289 days ago

细聊 Cocoapods 与 Xcode 工程配置

前言

文章比较长，所以在文章的开头我打算简单介绍一下这篇文章将要讲述的内容，读者可以选择通篇细度，也可以直接找到自己感兴趣的部分。

既然是谈 Cocoapods，那首先要搞明白它出现的背景。有经验的开发者都知道 Cocoapods 在实际使用中，经常遇到各种问题，存在一定的使用成本，因此衡量 Cocoapods 的成本和收益就显得很关键。

Cocoapods 的本质是一套自动化工具。那么了解自动化流程背后的原理就很重要，如果我们能手动的模拟 Cocoapods 的流程，无论是对 Cocoapods 还是 Xcode 工程配置的学习都大有裨益。比如之前曾经和同事研究过静态库嵌套的问题，很遗憾当时没能解决，现在想来还是对相关知识理解还不够到位。这一部分主要是介绍 Xcode 的工程配置，以及 target/project/workspace 等名词的概念。

最后，我会结合实际的例子，谈谈如何发布自己的 Pod，提供给别人使用。算是对 Cocoapods 的实践总结。

由于实践性的操作比较多，我为本文制作了一个 demo，提交在 我的 Github: CocoaPodsDemo 上，感兴趣的读者可以下载下来，研究一下提交历史，或者自己操作一遍。友情提醒: 本文所涉及的静态库均为模拟器制作，请勿真机运行。

为什么要使用 Cocoapods

我们知道，再大的项目最初都是从 Xcode 提供的一个非常简单的工程模板慢慢演化来的。在项目的演化过程中，为了实现新的功能，不断有新的类被创建，新的代码被添加。不过除了自己添加代码，我们也经常会直接把第三方的开源代码导入到项目中，从而避免重复造轮子，节约开发时间。

直接把代码导入到项目中看起来很容易，但在实践过程中，会遇到诸多问题。这些问题会困扰代码的使用者，大大的增加了集成代码的难度。

使用者的困扰

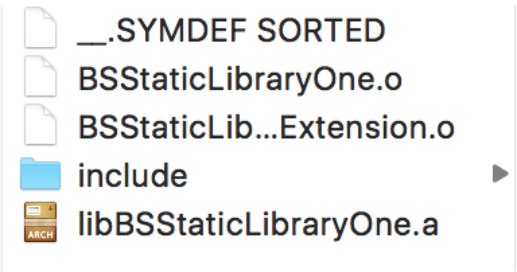
最直接的问题就是代码的后续维护。假设代码的发布者在未来的某一天更新了代码，修复了一个重大 bug 或者提供了新的功能，那么使用者就很难集成这些变动。

代码有增有删，如果把代码编译成静态库再提供给使用者， 就可以省掉很多问题。然而如果这么做的话，就会遇到另一个经典的问题: "Other linker flag"。

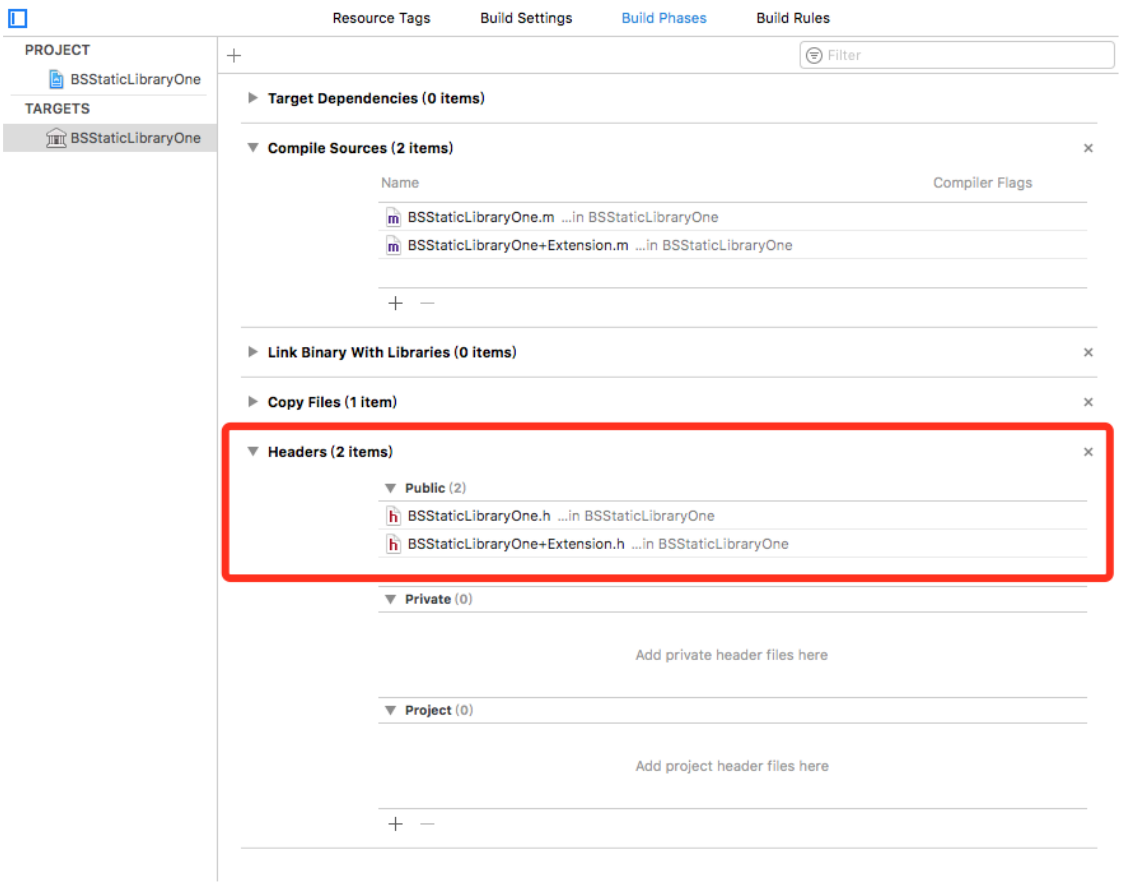
举个例子来说，可以在 Demo 的 `BSStaticLibraryOne` 这个项目中看到，这个静态库一共有两个类，其中一个是拓展 Extension。项目编译后就会得到一个 `.a` 文件。

