# 关于Xcode编译性能优化的研究工作总结

原创 2016年08月12日 18:01:48

标签：xcode (http://so.csdn.net/so/search/s.do?q=xcode&t=blog) / 性能优化 (http://so.csdn.net/so/search/s.do?q=性能优化&t=blog) / 经验 (http://so.csdn.net/so/search/s.do?q=经验&t=blog)

1409

# 关于Xcode编译性能优化的研究工作总结

本文为原创文章，转载注明出处，谢谢！本文链接：关于Xcode编译性能优化的研究工作总结 (http://blog.csdn.net/qq_25131687/article/details/52194034)

近来（8月1—8月12）结合Xcode的官方文档和网上资料经验对Xcode的一些配置选项进行了编译优化的尝试研究，所谓优化主要从编译耗时及编译出的安装包大小进行优化。在研究分析过程中将手上的几个Demo项目进行了编译测试，有Swift项目也有Object-C项目。此外，对于不同配置的相应原理也做了较深入的挖掘分析。

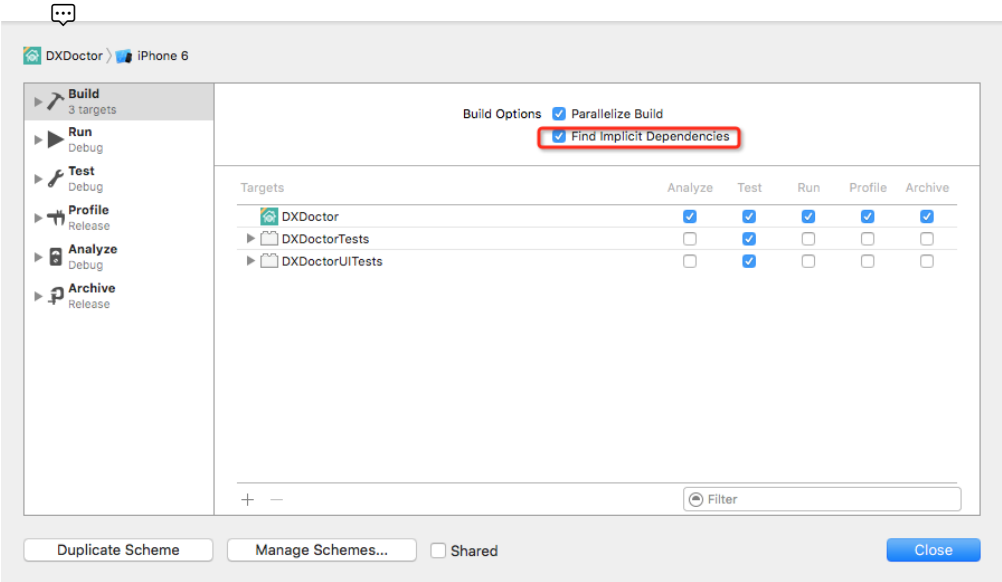
总的来说，对Xcode的Build Setting 进行配置选项的修改是最直接的编译设置。本工作总结除了从Xcode本身的配置进行优化以外，还从外部环境、Xcode插件以及外部硬件配置的编译优化进行了研究分析。

**目录**

十四、Injection for Xcode 高效Xcode编译调试插件
14.1 Injection
14.2 Limitations of Injection(局限性)

# 一、编译时长优化 Swift编译优化 Find Implicit Dependencies

对所编译项目的Scheme进行配置
Product > Scheme > Edit Scheme > Build
Build Opitions选项中，去掉Find Implicit Dependencies.




原理：
选中Find Implicit Dependencies时，编译以下内容：
* 项目所有的文件
* 被修改的frameworks文件
未选中Find Implicit Dependencies时，编译以下内容：
* 项目中必要的文件
* 不会重新编译frameworks文件，即时你对其中的文件做了修改

Test:
对不同设置下（是否选中Find Implicit Dependencies）的项目编译时间进行比较。
注：每次编译前 进行clean操作（shift + command + k），达到消除Xcode自身增量编译带来的干扰。

Result：
对手头的两个demo进行了编译耗时的比较：

| 项目 | 选中Find Implicit Dependencies 耗时 | 未选Find Implicit Dependencies 耗时 |
|---|---|---|
| LoveFreshBeen 爱鲜蜂(Swift) | 23″ | 23″ |
| DXDoctor 丁香医生(Swift) | 12″ | 7″ |

对于两个不同的项目，该配置所带来的编译优化效果并不一定都能体现。


对两个工程的framework文件进行对比之后发现，LoveFreshBeen的framework文件要比DXDoctor的少得多。所以应用该配置时，DXDoctor编译时所反映出来的效果会更明显。

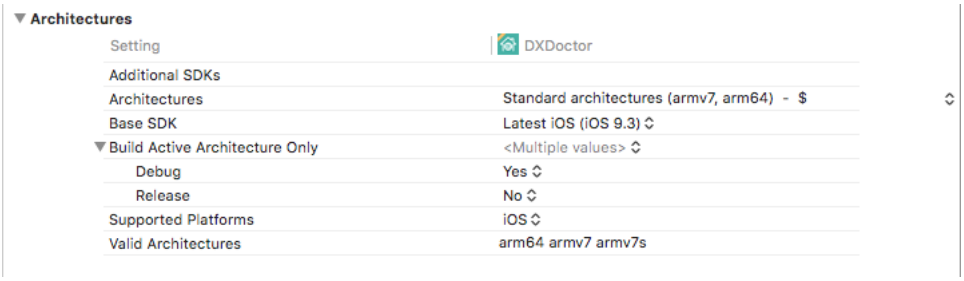
```
缺点分析：
在这个选项（Find Implicit Dependencies）被选中的情况下，即使你只是对项目进行了很细微的改变，项目中的所有资源文件都会被重新编译一遍。也会对所有被改变的frameworks进行编译。没有选中这个选项时，只会对文件中的一些Swift文件进行编译，编译耗时会显著的下降。只是，在这种模式下，你对frameworks中的文件所进行的修改将不会进行重新编译。

结论：
视修改的项目文件的不同，对两种Scheme进行选择，择一使用以提高编译性能。
```

