写代码，而且我每天都写，因为我就是个极客嘛。而且，其实我很早就开始做管理的工作了。不过我一直作为技术领导人的角色带 2 到 3 个人的，我只是在写代码方面把把关，给他们提提建议这样。

后来带的人多了，队伍也大了。我不仅管程序员，还管小组经理和其他技术领导人。虽然我一直喜欢写代码，但是管理对我来说是一个必须要去做的事。现在回过头来，我觉得干得还不错。跟大家一起工作之后我知道很多事协同工作效果更好，和同事交流你就会理解他们的想法，这样我就可以制定更好的计划路线。

其实我没感觉整个过程有什么不同。直到今天我还夜以继日、废寝忘食得写代码，我并不是坐那边动动嘴皮子，指挥别人干活的老板。我其实每个周末都在写代码，我很忙的。

Swift

7. Swift 是如何诞生的？在苹果这样一个大厂，决定做出如此巨大的变革，同时还是在封闭的环境下，你是如何一步步实现的？

首先，苹果内部所有的项目都不尽相同 -- 工作流程、战略规划、实施细节，到最后发布。Swift 也一样，没有可比性。因为苹果本身就是小组单兵作战模式 -- 每个组负责不同的大方向，组里自己计划和工作，甚至招人都是各自招。

言归正传，契机发生在2010年了。当时好像是我们刚刚完成了 Clang 对 C++的支持。你也知道 C++ 写起来有多丑，但是做个编辑器支持 C++，完善 C++ 这门语言就是另一回事了，我们当时搞了好久终于完成的时候特别有成就感。当然 Clang 远没有到达完美的地步。

我又扯远了。除了做 Clang 以外，无论是 C语言，C++，还是 Objective-C，都有一些我不是很满意的地方。所以我就想要不我们搞个新的语言来吧。新的语言要越简单越好。一开始大家都没认真，后来我跟很多同事聊了之后觉得新语言的计划可行，而且大家都很亦可赛艇。于是我们就用业余时间开始顶层设计和写代码。

现在问题来了，因为我们已经有 Objective-C 了。虽然它有几个地方很丑，比如老是用 "@", 每句结束了还要打分号，但是这些并不妨碍它是一门伟大的语言。所以，我们为什么要开发新语言，而不是把精力花在优化 Objective - C 上？



原因有三。

第一，如果我们大幅优化 Objective - C，把很多 Swift 的特性加进去，这对开发者来说是灾难性的，因为他们要对原来的 APP 要进行大幅修改；

第二，Objective - C 很多特性积重难返，比如它安全性上的问题；

第三，Objective - C 是基于 C 开发的语言，所以你无论怎么优化，它必然有 C 语言自身的缺陷。

于是我们就动手做 Swift 了，它的背后有着数百人的努力：支持 Xcode，开发 Playground，兼容调试器和编译器。我个人感到最骄傲的一点是，我们并不打算自己内部把它做到完美 -- 我们开源、我们依靠社区，这样一门语言才能在无数开发者的实战中得到检验和改进，我想这才是 Swift 最棒的地方。

8. 你之前在优化 Objective-C 的时候，有没有想到什么地方是未来 Swift 可以用得到的？

ARC。我们其实一直都在争论是用垃圾回收机制（garbage collection）还是 ARC，后来决定了是 ARC。

另一个是模块化，我们也将这一部分的经验带到了 Swift 开发中。

其实，很多数组和字典方面的语法优化本来是计划在 Objective - C 上面的。但是后来我们开发了 Swift，于是这些改进被直接用在了新语言上，所以大家会在写 Swift 的时候觉得似曾相识，因为本来这些就是 Objective-C 的升级版本嘛。

我可以透露一个有意思的事情。我们在做 Swift 的时候，很多 iOS 开发者，包括苹果内部的工程师，都在吐槽我们这几年在 Objective - C 上毫无建树，都在说你们为什么不做这个那个。我们当然不能告诉他们我们在全力开发 Swift，而他们所要的语法功能我们都会给。

9. 苹果内部对于 Swift 的使用情况和开发是怎么看得？

Swift 团队对于开发上有明确的目标和计划，应用二进制接口（ABI）的稳定性一直是我们的首要目标。很多人很喜欢我们开源的 Swift Playground。同时 iOS 系统内置的 Music App 也是 Swift 写的。其实用不用 Swift 主要是技术和开发方面的考量，苹果内部同时得兼顾稳定性和开发效率，这不是说大家喜不喜欢这个语言的问题。

Swift 刚发布的时候，内部很多组都很惊讶：我们已经有了 Objective-C，为什么还要搞新的 Swift？而且 Objective-C 本身就很不错，开发起来也很顺手。后来渐渐 Swift 成熟了，大家也爱上了这个新生儿。

内部其实对于 Swift 一个很大的顾虑在于，苹果的所有开发必须兼容32位机器，而32位的应用都采用了 Objective-C 的 runtime 机制。这就要求 Swift 团队也弄出个类似的机制，或者弄个兼容的方案，否则 Swift 无法与 AppKit 适配。

10. 开源后的 Swift 发展态势喜人，你对此有什么看法？

开源之后，Swift 发展之好让我咋舌，然而这也是问题所在。

当年我们开源了 LLVM 和 Clang，它们也发展喜人。我们的对手 AMD 们完全跟不上我们。但是跟 Swift 比起来，它们的发展也太慢了，LLVM 和 Clang 开源后完全没有 Swift 这么火。

Swift 就不同了，开源一年之后，我们就有了上百万的开发者在使用这门语言 -- 我和很多有丰富开源经验的老工程师都吓了一跳，这简直了！然后我们每天收到无数的邮件和 pull requests，要求更新这个、要求优化那个，我们的节奏完全被打乱了。我们如何规划开发？我们如何把 Swift 的开发导向一个正确的方向？这些问题随着时间的推移和经验的积累，慢慢找到了解决之道。

我现在觉得开源这个决定至关重要。一来大家会帮着优化；二来我们有个巨大的论坛，在那里大家可以畅所欲言，全世界的人都在帮着 Swift 进步，这真的很棒。我们虽然没有一开始就具体计划要开源，但是苹果内部当时都觉得 Swift 肯定有一天要开源。

苹果与特斯拉



11. 苹果好像一直是个封闭的公司，你们内部对于开源怎么看？

苹果其实有开源的传统。LLVM 虽然不是始于苹果，但是最终是苹果完成并将其开源。Clang 则完全是生于斯开源于斯。还有其他工具，比如 LLDB，libc+，以及compiler-rt 都是如此。

所以对于 Swift 来说，开源只是时间问题。当年从 Swift 1.0 到 Swift 2.0，一切都乱七八糟。当时我们重点在开发错误处理机制，还有协议、拓展等一系列重要的功能。所以开源 Swift 1.0，并不是一个好选择，因为这些重要的东西都没有，而这些开发是当务之急。当 Swift 2.0 到来的时候，我们才有空去开源、去做社区拓展和论坛搭建。开源社区可以帮我们修复细节，我们这时候可以更多的投入在架构设计上。

12. 苹果最让你怀念的是什么？

苹果是这样一个公司，你可以选择你喜欢的东西，然后努力工作去实现它，最终你的工作会落实在产品上，影响亿万计的人。

有很多公司，你可以努力工作，但是不一定能做你喜欢的东西；你做出东西，可能会被束之高阁；你做的产品，也许最后很幸运的发布，但是并不一定有很多人会用。在苹果，你的工作可以真正改变世界，很有成就感。

13. 你得到特斯拉之后，还会努力为 Swift 做出贡献吗？

特斯拉的工作非常有挑战性，这是我最开心的地方。我现在还没入职，所以也不知道我之后对 Swift 能做多少工作。也许我还会夜以继日的发 Pull Request，也许我就周末写写 Swift 代码。我应该会从各个方面 -- 无论是顶层设计还是具体代码实现，与苹果的核心团队合作，为这个语言做贡献。

其实我一直想说，Swift 只是我在苹果工作的一小部分，我花了大量的时间在其他事情上。**实际上在苹果我也就晚上或者周末有空写写 Swift**。我希望到了特斯拉之后我还能花同样的精力和时间在 Swift 上，毕竟我对这门语言统治世界充满期待。

ABI 稳定性

14. 现在 Swift 已经到了第3个版本了。我们也知道ABI稳定性的追求一直是你们的目标，但是它也一直被各种事情拖延。你对此有什么计划吗？或者说你从拖延中学到了什么经验教训吗？

ABI 推迟有两个原因。

第一是因为 Swift 的开发进程中有很多不确定性。当 Swift 开源之时，一堆人对我们提 pull request，提各种各样的 issue。这样我们就不得不去花大量的时间去维护开源社区，而不是专心去做计划内的工作。

第二个原因是，尽管稳定的 ABI 很重要，但是对于开发者来说，稳定的 ABI 对他们来说没有明显的好处，他们更关心是语法和兼容上的稳定和优化。所以我们后来修改了计划，语法和兼容上的稳定性被定为是最先要实现的目标。这样当 Swift 3.1 或者 Swift 4.0 出来的时候，大家不用担心语言上的转化会让 Xcode 崩溃，或是需要大家整个重构 APP。Swift 3.0 主要就是实现这个目标。



Convert to Latest Swift Syntax?

The target "PhoneRTC" contains source code developed with an earlier version of Swift. Choose "Convert" to update the source code in this target to the latest Swift syntax. This action can be performed later using "Convert to Latest Swift Syntax" in the Edit menu.

Cancel

Convert

15. 稳定的 ABI 什么时候推出？他会赶在异步和并发模型之前吗？

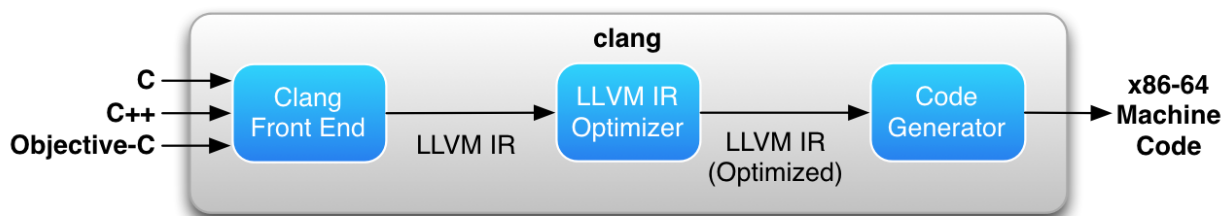
Swift 现有的内存管理机制对 ABI 稳定性造成了不小的影响。有些底层逻辑还需要调整，比如 getter 和 setter 的生成以及属性的内存分配问题，苹果内部正在做这件事，这之后我们才能完成 ABI。至于并发模型啥的就跟 ABI 没有关系了。

很多人担心 Swift 4.0 的时候苹果能不能推出稳定的 ABI，因为毕竟工作量太大。ABI 的工作正在井然有序地进行，而且对于开源社区来讲推出稳定的 ABI 至关重要。Ted（Chris Lattner 之后的 Swift 领导人）有一件事说对了，现在 Swift 当务之急就是让编译器更稳定，让错误处理更方便，提高编译速度，并且将 Swift 扩展到大规模系统中。

我在想 Swift 4.0 的时候究竟能看到什么。也许没有稳定的 ABI，但是一定会有重要的新功能加入。

ABI 将允许未来 Swift 版本开发的应用程序和编译库可以在二进制层次上与 Swift 3.0 版本的应用程序和编译库相互调用。这样，ABI 的稳定性将保证一定程度的二进制兼容性，并且第三方更容易发布二进制库。另外，ABI 将允许删除需要的 Swift 标准库和二进制文件，就像目前情况下通过 Xcode 创建的 iOS 和 OS X 应用程序一样。

补充



补充：LLVM的三层结构

- 第一层：Clang 编译器，负责编译各种语言
- 第二层：代码优化器，通过模块化操作优化代码，是 Bitcode 逻辑的主要部分
- 第三层：代码翻译器，针对不同平台和 GPU 将代码翻译成机器语言

补充：LLDB, llc++, compile rt

- LLDB: 一个有着 REPL 的特性和 C++ ,Python 插件的开源调试器。LLDB 绑定在 Xcode 内部，存

在于主窗口底部的控制台中；

- libc++, libc++ ABI: 高性能 C++ 标准库实现，支持 C++ 11
- compiler-rt: 为 LLVM 和 Clang 设计的编译器扩展函数库。针对 __fixunsdftdi 和其他目标机器上没有核心 IR (intermediate representation) 对应的短原生指令序列时，提供高度调优过的底层代码生成支持。

ABI 是什么？

Application Binary Interface，中文名：应用二进制接口。是 APP 和 操作系统、其他应用之间的二进制接口。它包括以下细节：

- 数据类型的大小、布局和对齐；
- 调用约定（控制着函数的参数如何传送以及如何接受返回值），例如，是所有的参数都通过栈传递，还是部分参数通过寄存器传递；哪个寄存器用于哪个函数参数；通过栈传递的第一个函数参数是最先push到栈上还是最后；
- 系统调用的编码和一个应用如何向操作系统进行系统调用；
- 以及在一个完整的操作系统ABI中，目标文件的二进制格式、程序库等等。

参考链接

音频: [ACCIDENTAL TECH PODCAST 205](#)

原文: [PEOPLE DON'T USE THE WEIRD PARTS](#)

[Chris Lattner's Homepage](#)

[深入剖析 iOS 编译 Clang LLVM](#)

[与调试器共舞 - LLDB 的华尔兹](#)

[compiler-rt" runtime libraries](#)

[应用二进制接口](#)

HTTP 2.0 的那些事

在我们所处的互联网世界中，HTTP协议算得上是使用最广泛的网络协议。最近http2.0的诞生使得它再次互联网技术圈关注的焦点。任何事物的消退和新生都有其背后推动的力量。对于HTTP来说，这力量复杂来说是各种技术细节的演进，简单来说是用户体验和感知的进化。用户总是希望网络上的信息能尽可能快的抵达眼球，越快越好，正是这种对“快”的追逐催生了今天的http2.0。

1. HTTP站在TCP之上

理解http协议之前一定要对TCP有一定基础的了解。HTTP是建立在TCP协议之上，TCP协议作为传输层协议其实离应用层并不远。HTTP协议的瓶颈及其优化技巧都是基于TCP协议本身的特性。比如TCP建立连接时三次握手有1.5个RTT（round-trip time）的延迟，为了避免每次请求的都经历握手带来的延迟，应用层会选择不同策略的http长链接方案。又比如TCP在建立连接的初期有慢启动（slow start）的特性，所以连接的重用总是比新建连接性能要好。

1.1 HTTP应用场景

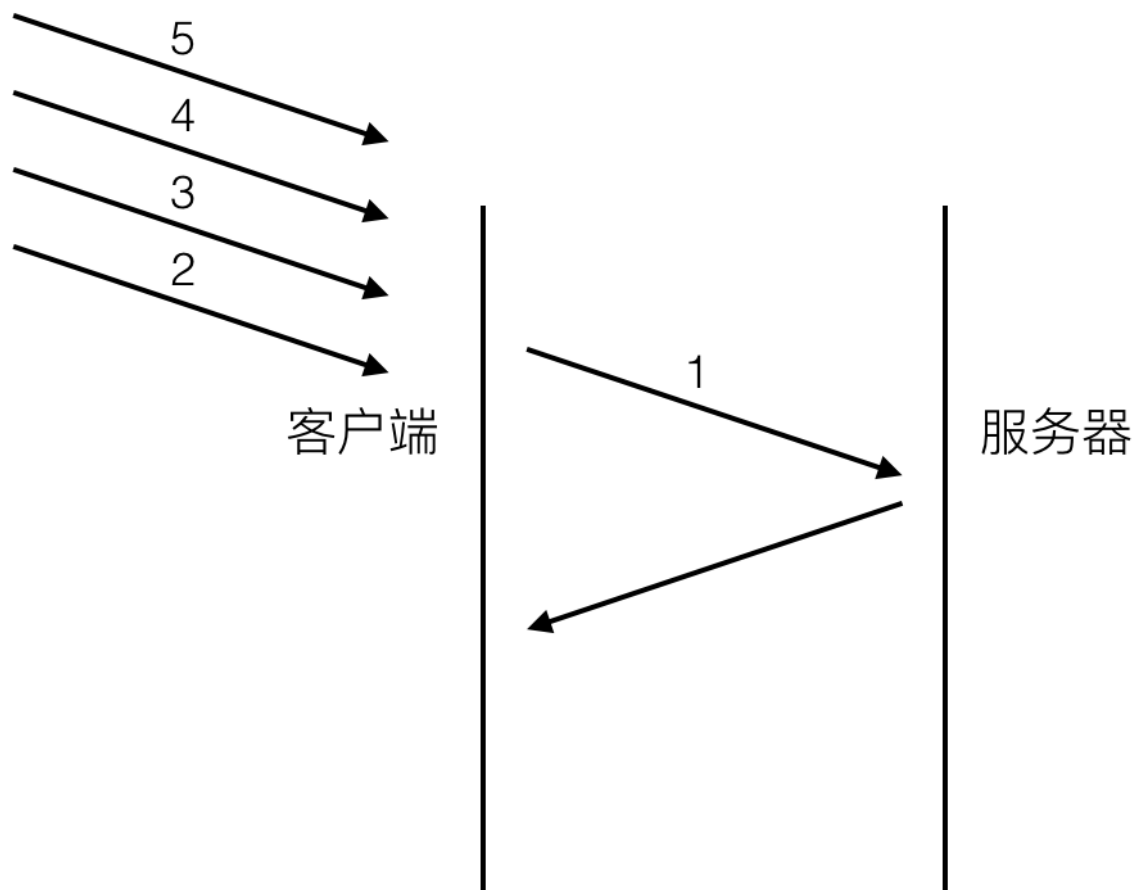
http诞生之初主要是应用于web端内容获取，那时候内容还不像现在这样丰富，排版也没那么精美，用户交互的场景几乎没有。对于这种简单的获取网页内容的场景，http表现得还算不错。但随着互联网的发展和web2.0的诞生，更多的内容开始被展示（更多的图片文件），排版变得更精美（更多的css），更复杂的交互也被引入（更多的js）。用户打开一个网站首页所加载的数据总量和请求的个数也在不断增加。今天绝大部分的门户网站首页大小都会超过2M，请求数量可以多达100个。另一个广泛的应用是在移动互联网的客户端app，不同性质的app对http的使用差异很大。对于电商类app，加载首页的请求也可能多达10多个。对于微信这类IM，http请求可能仅限于语音和图片文件的下载，请求出现的频率并不算高。

1.2 因为延迟，所以慢

影响一个网络请求的因素主要有两个，带宽和延迟。今天的网络基础建设已经使得带宽得到极大的提升，大部分时候都是延迟在影响响应速度。http1.0被抱怨最多的就是连接无法复用，和head of line blocking这两个问题。理解这两个问题有一个十分重要的前提：客户端是依据域名来向服务器建立连接，一般PC端浏览器会针对单个域名的server同时建立6~8个连接，手机端的连接数则一般控制在4~6个。显然连接数并不是越多越好，资源开销和整体延迟都会随之增大。

连接无法复用会导致每次请求都经历三次握手和慢启动。三次握手在高延迟的场景下影响较明显，慢启动则对文件类大请求影响较大。

head of line blocking会导致带宽无法被充分利用，以及后续健康请求被阻塞。假设有5个请求同时发出，如下图：



对于http1.0的实现，在第一个请求没有收到回复之前，后续从应用层发出的请求只能排队，请求2，3，4，5只能等请求1的response回来之后才能逐个发出。网络通畅的时候性能影响不大，一旦请求1的request因为什么原因没有抵达服务器，或者response因为网络阻塞没有及时返回，影响的就是所有后续请求，问题就变得比较严重了。

1.3 解决连接无法复用

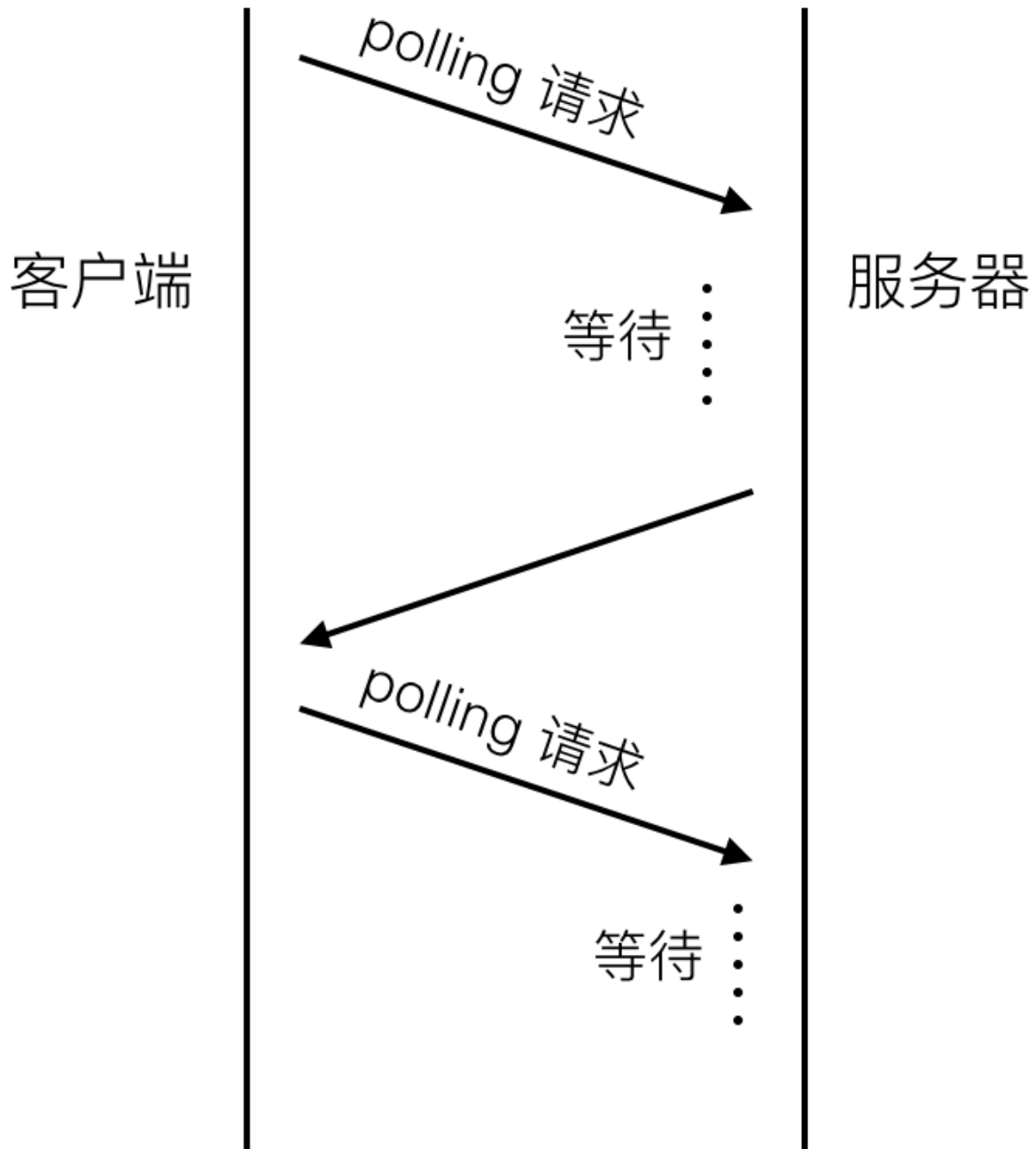
http1.0协议头里可以设置Connection:Keep-Alive。在header里设置Keep-Alive可以在一定时间内复用连接，具体复用时间的长短可以由服务器控制，一般在15s左右。到http1.1之后Connection的默认值就是Keep-Alive，如果要关闭连接复用需要显式的设置Connection:Close。一段时间内的连接复用在PC端浏览器的体验帮助很大，因为大部分的请求在集中在一小段时间以内。但对移动app来说，成效不大，app端的请求比较分散且时间跨度相对较大。所以移动端app一般会从应用层寻求其它解决方案，长连接方案或者伪长连接方案：

方案一：基于tcp的长链接

现在越来越多的移动端app都会建立一条自己的长链接通道，通道的实现是基于tcp协议。基于tcp的socket编程技术难度相对复杂很多，而且需要自己制定协议，但带来的回报也很大。信息的上报和推送变得更及时，在请求量爆发的时间点还能减轻服务器压力（http短连接模式会频繁的创建和销毁连接）。不止是IM app有这样的通道，像淘宝这类电商类app都有自己的专属长连接通道了。现在业界也有不少成熟的方案可供选择了，google的protobuf就是其中之一。

方案二：http long-polling

long-polling可以用下图表示：

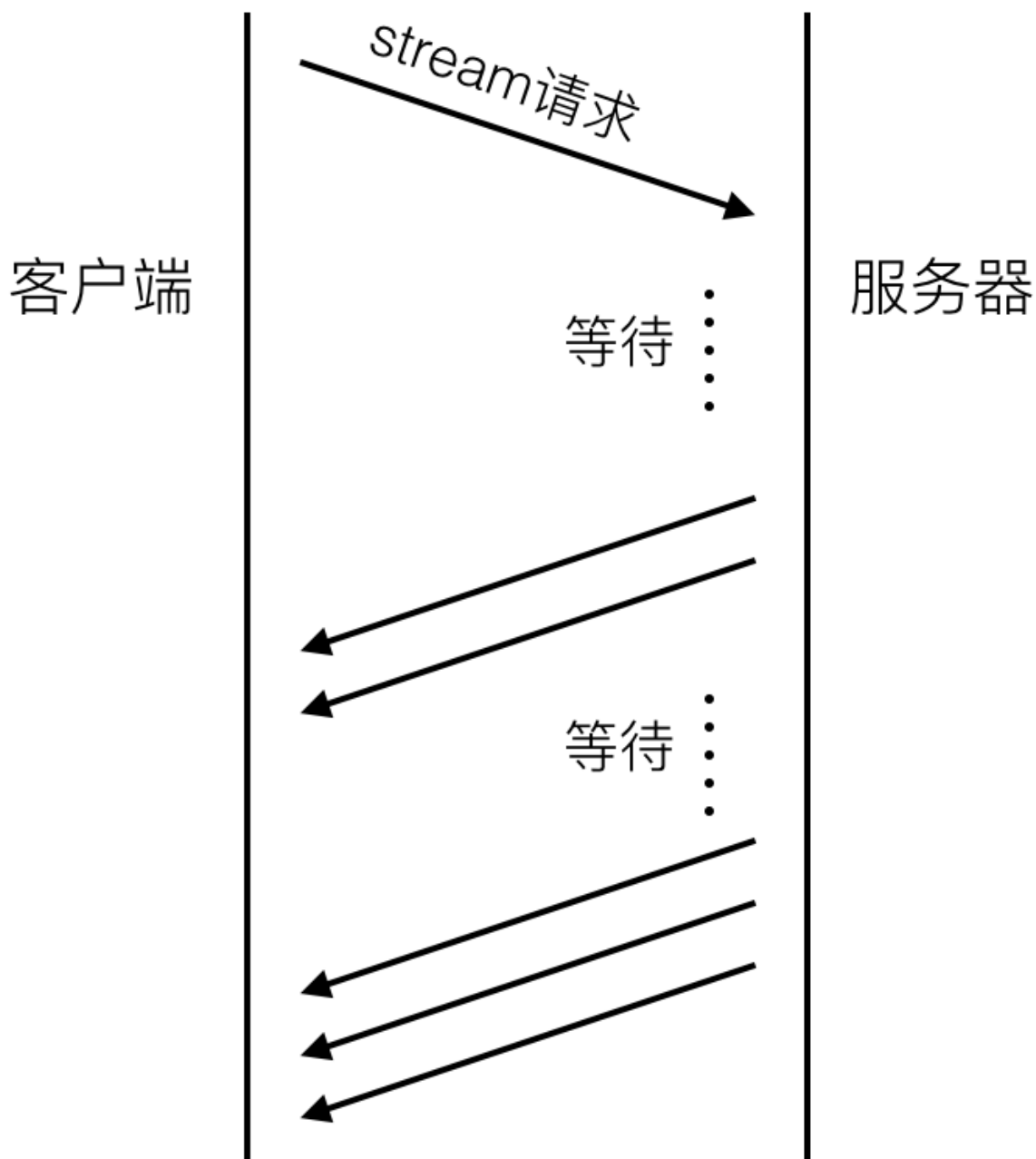


客户端在初始状态就会发送一个polling请求到服务器，服务器并不会马上返回业务数据，而是等待有新的业务数据产生的时候再返回。所以连接会一直被保持，一旦结束马上又会发起一个新的polling请求，如此反复，所以一直会有一个连接被保持。服务器有新的内容产生的时候，并不需要等待客户端建立一个新的连接。做法虽然简单，但有些难题需要攻克才能实现稳定可靠的业务框架：

- 和传统的http短链接相比，长连接会在用户增长的时候极大的增加服务器压力
 - 移动端网络环境复杂，像wifi和4g的网络切换，进电梯导致网络临时断掉等，这些场景都需要考虑怎么重建健康的连接通道。
 - 这种polling的方式稳定性并不好，需要做好数据可靠性的保证，比如重发和ack机制。
 - polling的response有可能会被中间代理cache住，要处理好业务数据的过期机制。
- long-polling方式还有一些缺点是无法克服的，比如每次新的请求都会带上重复的header信息，还有数据通道是单向的，主动权掌握在server这边，客户端有新的业务请求的时候无法及时传送。

方案三：http streaming

http streaming流程大致如下：



同long-polling不同的是，server并不会结束初始的streaming请求，而是持续的通过这个通道返回最新的业务数据。显然这个数据通道也是单向的。streaming是通过在server response的头部里增加“Transfer Encoding: chunked”来告诉客户端后续还会有新的数据到来。除了和long-polling相同的难点之外，streaming还有几个缺陷：

- 有些代理服务器会等待服务器的response结束之后才会将结果推送到请求客户端。对于streaming这种永远不会结束的方式来说，客户端就会一直处于等待response的过程中。
- 业务数据无法按照请求来做分割，所以客户端没收到一块数据都需要自己做协议解析，也就是说要做自己的协议定制。
streaming不会产生重复的header数据。

方案四：web socket

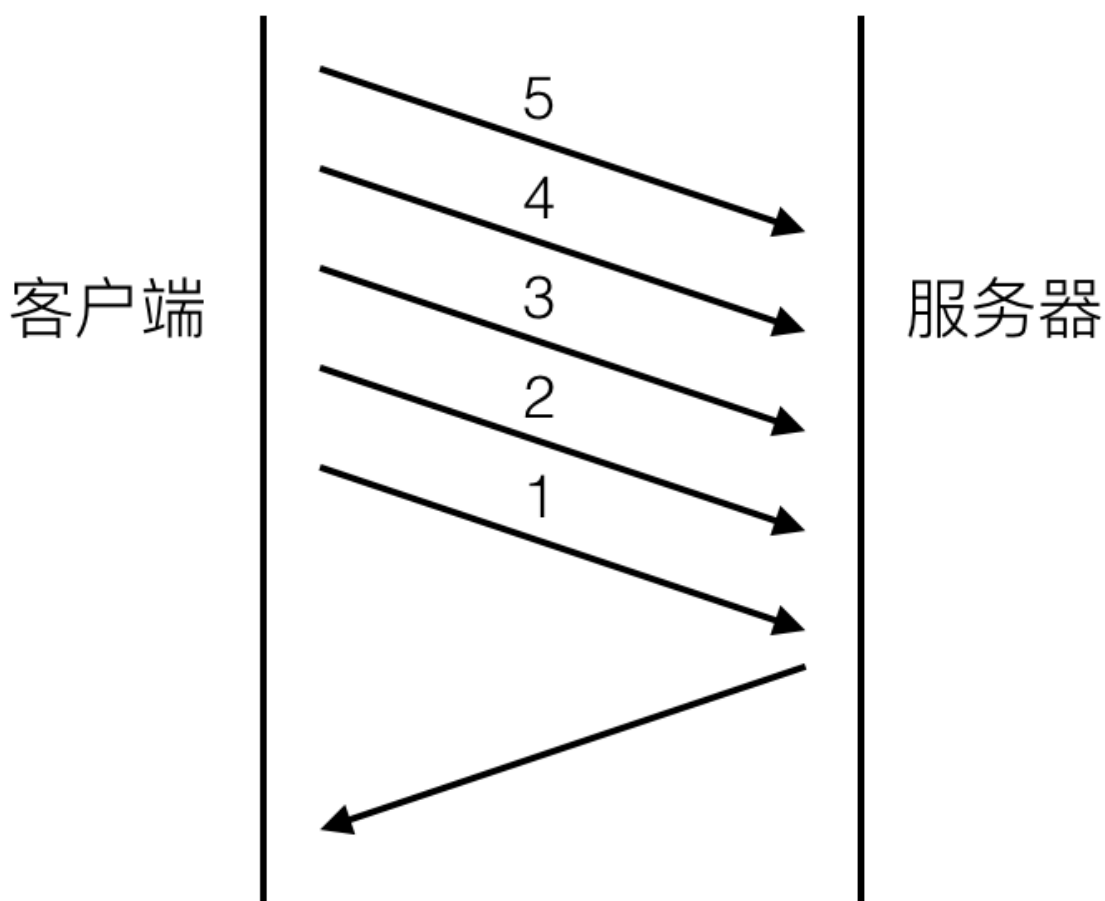
WebSocket和传统的tcp socket连接相似，也是基于tcp协议，提供双向的数据通道。WebSocket优势在于提供了message的概念，比基于字节流的tcp socket使用更简单，同时又提供了传统的http所缺少的长连接功能。不过WebSocket相对较新，2010年才起草，并不是所有的浏览器都提供了支持。各大浏览器厂商最新的版本都提供了支持。

