**Reduce Guide(iOS)**

**uweiyuan(袁有为)**

# 引言

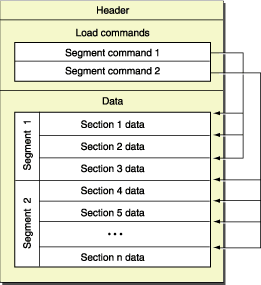
iOS安装包主要由三部分组成：二进制可执行文件、资源文件（图片和视频）和其他文件（sdb数据库、plist文件等），Reduce将从这几个方面来分析。

# mach-o文件

## 内部结构

我们的程序想要跑起来，肯定它的可执行文件格式要被操作系统所理解，比如 ELF 是 Linux下可执行文件的格式，PE32／PE32+是windows的可执行文件的格式，那么对于OS X和iOS 来说 Mach-O 是其可执行文件的格式。

我们平时了解到的可执行文件、库文件、DSYM文件、动态库、动态连接器都是这种格式的。Mach-O 的组成结构如下图所示包括了Header、Load commands、Data（包含Segement的具体数据）



Mach-o包含三个基本区域：

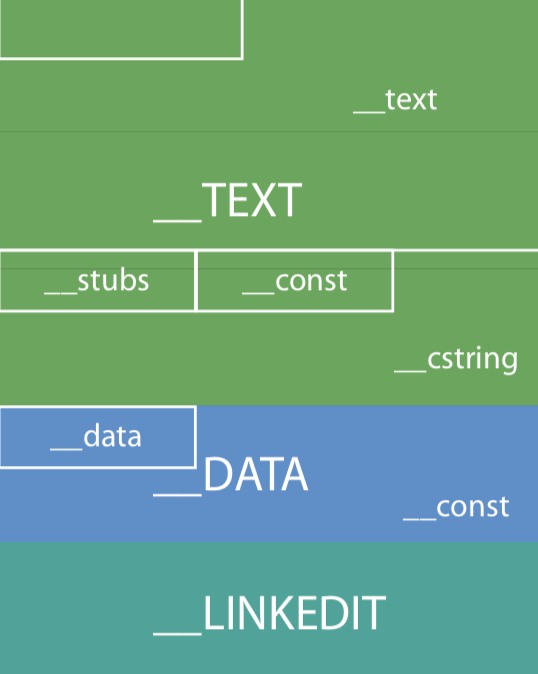
1.头部（header structure）保存了Mach-O的一些基本信息，包括了平台、文件类型、LoadCommands的个数等等，主要作用就是帮助系统迅速的定位Mach-O文件的运行环境，文件类型

2.加载命令（load command）紧跟在头部之后，这些加载指令清晰地告诉加载器如何处理二进制数据，有些命令是由内核处理的，有些是由动态链接器处理的。在源码中有明显的注释来说明这些是动态连接器处理的。它描述了文件中数据的具体组织结构，也说明了进程启动后，对应的内存空间结构是如何组织的

3.数据，即段（segment）。保存了具体的代码和数据。每一个段（segment）都拥有一段虚拟地址映射到进程的地址空间。

4.链接信息。一个完整的用户级Mach-o文件的末端是链接信息。其中包含了动态加载器用来链接可执行文件或者依赖库所需使用的符号表，字符串表等等

重要段中的section，如下图:



以上数据结构可以参照mac OS系统文件<loader.h>

## 生成原理和步骤

举例说明helloworld.c

#include <stdio.h>

int main(int argc, char \*argv[])

{

printf("Hello World!\n");

return 0;

}

Hello World 和编译器

时下 Xcode 中编译器默认选择使用 clang

简单的说，编译器处理过程中，将 helloworld.c 当做输入文件，并生成一个可执行文件 a.out。这个过程有多个步骤/阶段。我们需要做的就是正确的执行它们。

**预处理**

符号化 (Tokenization)

宏定义的展开

#include 的展开

**编译器**

1.语法和语义分析

将符号化后的内容转化为一棵解析树 (parse tree)

解析树做语义分析

输出一棵抽象语法树（Abstract Syntax Tree\* (AST)）

2.生成代码和优化

将 AST 转换为更低级的中间码 (LLVM IR)

对生成的中间码做优化

生成特定目标代码

输出汇编代码

**汇编器**

将汇编代码转换为目标对象文件

**链接器**

将多个目标对象文件合并为一个可执行文件 (或者一个动态库)

Ex:

