

# 基于 clang 插件的一种 iOS 包大小瘦身方案

infoq.cn/article/clang-plugin-ios-app-size-reducing

## 引子

包瘦身，包瘦身，包瘦身，重要的事情说三遍。

最近公司一款 iOS APP(本文只讨论使用 Objective C 开发的 iOS 安装包)一直在瘦身，我们团队的 APP 也愈发庞大了。而要解决这个问题，思路主要集中在两个方向，资源和代码。资源主要在于图片，方法包括移除未被引用的图片，只使用一套图片 (2x 或 3x)，图片伸缩等；代码层面主要思路包括重构消除冗余，linkmap 中 selector 引用分析等。除此之外，有没有别的路径呢？

众所周知，代码之间存在调用关系。假设 iOS APP 的主入口为 `-[UIApplication main]`，则所有开发者的源代码 (包括第三方库) 可分为两类：存在一条调用路径，使得代码可以被主入口最终调用 (称此类代码为被最终调用)；不存在一条调用路径，使得代码最终不能被主入口调用 (称此类代码为未被最终调用)。

假设有一个源代码级别的分析工具 (或编译器)，可以辅助分析代码间的调用关系，这样就使得分析最终被调用代码成为可能，剩下的就是未被最终调用的代码。

这种工具目前有成熟可用的吗？答案是肯定的，就是 clang 插件。除可用于分析未被最终调用代码外，clang 还可辅助发现重复代码。

## LLVM 与 clang 插件

LLVM 工程包含了一组模块化，可复用的编辑器和工具链。同其名字原意 (Low Level Virtual Machine) 不同的是，LLVM 不是一个首字母缩写，而是工程的名字。目前 LLVM 包含的主要子项目包括：

1. LLVM Core: 包含一个现在的源代码 / 目标设备无关的优化器，一集一个针对很多主流 (甚至一些非主流) 的 CPU 的汇编代码生成支持。
2. Clang: 一个 C/C++/Objective-C 编译器，致力于提供令人惊讶的快速编译，极其有用的错误和警告信息，提供一个可用于构建很棒的源代码级别的工具。
3. dragonegg: gcc 插件，可将 GCC 的优化和代码生成器替换为 LLVM 的相应工具。
4. LLDB: 基于 LLVM 提供的库和 Clang 构建的优秀本地调试器。
5. libc++、libc++ ABI: 符合标准的，高性能的 C++ 标准库实现，以及对 C++11 的完整支持。
6. compiler-rt: 针对 `__fixunsdftdi` 和其他目标机器上没有一个核心 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。
7. OpenMP: Clang 中对多平台并行编程的 runtime 支持。
8. vmkit: 基于 LLVM 的 Java 和 .NET 虚拟机实现
9. polly: 支持高级别的循环和数据本地化优化支持的 LLVM 框架。
10. libclc: OpenCL 标准库的实现
11. klee: 基于 LLVM 编译基础设施的符号化虚拟机
12. SAFECode: 内存安全的 C/C++ 编译器
13. lld: clang/llvm 内置的链接器

作为 LLVM 提供的编译器前端，clang 可将用户的源代码 (C/C++/Objective-C) 编译成语言 / 目标设备无关的 IR(Intermediate Representation) 实现。其可提供良好的插件支持，容许用户在编译时，运行额外的自定义动作。

我们的目标是使用 clang 插件减少包大小。其原理是，针对目标工程，基于 clang 的插件特性，开发者可以编写插件以分析所有源代码。编译过程中，将插件作为 clang 的参数载入并生成各种中间文件。编译完成后，还需编写一个工具去分析所有包含源码的方法 (包括用户编写，以及引入的第三方库源代码)，检查这些方法中哪些最终可被程序主入口调用，剩余即是疑似无用代码。简单的一个复查，移除那些确定无用的代码，重新编译，便可以有效去除无用的代码从而减少包大小。

本文相关内容如下：

1. 如何编写一个 clang 插件并集成到 Xcode
2. 如何实现代码级别的包瘦身
3. 局限与个性化定制
4. 其他

## 如何编写一个 clang 插件并集成到 Xcode

### Clone clang 源码并编译安装

```
cd /opt
sudo mkdir llvm
sudo chown `whoami` llvm
cd llvm
export LLVM_HOME=`pwd`

git clone -b release_39 git@github.com:llvm-mirror/llvm.git llvm
git clone -b release_39 git@github.com:llvm-mirror/clang.git llvm/tools/clang
git clone -b release_39 git@github.com:llvm-mirror/clang-tools-extra.git
llvm/tools/clang/tools/extra
git clone -b release_39 git@github.com:llvm-mirror/compiler-rt.git
llvm/projects/compiler-rt

mkdir llvm_build
cd llvm_build
cmake ../llvm -DCMAKE_BUILD_TYPE:STRING=Release
make -j`sysctl -n hw.logicalcpu`
```

### 编写 clang 插件

要实现自定义的 clang 插件 (以 C++ API 为例)，应按照以下步骤：

1. 自定义继承自
  - `clang::PluginASTAction` (基于 consumer 的抽象语法树 (Abstract Syntax Tree/AST) 前端 Action 抽象基类)
  - `clang::ASTConsumer` (用于客户读取抽象语法树的抽象基类)，
  - `clang::RecursiveASTVisitor` (前序或后续地深度优先搜索整个抽象语法树，并访问每一个节点的基类) 等基类。

2. 根据自身需要重载

`PluginASTAction::CreateASTConsumer`

```
ASTConsumer::HandleTranslationUnit
```

```
RecursiveASTVisitor::VisitDecl
```

```
RecursiveASTVisitor::VisitStmt
```

等方法，实现自定义的分析逻辑。

### 3. 注册插件

```
static FrontendPluginRegistry::Add<MyPlugin> X("my-plugin-name", "my-plugin-description");
```

更多 clang 插件：<http://clang.llvm.org/docs/ExternalClangExamples.html>

## 编译生成插件 (dylib)

假定你的 clang 插件源文件为 your-clang-plugin-source.cpp，需生成的插件名为 your-clang-plugin-name.dylib，可以使用如下命令 (载入了 llvm，clang 的 include 路径，生成的相关 lib 等) 生成：

```
clang -std=c++11 -stdlib=libc++ -L/opt/local/lib -
L/opt/llvm/llvm_build/lib -I/opt/llvm/llvm_build/tools/clang/include -
I/opt/llvm/llvm_build/include -I/opt/llvm/llvm/tools/clang/include -
I/opt/llvm/llvm/include -dynamiclib -Wl,-headerpad_max_install_names -lclang -
lclangFrontend -lclangAST -lclangAnalysis -lclangBasic -lclangCodeGen -
lclangDriver -lclangFrontendTool -lclangLex -lclangParse -lclangSema -
lclangEdit -lclangSerialization -lclangStaticAnalyzerCheckers -
lclangStaticAnalyzerCore -lclangStaticAnalyzerFrontend -lLLVMX86CodeGen -
lLLVMX86AsmParser -lLLVMX86Disassembler -lLLVMEExecutionEngine -lLLVMAsmPrinter -
lLLVMSelectionDAG -lLLVMX86AsmPrinter -lLLVMX86Info -lLLVMMCParser -
lLLVMCodeGen -lLLVMX86Utils -lLLVMScalarOpts -lLLVMInstCombine -
lLLVMTransformUtils -lLLVMAnalysis -lLLVMTarget -lLLVMCore -lLLVMMC -
lLLVMSupport -lLLVMBitReader -lLLVMOption -lLLVMProfileData -lpthread -lcurses -
-lz -lstdc++ -fPIC -fno-common -Woverloaded-virtual -Wcast-qual -fno-strict-
aliasing -pedantic -Wno-long-long -Wall -Wno-unused-parameter -Wwrite-strings -
fno-rtti -fPIC your-clang-plugin-source.cpp -o your-clang-plugin-name.dylib
```

