

对于一个操作系统来说，对其影响最大的莫过于运行其上的软件了。正因为有了这些形形色色的软件，操作系统才得以广泛应用。与Linux、Windows等主流操作系统一样，运行在macOS上的软件也有着独有的特点。

macOS上的软件有着独特的UI界面与操作方式。macOS的设计师一直秉承着独特的设计理念，设计出的Aqua界面别具风格。银灰的金属色主题是分辨macOS系统最快捷的方式。Windows界面通常将最小化按钮与关闭按钮设置到窗口右上角，而在Aqua界面上则位于软件的左上角，并且使用全屏按钮替换了Windows上的最大化按钮。进入全屏模式的软件会新开一个窗口，占据屏幕的全部空间，可以在触摸板上使用3根手指左右划动来切换不同的全屏窗口。

每个常规的Aqua界面上的程序都有一个菜单，菜单显示在用户屏幕的顶端，菜单的最左边始终是一个苹果图标，点击它会弹出系统设置、App Store、关机、重启等多个选项。

macOS使用Dock栏来管理显示常用的软件，它位于用户屏幕的底部，展示效果与Windows的任务栏类似。正在运行的macOS界面程序都可以在Dock上点击右键，在弹出的菜单中选择Options→Keep in Dock，将程序在Dock上保留下来，以后就可以直接从Dock中点击图标启动程

质上是一个文本文件，在脚本文件中指定运行它的解释器后，Shell在运行脚本时会调用解释器来解释运行它。任何一个文件都可以通过执行`chmod+x`命令给它加上可执行权限，拥有可执行权限的文件并不一定能够执行，只有满足某种解释器的语法规则才算得上是可执行文件。

除了主流的几种Shell脚本支持外，macOS还内置了目前比较流行的Perl、Python和Ruby等脚本的解释器，任何人都可以通过编写这些脚本并调用它们的解释器来运行，不需要安装额外的软件。

另外，苹果公司还开发了一种脚本：AppleScript（苹果脚本）。它用于运行macOS上的程序并实行自动化工作。苹果提供了一个单独的脚本编辑器来开发与调试AppleScript。它位于`/Applications/Utilities/Script Editor`中。使用Script Editor编写好的脚本保存的格式为`scpt`。随着macOS系统的不断升级，AppleScript也在不停地发展，目前甚至可以使用AppleScript来开发macOS上的界面程序。具体的步骤是在Xcode中选择`File→New→Project`，在打开的对话框中选择`Other→Cocoa-AppleScript`。有兴趣的读者可以阅读官方文档来深入了解AppleScript^①。

除了脚本外，可执行文件还包括可以由用户主动执行的程序与被动执行的程序文件。

主动执行的程序包括：GUI界面程序、命令行程序、游戏等；被动执行的程序包括被系统调用的程序，如Quick Look插件、屏幕保护程序、内核驱动与扩展等，以及被程序调用的框架、库、Bundle、XPC服务等。所有这些可执行文件都使用苹果独有的可执行文件格式：Mach-O。关于Mach-O文件的格式，我们将在本章后面的小节中进行详细讲解。

① <https://developer.apple.com/library/mac/documentation/AppleScript/Conceptual/AppleScriptX/AppleScriptX.html>。

4.2 下载与安装软件

苹果支持从App Store应用商店直接安装软件,也支持从第三方渠道(如其他磁盘介质、网络)下载来安装软件。从App Store下载软件是最方便快捷也是最安全的一种方式,苹果公司一向以软件审查严格闻名,应用商店中的应用软件在界面、功能以及对系统资源的使用上,都是经过严格审查与限制的。对于普通用户来说,这种下载方式最合适不过了。但对于专业用户来说,由于一些专业软件因为用户授权协议、资源访问或其他原因未能在App Store上架,就只能通过网络或第三方渠道来获取了。

4

4.2.1 免费与付费软件

App Store应用商店提供了丰富的免费与付费的软件供用户下载使用。下载免费的软件不需要用户付出额外的成本,只需要到官网<https://appleid.apple.com/>注册一个账号,登录App Store就可以下载了。

如果要下载App Store中的收费软件,则需要为账号绑定一张银行卡,购买软件成功后会直接从银行卡中扣取费用。另外,App Store中的软件还支持另一种收费方式:In-App Purchase(应用内付费),简称IAP。这种收费方式在苹果自家的iOS系统中应用非常广泛,它允许开发人员将软件的某些特定功能设定为需要购买才能使用。目前,很多软件开发商以此平台作为主要的软件收入来源。