### 预处理

编译过程中，编译器首先要做的事情就是对文件做处理。预处理结束之后，如果我们停止编译过程，那么我们可以让编译器显示出预处理的一些内容：

% xcrun clang -E helloworld.c | open -f

在顶部可以看到的许多行语句都是以 # 开头 (读作 hash)。这些被称为 行标记 的语句告诉我们后面跟着的内容来自哪里。如果再回头看看 helloworld.c 文件，会发现第一行是：

#include <stdio.h>

我们都用过 #include 和 import。它们所做的事情是告诉预处理器将文件 stdio.h 中的内容插入到 #include 语句所在的位置。这是一个递归的过程：stdio.h 可能会包含其它的文件。

由于这样的递归插入过程很多，所以我们需要确保记住相关行号信息。为了确保无误，预处理器在发生变更的地方插入以 # 开头的 行标记。跟在 # 后面的数字是在源文件中的行号，而最后的数字是在新文件中的行号。回到刚才打开的文件，紧跟着的是系统头文件，或者是被看做为封装了 extern "C" 代码块的文件。

如果滚动到文件末尾，可以看到我们的 helloworld.c 代码：

# 2 "helloworld.c" 2

int main(int argc, char \*argv[])

{

printf("Hello World!\n");

return 0;

}

在 Xcode 中，可以通过这样的方式查看任意文件的预处理结果：Product -> Perform Action -> Preprocess。注意，编辑器加载预处理后的文件需要花费一些时间 -- 接近 100,000 行代码。

### 编译

下一步：分析和代码生成。我们可以用下面的命令让 clang 输出汇编代码：

% xcrun clang -S -o - helloworld.c | open -f

我们来看看输出的结果。首先会看到有一些以点 . 开头的行。这些就是汇编指令。其它的则是实际的 x86\_64 汇编代码。最后是一些标记 (label)，与 C 语言中的类似。

我们先看看前三行：

.section \_\_TEXT,\_\_text,regular,pure\_instructions

.globl \_main

.align 4, 0x90

这三行是汇编指令，不是汇编代码。.section 指令指定接下来会执行哪一个段。

第二行的 .globl 指令说明 \_main 是一个外部符号。这就是我们的 main() 函数。这个函数对于二进制文件外部来说是可见的，因为系统要调用它来运行可执行文件。

.align 指令指出了后面代码的对齐方式。在我们的代码中，后面的代码会按照 16(2^4) 字节对齐，如果需要的话，用 0x90 补齐。

接下来是 main 函数的头部：

\_main: ## @main

.cfi\_startproc

## BB#0:

pushq %rbp

Ltmp2:

.cfi\_def\_cfa\_offset 16

Ltmp3:

.cfi\_offset %rbp, -16

movq %rsp, %rbp

Ltmp4:

.cfi\_def\_cfa\_register %rbp

subq $32, %rsp

上面的代码中有一些与 C 标记工作机制一样的一些标记。它们是某些特定部分的汇编代码的符号链接。其他不详细说，请参考链接。

更过关于汇编指令的资料可以在 苹果的 [OS X Assembler Reference](https://developer.apple.com/library/mac/documentation/DeveloperTools/Reference/Assembler/) 中看到。AMD 64 网站有关于 [ABI for x86 的文档](http://www.x86-64.org/documentation/abi.pdf)。另外还有 [Gentle Introduction to x86-64 Assembly](http://www.x86-64.org/documentation/assembly.html)。

重申一下，通过下面的选择操作，我们可以用 Xcode 查看任意文件的汇编输出结果：Product -> Perform Action -> Assemble.

### 汇编器

汇编器将可读的汇编代码转换为机器代码。它会创建一个目标对象文件，一般简称为 对象文件。这些文件以 .o 结尾。如果用 Xcode 构建应用程序，可以在工程的 derived data 目录中，Objects-normal 文件夹下找到这些文件。

### 链接器

稍后我们会对链接器做更详细的介绍。这里简单介绍一下：链接器解决了目标文件和库之间的链接。什么意思呢？还记得下面的语句吗：

callq \_printf

printf() 是 libc 库中的一个函数。无论怎样，最后的可执行文件需要能需要知道 printf() 在内存中的具体位置：例如，\_printf 的地址符号是什么。链接器会读取所有的目标文件 (此处只有一个) 和库 (此处是 libc)，并解决所有未知符号 (此处是 \_printf) 的问题。然后将它们编码进最后的可执行文件中 （可以在 libc 中找到符号 \_printf），接着链接器会输出可以运行的执行文件：a.out

## Link Map File

一个详细说明output image(execute file， dynamic library)中所有符号和地址的文件

LinkMap文件是Xcode产生可执行文件的同时生成的链接信息，用来描述可执行文件的构造成分，包括代码段（\_\_TEXT）和数据段（\_\_DATA）的分布情况