## 与 Xcode 集成

下载 XcodeHacking.zip：

<https://raw.githubusercontent.com/kangwang1988/kangwang1988.github.io/master/others/XcodeHacking.zip>

使用命令行编译时，可以用如下方式载入插件：

```
clang++ *** -Xclang -load -Xclang path-of-your-plugin.dylib -Xclang -add-
plugin -Xclang your-pluginName -Xclang -plugin-arg-your-pluginName -Xclang
your-pluginName-param
```

要在 Xcode 中使用 clang 插件，需要如下 hack Xcode。

```
sudo mv HackedClang.xcplugin xcode-select -print-
path/../../PlugIns/Xcode3Core.ideplugin/Contents/SharedSupport/Developer/Library/Xcode/Plug-
ins
```

```
sudo mv HackedBuildSystem.xcspec xcode-select -print-
path/Platforms/iPhoneSimulator.platform/Developer/Library/Xcode/Specifications
```

在 Xcode->Target-Build Settings->Build Options->Compiler for C/C++/Objective-C 选择 Clang LLVM Trunk 即可使得 Xcode 使用上文生成的 clang 来编译。至于其他命令行参数均可通过 Xcode 中的编译选项设置完成。

## 如何实现代码级别的包瘦身

本文所说的代码指的是 OC 中的形如 `-/[Class method:\*]` 这种形式的代码，调用关系典型如下：

```
@interface ViewController : UIViewController
@end
@implementation ViewController
- (void)viewDidLoad {
    [super viewDidLoad];
    [self.view setBackgroundColor:[UIColor redColor]];
}
@end
```

则称： `-[ViewController viewDidLoad]` 调用了：

```
-[UIViewController viewDidLoad]
-[ViewController view] (语法糖)
+[UIColor redColor]
-[UIView setBackgroundColor:]
```

这种调用关系可在 clang 遍历抽象语法树的时候得到。由于编译器访问抽象语法树时存在嵌套关系，如上例：编译器在访问类实现 `ViewController` 的时候，嵌套了访问 `-[ViewController viewDidLoad]` 的方法实现，而在访问 `-[ViewController viewDidLoad]` 的方法实现的时候，嵌套了访问消息发送 `-[UIViewController viewDidLoad]` (对应源码 `[super viewDidLoad]`)，`-[ViewController view]` (对应源码 `self.view`)，`+[UIColor redColor]` (对应源码 `[UIColor redColor]`)，`-[UIView setBackgroundColor:]` (对应源码 `[self.view setBackgroundColor:[UIColor redColor]]`) 等，这样通过记录相关信息即可了解我们关注的方法间调用关系。

## 数据结构

为了分析调用关系，用到的中间数据结构如下：

### 类接口与继承体系 (clsInterfHierachy)

此数据结构记录了所有位于抽象语法树上的接口内容，最终的解析结果如下图所示：



以 AppDelegate 为例，interfs 代表其提供的接口 (注: 它的 property window 对应的 getter 和 setter 也被认为是 interf 一部分); isInSrcDir 代表此类是否位于用户目录 (将 workspace 的根目录作为参数传给 clang) 下，protos 代表其遵守的协议，superClass 代表接口的父类。

这些信息获取入口位于 `VisitDecl(Decl *decl)` 的重载函数里，相关的 decl 有：

- `ObjCInterfaceDecl` (接口声明)
- `ObjCCategoryDecl` (分类声明)
- `ObjCPropertyDecl` (属性声明)
- `ObjCMethodDecl` (方法声明)



此数据结构记录了所有包含源代码的 OC 方法，最终解析结果如下所示：

(点击放大图像)

```

1 {
2   "+[ViewController sharedInstance]": {
3     "callee": [
4       "+[ViewController alloc]",
5       "-[ViewController init]"
6     ],
7     "filename": "/XcodeZombieCodeDemo/XcodeZombieCodeDemo/ViewController.mm",
8     "range": "347-467",
9     "sourceCode": "{\n  static dispatch_once_t onceToken;\n  dispatch_once(&onceToken, ^{\n    sVc = [[self alloc] init];\n  });\n\n  }\n",
10  },
11  "-[AppDelegate application:didFinishLaunchingWithOptions:]": {
12    "callee": [
13      "+[ViewController alloc]",
14      "-[ViewController init]",
15      "-[ViewController setDelegate:]",
16      "+[ViewController sharedInstance]",
17      "-[AppDelegate setWindow:]",
18      "-[UIWindow initWithFrame:]",
19      "+[UIWindow alloc]",
20      "+[UIScreen mainScreen]",
21      "-[UIScreen bounds]",
22      "-[UIWindow setRootViewController:]",
23      "-[AppDelegate window]",
24      "-[UIWindow makeKeyAndVisible]",
25      "-[NSNotificationCenter addObserver:selector:name:object:]",
26      "+[NSNotificationCenter defaultCenter]",
27      "-[ViewController initWithRootViewControllerURL:]",
28      "-[NoMethodUsedClass init]",
29      "+[NoMethodUsedClass alloc]"
30    ],
31    "filename": "/XcodeZombieCodeDemo/XcodeZombieCodeDemo/AppDelegate.m",
32    "range": "415-1456",
33    "sourceCode": "{\n  // Override point for customization after application launch.\n  ViewController *vc = [ViewController new];\n\n  }\n",
34  },

```

以 `-[AppDelegate application:didFinishLaunchingWithOptions:]` 为例, callee 代表其调用到的接口 (此处为可以明确类型的, 对于形如 `id<XXXDelegate>` 后文介绍), filename 为此方法所在的文件名, range 为方法所在的范围, sourceCode 为方法的具体实现源代码。

这些信息获取入口位于 `VisitDecl(Decl *decl)` 和 `VisitStmt Stmt *stmt)` 的重载函数里, 相关的 decl 有 `ObjCMethodDecl` (方法声明), stmt 有 `ObjCMessageExpr` (消息表达式)

此处除过正常的 `-/[Class method:\*]` 外, 还有其他较多的需要考虑的情形, 已知且支持的分析包括:

- NSObject 协议的 performSelector 方法簇  
`[obj performSelector:@selector(XXX)]` 不仅包含 `[obj performSelector:]` 也包含 `[obj XXX]` .(下同)
- 手势 / 按钮的事件处理 selector  
`addTarget:action:/initWithTarget:action:/addTarget:action:forControlEvents:`
- NotificationCenter 添加通知处理 Selector  
`addObserver:selector:name:object:`
- UIBarButtonItem 添加事件处理 Selector  
`< initWithTitle:style:target:action: style:target:action:initWithImage:landscapeImagePhone:>`
- Timer  
`scheduledTimerWithTimeInterval:target:selector:userInfo:repeats:/timerWithTimeInterval:target:selector:userInfo:repeats:/initWithFireDate:interval:target:selector:userInfo:repeats:`

detachNewThreadSelector:toTarget:withObject:/initWithTarget:selector:object:

- CADisplayLink

displayLinkWithTarget:selector:

- KVO 机制

`addObserver:forKeyPath:options:context:`，不同于别的都要处理方法本身调用和对应 `target:selector` 调用，这里 KVO 的 `addObserver` 则暗含了 `observeValueForKeyPath:ofObject:change:context:`。

- IBAction 机制

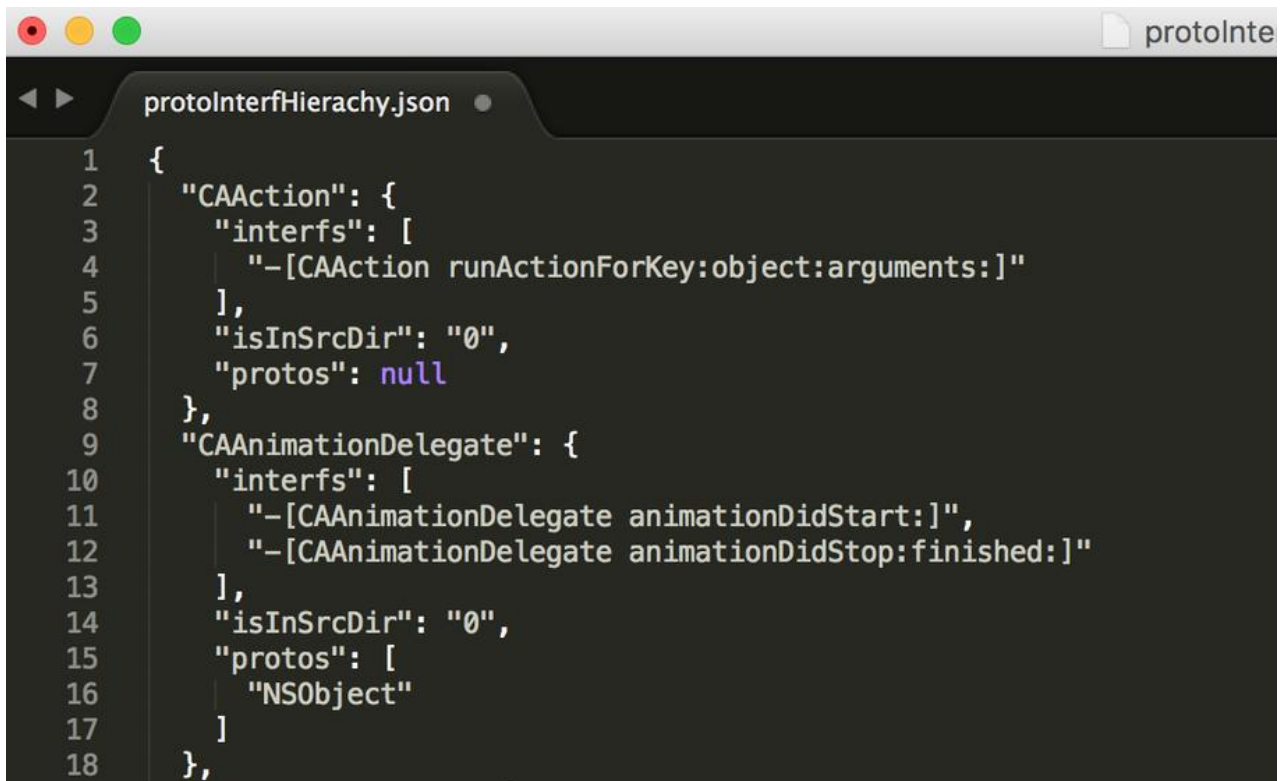
如基于 xib/Storyboard 的 ViewController 中 `-(IBAction)onBtnPressed:(id)sender` 方法，认为暗含了 `+[ViewController 的 alloc]` 对于 `+[ViewController 的 onBtnPressed:]` 的调用关系。

- `[XXX new]`

包含 `+[XXX alloc]` 和 `-[XXX init]`。

## 协议的接口与继承体系 (protoInterfHierachy)

此数据结构记录了所有位于抽象语法树上的协议内容，最终的解析结果如下图所示：



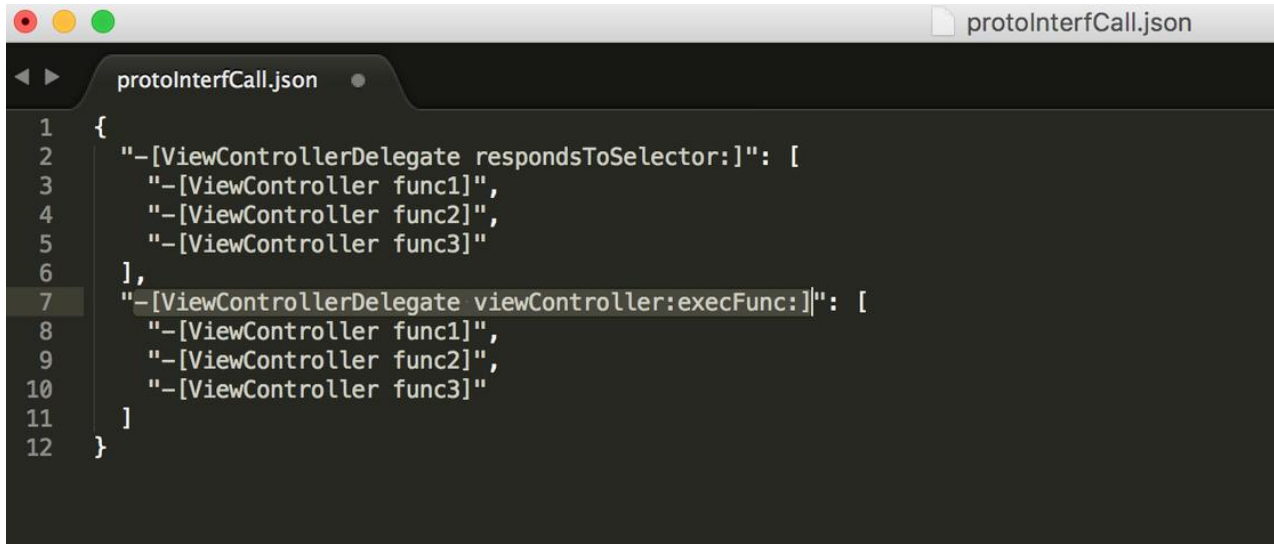
其中各字段定义同 `clsInterfHierachy`。

这些信息获取入口位于 `VisitDecl(Decl *decl)` 的重载函数里，相关的 `decl` 有：

- `ObjCProtocolDecl` (协议声明)
- `ObjCPropertyDecl` (属性声明)
- `ObjCMethodDecl` (方法声明)