1.4 解决head of line blocking

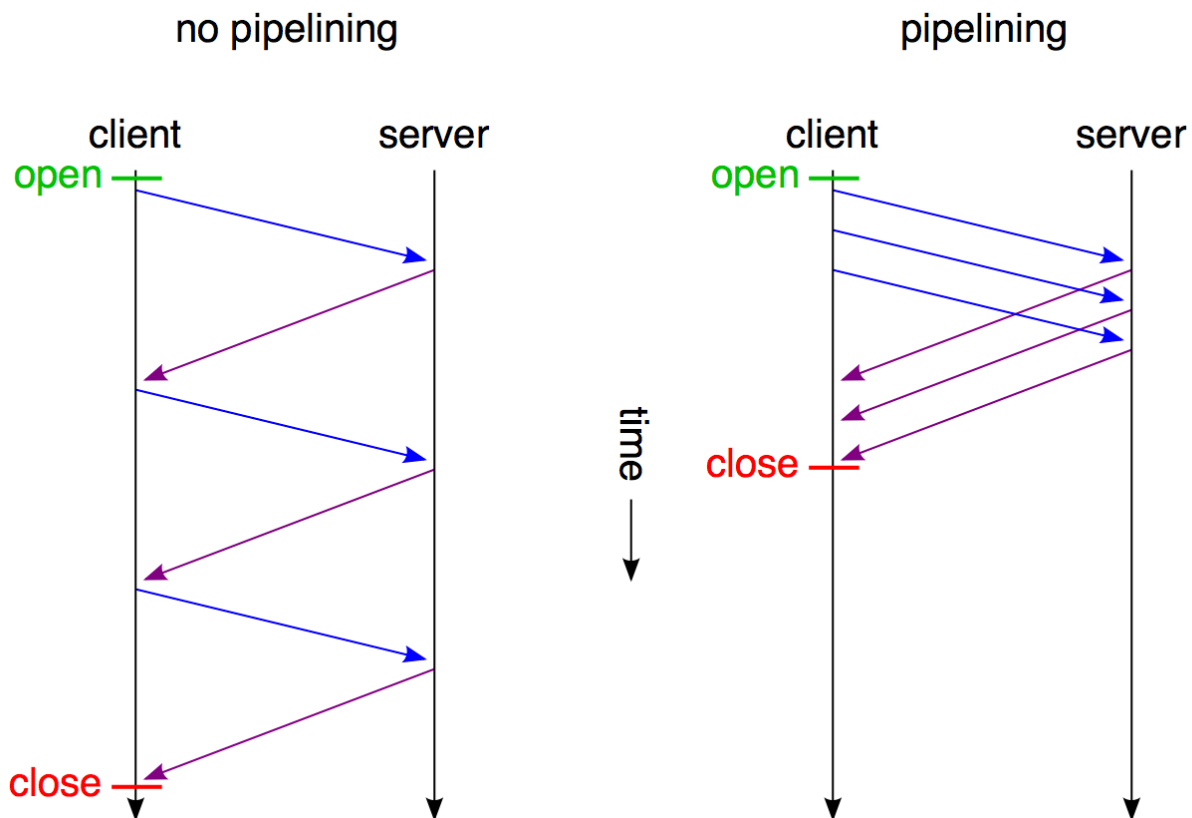
Head of line blocking(以下简称为holb)是http2.0之前网络体验的最大祸源。正如图1中所示，健康的请求会被不健康的请求影响，而且这种体验的损耗受网络环境影响，出现随机且难以监控。为了解决holb带来的延迟，协议设计者设计了一种新的pipelining机制。

http pipelining

pipelining的流程图可以用下图表示：



和图一相比最大的差别是，请求2，3，4，5不用等请求1的response返回之后才发出，而是几乎在同一时间把request发向了服务器。2，3，4，5及所有后续共用该连接的请求节约了等待的时间，极大的降低了整体延迟。下图可以清晰的看出这种新机制对延迟的改变：



不过pipelining并不是救世主，它也存在不少缺陷：

- pipelining只能适用于http1.1，一般来说，支持http1.1的server都要求支持pipelining。
- 只有幂等的请求（GET，HEAD）能使用pipelining，非幂等请求比如POST不能使用，因为请求之间可能会存在先后依赖关系。
- head of line blocking并没有完全得到解决，server的response还是要求依次返回，遵循FIFO(first in first out)原则。也就是说如果请求1的response没有回来，2，3，4，5的response也不会被送回来。
- 绝大部分的http代理服务器不支持pipelining。
- 和不支持pipelining的老服务器协商有问题。
- 可能会导致新的Front of queue blocking问题。

正是因为有这么多的问题，各大浏览器厂商要么是根本就不支持pipelining，要么就是默认关掉了pipelining机制，而且启用的条件十分苛刻。可以参考chrome对于pipelining的[问题描述](#)。

1.5 其它奇技淫巧

为了解决延迟带来的苦恼，永远都会有聪明的探索者找出新的捷径来。互联网的蓬勃兴盛催生出了各种新奇技巧，我们来依次看下这些“捷径”及各自的优缺点。

Spriting（图片合并）

Spriting指的是将多个小图片合并到一张大的图片里，这样多个小的请求就被合并成了一个大的图片请求，然后再利用js或者css文件来取出其中的小张图片使用。好处显而易见，请求数减少，延迟自然低。坏处是文件的粒度变大了，有时候我们可能只需要其中一张小图，却不得不下载整张大图，cache处理也变得麻烦，在只有一张小图过期的情况下，为了获得最新的版本，不得不从服务器下载完整的大图，即使其它的小图都没有过期，显然浪费了流量。

Inlining（内容内嵌）

Inlining的思考角度和spriting类似，是将额外的数据请求通过base64编码之后内嵌到一个总的文件当中。比如一个网页有一张背景图，我们可以通过如下代码嵌入：

```
background: url(data:image/png;base64,)
```

data部分是base64编码之后的字节码，这样也避免了一次多余的http请求。但这种做法也有着和spriting相同的问题，资源文件被绑定到了其它文件，粒度变得难以控制。

Concatenation（文件合并）

Concatenation主要是针对js这类文件，现在前端开发交互越来越多，零散的js文件也在变多。将多个js文件合并到一个大的文件里在做一些压缩处理也可以减小延迟和传输的数据量。但同样也面临着粒度变大的问题，一个小的js代码改动会导致整个js文件被下载。

Domain Sharding（域名分片）

前面我提到过很重要的一点，浏览器或者客户端是根据domain（域名）来建立连接的。比如针对www.example.com只允许同时建立2个连接，但mobile.example.com被认为是另一个域名，可以再建立两个新的连接。依次类推，如果我再多建立几个sub domain（子域名），那么同时可以建立的http请求就会更多，这就是Domain Sharding了。连接数变多之后，受限制的请求就不需要等待前面的请求完成才能发出了。这个技巧被大量的使用，一个颇具规模的网页请求数可以超过100，使用domain sharding之后同时建立的连接数可以多到50个甚至更多。

这么做当然增加了系统资源的消耗，但现在硬件资源升级非常之快，和用户宝贵的等待时机相比起来实在微不足道。

domain sharding还有一大好处，对于资源文件来说一般是不需要cookie的，将这些不同的静态资源文件分散在不同的域名服务器上，可以减小请求的size。

不过domain sharding只有在请求数非常之多的场景下才有明显的效果。而且请求数也不是越多越好，资源消耗是一方面，另一点是由于tcp的slow start会导致每个请求在初期都会经历slow start，还有tcp 三次握手，DNS查询的延迟。这一部分带来的时间损耗和请求排队同样重要，到底怎么去平衡这二者就需要取一个可靠的连接数中间值，这个值的最终确定要通过反复的测试。移动端浏览器场景建议不要使用domain sharding，具体细节参考[这篇文章](#)。

2.开拓者SPDY

http1.0和1.1虽然存在这么多问题，业界也想出了各种优化的手段，但这些方法手段都是在尝试绕开协议本身的缺陷，都有种隔靴搔痒，治标不治本的感觉。直到2012年google如一声惊雷提出了SPDY的方案，大家才开始从正面看待和解决老版本http协议本身的问题，这也直接加速了http2.0的诞生。实际上，http2.0是以SPDY为原型进行讨论和标准化的。为了给http2.0让路，google已决定在2016年不再继续支持SPDY开发，但在http2.0出生之前，SPDY已经有了相当规模的应用，作为一个过渡方案恐怕在还将一段时间内继续存在。现在不少app客户端和server都已经使用了SPDY来提升体验，http2.0在老的设备和系统上还无法使用（iOS系统只有在iOS9+上才支持），所以可以预见未来几年spdy将和http2.0共同服务的情况。

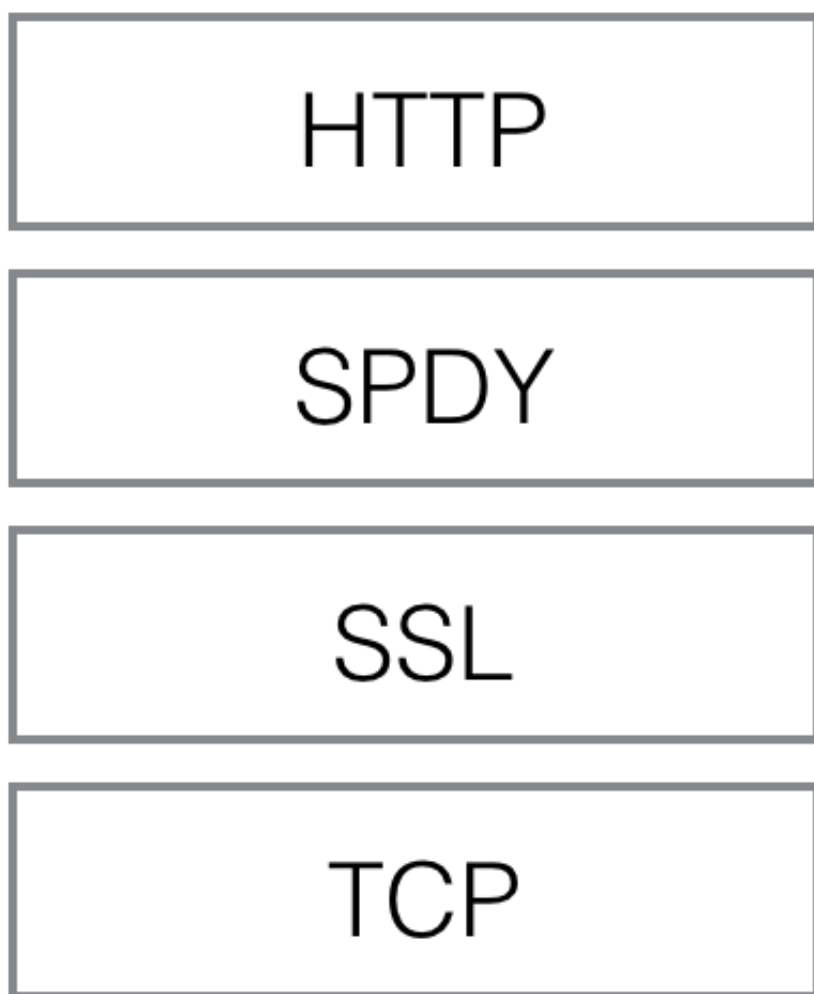
2.1 SPDY的目标

SPDY的目标在一开始就是瞄准http1.x的痛点，即延迟和安全性。我们上面通篇都在讨论延迟，至于安全性，由于http是明文协议，其安全性也一直被业界诟病，不过这是另一个大的话题。如果以降低延迟为目标，应用层的http和传输层的tcp都是都有调整的空间，不过tcp作为更底层协议存在已达数十年之久，其实现已深植全球的网络基础设施当中，如果要动必然伤经动骨，业界响应度必然不高，所

以SPDY的手术刀对准的是http。

- 降低延迟，客户端的单连接单请求，server的FIFO响应队列都是延迟的大头。
- http最初设计都是客户端发起请求，然后server响应，server无法主动push内容到客户端。
- 压缩http header，http1.x的header越来越膨胀，cookie和user agent很容易让header的size增至1kb大小，甚至更多。而且由于http的无状态特性，header必须每次request都重复携带，很浪费流量。

为了增加业界响应的可能性，聪明的google一开始就避开了从传输层动手，而且打算利用开源社区的力量以提高扩散的力度，对于协议使用者来说，也只需要在请求的header里设置user agent，然后在server端做好支持即可，极大的降低了部署的难度。SPDY的设计如下：



SPDY位于HTTP之下，TCP和SSL之上，这样可以轻松兼容老版本的HTTP协议(将http1.x的内容封装成一种新的frame格式)，同时可以使用已有的SSL功能。SPDY的功能可以分为基础功能和高级功能两部分，基础功能默认启用，高级功能需要手动启用。

SPDY基础功能

- 多路复用（multiplexing）。多路复用通过多个请求stream共享一个tcp连接的方式，解决了http1.x holb（head of line blocking）的问题，降低了延迟同时提高了带宽的利用率。
- 请求优先级（request prioritization）。多路复用带来一个新的问题是，在连接共享的基础之上有可能导致关键请求被阻塞。SPDY允许给每个request设置优先级，这样重要的请求就会优先

得到响应。比如浏览器加载首页，首页的html内容应该优先展示，之后才是各种静态资源文件，脚本文件等加载，这样可以保证用户能第一时间看到网页内容。

- header压缩。前面提到过几次http1.x的header很多时候都是重复多余的。选择合适的压缩算法可以减小包的大小和数量。SPDY对header的压缩率可以达到80%以上，低带宽环境下效果很大。

SPDY高级功能

- server推送（server push）。http1.x只能由客户端发起请求，然后服务器被动的发送response。开启server push之后，server通过X-Associated-Content header（X开头的header都属于非标准的，自定义header）告知客户端会有新的内容推送过来。在用户第一次打开网站首页的时候，server将资源主动推送过来可以极大的提升用户体验。
- server暗示（server hint）。和server push不同的是，server hint并不会主动推送内容，只是告诉有新的内容产生，内容的下载还是需要客户端主动发起请求。server hint通过X-Subresources header来通知，一般应用场景是客户端需要先查询server状态，然后再下载资源，可以节约一次查询请求。

2.2 SPDY的成绩

SPDY的成绩可以用google官方的一个数字来说明：页面加载时间相比于http1.x减少了64%。而且各大浏览器厂商在SPDY诞生之后的1年多里都陆续支持了SPDY，不少大厂app和server端框架也都将SPDY应用到了线上的产品当中。

google的官网也给出了他们自己做的一份测试数据。测试对象是25个访问量排名靠前的网站首页，家用网络%1的丢包率，每个网站测试10次取平均值。结果如下：

	DSL 2 Mbps downlink, 375 kbps uplink		Cable 4 Mbps downlink, 1 Mbps uplink	
	Average ms	Speedup	Average ms	Speedup
HTTP	3111.916		2348.188	
SPDY basic multi-domain* connection / TCP	2242.756	27.93%	1325.46	43.55%
SPDY basic single-domain* connection / TCP	1695.72	45.51%	933.836	60.23%
SPDY single-domain + server push / TCP	1671.28	46.29%	950.764	59.51%
SPDY single-domain + server hint / TCP	1608.928	48.30%	856.356	63.53%
SPDY basic single-domain / SSL	1899.744	38.95%	1099.444	53.18
SPDY single-domain + client prefetch / SSL	1781.864	42.74%	1047.308	55.40%

不开启ssl的时候提升在 27% - 60%，开启之后为39% - 55%。这份测试结果有两点值得特别注意：

连接数的选择

连接到底是基于域名来建立，还是不做区分所有子域名都共享一个连接，这个策略选择上值得商榷。google的测试结果测试了两种方案，看结果似乎是单一连接性能高于多域名连接方式。之所以出现这种情况是由于网页所有的资源请求并不是同一时间发出，后续发出的子域名请求如果能复用之前的tcp连接当然性能更好。实际应用场景下应该也是单连接共享模式表现好。

带宽的影响

测试基于两种带宽环境，一慢一快。网速快的环境下对减小延迟的提升更大，单连接模式下可以提升至60%。原因也比较简单，带宽越大，复用连接的请求完成越快，由于三次握手和慢启动导致的延迟损耗就变得更明显。

出了连接模式和带宽之外，丢包率和RTT也是需要测试的参数。SPDY对header的压缩有80%以上，整体包大小能减少大概40%，发送的包越少，自然受丢包率影响也就越小，所以丢包率大的恶劣环境下SPDY反而更能提升体验。下图是受丢包率影响的测试结果，丢包率超过2.5%之后就没有提升了：

	Average ms		Speedup
Packet loss rate	HTTP	SPDY basic (TCP)	
0%	1152	1016	11.81%
0.5%	1638	1105	32.54%
1%	2060	1200	41.75%
1.5%	2372	1394	41.23%
2%	2904	1537	47.7%
2.5%	3028	1707	43.63%

RTT越大，延迟会越大，在高RTT的场景下，由于SPDY的request是并发进行的，所有对包的利用率更高，反而能更明显的减小总体延迟。测试结果如下：

	Average ms		Speedup
RTT in ms	HTTP	SPDY basic (TCP)	
20	1240	1087	12.34%
40	1571	1279	18.59%
60	1909	1526	20.06%
80	2268	1727	23.85%
120	2927	2240	23.47%
160	3650	2772	24.05%
200	4498	3293	26.79%

SPDY从2012年诞生到2016停止维护，时间跨度对于网络协议来说其实非常之短。如果HTTP2.0没有出来，google或许能收集到更多业界产品的真实反馈和数据，毕竟google自己的测试环境相对简单。但SPDY也完成了自己的使命，作为一贯扮演拓荒者角色的google应该也早就预见这样的结局。SPDY对产品网络体验的提升到底如何，恐怕只有各大厂产品经理才清楚了。

3.救世主HTTP2.0

SPDY的诞生和表现说明了两件事情：一是在现有互联网设施基础和http协议广泛使用的前提下，是可以通过修改协议层来优化http1.x的。二是针对http1.x的修改确实效果明显而且业界反馈很好。正是这两点让IETF（Internet Engineering Task Force）开始正式考虑制定HTTP2.0的计划，最后决定以SPDY / 3为蓝图起草HTTP2.0，SPDY的部分设计人员也被邀请参与了HTTP2.0的设计。

3.1 HTTP2.0需要考虑的问题

HTTP2.0与SPDY的起点不同，SPDY可以说是google的“玩具”，最早出现在自家的chrome浏览器和server上，好不好玩以及别人会不会跟着一起玩对google来说无关痛痒。但HTTP2.0作为业界标准还没出生就是众人瞩目的焦点，一开始如果有什么瑕疵或者不兼容的问题影响可能又是数十年之久，所以考虑的问题和角度要非常之广。我们来看下HTTP2.0一些重要的设计前提：

- 客户端向server发送request这种基本模型不会变。
- 老的scheme不会变，使用http://和https://的服务和应用不会做任何更改，不会有http2://。
- 使用http1.x的客户端和服务端可以无缝的通过代理方式转接到http2.0上。
- 不识别http2.0的代理服务器可以将请求降级到http1.x。

因为客户端和server之间在确立使用http1.x还是http2.0之前，必须要确认对方是否支持http2.0，所以这里必须要有个协商的过程。最简单的协商也要有一问一答，客户端问server答，即使这种最简单的方式也多了一个RTT的延迟，我们之所以要修改http1.x就是为了降低延迟，显然这个RTT我们是无法接受的。google制定SPDY的时候也遇到了这个问题，他们的办法是强制SPDY走https，在SSL层完成这个协商过程。ssl层的协商在http协议通信之前，所以是最适合的载体。google为此做了一个tls的拓展，叫NPN（Next Protocol Negotiation），从名字上也可以看出，这个拓展主要目的是为了协商下一个要使用的协议。HTTP2.0虽然也采用了相同的方式，不过HTTP2.0经过激烈的讨论，最终还是没有强制HTTP2.0要走ssl层，大部分浏览器厂商（除了IE）却只实现了基于https的2.0协议。HTTP2.0没有使用NPN，而是另一个tls的拓展叫ALPN（Application Layer Protocol Negotiation）。SPDY也打算从NPN迁移到ALPN了。

各浏览器（除了IE）之所以只实现了基于SSL的HTTP2.0，另一个原因是走SSL请求的成功率会更高，被SSL封装的request不会被监听和修改，这样网络中间的网络设备就无法基于http1.x的认知去干涉修改request，http2.0的request如果被意外的修改，请求的成功率自然会下降。

HTTP2.0协议没有强制使用SSL是因为听到了很多的反对声音，毕竟https和http相比，在不优化的前提下性能差了不少，要把https优化到几乎不增加延迟的程度又需要花费不少力气。IETF面对这种两难的处境做了妥协，但大部分浏览器厂商（除了IE）并不买帐，他们只认https2.0。对于app开发者来说，他们可以坚持使用没有ssl的http2.0，不过要承担一个多余的RTT延迟和请求可能被破坏的代价。

3.2 HTTP2.0主要改动

HTTP2.0作为新版协议，改动细节必然很多，不过对应用开发者和服务提供商来说，影响较大的就几点。

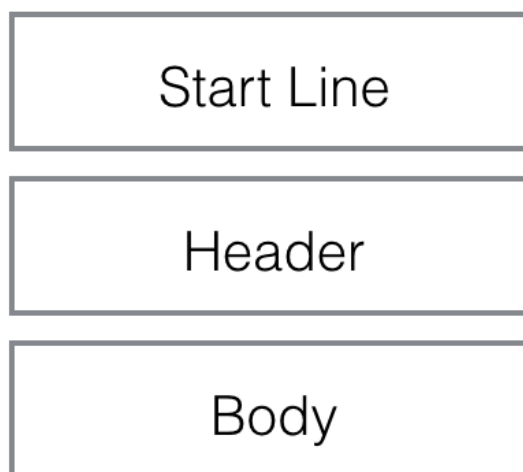
新的二进制格式（Binary Format）

http1.x诞生的时候是明文协议，其格式由三部分组成：start line（request line或者status line），header，body。要识别这3部分就要做协议解析，http1.x的解析是基于文本。基于文本协议的格式解析存在天然缺陷，文本的表现形式有多样性，要做到健壮性考虑的场景必然很多，二进制则不同，只认0和1的组合。基于这种考虑http2.0的协议解析决定采用二进制格式，实现方便且健壮。

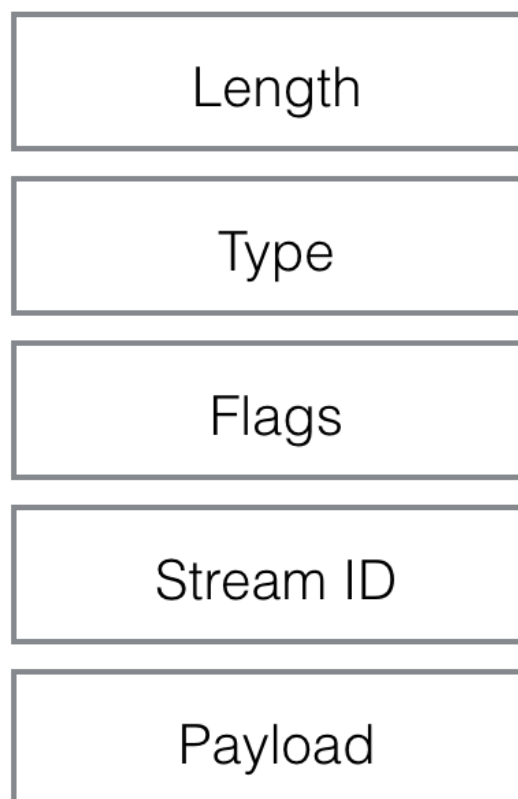
有人可能会觉得基于文本的http调试方便很多，像firebug，chrome，charles等不少工具都可以即时调试修改请求。实际上现在很多请求都是走https了，要调试https请求必须有私钥才行。http2.0的绝大部分request应该都是走https，所以调试方便无法作为一个有力的考虑因素了。curl，tcpdump，wireshark这些工具会更适合http2.0的调试。

http2.0用binary格式定义了一个一个的frame，和http1.x的格式对比如下图：

HTTP1.x

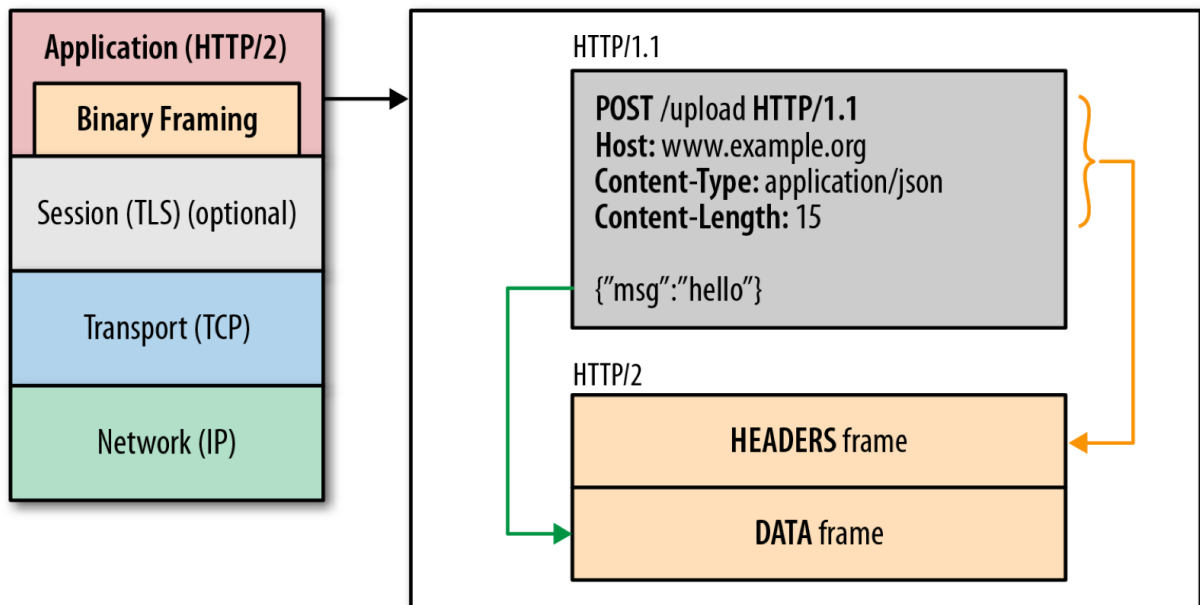


HTTP2.0



http2.0的格式定义更接近tcp层的方式，这种二机制的方式十分高效且精简。length定义了整个frame的开始到结束，type定义frame的类型（一共10种），flags用bit位定义一些重要的参数，stream id用作流控制，剩下的payload就是request的正文了。

虽然看上去协议的格式和http1.x完全不同了，实际上http2.0并没有改变http1.x的语义，只是把原来http1.x的header和body部分用frame重新封装了一层而已。调试的时候浏览器甚至会把http2.0的frame自动还原成http1.x的格式。具体的协议关系可以用下图表示：



连接共享

http2.0要解决的一大难题就是多路复用（MultiPlexing），即连接共享。上面协议解析中提到的stream id就是用作连接共享机制的。一个request对应一个stream并分配一个id，这样一个连接上可以有多个stream，每个stream的frame可以随机的混杂在一起，接收方可以根据stream id将frame再归属到各自不同的request里面。

前面还提到过连接共享之后，需要优先级和请求依赖的机制配合才能解决关键请求被阻塞的问题。http2.0里的每个stream都可以设置又优先级（Priority）和依赖（Dependency）。优先级高的stream会被server优先处理和返回给客户端，stream还可以依赖其它的sub streams。优先级和依赖都是可以动态调整的。动态调整在有些场景下很有用，假想用户在用你的app浏览商品的时候，快速的滑动到了商品列表的底部，但前面的请求先发出，如果不把后面的请求优先级设高，用户当前浏览的图片要到最后才能下载完成，显然体验没有设置优先级好。同理依赖在有些场景下也有妙用。

header压缩

前面提到过http1.x的header由于cookie和user agent很容易膨胀，而且每次都要重复发送。http2.0使用encoder来减少需要传输的header大小，通讯双方各自cache一份header fields表，既避免了重复header的传输，又减小了需要传输的大小。高效的压缩算法可以很大的压缩header，减少发送包的数量从而降低延迟。

这里普及一个小知识点。现在大家都知道tcp有slow start的特性，三次握手之后开始发送tcp segment，第一次能发送的没有被ack的segment数量是由initial tcp window大小决定的。这个initial tcp window根据平台的实现会有差异，但一般是2个segment或者是4k的大小（一个segment大概是1500个字节），也就是说当你发送的包大小超过这个值的时候，要等前面的包被ack之后才能发送后续的包，显然这种情况下延迟更高。intial window也并不是越大越好，太大会导致网络节点的阻塞，丢包率就会增加，具体细节可以参考[IETF这篇文章](#)。http的header现在膨胀到有可能会超过这个intial window的值了，所以更显得压缩header的重要性。

压缩算法的选择

SPDY/2使用的是gzip压缩算法，但后来出现的两种攻击方式[BREACH](#)和[CRIME](#)使得即使走ssl的SPDY也可以被破解内容，最后综合考虑采用的是一种叫[HPACK](#)的压缩算法。这两个漏洞和相关算法可以点击[链接](#)查看更多的细节，不过这种漏洞主要存在于浏览器端，因为需要通过javascript来注入内容并观察payload的变化。

重置连接表现更好

很多app客户端都有取消图片下载的功能场景，对于http1.x来说，是通过设置tcp segment里的reset flag来通知对端关闭连接的。这种方式会直接断开连接，下次再发请求就必须重新建立连接。http2.0引入RST_STREAM类型的frame，可以在不断开连接的前提下取消某个request的stream，表现更好。

Server Push

Server Push的功能前面已经提到过，http2.0能通过push的方式将客户端需要的内容预先推送过去，所以也叫“cache push”。另外有一点值得注意的是，客户端如果退出某个业务场景，出于流量或者其他因素需要取消server push，也可以通过发送RST_STREAM类型的frame来做到。

流量控制（Flow Control）

TCP协议通过sliding window的算法来做流量控制。发送方有个sending window，接收方有receive window。http2.0的flow control是类似receive window的做法，数据的接收方通过告知对方自己的flow window大小表明自己还能接收多少数据。只有Data类型的frame才有flow control的功能。对于flow control，如果接收方在flow window为零的情况下依然更多的frame，则会返回block类型的frame，这张场景一般表明http2.0的部署出了问题。

Nagle Algorithm vs TCP Delayed Ack

tcp协议优化的一个经典场景是：[Nagle](#)算法和Berkeley的[delayed ack算法](#)的对立。http2.0并没有对tcp层做任何修改，所以这种对立导致的高延迟问题依然存在。要么通过TCP_NODELAY禁用Nagle算法，要么通过TCP_QUICKACK禁用delayed ack算法。貌似http2.0官方建议是设置TCP_NODELAY。

更安全的SSL

HTTP2.0使用了tls的拓展ALPN来做协议升级，除此之外加密这块还有一个改动，HTTP2.0对tls的安全性做了近一步加强，通过黑名单机制禁用了几百种不再安全的加密算法，一些加密算法可能还在被继续使用。如果在ssl协商过程当中，客户端和server的cipher suite没有交集，直接就会导致协商失败，从而请求失败。在server端部署http2.0的时候要特别注意这一点。

