# 1 Xcode多工程联编

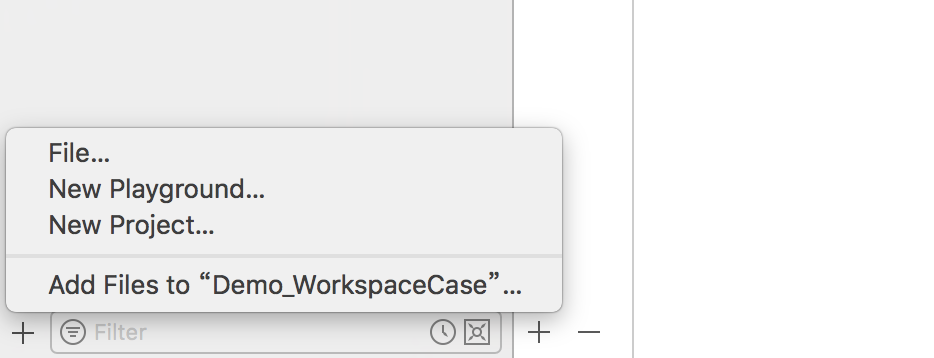
在工程比较大的时候，如果需要组件化，建议使用cocoapod进行管理，每个组件都使用一个git仓库进行管理，每个组件是一个独立的工程，可以单独运行编译（每个工程可以使用多个target，分别做不同用途），这样可以使得各个组件真正解耦，如果组件较多可以尝试搭建自己的组件库。