参考：Swift Slow Compile Times Fix (https://forums.developer.apple.com/message/89962#89962)

# 二、编译时长优化 Architectures

在Build Settings中，有个Architectures配置选项。




Architectures

是指定工程支持的指令集的集合，如果设置多个architecture，则生成的二进制包会包含多个指令集代码，提及会随之变大。

Valid Architectures
有效的指令集集合，Architectures与Valid Architectures的交集来确定最终的数据包含的指令集代码。

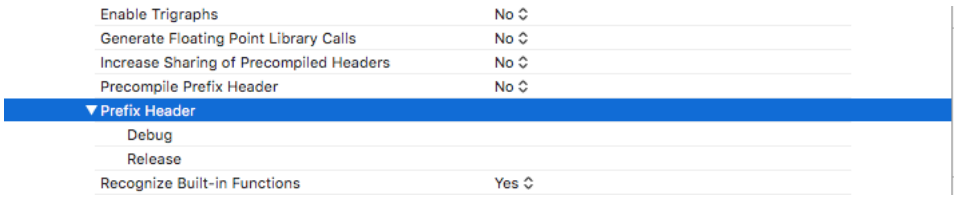
Build Active Architecture Only
指定是否只对当前连接设备所支持的指令集编译，默认Debug的时候设置为YES，Release的时候设为NO。Debug设置为YES时只编译当前的architecture版本，生成的包只包含当前连接设备的指令集代码；设置为NO时，则生成的包包含所有的指令集代码（上述的V艾力达Architecture与Architecture的交集）。所以为了更快的编译速度，Debug应设为YES，而Release应设为NO。

```
注：Debug设置为YES时，如果连接的设备是arm64的（iPhone 5s，iPhone 6（plus）等），则Valid Architecture中必须包含arm64，否则编译会出错。这种模式下编译出来的版本是向下兼容的，即：编译出的armv6版本可在armv7版本上运行。
```

参考：苹果官方“Xcode Build Setting Reference”(https://developer.apple.com/library/mac/documentation/DeveloperTools/Reference/XcodeBuildSettingRef/1-Build_Setting_Reference/build_setting_ref.html#//apple_ref/doc/uid/TP40003931-CH3-SW85)
关于Xcode “Build Setting”中的Architectures详解 (bengyuejiejie.github.io/blog/2015/03/09/first-blog/)

# 三、编译时长优化 Precompile Prefix Header 预编译头文件

Build Setting > Apple LLVM 7.1 – Language




Xcode 6及之后版本默认不使用pch文件参与项目编译，原因有二：
* 去掉自动导入的系统框架类库的头文件们可以提高源文件的复用性，便于迁移；
* 一个庞大的Prefix Header会增加Build耗时。

但对于原有项目应用了pch文件的情况，就需要对Xcode的Build Setting进行配置以使用pch。

当Precompile Prefix Header设为NO时，头文件pch不会被预编译，而是在每个用到它导入的框架类库中编译一次。每个引用了pch内容的.m文件都要编译一次pch，这会降低项目的编译速度。

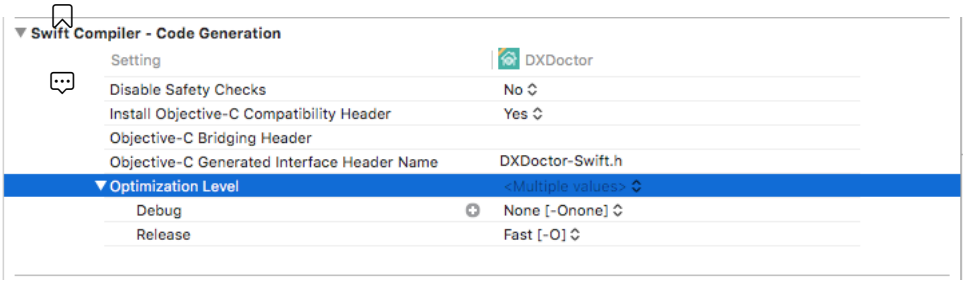
将Precompile Prefix Header设为YES时，pch文件会被预编译，预编译后的pch会被缓存起来，从而提高编译速度。
需要编译的pch文件在Prefix Header中注册即可。

手动创建pch文件： xcode6中如何添加pch文件(blog.csdn.net/lihuiqwertyuiop/article/details/39268101)
参考 ： Xcode Precompile Prefix Header浅析 (blog.csdn.net/jymn_chen/article/details/39314163)

Why isn't ProjectName-Prefix.pch created automatically in Xcode 6? (stackoverflow.com/questions/24158648/why-isnt-projectname-prefix-pch-created-automatically-in-xcode-6)


# 四、编译时长优化 Swift Compile – Code Generation Optimization Level

Build Setting > Swift Compile – Code Generation > Optimization Level




Swift 编译优化选项，对手上的两个demo进行了以下测试：

| | Debug Release | None [-Onone] Fast [-O] | Debug Release | Fast[-O] None | Debug Release | Fast[-O] Fast[-O] |
|---|---|---|---|---|---|---|
| LoveFreshBeen 爱鲜蜂(Swift) | 23″ | | 48″ | | 50″ | |
| SmallDay 小日子(Swift) | 14″ | | 20″ | | 20″ | |