3.3 HTTP2.0里的负能量

SPDY和HTTP2.0之间的暧昧关系，以及google作为SPDY的创造者，这两点很容易让阴谋论者怀疑google是否会成为协议的最终收益方。这其实是废话，google当然会受益，任何新协议使用者都会从中受益，至于谁吃肉，谁喝汤看的是自己的本事。从整个协议的变迁史也可以粗略看出，新协议的诞生完全是针对业界现存问题对症下药，并没有google业务相关的痕迹存在，google至始至终只扮演了一个角色：you can you up。

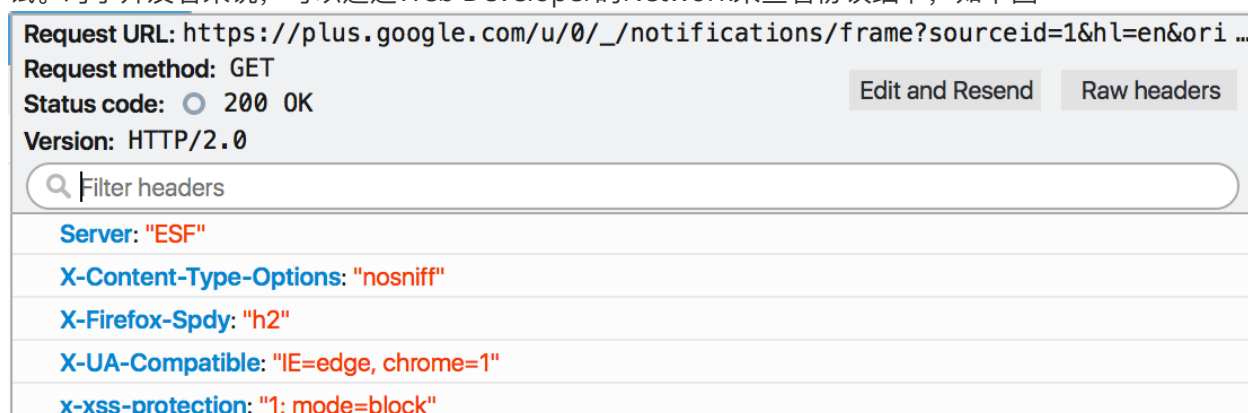
HTTP2.0不会是万金油，但抹了也不会有副作用。HTTP2.0最大的亮点在于多路复用，而多路复用的好处只有在http请求量大的场景下才明显，所以有人会觉得只适用于浏览器浏览大型站点的时候。这么说其实没错，但http2.0的好处不仅仅是multiplexing，请求压缩，优先级控制，server push等等都是亮点。对于内容型移动端app来说，比如淘宝app，http请求量大，多路复用还是能产生明显的体验提升。多路复用对延迟的改变可以参考下这个[测试网址](#)。

HTTP2.0对于ssl的依赖使得有些开发者望而生畏。不少开发者对ssl还停留在高延迟，CPU性能损耗，配置麻烦的印象中。其实ssl于http结合对性能的影响已经可以优化到忽略的程度了，网上也有不少文章可以参考。HTTP2.0也可以不走ssl，有些场景确实可能不适合https，比如对代理服务器的cache依赖，对于内容安全性不敏感的get请求可以通过代理服务器缓存来优化体验。

3.4 HTTP2.0的现状

HTTP2.0作为新版本的网络协议肯定需要一段时间去普及，但HTTP本身属于应用层协议，和当年的网络层协议IPV6不同，离底层协议越远，对网络基础硬件设施的影响就越小。HTTP2.0甚至还特意的考虑了与HTTP1.x的兼容问题，只是在HTTP1.x的下面做了一层framing layer，更使得其普及的阻力变小。所以不出意外，HTTP2.0的普及速度可能会远超大部分人的预期。

Firefox 2015年在其浏览器流量中检测到，有13%的http流量已经使用了http2.0，27%的https也是http2.0，而且还处于持续的增长当中。一般用户察觉不到是否使用了http2.0，不过可以装这样一个插件，安装之后如果网站是http2.0的，在地址栏的最右边会有个闪电图标。还可以使用这个网站来测试。对于开发者来说，可以通过Web Developer的Network来查看协议细节，如下图：



Request URL: https://plus.google.com/u/0/_/notifications/frame?sourceid=1&hl=en&ori ...
Request method: GET
Status code: 200 OK
Version: HTTP/2.0

Filter headers

- Server: "ESF"
- X-Content-Type-Options: "nosniff"
- X-Firefox-Spdy: "h2"
- X-UA-Compatible: "IE=edge, chrome=1"
- x-xss-protection: "1; mode=block"

其中Version: HTTP / 2.0已经很明确表明协议类型，Firefox还在header里面插入了X-Firefox-Spdy:"h2"，也可以看出是否使用http2.0。

Chrome在2015年检测到的http2.0流量大概有18%。不过这个数字本来会更高，因为Chrome现在很大一部分流量都在试验QUIC（google正在开辟的另一块疆土）。Chrome上也可以使用类似的插件来判断网站是否是使用http2.0。

4. 移动端HTTP现状

4.1 iOS下http现状

iOS系统是从iOS8开始才通过NSURLSession来支持SPDY的，iOS9+开始自动支持http2.0。实际上apple对http2.0非常有信心，推广力度也很大。新版本ATS机制默认使用https来进行网络传输。APN（Apple Push Notification）在iOS9上也已经是通过http2.0来实现的了。iOS9 sdk里的NSURLSession默认使用http2.0，而且对开发者来说是完全透明的，甚至没有api来知道到底是用的哪个版本的http协议。

对于开发者来说到底怎么去配置最佳的http使用方案呢？在我看来，因app而异，主要从两方面来考虑：一是app本身http流量是否大而且密集，二是开发团队本身的技术条件。http2.0的部署相对容易很多，客户端开发者甚至不用做什么改动，只需要使用iOS9的SDK编译即可，但缺点是http2.0只能适用于iOS9的设备。SPDY的部署相对麻烦一些，但优点是可以兼顾iOS6+的设备。iOS端的SPDY可以使用twitter开发的CocoaSPDY方案，但有一点需要特别处理：

由于苹果的TLS实现不支持NPN，所以通过NPN协商使用SPDY就无法通过默认443端口来实现。有两种做法，一是客户端和server同时约定好使用另一个端口号来做NPN协商，二是server这边通过request header智能判断客户端是否支持SPDY而越过NPN协商过程。第一种方法会简单一点，不过需要从框架层将所有的http请求都map到另一个port，url mapping可以参考我之前的[一篇文章](#)。twitter自己的网站twitter.com使用的是第二种方法。

浏览器端（比如Chrome），server端（比如Nginx）都陆续打算放弃支持spdy了，毕竟google官方都宣布要停止维护了。spdy会是一个过渡方案，会随着iOS9的普及会逐步消失，所以这部分的技术投入需要开发团队自己去衡量。

4.2 Android下http现状

android和iOS情况类似，http2.0只能在新系统下支持，spdy作为过渡方案仍然有存在的必要。

对于使用webview的app来说，需要基于chrome内核的webview才能支持spdy和http2.0，而android系统的webview是从android4.4（KitKat）才改成基于chrome内核的。

对于使用native api调用的http请求来说，okhttp是同时支持spdy和http2.0的可行方案。如果使用ALPN，okhttp要求android系统5.0+(实际上，android4.4上就有了ALPN的实现，不过有bug，知道5.0才正式修复)，如果使用NPN，可以从android4.0+开始支持，不过NPN也是属于将要被淘汰的协议。

结束语

以上是HTTP从1.x到SPDY，再到HTTP2.0的一些主要变迁技术点。HTTP2.0正处于逐步应用到线上产品和服务的阶段，可以预见未来会有不少新的坑产生和与之对应的优化技巧，HTTP1.x和SPDY也将在一段时间内继续发挥余热。作为工程师，需要了解这些协议背后的技术细节，才能打造高性能的网络框架，从而提升我们的产品体验。

参考链接

<http://http2-explained.haxx.se/content/en/part5.html>

<https://www.chromium.org/spdy/spdy-whitepaper>

Auto Layout: Snapkit源码剖析

Snapkit是目前Swift中通过代码进行Auto Layout布局时最流行的开源库。与OC中最主流的Auto Layout开源库Masonry是同一个团队维护，有着相似的API风格。

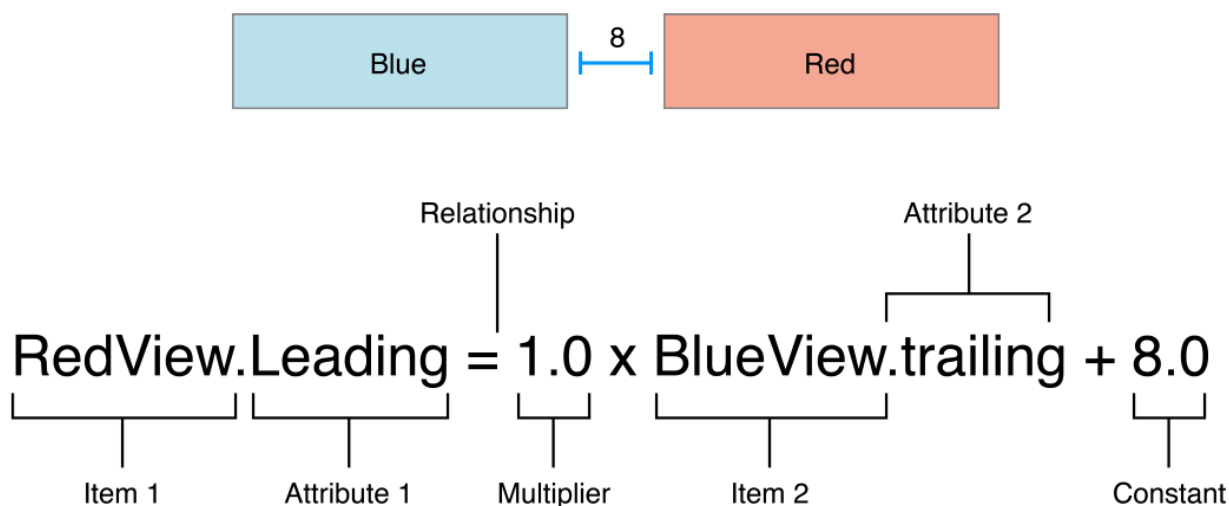
Auto Layout的本质

Auto Layout可以理解为描述一个界面中各个元素之间布局的关系的一种语言。每个关系描述对应的是一条约束。

平面关系里两个点的关系可以用一次函数（ $y = ax + b$ ）来表示，与此相似，每条约束的定义方式与一次函数也有着一样的参数：

```
item1.attribute1 = multiplier × item2.attribute2 + constant
```

再贴一张官方的示意图：



这条约束表示RedView的左边与BlueView右边的距离为8。

原生API的问题

Auto Layout从编码层面看本质就是约束（NSLayoutConstraint），通过给一个元素声明约束来表示它的布局关系。

Visual Format Language

Visual Format Language是创建约束的一种方式。通过一句符合指定语法的字符串来生成约束。

写起来的风格就像这样：

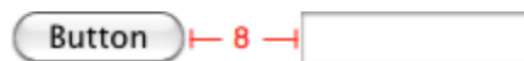
```
let viewsDictionary = ["label1": label1, "label2": label2, "label3": label3,
"label4": label4]
let metrics = ["labelHeight": 88]
let constraint = NSLayoutConstraint.constraints(withVisualFormat:
"V:[label1(labelHeight)]-[label2(labelHeight)]-[label3(labelHeight)]-
[label4(labelHeight)]-(>=10)-|",
options: [], metrics: metrics, views: viewsDictionary)
view.addConstraints(constraint)
```

有着自己的一套用于表示相关元素关系的语法。

下面截图展示了一些VFL的语法：

Standard Space

`[button]-[textField]`



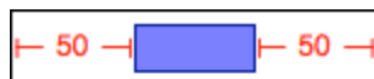
Width Constraint

`[button(>=50)]`



Connection to Superview

`| -50 - [purpleBox] -50 - |`



优点

- Xcode控制台输出约束时使用的就是这种语法。如果对这种语法很熟悉，调试约束时会有很大帮助。
- 一次可以表示几个物体在一个方向上的约束关系，所以可以一次创建几个约束。
- 在这种语法限制下只能创建出有效的约束。

缺点

- VFL的语法适合表述平面关系的约束，对于一些约束，比如比例（aspect ratios）相关的约束无法通过VFL创建。
- 编译器不会检查VFL的字符串，所以只能在运行时调试约束。

实际使用：不推荐

VFL通过一个约定格式的字符串表示约束关系。在实际项目中，一个无法被检查的长字符串很容易发生拼写错误。VFL的语法也有一定的学习门槛，与我们自然语言中描述约束关系的方式相差很远。约束通过一个字符串表示也导致了不好复用的问题。所以在实际项目中通常不会选择VFL。

iOS 7 & 8: NSLayoutConstraint

假设我们要对blueView布局为长宽都为100，在iOS 7中这样写：

```
blueView.translatesAutoresizingMaskIntoConstraints = false

// iOS 7
let widthConstraint = NSLayoutConstraint(item: blueView, attribute: .width,
relatedBy: .equal, toItem: nil, attribute: .notAnAttribute, multiplier: 1,
constant: 100)
blueView.addConstraint(widthConstraint)
let heightConstraint = NSLayoutConstraint(item: blueView, attribute:
.height, relatedBy: .equal, toItem: nil, attribute: .notAnAttribute,
multiplier: 1, constant: 100)
blueView.addConstraint(heightConstraint)
```

接着再设置水平竖直居中，在iOS 8中这样写：

```
// iOS 8
let centerXConstraint = NSLayoutConstraint(item: blueView, attribute:
.centerX, relatedBy: .equal, toItem: view, attribute: .centerX, multiplier:
1, constant: 0)
centerXConstraint.isActive = true

NSLayoutConstraint(item: blueView, attribute: .centerY, relatedBy: .equal,
toItem: view, attribute: .centerY, multiplier: 1, constant: 0).isActive =
true
```

创建 `NSLayoutConstraint` 对象的方法参数是一致的，区别只是在iOS 7中需要指出这条约束应该加在哪个对象上。在iOS 8中省去了这个判断，只需要设置这条约束的 `isActive` 为true时就可以。可以这样做是因为一条约束自身已经知道这条约束对应的是哪两个对象，再用代码表明这条约束应该加在哪个对象身上是不必须的。

评价

这样的API虽然完整的表达了约束所需要的参数，但是写法却非常繁琐。每条约束都需要7个参数，即使有的参数是不必须的。每次只能添加一条约束。通过上面的代码也可以直观的看出，只是设置一个view的简单约束代码已经一堆了，API的表现力很差。

iOS 9 : Layout Anchors

Auto Layout发布两年后，苹果推出了一套新的API来创建约束。我很怀疑苹果设计这组API参考了Snapkit，因为风格上有些接近。

和上面一节实现同样的布局，在iOS 9后代码这样写：

```
blueView.translatesAutoresizingMaskIntoConstraints = false
blueView.widthAnchor.constraint(equalToConstant: 100).isActive = true
blueView.heightAnchor.constraint(equalToConstant: 100).isActive = true
blueView.centerXAnchor.constraint(equalTo: view.centerXAnchor).isActive = true
blueView.centerYAnchor.constraint(equalTo: view.centerYAnchor).isActive = true
```

constraint函数利用了swift函数默认参数的语言特性，将multiplier和constant配置了默认值。完整的函数签名如下：

```
func constraint(equalTo anchor: NSLayoutDimension, multiplier m: CGFloat,
constant c: CGFloat) -> NSLayoutConstraint
```

总体而言相比之前的写法有了一个巨大的进步。约束的添加逻辑更加接近我们思维的方式，友好很多。

Snapkit

没有对比就没有伤害。同样的界面如果用Snapkit写起来是这样的：

```
blueView.snp.makeConstraints { (make) in
    make.width.height.equalTo(100)
    make.center.equalToSuperview()
}
```

总共只要3行代码，简洁清晰。

源码剖析

Attributes

先介绍一下最基础的attribute。

NSLayoutAttribute

系统API提供的属性是 `NSLayoutAttribute`，是一个枚举：

```
enum NSLayoutAttribute : Int {
    case left
    case right
    case top
    //...
    case notAnAttribute
}
```

ConstraintAttributes

`NSLayoutAttribute` 的不足之处在于一次只能表达一个属性。

然而有时我们几个属性都是对应一个表达式的值，比如下面的代码。

```
make.width.height.equalTo(100)
```

一个值可以对应多个选项，在Swift中就是 `OptionSet` 协议了。

这里稍微展示一下 `OptionSet` 的用法：

```
struct ShippingOptions: OptionSet {
    let rawValue: Int

    static let nextDay    = ShippingOptions(rawValue: 1 << 0)
    static let secondDay  = ShippingOptions(rawValue: 1 << 1)
    static let priority    = ShippingOptions(rawValue: 1 << 2)
    static let standard    = ShippingOptions(rawValue: 1 << 3)

    static let express: ShippingOptions = [.nextDay, .secondDay]
    static let all: ShippingOptions = [.express, .priority, .standard]
}
```

上面的代码定义的 `all` 的就是一个选项的集合。我们在动画中经常用到的

`UIViewAnimationOptions` 也是这样的类型。

`ConstraintAttributes` 是Snapkit中用于表示attribute的类型。定义了与 `NSLayoutAttribute` 一致的属性：

```
struct ConstraintAttributes : OptionSet {
    static var none: ConstraintAttributes { return self.init(0) }
    static var left: ConstraintAttributes { return self.init(1) }
    static var top: ConstraintAttributes { return self.init(2) }
    static var right: ConstraintAttributes { return self.init(4) }
    //...
}
```

细节：重载 + 运算符

苹果在 `OptionSet` 中定义了几个集合运算的函数，为了更便捷的使用，Snapkit自定义了几个运算符：

```
func + (left: ConstraintAttributes, right: ConstraintAttributes) ->
ConstraintAttributes {
    return left.union(right)
}

func +=(left: inout ConstraintAttributes, right: ConstraintAttributes) {
    left.formUnion(right)
}

func -=(left: inout ConstraintAttributes, right: ConstraintAttributes) {
    left.subtract(right)
}

func ==(left: ConstraintAttributes, right: ConstraintAttributes) -> Bool {
    return left.rawValue == right.rawValue
}
```

如何转换为NSLayoutAttribute

因为最后调用的还是系统的API，所以 `ConstraintAttributes` 需要转换为对应的 `NSLayoutAttribute` 类型。这段逻辑放在计算属性 `layoutAttributes` 中：

```
var layoutAttributes: [NSLayoutAttribute] {
    var attrs = [NSLayoutAttribute]()
    if (self.contains(ConstraintAttributes.left)) {
        attrs.append(.left)
    }
    if (self.contains(ConstraintAttributes.top)) {
        attrs.append(.top)
    }
    if (self.contains(ConstraintAttributes.right)) {
        attrs.append(.right)
    }
    //...
    return attrs
}
```

每次获取 `layoutAttributes` 属性时，就会根据自身的值生成一个对应的 `NSLayoutAttribute` 数组返回。

简单的基础Model

Snapkit不仅支持iOS平台，还支持tvOS、macOS其他平台。为了更方便的支持多平台，Snapkit把几个常用对象都重新封装了一次。

ConstraintView

`ConstraintView` 是一个别名，对应的就是iOS上的 `UIView`：

```
#if os(iOS) || os(tvOS)
    public typealias ConstraintView = UIView
#else
    public typealias ConstraintView = NSView
#endif
```

其他几个别名

同上节，Snapkit还定义其他几个别名，对应关系如下表：

Snapkit	iOS
ConstraintLayoutGuide	UILayoutGuide
ConstraintLayoutSupport	UILayoutSupport
ConstraintInsets	UIEdgeInsets
ConstraintInterfaceLayoutDirection	UIUserInterfaceLayoutDirection

ConstraintRelation

猜测由于历史原因 `ConstraintRelation` 没有采用别名的方式，而是重新封装了一遍。

```
enum ConstraintRelation : Int {
    case equal = 1
    case lessThanOrEqualTo
    case greaterThanOrEqualTo

    var layoutRelation: NSLayoutRelation {
        get {
            switch(self) {
                case .equal:
                    return .equal
                case .lessThanOrEqualTo:
                    return .lessThanOrEqualTo
                case .greaterThanOrEqualTo:
                    return .greaterThanOrEqualTo
            }
        }
    }
}
```

通过 `layoutRelation` 获取到对应的 `NSLayoutRelation` 值。

ConstraintItem

一条约束对应两个Item。

```
item1.attribute1 = multiplier × item2.attribute2 + constant
```

一个Item包含两个属性：target和attributes，表示这个Item的实例和属性：

```
public final class ConstraintItem {  
    weak var target: AnyObject?  
    let attributes: ConstraintAttributes  
    //...  
}
```

同时注意到这个类被标记为了final，这样可以提高编译的性能，也表明了这个类不能被继承。

重载 == 运算符

```
public func ==(lhs: ConstraintItem, rhs: ConstraintItem) -> Bool {  
    // pointer equality  
    guard lhs !== rhs else {  
        return true  
    }  
  
    // must both have valid targets and identical attributes  
    guard let target1 = lhs.target,  
          let target2 = rhs.target,  
          target1 === target2 && lhs.attributes == rhs.attributes else {  
        return false  
    }  
  
    return true  
}
```

`ConstraintItem` 还重载了 `==` 运算符。首先判断两个实例是不是指向同一块内存地址，注意这里是判断的运算符是 `!==`。接着再判断两个Item的attributes是否一样。

layoutConstraintItem

target的类型是 `AnyObject`，在进行逻辑判断时很不方便。专门暴露出了一个已经类型转换好的属性：

```
var layoutConstraintItem: LayoutConstraintItem? {  
    return self.target as? LayoutConstraintItem  
}
```

LayoutConstraintItem

`LayoutConstraintItem` 是一个协议，用于表示可以添加约束的对象。在iOS 9之前，只有 `UIView` 能添加约束，在iOS 9中，引入了可以参与辅助布局的 `UILayoutGuide`，可以和 `UIView` 一样添加约束。

```
public protocol LayoutConstraintItem: class {
}

@available(iOS 9.0, OSX 10.11, *)
extension ConstraintLayoutGuide : LayoutConstraintItem {
}

extension ConstraintView : LayoutConstraintItem {
}
```

那么这个协议定义那些函数和属性呢？

prepare

定义了一个 `prepare` 函数，如果是 `UIView` 对象，则把 `translatesAutoresizingMaskIntoConstraints` 设置为 `false`：

```
extension LayoutConstraintItem {

    internal func prepare() {
        if let view = self as? ConstraintView {
            view.translatesAutoresizingMaskIntoConstraints = false
        }
    }

    //...
}
```

superview

比如居中的约束常常是对于 `superview` 而言，在协议里扩展了一个 `superview` 属性：

```
var superview: ConstraintView? {
    if let view = self as? ConstraintView {
        return view.superview
    }

    if #available(iOS 9.0, OSX 10.11, *), let guide = self as?
    ConstraintLayoutGuide {
        return guide.owningView
    }

    return nil
}
```

constraints

每个UIView身上都可能有几条约束，所以定义了几个添加移除约束的函数，和一个用于保存约束的集合：

```
var constraints: [Constraint] {
    return self.constraintsSet.allObjects as! [Constraint]
}

func add(constraints: [Constraint]) {
    //...
}

func remove(constraints: [Constraint]) {
    //...
}

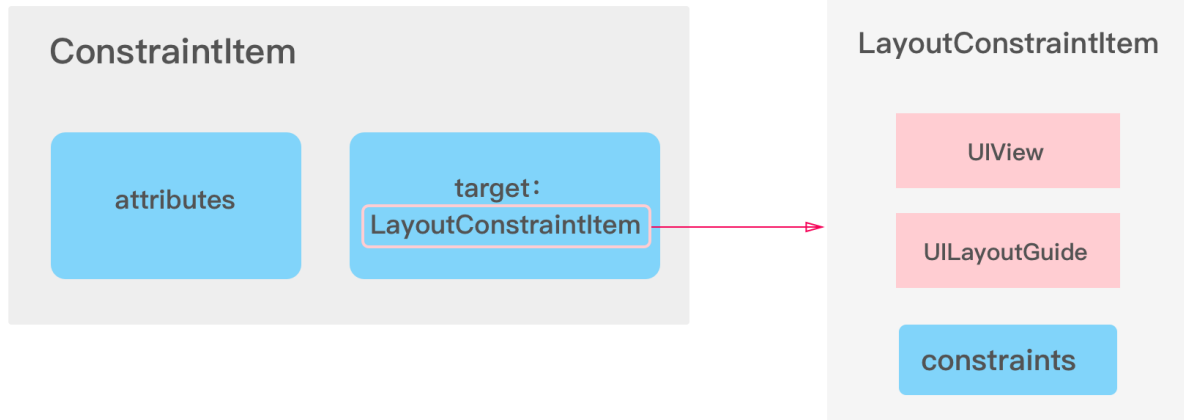
private var constraintsSet: NSMutableSet {
    let constraintsSet: NSMutableSet

    if let existing = objc_getAssociatedObject(self, &constraintsKey)
as? NSMutableSet {
        constraintsSet = existing
    } else {
        constraintsSet = NSMutableSet()
        objc_setAssociatedObject(self, &constraintsKey,
            constraintsSet, .OBJC_ASSOCIATION_RETAIN_NONATOMIC)
    }
    return constraintsSet
}
```

为了保证相同的约束不会被重复添加，内部用了 `NSMutableSet` 来保存。为什么不是swift中的 `Set` 类型呢？因为这个集合定义在扩展里，通过OC中的runtime方法 `objc_getAssociatedObject` 保存，所以需要是OC的对象，于是选择了 `NSMutableSet`。最后封装成了一个数组 `constraints` 暴露给外界。

图示

用一张图简单的表示就是这样：



ConstraintItem包含一组属性和一个对应的可布局实例（**LayoutConstraintItem**），这个实例可能是 `UIView` 也可能是 `UILayoutGuide`，这个可布局实例身上的 `constraints` 表示自身已经持有的一组约束。

LayoutConstraint

`LayoutConstraint` 继承自 `NSLayoutConstraint`。为了保持自身命名风格的统一，将 `identifier` 属性封装成了 `label`，换了一个属性名。

```
public class LayoutConstraint : NSLayoutConstraint {

    public var label: String? {
        get {
            return self.identifier
        }
        set {
            self.identifier = newValue
        }
    }

    weak var constraint: Constraint? = nil

}
```

Snapkit中一条约束可能对应几个 `NSLayoutConstraint`，所以这里有一个weak的 `constraint` 属性指向生成这条约束的Snapkit约束实例。

*Target

看这样的代码：

```
make.width.height.equalTo(100)
make.center.equalTo(self.view)
```

注意到 `equalTo` 的参数可以是一个值，也可以是一个 `UIView` 类型。

在iOS 9的API则是笨拙一点的两个函数：

```
open func constraint(equalToConstant c: CGFloat) -> NSLayoutConstraint

open func constraint(equalTo anchor: NSLayoutAnchor<AnchorType>) ->
NSLayoutConstraint
```

Snapkit通过定义了一个*Target的协议来达到这个目的。这里列举一个简单的例子 `ConstraintPriorityTarget`，这个协议就是指可以作为设置Priority的类型。

```
public protocol ConstraintPriorityTarget {

    var constraintPriorityTargetValue: Float { get }

}

extension Int: ConstraintPriorityTarget {

    public var constraintPriorityTargetValue: Float {
        return Float(self)
    }

}

// ....
extension Double: ConstraintPriorityTarget {

    public var constraintPriorityTargetValue: Float {
        return Float(self)
    }

}

extension CGFloat: ConstraintPriorityTarget {

    public var constraintPriorityTargetValue: Float {
        return Float(self)
    }

}

}
```

可以看到并没有什么太好的办法，在协议中定义了需要的类型*TargetValue，然后在每个实现了这个协议的类型中转换类型。这种方式虽然增加了一点代码的复杂度，但是对于使用API的用户而则省去了类型转换的步骤。

这样的*Target有以下几个：

协议	获取Value的属性、函数
ConstraintRelatableTarget	没有
ConstraintConstantTarget	constraintConstantTargetValueFor(layoutAttribute: NSLayoutAttribute)
ConstraintPriorityTarget	constraintPriorityTargetValue
ConstraintMultiplierTarget	constraintMultiplierTargetValue
ConstraintOffsetTarget	constraintOffsetTargetValue
ConstraintInsetTarget	constraintInsetTargetValue

RelatableTarget因为可以有好几种不同类型，因此无法在协议里声明一个通用的属性，在使用时进行类型转换判断：

```
public class ConstraintMakerRelatable {

    internal func relatedTo(_ other: ConstraintRelatableTarget, relation:
ConstraintRelation, file: String, line: UInt) -> ConstraintMakerEditable {
        let related: ConstraintItem
        let constant: ConstraintConstantTarget

        if let other = other as? ConstraintItem {
            //...
            related = other
            constant = 0.0
        } else if let other = other as? ConstraintView {
            related = ConstraintItem(target: other, attributes:
ConstraintAttributes.none)
            constant = 0.0
        } else if let other = other as? ConstraintConstantTarget {
            related = ConstraintItem(target: nil, attributes:
ConstraintAttributes.none)
            constant = other
        } else if #available(iOS 9.0, OSX 10.11, *), let other = other as?
ConstraintLayoutGuide {
            related = ConstraintItem(target: other, attributes:
ConstraintAttributes.none)
            constant = 0.0
        } else {
            fatalError("Invalid constraint. \(file), \(line)")
        }

        //...
    }
}
```

Constraint

`Constraint` 是Snapkit中表示约束的对象。

基础属性

最后 `Constraint` 还是要生成 `LayoutConstraint`，所以基础属性都一致：

```
public final class Constraint {  
  
    internal let sourceLocation: (String, UInt)  
    internal let label: String?  
  
    private let from: ConstraintItem  
    private let to: ConstraintItem  
    private let relation: ConstraintRelation  
    private let multiplier: ConstraintMultiplierTarget  
    private var constant: ConstraintConstantTarget  
    private var priority: ConstraintPriorityTarget  
  
    //...  
}
```

增加了一个 `sourceLocation` 属性，用了记录生成这条约束的源代码位置，是一个 `Tuple`，第一个值表示文件名，第二个值表示代码行数。这个属性可以在调试的时候输出相关信息。

layoutConstraints

在初始化时生成对应的 `LayoutConstraint` 数组。因为在Snapkit中的一个约束的属性是一个组合，所以可以对应几条原生约束。下面的代码在对应属性赋值完成后，遍历 `layoutFromAttributes` 生成对应的 `LayoutConstraint`，然后添加到 `layoutConstraints` 中。

```

public var layoutConstraints: [LayoutConstraint]

// MARK: Initialization

internal init(from: ConstraintItem,
              to: ConstraintItem,
              relation: ConstraintRelation,
              sourceLocation: (String, UInt),
              label: String?,
              multiplier: ConstraintMultiplierTarget,
              constant: ConstraintConstantTarget,
              priority: ConstraintPriorityTarget) {

    self.from = from
    self.to = to
    self.relation = relation
    self.sourceLocation = sourceLocation
    self.label = label
    self.multiplier = multiplier
    self.constant = constant
    self.priority = priority
    self.layoutConstraints = []

    //...
    let layoutFromAttributes = self.from.attributes.layoutAttributes
    for layoutFromAttribute in layoutFromAttributes {
        //...
        // 生成对应的LayoutConstraint
        let layoutConstraint = LayoutConstraint(
            item: layoutFrom,
            attribute: layoutFromAttribute,
            relatedBy: layoutRelation,
            toItem: layoutTo,
            attribute: layoutToAttribute,
            multiplier: self.multiplier.constraintMultiplierTargetValue,
            constant: layoutConstant
        )
        layoutConstraint.constraint = self
        self.layoutConstraints.append(layoutConstraint)
    }
}