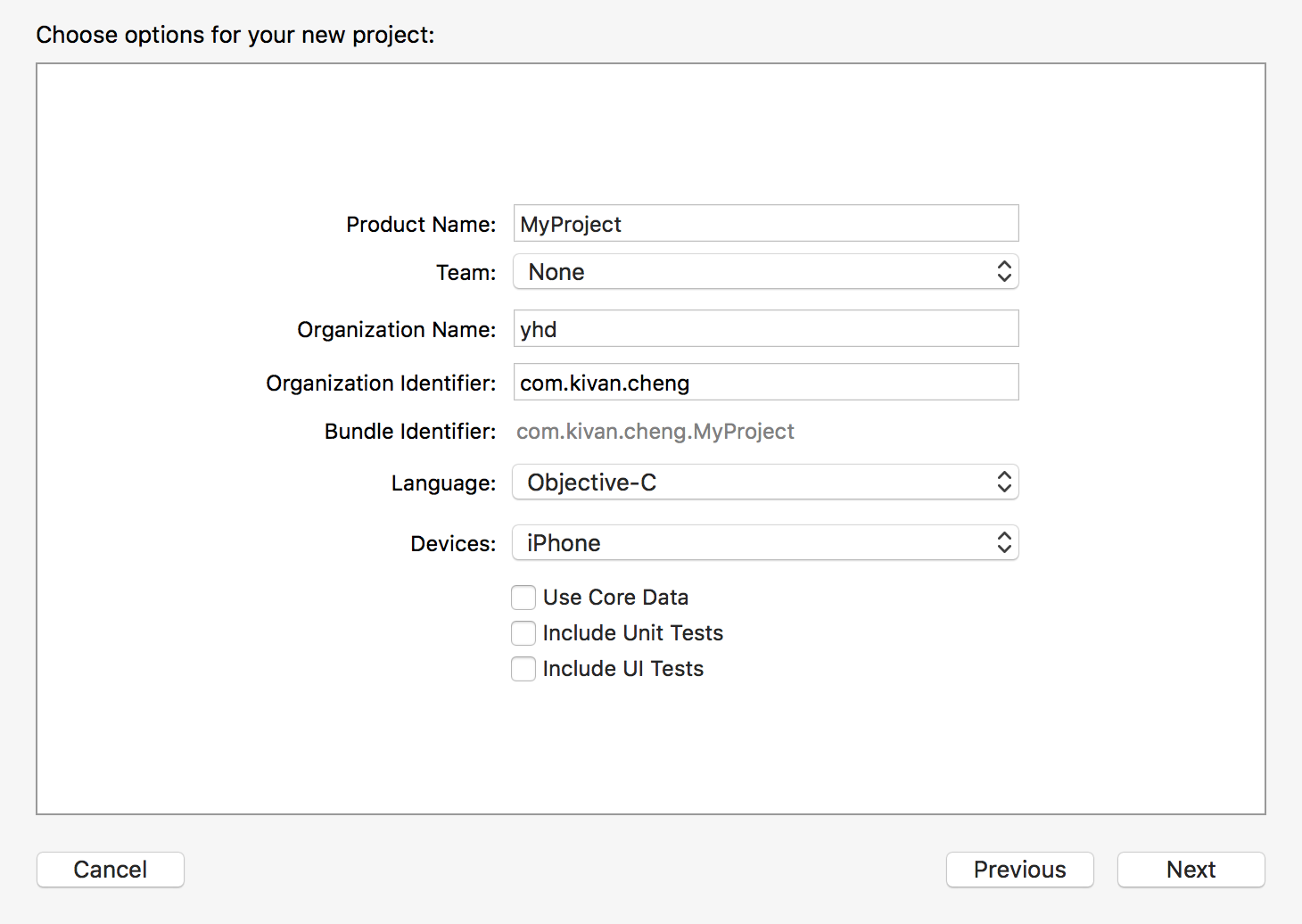
## 1.1多工程联编

主要是注意两个部分，头文件和库文件（与c类似，需要了解动态库和静态库相关知识）

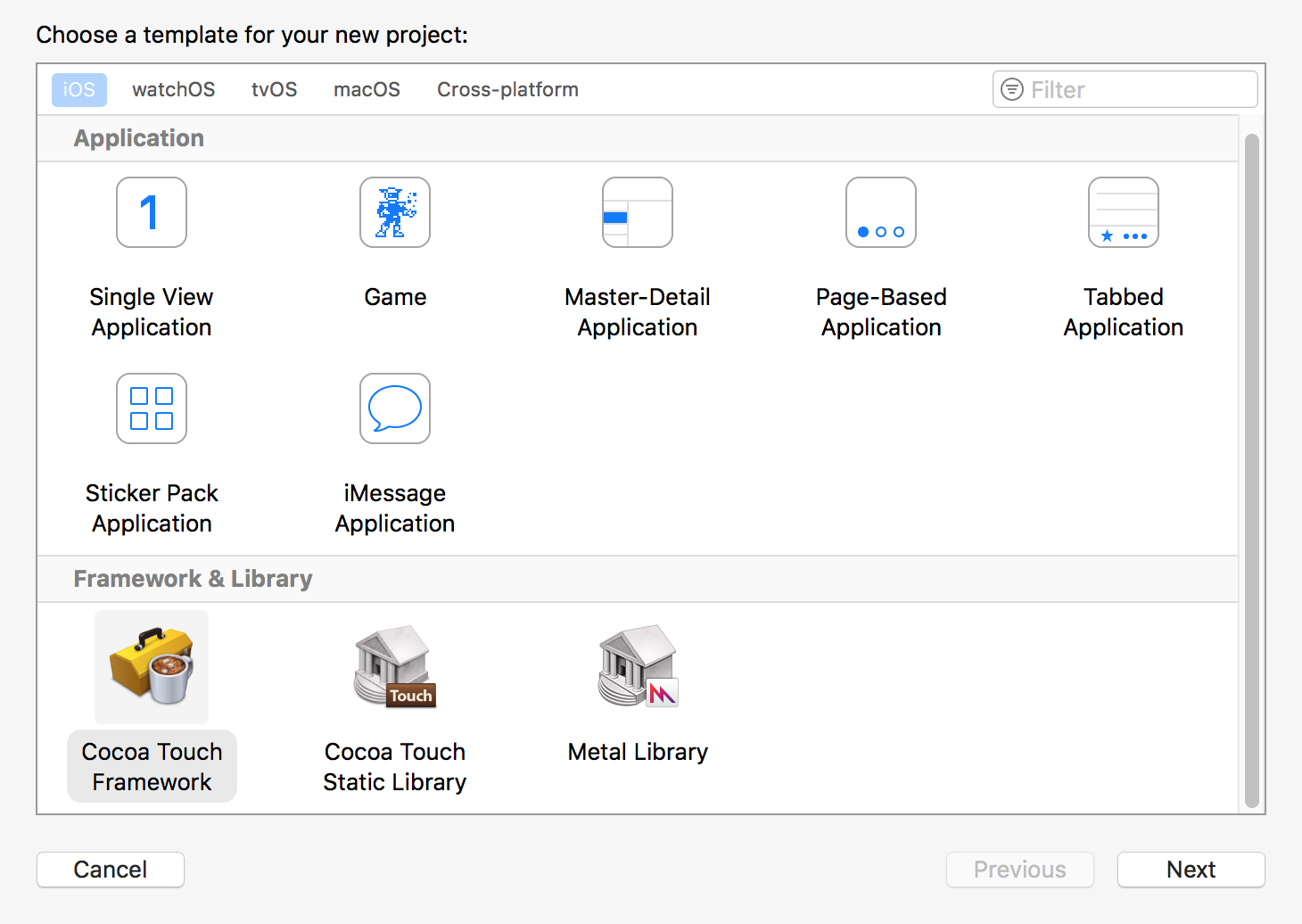
新建一个WorkSpace