我们都知道静态库的格式可以是 `.framework`，也可以是 `.a`。如果深究的话，`.a` 文件可以理解为一种归档文件，或者说是压缩文件。其中存储的是经过编译的 `.o` 格式的目标文件。我们可以通过 `ar -x` 命令来证明这一点:

```
ar -x libBSStaticLibraryOne.a
```



需要提醒的一点是，光有 `.a` 文件还不够，我们还需要提供头文件给使用者导入。为了完成这一点，我们需要在项目的 Build Phases 中新增一个 Headers Phase，然后把需要对外暴露的头文件放到 Public 一栏中:



此时编译后的头文件会放在 `.a` 文件所在目录下，`usr/local/include` 目录中。

接下来打开 `OtherLinkerFlag` 这个壳工程，引入 `.a` 文件和头文件，运行程序，结果一定是:

```
-[BSStaticLibraryOne sayOtherThing]: unrecognized selector sent to instance xxx
```

这就是经典的 `linker flag` 问题。首先，我们知道 `.a` 其实是编译好的目标文件的集合，因此问题出在链接这一步，而非编译。Objective-C 在使用静态库时，需要知道哪些文件需要链接进来，它依据的就是之前图中所示的 `__.SYMDEF SORTED` 文件。

可惜的是，这个文件不会包含所有的 `.o` 目标文件，而只是包含了定义了类的目标文件。我们可以执行 `cat __.SYMDEF\ SORTED` 来验证一下，你会看到其中并没有拓展类的信息。这样一来，`BSStaticLibraryOne+Extension.o` 虽然存在，但是不被链接到最终的可执行文件中，从而导致了找不到方法的错误。

解决上述问题的方法是调用者在 `Build Settings` 中找到 `other linker flag`，并写上 `-ObjC` 选项，这个选项会链接所有的目标文件。然而根据文档描述，如果静态库只有分类，而没有类， 即使加了 `-ObjC` 选项也会报错，应该使用 `-force_load` 参数。

由于第三方的代码使用分类几乎是必然事件，因此几乎每个使用者都要做如上配置，增加了复杂度和出错的几率。

除此以外，第三方的代码很有可能使用了系统的动态库。因此使用者还必须**手动引入这些动态库**(请记住这一点，静态库不支持递归引用，这是个很麻烦的事情，后面会介绍)，我们以百度地图 SDK 的集成为例，读者可以自行对比手动导入和 Cocoapods 集成的步骤区别: 配置开发环境iOS SDK 。

因此，我总结的使用 Cocoapods 的好处有如下几个:

1. 避免直接导入文件的原始方式，方便后续代码升级
2. 简化、自动化集成流程，避免不必要的配置
3. 自动处理库的依赖关系
4. 简化开发者发布代码流程

Cocoapods 工作原理

在我之前的一篇文章: 白话 Ruby 与 DSL 以及在 iOS 开发中的运用 中简单的介绍过，Cocoapods 是用 Ruby 开发的一套工具。每一份代码都是一个 Pod，安装 Pod 时首先会分析库的版本和依赖关系，这些都是在 Ruby 层面完成的，本文暂且不表。

我们首先假设已经找到了要下载的代码的地址(比如存在 Github 上)，从这一步开始，接下来的工作都与 iOS 开发有关。

如果你手头有一个 Cocoapods 项目，你应该会注意到以下几个特点:

1. 主工程中没有导入第三方库的代码或静态库
2. 主工程不显式的依赖各个第三方库，但是引用了 `libPods.a` 这个 Cocoapods 库
3. 不需要手动编译第三方库，直接运行主工程即可，隐式指定了编译顺序

这样做可以把引入第三方库对主工程造成的影响降到最低，不过无法完全降为零。比如引入 Cocoapods 以后，项目不得不使用 `xworkspace` 来打开，后面会介绍原因。

假设之前的 `BSStaticLibraryOne` 工程就是下载好的源码，现在我们要做的就是把它集成到一个已有的工程，比如叫 `ShellProject` 中。

我们遇到的第一个问题是，在之前的 demo 中，需要把静态库和头文件手动拖入到工程中。但这就和 Cocoapods 的效果不一致，毕竟我们希望主工程完全不受影响。

静态库和头文件导入

如果我们什么都不做，当然不可能在壳工程中引用另一个项目下的静态库和头文件。但这个问题也可以换个方式问:"Xcode 怎么知道它们可以引用，还是不可以引用呢？"，答案在于 Build Settings 里面的 Search Paths 这一节。默认情况下，**Header Search Path** 和 **Library Search Path** 都是空的，也就是说 Xcode 不会去任何目录下找静态库和头文件，除非他们被人为的导入到工程中来。

因此，只要对上述两个选项的值略作修改， Xcode 就可以识别了。我们目前的项目结构如下所示:

```
- CocoaPodsDemo(根目录)
    - BSStaticLibraryOne (被引用的静态库)
        - Build/Products/Debug-iphonesimulator (编译结果的目录)
            - libBSStaticLibraryOne.a  (静态库)
            - usr/local/include (头文件目录)
                - BSStaticLibraryOne.h
                - BSStaticLibraryOne+Extension.h
    - ShellProject (壳工程)
```