## 协议方法的调用 (protoInterfCall)

此数据结构记录了所有如: `-[ViewController func1]` 调用了 `-[id\<ViewControllerDelegate\> viewController:execFunc:]` 的形式, 最终结果如下所示:

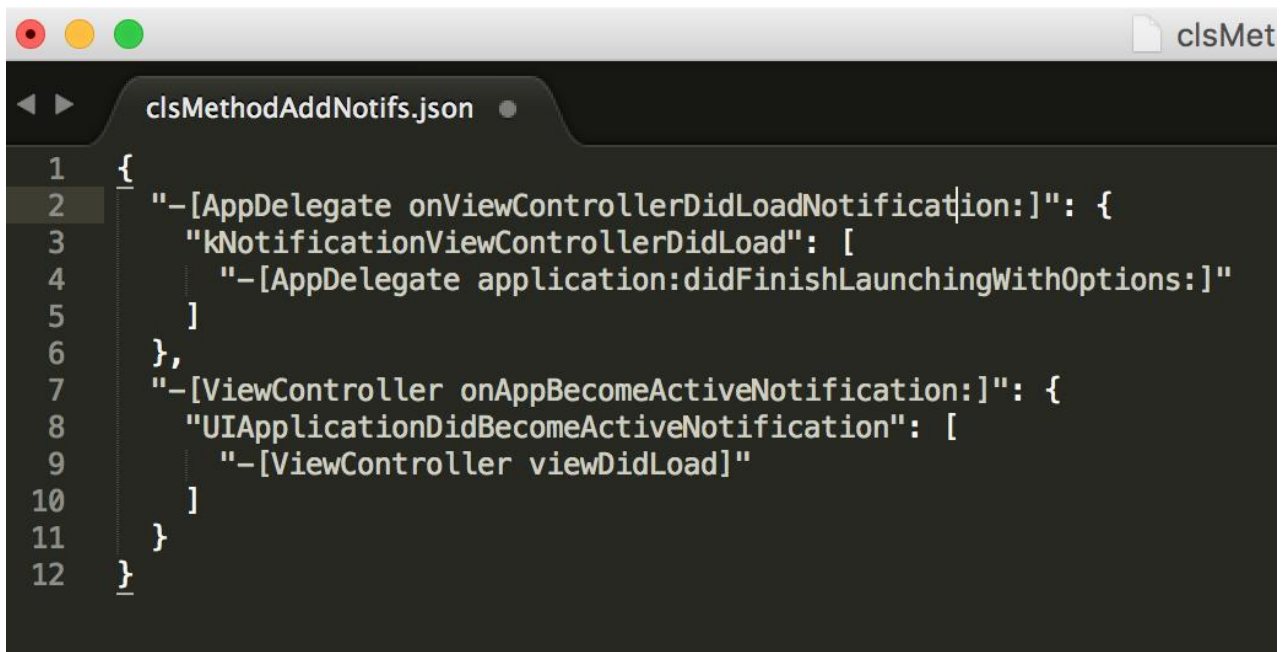


这些信息获取入口位于 `VisitStmt(Stmt *stmt)` 的重载函数里, 相关的 `stmt` 是 `ObjCMessageExpr` .

## 添加通知

以第一条记录为例, 其意思是说 `-[AppDelegate onViewControllerDidLoadNotification:]` 作为通知 `kNotificationViewControllerDidLoad` 的 Selector, 在 `-[AppDelegate application:didFinishLaunchingWithOptions:]` 中被添加。

(点击放大图像)

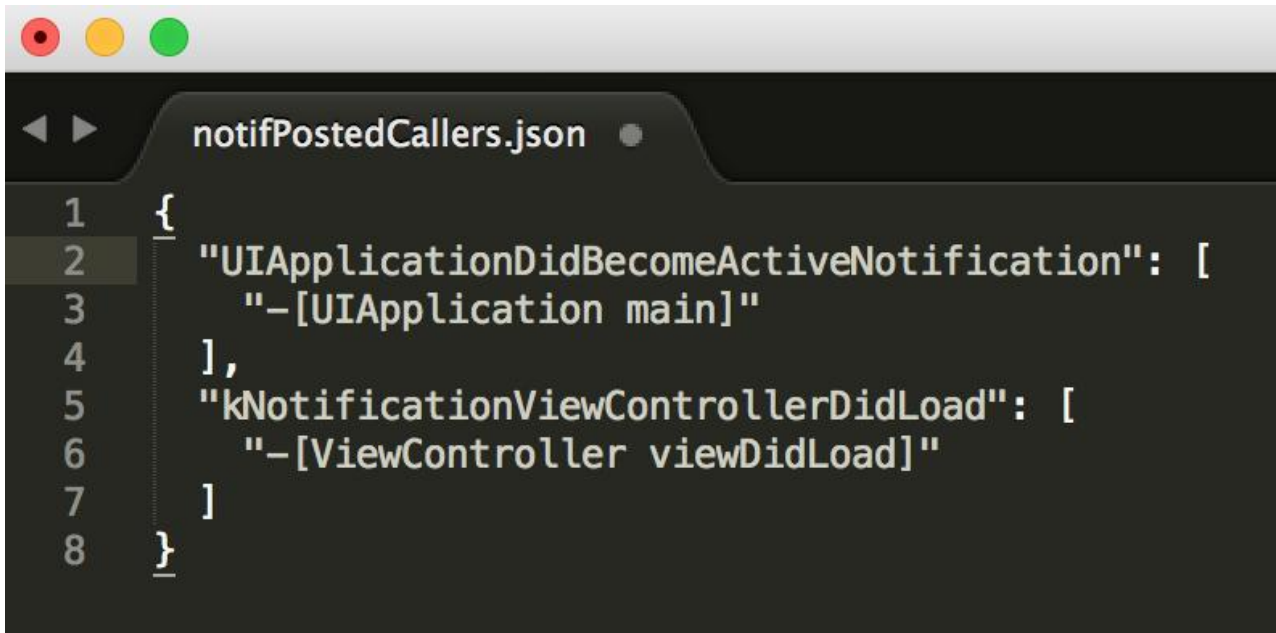


## 发送通知

第一条记录中, 作为系统级别的通知, 将被认为被 APP 主入口调用。

第二条记录则说明了, `-[ViewController viewDidLoad]` 发送了 `kNotificationViewControllerDidLoad`。





如果 `-[AppDelegate application:didFinishLaunchingWithOptions:]` 被 `-[UIApplication main]` (假定的主入口) 调用, 且 `-[ViewController viewDidLoad]` 被调用, 则 `-[AppDelegate onViewControllerDidLoadNotification:]` 被调用。其中, 如果通知是系统通知, 则只需要 `-[AppDelegate application:didFinishLaunchingWithOptions:]` 被调用即可。

这些信息获取入口位于 `VisitStmt(Stmt *stmt)` 的重载函数里, 相关的 `stmt` 有 `ObjCMessageExpr`。为了简单处理, 此处只处理形如 `addObserver:self` 这种 (也是最常见的情況), 否则 `Argu` 作为 `Expr*` 分析起来会很复杂。PS. 系统通知和本地通知的区别使用了名称上的匹配 (系统通知常以 `NS,UI,AV` 开头以 `Notification` 结束)。