```

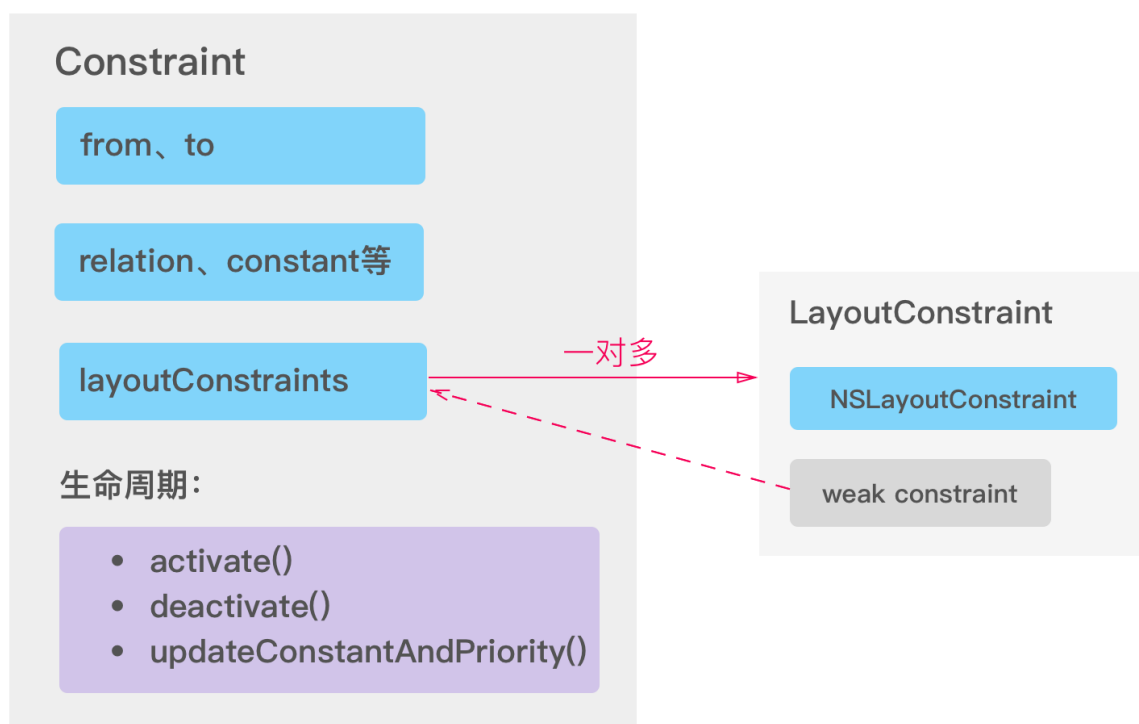
管理约束生命周期

`LayoutConstraint` 通过 `isActive` 让这条约束生效。在 `Constraint` 也有类似的管理生命周期的函数：

```
public final class Constraint {
    //...
    func activateIfNeeded(updatingExisting: Bool = false) {
        //...
    }

    func deactivateIfNeeded() {
        //...
    }
}
```

图示



ConstraintDescription

那么约束的一些默认值（multiplier默认为1，constan默认为0）是在什么地方设置的呢？答案就是 `ConstraintDescription`。

同时注意一下 `Constraint` 的基本属性的修饰符是 `let`。Snapkit希望 `Constraint` 只有一个职责，就是生成原生约束，管理这些约束的生命周期。而对于这个约束的参数不参与采集。这就是 `ConstraintDescription` 的意义：填充完整一个约束所需要的参数。

```

public class ConstraintDescription {

    internal let item: LayoutConstraintItem
    internal var attributes: ConstraintAttributes
    internal var relation: ConstraintRelation? = nil
    internal var sourceLocation: (String, UInt)? = nil
    internal var label: String? = nil
    internal var related: ConstraintItem? = nil
    internal var multiplier: ConstraintMultiplierTarget = 1.0
    internal var constant: ConstraintConstantTarget = 0.0
    internal var priority: ConstraintPriorityTarget = 1000.0

    internal lazy var constraint: Constraint? = {
        guard let relation = self.relation,
              let related = self.related,
              let sourceLocation = self.sourceLocation else {
            return nil
        }
        let from = ConstraintItem(target: self.item, attributes:
self.attributes)

        return Constraint(
            from: from,
            to: related,
            relation: relation,
            sourceLocation: sourceLocation,
            label: self.label,
            multiplier: self.multiplier,
            constant: self.constant,
            priority: self.priority
        )
    }()

    // MARK: Initialization

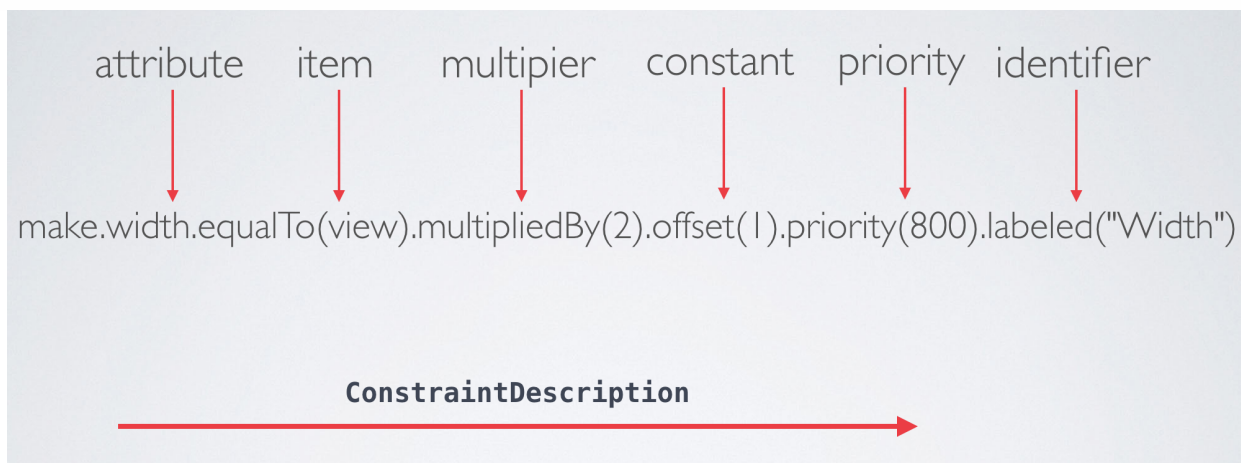
    internal init(item: LayoutConstraintItem, attributes:
ConstraintAttributes) {
        self.item = item
        self.attributes = attributes
    }

}

```

`ConstraintDescription` 定义了和 `Constraint` 一样的基本参数属性，只是属性是可变的 `var`。通过 `constraint` 属性生成 `Constraint` 类型的实例。也很清楚的看到在这里设置了 `multiplier`、`constant`、`priority` 的默认值。

什么时候填充的这些值呢？



使用make后的每一步里设置的值最后都会设置到一个 `ConstraintDescription` 实例中。最后从这个实例的 `constraint` 属性生成一个真正的 `Constraint`。

ConstraintViewDSL

现在从使用Snapkit的代码一步步深入来看。

```
blueView.snp.makeConstraints { (make) in
    make.width.height.equalTo(100)
}
```

snp

```
public extension ConstraintView {
    //...
    public var.snp: ConstraintViewDSL {
        return ConstraintViewDSL(view: self)
    }
}
```

snp是UIView的一个扩展属性，每次返回一个新的 `ConstraintViewDSL` 实例，这个实例的初始化方法会保存这个view。

contentHuggingPriority、contentCompressionResistancePriority

在DSL中封装了设置HuggingPriority和CompressionResistancePriority的属性。在set的时候调用UIView的方法。

```

public var target: AnyObject? {
    return self.view
}

internal let view: ConstraintView

internal init(view: ConstraintView) {
    self.view = view
}

public var contentHuggingHorizontalPriority: Float {
    get {
        return self.view.contentHuggingPriority(for: .horizontal)
    }
    set {
        self.view.setContentHuggingPriority(newValue, for: .horizontal)
    }
}

public var contentHuggingVerticalPriority: Float {
    //...
}

public var contentCompressionResistanceHorizontalPriority: Float{
    //...
}

public var contentCompressionResistanceVerticalPriority: Float {
    //...
}

```

ConstraintAttributesDSL

Snapkit中设置约束的时候有时还会这么写：

```
make.left.equalTo(self.view.snp.right)
```

这里的 `snp.right` 属性就是定义在 `ConstraintAttributesDSL` 里。 `ConstraintViewDSL` 实现了这个 `ConstraintAttributesDSL` 协议。

```

public struct ConstraintViewDSL: ConstraintAttributesDSL {

    public var target: AnyObject? {
        return self.view
    }
}

extension ConstraintBasicAttributesDSL {

    public var left: ConstraintItem {
        return ConstraintItem(target: self.target, attributes:
ConstraintAttributes.left)
    }

    public var top: ConstraintItem {
        return ConstraintItem(target: self.target, attributes:
ConstraintAttributes.top)
    }

    //...
}

```

每次都创建一个 `ConstraintItem` 对象。这个对象前面已经介绍过，包含target和ConstraintAttributes，表示一个item。

makeConstraints

这里当然还少不了 `makeConstraints` 函数。同时还定义了updateConstraints、remakeConstraints、removeConstraints等处理约束的方法。

很容易看出这里面操作的核心就是 `ConstraintMaker`。

```

    public func prepareConstraints(_ closure: (_ make: ConstraintMaker) ->
Void) -> [Constraint] {
        return ConstraintMaker.prepareConstraints(item: self.view, closure:
closure)
    }

    public func makeConstraints(_ closure: (_ make: ConstraintMaker) ->
Void) {
        ConstraintMaker.makeConstraints(item: self.view, closure: closure)
    }

    public func remakeConstraints(_ closure: (_ make: ConstraintMaker) ->
Void) {
        ConstraintMaker.remakeConstraints(item: self.view, closure: closure)
    }

    public func updateConstraints(_ closure: (_ make: ConstraintMaker) ->
Void) {
        ConstraintMaker.updateConstraints(item: self.view, closure: closure)
    }

    public func removeConstraints() {
        ConstraintMaker.removeConstraints(item: self.view)
    }

```

ConstraintMaker

先来看Maker使用的情况：

```
make.width.equalTo(100)
```

make的作用是通过各种函数把 `ConstraintDescription` 的值填充完整。

```

public class ConstraintMaker {

    public var left: ConstraintMakerExtendable {
        return self.makeExtendableWithAttributes(.left)
    }

    private let item: LayoutConstraintItem
    private var descriptions = [ConstraintDescription]()

    internal init(item: LayoutConstraintItem) {
        self.item = item
        self.item.prepare()
    }

    internal func makeExtendableWithAttributes(_ attributes:
ConstraintAttributes) -> ConstraintMakerExtendable {
        let description = ConstraintDescription(item: self.item, attributes:
attributes)
        self.descriptions.append(description)
        return ConstraintMakerExtendable(description)
    }

    //...
}

```

maker初始化时接收一个item作为参数，在这里调用了item的 `prepare()` 函数，所以每个UIView如果要被设置约束都会在这里设置 `translatesAutoresizingMaskIntoConstraints`：

```

func prepare() {
    if let view = self as? ConstraintView {
        view.translatesAutoresizingMaskIntoConstraints = false
    }
}

```

接着创建了一个空的 `ConstraintDescription` 数组。

每次设置maker的约束属性比如 `make.left`、`make.center` 就会创建一个对应的 `ConstraintDescription` 添加到 `descriptions` 这数组中。

ConstraintMakerExtendable

`ConstraintMakerExtendable` 继承自 `ConstraintMakerRelatable`。初始化方法定义在 `ConstraintMakerRelatable` 中：

```
public class ConstraintMakerRelatable {

    internal let description: ConstraintDescription

    internal init(_ description: ConstraintDescription) {
        self.description = description
    }
    //...
}
```

与此类似 `ConstraintMakerEditable` 继

承 `ConstraintMakerPriortizable`, `ConstraintMakerPriortizable` 继

承 `ConstraintMakerFinalizable`, `ConstraintMakerFinalizable` 也定义了一样的初始化方法:

```
public class ConstraintMakerFinalizable {

    internal let description: ConstraintDescription

    internal init(_ description: ConstraintDescription) {
        self.description = description
    }

    @discardableResult
    public func labeled(_ label: String) -> ConstraintMakerFinalizable {
        self.description.label = label
        return self
    }

    public var constraint: Constraint {
        return self.description.constraint!
    }

}
```

make.left

仔细看 `left` 的源码, 会发现返回的是 `self`:

```
public var left: ConstraintMakerExtendable {
    self.description.attributes += .left
    return self
}

public var top: ConstraintMakerExtendable {
    self.description.attributes += .top
    return self
}
```

make在第一层设置的都是attributes，设置的attributes由description保存。

ConstraintMakerRelatable

在确定了属性后就要确定item2，第二层ConstraintMakerRelatable定义了 `equalTo` 函数：

```

@discardableResult
public func equalTo(_ other: ConstraintRelatableTarget, _ file: String =
#file, _ line: UInt = #line) -> ConstraintMakerEditable {
    return self.relatedTo(other, relation: .equal, file: file, line:
line)
}

@discardableResult
public func lessThanOrEqualTo(_ other: ConstraintRelatableTarget, _
file: String = #file, _ line: UInt = #line) -> ConstraintMakerEditable {
    return self.relatedTo(other, relation: .lessThanOrEqualTo, file: file,
line: line)
}

internal func relatedTo(_ other: ConstraintRelatableTarget, relation:
ConstraintRelation, file: String, line: UInt) -> ConstraintMakerEditable {
    let related: ConstraintItem
    let constant: ConstraintConstantTarget
    // 对related (toItem) 、constant进行赋值
    if let other = other as? ConstraintItem {
        //...
        related = other
        constant = 0.0
    } else if let other = other as? ConstraintView {
        related = ConstraintItem(target: other, attributes:
ConstraintAttributes.none)
        constant = 0.0
    } else if let other = other as? ConstraintConstantTarget {
        related = ConstraintItem(target: nil, attributes:
ConstraintAttributes.none)
        constant = other
    } else if #available(iOS 9.0, OSX 10.11, *), let other = other as?
ConstraintLayoutGuide {
        related = ConstraintItem(target: other, attributes:
ConstraintAttributes.none)
        constant = 0.0
    } else {
        fatalError("Invalid constraint. \(file), \(line)")
    }
    // 生成 ConstraintMakerEditable 返回, 进入下一个填充值的流程中
    let editable = ConstraintMakerEditable(self.description)
    editable.description.sourceLocation = (file, line)
    editable.description.relation = relation
    editable.description.related = related
    editable.description.constant = constant
    return editable
}
//...

```

通过源码可以看出，调用 `equalTo` 的时候记录了源码的位置，填充了related (toltem)、constant。在这里已经确认了from、to、relation、constant。把description继承送给ConstraintMakerEditable。

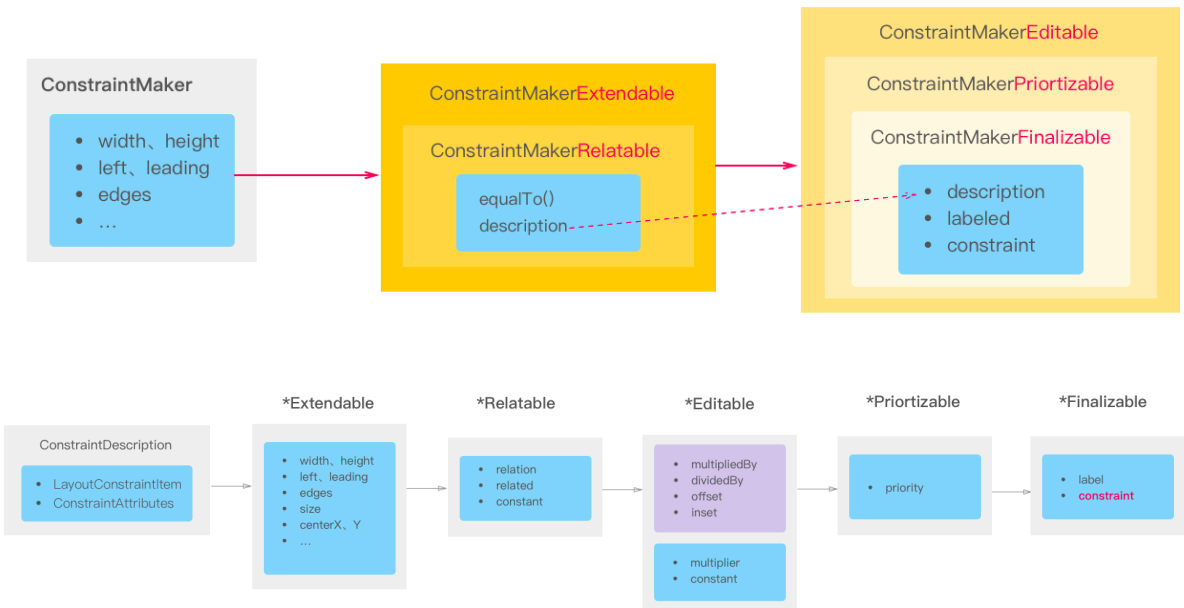
MakerEditable、MakerPriortizable、MakerFinalizable

接着的步骤就和上面的一样，每步的属性设置完成以后，设置父类的对应属性。

ConstraintMakerEditable	ConstraintMakerPriortizable	ConstraintMakerFinalizable
multipliedBy	priority	labeled
dividedBy		
offset		
insets		

最后从 `ConstraintMakerFinalizable` 保存的description里取出约束。

图示



makeConstraints

真正创建约束的地方在Maker里：

```
internal static func makeConstraints(item: LayoutConstraintItem,
closure: (_ make: ConstraintMaker) -> Void) {
    let maker = ConstraintMaker(item: item)
    closure(maker)
    var constraints: [Constraint] = []
    for description in maker.descriptions {
        guard let constraint = description.constraint else {
            continue
        }
        constraints.append(constraint)
    }
    for constraint in constraints {
        constraint.activateIfNeeded(updatingExisting: false)
    }
}
```

流程到这里已经非常清晰，显示传入当前的item，创建一个Maker。接着填充完这个maker里的 `ConstraintDescription` 数组，接着从这个数组中生成对应的约束，最后激活这些约束。

代码总结

Snapkit完整灵活的提供了Auto Layout的能力。对外的API也做的非常简洁。使用了协议扩展和继承来复用代码，很好的控制了代码的复杂度，最大的一个类代码行数没有超过300行。

当我们读源码的时候在读什么？

我想除了了解它的实现原理外，还应该注意它的结构设计。当一个开源库足够流行，可以说明它的解决方案得到了开发者的认可，在实现目的和代码实现上达到了一个良好的平衡。我们也应该思考学习它解决问题的模式和架构，尝试理解它做出的选择与取舍。

参考链接

[Auto Layout Guide](#)

[Visual Format Language](#)

[Snapkit](#)

[DEMO](#)

在服务端写Swift是一种什么样的体验

不得不说，在生产中使用Swift开发服务端是一次大胆的尝试。是其发展速度和活跃程度给了我足够的信心，而其开发速度和运行效率又给了我足够的动力。在实际项目中以 MySQL 和 HDFS 作为支撑，应用了包含 Web 服务、JSON 接口服务、以及基于 Socket 的文件传输和视频直播流服务在内的业务。这些服务最终都运行在 Linux 服务器上，实践的结果让人满意。因此笔者非常期待 Swift 能在服务端市场中能大显身手。

溯源

Swift在Linux上的表现

北京时间 2015 年 6 月 9 日凌晨的 WWDC 大会上，发布 Swift 2.0 的同时宣布 Swift 即将开源，开源内容包括编译器 and 标准库，并支持 Linux。开源和跨平台给语言带来了更宽的发展通道。

那么在 Linux 中 Swift 表现到底如何呢？

其实早期的在 iOS 上使用 Swift 的开发者可能会发现，iOS 和 macOS 中的 Swift 并没有脱离 Objective-C。很多 SDK 中的 Swift 类其实仅仅是 Objective-C 类的封装。例如在开发应用的时候不可避免的要使用 `UIViewController` 或者 `NSViewController` 这些标准框架类，这些类存在于 UIKit 等标准库中。虽然 App 主体使用了 Swift 开发，但这些标准库仍然是旧的 Objective-C 库，造成的结果就是这些 Swift 代码仍然跑在 Objective-C runtime 中。这常常让人觉得所谓纯 Swift 项目一点都不纯，写 Swift 简直多此一举，还不如直接操作 Objective-C 呢。

但是在 Linux 中，纯 Swift 可以真的是纯 **Swift**。Apple 这些年在软件产品上一直背负着沉重的历史负担，大量的库都使用 Objective-C 编写，运行在 Objective-C runtime 中。为了不使原本的功能失效，Apple 在把所有的库重新实现一遍之前是无法撤掉 runtime 的。而在 Linux 中，Apple 没有任何负担，可以直接使用新的库。包括 Foundation 在内的系统库已经被打造成为了 C 语言和 Swift 编写的库，它不再运行在 Objective-C runtime 上了。此时调用函数将在编译时被链接，代码的性能会得到提高，而原本的 runtime 的黑魔法也都将失效。

经过笔者这段时间对 Linux 上的 Swift 的体验，它基本上可以满足日常项目的需要。虽然有些 API 还未被实现，但总能找到替代的办法。其运行也是稳定的，至少笔者手上的数据是零崩溃。目前语言还在快速发展中，问题会逐渐被修复，库函数也会逐渐被完善，因此笔者对 Linux 上的 Swift 还是充满信心的。

Swift开发服务端的资质

按照目前 Swift 的情况是具有在 Linux 上开发完整服务端软件能力的。

Linux 中使用 `Swift Package Manager (SPM)` 来构建项目，SPM 以文件夹目录为项目分支结构，以 `Module` 为单位构建项目。其中，SPM 支持 C/C++ Module，同时也支持 stl 等标准 C++ 库，这为我们提供了无限的扩展可能。

操作系统本身提供了大量 C 语言接口可供调用。Swift 可以原生访问系统的库，同样是以 Module 的形式。C 语言的数据结构会被自动桥接为 `struct`，直接使用。当然从语法上来讲，最佳的办法是自己再包装一层 Module，毕竟在 Swift 中操作的是 `String` 而不是 `char *`，是 `Data` 而不是 `unsigned char *`。

只要是提供 C/C++ 接口的各类服务，都可以集成进 Swift 项目中。不论是 MySQL，还是 Hadoop，它们都提供了 C 语言的库，如果想要在 Swift 的服务端中调用，直接写一个 C Module 开放接口即可。由于 Swift 并非纯面向对象语言，因此在这里开放接口甚至都不需要用类来封装。具体的做法将在后面的内容中提到。

One more thing, Swift 可以很简单地调用系统命令行。熟悉 Cocoa 的读者应该知道 `NSTask` 这个类可以调用系统命令，同样的我们也可以用这个在 Linux 中调用系统命令行。这个类在 Swift 中改成了 `Process`，在 Linux 中这个类叫 `Task`，使用方法相同。既然可以调用系统命令行，那么我们就可以直接执行 shell 脚本来更多的事了。考虑到 macOS 和 Linux 中使用不同的 API 所造成的麻烦，我们可以用条件编译来很方便地解决这个问题：

```
#if os(Linux)
//在Linux中编译的代码
#else
//在macOS中编译的代码
#endif
```

Swift开发服务端的优点

既然 Swift 拥有了开发完整服务端软件的能力，那么就可以用它开发服务端了。可是现在 .Net、Java 等各大服务端框架已经非常成熟，如果用 Swift 开发，到底有哪些好处呢？

首先是代码层面：

- 编译后为原生代码，静态链接，运行效率高。
- Swift 在字符串、集合等数据的业务处理上有着较好的性能和极简的语法。
- 仍然可以使用如闭包、GCD 等开发者熟悉的东西，为开发带来极大便利。
- 内存管理仍然采用引用计数，及时回收，避免了例如 C++ 忘记手动回收造成泄露和 Java 延时回收难以 Debug 的弊端。
- Swift 目前已经开源，正处于快速迭代的成长期，社区非常活跃，具有潜力。

项目层面：

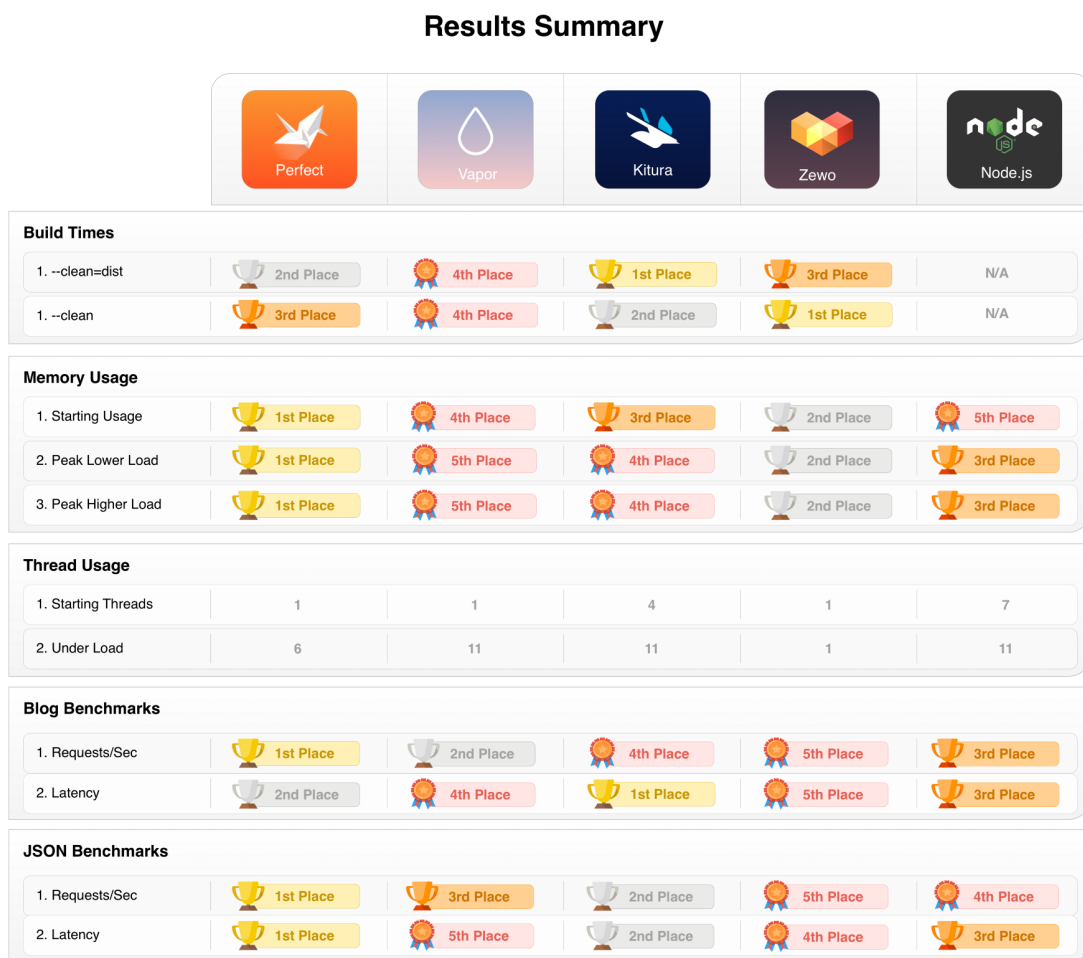
- 支持纯 C/C++ 语言 Module，无限的接入扩展性。
- SPM 支持直接从 GitHub 同步代码，方便管理。
- Swift 开发的服务端和 Swift 开发的 iOS、macOS 客户端可以共用同一套模型源文件，非常方便。
- 目前已有相对成熟的框架以及各类工具库可供使用，正在快速成长。

现在的 iOS 招聘信息里普遍都会考虑 Swift，有很多新项目已经采用 Swift 开发出来了。在客户端上，大家正在逐渐接受 Swift。那么在服务端，Swift 是否也可以占有一席之地呢？这个问题就留给时间回答吧。

实践

Swift服务端框架

目前 Swift 的后端框架主要有 Perfect、Vapor、Kitura 和 Zewo 等。前段时间掘金上有一篇文章在几个维度对比了几个框架的性能：[不服跑个分 - 顶级 Swift 服务端框架对决 Node.js](#)。文章中测试了以上四种框架以及 Node.js 在性能上的表现，结果汇总如下：



在测试结果中，Perfect 以非常优秀的成绩胜出，因此笔者最终也选择了 Perfect。PerfectlySoft 公司一直保持着比较高产的状态，同时也是一家非常亲中国的公司。不仅有简体中文的文档，还有官方微信账号。也有不少同行们收到了公司发来的中文邮件。

在实际的开发体验中，Perfect 的表现值得肯定。内存占用很低，官方提供的库也非常全面。从基本的 HTTP 服务器(Perfect-HTTPServer)，多线程库(Perfect-Threading)，日志库(Perfect-Logger)，到各类数据库链接库(Perfect-MySQL , Perfect-MongoDB 等)，甚至到 Hadoop(WebHDFS , MapReduce 等功能)和邮件库(Perfect-SMTP)都提供了。就在写文的今天，官方在服务器助手中新增了在 Mac 上交叉编译，一键测试 Linux 版的功能。这些更新每周都会在官方微信中推送，而且是简体中文的文章，这对中国开发者来说是极为友好的。关于详细的开发体验将在下文提到。

由于其他框架笔者接触的不多，因此就此带过。但也并不是说分数低其他框架就不好，例如 Vapor 的路由写法更简单一点，MVC 的结构更舒服一点，读者可以自由选择。另外对测试结果好奇的读者可以自行测试，测试代码都在原文链接之中。

基于 Perfect 的 HTTP Server 开发体验

如何使用 Perfect 开发服务端呢？这里笔者将分为开发过程和部署过程进行介绍。

开发过程

程序最终是跑在 Linux 机器里的，但依然可以继续 Mac 中开发和调试，最终再移植到 Linux 中进行编译运行。PerfectlySoft 提供了例子 [PerfectTemplate](#)，包含了简单的 HTTP 服务器创建的过程。README 中的过程是基于 SPM 编译管理的，但在 Mac 下，我们有更好的选择 —— Xcode。

克隆官方的例子后使用 `build` 命令自动下载依赖库：

```
$ swift build
```

之后这个命令生成 Xcode 工程：

```
$ swift package generate-xcodeproj
```

接着我们就可以像开发 macOS 应用那样开发服务端了。官方在文档中提醒我们不要直接编辑这个 `xcodeproj` 文件。因为有时我们需要修改 `Package.swift` 来下载更多依赖库，这时候这个 `xcodeproj` 会被重新生成，之前所做的所有内容都会被覆盖。本书的 Sample 只做演示用，所以并没有这样做。读者在自己正式的项目中要注意这一点，避免以后发生麻烦。

在现在版本的 README 中，提供了一种非常炫酷的创建服务的方式，直接构造一个多维数组，填写相关信息，之后就可以直接跑起一组服务：

```

import PerfectLib
import PerfectHTTP
import PerfectHTTPServer

let port1 = 8080, port2 = 8181

let confData = [
    "servers": [
        [
            "name": "localhost",
            "port": port1,
            "routes": [
                //...
            ],
            "filters": [
                //...
            ]
        ]
    ]
]

do {
    // Launch the servers based on the configuration data.
    try HTTPServer.launch(configurationData: confData)
} catch {
    fatalError("\(error)") // fatal error launching one of the servers
}

```

由于这种方式十分简洁干练但不利于读者理解内部实现的结构，笔者会使用原来的接口创建方法来讲。

在 Perfect 中创建 HTTP 服务极为简单，在路由的创建过程中，开发者只需要提供了一个处理该路由的回调函数，其余大量的工作 Perfect 已经帮我们完成了。当 Perfect 服务器接收到一个 HTTP 请求后，服务器首先会将请求交给当前已经注册了的过滤器。这些过滤器可能会修改或者重定向请求，转发给别的路由。当过滤器处理完成后服务端会寻找有没有相应的回调，如果有则会交给回调函数进行处理，如果没有则会返回 404。来直接看下面的例子，完成一个最简单的 HTTP Server 程序：

```

import Foundation
import PerfectLib
import PerfectHTTP
import PerfectHTTPServer

open class MyServer {

    fileprivate var server: HTTPServer

    internal init(root: String, port: UInt16) {
        //构造 HTTPServer 对象
        server = HTTPServer.init()
    }
}

```

```

//构造路由对象，这只是个容器，现在这里面并没有内容
var routes = Routes.init()
//配置路由，添加URL以及回调函数
configure(routes: &routes)
//将路由添加进服务
server.addRoutes(routes)
//设置端口和根目录
server.serverPort = port
server.documentRoot = root
}

//配置路由函数
fileprivate func configure(routes: inout Routes) {
    //添加接口，路径为/，方法为GET，回调函数为闭包
    routes.add(method: .get, uri: "/", handler: { request, response in
        //取得url中的参数，类型是`[(String, String)]`，遍历即可
        let param = request.params()
        //返回数据头
        response.setHeader(.contentType, value: "text/html")
        //返回数据体
        response.appendBody(string: "Hello World")
        //返回
        response.completed()
    })
}

//开始服务
open func start() {
    do {
        try self.server.start()
    } catch PerfectError.networkError(let err, let msg) {
        print("Network error thrown: \(err) \(msg)")
    } catch {
        print("Network unknow error")
    }
}
}