因此我们要做的是让壳工程的 **Library Search Path** 指向 **CocoaPodsDemo/BSStaticLibraryOne/Build/Products/Debug-iphonesimulator** 这个目录:

```
Library Search Path = $PROJECT_DIR/../BSStaticLibraryOne/Build/Products/Debug-iphonesimulator/
```

这里记得写相对路径，Xcode 会自动转成绝对路径。然后 **Header Search Path** 也如法炮制:

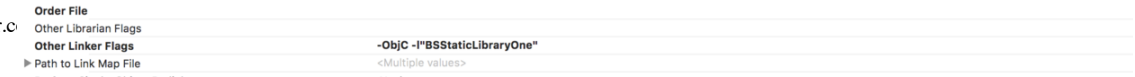
```
Header Search Path = $PROJECT_DIR/../BSStaticLibraryOne/Build/Products/Debug-iphonesimulator
```

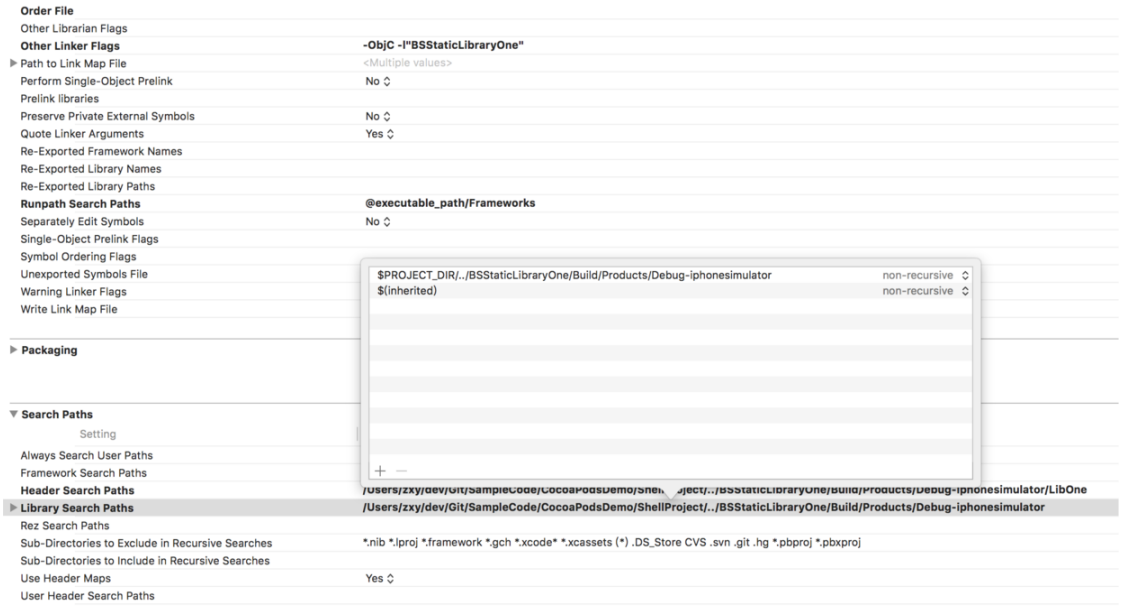
细心的读者也许会发现， LibOne 这个文件夹完全不存在。是这样的，因为我觉得 **usr/local/include** 这个路径太深，太丑，所以可以在静态库的项目配置中，在 **Packaging** 这一节中，找到 **Public Headers Folder Path**，将它的值从 **usr/local/include** 修改为 **LibOne**，然后重新编译，这时就会看到生成的头文件位置发生了变化。

当然，这时候还是无法直接引用静态库的。因为我们只是告诉 Xcode 可以去对应路径去找，但并没有明确声明要用，所以需要在 **Other Linker Flags** 中添加一个选项: `-l"BSStaticLibraryOne"`，引号中的内容就是静态库的工程名。

需要提醒的是， 静态库编译出来的 `.a` 文件会被手动加上 `lib` 前缀，在写入到 **Other Linker Flags** 的时候千万要注意去掉这个前缀，否则就会出现 **Library not found** 的错误。

配置好以后的工程如下图所示:





现在项目中没有任何第三方的库或者代码，依然可以正常引用第三方的类并运行成功。

引用多个第三方库

当我们的项目需要引用多个第三方库的时候，就有两种思路:

1. 每份第三方代码作为一个工程，分别打出一个静态库和头文件。
2. 所有第三方代码放在同一个工程中，建立多个 target，每个 target 对应一个静态库。

从直觉来看，第二种组织方式看上去更加集中，易于管理。考虑后面我们还要解决库的依赖问题，而且项目内的依赖处理比 workspace 中的依赖处理要容易很多(后面会介绍到)，所以第二种组织方式更具有可行性。