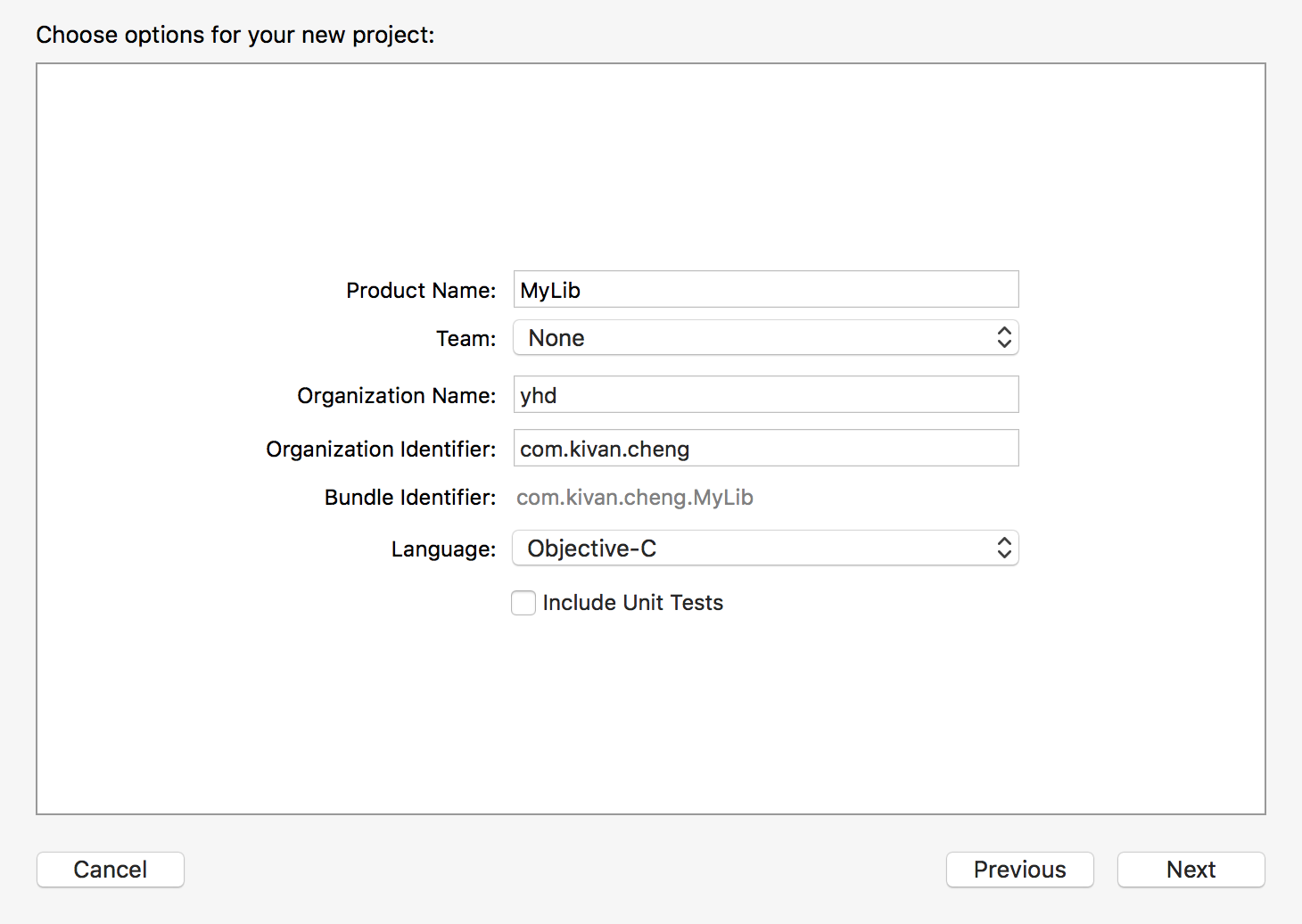
在XCode左下方的加号处添加需要新建的工程，这里新建project才是将project添加进入workspace(不是file->new->project),这里添加两个工程：MyProject和MyLib。MyLib是一个动态库。