在项目里新建一个类，给它添加几个方法，但不要在任何地方import它，build完项目后观察linkmap，你会发现这个类还是被编译进可执行文件了。这是因为object-c的runtime 性质，按C++的经验，没有被使用到的类和方法编译器都会优化掉，不会编进最终的可执行文件，但object-c不一样，因为object-c的动态特性，它可以通过类和方法名反射获得这个类和方法进行调用，所以就算在代码里某个类没被使用到，编译器也没法保证这个类不会在运行时通过反射去调用，所以只要是在项目里的文件，无论是否又被使用到都会被编译进可执行文件。

### 生成

可以在XCode中进行设置，Build Settings->Linking->Write Link Map File设置为YES，在编译app的时候就会生成相应的LinkMap文件，具体生成路径，也是可以在XCode中进行设置的

### 构成

（1）#Path：镜像文件路径

（2）#Arch：当前镜像文件支持的CPU架构

（3）#Object files：镜像文件所包含的目标文件列表，索引+路径

（4）# Sections：镜像文件所包含的全部Section列表，起始地址+大小+所属段+名称

（5）# Symbols：镜像文件所包含的符号在对应目标文件中的起始地址和大小，起始地址+大小+对应(3)中目标文件索引+符号名

**符号显示规则**

**OC规则的函数显示：**

**0x00006E04 0x000002EC [ 3] +[XGKeyChain load:]**

**C语言的函数显示：**

**0x10470E254 0x00000006 [2063] \_sqlite3\_bind\_blob**

### 计算

1）在LinkMap文件中，查找目标文件的索引

2）对每个索引，在#Symbols区域得到索引所对应行的Size

3）将上一步统计的同一索引的所有Size进行累加

4）对其他索引，重复2）3）步

5）将全部索引Size累加，即可得到镜像文件在指定CPU架构下的Size

由于\_\_DATA.\_\_bbs是代表未初始化的静态变量，Size表示应用运行时占用的堆大小，并不占用可执行文件，所以计算obj占用大小时，要排除这个段的Size

## Reduce Code

### 解析工具

otool,目标文件解析工具

可以显示对象文件或者是库的指定部分，otool可以很好的解析Mach-O文件和通用文件格式。

nm，展示目标文件的符号表

### 查找无用类

查找无用oc类有两种方式，

一种是类似于查找无用资源，通过搜索"[ClassName alloc/new"、"ClassName \*"、"[ClassName class]"等关键字在代码里是否出现。

另一种是通过otool命令逆向\_\_DATA.\_\_objc\_classlist段和\_\_DATA.\_\_objc\_classrefs段来获取当前所有oc类和被引用的oc类，两个集合相减就是无用oc类。

$ otool -s \_\_DATA \_\_objc\_classlist TestLinkMapWithOC:

Contents of (\_\_DATA,\_\_objc\_classlist) section

00000001000010d0 48 12 00 00 01 00 00 00 98 12 00 00 01 00 00 00

查询结果是目标文件中被编译入的OC类，前面是段起始地址 后面是小端地址表示法的类的地址，8位

$ otool -s \_\_DATA \_\_objc\_classrefs TestLinkMapWithOC:

Contents of (\_\_DATA,\_\_objc\_classrefs) section

0000000100001218 48 12 00 00 01 00 00 00

查询结果是目标文件中引用的类，前面是段起始地址，后面是小端表示法的类的地址

以上两步得到：0x100001298（00 98 12 00 00 01 00）这个地址的类是无用类

查询目标文件中DATA段的OC类的结构化信息

$ otool -o TestLinkMapWithOC:

Contents of (\_\_DATA,\_\_objc\_classlist) section

00000001000010d0 0x100001248

isa 0x100001220

superclass 0x0

cache 0x0

vtable 0x0

data 0x100001130 (struct class\_ro\_t \*)

flags 0x80

instanceStart 8

instanceSize 8

reserved 0x0

ivarLayout 0x0

name 0x100000f48 MyClass

baseMethods 0x0 (struct method\_list\_t \*)

baseProtocols 0x0

ivars 0x0

weakIvarLayout 0x0

baseProperties 0x0

Meta Class

isa 0x0

superclass 0x0

cache 0x0

vtable 0x0

data 0x1000010e8 (struct class\_ro\_t \*)

flags 0x81 RO\_META

instanceStart 40

instanceSize 40

reserved 0x0

ivarLayout 0x0

name 0x100000f48 MyClass

baseMethods 0x0 (struct method\_list\_t \*)