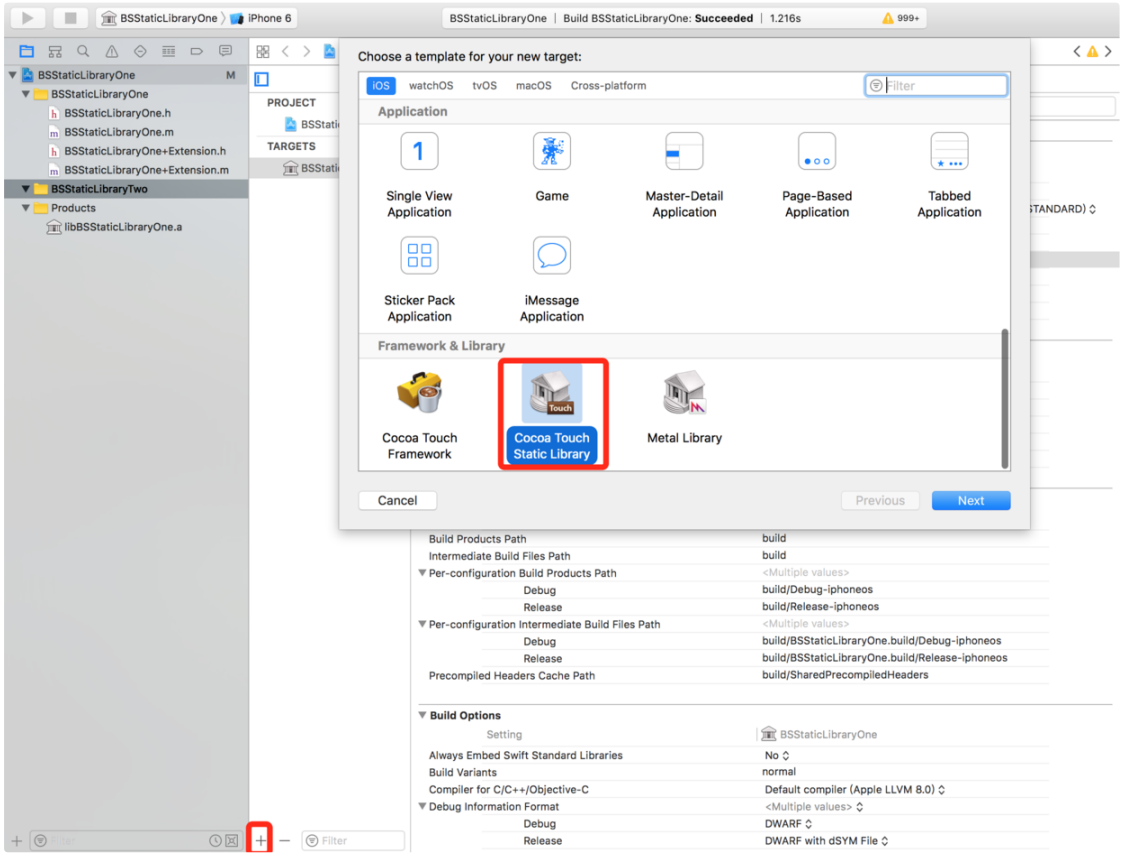
如果读者手头有使用了 Cocoapods 的项目，可以看到它的文件组织结构如下:

```
- ShellProject(根目录，壳工程)
    - ShellProject （项目代码）
    - ShellProject.xcodeproj （项目文件）
    - Pods （第三方库的根目录）
        - Pods.xcodeproj （第三方库的总工程）
        - AFNetworking （某个第三方库）
        - Mantle （另一个第三方库）
        - ……
```

而在我的 demo 中，为了偷懒，没有把第三方库放在壳工程目录下，而是选择和它平级。这其实没有太大的区别，只是引用路径不同而已，不用太关心。我们现在模拟添加一个新的第三方库，完成后的代码结构如下:

```
- CocoaPodsDemo(根目录)
    - BSStaticLibraryOne （第三方库总的文件夹，相当于 Pods，因为偷懒，名字就不改了）
        - BSStaticLibraryOne （第一个第三方库）
        - BSStaticLibraryTwo （新增一个第三方库）
        - BSStaticLibraryOne.xcodeproj （第三方库的项目文件）
        - Build/Products/Debug-iphonesimulator （编译结果的目录）
    - ShellProject （壳工程）
```

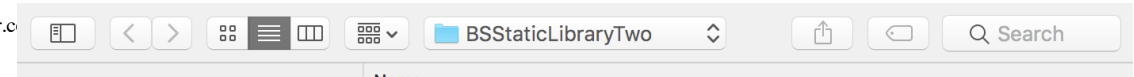
首先要新建一个文件夹 **BSStaticLibraryTwo** 并拖入到项目中，然后新增一个 Target(如下图所示)。