```

包含一个 URL 接口的 HTTP Server 就写完了，最终在 `main.swift` 中调用：

```

import Foundation

let myServer = MyServer.init(root: "Your Path To Root", port: 8080)

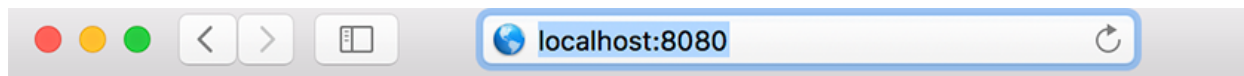
myServer.start()

```

此时可以看到控制台输出：

```
[INFO] Starting HTTP server on 0.0.0.0:8080 with document root "Your Path To Root"
```

现在我们就可以在浏览器看到结果了：



Hello World

如果在 Response 中返回页面 HTML 代码，那么这就是个 Web 服务器。如果在 Response 中返回 JSON/XML 字符串，那么这就是个业务接口，可以给手机 App 使用。

HelloWorld 固然简单，实际情况往往没那么简单。尚若要建个站点，必然包含了大量 `css`、`JavaScript` 以及图片等静态资源，这些静态资源如果需要开发者手动加入路由显然不现实。所以在 `PerfectHTTP` 中提供了 `StaticFileHandler` 模块帮我们实现了静态资源的处理。

若不修改任何配置，则在最初设置的 root 目录下所有的文件都是可以被直接下载的。描述文件 URL 的这些请求会被方法 `handleRequest` 处理，在路由设置不存在的情况下会访问本地文件，若文件存在则会直接发送文件。如果读者需要建立一个静态 Web 站点，甚至可以一个路由都不配置，直接运行一个空的 `HTTPServer`，这些静态资源会被直接开放在网络中，访问者可以直接根据 URL 访问不同的页面。当然，也可以手动映射静态资源，利用通配符将一个本地目录映射到一个 URL 上，具体实现过程可以参考官方文档，在此不多介绍。

另外官方提供的 `File` 模块也能以数据流的形式很便利地操作文件。在页面方面，Perfect 同样支持 `Mustache` 模板功能。官方提供的 `Perfect-Mustache` 库提供了相关的功能支撑。这是一个非常好用的页面模板引擎，一个很好的页面中动态元素替换的解决方案。在页面 HTML 代码中嵌入例如 `{{var}}` 的占位字段，在 Swift 代码中提供一个字典，将页面中的占位符替换成字典中的值。详细的使用过程可以参阅官方文档，这个库可以独立于服务器使用。

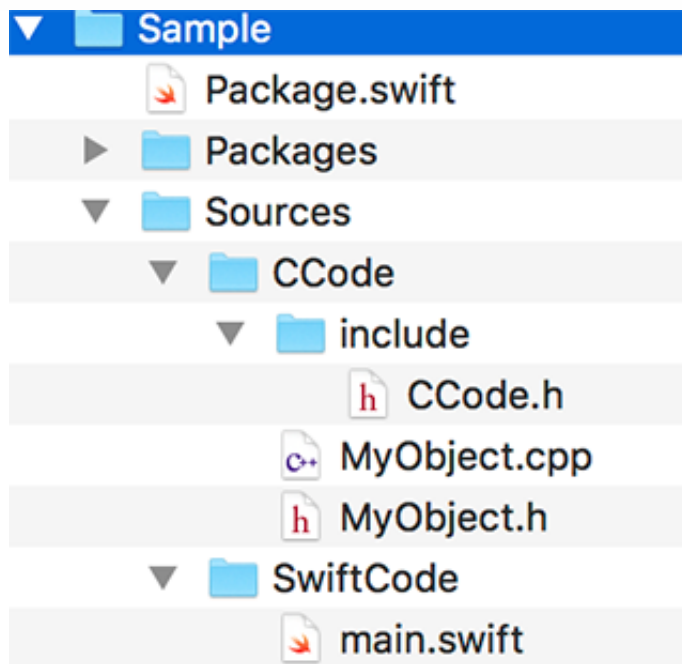
总之，官方提供了大量的工具库且正在以非常高的活跃度继续开发，大大便利开发者们。

部署过程

Linux 上的 Swift 环境配置在此不做介绍，具体配置过程可使用 PerfectlySoft 提供的安装脚本：[Perfect-Ubuntu](#)，或自行搜索配置。

现在我们的服务已经在 Mac 上跑起来了，那么如何部署到 Linux 服务器中呢？例子中的 HelloWorld 直接拷贝到 Linux 服务器中 build 是直接可以通过的，这个例子太过初级，这里讲个稍微复杂一点的帮助读者理解。

在 Linux 中我们使用 SPM 管理项目。例如我们的项目包含了 Swift 主函数代码和一个纯 C 语言的 Framework。在 Xcode 中开发时，可以按照自己的习惯建立两个工程，主工程链接库工程生成的库，这个过程相信大家都很熟悉了。但为了 Linux 工程的管理，源文件需要放在两个独立的文件夹中。具体目录结构如下图：



SPM 会按照 `Package.swift` 文件来为我们生成 `Target`，处理依赖关系。`SwiftCode` 模块依赖了 `CCode`，因此此时要将该文件改成这样：

```
import PackageDescription

let package = Package(
    name: "PerfectTemplate",
    targets: [Target(name: "SwiftCode", dependencies: ["CCode"])],
    dependencies: [
        .Package(url: "https://github.com/PerfectlySoft/Perfect-
HTTPServer.git", majorVersion: 2, minor: 0),
    ]
)
```

这时候在 Linux 中 build 该项目时，`CCode Module` 会被单独编译成 `CCode.so` 供主模块调用。无论是混编还是纯 Swift 项目也可以这样处理，当然，别想着混编 Objective-C 了，这儿已经没有 Objective-C runtime 了。按照这样的目录结构在 Linux 中 build，将会生成隐藏的 `.build` 目录，执行代码运行：

```
$ ./build/debug/SwiftCode
```

这样我们的服务就已经在 Linux 上跑起来了。嗯，如果代码确认没问题，需要发行 Release 版本呢？

```
$ swift build -c release
```

之后会生成 `./build/release` 文件夹，其中包含了 Release 版的库和可执行文件。

清理工程可以使用下面这个命令：

```
swift build --clean
//当然，也可以粗暴一点，直接删除.build目录，效果是一样的
rm -rf .build
```

如果需要连带依赖库一起清理，使用下面这个命令：

```
swift build --clean=dist
//同样的，可以粗暴地直接手动删除目录
rm -rf .build
rm -rf Packages
```

链接其他第三方库如 libhdfs.so 并编译 release 版本：

```
$ swift build -Xlinker -lhdfs -c release
```

静态编译：

```
$ swift build -static-stdlib
```

其他编译器功能请参阅：

```
$ swift --help
```

小结

通过以上这个例子，我们创建了一个简单的 HTTP 服务器，并且部署到了 Linux 服务器上。由于框架是开源的，甚至可以直接修改源代码来完成例如埋点等功能。在接下来的部分，笔者将会添加一些常规的服务来进一步让大家体会 Swift 在服务端的开发体验。

基于Perfect的常规服务接入体验

一个完整的服务器只包含 HTTP 服务和文件系统功能肯定是不够的，必须有一种方法让开发者接入其他第三方服务。第三方服务的 SDK 有 Swift 版的吗？嗯，很少很少。但多数厂商都会提供 C 语言接口，这就可以作为开发者们的突破口，利用 C 语言接口接入服务。

笔者会举两个简单的例子来介绍第三方服务接入的过程：**MySQL**和**HDFS**。

MySQL接入

对于厂商没有提供接口的服务，Perfect 官方“造了很多轮子”来帮助开发者接入。MySQL 的接入可以直接选用官方提供的 `Perfect-MySQL`。

编辑 Package.swift，在依赖库中加入对 `Perfect-MySQL` 的依赖：

```
.Package(url:"https://github.com/PerfectlySoft/Perfect-MySQL.git",
majorVersion: 2, minor: 0)
```

下载依赖后就可以使用了，同样来看一个简单的例子：

```
import Foundation
import PerfectLib
import PerfectHTTP
import MySQL
//定义在Linux中的数据库参数
#if os(Linux)
    let testHost = "数据库IP"
    let testUser = "数据库登录名"
    let testPassword = "数据库密码"
    let testSchema = "Schema名"
#else
    //定义在其他平台的数据库参数参数
#endif

internal class MyDB {
    //构造一个库中的MySQL对象
    fileprivate let mysql = MySQL.init()

    internal init?() {
        //设置客户端字符集，这是非常必要的操作，否则所有中文可能都会变成问号
        guard mysql.setOption(.MYSQL_SET_CHARSET_NAME, "utf8mb4") else {
            return nil
        }
        //连接数据库
        guard mysql.connect(host: testHost, user: testUser, password:
testPassword) else {
            return nil
        }
    }
    //执行SQL语句并返回结果
    @discardableResult
    internal func query(_ s: String) -> [[String?]]? {
        guard mysql.selectDatabase(named: testSchema),
mysql.query(statement: s) else {
            return nil
        }
        let results = mysql.storeResults()
        var resultArray = [[String?]]()
        while let row = results?.next() {
            resultArray.append(row)
        }
        return resultArray
    }
}
```

一个简单的类就可以帮助开发者操作数据库了，使用时直接调用 `query` 方法即可。

`Perfect-MySQL` 基于C接口实现，其默认链接的库为 `libmysqlclient.so`。官方为开发者做了一次中间封装，处理了大量指针操作，让语法更符合 Swift 标准。库中查询语句的结果会返回一个 `MySQL.Results` 对象，其包含了查询结果的记录数、字段数等方法可供调用。可能是为了保证性能，官方并未把查询结果全部取出来做 Swift 数据结构封装。数据仍然存在于C++层，以指针 `UnsafeMutablePointer<MYSQL_RES>` 进行操作，当 `MySQL.Results` 对象被 ARC 回收时释放 C++资源。这种方法相当于维护了一个服务端，在客户端操作类似于操作一个状态机，造成在处理查询结果上语法还是充满了 C Style，例如需要取第N条数据时必须先调用以下方法将指针移动到目标位置然后读取，读取时也只能用类似生成器的方法读：

```
//移动到第N条记录
public func dataSeek(_ offset: UInt)
//读取这条内容，指针自动后移到下一条记录处
public func next() -> Element?
```

需要注意的是，这个库默认链接的是 `libmysqlclient.so`，在库内部调用 `mysql_real_connect` 可能会出现线程安全问题。因此多线程调用 `connect` 方法时需要处理好线程问题。

HDFS接入

这是一个典型的调用C接口接入的服务。

HDFS 是一个运行在 Java 上的分布式文件系统，它提供了C接口可供开发者调用。在使用之前需要开发者配置相关环境以及在本机编译 Hadoop 的 Native 库，这个过程不在讨论范围之内。编译完成后会得到一系列 Hadoop 的库，在本文中只选用 HDFS 库，即 `libhdfs.so`。

在上文中提到了如何利用 SPM 建立 C Module，而C Module可以被自动桥接到 Swift。但不幸的是用于描述文件信息的结构体 `hdfsFileInfo` 中的变量在 Swift 中是无法被访问到的，因此在数据结构上需要在 C++ 层重新做一下封装。做完封装后将接口封装为纯C语言接口即可供 Swift 端调用。

这里需要一点C语言基础，由于接口数量众多，本文中仅举例连接 HDFS 的函数封装过程。根据 SPM 管理 Module 的目录结构，假设结构如下图所示：

名称
 Package.swift
▶  Packages
▼  Sources
▼  CCode
 CWrap.cpp
 CWrap.h
 HDFSType.h
▼  include
 CCode.h
▶  SwiftCode

- CCode.h：Module 暴露的接口。

```
#include "../CWrap.h"
```

- HDFSType.h：针对库中的数据结构的另一层封装，在原有数据结构前加入 cw_ 前缀定义新结构。
- CWrap.h：HDFS 接口头文件。

```
#include "HDFSType.h"
#ifdef __cplusplus
extern "C" {
    cw_hdfsFS cw_func_connect(const char* nn, cw_tPort port);
}
#endif
```

CWrap.cpp：HDFS 接口封装的具体实现过程。

```
#include "CWrap.h"
#include "HDFS头文件路径/hdfs.h"

cw_hdfsFS cw_func_connect(const char* nn, cw_tPort port) {
    hdfsBuilder * builder = hdfsNewBuilder();
    hdfsBuilderSetNameNode(builder, nn);
    hdfsBuilderSetNameNodePort(builder, port);
    hdfsBuilderConfSetStr(builder, "dfs.support.append", "true");
    hdfsBuilderConfSetStr(builder, "dfs.replication", "1");
    return (cw_hdfsFS)hdfsBuilderConnect(builder);
}
```

最终，方`cw_func_connect`能在 Swift 端被调用：

```
let fs = cw_func_connect("目标NameNode", cw_tPort(目标端口))
```

事实上笔者在C++层不仅是简单封装了接口，更是封装了功能。在平时的开发过程中经常也会在厂商 SDK 和自己的业务代码中间做封装，笔者将其移到了C++中实现，同时也可以暴露更少的数据结构和细节。当然这一层也可以放到 Swift 中去完成，但封装数据结构的工作量会更大。

最后使用如下命令链接 HDFS 库并编译：

```
$ swift build -Xlinker -lhdfs
```

实际应用中，笔者的项目是接入了 HDFS 的。在 Swift 端调用时，因为是原生调用，因此性能和稳定性都不错。但关于 HDFS 的开发，其实还存在很多坑。在 Mac 上开发时会遇到例如 `CLASSPATH` 环境变量未定义的问题，无法 `Debug` 执行的问题等等。读者若有兴趣可以自行研究。

小结

其实 Swift 端在调用原生服务时并没有给开发者带来太多的额外工作量，其麻烦的主要来源也在各语言桥接上。目前桥接C语言还是比较方便的，其他语言的接入笔者并没有深入研究。但随着 Swift 的发展，如果将来可以普及，那么会有更多厂商推出 Swift SDK，这些问题也就可以解决了。

基于BlueSocket的Socket服务端开发体验

很多服务是基于 Socket 的，那么 Swift 的 Socket 服务端开发体验又如何呢？这里笔者推荐使用 [BlueSocket](#)。这是IBM推出的一款在 iOS、macOS 和 Linux 上通用的 Socket 库，在三个平台上通用，能在调试上带来很大的便利。

做 Socket 的应用层开发，开发者只要操作收、发缓冲区，即可实现数据的接收和发送。在 macOS 和 iOS 上，Socket 相关 API 存在于 `Darwin` 库中，而 Linux 上则是 `Glibc` 库中，是一组C语言接口，开发者也可以自己向内核申请 Socket 去完成业务逻辑。在这里BlueSocket 已经帮助开发者实现了 Socket 的内核调用了。

既然是调用系统库的C语言接口，那么调用逻辑肯定是跟其他语言是一样的，这一点可以让熟悉 Socket 开发的开发者直接上手。现在来看一个简单的 TCP Socket 服务端的创建过程：

```

//声明服务器类
internal class SocketService {
    //停止标记
    fileprivate var stopFlag = false
    //监听Socket
    fileprivate var listenSocket: Socket
    //记录客户端列表，以便退出时关闭连接
    fileprivate var clients = Dictionary<Int32, Socket>()

    //开始运行服务端
    internal func start(port: Int) throws {
        //创建一个Socket
        self.listenSocket = try Socket.create()
        //开始监听
        try listenSocket.listen(on: port)
        //死循环监听客户端的连接
        while !stopFlag {
            //当有客户端连接时添加到客户端列表
            let client = try listenSocket.acceptClientConnection()
            self.add(client: client)
        }
    }

    //停止时关闭客户端所有连接
    internal func stop() {
        stopFlag = true
        for socket in clients.values {
            socket.close()
        }
        listenSocket.close()
    }

    //添加客户端方法
    fileprivate func add(client: Socket) {
        DispatchQueue.global().async {
            do {
                //登记客户端
                self.clients[client.socketfd] = client
                //死循环读取数据
                while !self.stopFlag {
                    var tmpBuffer = Data.init()
                    //读取接收缓冲区数据，如果没有数据，线程会等在这里
                    let readSize = try client.read(into: &tmpBuffer)
                    //如果长度为0，表示连接断开
                    if readSize == 0 {
                        break
                    } else {
                        //处理缓冲区数据
                    }
                }
            }
        }
    }
}

```

```

        }
        //注销客户端的登记
        self.clients[client.socketfd] = nil
    } catch {
        //处理异常
    }
}
}
}
}
}

```

从代码中可以看出，每当一个客户机申请连接时，GCD 都会获取一条线程来跑。理论上一个客户端独占一条线程，这和其他语言的开发是一样的，但不同的是，Swift 使用的 GCD 和闭包在这里能极大提高代码可读性。`read` 方法会将缓冲区数据写出来交给开发者处理。TCP 的数据是基于流的，包与包之间没有固定的分隔，所有包粘在一起。这里如果需要拆固定长度的包，Swift 有一种很好的方式。

代码中定期调用 `read` 方法，它会将缓冲区里所有的内容都写出来。这个内容的长度是不确定的，取决于发送方和网络状况，以及这段代码中处理数据部分的耗时。在实际开发中开发者会定义自己的数据包，可能是有意义的间隔符，也有可能是固定长度。笔者在这里举一个固定长度包的例子，利用 Swift 的数组操作方法来拆包：

```

//定义数据包长度
let packetSize = 4096
//声明存放数据的缓冲区，如需限制大小，可自己修改代码
var buffer = Array<UInt8>()
while !self.stopFlag {
    //临时缓冲区
    var tmpBuffer = Data.init()
    let readSize = try client.read(into: &tmpBuffer)
    if readSize == 0 {
        break
    } else {
        //将临时缓冲区的数据追加到上面定义的缓冲区里
        buffer += tmpBuffer.bytes
        //当数据足够长时
        while buffer.count >= packetSize {
            //获取包长度的数据
            let packetBytes = Array(buffer.dropLast(buffer.count -
self.packetSize))
            //删去取出的部分
            buffer.removeFirst(packetSize)
            //处理数据
        }
    }
}
}
}
}
}

```

以上代码能实现一个缓冲区队列来按顺序取所需要的数据包，代码简单且有效。但需要注意的是这里的 `dropLast` 和 `removeFirst` 都是时间复杂度 $O(n)$ 的方法。但作为 Socket 缓冲区，这个数组并不会很大，因此这里对性能的影响很小。但如果服务端对性能极为敏感且缓冲区有可能会很大的时候，这里最好采用常规**翻转栈**的方法设计队列，这里不再涉及。

在设计模式上例如处理数据部分，可以使用 `Delegate` 来将数据包回调给委托方实现，这样一个通用的定长数据包的 `SocketServer` 工具类就这样简单地实现了。关于客户端的实现，同样极为简单，开发者可以参考上面给出的 `BlueSocket` 的链接，查看官方的 README 文档以及历程，笔者在这里不再贴出代码了。

小结

使用 `BlueSocket` 能让开发者在很短的时间内创建自己的 Socket 服务器。无论是做即时通信，文件传输还是直播，Swift 都能轻松胜任。正如本文开头提到的，Swift 开发服务端时，GCD 等工具和极简的语法能给开发者带来极大便利。同时也是个正在快速发展的语言，将来会有更多这样优秀的工具库的诞生，让开发过程更加高效。

Swift服务端和iOS客户端同时进行的开发体验

使用 Swift 做服务端一个很大的优势就在于，可以和客户端共用一套模型源文件。在服务端传送一个对象到客户端是业务开发中经常遇到的需求，数据发送时开发者有很多方法，转成数据二进制流、XML字符串、JSON字符串、ProtoBuf等发送，当客户端收到时再转模型。如果开发者使用 Swift 开发，那么这套代码只需要写一次，且以后再修改需求加字段，也不会出现服务端改了客户端忘改了的现象了。

笔者在这里举一个简单的使用 JSON 字符串进行传输的模型，解析库使用的是 [SwiftlyJSON](#)。

```
//声明一个协议，可以被JSON序列化以及反序列化
public protocol JSONable {
    init?(json: JSON)
    func toJSON() -> JSON
}
```

```
//声明对象，遵守JSON序列化协议
open class MyObject: JSONable {

    open var mName: String!

    public required init?(json: JSON) {
        guard let mName = json["mName"].string else {
            return nil
        }
        self.mName = mName
    }

    open func toJSON() -> JSON {
        var dict = Dictionary<String, Any>()
        dict["mName"] = mName
        return JSON.init(dict)
    }
}
```

建完模型以后，这个源文件在两端都是可以用的，服务端和客户端同时引用这个源文件。当需求发生变更，需要加入字段时，服务端开发者直接修改这个源文件，由于客户端引用的也是同一个文件，因此客户端那边也被修改了。如果服务端需要在对象里加入一个验证算法，例如根据当前时间计算出一个 Key 来验证，那么直接在这个源文件里写就可以了，客户端开发者根本不需要关心这个算法是什么，叫什么，放在哪，因为服务端的哥们已经全部完成了。当对象被序列化调用 `toJSON` 方法时，这个方法已经被服务端开发者重写了，因此新的参数已经被加入序列了。因此这些与客户端业务毫无关系的代码，甚至可以做到不需要客户端开发者的参与就可以完成。

另外关于跨平台的问题，与 Cocoa 无关的 Swift 代码，理论上是可以三个平台中通用的。笔者在前段时间完成了包括文件流和内存流的操作库，这些代码在一行未改的情况下在三个平台都可以直接使用，因此极大方便了“现造的轮子”在项目中的快速推进。

综上，在和 iOS 和 macOS 客户端协作开发的情况下，Swift 有着天生的优势，为开发提供便利。

尾声

问题和不足

在 macOS 上正常工作的代码在 Linux 上并不总能正常工作。笔者在开发过程中也遇到过许多问题。主要的问题归纳起来可以总结为以下几点：

- **未实现的 API**：一些较为常用的 API 未被实现，典型的有 `String` 的 `init(contentsOf url: URL)`，`FileManager` 的 `default` 属性等等。不过这个问题随着时间的推进，最终都将被解决。
- **API 执行结果不一致**：体现为 macOS 上工作良好的代码在 Linux 上罢工。例如从 `Data` 构造 `String` 时若遇到 `\0` 字符，在 macOS 上会忽略这个字符，而在 Linux 上字符串会直接截断。这些问题可能由于底层实现不同所导致的。
- **API Bug**：表现为 macOS 上工作良好的代码在 Linux 出现非代码逻辑错误的问题。例如同时在 N 条使用 `DispatchQueue.global()` 获得的线程里使用 `Data` 的 `init(contentsOf url:`

URL) 时直接崩溃报错 `fatal error` 的问题。

大多数情况下发生以上情况都能找到替代的 API 来实现。例如上面提到的 `Data` 的 `init(contentsOf url: URL)` 时报错的问题可以使用 `URLSession` 来解决, `DateFormatter` 时区错乱的问题可以使用 `TimeZone` 的 `init(secondsFromGMT: Int)` 来手动设置时差秒数。但这些都是临时的解决方案。笔者认为只有当原生的 API 足够可靠时, 才会吸引更多的开发者加入阵营。

总结和体会

本文主要讲了如何使用 Swift 语言开发简单的 HTTP 和 Socket 服务器, 以及如何连接 MySQL 数据库和 HDFS, 并最终部署到 Linux 服务器上。所用到的框架为 `Perfect` 和 `BlueSocket`, 以及和 `iOS` 客户端协作开发时的体验。

使用 Swift 开发服务端, 开发时的体验是极好的。干净的文件结构, 没有各项杂乱的环境配置, 因此工程也极少出问题。在 Xcode 中开发时由于也是笔者比较熟悉的环境, 可以使用 GCD 等熟悉的工具, 因此开发过程是非常舒服的。最终部署时也非常简单, 运行情况也比较好。Swift 诞生时间虽然不长, 但是网络上资料已经比较多了, 不再是以前的满眼 HelloWorld 了, Perfect 官方更是提供了简体中文的文档, 因此现在想要深入学习 Swift 的话时机是非常合适的。对于目前存在的问题, 相信随着时间的推移, 会逐步得到解决。

Swift 虽好, 但完善之路任重而道远。

参考链接

- [PerfectlySoft 官方网站](#)
- [PerfectlySoft GitHub](#)
- [Perfect 简体中文官方文档](#)
- [IBM BlueSocket GitHub](#)
- [笔者博客中的 Perfect TAG](#)

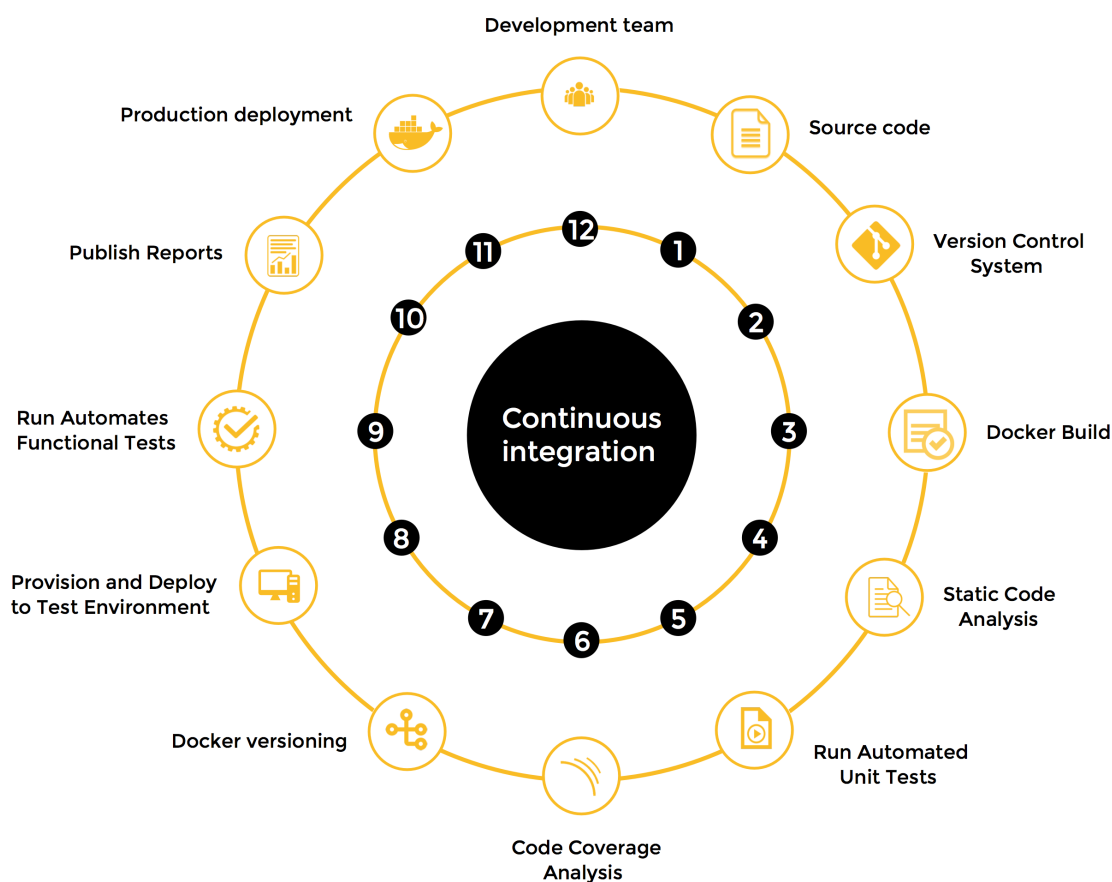
Fastlane 实战演练

众所周知，现在App的竞争已经到了用户体验为王，质量为上的白热化阶段。用户们都是很挑剔的。每次快速迭代的周期结束，App上线，从上传截图，打包，上传包到Itunes Connect，提交，每次这些“枯燥繁琐”的步骤都要折腾半个多小时。

在日常开发中，打包也是最后上线不可缺少的环节，如果需要把工程打包成 ipa 文件，通常的做法就是在 Xcode 里点击「Product -> Archive」，当整个工程 archive 后，然后在自动弹出的

「Organizer」中进行选择，根据需要导出 ad hoc，enterprise 类型的 ipa 包。虽然Xcode已经可以很完美的做到打包的事情，但是还是需要我们手动点击5，6下。加上我们现在需要持续集成，用打包命令自动化执行就顺其自然的需要了。

那么持续集成能给我们带来些什么好处呢？这里推荐一篇文章[文章](#)，文章中把[Continuous integration](#) (CI) and [test-driven development](#) (TDD)分成了12个步骤。然而带来的好处成倍增加，有24点好处。



Fastlane 简介

fastlane是一套自动化打包的工具集，用 Ruby 写的，用于 iOS 和 Android 的自动化打包和发布等工作。gym是其中的打包命令。

fastlane 的官网看[这里](#), fastlane 的 github 看[这里](#)

fastlane包含了我们日常编码之后要上线时候进行操作的所有命令。

deliver: 上传屏幕截图、二进制程序数据和应用程序到AppStore
snapshot: 自动截取你的程序在每个设备上的图片
frameit: 应用截屏外添加设备框架
pem: 可以自动化地生成和更新应用推送通知描述文件
sigh: 生成下载开发商店的配置文件
produce: 利用命令行在iTunes Connect创建一个新的iOS app
cert: 自动创建iOS证书
pilot: 最好的在终端管理测试和建立的文件
boarding: 很容易的方式邀请beta测试
gym: 建立新的发布的版本, 打包
match: 使用git同步你成员间的开发者证书和文件配置
scan: 在iOS和Mac app上执行测试用例

一个最最完整的发布过程可以用fastlane描述成下面这样

```
lane :appstore do
  increment_build_number
  cocoapods
  xctool
  snapshot
  sigh
  deliver
  frameit
  sh "./customScript.sh"

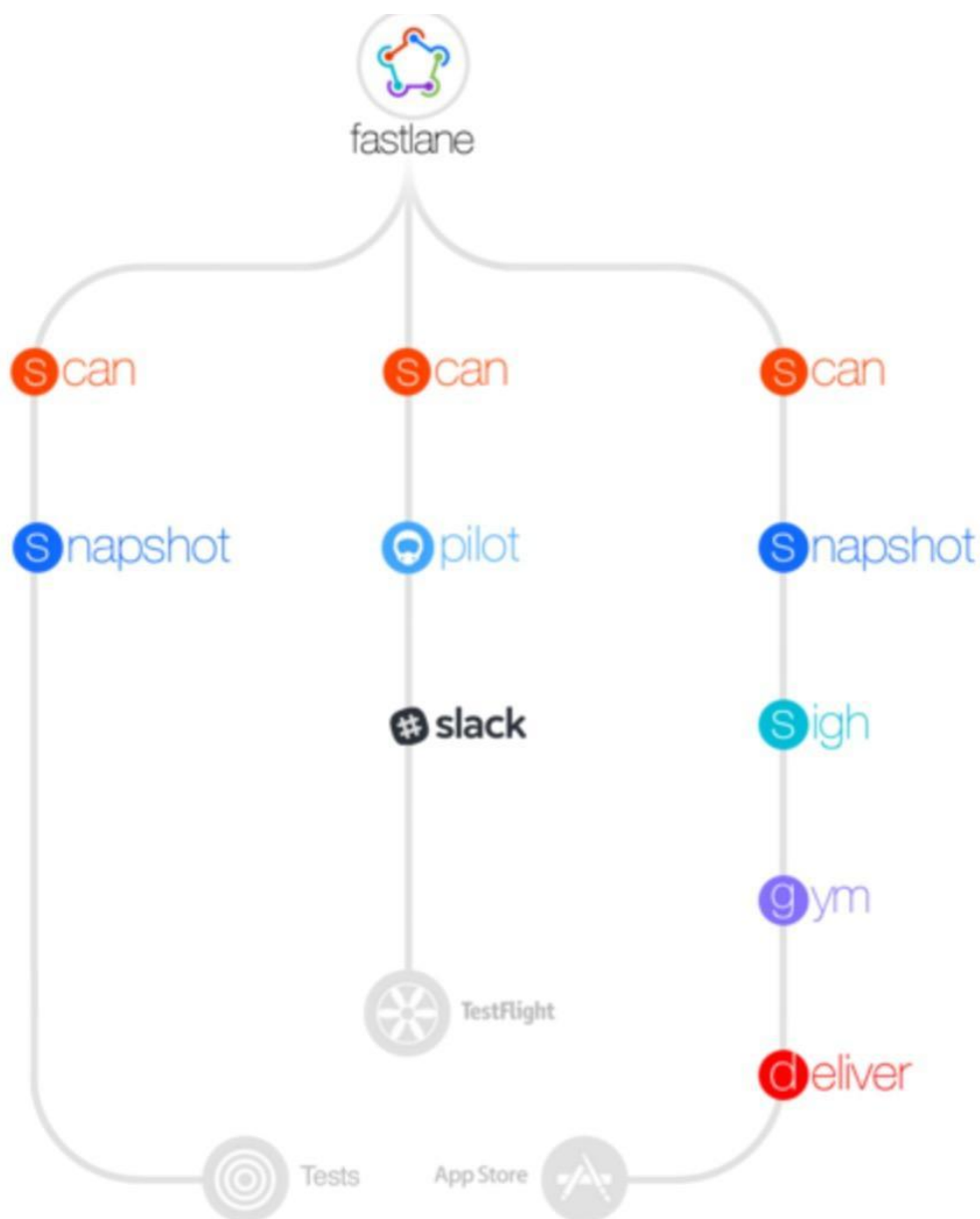
  slack
end
```