baseProtocols 0x0

ivars 0x0

weakIvarLayout 0x0

baseProperties 0x0

00000001000010d8 0x100001298

isa 0x100001270

superclass 0x0

cache 0x0

vtable 0x0

data 0x1000011c0 (struct class\_ro\_t \*)

flags 0x80

instanceStart 8

instanceSize 8

reserved 0x0

ivarLayout 0x0

name 0x100000f50 TestClass

baseMethods 0x0 (struct method\_list\_t \*)

baseProtocols 0x0

ivars 0x0

weakIvarLayout 0x0

baseProperties 0x0

Meta Class

isa 0x0

superclass 0x0

cache 0x0

vtable 0x0

data 0x100001178 (struct class\_ro\_t \*)

flags 0x81 RO\_META

instanceStart 40

instanceSize 40

reserved 0x0

ivarLayout 0x0

name 0x100000f50 TestClass

baseMethods 0x0 (struct method\_list\_t \*)

baseProtocols 0x0

ivars 0x0

weakIvarLayout 0x0

baseProperties 0x0

Contents of (\_\_DATA,\_\_objc\_classrefs) section

0000000100001218 0x100001248

Contents of (\_\_DATA,\_\_objc\_imageinfo) section

version 0

flags 0x40

根据这个结果中的类信息，找到上一步得到的未被使用的类的地址，从而确定未被使用的类的符号。

### 查找无用方法

1. 使用otool命令otool -V -s \_\_DATA \_\_objc\_selrefs逆向\_\_DATA.\_\_objc\_selrefs段，提取可执行文件里引用到的方法名;
2. 使用otool命令otool -V -s \_\_TEXT \_\_objc\_methname逆向\_\_TEXT \_\_objc\_methname段，提取可执行文件里全部的方法名
3. 根据以上两个集合，求差集，得出无用的方法列表

第三方库的无用selector也可以扫出来

$ otool -V -s \_\_TEXT \_\_objc\_methname TestLinkMapWithOC:

Contents of (\_\_TEXT,\_\_objc\_methname) section

0000000100000f5c  callMethod:

0000000100000f68  emptyMethod:age:

0000000100000f79  new

0000000100000f7d  numberWithInt:

0000000100000f8c  description

查询结果是目标文件中被编译的全部方法名列表

$ otool -V -s \_\_DATA \_\_objc\_selrefs TestLinkMapWithOC:

Contents of (\_\_DATA,\_\_objc\_selrefs) section

0000000100001280 \_\_TEXT:\_\_objc\_methname:new

0000000100001288 \_\_TEXT:\_\_objc\_methname:numberWithInt:

0000000100001290 \_\_TEXT:\_\_objc\_methname:callMethod:

0000000100001298 \_\_TEXT:\_\_objc\_methname:description

查询结果是目标文件中所引用到的方法名列表

以上两步求差，得到emptyMethod:age:，即此方法是无用方法

注意，系统API的Protocol可能被列入无用方法名单里，如UITableViewDelegate的方法，我们只需要对这些Protocol里的方法加入白名单过滤即可。

### 查找无用属性

从无用的方法列表中找到以下划线\_开头的字符串，然后在所有方法列表中找到对应的setter和getter，再从引用方法列表中查找对应的setter或者是getter，从而可以标识出属性是只读(readonly)，或者是只写(writeonly)的，或者是属性从未被使用

(适用于精简网络协议库)

### 扫描重复代码

借用simian工具

### 重构代码

设计模式优化代码逻辑结构，用最少的代码完成业务逻辑

### 编译选项

* Strip Link Product设成YES，
* Generate Debug Symbols 设置为NO
* Make Strings Read-Only设为YES，
* 去掉异常支持，Enable C++ Exceptions和Enable Objective-C Exceptions设为NO，并且Other C Flags添加-fno-exceptions，

(可执行文件减少了27M，其中\_\_gcc\_except\_tab段减少了17.3M，\_\_text减少了9.7M，效果特别明显。可以对某些文件单独支持异常，编译选项加上-fexceptions即可。但有个问题，假如ABC三个文件，AC文件支持了异常，B不支持，如果C抛了异常，在模拟器下A还是能捕获异常不至于Crash，但真机下捕获不了)

### Clang插件(高级)

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)。

<http://www.infoq.com/cn/articles/clang-plugin-ios-app-size-reducing>

# 资源文件

## 删除未被使用的资源

以图片举例

原理：扫描项目里面所有png和jpg图片，然后在二进制文件、plist、xib里面匹配文件名字符串，得出图片没被引用

具体方法：在项目工程中，一般使用到图片资源均会在代码中使用字符串常量来引入图片资源。那么通过otool命令逆向\_\_TEXT\_\_cstring段来获取二进制文件中所有的字符串常量，并检查这些字符串常量是否匹配安装包中任意图片资源名（去除文件后缀，如@3x.jpg）。之后，没有被匹配的安装包中的图片资源就标记为疑似无用图片，然后做进一步排查处理。