App Store中的收费软件只有IAP与直接购买这两种方式,而网络下载的收费软件的付费形式

则丰富很多。部分软件官网只提供软件基础功能的演示版本供用户下载,如果需要使用正式版本,则需要联系软件开发商购买完整版本。例如著名的反汇编软件IDA Pro,官网就只提供了Demo版本可供下载^①,正式版本需要找软件开发商或代理商购买。也有些软件的官网会提供完整版本下载,但会有使用时间或功能限制。如果需要正常无限制使用软件,则需要购买软件授权,如反汇编软件Hopper^②。还有部分软件与传统Windows付费软件一样,使用用户名与注册码的形式来售卖软件,如010 Editor^③。

4.2.2 安装软件

普通的macOS软件只是一个以扩展名.app结尾的目录,这种目录有着特定的组织结构,macOS称之为Bundle。安装这类软件只需要将Bundle复制到系统的/Applications目录下即可。复制完成后,Launchpad面板会自动更新安装好的软件图标,启动软件不需要到/Applications目录中寻找,只需要打开Launchpad,找到软件的图标,点击就可以运行该程序了。

从网络上下载的软件通常是经过打包后发布的,普通的软件常常通过zip或其他方式压缩,以压缩包的形式提供下载,还有的使用磁盘工具将软件打包成一个dmg磁盘镜像文件,这类dmg文件内通常还会有一个Applications目录的软链接,安装的时候,只需要将dmg中的软件拖放到该软链接上就算完成安装了。图4-2是AppDelete的安装镜像。

① Demo下载地址: https://www.hex-rays.com/products/ida/support/download_demo.shtml。

② Hopper下载地址: <http://hopperapp.com/>。

③ 010 Editor下载地址: <http://www.sweetscape.com/010editor/>。



图4-2 AppDelete的安装镜像

macOS上的软件还有一种以pkg或mpkg结尾的安装包，类似于Windows系统上的msi或exe安装程序，通过不停地点击下一步就可以完成安装，这类软件除了将主程序写入/Applications目录外，一般还会在系统上做一些只有具有管理员权限才能完成的动作，比如，为特定的目录或文件创建软链接、安装与卸载内核扩展、复制命令行程序到用户可执行文件目录/usr/local/bin中，等等。因此，在安装的过程中，可能会提示输入管理员用户名与密码来执行需要Root权限的操作。

另一种是命令行工具，这类程序的安装使用第1章中介绍的Homebrew即可，此处不再赘述。

4.3 Bundle

Bundle是苹果系统独有的特色，苹果系统中大量使用了Bundle。本节我们会简要介绍Bundle目录结构以及如何在代码中访问Bundle。

4.3.1 Bundle 目录结构

安装到macOS系统上的软件有着特定的格式，通常是以.app扩展名结尾的Bundle目录结构。

Bundle有着固定的组织格式，在Finder中查看Bundle的目录内容，可以在程序上点击右键，在弹出的菜单中选择Show Package Contents。例如，/Applications目录下的App Store的目录结构如图4-3所示。

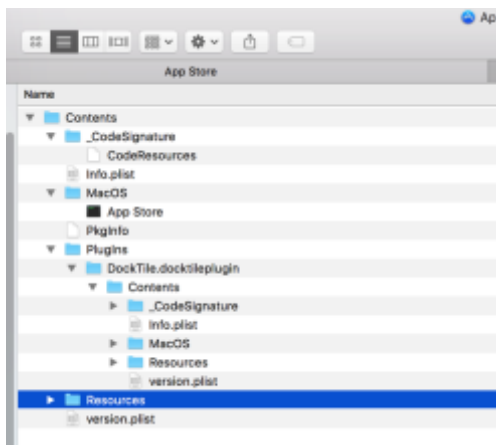


图4-3 App Store目录结构

在App Store.app目录下，只有一个Contents子目录，所有软件的内容都在此目录下。

- ❑ CodeSignature目录。此目录下只有一个CodeResources文件，它是一个plist格式的文件，保存了软件包中所有文件的签名信息。
- ❑ Info.plist文件。此文件记录了软件的一些信息，如软件构建的机器版本BuildMachineOSBuild、可执行文件名CFBundleExecutable、软件的标识CFBundleIdentifier、软件包名CFBundleName等。
- ❑ MacOS目录。此目录存放了可执行文件。
- ❑ PkgInfo文件。软件包的8字节标识符。

- ❑ Resources目录。软件运行所需要的资源，包括.lproj本地化资源、.nib资源、图片、字体、声音、文档以及其他文件。
- ❑ Plugins目录。插件目录，存放了软件用到的插件。插件也是使用Bundle目录结构进行组织的一种程序。

除了Plugins目录外，根据软件需求与实现的不同，Contents下可能还会有Frameworks与XPCServices目录。其中，Frameworks里存放了软件需要用到的框架，它是以.framework结尾的Bundle结构；XPCServices则存放了软件用到的XPC服务，它是以.xpc结尾的Bundle结构。还有一种以.bundle扩展名结尾的Bundle，它是“纯粹”的Bundle，目录结构与其他Bundle无异，存放的内容可以是二进制代码，可以是资源，也可以二者同时存放。如果存放二进制代码的话，可供程序在代码中使用dlopen()函数打开，这种Bundle通常用于制作软件的插件，存放在软件Bundle的Resources目录下。