- 1.提高版本号
- 2.cocoapods进行相关pod配置
- 3.xctool进行编译
- 4.snapshot自动生成截图
- 5.sigh处理 provision profile 相关的事情
- 6.deliver上传截图
- 7.frameit将应用截图快速的放入对应的设备尺寸中
- 8.执行一些自动化的脚本
- 9.把结果发送到slack

这是一个完成的自动化的过程。不过实际发布过程中, 截图那部分笔者所在公司还是自己手动上传了, fastlane基本还是用来自动化打包。

这些命令的详细分析可以见本刊的另外一篇文章《fastlane 的神秘花园》里面的详细分析。

Fastlane实战



这张图是官方对fastlane的功能的总结。接下来我们就来看看具体实战中怎么用。

安装fastlane

1. 确保xcode命令行工具已经安装

```
xcode-select --install
```

2. 参考官方文档有三种安装方式任选一种进行安装

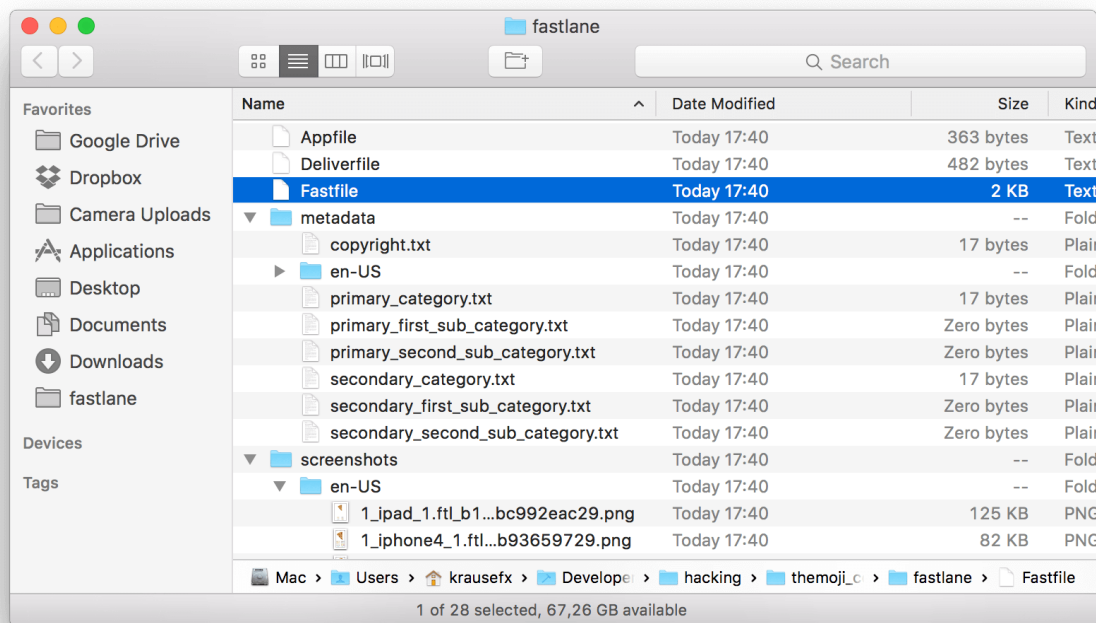
Homebrew	Installer Script	Rubygems
macOS	macOS	macOS or Linux with Ruby 2.0.0 or above
<code>brew cask install fastlane</code>	Download the zip file . Then double click on the <code>install</code> script (or run it in a terminal window).	<code>sudo gem install fastlane -NV</code>

初始化

在项目根目录下，初始化Fastlane：

```
fastlane init
```

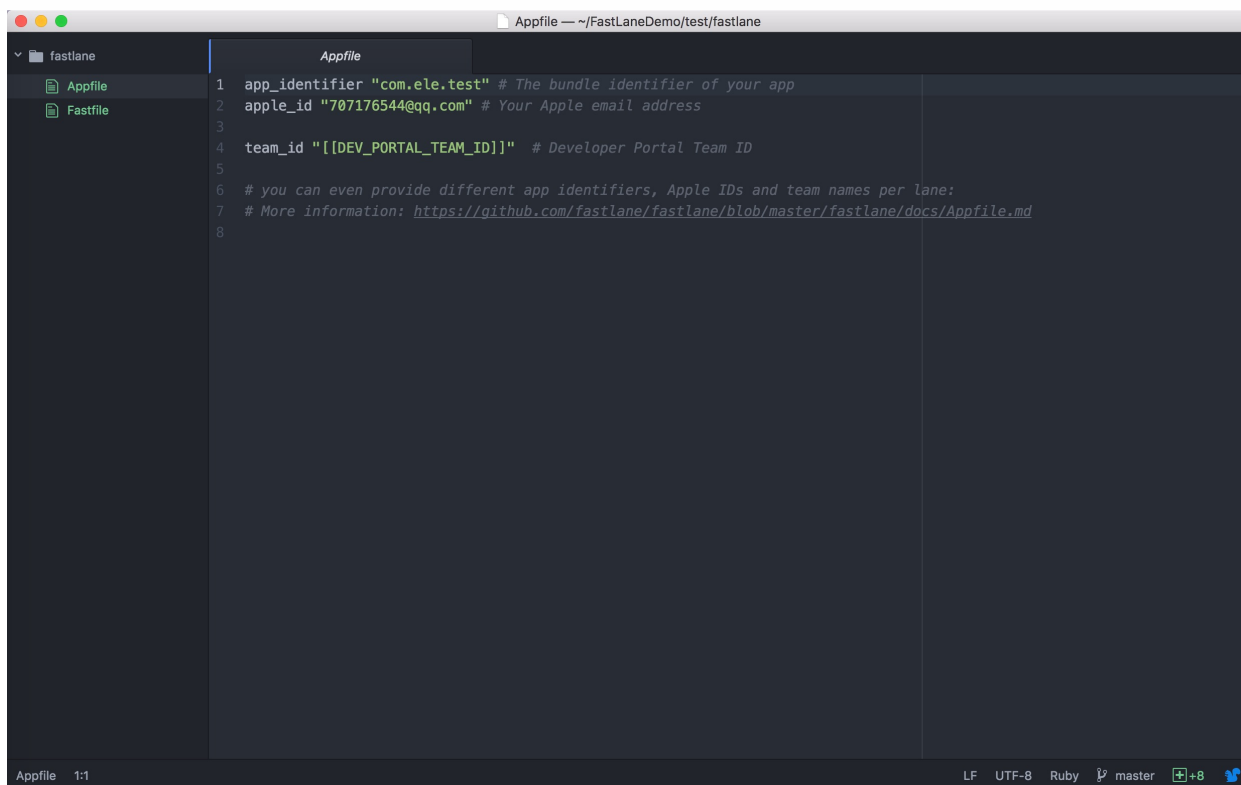
使用命令行工具进入项目目录执行fastlane init命令.初始化的过程中会要求填写一些项目信息比如 Apple ID，fastlane会自动检测当前目录下项目的App Name和App Identifier。如果检测的不对，选择n自行输入。同时会在项目中生成一个fastlane的文件夹。



目录结构如下：

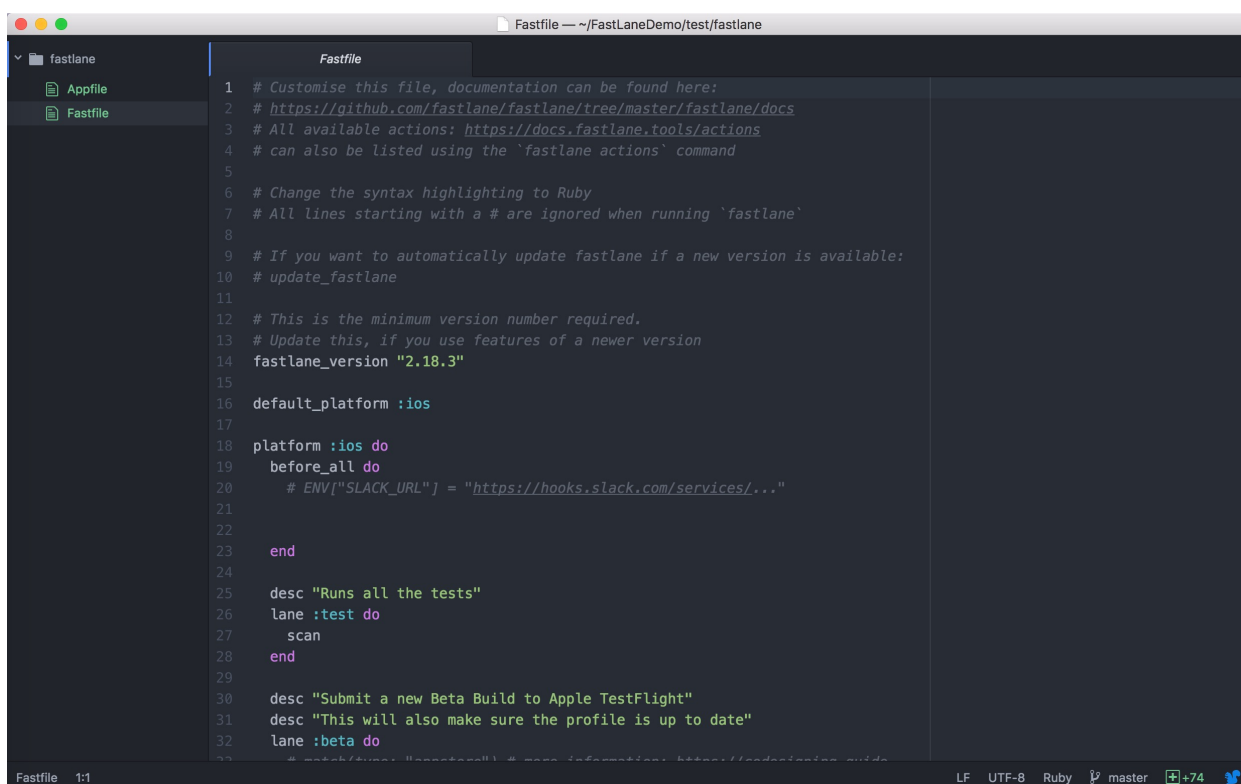
```
fastlane
├─ Appfile
├─ Deliverfile
├─ Fastfile
├─ metadata
│   ├─ copyright.txt
│   ├─ en-US
│   │   ├─ description.txt
│   │   ├─ keywords.txt
│   │   ├─ marketing_url.txt
│   │   ├─ name.txt
│   │   ├─ privacy_url.txt
│   │   ├─ release_notes.txt
│   │   └─ support_url.txt
│   ├─ primary_category.txt
│   ├─ primary_first_sub_category.txt
│   ├─ primary_second_sub_category.txt
│   ├─ secondary_category.txt
│   ├─ secondary_first_sub_category.txt
│   ├─ secondary_second_sub_category.txt
│   └─ zh-Hans
│       ├─ description.txt
│       ├─ keywords.txt
│       ├─ marketing_url.txt
│       ├─ name.txt
│       ├─ privacy_url.txt
│       ├─ release_notes.txt
│       └─ support_url.txt
└─ screenshots
    ├─ README.txt
    ├─ en-US
    │   └─ png图片
```

上面这些文件中，最重要的两个文件就是Appfile和Fastfile。

A screenshot of a code editor window titled 'Appfile -- ~/FastLaneDemo/test/fastlane'. The left sidebar shows a file tree with 'fastlane' containing 'Appfile' and 'Fastfile'. The main editor area shows the 'Appfile' content with line numbers 1 through 8. The code defines app identifiers, Apple ID, and team ID, with comments explaining their purpose and providing a link to Fastlane documentation. The status bar at the bottom indicates 'Appfile 1:1', 'LF UTF-8 Ruby', and 'master' branch with 8 additions.

```
1 app_identifier "com.ele.test" # The bundle identifier of your app
2 apple_id "707176544@qq.com" # Your Apple email address
3
4 team_id "[[DEV_PORTAL_TEAM_ID]]" # Developer Portal Team ID
5
6 # you can even provide different app identifiers, Apple IDs and team names per lane:
7 # More information: https://github.com/fastlane/fastlane/blob/master/fastlane/docs/Appfile.md
8
```

Appfile里面存放了App的基本信息包括app_identifier、apple_id、team_id。如果在init的时候你输入了正确的appId账号和密码会在这里生成正确的team_id信息。如果没有team，这里就不会显示。

A screenshot of a code editor window titled 'Fastfile -- ~/FastLaneDemo/test/fastlane'. The left sidebar shows the same file tree. The main editor area shows the 'Fastfile' content with line numbers 1 through 32. The code contains comments about documentation, actions, and syntax, followed by configuration for the default platform (ios), a before_all hook for Slack, and two lane definitions: 'test' and 'beta'. The status bar at the bottom indicates 'Fastfile 1:1', 'LF UTF-8 Ruby', and 'master' branch with 74 additions.

```
1 # Customise this file, documentation can be found here:
2 # https://github.com/fastlane/fastlane/tree/master/fastlane/docs
3 # All available actions: https://docs.fastlane.tools/actions
4 # can also be listed using the `fastlane actions` command
5
6 # Change the syntax highlighting to Ruby
7 # All lines starting with a # are ignored when running `fastlane`
8
9 # If you want to automatically update fastlane if a new version is available:
10 # update_fastlane
11
12 # This is the minimum version number required.
13 # Update this, if you use features of a newer version
14 fastlane_version "2.18.3"
15
16 default_platform :ios
17
18 platform :ios do
19   before_all do
20     # ENV["SLACK_URL"] = "https://hooks.slack.com/services/..."
21
22   end
23
24   desc "Runs all the tests"
25   lane :test do
26     scan
27   end
28
29   desc "Submit a new Beta Build to Apple TestFlight"
30   desc "This will also make sure the profile is up to date"
31   lane :beta do
32     # setup/submit to TestFlight # more information: https://docs.fastlane.tools
```

Fastfile是最重要的一个文件，在这个文件里面可以编写和定制我们打包脚本的一个文件，所有自定义的功能都写在这里。

如果在init的时候选择了在iTunes Connect创建App，那么fastlane会调用produce进行初始化，如果现在还不想创建，也可以之后再运行produce init进行这个流程。如果不执行produce的流程，deliver的流程不会被执行，当然之后也可以deliver init运行完全一样的流程。

在iTunes Connect中成功创建App之后，fastlane的文件夹里面就有Deliverfile文件了。

Deliverfile文件里面主要是deliver的配置文件和Deliverfile的一些帮助。

Fastfile文件

Fastfile这个文件，就是我们打包,发布到fir,testFlight, appstore等操作的配置文件。

```
fastlane_version "2.18.3"

default_platform :ios

platform :ios do
  before_all do
    # ENV["SLACK_URL"] = "https://hooks.slack.com/services/..."

  end

  desc "Runs all the tests"
  lane :test do
    scan
  end

  desc "Submit a new Beta Build to Apple TestFlight"
  desc "This will also make sure the profile is up to date"
  lane :beta do
    # match(type: "appstore") # more information: https://codesigning.guide
    gym(scheme: "BlusinessAreaPlat") # Build your app - more options
  available
    pilot

    # sh "your_script.sh"
    # You can also use other beta testing services here (run `fastlane
actions`)
  end

  desc "Deploy a new version to the App Store"
  lane :release do
    # match(type: "appstore")
    # snapshot
    gym(scheme: "BlusinessAreaPlat") # Build your app - more options
  available
    deliver(force: true)
    # frameit
  end

  # You can define as many lanes as you want

  after_all do |lane|
    # This block is called, only if the executed lane was successful
```

```

    # slack(
    #   message: "Successfully deployed new App Update."
    # )
  end

  error do |lane, exception|
    # slack(
    #   message: exception.message,
    #   success: false
    # )
  end
end
end

```

接下来解析一下Fastfile文件里面的所有参数：

fastlane_version

指定fastlane最小版本。

default_platform

指定当前平台，可以选择ios,android,mac。

before_all

这个指的是在执行每一个lane之前都先执行这个功能。

after_all

在每个lane执行之后都执行这个功能。

error do

在每个lane执行出错的时候都执行这个功能。

desc

一个lane的描述，一般说明lane的用途。

lane

任务的名字。我们在执行这个任务的时候，调用命令的格式为 fastlane 任务的名字，比如fastlane release。

do

后面跟着具体指令。在Fastfile文件中会有很多的lane，我们通俗的把理解为小任务，每个小任务都负责一个功能。然后我们调用不同的小任务，来实现打包、上传到development、上传到testFlight、上传到app store等功能。至于我们怎么打包，怎么上传，上传到哪里，上传之后结束做什么事情，都是在这里设置的。

Fastfile文件的编写

一般我们会打debug包和release包。下面就分别给出这两种包的例子：

debug包配置如下：

```
desc "Build a new version use the debugMyApp"
lane : debugMyApp do |op|
  increment_version_number(version_number: op[:version])
  increment_build_number(build_number: op[:version])

  set_info_plist_value(path: "./xxx/Info.plist",
                        key: "UIFileSharingEnabled",
                        value: true)

  set_info_plist_value(path: "./xxx/hostAddress.plist",
                        key: "host",
                        value: "https://halfrost:xx/xxx/xxx")

  # 将Development版本的.mobileprovision文件保存在里面，名称随意。
  update_project_provisioning(profile:
"./provisions/development.mobileprovision")

  update_project_team(path: "xxx.xcodeproj",
                      teamid: "XXXXXXX")

  gym(use_legacy_build_api: true,
      output_name: "debugMyApp",
      silent: true,
      clean: true,
      configuration: "Debug",
      buildlog_path: "./fastlane.log",
      codesigning_identity: "iPhone Developer: xxx (xxxxxxxxxx)",
      output_directory: "/Users/xxx/Desktop"
  )
end
```

release包配置如下：

```

desc "Deploy a new version to the App Store"
lane :releaseMyApp do |op|
  increment_version_number(version_number: op[:version]) #根据入参version获取
app版本号
  increment_build_number(build_number: op[:version]) #将build号设置与app版本
号相同

  # 设置app的info.plist文件项
  set_info_plist_value(path: "./xxx/Info.plist", #info.plist文件目录
                        key: "UIFileSharingEnabled", # key, 将plist文件以
Source Code形式打开可查询对应的key
                        value: false) # value

  # 设置自定义plist文件项, 用于给app配置不同的服务器URL
  set_info_plist_value(path: "./xxx/hostAddress.plist",
                        key: "host",
                        value: "https://halfrostApp:xx/xxx/xxx")

  # 更新Provisioning Profile
  # 在项目当前目录下创建provisions文件夹, 并将App Store版本的.mobileprovision文件
保存放在里面。
  update_project_provisioning(profile:
"./provisions/appstore.mobileprovision")

  # 更新项目团队project_team
  update_project_team(path: "xxx.xcodeproj",
                      teamid: "XXXXXXXX")

  # 开始打包
  gym(use_legacy_build_api: true,
      output_name: "appstore", # 输出的ipa名称
      silent: true, # 隐藏不必要的信息
      clean: true, # 在构建前先clean
      configuration: "Release", # 配置为Release版本
      codesigning_identity: "iPhone Distribution: xxx Co.,Ltd.
(XXXXXXXXXX)", # 代码签名证书
      buildlog_path: "./fastlane.log", # fastlane构建ipa的日志输出目录
      output_directory: "/Users/xxx/Desktop") # ipa输出目录

end

```

如果需要代码提交完成之后打出2个包, 那么再定义一个lane:

```
desc "build all version ipa"
lane :all do |op|
  t = op[:version]
  debugMyApp version:t
  releaseMyApp version:t
end
```

最终我们打包的时候只要输入：

```
fastlane debugMyApp version:7.10.9
```

这样就可以打包出debug包，后面跟的是一个版本号。

```
fastlane releaseMyApp version:5.9.10
```

这样就可以打包出release包，后面跟的是一个版本号。

```
fastlane all version:1.9.0
```

这样就可以一个命令打出2个包，版本号都是1.9.0

最后如果还有更多自定义的操作的话，可以查看官方文档：

fastlane的官方repo地址：

<https://github.com/fastlane/fastlane>

fastlane的快速上手文档：

<https://docs.fastlane.tools/>

fastlane支持的action文档

<https://docs.fastlane.tools/actions/>

最后推荐一下官方给的一些例子，是国外很多优秀的例子，可以直接借鉴过来。

<https://github.com/fastlane/examples>

找到自己想要的，或者相似的操作，直接下载Fastfile即可。

结尾

打包对于开发人员来说是一件很耗时，而且没有很大技术含量的工作。如果开发人员一多，相互改的代码冲突的几率就越大，加上没有产线管理机制，代码仓库的代码质量很难保证。团队里面会花一些时间来解决冲突，解决完了冲突还需要自己手动打包。这个时候如果证书又不对，又要耽误好长时间。这些时间其实可以用自动化脚本来节约起来的。一天两天看着不多，但是按照年的单位来计算，可以节约很多时间！希望大家都能感受到自动化流水线操作带来的高效！

Fastlane 的神秘花园

时光荏苒，岁月如梭。转眼间来到了2017年，在这个机器与人脑针锋相对的年代，自动化是我们躲不开又逃不掉的话题。今天要介绍的是一款 iOS/Android 上用于打包发布的持续集成工具链。由于市面上已经有许多介绍 fastlane 用法文章了，所以本文将从源码的角度，走进 fastlane 的秘密花园，带你看看这套工具背后的故事。

1. fastlane 工具集概览

本节，将汇总介绍 fastlane 提供的工具，后面将挑选一些命令，从源码的角度进行分析。

- `deliver`: 将应用截图，元数据和 app 上传到 App Store
- `supply`: 将安卓 app 和元数据上传到 Google Play(我知道这个命令没人用)
- `snapshot`: 在 iOS 或者 tvOS 的设备上自动的拍摄截图
- `screengrab`: 在安卓设备上自动拍摄截图(应该也没人用)
- `frameit`: 将应用截图快速的放入对应的设备尺寸中
- `pem`: 自动的生成或者更新推送配置文件
- `sigh`: 负责处理 provision profile 相关的事情
- `produce`: 使用命令行在 iTunes Connect 后台和 Dev Portal 创建新的 iOS app
- `cert`: 自动生成并维护签名证书
- `spaceship`: 一个获取苹果开发者后台和 iTunes Connect 的 Ruby 库
- `pilot`: 通过终端优雅的管理 TestFlight 用户
- `boarding`: 以最简单的方式，邀请你的 TestFlight 用户
- `gym`: 打包 iOS App，从未如此简单
- `match`: 通过 Git，在你的团队中间，轻松的同步证书和配置文件
- `scan`: iOS 和 Mac app 上最简单的跑测试的方式。

2 FastlaneCore

在详细介绍每个工具之前，有必要先了解一下 FastlaneCore，这是 fastlane 的基础库，后面的工具大量使用到这个，所以先准备点预备知识是很有必要的。FastlaneCore 模块的代码很多，大致上可以分为3个部分：配置项模块，基础UI模块，基础工具模块。

配置项模块有4个文件，分别是: `config_item.rb`, `commander_generator.rb`, `configuration.rb`, `configuration_file.rb`。

`config_item.rb` 文件中定义了 `ConfigItem` 类，该类定义了基本的可配置项以及一些基本的验证函数。大约是这样：

```

# 可配置项
class ConfigItem
  # [符号]命令参数或key
  attr_accessor :key

  # [字符串]环境变量，当没有设置其他值的时候生效
  attr_accessor :env_name

  # [字符串] 显示给用户的描述文字
  attr_accessor :description

  # [字符串]长度为一个字符的命令参数（例如：-f）
  attr_accessor :short_option

  # 默认值，当没有给定值或者环境变量时生效
  attr_accessor :default_value

  # 一个可选的 block,当设置了新的值的时候会被调用。用来检验值是否合法，
  # 可以用作类型检验或者查看文件(文件夹)是否存在。
  # 如果发生错误，需要抛出一个异常，并且将错误信息设置为红色。
  attr_accessor :verify_block

  # [数组] 装载冲突的配置 key(@param key) 的数组，这使得解决冲突的方法更科学
  attr_accessor :conflicting_options

  # 可选的 block, 当配置项发生冲突时被调用
  attr_accessor :conflict_block

  # [String] 判断配置是否被废弃。 一个废弃的配置项应该是可选的。
  attr_accessor :deprecated

  # [Boolean] 判断变量是否是敏感信息，防止将密码或者 API token 输出到控制台
  attr_accessor :sensitive
end

```

`command_generator.rb` 文件是用来生成命令的基础文件，这个文件在每一个工具中都有其具体的实现，这一点对后面的理解比较重要。

`configuration.rb` 文件是整个 fastlane 配置中心的基础功能文件，包括了初始化配置项，设置配置项，校验等功能，在后文中会看到，其他工具使用该基础类创建配置。

fastlane_core 文件夹下还有很多其他文件，笔者在此不一一赘述，读者可自行查看。

3 示例分析

3.1 Produce

`produce` 工具是用来在 iTunes Connect 和 开发者后台创建 app 的，用法大约是下面这个样子：

```

lane :release do
  produce(
    username: 'felix@krausefx.com',
    app_identifier: 'com.krausefx.app',
    app_name: 'MyApp',
    language: 'English',
    app_version: '1.0',
    sku: '123',
    team_name: 'SunApps GmbH' # only necessary when in multiple teams
  )
end

```

下面，我们从源码的角度分析一下该工具的流程。打开位于 produce/lib 文件夹下的 `produce.rb` 文件，源码如下：

```

module Produce
  class << self
    attr_accessor :config
  end

  Helper = FastlaneCore::Helper # you gotta love Ruby: Helper.* should use
  the Helper class contained in FastlaneCore
  UI = FastlaneCore::UI
  ROOT = Pathname.new(File.expand_path('../..', __FILE__))

  ENV['FASTLANE_TEAM_ID'] ||= ENV["PRODUCE_TEAM_ID"]
  ENV['DELIVER_USER'] ||= ENV["PRODUCE_USERNAME"]
end

```

由之前的内容可知，每一个具体工具都有一个 `commands_generator.rb` 文件用以真正执行命令，该文件里面的 `run` 函数有这么一段代码：

```

command :create do |c|
  c.syntax = 'fastlane produce create'
  c.description = 'Creates a new app on iTunes Connect and the Apple
Developer Portal'
  c.action do |args, options|
    Produce.config =
FastlaneCore::Configuration.create(Produce::Options.available_options,
options.__hash__)
    puts Produce::Manager.start_producing
  end
end

```

由此可以看出，create 函数首先调用了 `FastlaneCore::Configuration.create` 方法给 Produce 的 config 属性赋值，然后调用 Produce 里 Manager 类的 `start_producing` 方法，在这里，我们可以推测，真正执行创建的就是这个 `start_producing` 方法，点开 `manager.rb` 文件，看到源码之后，你应该能露出灿烂的笑容，因为这和我们之前的推测是一致的。`manager.rb` 源码如下：

```
class Manager
  # Produces app at DeveloperCenter and ItunesConnect
  def self.start_producing
    FastlaneCore::PrintTable.print_values(config: Produce.config,
    hide_keys: [], title: "Summary for produce #{Fastlane::VERSION}")

    Produce::DeveloperCenter.new.run unless
Produce.config[:skip_devcenter]
    return Produce::ItunesConnect.new.run unless Produce.config[:skip_itc]
  end
end
```

这里分别调用了 `DeveloperCenter` 和 `ItunesConnect` 类完成创建 app 的操作。

那么我们再来看一看 `produce` 命令如何关联上 iTunes Connect 和 app Dev Center 的吧，在 `produce` 文件夹下面有 `developer_center.rb` 和 `itunes_connect.rb`，我们以 `developer_center.rb` 为例，`itunes_connect.rb` 的原理类似。

`develop_center.rb` 的核心代码如下：

```

def create_new_app
  ENV["CREATED_NEW_APP_ID"] = Time.now.to_i.to_s
  if app_exists?
    UI.success "[DevCenter] App '#{Produce.config[:app_identifier]}'
already exists, nothing to do on the Dev Center"
    ENV["CREATED_NEW_APP_ID"] = nil
    # Nothing to do here
  else
    app_name = Produce.config[:app_name]
    UI.message "Creating new app '#{app_name}' on the Apple Dev Center"

    app = Spaceship.app.create!(bundle_id: app_identifier,
                                name: app_name,
                                enabled_features: enabled_features,
                                mac: Produce.config[:platform] ==
"osx")

    if app.name != Produce.config[:app_name]
      UI.important("Your app name includes non-ASCII characters, which
are not supported by the Apple Developer Portal.")
      UI.important("To fix this a unique (internal) name '#{app.name}'
has been created for you. Your app's real name '#
{Produce.config[:app_name]}'")
      UI.important("will still show up correctly on iTunes Connect and
the App Store.")
    end

    UI.message "Created app #{app.app_id}"

    UI.crash!("Something went wrong when creating the new app - it's not
listed in the apps list") unless app_exists?

    ENV["CREATED_NEW_APP_ID"] = Time.now.to_i.to_s

    UI.success "Finished creating new app '#{app_name}' on the Dev
Center"
  end

  return true
end

```

由上述代码可知，改函数首先检验 app 在 Dev Center 的后台是否已经存在，如果不存在，则使用 `Spaceship` 工具创建 app，同时，打印一些 app 相关的基本信息。那么我们接下来就看一下 `Spaceship` 这个工具。

3.2 Spaceship

`spaceship` 是一个和 ITC, Apple Dev Center 通讯的工具，他是 fastlane 工具集中不可或缺的重要部分，使你的脚本化工作流前所未有的轻松。

如果没有 `spaceship`，fastlane 工具集使用的都是从苹果网站上抓取的url，而 `spaceship` 提供了优雅的 API 供使用。不使用 `spaceship` 的时候 `sigh` 命令大约要花1分钟的时间，而使用了 `spaceship` 之后，`sigh` 只需要不到5秒钟。`spaceship` 大体上提供了 Dev Center 和 iTC 的常用 API，具体API可以看这两个网址: [Apple Developer Portal API](#), [iTunes Connect API](#)

`spaceship.rb` 的源码如下:

```
module Spaceship
  ROOT = Pathname.new(File.expand_path('../..', __FILE__))

  # Dev Portal
  Certificate = Spaceship::Portal::Certificate
  ProvisioningProfile = Spaceship::Portal::ProvisioningProfile
  Device = Spaceship::Portal::Device
  App = Spaceship::Portal::App
  AppGroup = Spaceship::Portal::AppGroup
  WebsitePush = Spaceship::Portal::WebsitePush
  AppService = Spaceship::Portal::AppService

  # iTunes Connect
  AppVersion = Spaceship::Tunes::AppVersion
  AppSubmission = Spaceship::Tunes::AppSubmission
  Application = Spaceship::Tunes::Application
  Members = Spaceship::Tunes::Members
  Persons = Spaceship::Portal::Persons

  DESCRIPTION = "Ruby library to access the Apple Dev Center and iTunes
Connect".freeze
end
```

从上面的代码可以清晰的看出 `Spaceship` 模块的2大块功能——Dev Center 和 iTunes Connect，在此，笔者不得不感叹 fastlane 代码的优秀，任何一个门外汉都能很清楚的看懂哎。Dev Center 包括了证书，配置文件，设备，App，推送服务等模块。iTunes Connect 则包括了 app 的一些基本信息，比如版本号，成员等。

`spaceship` 文件夹下面有4个文件夹，分别是 `du` 文件夹，`helper` 文件夹，`portal` 文件夹，`tunes` 文件夹，笔者以 `portal` 文件夹为例子分析，`tunes` 的情形读者可以自己看源码学习。

在 `portal` 文件夹中，有一个 `portal_client.rb`，这个文件提供了 `portal` 的很多基础的功能，包括初始化和登录，App 相关的操作(创建，删除等)，网页推送相关功能，App Group 相关功能，Team 相关功能，设备相关功能，证书相关功能，Provisioning Profiles 相关功能。后面的类里都有使用到这个 `client` 的相关功能。

介绍完了 `client`，按照 `spaceship.rb` 里面的顺序，我们先看 `certificate.rb` 核心代码如下:

```

# 通过发起证书签名请求，从而创建证书
# @param csr (OpenSSL::X509::Request) (必要参数)：证书签名请求。通过
`create_certificate_signing_request` 函数获取。
# @param bundle_id (String) (可选参数)：该证书下的 app 标识符，该参数只有在需要创建
推送配置文件的时候才需要，通常情况下可不传

# @example
# 创建一个证书签名请求
# csr, pkey = Spaceship::Certificate.create_certificate_signing_request
#
# 使用该请求创建一个新的发布证书
# Spaceship::Certificate::Production.create!(csr: csr)
# @return (Certificate): The newly created certificate

def create!(csr: nil, bundle_id: nil)
  type = CERTIFICATE_TYPE_IDS.key(self)
  mac = MAC_CERTIFICATE_TYPE_IDS.include? type
  # look up the app_id by the bundle_id

  if bundle_id
    app = Spaceship::App.find(bundle_id)
    raise "Could not find app with bundle id '#{bundle_id}'" unless app
    app_id = app.app_id
  end
  # ensure csr is a OpenSSL::X509::Request
  csr = OpenSSL::X509::Request.new(csr) if csr.kind_of?(String)
  # if this succeeds, we need to save the .cer and the private key in
  keychain access or wherever they go in linux

  response = client.create_certificate!(type, csr.to_pem, app_id, mac)
  # munge the response to make it work for the factory
  response['certificateTypeId'] = response['certificateType']
  ['certificateTypeId']
  self.new(response)
end

```

该函数首先查看是否传入 bundle_id, 然后使用 `OpenSSL::X509::Request` 发起签名请求，最后使用该请求创建证书。

下面我们再看一下 `provisioning_profile.rb`，其实熟悉了 `certificate.rb` 的套路之后，`provisioning_profile.rb` 代码的结构应该也就很清晰了，其实这也反应的优秀的代码应该是什么样子的——代码结构类似，做相同事情的函数名类似。这样别人在看懂一个文件的代码之后可以迅速的理解其他文件的结构。

`provisioning_profile.rb` 文件也是先定义了配置文件相关的一些属性，后面添加了一些相关的操作方法，属性包括：uuid, expires, distribution_method, name, version, platform 等，同样我依然截取一段代码看看是如何创建配置文件的。

```

def create_provisioning_profile!(name, distribution_method, app_id,
certificate_ids, device_ids, mac: false, sub_platform: nil)
  ensure_csrf(Spaceship::ProvisioningProfile) do
    fetch_csrf_token_for_provisioning
  end

  params = {
    teamId: team_id,
    provisioningProfileName: name,
    appIdId: app_id,
    distributionType: distribution_method,
    certificateIds: certificate_ids,
    deviceIds: device_ids
  }
  params[:subPlatform] = sub_platform if sub_platform

  r = request(:post, "account/#{
{platform_slug(mac)}/profile/createProvisioningProfile.action", params)
  parse_response(r, 'provisioningProfile')
end

```

由上述代码可知，首先确保获取到了 csrf，然后使用 teamId, deviceIds 等字段组成的字典，发起创建配置文件的请求，最后解析该请求结果并生成配置文件。

3.3 gym

前面看了这么多"废柴"命令，你丫不是跟我说这是一个自动化打包工具么，怎么没有包？。。。那么，我们就看一下最实在的打包命令吧。

`gym` 是 fastlane 里面的打包工具，让你从繁杂的打包命令中解放出来，轻松地生成 `ipa` 或者 `app` 文件。

Before gym，使用命令行打包可能是这样的：

```

xcodebuild clean archive -archivePath build/MyApp \
                        -scheme MyApp
xcodebuild -exportArchive \
    -exportFormat ipa \
    -archivePath "build/MyApp.xcarchive" \
    -exportPath "build/MyApp.ipa" \
    -exportProvisioningProfile "ProvisioningProfileName"

```

WTF!!!!