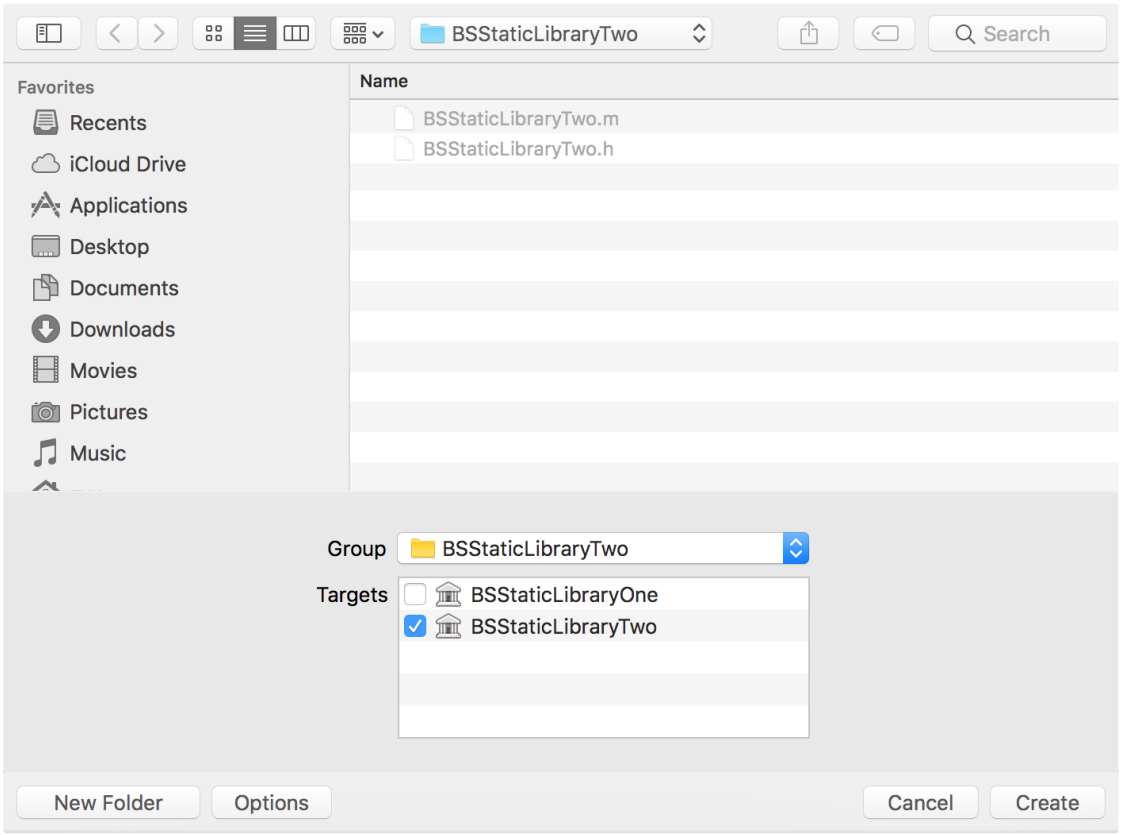


在 Xcode 工程中，我们都接触过 Project。打开 **.xcodeproj** 文件就是打开一个项目(Project)。Project 负责的是项目代码管理。一个 Project 可以有多个 Target，这些 target 可以使用不同的文件，最后也就可以得出不同的编译产物。

通过使用多个 target，我们可以用少许不同的代码得到不同的 app，从而避免了开多个工程的必要。不过我们这里的几个 target 并不含有相同代码，而是一个第三方库对应一个 target。

接下来我们新建一个类，记得要加入到 **BSStaticLibraryTwo** 这个 target 下，记得和之前一样修改 **Public Headers Folder Path** 并添加一个 **Build Phase**。

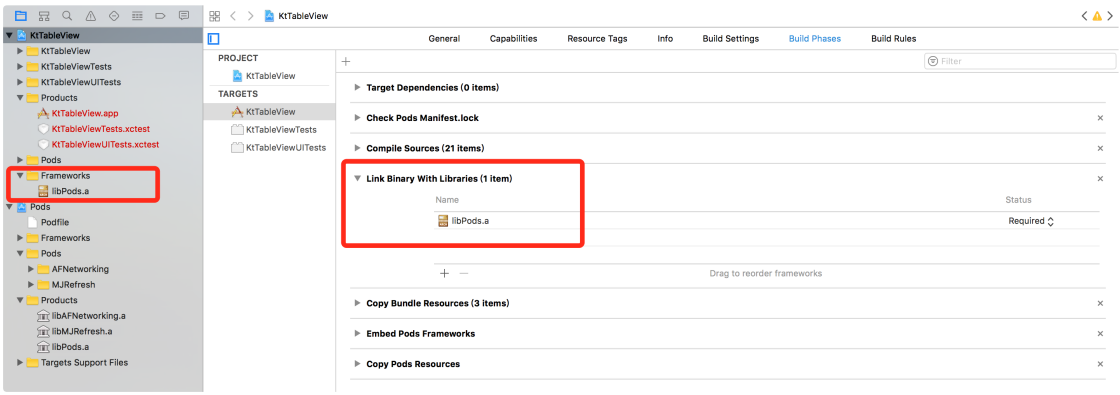




在左上角将 Scheme 选择为 **BSStaticLibraryTwo** 再编译，可以看到新的静态库已经生成了。

项目内依赖

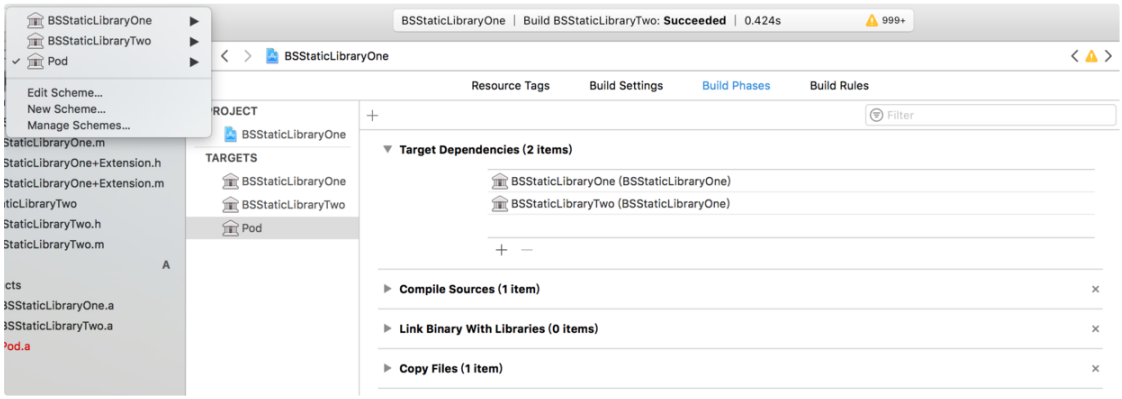
对于主工程来说，必须在子工程(第三方库)编译完后才开始编译，或者换句话说，我们在主工程中按下 Command + R/B 时，所有子工程必须先被编译。对于这种跨工程的库依赖，我们无法直接指明依赖关系，必须隐式的设置依赖关系，我们还是以 Cocoapods 工程举例:



主工程中用到了 **libPod.a** 这个静态库，而且它并不是在主工程中生成，而是在 Pods 这个项目中编译生成。一旦存在这种**引用关系**，那么也就建立了隐式的依赖关系。在编译主工程时，Xcode 会确保它引用的所有静态库都先被编译。

之前我们讨论过两种管理多个静态库的方法，如果选择第一种方法， 每个静态库对应一个 Xcode 项目，虽然不是不可以，但主工程看上去就就会比较复杂，这主要是跨项目依赖导致的。

而在项目内部管理 target 的依赖相对而言就简单很多了。我们只要新建一个总的 target，不妨也叫作 Pod。它什么也不做，只需要依赖另外两个静态库就可以了，设置 **Target Dependencies**:



此时选择 Pod 这个 target 编译，另外两个静态库也会被编译。因此接下来的任务就是让主工程直接依赖于 Pod 这个 target，自然也就间接依赖于真正有用的各个第三方静态库了。

接下来我们重复之前的步骤，设置好头文件和静态库的搜索路径，并在 Other Linker Flags 里面添加: `-l"BSStaticLibraryTwo"`，就可以使用第二个静态库了。

Workspace

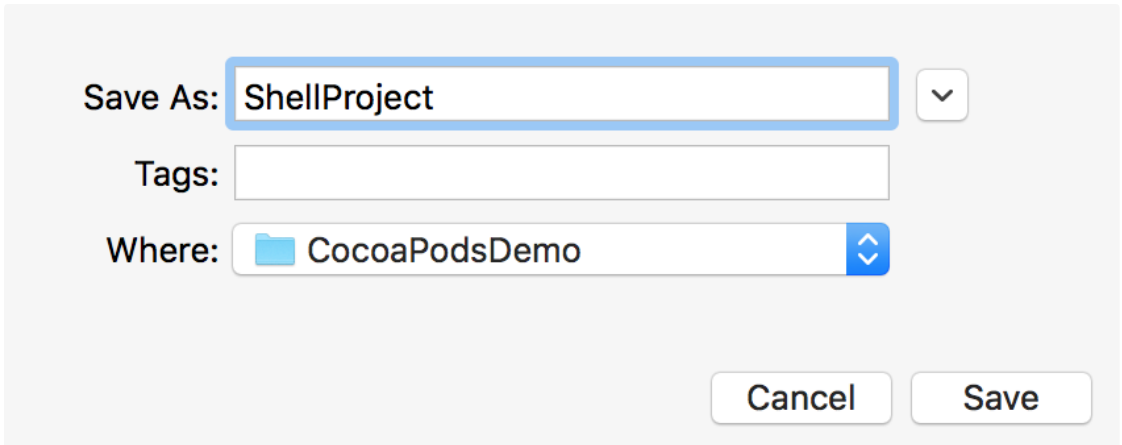
到目前为止，我们模拟了多个静态库的组织，以及如何在主工程中引用他们。不过还存在一些小瑕疵，我截了 Xcode 中的一幅图:

```
@interface ViewController ()

@end

@implementation ViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    [[[BSStaticLibraryOne alloc] init] saySomething];
    [[[BSStaticLibraryTwo alloc] init] saySomething];
    // Do any additional setup after loading the view, typically from a nib.
}
```

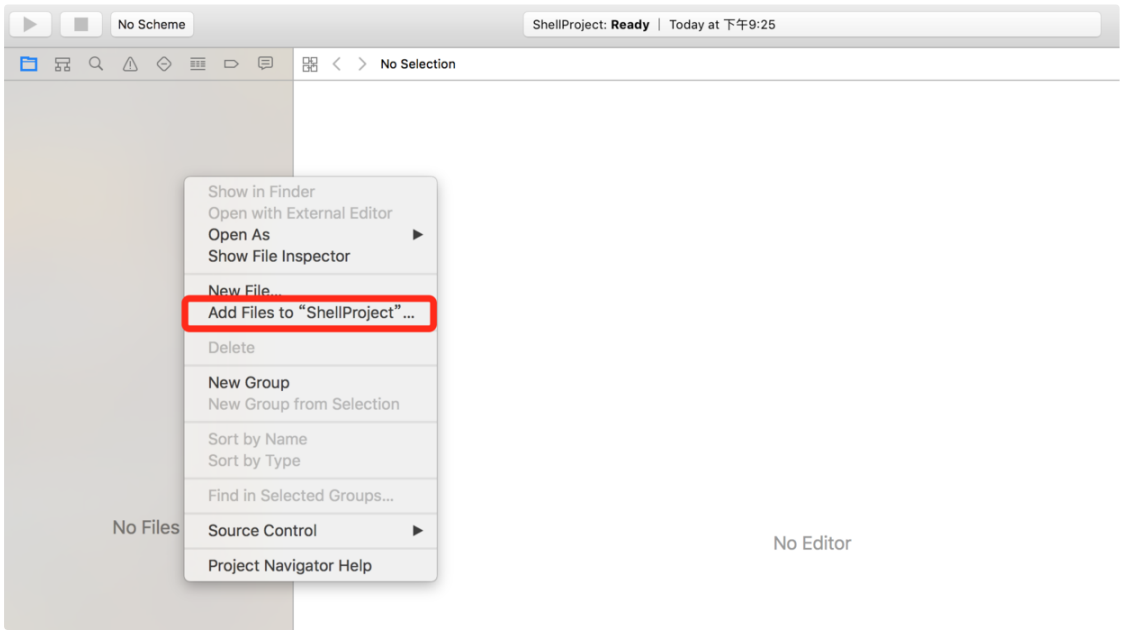
从图中可以很明显的发现: 第三方库中的代码被认为是系统代码，颜色为蓝色。而正常的自定义方法应该绿色，会对开发者造成困扰。

除了这个小瑕疵以外，在之前谈到的跨项目依赖中，一个项目不仅仅需要引用另一个项目的产物，还有一个先决条件: **把这两个项目放入同一个 Workspace 中**。Workspace 的作用是组织多个 Project，使得各个 Project 直接可以有引用依赖关系，同时也能让 Xcode 识别出各个 Project 中的代码和头文件。