4.3.2 在代码中访问 Bundle

在程序中，开发人员可以使用Cocoa框架提供的NSBundle类来获取程序的Bundle信息。调用NSBundle的mainBundle()方法可以返回当前程序的主Bundle对象，调用方法如下：

```
NSBundle *bundle = [NSBundle mainBundle]
```

使用主Bundle对象的infoDictionary()方法可以访问软件Bundle目录下Info.plist文件中的信息，它返回的是一个字典对象。以下是获取程序标识符的代码：

```
[[[NSBundle mainBundle] infoDictionary] objectForKey:@"CFBundleIdentifier"]]
```


使用主Bundle对象的pathForResource()方法可以访问Bundle目录下任意资源文件。以下是访问Bundle根目录下的monkey.png文件：

```
NSString *monkey = [[NSBundle mainBundle] pathForResource:@"monkey" ofType:@"png"];
```

与主Bundle对应的是自定义Bundle，这一类Bundle的访问可以使用以下方式调用：

```
NSString *resourceBundle = [[NSBundle mainBundle] pathForResource:@"ResPack" ofType:@"bundle"];
NSLog(@"resourceBundle: %@", resourceBundle);
NSString *monkey = [[NSBundle bundleWithPath:resourceBundle] pathForResource:@"monkey"
                                     ofType:@"png" inDirectory:@"Images"];

NSLog(@"monkey path: %@", monkey);
```

这段代码访问了ResPack.bundle中Images目录下的monkey.png文件。

4.4 通用二进制格式

虽然macOS系统使用了很多UNIX上的特性，但它并没有使用ELF作为系统的可执行文件格式，而是使用独创的Mach-O文件格式。

macOS系统支持的CPU及硬件平台发生了很大的变化，从早期的PowerPC平台，到后来的x86，再到现在主流的ARM、x86-64平台。软件开发人员为了保证不同硬件平台的兼容性，需要为每一个平台编译一个可执行文件，这是非常繁琐的。为了解决软件在多个硬件平台上的兼容性问题，苹果开发了一个通用的二进制文件格式（Universal Binary），又称为胖二进制（Fat Binary）。通用二进制文件将多个支持不同CPU架构的二进制文件打包成一个文件，系统在加载运行该程序时，会根据通用二进制文件中提供的多个架构来与当前系统平台做匹配，运行适合当前系统的那个版本。

苹果系统中存在着很多通用二进制文件，比如/usr/bin/python，在终端中执行file命令可以查看它的信息：

```
$ file /usr/bin/python
/usr/bin/python: Mach-O universal binary with 2 architectures
/usr/bin/python (for architecture x86_64):  Mach-O 64-bit executable x86_64
/usr/bin/python (for architecture i386):  Mach-O executable i386
```

系统提供了一个命令行工具lipo来操作通用二进制文件。它可以添加、提取、删除以及替换通用二进制文件中特定架构的二进制版本。例如提取python中x86_64版本的二进制文件可以执行：

```
lipo /usr/bin/python -extract x86_64 -output ~/Desktop/python.x64
```

删除x86版本的二进制文件可以执行：

```
lipo /usr/bin/python -remove i386 -output ~/Desktop/python.x64
```

或者直接瘦身为x86_64版本：

```
lipo /usr/bin/python -thin x86_64 -output ~/Desktop/python.x64
```

通用二进制文件不止针对可以直接运行的可执行程序，系统中的动态库dylib、静态库.a文件以及框架等都可以是通用二进制文件，对它们同样可以使用lipo命令来进行管理。

来看一下通用二进制的文件格式。安装好macOS程序开发的SDK，或者在xnu的内核源代码中，都可以在文件中找到通用二进制文件格式的声明。从文件命名上看，将通用二进制称为胖二进制更方便一些。胖二进制头部结构fat_header定义如下：

```
#define FAT_MAGIC    0xcafebabe
#define FAT_CIGAM    0xbefafeca

struct fat_header {
    uint32_t    magic;
    uint32_t    nfat_arch;
```

```
};
```

magic字段被定义为常量FAT_MAGIC，它的取值是固定的：0xcafebabe，表示这是一个通用的二进制文件。nfat_arch字段指明了通用二进制中包含多少个Mach-O文件。