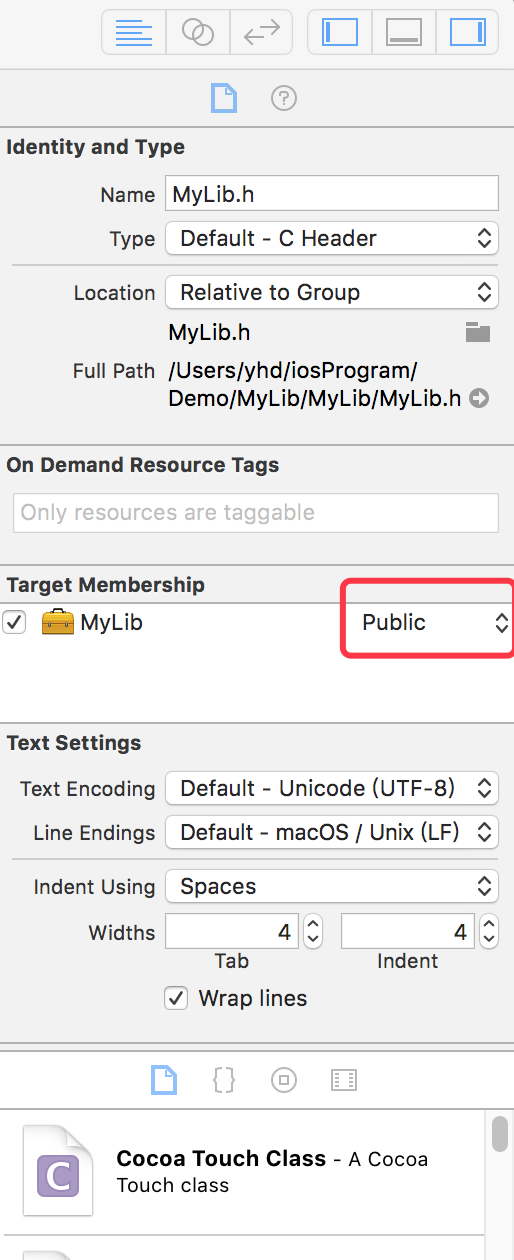
（这里有三种库可以创建，根据需要来创建）



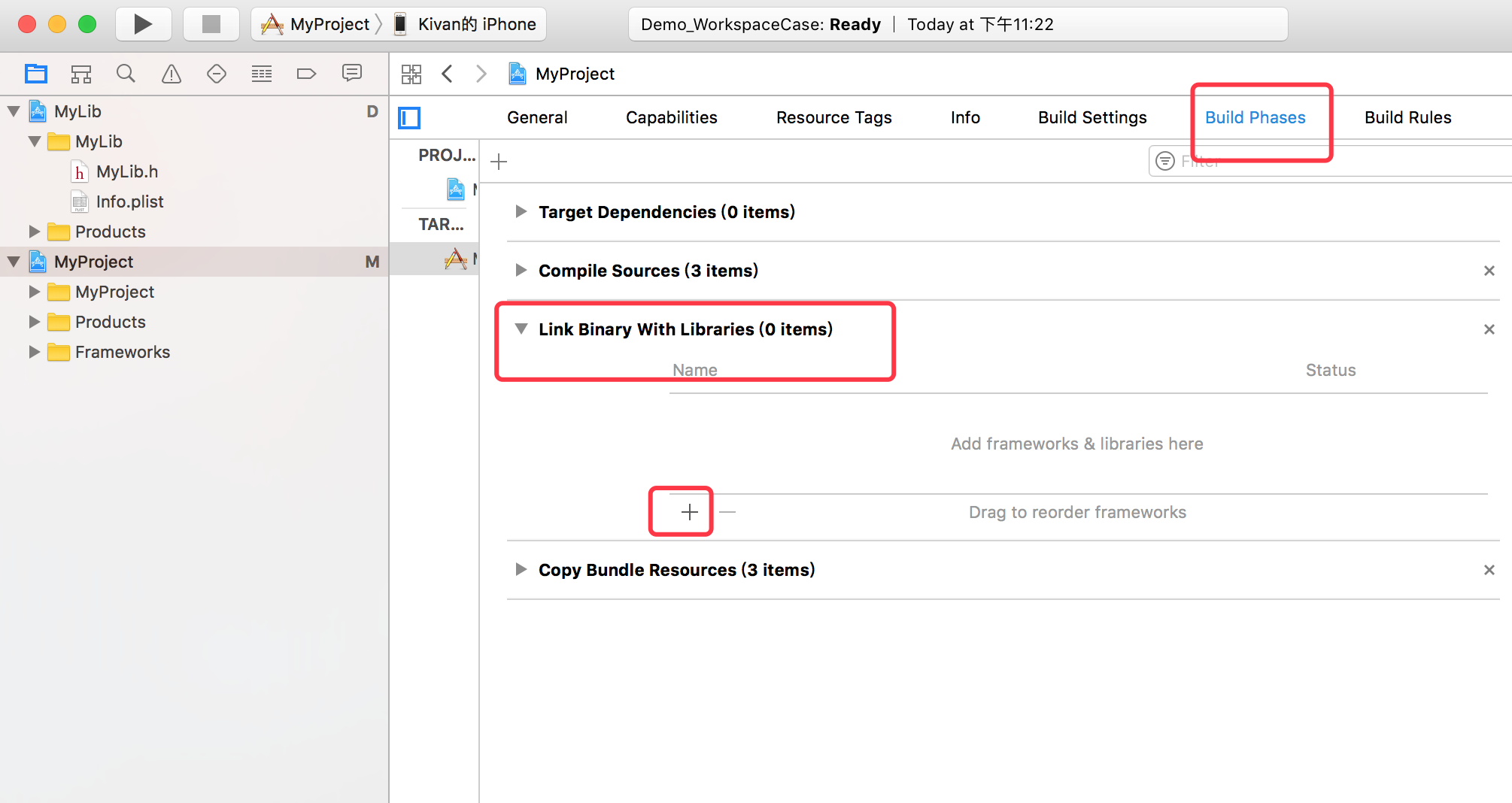


设置MyLib库中需要公开的头文件，并其加入到MyProject中（Build Phases ->Link Binary With Libraries 如下图

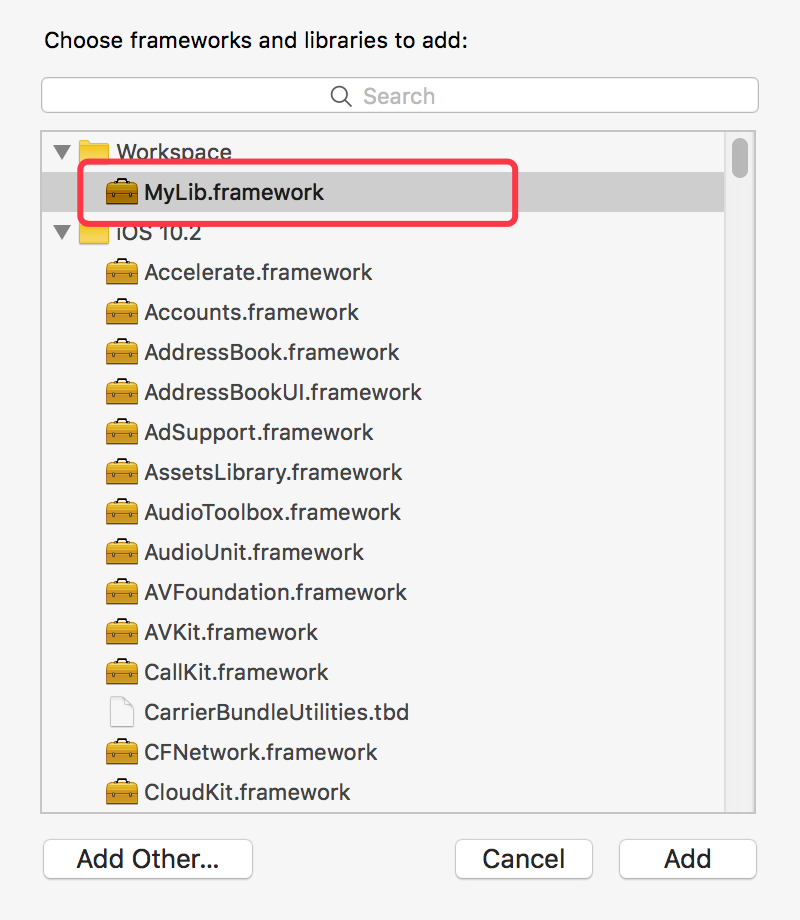
），在MyProject工程中设置好USER\_HEADER\_SEARCH\_PATHS路径，在其中添加MyLib库工程文件的路径，这样MyProject工程就可以使用#import来引入MyLib了。



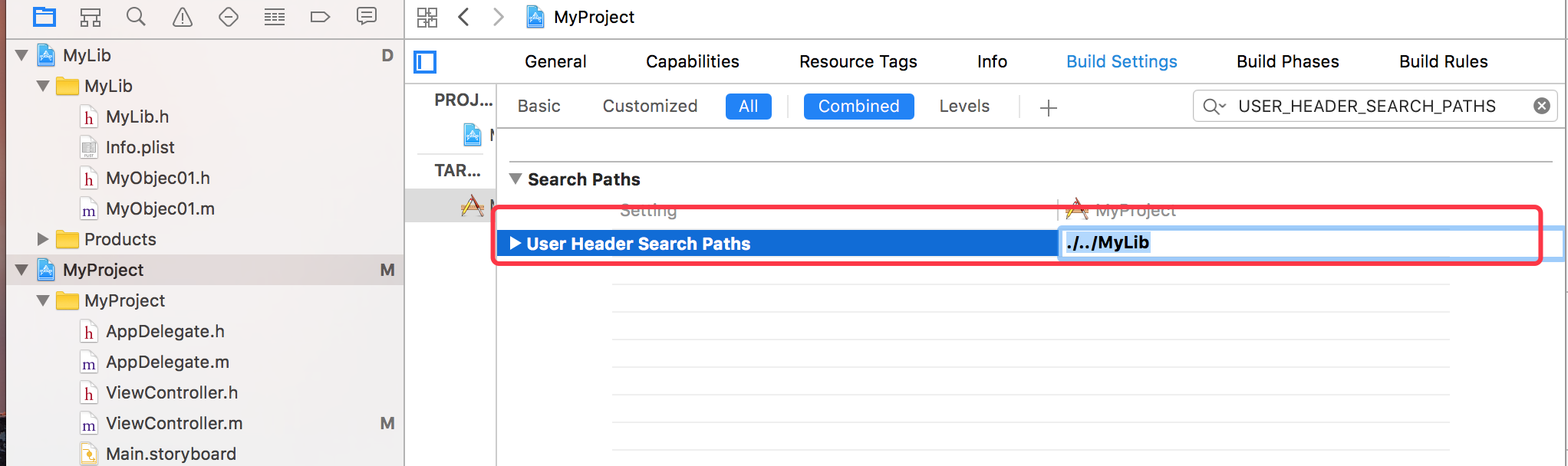
（此处用于引入相关库文件，之后就需要引入该库文件的头文件）



Build Phases ->Link Binary With Libraries



（此处为引入库的头文件，通过引入头文件路径来实现）最好是使用相对路径,系统提供了很多变量如：$(inherited) $(SRCROOT)