Debug None[-Onone] Release Fast[-O] 是Xcode在Debug模式下编译Swift项目的最优选项，通过测试可以看出，在默认配置情况下和自定义情况下的编译耗时存在比较明显的差异。

# 五、加载RAM磁盘编译Xcode项目

DerivedData

Xcode会在文件系统中集中的缓存临时信息。

每次对Xcode iOS项目进行clean、build或者在iOS虚拟机上launch，Xcode都会在DeriveData文件夹中进行读写操作。换句话说，就是将Derived Data的读写从硬盘移动到内存中。
DeriveData文件夹中包含了所有的build信息、debug- 和 release- built targets以及项目的索引。当遇到零散索引（odd index）问题（代码块补全工作不正常、经常性的重建索引、或者运行项目缓慢）时，它可以有效地删除衍生数据。删除这个文件夹将会导致所有Xcode上的项目信息遭到破坏。

Step 1
将DeriveData下的文件删除：

```
1 rm -rf ~/Library/Developer/Xcode/DerivedData/*
```

删除的这些数据，Xcode会在Build时重新写入的。

Step 2
在~/Library/Developer/Xcode/DerivedData.上部署安装2 GB大小的RAM磁盘。
进到~/Library/Developer/Xcode/DerivedData.

```
cd ~/Library/Developer/Xcode/DerivedData
```

创建2 GB的RAM磁盘（size的计算公式 size = 需要分配的空间(M) * 1024 * 1024 / 512）：

```
hdid -nomount ram://4194304
```

此行命令后将会输出RAM磁盘的驱动名字：/dev/diskN（N为数字）。

初始化磁盘：

```
newfs_hfs -v DerivedData /dev/rdiskN
```

有以下输出：
Initialized /dev/rdisk3 as a 2 GB case-insensitive HFS Plus volume

安装磁盘：

```
diskutil mount -mountPoint ~/Library/Developer/Xcode/DerivedData /dev/diskN
```

这会在已存在的DeriveData上安装一个卷，用于隐藏旧的文件。这些文件仍会占据空间，但在移除RAM磁盘之前都无法访问。

在重启或从Finder中弹出RAM磁盘时，磁盘中的内容将会消失。下次再创建磁盘时，Xcode将会重新构建它的索引和你的项目中间文件。

创建虚拟磁盘后, 并不是直接占用掉所有分配的空间, 而是根据虚拟磁盘中的文件总大小来逐渐占用内存.
注：如果创建的虚拟磁盘已满, 会导致编译的失败. 此时清除掉Derived Data后重新编译, 就算有足够的空间也还是有可能会导致编译失败. 重启Xcode可以解决此问题.

对手头Demo进行编译测试，由于编译本身读写内容较少，耗时较短，都在10s到20s之内，所以提速感觉不明显，在1s到2s间（10%左右），也许应用到较大的项目中会有比较好的体现。

参考： Reduce XCode build times (https://coderwall.com/p/1p4mha/reduce-xcode-build-times)
【iOS Tip】提高Xcode编译速度 (nszzy.me/2016/03/20/reduce-xcode-build-times/)

# 六、编译线程数和Debug Information Format

## 6.1、 提高XCode编译时使用的线程数

```
defaults write com.apple.Xcode PBXNumberOfParallelBuildSubtasks 8
```

其后的数字为指定的编译线程数。XCode默认使用与CPU核数相同的线程来进行编译，但由于编译过程中的IO操作往往比CPU运算要多，因此适当的提升线程数可以在一定程度上加快编译速度。

## 6.2、 将Debug Information Format改为DWARF

在工程对应Target的Build Settings中，找到Debug Information Format这一项，将Debug时的DWARF with dSYM file改为DWARF。

这一项设置的是是否将调试信息加入到可执行文件中，改为DWARF后，如果程序崩溃，将无法输出崩溃位置对应的函数堆栈，但由于Debug模式下可以在XCode中查看调试信息，所以改为DWARF影响并不大。

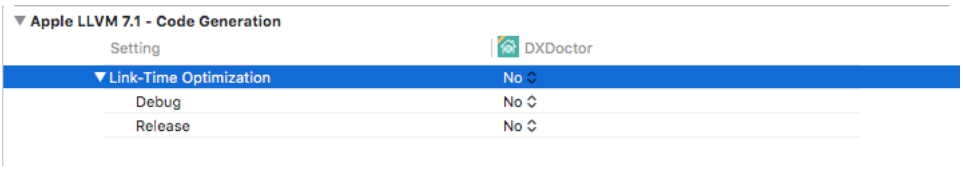
需要注意的是，将Debug Information Format改为DWARF之后，会导致在Debug窗口无法查看相关类类型的成员变量的值。当需要查看这些值时，可以将Debug Information Format改回DWARF with dSYM file，clean（必须）之后重新编译即可。

```
注：6.2 的解决方案为xcode的默认设置，进行反向设置时，编译速度改变不大；
```

参考：To speed up the XCode compile and link speed (200%+) (www.programering.com/a/MTN1ATNwATQ.html)

# 七、Link-Time Optimizations 链接时优化

Apple LLVM 7.1 – Code Generation Link-Time Optimization




Link-Time Optimization执行链接时优化（LTO）。在Clang/LLVM领域，这意味着链接器获得的是LLVM字节码，而不是通常的目标文件。这些字节码在一种更抽象的层次上代表程序这里写链接内容(www.pc6.com/infoview/Article_61969.html)的执行过程，允许LTO得以进行，但是坏处是，仍然需要将他们转换成机器代码，在链接时需要额外的处理时间。