每个通用二进制架构信息都使用fat_arch结构表示，在fat_header结构体之后，紧接着是一个或多个连续的fat_arch结构体，它的定义如下：

```
struct fat_arch {
    cpu_type_t  cputype;
    cpu_subtype_t  cpusubtype;
    uint32_t  offset;
    uint32_t  size;
    uint32_t  align;
};
```

cputype指定了具体的CPU类型，它的类型是cpu_type_t，定义位于mach/machine.h中。CPU的

常用类型主要有如下几种：

```
#define CPU_TYPE_X86      ((cpu_type_t) 7)
#define CPU_TYPE_I386     CPU_TYPE_X86
#define CPU_TYPE_X86_64   (CPU_TYPE_X86 | CPU_ARCH_ABI64)
#define CPU_TYPE_MC98000  ((cpu_type_t) 10)
#define CPU_TYPE_HPPA     ((cpu_type_t) 11)
#define CPU_TYPE_ARM      ((cpu_type_t) 12)
#define CPU_TYPE_ARM64     (CPU_TYPE_ARM | CPU_ARCH_ABI64)
#define CPU_TYPE_MC88000  ((cpu_type_t) 13)
#define CPU_TYPE_SPARC     ((cpu_type_t) 14)
#define CPU_TYPE_I860     ((cpu_type_t) 15)
#define CPU_TYPE_POWERPC  ((cpu_type_t) 18)
#define CPU_TYPE_POWERPC64 (CPU_TYPE_POWERPC | CPU_ARCH_ABI64)
```

在macOS平台上的CPU类型一般为CPU_TYPE_X86_64。

cpusubtype指定了CPU的子类型为cpu_subtype_t。CPU子类型主要有如下几种。

```
#define CPU_SUBTYPE_MASK  0xff000000
#define CPU_SUBTYPE_LIB64 0x80000000
#define CPU_SUBTYPE_X86_ALL ((cpu_subtype_t)3)
#define CPU_SUBTYPE_X86_64_ALL ((cpu_subtype_t)3)
#define CPU_SUBTYPE_X86_ARCH1 ((cpu_subtype_t)4)
#define CPU_SUBTYPE_X86_64_H ((cpu_subtype_t)8)
```

.....

其中，CPU_SUBTYPE_LIB64与CPU_SUBTYPE_X86_64_ALL比较常见。

offset字段指明了当前CPU架构数据相对于当前文件开头的偏移值，size字段指明了数据的大小。align字段指明了数据的内存对齐边界，取值必须是2的次方，它确保了当前CPU架构的目标文件在加载到内存中时，数据是经过内存优化对齐的。

可以使用otool工具打印本机安装的python程序的fat_header信息，如下所示：

```
otool -f -V /usr/bin/python
Fat headers
fat_magic FAT_MAGIC
nfat_arch 2
architecture i386
  cputype CPU_TYPE_I386
  cpusubtype CPU_SUBTYPE_I386_ALL
  capabilities 0x0
  offset 4096
  size 29632
  align 2^12 (4096)
architecture x86_64
  cputype CPU_TYPE_X86_64
  cpusubtype CPU_SUBTYPE_X86_64_ALL
  capabilities CPU_SUBTYPE_LIB64
  offset 36864
  size 29872
  align 2^12 (4096)
```

如果你使用UNIX，并且经常使用GNU里面的binutils提供的objdump查看可执行文件信息的话，那么在macOS上可以使用它的移植版本gobjdump，使用HomeBrew运行以下命令进行安装：

```
$ brew install binutils
```

完装完成后，执行以下命令可以查看python程序的fat_header信息：

```
$ gobjdump -f /usr/bin/python
In archive /usr/bin/python:

i386:   file format mach-o-i386
architecture: i386, flags 0x00000012:
EXEC_P, HAS_SYMS
```

```
start address 0x00001be0

i386:x86-64:  file format mach-o-x86-64
architecture: i386:x86-64, flags 0x00000012:
EXEC_P, HAS_SYMS
start address 0x0000000100000e20
```

fat_arch结构体往下就是具体的Mach-O文件格式了，这部分内容比较复杂，我们将在下一节进行详细讨论。

4.5 Mach-O 文件格式

Mach-O (Mach Object File Format) 描述了macOS系统上可执行文件的格式。熟悉Mach-O文件格式有助于了解苹果底层软件的运行机制，更好地掌握dyld加载Mach-O的步骤，为动手开发Mach-O相关的加解密工具打下良好的基础。

4.5.1 Mach-O 简介

一个典型的Mach-O文件格式如图4-4所示。

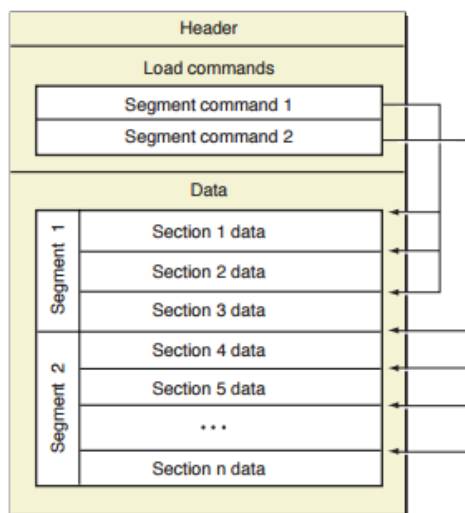


图4-4 Mach-O文件格式

可以看出，Mach-O主要由以下3部分组成。

- ❑ Mach-O头部（mach header）。描述了Mach-O的CPU架构、文件类型以及加载命令等信息。
- ❑ 加载命令（load command）。描述了文件中数据的具体组织结构，不同的数据类型使用不同的加载命令表示。
- ❑ Data。Data中每个段（segment）的数据都保存在这里，段的概念与ELF文件中段的概念类似。每个段都有一个或多个Section，它们存放了具体的数据与代码。