## 重复代码分析

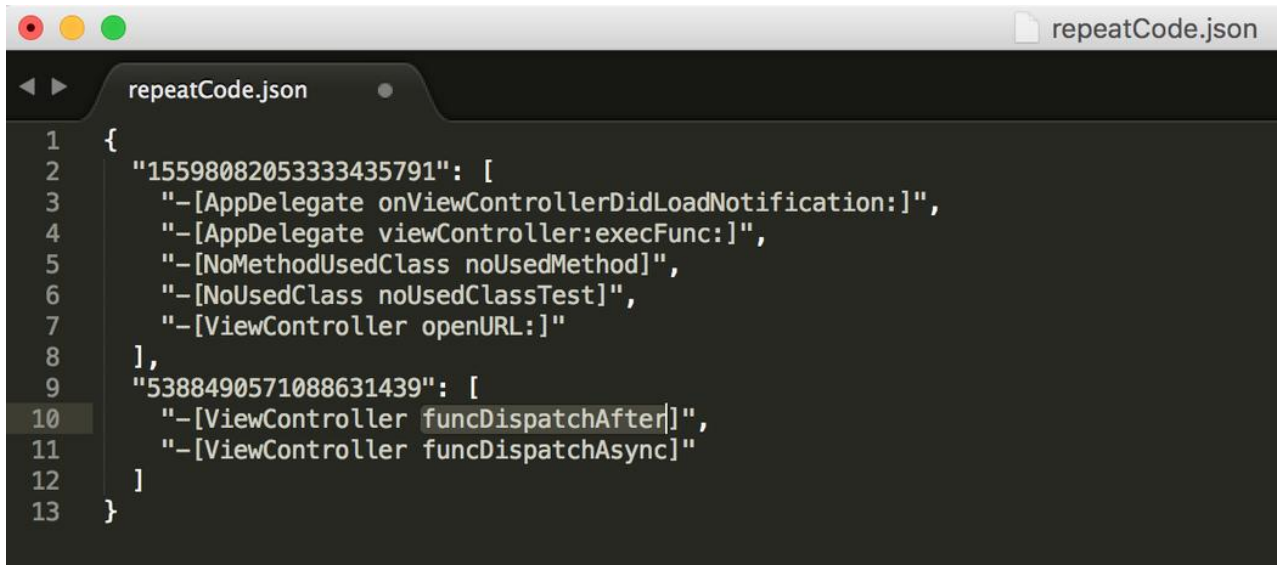
此处的重复代码针对的是某两个 (或两个以上) `-[Class method:\*]` 的实现是一模一样的。参考上文提到的 `clsMethod` 中的 `sourceCode`, 可以获得每一个方法实现的源代码。同时为了消除诸如格式上的差异 (如多了一个空格, 少了一个空格之类) 引起的差异, 先基于 `clang` 提供的 `format` 功能, 按照某种风格 (`google/llvm` 等) 将所有方法实现源码格式化, 再进行分析即可。

使用 `LLVM` 风格将代码 `format`:

```
find $prjDir -type f -name "*.m" | xargs /opt/llvm/llvm_build/bin/clang-format -i -style=LLVM
```

本文示例工程得到的一个重复代码结果如下所示:

(点击放大图像)



```
repeatCode.json
1  {
2    "1559808205333435791": [
3      "-[AppDelegate onViewControllerDidLoadNotification:]",
4      "-[AppDelegate viewController:execFunc:]",
5      "-[NoMethodUsedClass noUsedMethod]",
6      "-[NoUsedClass noUsedClassTest]",
7      "-[ViewController openURL:]"
8    ],
9    "5388490571088631439": [
10     "-[ViewController funcDispatchAfter]",
11     "-[ViewController funcDispatchAsync]"
12   ]
13 }
```

## 未被最终调用代码分析

分析的对象在于 clsMethod.json 里面所有的 key，即实际拥有源代码的所有方法。

1. 初始化默认的调用关系 usedClsMethodJson: `{-[AppDelegate alloc], "-[UIApplication main]", "-[UIApplication main]", "-[UIApplication main]", "+[NSObject alloc]", "-[UIApplication main]"}`，其中 AppDelegate 由用户传给 Analyzer。
2. 分析所有含源码方法是否存在一条路可以被已经调用 usedClsMethodJson 中的 key 调用。

对于某一个 clsMethod，其需要检查的路径包括三个，类继承体系，协议体系和通知体系。

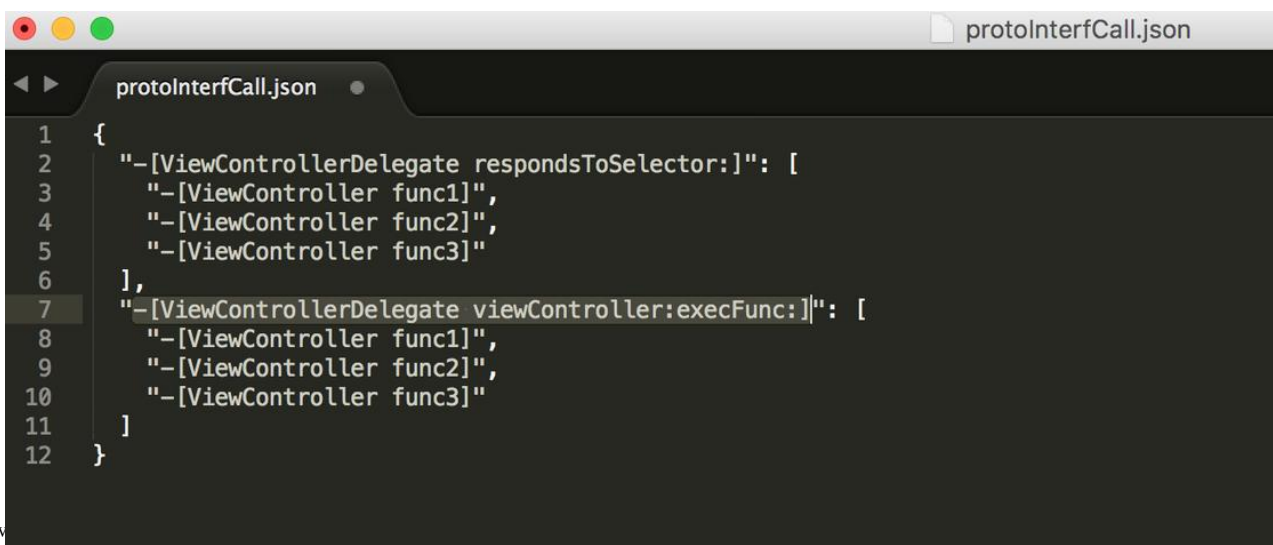
针对类继承体系，从当前类一直向上追溯 (直到发现有被调用或者 NSObject)，每一个基类对应的 `-/[Class method:*` 是否被隐含的调用关系所调用，如 `-[ViewController viewDidLoad]` 被 `-[ViewController alloc]` 隐含调用，当 `-[ViewController alloc]` 已经被调用的时候，`-[ViewController viewDidLoad]` 也将被认为调用。这里需要注意需要写一个隐含调用关系表以供查询，如下所示：

```

33  {"-[UITableViewCell initWithStyle:reuseIdentifier:]","+[UITableViewCell
    alloc]"},
34  {"-[UITableViewCell prepareForReuse]","+[UITableViewCell alloc]"},
35  {"-[UITableViewCell willTransitionToState:]","+[UITableViewCell alloc]"},
36  {"-[UITableViewCell didTransitionToState:]","+[UITableViewCell alloc]"},
37  {"-[NSObject init]","+[NSObject alloc]"},
38  {"-[NSObject copyWithZone]","+[NSObject alloc]"},
39  {"-[NSObject dealloc]","+[NSObject alloc]"},
40  {"-[NSObject description]","+[NSObject alloc]"},
41  {"-[NSObject debugDescription]","+[NSObject alloc]"},
42  {"-[NSObject valueForKey:]","+[NSObject alloc]"},
43  {"-[NSObject mutableArrayValueForKey:]","+[NSObject alloc]"},
44  {"-[NSObject mutableOrderedSetValueForKey:]","+[NSObject alloc]"},
45  {"-[NSObject mutableSetValueForKey:]","+[NSObject alloc]"},
46  {"-[NSObject valueForKeyPath:]","+[NSObject alloc]"},
47  {"-[NSObject mutableArrayValueForKeyPath:]","+[NSObject alloc]"},
48  {"-[NSObject mutableOrderedSetValueForKeyPath:]","+[NSObject
    alloc]"},
49  {"-[NSObject mutableSetValueForKeyPath:]","+[NSObject alloc]"},
50  {"-[NSObject valueForKeyUndefinedKey:]","+[NSObject alloc]"},
51  {"-[NSObject dictionaryWithValuesForKeys:]","+[NSObject alloc]"},
52  {"+[NSObject load]","+[NSObject alloc]"},
53  {"+[NSObject initialize]","+[NSObject alloc]"},
54  };
55