With gym, 你只需要

```
fastlane gym
```

泡杯茶吧

`gym` 还有很多优势，详情可参见[官网](#)

这里我们依然关注的是源码。`gym` 的源码主要是在 `runner.rb` 里

```
def run

  unless Gym.config[:skip_build_archive]
    clear_old_files
    build_app
  end

  verify_archive

  FileUtils.mkdir_p(File.expand_path(Gym.config[:output_directory]))

  if Gym.project.ios? || Gym.project.tvos?

    fix_generic_archive # See
    https://github.com/fastlane/fastlane/pull/4325
    package_app
    fix_package
    compress_and_move_dsym
    path = move_ipa
    move_manifest
    move_app_thinning
    move_app_thinning_size_report
    move_apps_folder

  elsif Gym.project.mac?

    path = File.expand_path(Gym.config[:output_directory])
    compress_and_move_dsym

    if Gym.project.mac_app?
      copy_mac_app
      return path
    end

    copy_files_from_path(File.join(BuildCommandGenerator.archive_path,
    "Products/usr/local/bin/*")) if Gym.project.command_line_tool?
  end

  path
end
```

从代码中，我们可以很清楚的看出来这个函数做了哪些事情，得益于 ruby 调用函数可以省略()这个特性，ruby 代码读起来特别像文章一样，这也是 ruby 作为DSL语言的原因之一。

从上面的文章可以看到，主要负责生成 app 的函数是 `build_app`，so 二话不说，我们马上去看一看

```

def build_app
  command = BuildCommandGenerator.generate
  print_command(command, "Generated Build Command") if $verbose
  FastlaneCore::CommandExecutor.execute(command: command,
                                         print_all: true,
                                         print_command: !Gym.config[:silent],
                                         error: proc do |output|

ErrorHandler.handle_build_error(output)

                                         end)

  mark_archive_as_built_by_gym(BuildCommandGenerator.archive_path)
  UI.success "Successfully stored the archive. You can find it in the
Xcode Organizer." unless Gym.config[:archive_path].nil?
  UI.verbose("Stored the archive in: " +
BuildCommandGenerator.archive_path)
end

```

由上述代码可见，`build_app` 的主要实现就是先生成一个 `command`，然后调用 `fastlaneCore` 里面的 `CommandExector` 执行该命令。

故事到这里似乎还没有结束，也许你会问，`gym` 到底是如何生成 `ipa` 的咩？这里就不得不提到 `xcodebuild` 和 `xcrun` 了。

`xcodebuild` 命令是用来编译 `xcode` 工程的，用来生成 `app` 文件。具体用法在此不再赘述，文档已经说的很清楚了。

`xcrun` 提供了一种在命令行调用 `develop tool` 的方式。比如用以生成 `ipa` 文件。

在 `gym` 文件夹里面，有一个 `generators` 的文件夹，这个就是用来生成 `ipa` 的。其中，真正实现打包功能的是 `package_command_generator_legacy.rb` 和 `package_command_generator_xcode7.rb` 这两个文件。`package_command_generator.rb` 可以算是这两个文件的抽象类。

从文件名也可以看出，一个是负责 `Xcode 7` 之前的打包，一个是负责 `Xcode 7` 之后的打包，我们本着『有新用新』的原则，就看一下 `package_command_generator_xcode7.rb` 的核心代码。

```
def generate
  print_legacy_information

  parts = ["/usr/bin/xcrun #
{XcodebuildFixes.wrap_xcodebuild.shellescape} -exportArchive"]
  parts += options
  parts += pipe

  File.write(config_path, config_content) # overwrite everytime. Could
  be optimized

  parts

end
```

至此，已经真相大白，gym 的底层，还是调用了 `xcrun` 实现生成 ipa 的功能，然后将 ipa 文件输出到指定的目录下。接下来，我们再看一个也很有用的工具 `sigh`

3.4 sigh

还记得你被 provisioning profiles，弄得心烦意乱的场景么？(说没有的我都不信的~)，现在有了 `sigh`，世界也许会变得美好一些。

`sigh` 可以只用一行命令，就完成创建，更新，下载，配对 provisioning profiles 的任务，它支持 App Store, AdHoc, Development 和 Enterprise 模式。她具有以下的特色功能

- 下载最新的配置文件
- 更新配置文件
- 修复配置文件
- 创建配置文件
- 支持的开发模式多样
- 支持多账户和多Team

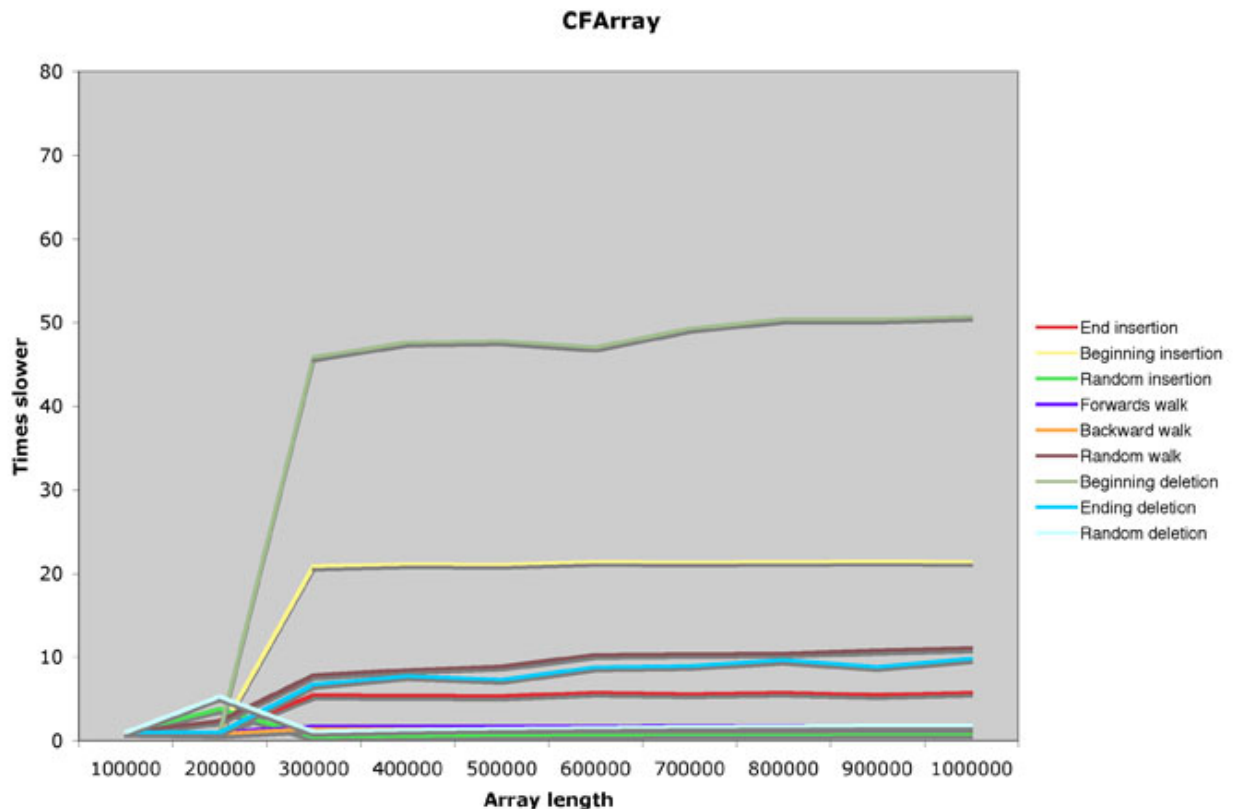
有了前面的经验，加上 fastlane 代码的优雅，这次我们可以直奔主题了，`sigh` 文件夹下的 `runner.rb` 文件定义了关于配置文件的一些基本操作，包括判断配置文件类型，获取配置文件，创建配置文件等。

3.5 Match

`match` 提供了全新的一套管理证书和配置文件的方式，让你不必再为多人开发中证书和配置文件的问题再烦恼。`match` 是依据 <https://codesigning.guide> 的思路实现管理的，大致是使用 git 仓库去管理每一个项目的证书和配置文件，具体方式大家可以去前面的网站查阅。`match` 文件夹中 `runner.rb` 代码主要做了如下的事情：使用 `GitHelper` 模块 clone 证书仓库，然后使用 `Spaceship` 工具对证书和配置文件做一些基本的校验(比如 bundleID和证书是否存在)，最后再使用 `GitHelper` 模块生成提交记录并执行提交操作。具体代码读者可参阅源码学习。

总结

到此介绍了一些 fastlane 的常用工具的源码，笔者的目的也只是希望打开开发者的另一扇门，不只是停留在使用层面，这样，有时需要自己写一个小工具的时候，能够有所启发就很 nice 了，由于 fastlane 的源码非常多，在此也不可能面面俱到，有兴趣的读者可以自己下载源码，研究学习。



上图分析可以看出，`CFArray` 在头插、尾插插入时候的效率近乎常数，而对于中间元素的操作会从小数据的线性效率在一个阈值上突然转变成线性效率，而这个跃变不由得想起在 Java 8 当中的 `HashMap` 的数据结构转变方式。

在 ObjC 的初期，`CFArray` 是使用 *deque* 双端队列实现，所以会呈现出头尾操作高效，而中间操作成线性的特点。在容量超过 300000 左右时（实际应该是 $262140 = 2^{18}$ ），时间复杂度发生陡变。在源代码中，阈值被宏定义为 `__CF_MAX_BUCKETS_PER_DEQUE`，具体代码可以见 [CF-550-CFArray.c](#)（2011 年版本）：

```
if (__CF_MAX_BUCKETS_PER_DEQUE <= futureCnt) {  
    // 创建 CFStorage 引用  
    CFStorageRef store  
    // 转换 CFArray 为 Storage  
    __CFArrayConvertDequeToStore(array);  
    store = (CFStorageRef)array->_store;  
}
```

可以看到，当数据超出阈值 `__CF_MAX_BUCKETS_PER_DEQUE` 的时候，会将数据结构从 `CFArray` 转换成 `CFStorage`。`CFStorage` 是一个平衡二叉树的结构，为了维护数组的顺序访问，将 Node 的权值使用下标完成插入和旋转操作。具体的体现可以看 `CFStorageInsertValues` 操作。具体代码可以查看 [CF-368.18-CFStorage.c](#)。

在 2011 年以后的 [CF-635.15-CFArray.c](#) 版本中，`CFArray` 取消了数据结构转换这一功能。或许是为了防止大数据时候二叉树建树的时间抖动问题从而取消了这一特性。直接来看下数据结构的描述：

```
struct __CFArrayDeque {
    uintptr_t _leftIdx; // 自左开始下标位置
    uintptr_t _capacity; // 当前容量
};

struct __CFArray {
    CFRuntimeBase _base;
    CFIndex _count; // 元素个数
    CFIndex _mutations; // 元素抖动量
    int32_t _mutInProgress;
    __strong void *_store;
};
```

从命名上可以看出 `CFArray` 由单一的双端队列进行实现，而且记录了一些容器信息。

C 数组的一些问题

C 语言中的数组，会开辟一段连续的内存空间来进行数据的读写、存储操作。另外说一句，**数组和指针并不相同**。有一种被很多教材书籍上滥用的说法：一块被 `malloc` 过的内存空间等于一个数组。这是错误的。最简单的解释，指针需要申请一个指针区域来存储（指向）一块空间的起始位置，而数组（的头部）是对一块空间起始位置的直接访问。另外想了解更多可以看 [Are pointers and arrays equivalent in C?](#) 这篇博文。

C 中的数组最显著的缺点就是，在下标 0 处插入时，需要移动所有的元素（即 `memmove()` 函数的原理）。类似的，当删除第一个元素、在第一个元素前插入一个元素也会造成 **O(n)** 复杂度的操作。然而数组是常读写的容器，所以 O(n) 的操作会造成很严重的时间开销。

当前版本中 CFArray 的部分实现细节

在 [CF-855.17](#) 中，我们可以看到当前版本的 `CFArray` 的实现。文档中对 `CFArray` 有如下的描述：

`CFArray` 实现了一个可被指针顺序访问的紧凑容器。其值可通过整数键（索引下标）进行访问，范围从 0 至 N-1，其中 N 是数组中值的数量。称其**紧凑 (compact)** 的原因是该容器进行删除或插入某个值的时候，不会再内存空间中留下间隙，访问顺序仍旧按照原有键值数值大小排列，使得有效检索集合范围总是在整数范围 [0, N-1] 之中。因此，特定值的下标可能会随着其他元素插入至数组或被删除时而改变。

数组有两种类型：**不可变 (immutable)** 类型在创建数组之后，不能向其添加或删除元素，而 **可变 (mutable)** 类型可以添加或从中删除元素。可变数组的元素数量无限制（或者称只受 `CFArray` 外部的约束限制，例如可用内存空间大小）。与所有的 CoreFoundation 集合类型同理，数组将保持与元素对象的强引用关系。

为了进一步弄清 `CFArray` 的细节，我们来分析一下 `CFArray` 的几个操作方法：

```

// 通过下标查询元素值
const void *CFArrayGetValueAtIndex(CFArrayRef array, CFIndex idx) {
    // 这个函数尚未开源
    // 通过给定的 CFTypeID 来验证指定元素是否匹配 Core Foundation 桥接类
    CF_OBJC_FUNCDISPATCHV(__kCFArrayTypeID, const void *, (NSArray *)array,
objectAtIndex:idx);
    // 尚未开源
    // 通过给定的 CFTypeID 来验证 Core Foundation 类型合法性
    __CFGenericValidateType(array, __kCFArrayTypeID);
    CFAssert2(0 <= idx && idx < __CFArrayGetCount(array), __kCFLogAssertion,
"%s(): index (%d) out of bounds", __PRETTY_FUNCTION__, idx);
    CHECK_FOR_MUTATION(array);
    // 从内存位置取出元素
    return __CFArrayGetBucketAtIndex(array, idx)->_item;
}

// 返回查询元素的地址
CF_INLINE struct __CFArrayBucket *__CFArrayGetBucketAtIndex(CFArrayRef
array, CFIndex idx) {
    switch (__CFArrayGetType(array)) {
        // 只允许两种数组类型
        // 不可变对应普通线性结构，可变对应双端队列
        case __kCFArrayImmutable:
        case __kCFArrayDeque:
            // 取地址再加上索引偏移量，返回元素地址
            return __CFArrayGetBucketsPtr(array) + idx;
        }
        return NULL;
    }
}

```

通过索引下标查询操作中，`CFArray` 仍然继承了传统数组的连续地址空间的性质，所以其时间仍然可保持在 $O(1)$ 复杂度，十分高效。

```

void CFArrayInsertValueAtIndex(CFMutableArrayRef array, CFIndex idx, const
void *value) {
    // 通过给定的 CFTypeID 来验证指定元素是否匹配 Core Foundation 桥接
    CF_OBJC_FUNCDISPATCHV(__kCFArrayTypeID, void, (NSMutableArray *)array,
insertObject:(id)value atIndex:(NSUInteger)idx);
    // 通过给定的 CFTypeID 来验证 Core Foundation 类型合法性
    __CFGenericValidateType(array, __kCFArrayTypeID);
    CFAssert1(__CFArrayGetType(array) != __kCFArrayImmutable,
__kCFLogAssertion, "%s(): array is immutable", __PRETTY_FUNCTION__);
    CFAssert2(0 <= idx && idx <= __CFArrayGetCount(array),
__kCFLogAssertion, "%s(): index (%d) out of bounds", __PRETTY_FUNCTION__,
idx);
    // 类型检查
    CHECK_FOR_MUTATION(array);
    // 调用该函数进行具体的数组变动过程
}

```

```

    __CFArrayReplaceValues(array, CFRangeMake(idx, 0), &value, 1);
}

// 这个函数没有经过 objc 的调度检查, 即 CF_OBJC_FUNCDISPATCHV 方法
// 所以为安全考虑, 只能用在已经进行调度检查的函数入口之后
void __CFArrayReplaceValues(CFMutableArrayRef array, CFRange range, const
void **newValues, CFIndex newCount) {
    // 进一步类型检查
    CHECK_FOR_MUTATION(array);
    // 加锁操作, 增加自旋锁防止竞争
    BEGIN_MUTATION(array);
    // 声明回调
    const CFArrayCallbacks *cb;
    // 偏移下标, 元素总数, 数组改变后元素总数
    CFIndex idx, cnt, futureCnt;
    const void **newv, *buffer[256];
    // 获取数组中元素个数
    cnt = __CFArrayGetCount(array);
    // 新数组元素总数 = 原数组元素总数 - 删除的元素个数 + 增加的元素个数
    futureCnt = cnt - range.length + newCount;
    CFAssert1(newCount <= futureCnt, __kCFLogAssertion, "%s(): internal
error 1", __PRETTY_FUNCTION__);
    // 获取数组中定义的回调方法
    cb = __CFArrayGetCallbacks(array);
    // 构造分配释放内存抽象
    CFAllocatorRef allocator = __CFGetAllocator(array);
    // 需要的情况下持有新元素, 并为其分配一个临时缓冲区
    // 标准是新元素的个数是否超过256
    if (NULL != cb->retain && !hasBeenFinalized(array)) {
        newv = (newCount <= 256) ? (const void **)buffer : (const void
**)CFAllocatorAllocate(kCFAllocatorSystemDefault, newCount * sizeof(void *),
0);
        if (newv != buffer && __CFOASafe) __CFSetLastAllocationEventName(newv,
"CFArray (temp)");
        // 为新元素增加数据缓冲区
        for (idx = 0; idx < newCount; idx++) {
            newv[idx] = (void *)INVOKE_CALLBACK2(cb->retain, allocator,
(void *)newValues[idx]);
        }
    } else {
        newv = newValues;
    }
    // 数据抖动量自加
    array->_mutations++;
    // 现在将一个数组的存储区域分成了三个部分, 每个部分都有可能为空
    // A: 从索引下标零的位置到小于 range.location 的区域
    // B: 传入的 range.location 区域
    // C: 从 range.location + range.length 到数组末尾

```

// 需要注意的是，索引0的位置不一定位于可用存储的最低位，当变化位置新值数量与旧值数量不同时，B区域需要先释放再替换，然后A和C中的值根据情况进行位移

```
if (0 < range.length) {
    // 正常释放变化区域操作
    __CFArrayReleaseValues(array, range, false);
}
// B 区现在为清空状态，需要重新填充数据
if (0) {
    // 此处隐藏了判断条件和代码。
    // 大概操作是排除其他的干扰项，例如 B 区数据未完全释放等。
} else if (NULL == array->_store) {
    // 通过数据的首地址引用指针来判断 B 区释放
    if (0) {
        // 此处隐藏了判断条件和代码
        // 排除干扰条件，例如 futureCnt 不合法等
    } else if (0 <= futureCnt) {
        // 声明一个双端队列对象
        struct __CFArrayDeque *deque;
        // 根据元素总数确定环状缓冲区域可载元素总个数
        CFIndex capacity = __CFArrayDequeRoundUpCapacity(futureCnt);
        // 根据元素个数确定空间分配大小
        CFIndex size = sizeof(struct __CFArrayDeque) + capacity *
sizeof(struct __CFArrayBucket);
        // 通过缓冲区构造器来构造存储缓存
        deque = (struct __CFArrayDeque
*)CFAllocatorAllocate((allocator), size, isStrongMemory(array) ?
__kCFAllocatorGCScannedMemory : 0);
        if (__CFOASafe) __CFSetLastAllocationEventName(deque, "CFArray
(store-deque)");
        // 确定双端队列左值
        deque->_leftIdx = (capacity - newCount) / 2;
        deque->_capacity = capacity;
        __CFAssignWithWriteBarrier((void **)&array->_store, (void
*)deque);
        // 完成 B 区构造，安全释放数组
        if (CF_IS_COLLECTABLE_ALLOCATOR(allocator))
auto_zone_release(objc_collectableZone(), deque);
    }
} else {
    // Deque
    // 根据 B 区元素变化，重新定位 A 和 C 区元素存储状态
    if (0) {
    } else if (range.length != newCount) {
        // 传入 array 引用，最终根据变化使得数组更新A、B、C分区规则
        __CFArrayRepositionDequeRegions(array, range, newCount);
    }
}
// 将区域B的新变化拷贝到B区域
if (0 < newCount) {
    if (0) {
```

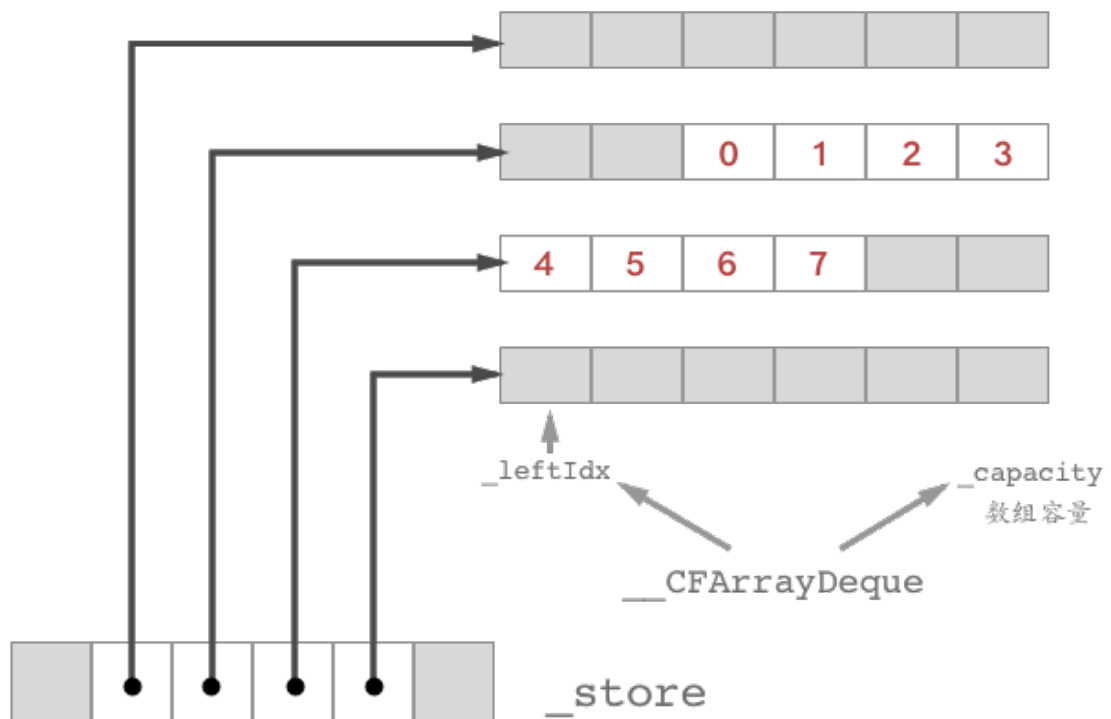
```

        } else { // Deque
            // 访问线性存储区
            struct __CFArrayDeque *deque = (struct __CFArrayDeque *)array->_store;

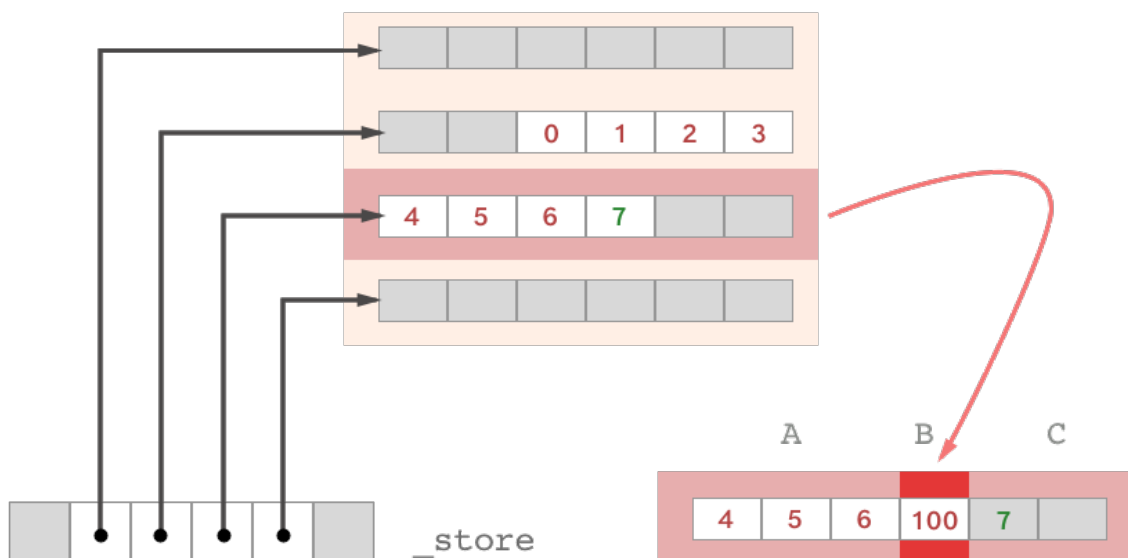
            // 在原基础上, 增加一段缓存区域
            struct __CFArrayBucket *raw_buckets = (struct __CFArrayBucket *)
            ((uint8_t *)deque + sizeof(struct __CFArrayDeque));
            // 更改B区域数据, 类似与 memcpy, 但是有写屏障(write barrier), 线程安全
            objc_memmove_collectable(raw_buckets + deque->_leftIdx +
            range.location, newv, newCount * sizeof(struct __CFArrayBucket));
        }
    }
    // 设置新的元素个数属性
    __CFArraySetCount(array, futureCnt);
    // 释放缓存区域
    if (newv != buffer && newv != newValues)
        CFAllocatorDeallocate(kCFAllocatorSystemDefault, newv);
    // 解除线程安全保护
    END_MUTATION(array);
}

```

在 `CFArray` 的插入元素操作中, 可以很清楚的看出这是一个双端队列(dequeue)的插入元素操作, 而且是一种仿照 C++ STL 标准库的存储方式, 缓冲区嵌套 map 表的静态实现。用示意图来说明一下数据结构:



在 STL 中的 deque，是使用的 map 表来记录的映射关系，而在 Core Foundation 中，`CFArray` 在保证这样的二次映射关系的时候很直接地运用了二阶指针 `_store`。在修改元素的操作中，`CFArray` 也略显暴力一些，先对数组进行大块的分区操作，再按照顺序填充数据，组合成一块新的双端队列，例如在上图中的双端队列中，在下标为 7 的元素之前增加一个值为 `100` 的元素：



根据索引下标会找到指定部分的缓存区，将其拿出并进行重新构造。构造过程中或将其划分成 A、B、C 三个区域，B 区域是修改部分。当然如果不够的话，系统会自己进行缓存区的扩容，即 `CFAllocatorRef` 官方提供的内存分配/释放策略。

`CFAllocatorRef` 是 Core Foundation 中的分配和释放内存的策略。多数情况下，只需要用默认分配器 `kCFAllocatorDefault`，等价于传入 `NULL` 参数，这样会用 Core Foundation 所谓的“常规方法”来分配和释放内存。这种方法可能会有变化。用到特殊分配器的情况很少，下列是官方文档中给出的标准分配器及其功能。

<code>kCFAllocatorDefault</code>	默认分配器，与传入 <code>NULL</code> 等价。
<code>kCFAllocatorSystemDefault</code>	原始的默认系统分配器。这个分配器用来应对万一用 <code>CFAllocatorSetDefault</code> 改变了默认分配器的情况，很少用到。
<code>kCFAllocatorMalloc</code>	调用 <code>malloc</code> 、 <code>realloc</code> 和 <code>free</code> 。如果用 <code>malloc</code> 创建了内存，那这个分配器对于释放 <code>CFData</code> 和 <code>CFString</code> 就很有用。
<code>kCFAllocatorMallocZone</code>	在默认的 <code>malloc</code> 区域中创建和释放内存。在 Mac 上开启了垃圾收集的话，这个分配器会很有用，但在 iOS 中基本上没什么用。
<code>kCFAllocatorNull</code>	什么都不做。跟 <code>kCFAllocatorMalloc</code> 一样，如果不想释放内存，这个分配器对于释放 <code>CFData</code> 和 <code>CFString</code> 就很有用。
<code>KCFAllocatorUseContext</code>	只有 <code>CFAllocatorCreate</code> 函数用到。创建 <code>CFAllocator</code> 时，系统需要分配内存。就像其他所有的 <code>Create</code> 方法，也需要一个分配器。这个特殊的分配器告诉 <code>CFAllocatorCreate</code> 用传入的函数来分配 <code>CFAllocator</code> 。

在 `_CFArrayReplaceValues` 方法中的最后一个判断：

```
if (newv != buffer && newv != newValues)
    CFAllocatorDeallocate(kCFAllocatorSystemDefault, newv);
```

会检查一下缓存区的数量问题，如果数量过多会释放掉多余的缓存区。这是因为这个方法具有通用性，不仅仅可以使用在插入元素操作，在增加(`CFArrayAppendValue`)、替换(`CFArrayReplaceValues`)、删除(`CFArrayRemoveValueAtIndex`)操作均可使用。由于将数据结构采取分块管理，所以时间分摊，复杂度大幅度降低。所以，我们看到 `CFArray` 的时间复杂度在查询、增添元素操作中均有较高的水平。