4.5.2 Mach-O 头部

与Mach-O文件格式有关的结构体可以直接或间接地在mach-o/loader.h文件中找到。32位与64位架构的CPU，分别使用了mach_header与mach_header_64结构体来描述Mach-O头部。

mach_header结构体的定义如下：

```
struct mach_header {  
    uint32_t  magic;  
    cpu_type_t  cputype;  
    cpu_subtype_t  cpusubtype;  
    uint32_t  filetype;  
    uint32_t  ncmds;  
    uint32_t  sizeofcmds;  
    uint32_t  flags;  
};
```

这里的magic字段与fat_header结构体中的magic字段一样，表示Mach-O文件的魔数值。对于32位架构的程序来说，它的取值是MH_MAGIC，固定为0xfeedface。

cputype与cpusubtype字段与fat_header结构体中的含义完全相同。

filetype字段表示Mach-O的具体文件类型。它的取值如下所示：

```
#define MH_OBJECT 0x1  
#define MH_EXECUTE 0x2  
#define MH_FVMLIB 0x3  
#define MH_CORE 0x4  
#define MH_PRELOAD 0x5  
#define MH_DYLIB 0x6  
#define MH_DYLINKER 0x7  
#define MH_BUNDLE 0x8  
#define MH_DYLIB_STUB 0x9  
#define MH_DSYM 0xa  
#define MH_KEXT_BUNDLE 0xb
```

本节主要关注MH_EXECUTE与MH_DYLIB这两种文件格式。

接下来，ncmds指明了Mach-O文件中加载命令的数量。sizeofcmds字段指明了Mach-O文件加载命令所占的总字节大小。flags字段表示文件标志，它是一个含有一组位标志的整数，指明了Mach-O文件的一些标志信息，可用的值如下所示：

```
#define MH_NOUNDEFS 0x1  
#define MH_INCRLINK 0x2
```

```
#define MH_DYLDLINK 0x4
#define MH_LAZY_INIT 0x40
#define MH_TWOLEVEL 0x80
#define MH_PIE 0x200000
.....
```

64位Mach-O的mach_header_64结构体定义如下：

```
struct mach_header_64 {
    uint32_t    magic;
    cpu_type_t  cputype;
    cpu_subtype_t  cpusubtype;
    uint32_t    filetype;
    uint32_t    ncmds;
    uint32_t    sizeofcmds;
    uint32_t    flags;
    uint32_t    reserved;
};
```

相比mach_header，它多了一个reserved字段，目前它的取值系统保留。mach_header_64结构体中的字段与mach_header中的基本一致，不同的是magic字段的取值是MH_MAGIC_64，固定的值为0xfeedfacf。

学习Mach-o文件格式时，可以使用辅助工具查看具体的文件结构，效果会更加直观。图4-5是使用MachOView查看optool程序Mach64 Header的效果，图4-6是使用010 Editor查看optool程序Mach64 Header的效果，图4-7是使用Synalyze It!查看optool程序Mach64 Header的效果。