参数设为YES时，能够优化链接时间；目标文件以LLVM二进制文件格式存储，在链接期间，优化了整个程序。

将其设为NO时，可以减少Link阶段的时间。对于Link阶段耗时较长的项目，整体编译优化体现较为明显。

# 八、加装SSD固态硬盘

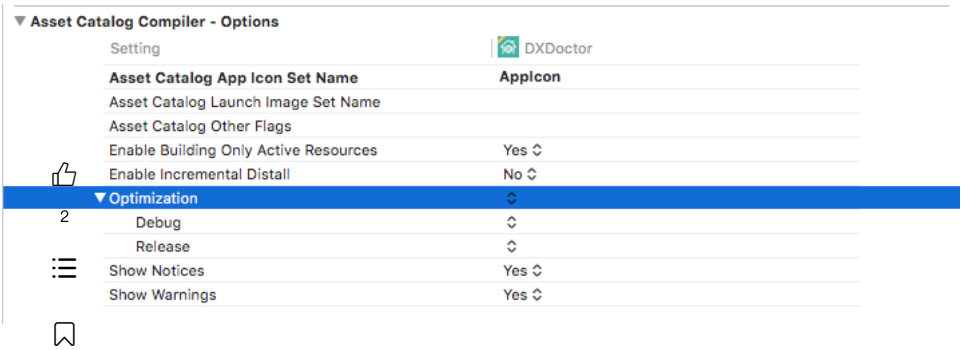
固态硬盘传输速度能达到500MB/s，其中读取速度达到400-600MB每秒，写入速度达到200MB每秒。而传统硬盘读取速度极限也无法超越200MB每秒，写入速度在100MB每秒左右。如果遇到非连续的散片数据，SSD能体现出极快的读写速度。而传统机械硬盘由于磁头寻道等原因，传输速度偏慢。

SSD加快了程序的I/O速率，读写速度比普通硬盘快，从而提升Xcode的编译速度。受限于各种硬件原因，没有进行测试。

参考： SSD 硬盘能否明显加快编译的速度？ (https://www.v2ex.com/t/64236)
固态硬盘的好处，固态硬盘与普通硬盘的区别 (www.pc6.com/infoview/Article_61969.html)

# 九、安装包大小优化 Asset Catalog Compiler – Options Optimization

Build Setting > Asset Catalog Compiler – Options





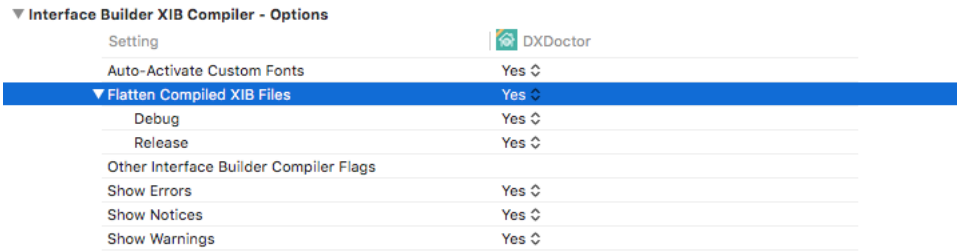

在Optimization 优化设置项有三个选项，不指定、time和Space。

| | Optimization nothing | Optimization space | Optimization time |
|---|---|---|---|
| LoveFreshBeen 爱鲜蜂（Swift） | 23″ app size 28.6M | 23″ app size 26.7M | 23″ app size 28.6M |
| Hardest Game （Object - C） | 13″ app size 28.6M | 13″ app size 24.8M | 13″ app size 28.6M |

Optimization nothing是Xcode默认的设置。
与预想的不同，在选择Optimization time 时，编译时长并没有得到优化。
但在Optimization space时，编译耗时基本没有波动，但编译生成的app 大小有不小程度的优化。

# 十、安装包大小优化 Flatten Compiles XIB Files

是否扁平化编译XIB文件。




官方解释是：指定是否在编译时剥离nib文件以优化它们的大小，设为YES时编译出来的nib文件会被压缩但是不能编辑。

> Description: Boolean value. Specifies whether to strip a nib files to reduce their size. The resulting nib file is more compact but is not editable.

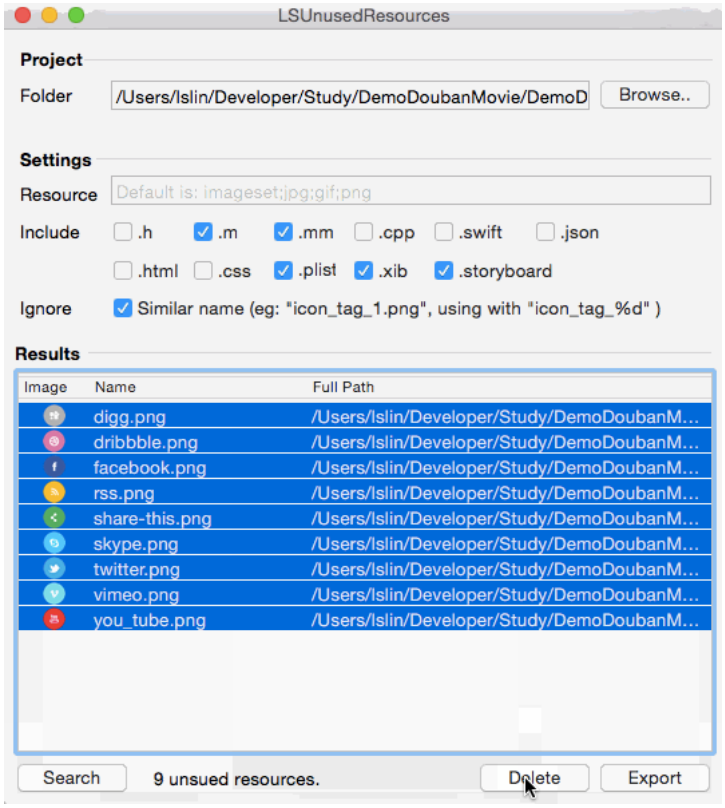
| Flatten Compile XIB Files | YES | NO |
|---|---|---|
| LoveFreshBeen 爱鲜蜂（Swift） | 28.7M | 28.9M |
| SmallDay 小日子（Swift） | 22.5M | 22.8M |
| DXDoctor 丁香医生（Swift） | 24.4M | 24.5M |
| HardestGame (Object-C) | **28.6M** | **28.8M** |
| HuntForCity 城觅（Object-C） | **16.7M** | **16.9M** |