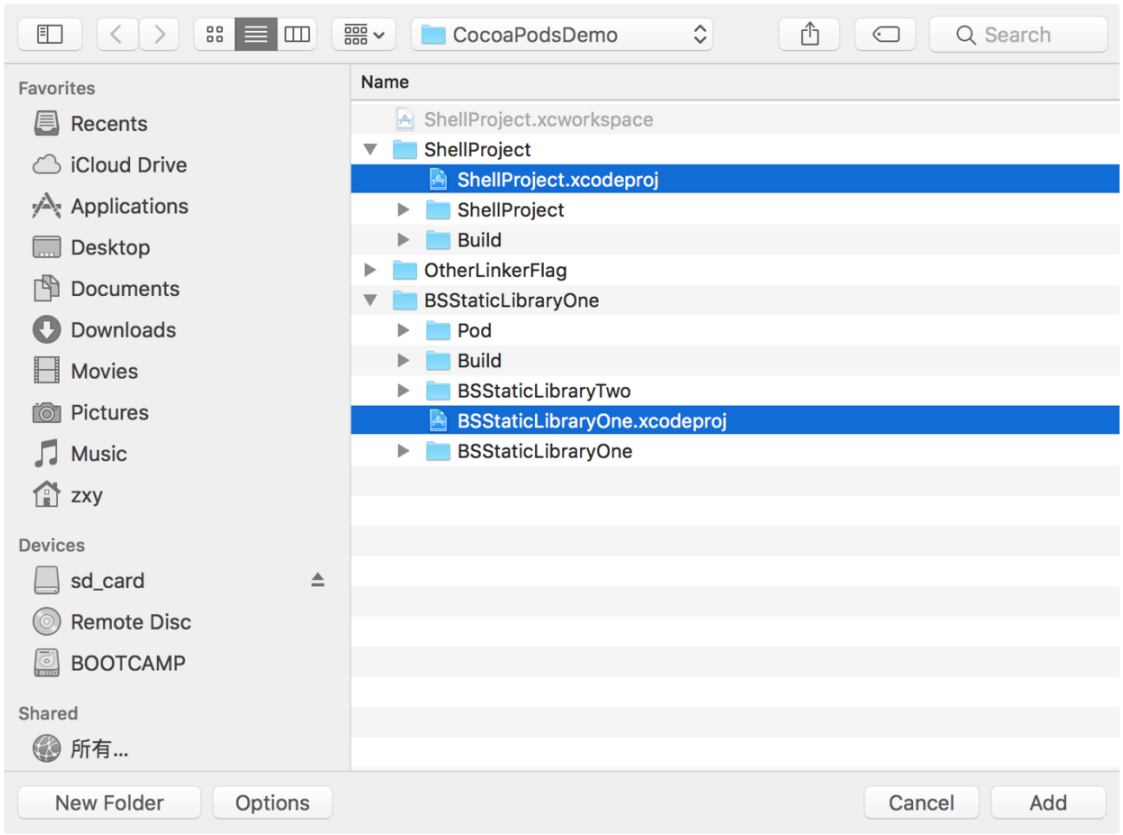
按住 Command + Control + N 可以新建一个 Workspace:



完成以后就会看到一个完全空白的项目，在左侧按下右键，选择 Add Files to:



然后选中静态库项目和主工程的 .xcodeproj 文件，把这两个工程都加进来:



需要提醒的是，切换到 Workspace 以后， Xcode 会把 Workspace 所在目录当做项目根目录，因此静态库的编译结果会放在 /CocoaPodsDemo/Build/Products/...，而不再是之前的 /CocoaPodsDemo/BSStaticLibraryOne/Build/Products/...，因此需要手动对主工程中的搜索路径做一下调整。

做好上述改动后，即使我们删除掉 BSStaticLibraryOne 这个项目的编译结果，只在 Workspace 中编译主项目，Xcode 也会自动为我们编译被依赖的静态库。这就是为什么我们只需要执行 `pod install` 下载好代码，就可以不用做别的操作，直接在主项目中运行。

当然，代码颜色错误的小问题也在 Workspace 恢复正常了。

静态库嵌套

到这里，基本上关于 Cocoapods 的工作原理就算是分析完了。上述操作除了文件增加，基本上都是修改 .pbxproj 文件。所有的 Xcode 都会在该文件中得到反映，同理，只要修改该文件，也能达到上述手动操作的效果。而 Cocoapods 开发了一套 Ruby 工具，用来封装这些修改，从而实现了自动化。

文章开头，我们提到作为代码提供者，如果自己的代码还引用别的第三方库，那么提供代码会变得很麻烦，这主要是由于静态库不会递归引用导致的。我们已经知道静态库其实就是一堆编译好的目标文件(.o 文件)的打包形式，它需要配合头文件来使用。所谓的不会递归引用是指，假设项目 A 引用了静态库 B(或者是动态库，也是一样)，那么 A 编译后得到的静态库中，并不含有静态库 B 的目标文件。如果有人拿到这样的静态库 A，就必须补齐静态库 B，否则就会遇到 "Undefined symbol" 错误。

如果我们提供的代码引用了系统的动态库，问题还比较简单，只要在文档里面注明，让使用者自己导入即可。但如果是第三方代码，那么这简直是一起灾难。即使使用者找到了提供者使用的静态库，那个静态库也很有可能已经进行了升级，而版本不一致的静态库可能具有完全不同的 API。也就是说代码提供者还要在文档中注明使用的静态库的版本，然后由使用者去找到这个版本。我想，这才是 Cocoapods 真正致力于解决的任务。

CocoaPods 的做法比较简单，因为他有一套统一的版本表示规则，也可以自动分析依赖关系，而且每个版本的代码都有记录。后面会介绍 Cocoapods 的相关实践，这里我们先思考一下如何手动解决静态库嵌套的问题。