```

针对 Protocol 体系，需要参考类似 Protocol 引用体系向上追溯 (直到发现有被调用或者 `NSObject` 协议)，针对某一个特定的 Protocol 判断的时候，需要区分两种，一种是系统级的 Protocol，如 `UIApplicationDelegate`，对于 `-[AppDelegate application:didFinishLaunchingWithOptions:]` 这种，参考 `AppDelegate<UIApplicationDelegate>`，如果 `-[AppDelegate alloc]` 被调用则认为 `-[AppDelegate application:didFinishLaunchingWithOptions:]` 被调用。针对用户定义的 Protocol，如 `ViewControllerDelegate`，对于 `-[AppDelegate viewController:execFunc:]` 不仅需要 `-[AppDelegate alloc]` 被调用并且 `protoInterfCall.json` 中 `-[ViewControllerDelegate viewController:execFunc:]` 对应的 Callers 有已经存在于 `usedClsMethodJson` 的 Caller。



```

protoInterfCall.json
1  {
2    "-[ViewControllerDelegate respondsToSelector]": [
3      "-[ViewController func1]",
4      "-[ViewController func2]",
5      "-[ViewController func3]"
6    ],
7    "-[ViewControllerDelegate viewController:execFunc:]": [
8      "-[ViewController func1]",
9      "-[ViewController func2]",
10     "-[ViewController func3]"
11   ]
12  }

```



针对通知体系，前文已经有过分析。

本例分析使用到的 ClsMethod 结果如下：

(点击放大图像)

```

1 {
2   "[AppDelegate alloc]": "-[UIApplication main]",
3   "[NSNotificationCenter alloc]": "-[AppDelegate application:didFinishLaunchingWithOptions:]",
4   "[NSNotificationCenter defaultCenter]": "-[AppDelegate application:didFinishLaunchingWithOptions:]",
5   "[NSObject alloc]": "-[AppDelegate application:didFinishLaunchingWithOptions:]",
6   "[NoMethodUsedClass alloc]": "-[AppDelegate application:didFinishLaunchingWithOptions:]",
7   "[UIViewController alloc]": "-[ViewController viewDidLoad]",
8   "[UIColor alloc]": "-[ViewController viewDidLoad]",
9   "[UIColor redColor]": "-[ViewController viewDidLoad]",
10  "[UIResponder alloc]": "-[AppDelegate alloc]",
11  "[UIScreen alloc]": "-[AppDelegate application:didFinishLaunchingWithOptions:]",
12  "[UIScreen mainScreen]": "-[AppDelegate application:didFinishLaunchingWithOptions:]",
13  "[UIView alloc]": "-[AppDelegate application:didFinishLaunchingWithOptions:]",
14  "[UIViewController alloc]": "-[AppDelegate application:didFinishLaunchingWithOptions:]",
15  "[ViewController alloc]": "-[AppDelegate application:didFinishLaunchingWithOptions:]",
16  "[ViewController sharedInstance]": "-[AppDelegate application:didFinishLaunchingWithOptions:]",
17  "[AppDelegate alloc]": "-[AppDelegate alloc]",
18  "[AppDelegate application:didFinishLaunchingWithOptions:]": "-[AppDelegate alloc]",
19  "[AppDelegate applicationDidEnterBackground:]": "-[AppDelegate alloc]",
20  "[AppDelegate applicationWillEnterForeground:]": "-[AppDelegate alloc]",
21  "[AppDelegate applicationWillResignActive:]": "-[AppDelegate alloc]",
22  "[AppDelegate applicationWillTerminate:]": "-[AppDelegate alloc]",
23  "[AppDelegate onViewControllerDidLoadNotification:]": "-[ViewController viewDidLoad]",
24  "[AppDelegate setWindow:]": "-[AppDelegate application:didFinishLaunchingWithOptions:]",
25  "[AppDelegate viewController:execFunc:]": "-[ViewController func2]",
26  "[AppDelegate window:]": "-[AppDelegate application:didFinishLaunchingWithOptions:]",
27  "[NSNotificationCenter addObserver:selector:name:object:]": "-[AppDelegate application:didFinishLaunchingWithOptions:]",
28  "[NSNotificationCenter postNotificationName:object:]": "-[ViewController viewDidLoad]",
29  "[NSObject alloc]": "-[AppDelegate application:didFinishLaunchingWithOptions:]",
30  "[NSObject init]": "-[ViewController sharedInstance]",
31  "[NSObject performSelector:withObject:afterDelay:]": "-[ViewController viewDidLoad]",
32  "[NoMethodUsedClass alloc]": "-[AppDelegate application:didFinishLaunchingWithOptions:]",
33  "[NoMethodUsedClass init]": "-[AppDelegate application:didFinishLaunchingWithOptions:]",
34  "[NoMethodUsedClass noUsedMethod]": "-[NoMethodUsedClass init]",
35  "[UIApplication main]": "-[UIApplication main]",
36  "[UIScreen alloc]": "-[AppDelegate application:didFinishLaunchingWithOptions:]",
37  "[UIScreen bounds]": "-[AppDelegate application:didFinishLaunchingWithOptions:]",
38  "[UIView alloc]": "-[AppDelegate application:didFinishLaunchingWithOptions:]",
39  "[UIView initWithFrame:]": "-[AppDelegate application:didFinishLaunchingWithOptions:]",
40  "[UIView setBackgroundColor:]": "-[ViewController viewDidLoad]",
41  "[UIViewController alloc]": "-[ViewController didReceiveMemoryWarning]",
42  "[UIViewController didReceiveMemoryWarning]": "-[AppDelegate application:didFinishLaunchingWithOptions:]",
43  "[AppDelegate applicationWillResignActive:]": "-[AppDelegate alloc]"