测试app大小的同时也对编译耗时进行了测试，在两种编译模式下的编译耗时基本没有变化。

参考：What does the “Flatten compiled xib files” project build option “do” (stackoverflow.com/questions/5782868/what-does-the-flatten-compiled-xib-files-project-build-option-do)
官方文档 Interface Builder Compiler Build Settings (https://developer.apple.com/library/prerelease/content/documentation/DeveloperTools/Reference/XcodeBuildSettingRef/1-Build_Setting_Reference/build_setting_ref.html)



## 十四、安装包大小优化 清理未被使用的图片资源 LSUnusedResources




项目的开发过程总是会经历较长期的迭代，不断的添加功能的同时会引入大量的图片资源。需求变更、业务逻辑修改等需要移除某些功能模块时就会导致这些前期加入的图片资源问价被忽略而遗留在编译的安装包中，长此以往会使得安装包变得格外臃肿。特别是类似于手Q项目的开发，开发人员多，产品迭代频繁，开发时间紧俏，开发人员轮换等特点更有可能导致这样的后果。

一个较为传统的清理方法时将图片资源的文件名一一复制粘贴到Xcode的全局变量查找中去查找该字符串，如果返回的结果为零，则该资源很有可能没被使用。之所以是“很有可能”，是因为在代码中，资源有时是通过字符串拼接的方式进行引用的。

在这里提供一个github上的开源工具 LSUnusedResources ，这个工具是对github上的另一个开源工具Unused的优化改进（匹配速度、结果准确性），作者针对源码、Xib、Storyboard 和 plist 等文件，先全文搜索其中可能是引用了资源的字符串，然后用资源名和字符串做匹配，从而找出未被使用的资源，比Unused的查找速度要快得多。

使用起来也比较简单：

1、将工程目录路径拷贝到Folder或通过Browse浏览文件目录；

2、在Resource指定要查找的资源类型；

（经过本人测试，发现该工具在未指定Resource类型时所查找出来的资源不是很准确，列举出 的资源事实上是正在使用的，所以我在测试时指定查找了png类型的文件。）

3、单击Search以查阅结果。

```
1 注：为了避免对资源的误删操作，建议在该工具输出结果后对结果中的资源名复制并在Xcode的全局查找中进行校验。
```

下载安装：LSUnusedResources.app.zip (https://github.com/tinymind/LSUnusedResources/raw/master/Release/LSUnusedResources.app.zip)

Github地址：LSUnusedResources (https://github.com/tinymind/LSUnusedResources)

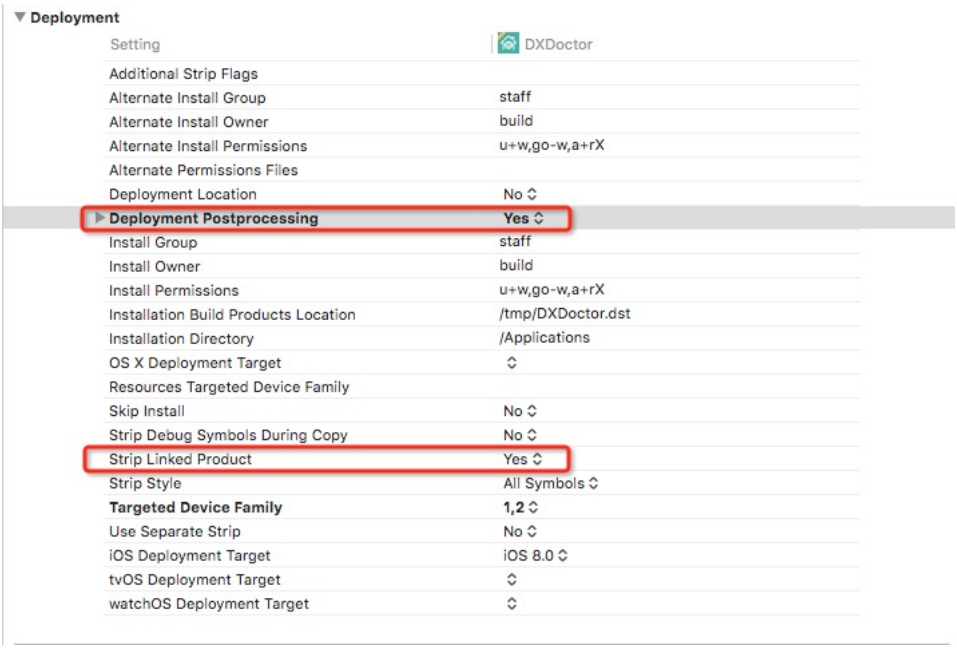
参考链接：查找XCode工程中没被使用的图片资源 (blog.lessfun.com/blog/2015/09/02/find-unused-resources-in-xcode-project/)