因为在代码中，通过一些拼接字符串引用的图片资源在此方法中会被标记为无用图片，而实际中是有被使用到，这部分需要开发排查。

## 压缩资源

以图片举例

imageOptim进行无损压缩(主要是如文件的EXIF标签、颜色配置文件、作者等信息)。

## 减少资源使用

* 纯色的背景图片，使用工具获取图片的RGBA值，使用代码描绘出图片。
* 简单的带有文字的图片处理与上述类似

## 按需加载资源(iOS 9)

按需资源是App Store上托管的应用程序内容，与您下载的相关应用程序包是分开的。它们可以实现更小的应用程序捆绑，更快的下载和更丰富的应用程序内容。该应用程序请求一组按需资源，操作系统管理下载和存储。应用程序使用资源，然后释放请求。下载后，资源可能会停留在设备上多个启动周期，使访问更快。

* NSBundleResourceRequest

将资源文件分成三类

1. Initial install tags
2. Prefetch tag order
3. Dowloaded only on demand

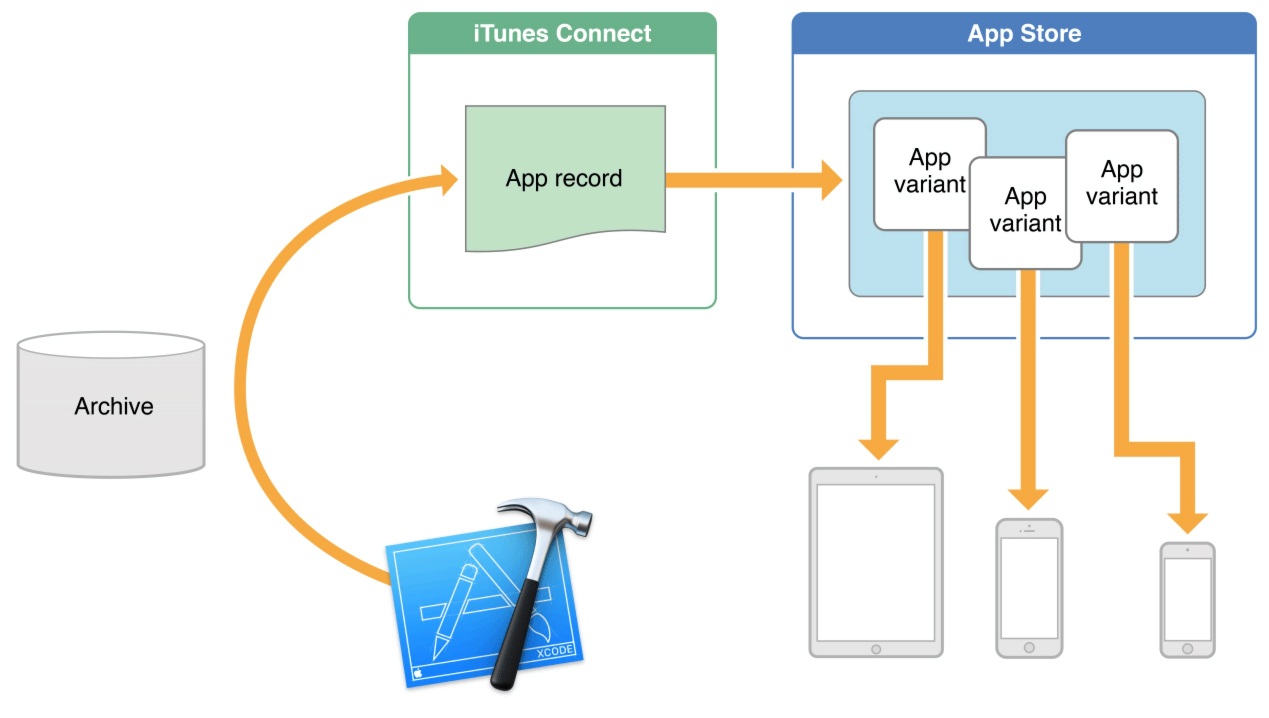
具体参照：[On-Demand Resources Guide](https://developer.apple.com/library/content/documentation/FileManagement/Conceptual/On_Demand_Resources_Guide/index.html" \l "//apple_ref/doc/uid/TP40015083-CH2-SW1)

## 编译选项

1. File Inspector中 Asset Catalog 【Compression】设置
2. Build Setting 中 Enable on Demand Resources设置为YES
3. Build Setting 中 Asset Catalog Compiler 中 Optimization设置为space