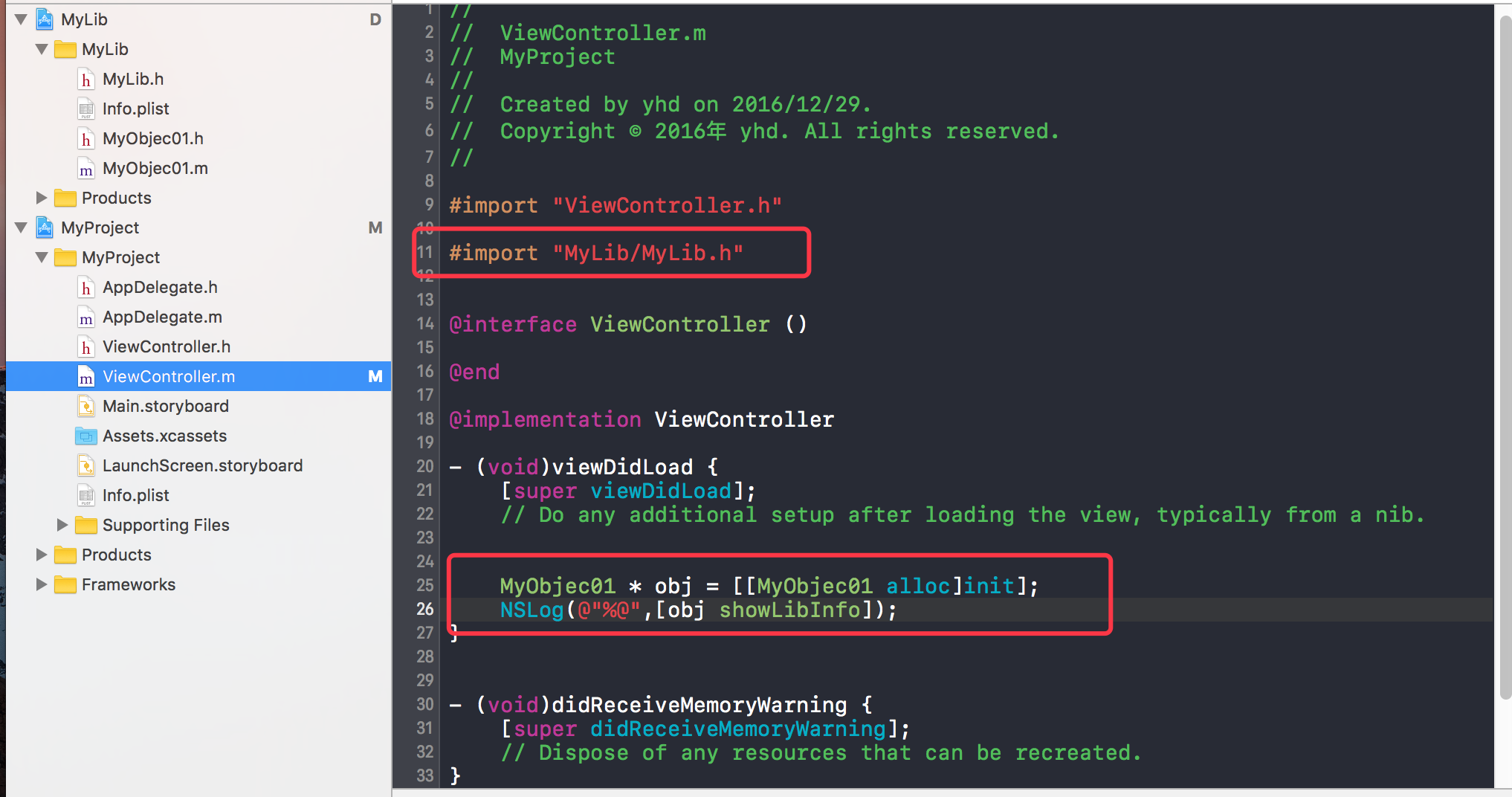
【注】

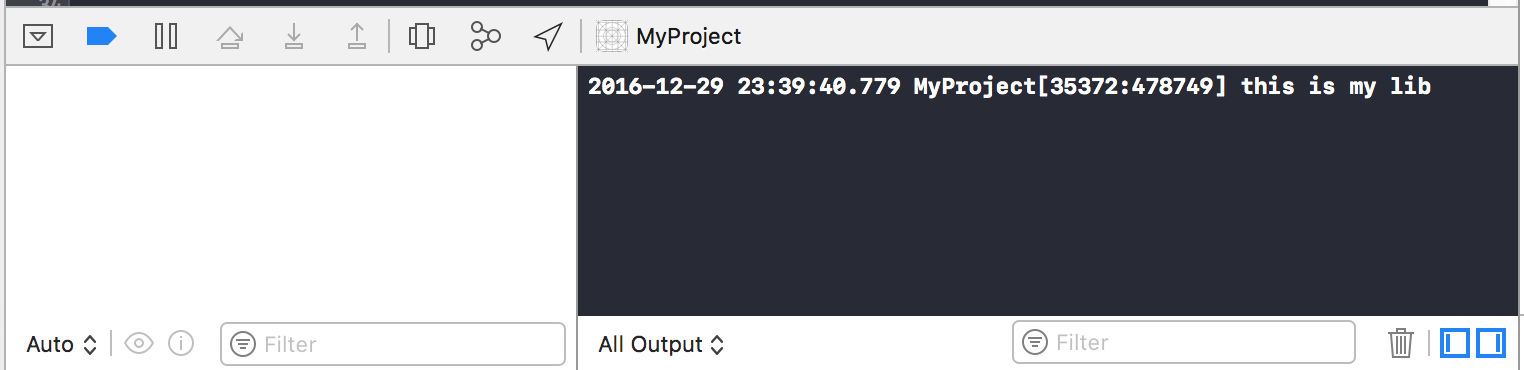
USER\_HEADER\_SEARCH\_PATHS路径,可以写绝对路径和相对路径

$(inherited) $(SRCROOT)

Use the User Header Search Paths for paths you want searched for #include "..." and use the Header Search Paths for #include <...>. Of course, if you check the option to Always Search User Paths, then #include <...> will also work for the user paths.

that $(inherited) can be used to inherit build settings from the project level to the target level. When you define library or header search paths at the project level you can use $(inherited) in the target build settings to use these search paths in the search paths of the project targets





## 1.2 库文件

## 1.3 多个工程依赖

# 2 工程文件

## 2.1 .xcworkspace文件

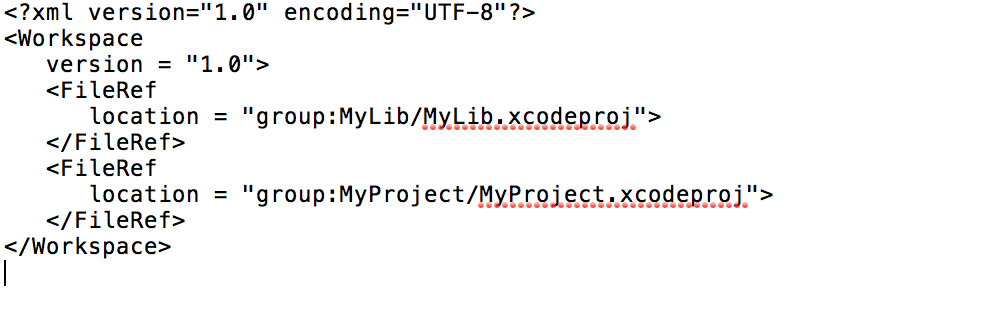
实际是是一个文件夹：.xcworkspace/

-> contents.xcworkspacedata

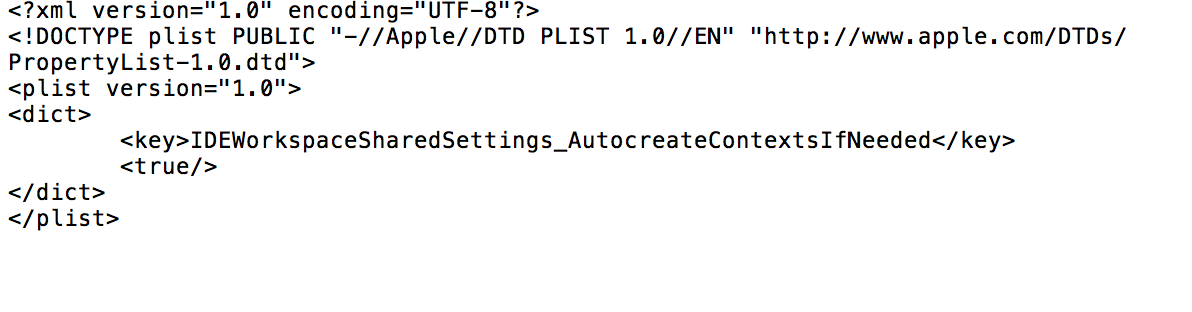
->xcuserdata/.xcuserdatad/UserInterfaceState.xcuserstate