而在 `NSMutableArray` 的实现中，苹果为了解决移动端的小内存特点，使用 `CFArray` 中在两端增加可扩充的缓存区则会造成大量的浪费。在 [NSMutableArray原理揭露](#) 一文中使用逆向的思路，挖掘 `NSMutableArray` 的实现原理，其做法是使用环形缓冲区对缓存部分做到最大化的压缩，这是苹果针对于移动设备的局限而提出的方案。

参考链接

[Let's Build NSMutableArray](#)

[GNUStep · NSArray](#)

[What is the data structure behind NSMutableArray?](#)

[Apple Source Code - CF-855.17](#)

load 方法全程跟踪

Github 的 [RetVal](#) 大神在16年11月更新了可 debug 版本的 706 `<objc/runtime.h>` 源码，于是让源码阅读学习得以继续。本文将介绍个人学习 `load` 方法的全部流程。

load 方法的调用时机

从 *Effective Objective-C 2.0 - 52 Specific Ways to Improve Your iOS and OS X Programs* 一书中讲述到：Objective-C 中绝大多数类都继承自 `NSObject` 根类，每个类都有两个初始化方法，其中之一就是 `load` 方法。

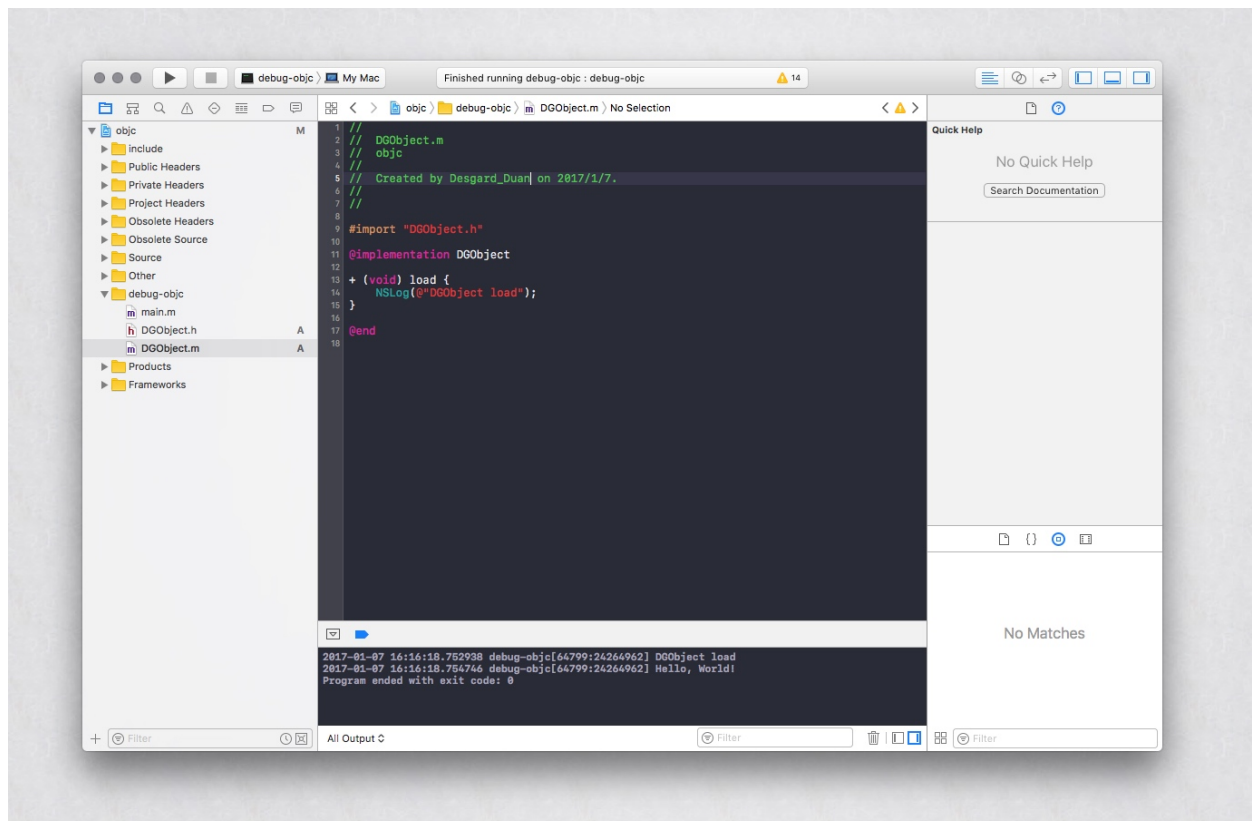
```
+ (void)load
```

对于每一个 *Class* 和 *Category* 来说，必定会调用此方法，而且仅调用一次。当包含 *Class* 和 *Category* 的程序库载入系统时，就会执行此方法，并且此过程通常是在程序启动的时候执行。

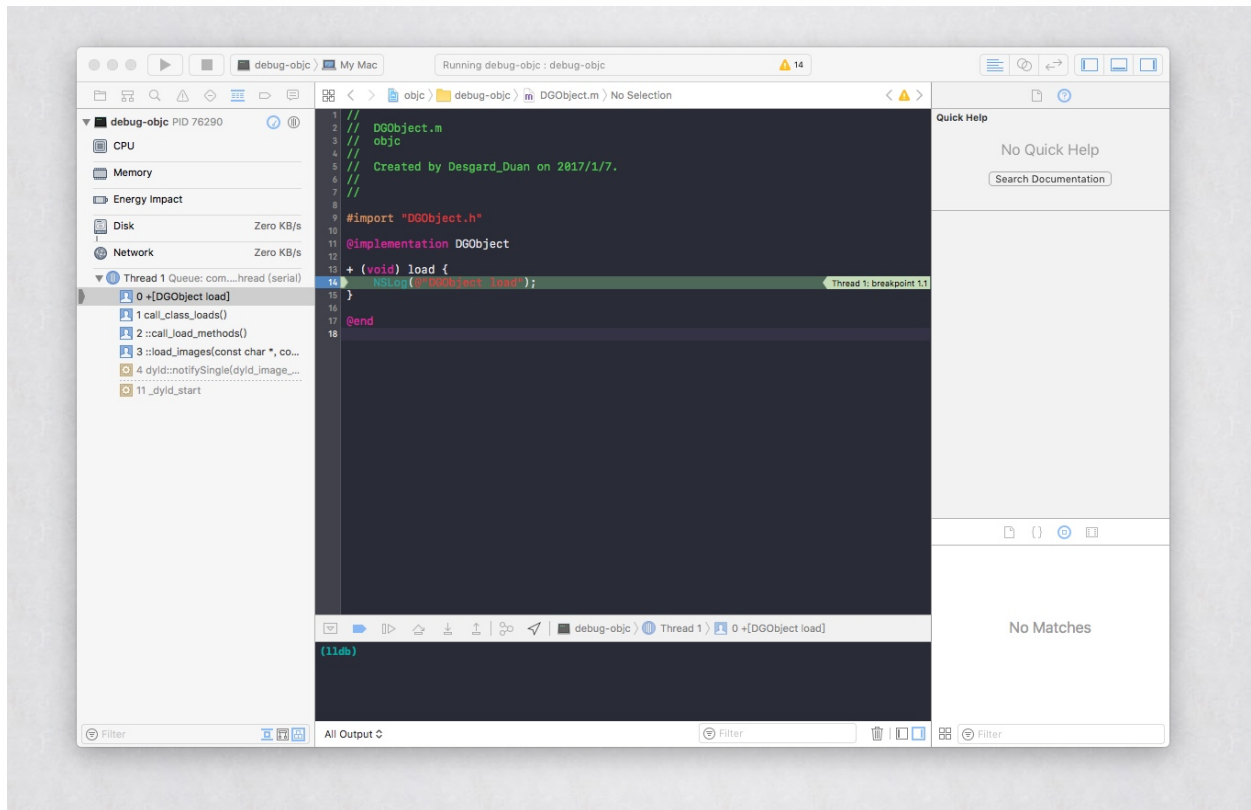
不同的是，现在的 iOS 系统中已经加入了**动态加载特性（Dynamic Loading）**，这是从 macOS 应用程序中迁移而来的特性，等应用程序启动好之后再去加载程序库。如果 *Class* 和其 *Category* 中都重写了 `load` 方法，则先调用 *Class* 中的。

我们通过 [RetVal](#) 封装好的 debug 版最新源码进行断点调试，来追踪一下 `load` 方法的全部处理过程，以便于了解这个函数以及 Objective-C 强大的动态性。

创建一个 *Class* 文件 `DGObject.m`，然后在其中增加 `load` 方法。在运行 proj 后，可以看见 `load` 方法的调用时机是在入口函数主程序之前。



下面在 `load` 方法下增加断点，查看其调用栈并跟踪函数执行时候的上层代码：



调用栈显示栈情况为如下方法对象：

```
0  +[XXObject load]
1  call_class_loads()
2  call_load_methods
3  load_images
4  dyld::notifySingle(dyld_image_states, ImageLoader const*)
11 _dyld_start
```

追其源头，从 `_dyld_start` 开始探究。**dyld(The Dynamic Link Editor)**是 Apple 的动态链接库，系统内核做好启动程序的初始准备后，将其他事务交给 dyld 负责。对于 dyld 这里不再细究，在以后对于动态库的学习时进行研究。

在研究 `load_images` 方法之前，先来研究一下什么是 **images**。**images**表示的是二进制文件（可执行文件或者动态链接库.so文件）编译后的符号、代码等。所以 `load_images` 的工作是传入处理过的二进制文件并让 **Runtime** 进行处理，并且每一个文件对应一个抽象实例来负责加载，这里的实例是 `ImageLoader`，我们从调用栈的方法 4 可以清楚的看到参数类型：

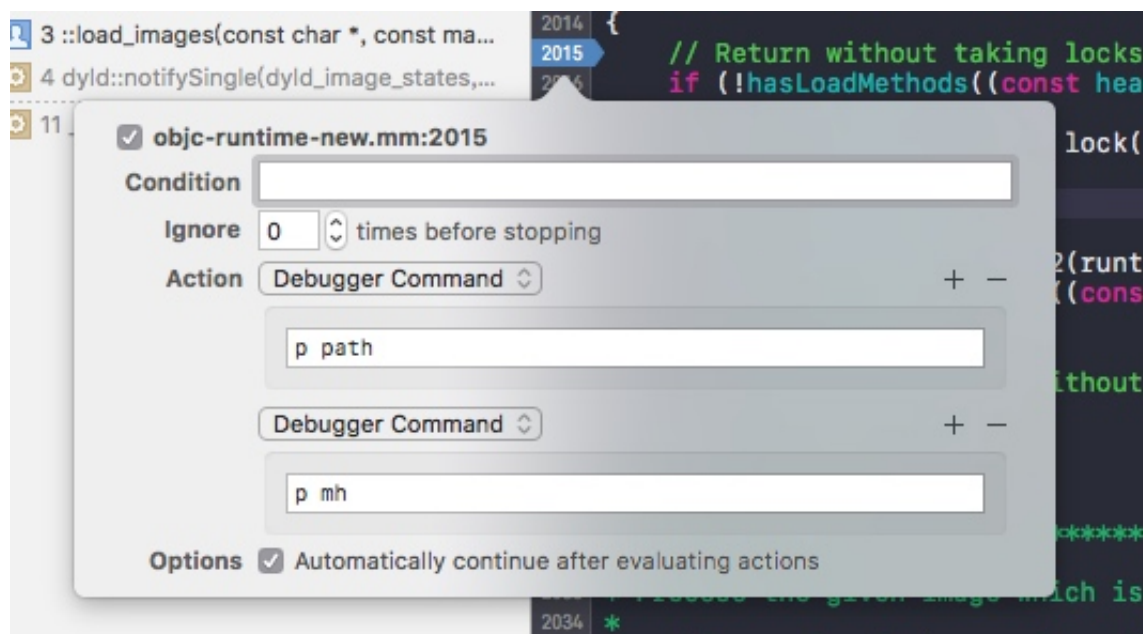
```
dyld::notifySingle(dyld_image_states, ImageLoader const*)
```

`ImageLoader` 处理二进制文件的时机是在 `main` 入口函数以前，它在加载文件时主要做两个工作：

- 在程序运行时它先将动态链接的 image 递归加载（也就是上面测试栈中一串的递归调用的时刻）
- 再从可执行文件 image 递归加载所有符号

简单了解 image

在 [你真的了解 load 方法么?](#) 这篇文章中, Draveness 提供了一种断点发来打印出所有加载的镜像。



这样可以将当前载入的 image 全部显示, 我们展示的是 image 的 *path* 和 *slice* 信息。

```
...
(const char *) $22 = 0x00007fff9c1f07a0
"/System/Library/Frameworks/CoreServices.framework/Versions/A/Frameworks/Dic
tionaryServices.framework/Versions/A/DictionaryServices"
(const mach_header *) $23 = 0x00007fff9c1f0000
(const char *) $24 = 0x00007fff9c51bb10
"/System/Library/Frameworks/CoreServices.framework/Versions/A/Frameworks/Sha
redFileList.framework/Versions/A/SharedFileList"
(const mach_header *) $25 = 0x00007fff9c51b000
(const char *) $26 = 0x00007fff9ca70d90
"/System/Library/Frameworks/Foundation.framework/Versions/C/Foundation"
(const mach_header *) $27 = 0x00007fff9ca70000
(const char *) $28 = 0x00007fff5fbff870
"/Users/Desgard_Duan/Library/Developer/Xcode/DerivedData/objc-
frsvxngqnjxvxwahvxtwjglbkjlt/Build/Products/Debug/debug-objc"
(const mach_header *) $29 = 0x0000000010000000
```

这里会传入很多的动态链接库 `.dylib` 以及官方静态框架 `.framework` 的 image, 而 *path* 就是其对应的二进制文件的地址。在 `<mach-o/dyld.h>` 动态库头文件中, 也为我们提供了查询所有动态库 image 的方法, 在这里也简单介绍一下:

```

#include <mach-o/dyld.h>
#include <stdio.h>

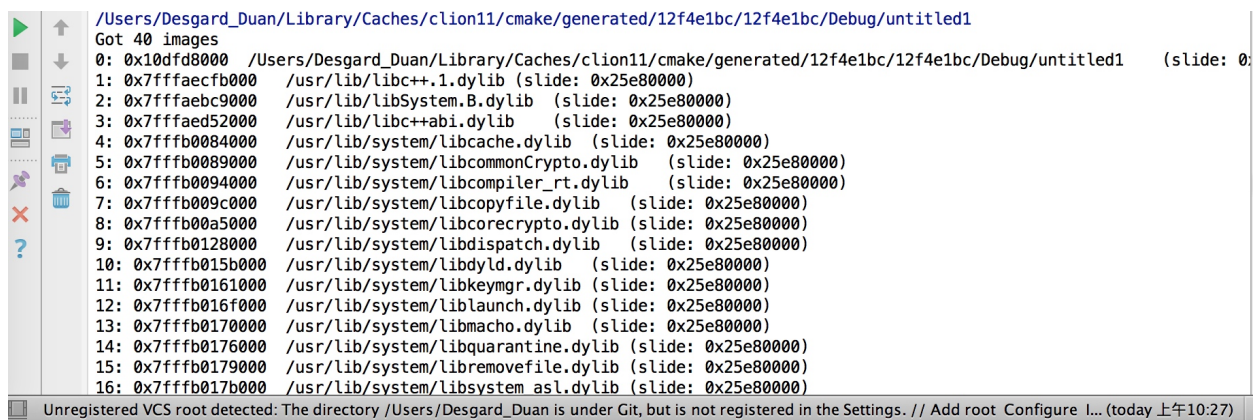
void listImages(){
    uint32_t i;
    uint32_t ic = _dyld_image_count();

    printf("Got %d images\n", ic);
    for (i = 0; i < ic; ++ i) {
        printf("%d: %p\t%s\t(slide: %p)\n",
            i,
            _dyld_get_image_header(i),
            _dyld_get_image_name(i),
            _dyld_get_image_vmaddr_slide(i));
    }
}

int main() {
    listImages();
    return 0;
}

```

我们可以通过系统库提供的接口方法，来深入学习官方的动态库情况。



```

/Users/Desgard_Duan/Library/Caches/clion11/cmake/generated/12f4e1bc/12f4e1bc/Debug/untitled1
Got 40 images
0: 0x10dfd8000 /Users/Desgard_Duan/Library/Caches/clion11/cmake/generated/12f4e1bc/12f4e1bc/Debug/untitled1 (slide: 0x25e80000)
1: 0x7fffaecfb000 /usr/lib/libc++.1.dylib (slide: 0x25e80000)
2: 0x7fffaebc9000 /usr/lib/libSystem.B.dylib (slide: 0x25e80000)
3: 0x7fffaed52000 /usr/lib/libc++abi.dylib (slide: 0x25e80000)
4: 0x7fffb0084000 /usr/lib/system/libcache.dylib (slide: 0x25e80000)
5: 0x7fffb0089000 /usr/lib/system/libcommonCrypto.dylib (slide: 0x25e80000)
6: 0x7fffb0094000 /usr/lib/system/libcompiler_rt.dylib (slide: 0x25e80000)
7: 0x7fffb009c000 /usr/lib/system/libcopyfile.dylib (slide: 0x25e80000)
8: 0x7fffb00a5000 /usr/lib/system/libcorecrypto.dylib (slide: 0x25e80000)
9: 0x7fffb0128000 /usr/lib/system/libdispatch.dylib (slide: 0x25e80000)
10: 0x7fffb015b000 /usr/lib/system/libdyld.dylib (slide: 0x25e80000)
11: 0x7fffb0161000 /usr/lib/system/libkeymgr.dylib (slide: 0x25e80000)
12: 0x7fffb016f000 /usr/lib/system/liblaunch.dylib (slide: 0x25e80000)
13: 0x7fffb0170000 /usr/lib/system/libmacho.dylib (slide: 0x25e80000)
14: 0x7fffb0176000 /usr/lib/system/libquarantine.dylib (slide: 0x25e80000)
15: 0x7fffb0179000 /usr/lib/system/libremovefile.dylib (slide: 0x25e80000)
16: 0x7fffb017b000 /usr/lib/system/libsystem_asl.dylib (slide: 0x25e80000)

```

Unregistered VCS root detected: The directory /Users/Desgard_Duan is under Git, but is not registered in the Settings. // Add root Configure I... (today 上午10:27)

继续研究 load_images

```

// load_images
// 执行 dyld 提供的并且已被 map_images 处理后的 image 中的 +load
// 锁定状态: runtimeLock写操作和 loadMethodLock 方法, 保证线程安全
extern bool hasLoadMethods(const headerType *mhdr);
extern void prepare_load_methods(const headerType *mhdr);
void
load_images(const char *path __unused, const struct mach_header *mh) {
    // 没有查询到传入 Class 中的 load 方法, 视为锁定状态
    // 则无需给其加载权限, 直接返回
    if (!hasLoadMethods((const headerType *)mh)) return;

    // 定义可递归锁对象
    // 由于 load_images 方法由 dyld 进行回调, 所以数据需上锁才能保证线程安全
    // 为了防止多次加锁造成的死锁情况, 使用可递归锁解决
    recursive_mutex_locker_t lock(loadMethodLock);

    // 收集所有的 +load 方法
    {
        // 对 Darwin 提供的线程写锁的封装类
        rwlock_writer_t lock2(runtimeLock);
        // 提前准备好满足 +load 方法调用条件的 Class
        prepare_load_methods((const headerType *)mh);
    }

    // 调用 +load 方法 (without runtimeLock - re-entrant)
    call_load_methods();
}

```

重新回到 `load_images` 方法, `hasLoadMethods` 函数引起注意。其中为了查询 `load` 函数列表, 会分别查询该函数在内存数据段上指定 section 区域是否有所记录。

```

// 快速查询是否存在 +load 函数列表
bool hasLoadMethods(const headerType *mhdr) {
    size_t count;
    if (_getObjc2NonlazyClassList(mhdr, &count) && count > 0) return true;
    if (_getObjc2NonlazyCategoryList(mhdr, &count) && count > 0) return
true;
    return false;
}

```

在 `objc-file.mm` 文件中存有如下定义:

```

// 类似于 c++ 的模板写法，通过宏来处理泛型操作
// 函数内容是从内存数据段的某个区下查询该位置的情况，并回传指针
#define GETSECT(name, type, sectname) \
    type *name(const headerType *mhdr, size_t *outCount) { \
        return getDataSection<type>(mhdr, sectname, nil, outCount); \
    } \
    type *name(const header_info *hi, size_t *outCount) { \
        return getDataSection<type>(hi->mhdr(), sectname, nil, outCount); \
    }

// 根据 dyld 对 images 的解析来在特定区域查询内存
GETSECT(_getObjc2ClassList,          classref_t,      "__objc_classlist");
GETSECT(_getObjc2NonlazyCategoryList, category_t *,   "__objc_nlcatlist");

```

在 Apple 的官方文档中，我们可以在 `__DATA` 段中查询到 `__objc_classlist` 的用途，主要是用在访问 **Objective-C** 的类列表，而 `__objc_nlcatlist` 用于访问 **Objective-C** 的 `+load` 函数列表，比 `__mod_init_func` 更早被执行。这一块对类信息的解析是由 dyld 处理时期完成的，也就是我们上文提到的 `map_images` 方法的解析工作。而且从侧面可以看出，Objective-C 的强大动态性，与 dyld 前期处理密不可分。

可递归锁

在 `load_images` 方法所在的 `objc-runtime-new.mm` 中，全局 `loadMethodLock` 是一个 `recursive_mutex_t` 类型的变量。这个是苹果公司通过 C 实现的一个互斥递归锁 Class，来解决多次上锁而不会发生死锁的问题。

其作用与 `NSRecursiveLock` 相同，但不是由 `NSLock` 再封装，而是通过 C 为 Runtime 的使用场景而写的一个 Class。更多关于线程锁的知识，强烈推荐 [bestswifter](#) 这篇博文 [深入理解 iOS 开发中的锁](#)。

准备 +load 运行的从属 Class

```

void prepare_load_methods(const headerType *mhdr) {
    size_t count, i;

    runtimeLock.assertWriting();

    // 收集 Class 中的 +load 方法
    // 获取所有的类的列表
    classref_t *classlist =
        _getObjc2NonlazyClassList(mhdr, &count);
    for (i = 0; i < count; i++) {
        // 通过 remapClass 获取类指针
        // schedul_class_load 递归到父类逐层载入
        schedule_class_load(remapClass(classlist[i]));
    }

    // 收集 Category 中的 +load 方法
    category_t **categorylist = _getObjc2NonlazyCategoryList(mhdr, &count);
    for (i = 0; i < count; i++) {
        category_t *cat = categorylist[i];
        // 通过 remapClass 获取 Category 对象存有的 Class 对象
        Class cls = remapClass(cat->cls);
        if (!cls) continue;
        // 对类进行第一次初始化，主要用来分配可读写数据空间并返回真正的类结构
        realizeClass(cls);
        assert(cls->ISA()->isRealized());
        // 将需要执行 load 的 Category 添加到一个全局列表中
        add_category_to_loadable_list(cat);
    }
}

```

`prepare_load_methods` 作用是为 load 方法做准备，从代码中可以看出 Class 的 load 方法是优先于 Category。其中在收集 Class 的 load 方法中，因为需要对 Class 关系树的根节点逐层遍历运行，在 `schedul_class_load` 方法中使用深层递归的方式递归到根节点，优先进行收集。

```

// 用来规划执行 Class 的 load 方法，包括父类
// 递归调用 +load 方法通过 cls 指针以及
// 要求是 cls 指针的 Class 必须已经进行链接操作
static void schedule_class_load(Class cls) {
    if (!cls) return;
    // 查看 RW_REALIZED 是否被标记
    assert(cls->isRealized());

    // 查看 RW_LOADED 是否被标记
    if (cls->data()->flags & RW_LOADED) return;

    // 递归到深层（超类）运行
    schedule_class_load(cls->superclass);

    // 将需要执行 load 的 Class 添加到一个全局列表中
    add_class_to_loadable_list(cls);
    // 标记 RW_LOADED 符号
    cls->setInfo(RW_LOADED);
}

```

在 `schedule_class_load` 中，Class 的读取方式是用 cls 指针方式，其中有很多内存符号位用来记录状态。`isRealized()` 查看的就是 `RW_REALIZED` 位，该位记录的是当前 **Class** 是否初始化一个类的指标。而之后查看的 `RW_LOADED` 是记录当前类的 `+load` 方法是否被调用。

在存储静态表的方法中，方法对象会以指针的方式作为参数传递，然后用名为 `loadable_classes` 的静态类数组对即将运行的 load 方法进行存储，其下标索引 `loadable_classes_used` 为（从零开始的）全局量，并在每次录入方法后做自加操作实现索引的偏移。

由此可以看到，在 `prepare_load_methods` 方法中，Runtime 方法进行了 Class 和 Category 的筛选过滤工作，并且将即将执行的 load 方法以指针的形式组织成了一个线性表结构，为之后执行操作中打下基础。

通过函数指针让 load 方法跑起来

经过加载镜像、缓存类列表后，开始执行 `call_load_methods` 方法。

```

void call_load_methods(void) {
    // 是否已经录入
    static bool loading = NO;
    // 是否有关联的 Category
    bool more_categories;
    loadMethodLock.assertLocked();

    // 由于 loading 是全局静态布尔量，如果已经录入方法则直接退出
    if (loading) return;
    loading = YES;
    // 声明一个 autoreleasePool 对象
    // 使用 push 操作其目的是为了创建一个新的 autoreleasePool 对象
    void *pool = objc_autoreleasePoolPush();

    do {
        // 重复调用 load 方法，直到没有
        while (loadable_classes_used > 0) {
            call_class_loads();
        }

        // 调用 Category 中的 load 方法
        more_categories = call_category_loads();

        // 继续调用，直到所有 Class 全部完成
    } while (loadable_classes_used > 0 || more_categories);
    // 将创建的 autoreleasePool 对象释放
    objc_autoreleasePoolPop(pool);
    // 更改全局标记，表示已经录入
    loading = NO;
}

```

其实 `call_load_methods` 由以上代码可知，仅是运行 load 方法的入口。其中最重要的方法 `call_class_loads` 会从一个待加载的类列表 `loadable_classes` 中寻找对应的类，并使用 `selector(load)` 的实现并执行。

```

static void call_class_loads(void) {
    // 声明下标偏移
    int i;
    // 分离加载的 Class 列表
    struct loadable_class *classes = loadable_classes;
    // 调用标记
    int used = loadable_classes_used;
    loadable_classes = nil;
    loadable_classes_allocated = 0;
    loadable_classes_used = 0;

    // 调用列表中的 Class 类的 load 方法
    for (i = 0; i < used; i++) {
        // 获取 Class 指针
        Class cls = classes[i].cls;
        // 获取方法对象
        load_method_t load_method = (load_method_t)classes[i].method;
        if (!cls) continue;
        if (PrintLoading) {
            _objc_inform("LOAD: +[%s load]\n", cls->nameForLogging());
        }
        // 方法调用
        (*load_method)(cls, SEL_load);
    }
    // 释放 Class 列表
    if (classes) free(classes);
}

```

读完源码，也许会好奇为什么 `(*load_method)(cls, SEL_load);` 这一句可以调用 load 方法？

其实这是 C 中的函数指针基本概念。在这里我用一个简单的例子做个简要说明（如果没有看懂，需要补补基础了0.0）：

```

#include <stdio.h>
#import <Foundation/Foundation.h>

void run() {
    printf("Hello World\n");
}

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        void (*dy_run)() = run;
        (*dy_run)();
    }
    return 0;
}

```

其结果会发现执行了 `run` 方法，并输出了 `Hello World`。这里，我们通过一个 `void (*fptr)()` 类型的函数指针，将 `run` 函数获取出，并运行函数。实际上其中的工作是抓取 `run` 函数的地址并存储在指针变量中。我们通过指针运行对应的地址部分，其效果为执行了 `run` 函数。

返回方法中的 `load_method_t`，我们在全局位置发现了该类型的定义：

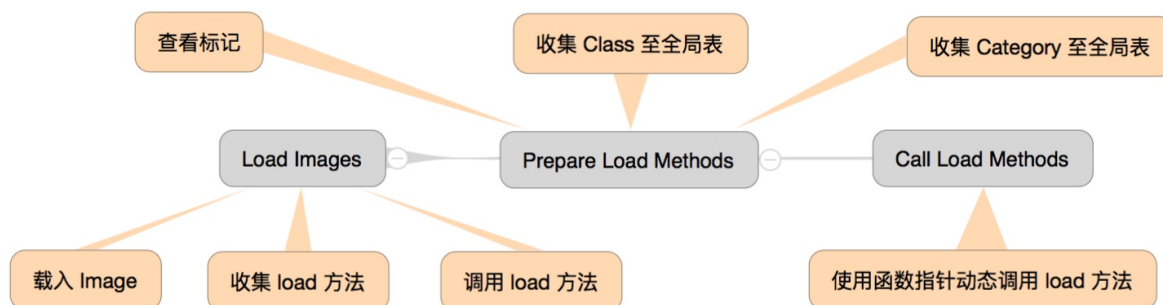
```
typedef void(*load_method_t)(id, SEL);
```

`id` 参数可以传递一个类信息，这里是将 `cls` Class 的指针和 `SEL` 选择子作为参数传入。

至此完成了 load 方法的动态调用。

全局 Class 存储线性表数据结构

总结一下 Class 中 load 方法的全部流程，用流程图将其描述一下：



下面来研究一下，存储 Class 的全局表数据结构是怎样的。

找到之前的 `add_class_to_loadable_list` 开始分析：

```

void add_class_to_loadable_list(Class cls) {
    // 定义方法指针
    // 目的是构造函数指针
    IMP method;

    loadMethodLock.assertLocked();
    // 通过 cls 中的 getLoadMethod 方法，直接获得 load 方法体存储地址
    method = cls->getLoadMethod();
    // 没有 load 方法直接返回
    if (!method) return;
    if (PrintLoading) {
        _objc_inform("LOAD: class '%s' scheduled for +load",
                    cls->nameForLogging());
    }
    // 判断数组是否已满
    if (loadable_classes_used == loadable_classes_allocated) {
        // 动态扩容，为线性表释放空间
        loadable_classes_allocated = loadable_classes_allocated*2 + 16;
        loadable_classes = (struct loadable_class *)
            realloc(loadable_classes,
                    loadable_classes_allocated *
                    sizeof(struct loadable_class));
    }
    // 将 Class 指针和方法指针记录
    loadable_classes[loadable_classes_used].cls = cls;
    loadable_classes[loadable_classes_used].method = method;
    // 游标自加偏移
    loadable_classes_used++;
}

```

在记录过程中，可以看到其 Class 指针和方法指针的记录手段是通过构造 `loadable_classes` 这个类型的数组进行静态线性表记录。这个类型的数组其数据结构定义如下：

```

typedef struct objc_class *Class;
struct loadable_class {
    Class cls;
    IMP method;
};

```

其 `objc_class` 结构笔者在[用 isa 承载对象的类信息](#)一文中较为详细的介绍，这是对于 Class 的抽象。从此看出，全局 Class 存储线性表结构，内部记录的信息只有 Class 指针和方法指针，这已经足够了。

load 方法的调用情况至此已经全部清晰。思路梳理如下三大流程：

- Load Images: 通过 `dylld` 载入 image 文件，引入 Class。
- Prepare Load Methods: 准备 load 方法。过滤无效类、无效方法，将 load 方法指针和所属 Class 指针收集至全局 Class 存储线性表 `loadable_classes` 中，其中会涉及到自动扩展空间

和父类优先的递归调用问题。

- Call Load Methods: 根据收集到的函数指针，对 load 方法进行动态调用。进一步过滤无效方法，并记录 log 日志。

Load 方法作用

`load` 方法是在开发中最接近 app 启动的可控方法。即在 app 启动以后，入口函数 `main` 之前。

由于调用有着 *non-lazy* 属性，并且在运行期只调用一次，于是我们可以使用 `load` 独有的特性和调用时机来尝试 Method Swizzling。当然因为 load 调用时机过早，并且当多个 Class 没有关联（继承与派生），我们无法知道 Class 中 load 方法的优先调用关系，所以一般不会在 load 方法中引入其他的类，这是在开发当中需要注意的。

参考链接

[你真的了解 Objective-C 中的load 方法么？](#)

[NSObject +load and +initialize - What do they do?](#)

[Objective-C Class Loading and Initialization](#)

[+load VS +initialize](#)

[Objective-C +load vs +initialize](#)

[Objective-C: What is a lazy class?](#)