# iTC

开发者上传到iTC的App包是包含支持所有设备CPU架构的，支持所有设备分辨率的资源文件，iTC会根据设备的特性归类裁剪上传的App包，当用户从AppStore下载App的时候，会根据当前设备的特性来下载对应CPU架构和分辨率的包，GPU资源根据设备功能进行分割，参照下图



在开发期间Xcode会模拟裁剪

上传开启了bitcode 的App到iTC，允许Apple重新优化App 二进制文件

在构建的时候Xcode默认隐藏了产生的符号

以上是iOS9 or Later支持

详细参照：[App Thinning (iOS, tvOS, watchOS)](https://developer.apple.com/library/content/documentation/IDEs/Conceptual/AppDistributionGuide/AppThinning/AppThinning.html" \l "//apple_ref/doc/uid/TP40012582-CH35-SW1)

# 附录

## protobuf精简改造

protobuf是Google推出的一种轻量高效的结构化数据存储格式，在微信用于网络协议和本地文件序列化。但google默认工具生成的代码比较冗余，像序列化、反序列化、计算序列化大小等方法都生成在具体的pb类里，每个类的实现大同小异。通过代码分析以及结合protobuf原理，要想把这些方法抽象到基类，派生类提供每个字段相关信息就够了：

* field number
* field label, optional, required or repeated
* wire type, double, float, int, etc
* 是否packed
* repeated的数据类型

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | typedef struct {      Byte \_fieldNumber;      Byte \_fieldLabel;      Byte \_fieldType;      BOOL \_isPacked;      int \_enumInitValue;      union {          \_\_unsafe\_unretained NSString\* \_messageClassName;          \_\_unsafe\_unretained Class \_messageClass; // ClassName对应的Class          IsEnumValidFunc \_isEnumValidFunc; // 检测枚举值是否合法函数指针      };  } PBFieldInfo; |

另外通过无用selector列表，发现不少pb类属性的getter或setter没有被使用。原先的pb类属性是用@synthesize修饰，编译器会自动生成getter和setter。如果不想编译器生成，则要用@dynamic。甚至我们可以把pb类的成员变量去掉。做法如下：

* 基类增加id类型数组ivarValues（参考了objc\_class结构体ivars做法），用于存放对象的属性值。对象属性值统一用oc对象表示，如果类型是基础类型（primitive，如int、float等），则用NSValue存
* 重载methodSignatureForSelector:方法，返回属性getter、setter的方法签名
* 重载forwardInvocation:方法，分析invocation.selector类型。如果是getter，从ivarValues获取属性值并设置为invocation的returnValue；如果是setter，从invocation第二个argument获取属性值，并存放到ivarValues里
* 重载setValue:forUndefinedKey:、valueForUndefinedKey:，防止通过KVO访问属性Crash
* 做下性能优化，如pb类在initialize做一次初始化，缓存属性名的hash值，属性的getter、setter方法的objcType等；属性值不用std::map（属性名->属性值），而是改用数组；MRC代替ARC（有些时候ARC自动添加的retain/release挺影响性能的）；等等

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | class PBClassInfo {  public:      PBClassInfo(Class cls, PBFieldInfo\* fieldInfo);      ~PBClassInfo();  public:      unsigned int \_numberOfProperty;      std::string\* \_propertyNames;      size\_t\* \_propertyNameHashes;      std::string\* \_getterObjCTypes;      std::string\* \_setterObjCTypes;      PBFieldInfo\* \_fieldInfos;  };  @interface WXPBGeneratedMessage () {      uint32\_t \_has\_bits\_[3]; // 最多96个属性，表示属性是否有赋值      int32\_t \_serializedSize;      PBClassInfo\* \_classInfo;      id\* \_ivarValues;  }  - (NSMethodSignature\*) methodSignatureForSelector:(SEL) aSelector;  - (void) forwardInvocation:(NSInvocation\*) anInvocation;  - (void) setValue:(id) value forUndefinedKey:(NSString\*) key;  - valueForUndefinedKey:(NSString\*) key;  @end |