# 十二、安装包大小优化 Deployment Postprocessing和Strip Linked Product

Xcode中Strip Linked Product 的默认设置为YES，但是Deployment Postprocessing的默认设置为NO。在Deployment Postprocessing 是Deployment的总开关，所以在打开这个选项之前 Strip Linked Product 是不起作用的。

```
1 注：当Strip Linked Product设为YES的时候，运行app，断点不会中断，在程序中打印[NSThread callStackSym
2 bols]也无法看到类名和方法名。而在程序崩溃时，函数调用栈中也无法看到类名和方法名。
```



打开这两个选项之后进行编译，编译出的安装包大小有了较大程度的优化：

| | Deployment Postprocessing NO<br>Strip Linked Product YES | Deployment Postprocessing YES<br>Strip Linked Product YES |
|---|---|---|
| LoveFreshBeen<br>爱鲜蜂（Swift） | 28.5M | **25.4M** |
| SmallDay<br>小日子（Swift） | 22.5M | 20.3M |
| DXDoctor<br>丁香医生（Swift） | 24.4M | 23.7M |
| HardestGame<br>三（Object-C） | **26.8M** | **26M** |
| HuntForCity<br>城（Object-C） | **16.7M** | **15.5M** |

参考：Xcode中和symbols有关的几个设置 (www.voidcn.com/blog/skylin19840101/article/p-6112678.html)

# 十三、安装包大小优化 Linking->Dead Code Stripping

将Dead Code Stripping 设置为YES 也能够一定程度上对程序安装包进行优化，只是优化的效果一般，对于一些比较小的项目甚至没有什么优化体现，所以这里也就没有上测试数据。

Dead Code Stripping 是对程序编译出的可执行二进制文件中没有被实际使用的代码进行Strip操作。

对于更深层次的解读，在参考链接的文章里有详细描述。
参考：Dead Code Stripping (https://garbageout.wordpress.com/2015/03/24/dead-code-stripping/)

# 十四、Injection for Xcode 高效Xcode编译调试插件

## 14.1 Injection

github上的开源项目，Xcode插件。

对于iOS开发者来说，XCode有个另人十分难耐的特性——编译时长的问题。也许工作的时候你能够为自己找到一个闲下来喝杯咖啡的正当的借口，然而，多次的调试编译过程足以让你喝上好多杯咖啡了。应该说，Injection是iOS开发者的福音，它在很大程度上优化了XCode的性能，提升了开发者的工作效率。

Injection能够在app运行时动态地向Swift或者OC文件注入新代码并且即时地呈现在运行中的模拟器的app上，从而达到提高程序编译速度，提高开发效率的目的。开发者不需要重新编译重新运行整个项目，这样的优化使得编译周期从7秒缩短至1秒。从XCode的输出台来看，每次在进行代码注入之后都只会编译被注入了代码的文件。这么一听有点类似于增量编译。

设想这样一个场景，对于一个编译启动需要10分钟的项目，如果你想对某个功能的动画效果进行微调，是否意味着你需要以至少10分钟为一个调试周期去对你的改动进行测试，而injection则能够在程序运行时动态的改动方法实现，并呈现在模拟器或真机上。

Injection Github https://github.com/johnno1962/injectionforxcode
(https://github.com/johnno1962/injectionforxcode)；
或者到这里去看看他的演示：https://www.youtube.com/watch?v=uftvtmyZ8TM
(https://www.youtube.com/watch?v=uftvtmyZ8TM)；

对于Injection的安装使用，可以到第一个链接里下载package。Injection团队为开发者准备了一套傻瓜式的配置流程，基本上都是单击continue就行了，然后重启你的Xcode。装成功后你会看到product > injection plugin。此时你应该已经装成功了。点击 Product >Injection Plugin > Patch Project for Injection 选项，之后插件会在main.m 中插入两段代码。

```
#ifdef DEBUG
static char _inMainFilePath[] = __FILE__;
static const char *_inIPAddresses[] = {"10.12.1.67", "127.0.0.1", 0};

#define INJECTION_ENABLED
#import "/tmp/injectionforxcode/BundleInjection.h"
#endif
```

这不会影响程序原有代码，如果要还原，随时可以通过点击 Revert Injection's Changes 选项来还原。你可以开搞了。

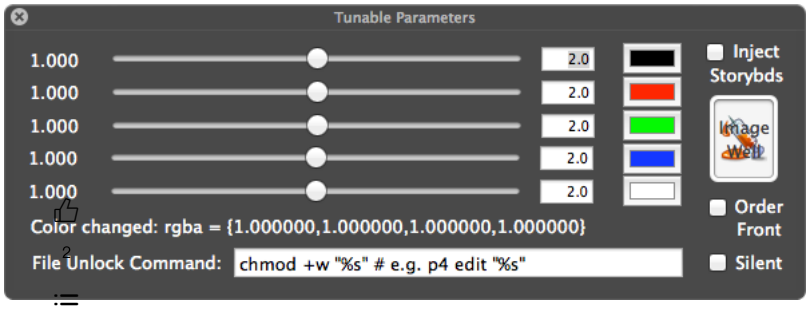
用一个demo做实验，将project运行起来，在运行时对你的代码进行改动，可以使用快捷键Ctrl + =快速运行。也可以在 product > injection plugin > inject and reset app。 你会发现你改动的代码所在类的左上角有一个蓝色的进度条，一秒不到的时间就能够完成注入并运行在你的app上。当然，你也能够在你改动的代码的方法里边加上一个断点，快捷键Ctrl + = ，你会发现运行时会停在你设定的breakpoint上。

对于Swift文件injection好像还不能做到完美支持，github上有相关的解释，我还没有深入的尝试，有兴趣的童鞋可以去看看，顺便交流交流。
injection是Xcode IDE的一个扩展，允许你去对类的一个方法实现打补丁而不需要重启app。官方的原理如下：

> It performs this by parsing the build logs of the application to determine how a source file was last compiled. With this it wraps the result of re-compiling into a bundle which is injected into the application using the dynamic loader. At this point there are two versions of a class in the app, the original and a new modified version from the bundle. The modified version is then "swizzled" onto the original class so changes take effect.