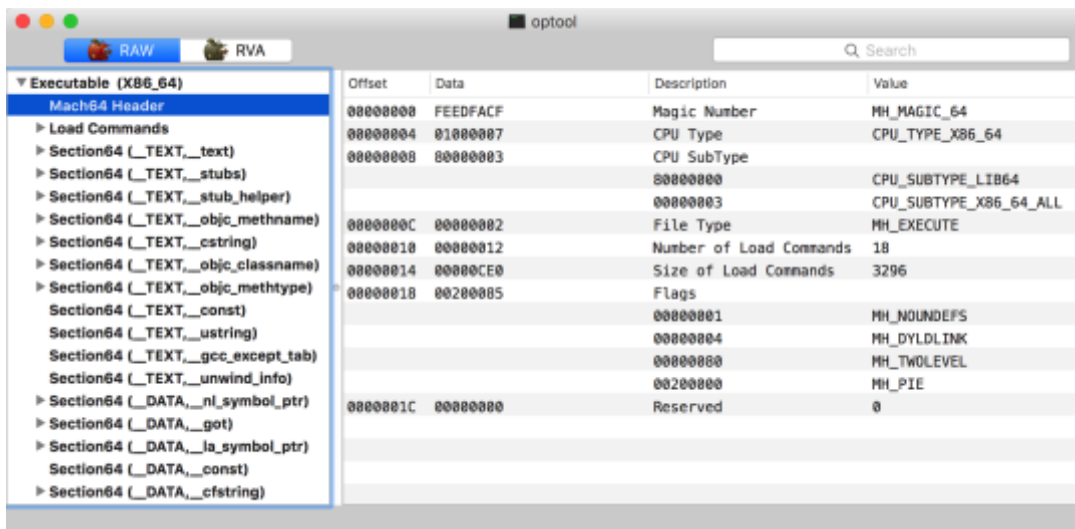


图4-5 MachOView查看的效果

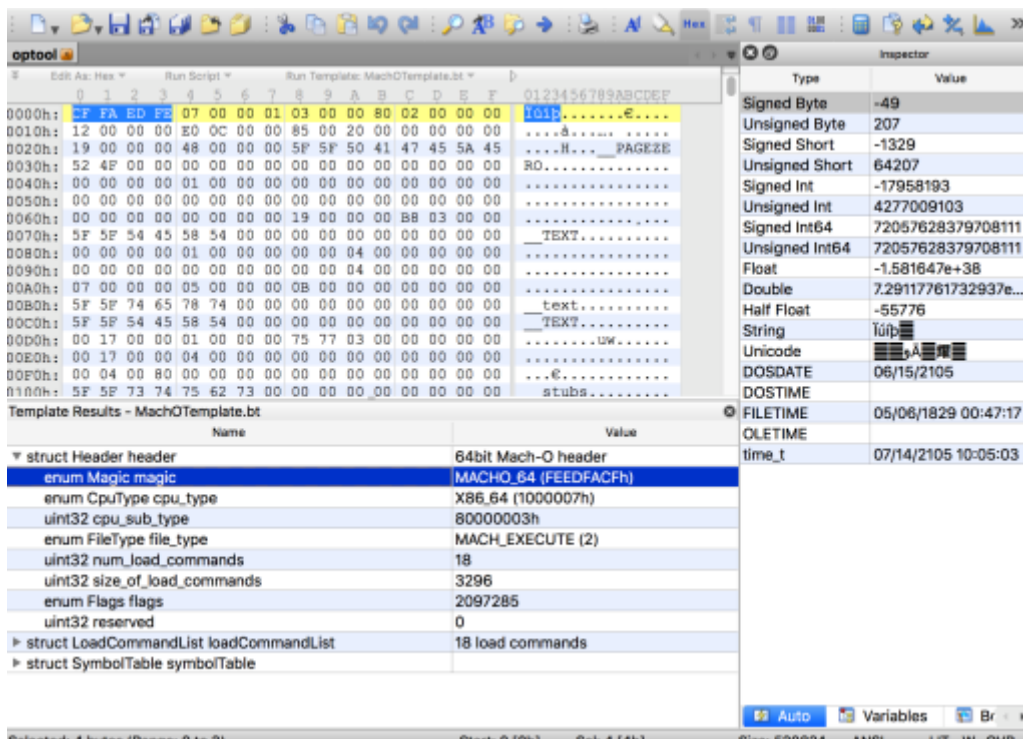


图4-6 010 Editor查看的效果

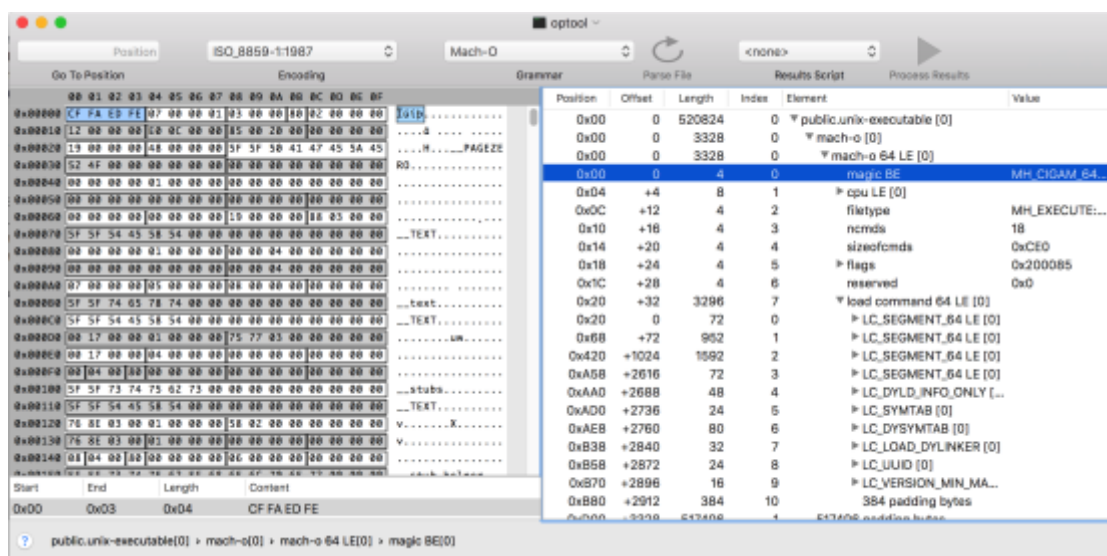


图4-7 Syanalyze It!查看的效果

这三款工具对于学习Mach-O文件格式都是非常有帮助的，建议读者在实际分析时可以适当使用。

4.5.3 加载命令

在mach_header之后是Load Command加载命令，这些加载命令在Mach-O文件加载解析时，会被内核加载器或者动态链接器调用，基本的加载命令的数据结构如下：

```
struct load_command {
    uint32_t cmd; /* type of load command */
    uint32_t cmdsize; /* total size of command in bytes */
};
```

此结构对应的成员只有两个：cmd字段代表当前加载命令的类型，cmdsize字段代表当前加载命令的大小。

cmd的类型不同，所代表的加载命令的类型就不同，它的结构体也会有所不同。不同类型的

加载命令会在load_command结构体后面加上一个或多个字段来表示特定的结构体信息。

在macOS系统进化的过程中,加载命令是更新比较频繁的一个数据结构体。截至macOS 10.12

系统,加载命令的类型cmd的取值共有48种,它们的部分定义如下:

```
#define LC_SEGMENT 0x1
#define LC_SYMTAB 0x2
#define LC_SYMSEG 0x3
#define LC_THREAD 0x4
#define LC_UNIXTHREAD 0x5
#define LC_LOADFVMLIB 0x6
#define LC_IDFVMLIB 0x7
#define LC_IDENT 0x8
#define LC_FVMFILE 0x9
#define LC_PREPAGE 0xa
#define LC_DYSYMTAB 0xb
#define LC_LOAD_DYLIB 0xc
.....
#define LC_ENCRYPTION_INFO_64 0x2C
#define LC_LINKER_OPTION 0x2D
#define LC_LINKER_OPTIMIZATION_HINT 0x2E
#ifndef __OPEN_SOURCE__
#define LC_VERSION_MIN_TVOS 0x2F
#endif
#define LC_VERSION_MIN_WATCHOS 0x30
```

所有这些加载命令由系统内核加载器直接使用,或由动态链接器处理。其中几个常见的加载命令为LC_SEGMENT、LC_LOAD_DYLINKER、LC_LOAD_DYLIB、LC_MAIN、LC_CODE_SIGNATURE、LC_ENCRYPTION_INFO等。下面我们分别进行介绍。

❑ LC_SEGMENT:表示这是一个段加载命令,需要将它加载到对应的进程空间中。段加载