```

本例分析未被使用到的 ClsMethod 结果如下：

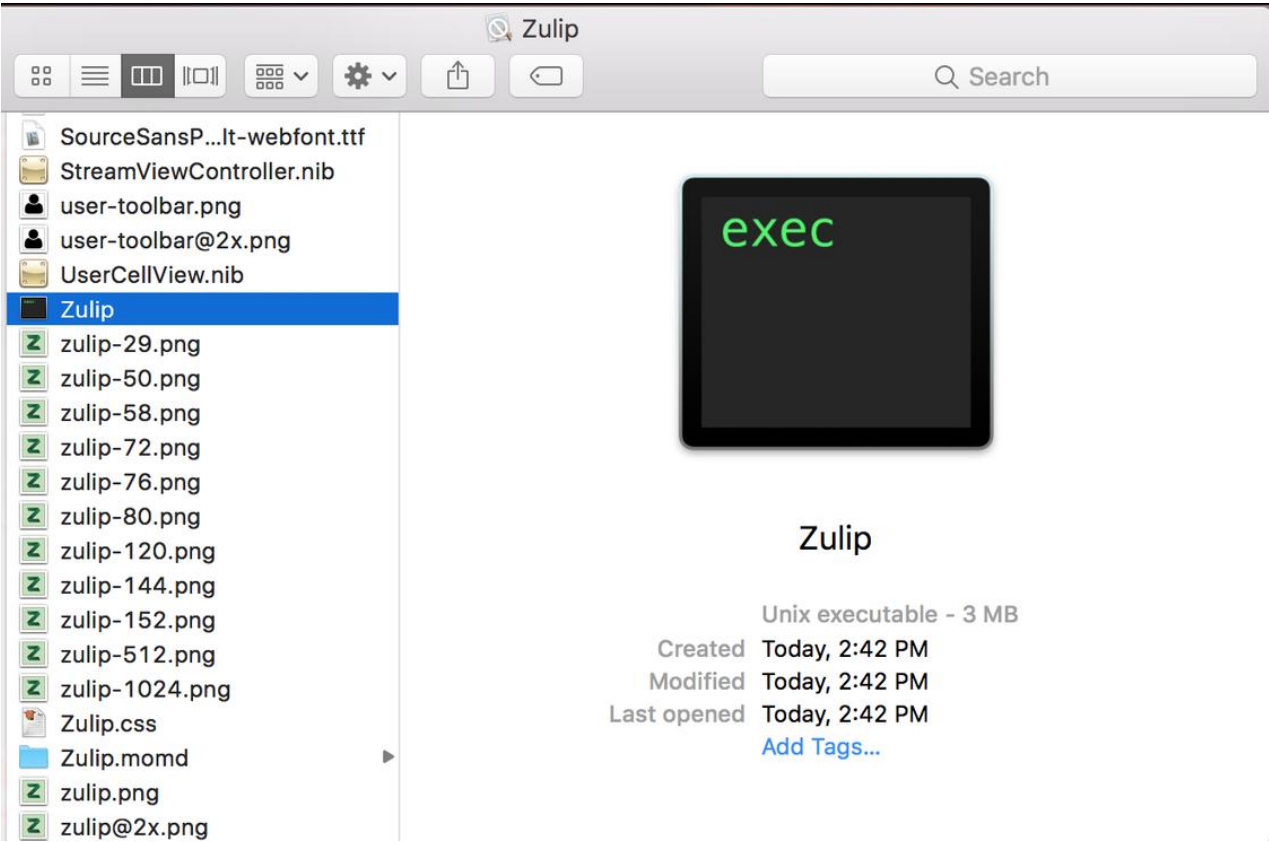
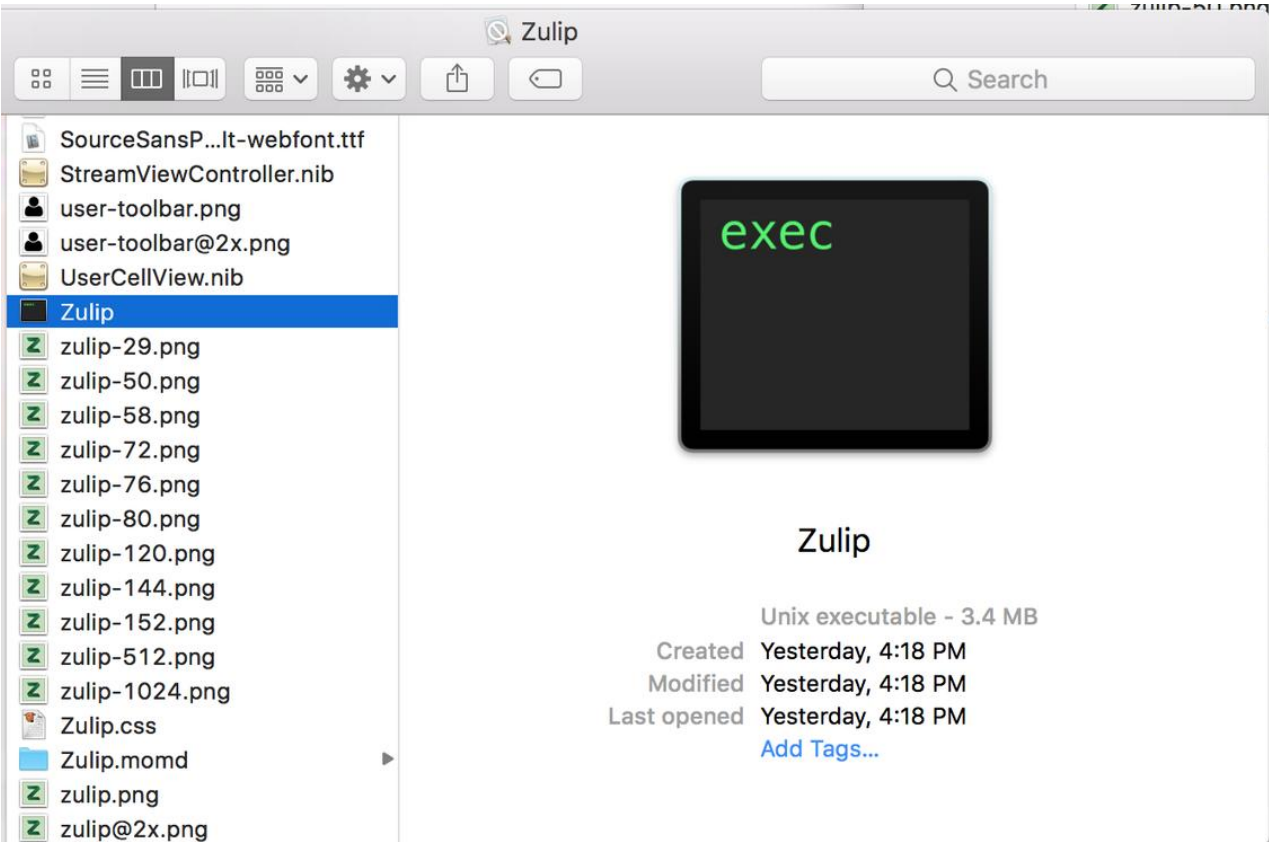
```

1 {
2   "-[NoUsedClass noUsedClassTest]": {
3     "filename": "/XcodeZombieCodeDemo/XcodeZombieCodeDemo/NoUsedClass.m",
4     "range": "385-387",
5     "sourceCode": "{\n\n}"
6   },
7   "-[ViewController extensionTest]": {
8     "callee": [
9       "+[NSObject superclass]"
10    ],
11    "filename": "/XcodeZombieCodeDemo/XcodeZombieCodeDemo/ViewController+Extension.m",
12    "range": "257-284",
13    "sourceCode": "{\n  [NSObject superclass];\n}"
14  },
15  "-[ViewController func1]": {
16    "callee": [
17      "-[ViewController delegate]"
18    ],
19    "filename": "/XcodeZombieCodeDemo/XcodeZombieCodeDemo/ViewController.mm",
20    "range": "1655-1808",
21    "sourceCode": "{\n  NSInteger x = 100;\n  if ([self.delegate respondsToSelector:@selector(viewController:execFunc:)])\n    [self.delegate respondsToSelector:@selector(viewController:execFunc:)]\n}"
22  },
23  "-[ViewController func3]": {
24    "callee": [
25      "-[ViewController delegate]"
26    ],
27    "filename": "/XcodeZombieCodeDemo/XcodeZombieCodeDemo/ViewController.mm",
28    "range": "1995-2148",
29    "sourceCode": "{\n  NSInteger x = 100;\n  if ([self.delegate respondsToSelector:@selector(viewController:execFunc:)])\n    [self.delegate respondsToSelector:@selector(viewController:execFunc:)]\n}"
30  },
31  "-[ViewController openURL]": {
32    "filename": "/XcodeZombieCodeDemo/XcodeZombieCodeDemo/ViewController.mm",
33    "range": "1318-1320",
34    "sourceCode": "{\n\n}"
35  }
36 }