（个人翻译）它通过解析程序的编译日志来确定最后一次编译的源文件。通过动态加载程序把重新编译的结果打包到被注入代码的app中。此时有两个版本的类应用,最初的和一个新的修改版本的包。这个修改后的版本,被"swizzled到"原始类中而生效。

除此之外，injection插件还有一个参数调节器Tunable Parameters，对于UI开发来说是个利器。比如对颜色的确定，对字体大小的界定等等。运行app，然后对参数进行修改就能够动态的进行调试了。直观而且方便。






对于 Tunable Parameters的使用我还没有涉足，它的使用目前仅限于Swift项目，还需要在项目中进行一些诸如添加头部代码的配置，有兴趣的童鞋可以到这里了解：
https://github.com/johnno1962/injectionforxcode/blob/master/documentation/tunable_parameters.md
(https://github.com/johnno1962/injectionforxcode/blob/master/documentation/tunable_parameters.md)
其实也不复杂，就是在新建一个main.m文件之后加上几行代码。
在使用injection时，一个新的Xcode项目文件将会在原本项目的文件里生成（iOSInjectionProject或OSXInjectionProject）。这个文件是用于存放那些被injecte的项目文件的，建议将其加入到.gitignore中，直接忽略。

每一次的项目文件被injected，在injection项目目录里的injectionCount.txt中的数字就会增加。它可以很直观的告诉你通过injection进行了多少的文件改动。

如果你想在真机或Appcode上进行测试：
你需要做一些轻量级的配置：在你的main.m文件加上如下几行代码：

```
#ifdef DEBUG
static char _inMainFilePath[] = __FILE__;
static const char *_inIPAddresses[] = {"10.12.1.67", "127.0.0.1", 0};

#define INJECTION_ENABLED
#import "/tmp/injectionforxcode/BundleInjection.h"
#endif
```

这个配置也可以通过Product > Injection Plugin > Patch Project For Injection 来进行自动配置。对于Swift文件，你需要添加一个空的main.m文件来完成配置。

至于使用Appcode的盆友，可以上github上看看教程：
https://github.com/johnno1962/injectionforxcode
(https://github.com/johnno1962/injectionforxcode)

## 14.2 Limitations of Injection(局限性)

贴github原文：

There are limitations of course, largely centering around static variables, static or global functions and their Swift equivalents. Consider the following Objective–C code.

```
#import "Injectable.h"

void dispatch_on_main( void (^block)(void) ) {
    dispatch_async( dispatch_get_main_queue(), block );
}

static id sharedInstance;

@implementation Injectable

+ (instancetype)sharedInstance {
    static dispatch_once_t once;
    dispatch_once( &once, ^{
        sharedInstance = [[self alloc] init];
    } );
    return sharedInstance;
}

- (void)injected {
    NSLog( @"injected: %@", self );
}

- (void)doSomething {
    dispatch_on_main( ^{ NSLog( @"main queue: %@", self ); } );
}

@end
```

- One potential problem is when the new version of the class is loaded, it comes with it's own versions of static variables such as sharedInstance and the once token. After injection has occurred, this would generate a new singleton instance.

To prevent this, class methods with the prefix "shared" are not swizzled on injection to support this common idiom.

- It can be tough to look through all of the memory of a running application. In order to determine the classes and instances to call the injected callbacks on, Injection performs a "sweep" to find all objects in memory. Roughly, this involves looking at an object, then recursively looking through objects which it refers to. For example, the object's instance variables and properties.

  This process is seeded using the application's delegate and all windows. Once all the in–memory reference are collected, Injection will then filter these references to ones that it has compiled and injected. Then sending them the messages referenced in the callbacks section.

  If no references are found, Injection will look through all objects that are referred to via sharedInstance. If that fails, well, Injection couldn't find your instance. This is one way in which you may miss callbacks in your app.
- The function dispatch_on_main does not inject, as it has been statically linked into the application. It does however, inject by proxy in the case shown via the doSomething method. dispatch_on_main will have been linked locally to a version in the object file being injected.





injection作为Xcode的插件，还是有局限性的。
injection的作用域主要集中在静态变量、静态或全局函数及其Swift的当量（按：Swift equivalents）。

以下是作者贴的示例代码：

```
#import "Injectable.h"

void dispatch_on_main( void (^block)(void) ) {
    dispatch_async( dispatch_get_main_queue(), block );
}

static id sharedInstance;

@implementation Injectable

+ (instancetype)sharedInstance {
  2 static dispatch_once_t once;
    dispatch_once( &once, ^{
        sharedInstance = [[self alloc] init];
    } );
    return sharedInstance;
}

-(void)injected {
    NSLog( @"injected: %@", self );
}

-(void)doSomething {
    dispatch_on_main( ^{ NSLog( @"main queue: %@", self ); } );
}

@end

```

1）有一个潜在的问题，当类的新版本被加载，它带有自己的静态变量版本sharedInstance和once标记。发生injected后，将产生一个新的单一实例。

```
  To prevent this, class methods with the prefix "shared" are not swizzled on injection to
support this common idiom.
  以上这句我捉摸了很久还是没有吃透。
```