命令将在下一节讨论。

❑ LC_LOAD_DYLIB:表示这是一个需要动态加载的链接库。它使用dylib_command结构体表

示。定义如下:

```
struct dylib_command {
```

```
uint32_t cmd;
uint32_t cmdsize;
struct dylib dylib;
};
```

□ 当 cmd 类型是 LC_ID_DYLIB、LC_LOAD_DYLIB、LC_LOAD_WEAK_DYLIB 与

LC_REEXPORT_DYLIB时，统一使用dylib_command结构体表示。使用dylib结构体来存储要

加载的动态库的具体信息，如下所示：

```
struct dylib {
    union lc_str name;
    uint32_t timestamp;
    uint32_t current_version;
    uint32_t compatibility_version;
};
```

name字段是动态库的完整路径，动态链接器在加载动态库时，会通过此路径进行加载。

timestamp字段描述了动态库构建时的时间戳。current_version与compatibility_version指明了

当前版本与兼容的版本号。

□ LC_MAIN：此加载命令记录了可执行文件的主函数main()的位置，它使用entry_point_

command结构体表示。定义如下：

```
struct entry_point_command {
    uint32_t cmd;
    uint32_t cmdsize;
    uint64_t entryoff;
    uint64_t stacksize;
};
```

entryoff字段指定了main()函数的文件偏移。stacksize指定了初始的堆栈大小。

4.5.4 LC_CODE_SIGNATURE

LC_CODE_SIGNATURE是代码签名加载命令，描述了Mach-O的代码签名信息，它属于链接信

息，使用linkedit_data_command结构体表示。定义如下：

```
struct linkedit_data_command {  
    uint32_t  cmd;  
    uint32_t  cmdsize;  
    uint32_t  dataoff;  
    uint32_t  datasize;  
};
```

dataoff字段指明了相对于__LINKEDIT段的文件偏移位置，datasize字段指明了数据的大小。

那么，如何删除Mach-O中包含的代码签名信息呢？与代码签名相关的数据定义可以在xnu内核代码的bsd/sys/codesign.h文件中找到。整个代码签名部分的头部使用一个CS_SuperBlob结构体定义，它的原型如下：

```
typedef struct __SC_SuperBlob {  
    uint32_t magic;  
    uint32_t length;  
    uint32_t count;  
    CS_BlobIndex index[];  
} CS_SuperBlob;
```