```

## zulip-ios 的应用效果对比

鉴于示例工程规模较小，另选取开源的 zulip-ios 工程，其中原始工程 Archive 生成的可执行文件大小为 3.4MB，结合本文所述方法去除未被最终调用的代码 (包括业务代码，第三方库) 后，可执行文件变为 3MB。对于这样一个设计良好的工程，纯代码的瘦身效果还是比较可观的。



## 局限与个性化定制



这种静态分析适合可以判断出消息接收者类型的情况，面对运行时类型和静态分析类型不一致，或者静态分析不出来类型时，不可用。这种分析要求代码书写规范。例如一个 Class 实现了某个 Protocol，一定要在声明里说明，或者 Property 中 delegate 是 `id<XXXDelegate>` 的时候也要注明。

虽然此项目已经给了一个完整的重复代码和无用代码分析工具，但也有其局限性 (主要是动态特性)。具体分析如下：

#### 1. openUrl 机制

假设工程设置里使用了 `openUrl:"XXX://XXViewController"` 来打开一个 VC，则 Clang 插件里面需要分析 openUrl 的参数，如果参数是 XXViewController，则暗含了 `+[XXViewController alloc]` 和 `-[XXViewController init]`。

#### 2. Model 转化

如如果 MTLModel 使用到了 `modelOfClass:[XXXModel class]` `fromJSONDictionary:error:`，则暗含了 `+[XXXModel alloc]` 和 `+[XXXModel init]`。

#### 3. Message swizzle

假设用户 swizzle 了 `-[UIViewController viewDidLoad]` 和 `-[UIViewController XXviewDidLoad]`，则需要在 implicitCallStackJson 中添加 `-[UIViewController XXviewDidLoad]`，`-[UIViewController viewDidLoad]`。

#### 4. 第三方 Framework 暗含的逻辑

如高德地图的 AnnotationView，需要 implicitCallStackJson 中添加 `"-[MAAnnotationView prepareForReuse:]","+[MAAnnotationView alloc]"` 等。包括第三方 Framework 里面的一些 Protocol，可能也需要参考前文提到的 UIApplicationDelegate 按照系统级别的 Protocol 来处理。

#### 5. 一些遗漏的重载方法

如 `-[XXDerivedManager sharedInstance]` 并无实现，而 XXDerivedManager 的基类 XXBaseManager 的 sharedInstance 调用了 `-[self alloc]`，但因为 self 静态分析时被认定为 XXBaseManager，这就导致 `-[XXDerivedManager sharedInstance]` 虽然被 usedclsmethod.json 调用，但是 `-[XXDerivedManager alloc]` 却不能被调用。这种情况，可以在 usedClsMethodJson 初始化的时候，加入 `"+[XXDerivedManager alloc]","-[UIApplication main]"`。

#### 6. 类似 Cell Class

我们常会使用动态的方法去使用 `[[[XXX cellClassWithCellModel:] alloc] initWithStyle:reuseIdentifier:]` 去构造 Cell，这种情况下，应该针对 `cellClassWithCellModel` 里面会包含的各种 `return [XXXCell class]`，在 implicitCallStackJson 中添加 `[[XXXCell alloc] initWithStyle:reuseIdentifier:],-[XXX cellClassWithCellModel:]` 这种调用。

#### 7. Xib/Storyboard 会暗含一些 UI 元素 (Controller,Table,Button,Cell,View 等) 的 alloc 方法或调用关系。

#### 8. 其他隐含的逻辑或者动态特性导致的调用关系遗漏。

## 其他

对于包大小而言，可以参考以下的思路去瘦身代码：

#### 1. 重复代码的提取重构

#### 2. 无用代码的移除

3. 使用率较低的第三方库的处理 (本文不仅可以查找到重复, 无用的代码, 进一步分析 `clsMethod.json/unusedClsMethod.json` 更可以获取到每一个 framework 里面有多少个方法, 各方法有多少代码, 多少个方法又被 `-[UIApplication main]` 调用到了), 面对使用率很低的库, 需要考虑是不是要全部引入或者重写。
4. 重复引用的第三方库的处理 (曾经发现团队项目的工程里面引用了其他团队的库, 但由于多个库里面均有一份自己的 Zip 的实现, 面对这种情况, 可以考虑将此种需求全部抽象出来一个公共的 Framework 去处理, 其他人都引用此项目, 或者干脆使用系统本身自带的 libz 去处理会更好些)。

因为可在源码级别分析, 使用 clang 插件可做的工作很多。笔者还使用了 clang 插件去实现了代码风格检查, API 有效性验证, 相关示例项目如下:

代码风格检查: <https://github.com/kangwang1988/XcodeCodingStyle.git>

API 有效性验证: <https://github.com/kangwang1988/XcodeValidAPI.git>

---

感谢徐川对本文的审校。

给 InfoQ 中文站投稿或者参与内容翻译工作, 请邮件至 [editors@cn.infoq.com](mailto:editors@cn.infoq.com)。也欢迎大家通过新浪微博 ( @InfoQ , @丁晓昀 ), 微信 (微信号: InfoQChina) 关注我们。