2）它可以浏览所有的正在运行的应用程序的内存。为了确定类和实例能够调用injectied的回调，injection会执行一次“扫描”，找到在内存中的所有对象。粗略说，这涉及了一个对象，然后通过递归寻找它所指向的对象。例如对象的实例变量和内容（properties）。

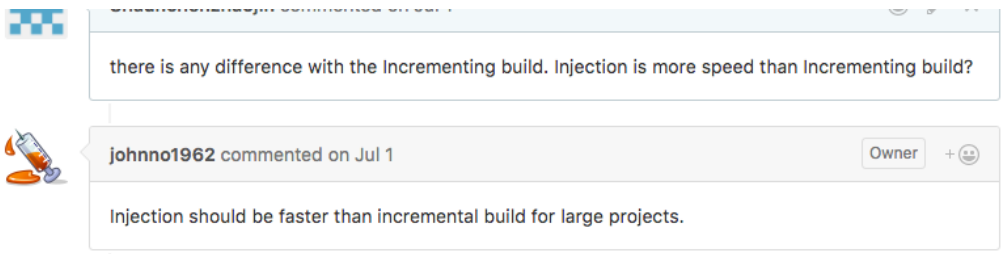
3）This process is seeded using the application’s delegate and all windows.（按：这个过程通过应用程序的代理和所有的窗口）。一旦所有在内存中的引用被收集，injection将会过滤这些它已经编译和injected的引用，。然后再将被引用信息的回调部分发送出去。

4）如果没有找到引用注入的内容，Injection将通过sharedInstance查找所有被涉及到的对象。如果没有找到任何对象，那么，Injection将找不到你的实例。这会导致你无法在你的app中进行回调函数的调用。

5）函数dispatch_on_main无法被injected，因为它已被静态地链接到应用程序。但是，injection可以通过代码示例里的doSomething方法进行inject。dispatch_on_main将会被链接到本地的在被injected对象文件的一个新版本中。

以上内容参考：https://github.com/johnno1962/injectionforxcode (https://github.com/johnno1962/injectionforxcode)

对于某些童鞋的疑问：injection的编译效率与XCode自身的增量编译有什么优势？我已经在github上Issue了作者并得到了如下回复：




但是具体到底能够提升多少，这个有待进一步的测试。


**swift 影响Xcode编译速度的注意事项 (http://blog.csdn.net/qq_29846663/article/details/...**

、、、 首先尝试次方发，看起来很有效 背景 随着 Xcode8 和 swift3.0 的正式到来，我开始着手将 swift2.3 的项目转到 swift3 （至于转换过程，这里不多做...

qq_29846663 (http://blog.csdn.net/qq_29846663) 2017年06月28日 11:01 714

**不使用cornerRadius设置图片圆角 (http://blog.csdn.net/huangyongf/article/details/5247...**

不使用cornerRadius设置图片圆角

huangyongf (http://blog.csdn.net/huangyongf) 2016年09月08日 22:33 3224

**物元分析的理论框架与应用_可拓集合和不相容问题_发表五年来研究工作的综...**

PDF (http://downloa... 2012年12月07日 17:14 642KB 下载

**中科院计算所在可信大数据软件技术方面的研究工作【DOC+PPT下载】 (http://blog.csdn.ne...**

http://pan.baidu.com/s/1qWOCMxm 清单： 中科院计算所在可信大数据软件技术方面的研究工作.doc 【本文doc文档】 中科院计算所在可信大数据软件技术方面的研究工作.pp...

hsluoyc (http://blog.csdn.net/hsluoyc) 2015年02月13日 21:51 4035

**机器视觉和图像处理方面的研究工作应该知道 (http://blog.csdn.net/lxy_2011/article/details...**

作机器视觉和图像处理方面的研究工作，最重要的两个问题：其一是要把握住国际上最前沿的内容；其二是所作工作要具备很高的实用背景。解决第一个问题的办法就是找出这个方向公认最高成就的几个超级专家(看看他们都在...

lxy_2011 (http://blog.csdn.net/lxy_2011) 2015年05月26日 15:19 688

**圆周率 我国古代数学家对圆周率方面的研究工作，成绩是突出的。三国时期的刘徽、南北朝时...**

/* * 圆周率 我国古代数学家对圆周率方面的研究工作，成绩是突出的。三国时期的刘徽、南北朝时期的祖冲之都在这个领域取得过辉煌战绩。 有了计算机，圆周率的计算变得十分容易了。如今，人们创造了上百...

hanshileiai (http://blog.csdn.net/hanshileiai) 2013-03-28 22:27 2774

**中国移动研究院的Hadoop相关研究工作 (http://download.csdn.net/detail...**

PDF (http://downloa... 2014-12-21 22:02 1.08MB 下载

**华为公司是如何开展信号完整性和电源完整性分析研究工作的 (http://downl...**

PDF (http://downloa... 2008-10-19 19:51 81KB 下载

**hadoop学习工作总结（三）之数据优化 (/xiaozhuangfeng/article/details/39006593)**

数据优化： 1、小表放在前面，大表放在后面。因为会把前面的表读进内存再进行关联。 2、把分区的条件在on关系后面，不要放在where后面。因为放where后面会把所有分区关联后再按分区过滤。 3...

xiaozhuangfeng (http://blog.csdn.net/xiaozhuangfeng) 2014-09-02 15:11 301

**第二讲 模拟集成电路设计自动化的研究工作 (http://download.csdn.net/detail/ppdongdong...**