->xcshareddata/WorkspaceSettings.xcsettings

contents.xcworkspacedata文件



WorkspaceSettings.xcsettings文件



## 2.2 .xcodeproj 文件

XXXXX.xcodeproj/

project.pbxproj project.xcworkspace xcshareddata xcuserdata

xcodeproj文件中project.pbxproj 文件解析：

通过cocoPods的实现类的部分注解和具体pbxproj来分析。

http://www.rubydoc.info/github/CocoaPods/Xcodeproj/Xcodeproj/Project/Object/PBXBuildFile

PBXBuildFile:

Contains the information about the build settings of a file

包含工程编译所有文件信息：id和其注解

PBXContainerItemProxy:

Apparently a proxy for another object which might belong another project contained in the same workspace of the project document.

PBXCopyFilesBuildPhase :

Phase that copies the files to the bundle of the target

Note: This phase can appear multiple times in a target.

PBXFileReference:

This class represents a reference to a file in the file system.

PBXFrameworksBuildPhase:

The phase responsible on linking with frameworks. Known as Link Binary With Libraries in the UI.

Note: This phase can appear only once in a target.

包含 工程每个target编译阶段需要的framework文件

PBXGroup:

This class represents a group. A group can contain other groups (PBXGroup) and file references (PBXFileReference)

表示整个工程文件结构，包括文件夹和文件夹中包含的文件

PBXNativeTarget:

Represents a target handled by Xcode.

当前工程所拥有的target,并包含每个target对应各种配置信息

PBXProject :

This class represents the root object of a project document.

表示本工程的根对象，包含了工程相关的信息，如证书，target，buid setting，引用的工程等

PBXReferenceProxy:

Apparently a proxy for a reference object which might belong another project contained in the same workspace of the project document.

This class is used for referencing the products of another project.

PBXResourcesBuildPhase:

Note: This phase can appear only once in a target.

The resources build phase apparently is a specialized copy build phase for resources. Known as Copy Bundle Resources in the UI. It is unclear if this is the only one capable of optimizing PNG.

PBXShellScriptBuildPhase:

Note: This phase can appear multiple times in a target.

A phase responsible of running a shell script (aka Run Script).

表示在Build Phase阶段所添加运行的脚本

PBXSourcesBuildPhase:

Note: This phase can appear only once in a target.

The phase responsible of compiling the files. Known as Compile Sources in the UI.

PBXTargetDependency:

Represents a dependency of a target on another one.

PBXVariantGroup:

This class is used to gather localized files into one entry.

XCBuildConfiguration:

Encapsulates the information a specific build configuration referenced by a XCConfigurationList which in turn might be referenced by a PBXProject or a PBXNativeTarget.

本工程，所有的配置，每套配置都有一个独一无二的ID ，每个配置都含有其对应target信息，构建设置等数据

XCConfigurationList:

The primary purpose of this class is to maintain a collection of related build configurations of a PBXProject or a PBXNativeTarget.

本工程文件下，target配置列表（每个target的配置，中含有具体的配置XCBuildConfiguration ID），每个个target分 debug和release两种配置

## 2.3 target文件

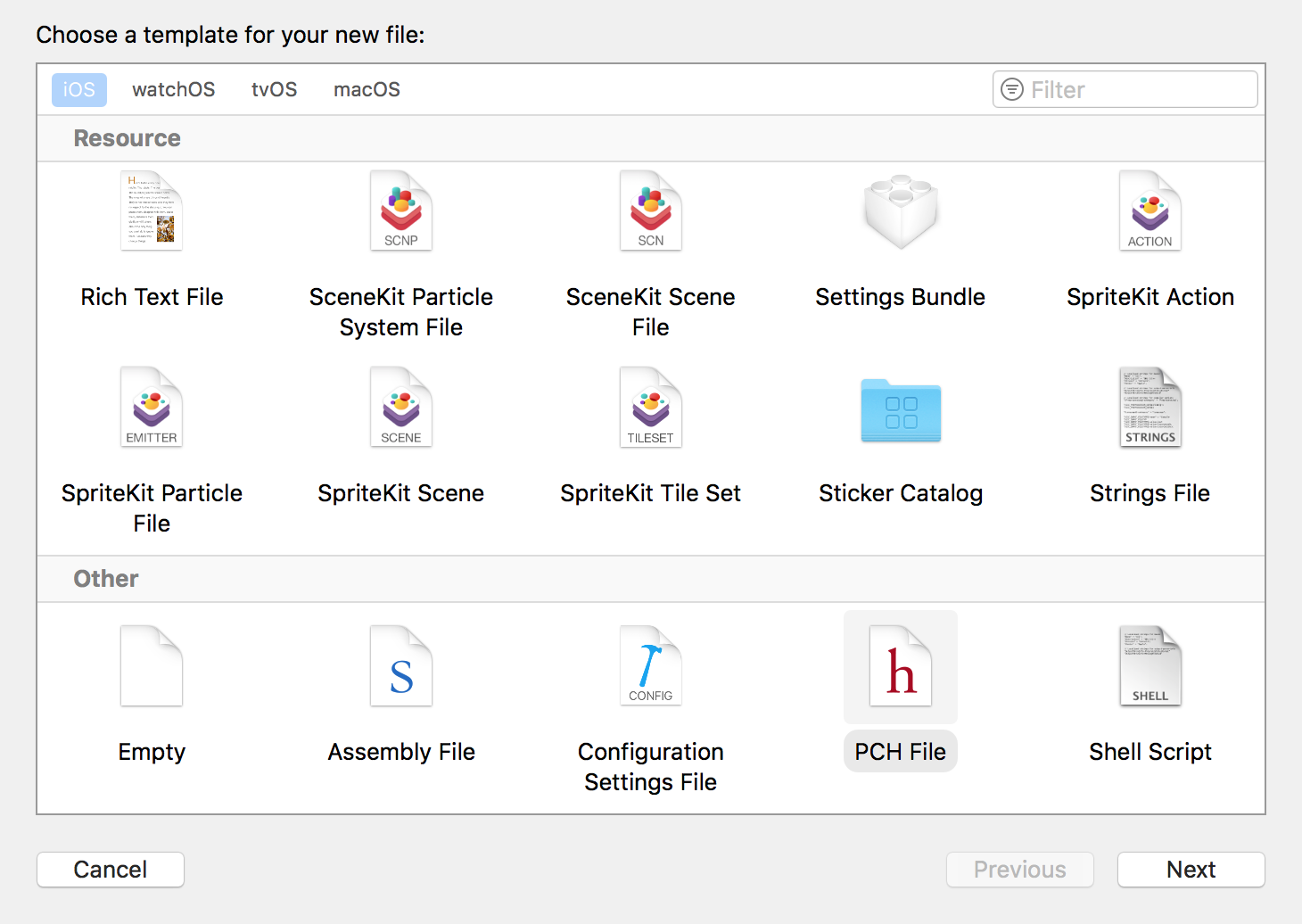
target指定了工程(project)构建的product ，包含构建该product的指令集。Target和project（或workspace）是多对一的关系。每个target可以有自己的环境配置（如，源文件、build setting、build pharmas），但是project只能指定一个target，这个具体可以便是为xcode的scheme。