把冗余代码去掉后，整个类清爽多了。像GameResourceReq只有3个属性的proto结构体，类方法代码行数由以前的127行变成现在的8行。protobuf精简改造中，精简类方法减少了可执行文件8.8M，去掉类成员变量和类属性改用@dynamic减少了2.5M。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | message GameResourceReq {      required BaseRequest BaseRequest = 1;      required int32 PropsCount = 2;      repeated uint32 PropsIdList = 3[packed=true];  }  // 老实现  @implementation GameResourceReq  @synthesize hasBaseRequest;  @synthesize baseRequest;  @synthesize hasPropsCount;  @synthesize propsCount;  @synthesize mutablePropsIdListList;  @dynamic propsIdList;  - (id) init {...}  - (void) SetBaseRequest:(BaseRequest\*) value {...}  - (void) SetPropsCount:(int32\_t) value {...}  - (NSArray\*) propsIdListList {...}  - (NSMutableArray\*)propsIdList {...}  - (void)setPropsIdList:(NSMutableArray\*) values {...}  - (BOOL) isInitialized {...}  - (void) writeToCodedOutputStream:(PBCodedOutputStream\*) output {...}  - (int32\_t) serializedSize {...}  + (GameResourceReq\*) parseFromData:(NSData\*) data {...}  - (GameResourceReq\*) mergeFromCodedInputStream:(PBCodedInputStream\*) input {...}  - (void) addPropsIdList:(uint32\_t) value {...}  - (void) addPropsIdListFromArray:(NSArray\*) values {...}  @end  // 新实现  @implementation GameResourceReq  PB\_PROPERTY\_TYPE baseRequest;  PB\_PROPERTY\_TYPE opType;  PB\_PROPERTY\_TYPE brandUserName;  + (void) initialize {    static PBFieldInfo \_fieldInfoArray[] = {      {1, FieldLabelRequired, FieldTypeMessage, NO, 0, .\_messageClassName = STRING\_FROM(BaseRequest)},      {2, FieldLabelRequired, FieldTypeInt32, NO, 0, 0},      {3, FieldLabelRepeated, FieldTypeUint32, NO, 0, 0},    };    initializePBClassInfo(self, \_fieldInfoArray);  }  @end |

# 参考

## Segment of mach-o

The \_\_TEXT segment contains executable code and other read-only data.

The \_\_DATA segment contains writable data.

The \_\_OBJC segment contains data used by the Objective-C language runtime support library

The \_\_IMPORT segment contains symbol stubs and non-lazy pointers to symbols not defined in the executable. This segment is generated only for executables targeted for the IA-32 architecture

The \_\_LINKEDIT segment contains raw data used by the dynamic linker, such as symbol, string, and relocation table entries

|  |  |
| --- | --- |
| **Table 1**The sections of a \_\_TEXT segment | |
| **Segment and section name** | **Contents** |
| \_\_TEXT,\_\_text | Executable machine code. The compiler generally places only executable code in this section, no tables or data of any sort. |
| \_\_TEXT,\_\_cstring | Constant C strings. A C string is a sequence of non-null bytes that ends with a null byte ('\0'). The static linker coalesces constant C string values, removing duplicates, when building the final product. |
| \_\_TEXT,\_\_picsymbol\_stub | Position-independent indirect symbol stubs. See “[Position-Independent Code](https://web.archive.org/web/20091111150734/http:/developer.apple.com:80/mac/library/documentation/DeveloperTools/Conceptual/MachOTopics/1-Articles/dynamic_code.html#//apple_ref/doc/uid/TP40002528)” in [*Mach-O Programming Topics*](https://web.archive.org/web/20091111150734/http:/developer.apple.com:80/mac/library/documentation/DeveloperTools/Conceptual/MachOTopics/index.html#//apple_ref/doc/uid/TP40001519) for more information. |
| \_\_TEXT,\_\_symbol\_stub | Indirect symbol stubs. See “[Position-Independent Code](https://web.archive.org/web/20091111150734/http:/developer.apple.com:80/mac/library/documentation/DeveloperTools/Conceptual/MachOTopics/1-Articles/dynamic_code.html#//apple_ref/doc/uid/TP40002528)” in [*Mach-O Programming Topics*](https://web.archive.org/web/20091111150734/http:/developer.apple.com:80/mac/library/documentation/DeveloperTools/Conceptual/MachOTopics/index.html#//apple_ref/doc/uid/TP40001519) for more information. |
| \_\_TEXT,\_\_const | Initialized constant variables. The compiler places all nonrelocatable data declared const in this section. (The compiler typically places uninitialized constant variables in a zero-filled section.) |
| \_\_TEXT,\_\_literal4 | 4-byte literal values. The compiler places single-precision floating point constants in this section. The static linker coalesces these values, removing duplicates, when building the final product. With some architectures, it’s more efficient for the compiler to use immediate load instructions rather than adding to this section. |
| \_\_TEXT,\_\_literal8 | 8-byte literal values. The compiler places double-precision floating point constants in this section. The static linker coalesces these values, removing duplicates, when building the final product. With some architectures, it’s more efficient for the compiler to use immediate load instructions rather than adding to this section. |
| **Table 2**The sections of a \_\_DATA segment | |
| **Segment and section name** | **Contents** |
| \_\_DATA,\_\_data | Initialized mutable variables, such as writable C strings and data arrays. |
| \_\_DATA,\_\_la\_symbol\_ptr | Lazy symbol pointers, which are indirect references to functions imported from a different file. See “[Position-Independent Code](https://web.archive.org/web/20091111150734/http:/developer.apple.com:80/mac/library/documentation/DeveloperTools/Conceptual/MachOTopics/1-Articles/dynamic_code.html#//apple_ref/doc/uid/TP40002528)” in [*Mach-O Programming Topics*](https://web.archive.org/web/20091111150734/http:/developer.apple.com:80/mac/library/documentation/DeveloperTools/Conceptual/MachOTopics/index.html#//apple_ref/doc/uid/TP40001519) for more information. |
| \_\_DATA,\_\_nl\_symbol\_ptr | Non-lazy symbol pointers, which are indirect references to data items imported from a different file. See “[Position-Independent Code](https://web.archive.org/web/20091111150734/http:/developer.apple.com:80/mac/library/documentation/DeveloperTools/Conceptual/MachOTopics/1-Articles/dynamic_code.html#//apple_ref/doc/uid/TP40002528)” in [*Mach-O Programming Topics*](https://web.archive.org/web/20091111150734/http:/developer.apple.com:80/mac/library/documentation/DeveloperTools/Conceptual/MachOTopics/index.html#//apple_ref/doc/uid/TP40001519) for more information. |
| \_\_DATA,\_\_dyld | Placeholder section used by the dynamic linker. |
| \_\_DATA,\_\_const | Initialized relocatable constant variables. |
| \_\_DATA,\_\_mod\_init\_func | Module initialization functions. The C++ compiler places static constructors here. |
| \_\_DATA,\_\_mod\_term\_func | Module termination functions. |
| \_\_DATA,\_\_bss | Data for uninitialized static variables (for example, static int i;). |
| \_\_DATA,\_\_common | Uninitialized imported symbol definitions (for example, int i;) located in the global scope (outside of a function declaration). |