既然静态库只是目标文件的打包形式，那么我只需要找到被嵌套的静态库，拿到其中的目标文件，然后和外层的静态库放在一起重新打包即可。这个过程比较简单， 我也就没有做 demo，用代码应该就可以说明得很清楚。假设我们有静态库 A.a 和 B.a，其中 A 需要引用 B，现在我希望对外发布 A，并且集成 B:

```
lipo A.a -thin x86_64 output A_64.a # 如果是多 CPU 架构，先提取出某一种架构下的 .a 文件
lipo B.a -thin x86_64 output B_64.a
ar -x A_64.a # 解压 A 中的目标文件
ar -x B_64.a # 解压 B 中的目标文件
libtool -static -o Together.a *.o # 把所有 .o 文件一起打包到 Together.a 中
```

这时候 **Together.a** 文件就可以当做完整版的静态库 A 给别人使用了。

Cocoapods 使用

本来 Cocoapods 的使用就比较简单。尤其是了解完原理后，使用起来应该更加得心应手了，对于一些常见的错误也有了分析能力。不过有个小细节还是需要注意一下:

Podfile.lock

关于 Cocoapods 文件是否要加入版本控制并没有明确的答案。我以前的习惯是不加入版本控制。因为这样会让提交历史明显变得复杂，如果不同分支上使用的不同版本的 pod，在合并分支时就会出现大量冲突。

然而官方的推荐是把它加入到版本控制中去。这样别人不再需要执行 `pod install`，而且能够确保所有人的代码一定一致。

然而虽然不强制把整个 Pod 都加入版本控制，但是 **Podfile.lock** 无论如何必须添加到版本控制系统中。为了解释这个问题，我们先来看看 Cocoapods 可能存在的问题。

假设我们在 Podfile 中写上: `pod 'AFNetWorking'`，那么默认是安装 AFNetworking 的最新代码。这就导致用户 A 可能装的是 3.0 版本，而用户 B 再安装就变成了 4.0 版本。即使我们在 Podfile 中指定了库的具体版本，那也不能保证不出问题。因为一个第三方库还有可能依赖其他的第三方库，而且不保证它的依赖关系是具体到版本号的。

因此 **Podfile.lock** 存在的意义是将某一次 `pod install` 时使用的各个库的版本，以及这个库依赖的其他第三方库的版本记录下来，以供别人使用。这样一来，`pod install` 的流程其实是:

1. 判断 Podfile.lock 是否存在，如果不存在，按照 Podfile 中指定的版本安装
2. 如果 Podfile.lock 存在，检查 Podfile 中每一个 Pod 在 Podfile.lock 中是否存在
3. 如果存在， 则忽略 Podfile 中的配置，使用 Podfile.lock 中的配置(实际上就是什么都不做)
4. 如果不存在，则使用 Podfile 中的配置，并写入 Podfile.lock 中

而另一个常用命令 `pod update` 并不是一个日常更新命令。它的原理是忽略 Podfile.lock 文件，完全使用 Podfile 中的配置，并且更新 Podfile.lock。一旦决定使用 `pod update`，就必须所有团队成员一起更新。因此在使用 `update` 前请务必了解其背后发生的事情和对团队造成的影响，并且确保有必要这么做。

发布自己的 Pod

很多教程都有介绍开源 Pod 的流程，我在实践的时候主要参考了以下两篇文章。相对来说比较详细，条理清晰，也推荐给大家:

1. Cocoapods系列教程(二)——开源主义接班人
2. Cocoapods系列教程(三)——私有库管理和模块化管理

如果要创建公司内部的私有库，首先要建立一个自己的仓库，这个仓库在本地也会有存储:

如图中所示，master 是官方仓库，而 baidu 则是我用来测试的私有仓库。仓库中会存有所有 Pod 的信息，每个文件夹下都按照版本号做了区分，每个版本对应一个 podspec 文件。从图中可以看到，cocoapods 会缓存所有的 podspec 到本地，但不会缓存每个 Pod 的具体代码。每当我们执行 `pod install` 时，都会先从本地查找 podspec 缓存是否存在，如果不存在则会去中央仓库下载。

我们经常遇到的 `pod install` 很慢就是因为默认情况下会更新整个 master。此时 master 不仅仅存储着本地使用 Pod 的 PodSpec 文件，而是存储了所有的已有的 Pod。所以这个更新过程看起来异常缓慢。有些解决方案是使用:

```
pod install --verbose --no-repo-update
```

这其实是治标不治本的姑息治疗方法，因为本地的仓库迟早要被更新，否则就拿不到最新的 PodSpec。要想彻底解决这一问题，除了定期更新外，还可以选择其他速度较快的镜像仓库。

podspec 文件是我们开源 Pod 时需要填写的文件，主要是描述了 Pod 的基础信息。除了一些无关紧要的配置和介绍信息外，最重要的填写 **source_files** 和 **dependency**。前者用来规定哪些文件会对外公布，后者则指定此 Pod 依赖于哪些其他 Pod。比如在上图中，我的 PrivatePod 就依赖于 CorePod，在公司内部的项目中使用 PodS 依赖可以大量简化代码的集成流程。一个典型的 PodSpec 可能长这样:

```
Pod::Spec.new do |s|
  s.name         = "PrivatePod"
  s.version      = "0.0.3"
  s.summary      = "A detailed description of PrivatePod."
  s.homepage     = "https://bestswifter.com"
  s.description  = "Summary long long long long logn long long enough?"
  s.license      = { :type => "MIT", :file => "LICENSE.md" }
  s.author       = { "zhangxingyu" => "zhangxingyu@baidu.com" }
  s.platform     = :ios, "8.0"
  s.source       = { :git => "ssh://g@gitlab.baidu.com:8022/zhangxingyu/CocoaPodDemo.git", :tag => "#{s.version}" }
  s.source_files = "PrivatePod/PrivatePod/PrivateUtility.{h,m}"
  s.requires_arc = true
  s.dependency 'CorePod'
end
```

填写好上述信息后，我们只要先 lint 一下 podspec，确保格式无误，就可以提交了。