Workspace中的依赖关系就是基于target的。

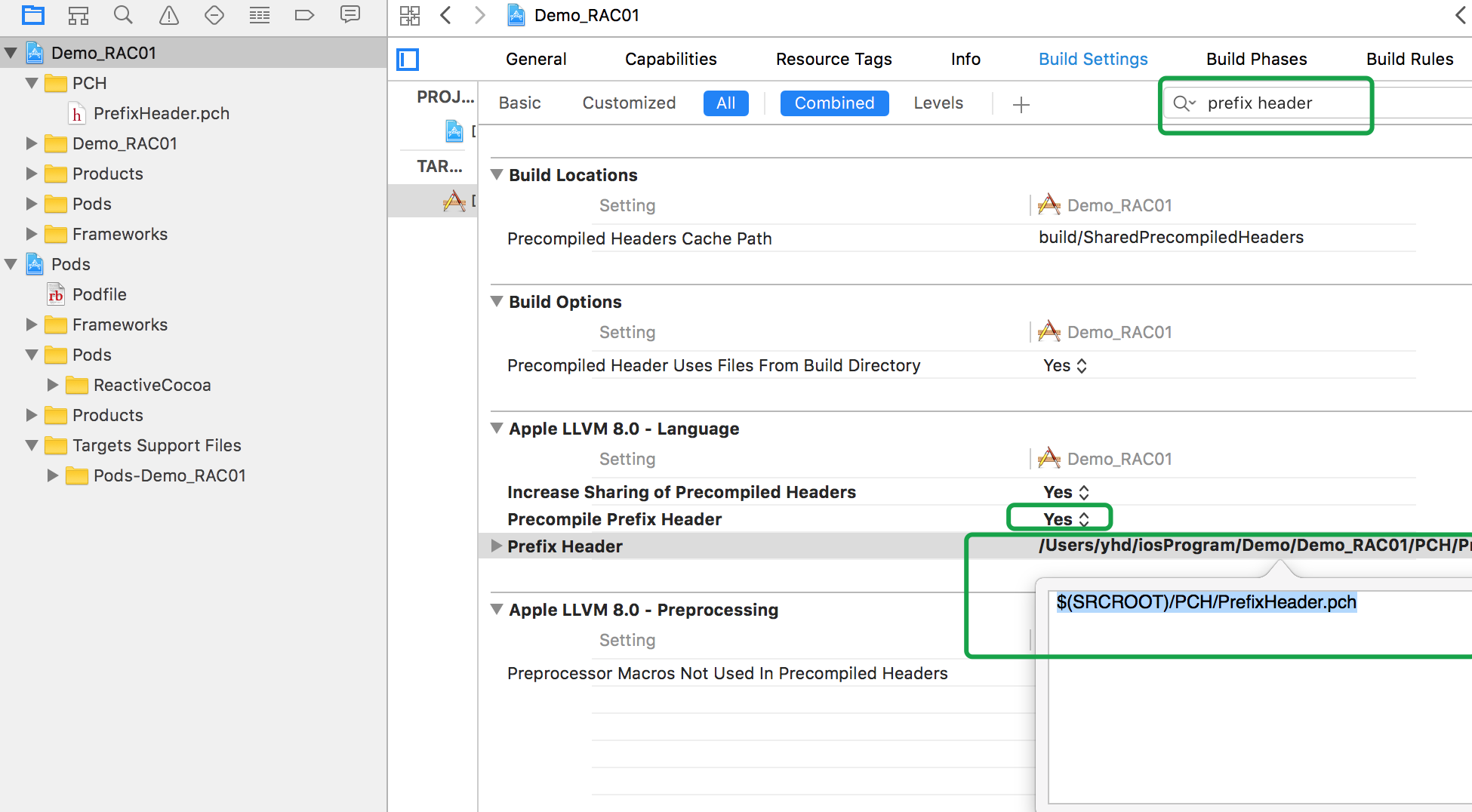
## 2.4 PCH文件

.pch文件在xcode6之前都是在创建项目后自带的，xcode6.0后就取消了。因为用户将大量的头文件、宏、模块开关(日志系统等的开关)等都放在预处理文件中，导致工程编译时间增多，，苹果公司就将.pch默认文件取消了。下面就是具体.pch的创建方式

1. 创建PrefixHeader.pch 文件



2设置build setting



**注意**：在设置Prefix Header的路径时，一定要指出具体嘚.pch文件，而不只是文件夹。$(SRCROOT) 指的就是工程文件的更目录，设置完成后系统会自动转化成绝对路径。

【注：】上文的路径都是值得Finder中的路径，而不是xcode中的Group

## 2.4 Target配置

1 .Library Search Paths VS LinkBinary With Librarys

|  |
| --- |
| [Difference between Header Search Path and Library Search Path?](http://stackoverflow.com/questions/11450540/difference-between-header-search-path-and-library-search-path) |
|  |

# 3创建纯代码的工程

由于xcode高版本取消了穿件空的工程，所以当要创建不含storyboard的工程时需要另外想办法。在xcode8中创建了工程后，将storyboard删掉，并去掉其对应的info.plish文件字段，在Appdelegate中的applicatoin:didFinishLaunchingWithOptions:方法中创建window并设置其rootviewController。实例代码如下：

|  |
| --- |
| - (BOOL)application:(UIApplication \*)application didFinishLaunchingWithOptions:(NSDictionary \*)launchOptions {    self.window = [[UIWindow alloc]initWithFrame:[UIScreen mainScreen].bounds];  [self.window makeKeyAndVisible];  MYLMainViewController \* rootVC = [[MYLMainViewController alloc]init];  rootVC.view.backgroundColor = [UIColor whiteColor];  self.window.rootViewController = rootVC;    return YES;  } |