|  |  |
| --- | --- |
| **Table 3**The sections of a \_\_IMPORT segment | |
| **Segment and section name** | **Contents** |
| \_\_IMPORT,\_\_jump\_table | Stubs for calls to functions in a dynamic library. |
| \_\_IMPORT,\_\_pointers | Non-lazy symbol pointers, which are direct references to functions imported from a different file. |

**Note:** Compilers or any tools that create Mach-O files are free to define additional section names. These additional names do not appear in Table 1.

## 文献

[iOS编译过程的原理和应用](http://blog.csdn.net/hello_hwc/article/details/53557308)

[iOS代码瘦身实践](http://www.cnblogs.com/wdsunny/p/7486617.html)

iOS微信安装包瘦身

<http://www.cocoachina.com/ios/20151211/14562.html>

减小ipa体积之删除frameWork中无用mach-O文件

http://jaq.alibaba.com/community/art/show?articleid=229

当我们谈论iOS瘦身的时候，我们到底在谈论些什么

<http://www.jianshu.com/p/fb95238cd511>

手机APP安装包缩减方案

<http://tmq.qq.com/2016/11/mobileapp_reduceprogram-2/>

<http://blog.csdn.net/ce123_zhouwei/article/details/6971544>

<http://www.jianshu.com/p/193ba07dadcf>

Find unused Objective-C imports.

<https://github.com/dblock/fui>

**AppCode**

<http://www.jetbrains.com/objc/>

<http://cdn2.jianshu.io/p/a53480ad0364>

Mach-O Programming Topics

<https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/MachOTopics/1-Articles/dynamic_code.html>

Mac OS X ABI Mach-O File Format Reference

[https://web.archive.org/web/20091111150734/http://developer.apple.com:80/mac/library/documentation/DeveloperTools/Conceptual/MachORuntime/Reference/reference.htmll](http://developer.apple.com:80/mac/library/documentation/DeveloperTools/Conceptual/MachORuntime/Reference/reference.html)

OS X Assembler Reference

<https://developer.apple.com/library/content/documentation/DeveloperTools/Reference/Assembler/000-Introduction/introduction.html>

mach-o文件格式分析

<http://turingh.github.io/2016/03/07/mach-o%E6%96%87%E4%BB%B6%E6%A0%BC%E5%BC%8F%E5%88%86%E6%9E%90/>

PARSING MACH-O FILES

<https://lowlevelbits.org/parsing-mach-o-files/>