# 4 CocoaPods

简介：

## 4.1 Podfile文件

## 4.2 现有xcworlspace上添加CocoaPods

【参考】http://nuoerlz.is-programmer.com/posts/47272.html

在当前xcworkspace文件下新建 Podfile。配置事例如下：

|  |
| --- |
| inhibit\_all\_warnings!  # 使用现有的 workspace(DemoWK.xcworkspace), 不要生成新的!  workspace 'DemoWK.xcworkspace'  # 配置默认 xcodeproj, 路径是相对于 Podfile 的路径. 这是必须的, 否则会报Syntax错误.  xcodeproj './DemoApp001/DemoApp001.xcodeproj'  ## 下面就是配置各个Target了! 也就是说, CocoaPods 面向的主配置对象是Target而不  ## 是Project. 当然你得指定该Target属于那个Project, 否则会使用前面配置默认的.  # 配置 DemoKit(Target)  target :DemoKit do  # 该Target属于的工程  xcodeproj 'DemoKit/DemoKit.xcodeproj'  # 平台和系统版本  platform :ios, "8.0"  # UMeng pod  pod 'UMeng', '2.2.1'  end  # 配置 DemoApp001(Target)  target :DemoApp001 do  xcodeproj 'DemoApp001/DemoApp001.xcodeproj'  platform :ios, "8.0"  pod 'UMeng', '2.2.1'  pod "AFNetworking", "~> 2.0"  #pod 'RSTWebViewController', :git => 'https://github.com/rileytestut/RSTWebViewController.git'  pod 'YYKit'  end |

## 4.3 私有podspec

【参考】<http://blog.wtlucky.com/blog/2015/02/26/create-private-podspec/>

搭建私有podspec的几个步骤：

**1 搭建git**

**2 创建私有spec仓库：MallSpecs，并添加到本地pod库**

pod repo add MallSpecs git@10.211.55.3:MallSpecs/MallSpecs.git

**3 创建工程如MallBase**

**4 创建MallBase 的git仓库，并上传代码**

如果下面步骤出现MallBase版本问题则提交tag信息：

$ git tag -m "first release" 0.1.0

$ git push --tags #推送tag到远端仓库

git@10.211.55.3:MyMall/MyMallBase.git

**5 在MallBase工程文件目录创建spec文件：MallBase.spec**

使用命令：pod spec create MallBase git@10.211.55.3:MyMall/MyMallBase.git

忽略警告：pod lib lint --allow-warnings

6 将MallBase.spec推送到spec库中

使用命令：pod repo push MallSpecs MallBase.podspec

忽略警告：pod repo push MallSpecs MallBase.podspec --allow-warnings

**6 在podfile文件中添加私有库源**

source 'https://git.coding.net/wtlucky/WTSpecs.git' # 私有库

添加 pod ‘MallBase’ ,’->0.1.0’

**7 pod install 完成**

【**Error解决**】

1 如果出现， cocoapods diff: /../Podfile.lock: No such file or directory 错误，则需要在当前xcodeproj目录下执行如下命令，解除Pod的一体化管理：

|  |
| --- |
| pod deintegrate |

1. pod lib lint 验证通过，但是 pod repo push 确报错：

- ERROR | file patterns: The `source\_files` pattern did not match any file.

主要原因是因为，podspec文件中的source路径和当前版本不匹配，需要重新设置。如可以更新tag后修改podspec文件中的source即可.

|  |
| --- |
| <http://stackoverflow.com/questions/20915916/podspec-error-source-files-pattern-did-not-match-any-file> |

## 4.4 Git相关

### 4.4.1 配置用户信息

第一个要配置的是用户名称和电子邮件地址。这两条配置很重要，每次 Git 提交时都会引用这两条信息，说明是谁提交了更新，所以会随更新内容一起被永久纳入历史记录：

$ git config --global user.name "John Doe"

$ git config --global user.email johndoe@example.com

如果用了 --global 选项，那么更改的配置文件就是位于你用户主目录下的那个，以后你所有的项目都会默认使用这里配置的用户信息。如果要在某个特定的项目中使用其他名字或者电邮，只要去掉 --global 选项重新配置即可，新的设定保存在当前项目的 .git/config 文件里

# 5 组件化与插件化

# 6 非xcode下的Objective-C

## 6.1 clang 和LLVM

1 预处理

宏替换

内联替换

2 词法分析标记

每一个标记都包含了对应的源码内容和其在源码中的位置

3 解析

解析过后，源程序变成了一棵抽象语法树：一棵代表源程序的树

4 静态分析 ： 一旦编译器把源码生成了抽象语法树，编译器可以对这棵树做分析处理，以找出代码中的错误，比如类型检查：即检查程序中是否有类型错误

5 类型检查 （静态，动态：动态的在运行时做检查，静态的在编译时做检查）

其他分析

6 代码生成

代码编译成 LLVM 字节码（绝大多数情况下是二进制码格式）

7 优化

8 汇编器

9 链接器

10 预处理

【相关阅读】[objc.io 第6期 编译器](https://objccn.io/issue-6-2/)

## 6.2 clang编译oc代码

### 6.2.1 终端编译oc

创建如下oc代码：

|  |
| --- |
| #import <Foundation/Foundation.h>    int main (int argc, const char \* argv[])  {      @autoreleasepool  {        //printf("Hello, World!\n");   //直接写c语言也可以          NSLog(@"Hello, Objective-C!");  }      return 0;  } |

保存为test.m,终端进入其目录执行如下clang命令：

|  |
| --- |
| $ clang -fobjc-arc -framework Foundation test.m -o test |

只编译不链接

|  |
| --- |
| $ xcrun clang –c test.m |

注：**-fobjct-arc** 参数指的是使用ARC特性

**-framework Foundation** 指的是使用Foundation框架

**-o test** 指的是 生成的可执行文件名为test

注意如果报错可能是是应为默认xcode版本与当前clang版本不一致，更新到最新即可

\* 使用xcrun 可以直接在终端中编译.c文件：

|  |
| --- |
| $ xcrun clang helloworld.c  $ ./a.out |

### 6.2.2 clang -rewrite-objc

将oc代码转写成c/c++代码

## 6.3 xcrun

编译.m文件命令：

|  |
| --- |
| $ xcrun clang –c helloworld.m |

查看mach-o文件函数符号

|  |
| --- |
| $ xcrun nm –nm helloworld.o  或  $ xcrun nm –nm a.out |

链接库文件生成可执行文件：

|  |
| --- |
| $ xcrun clang helloworld.o Foo.o -Wl,`xcrun --show-sdk-path`/System/Library/Frameworks/Foundation.framework/Foundation |

红色部分是打印sdk路径的命令可去掉， -Wl是链接指令