magic字段指明了Blob的类型，可选值如下：

```
enum {  
    CSMAGIC_REQUIREMENT = 0xfade0c00,  
    CSMAGIC_REQUIREMENTS = 0xfade0c01,  
    CSMAGIC_CODEDIRECTORY = 0xfade0c02,  
    CSMAGIC_EMBEDDED_SIGNATURE = 0xfade0cc0,  
    CSMAGIC_EMBEDDED_SIGNATURE_OLD = 0xfade0b02,  
    CSMAGIC_EMBEDDED_ENTITLEMENTS = 0xfade7171,  
    CSMAGIC_DETACHED_SIGNATURE = 0xfade0cc1,  
    CSMAGIC_BLOBWRAPPER = 0xfade0b01,  
}
```

对于第一个Blob来说，它的值必定是CSMAGIC_EMBEDDED_SIGNATURE，表示代码签名采用的是嵌入式的签名信息。

length字段指明了整个SuperBlob的大小，其中包含了即将介绍的CodeDirectory、Requirement

与Entitlement的大小。count字段指明了接下来会有多少个子条目。从index开始，就是每一个子条目的索引了，它的结构是CS_BlobIndex，定义如下：

```
typedef struct __BlobIndex {
    uint32_t type;
    uint32_t offset;
} CS_BlobIndex;
```

type指明了子条目的类型，可选值如下：

```
CSSLOT_CODEDIRECTORY = 0,
CSSLOT_INFOSLOT = 1,
CSSLOT_REQUIREMENTS = 2,
CSSLOT_RESOURCEDIR = 3,
CSSLOT_APPLICATION = 4,
CSSLOT_ENTITLEMENTS = 5,
CSSLOT_SIGNATURESLOT = 0x10000,
```

offset字段指明了子条目距离代码签名数据起始的文件偏移。

通常，对于签名后的程序，签名数据的第一个子条目指向的是一个 type 为

CSSLOT_CODEDIRECTORY的结构，它是一个CS_CodeDirectory结构体，定义如下：

```
typedef struct __CodeDirectory {
    uint32_t magic;
    uint32_t length;
    uint32_t version;
    uint32_t flags;
    uint32_t hashOffset;
    uint32_t identOffset;
    uint32_t nSpecialSlots;
    uint32_t nCodeSlots;
    uint32_t codeLimit;
    uint8_t hashSize;
    uint8_t hashType;
    uint8_t platform;
    uint8_t pageSize;
    uint32_t spare2;
    uint32_t scatterOffset;
    uint32_t teamOffset;
} CS_CodeDirectory;
```

该结构体数据字段较多，此处只关注与签名相关的字段。hashOffset指明了hash数据的文件相

对偏移（注意是相对于当前结构体CS_CodeDirectory）。hashType与hashSize指明了代码签名时使用的算法和每一项签名数据的长度，目前macOS使用的签名算法是SHA-1，长度为20字节。

nSpecialSlots与nCodeSlots指定了代码签名数据条目的个数，前者针对代码签名中所有的Blob，后者针对程序文件内容。codesign程序在对程序进行签名时，会对SuperBlob中每个子条目进行签名，即对Blob的内容调用SHA-1算法取Hash值，nSpecialSlots的值就是子条目Blob的个数。同时，codesign会以pageSize字段指定的页大小为单位（通常取值是0x1000）对程序数据进行签名，每一页签名后生成一条签名数据，nCodeSlots的值就是签名数据的页数，即程序数据大小除以pageSize字段后的值。

在CS_CodeDirectory之后，就是Requirements了，它是一个CS_SuperBlob结构体，指明了Requirement的个数和每一个的偏移。接下来就是每一个Requirement数据了，它是一个CS_GenericBlob结构体，定义如下：

```
typedef struct __SC_GenericBlob {
    uint32_t magic;
    uint32_t length;
    char data[];
} CS_GenericBlob;
```

可以看到，前两个字段与CS_SuperBlob是一样的，只是后面多了一个data字段，用来存放Blob的数据长度。

在Requirement数据下面，就是Entitlement了，它同样是CS_GenericBlob结构。拿本机Calculator计算器程序来说，它的Entitlement的数据内容是一个xml文件，提取出来的内容如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
```

```

<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>com.apple.security.app-sandbox</key>
  <true/>
  <key>com.apple.security.files.user-selected.read-write</key>
  <true/>
  <key>com.apple.security.network.client</key>
  <true/>
  <key>com.apple.security.print</key>
  <true/>
</dict>
</plist>

```

最后一个Blob通常是签名使用的证书，Certificates签名证书也是CS_GenericBlob结构，提取它

4

的证书数据后保存为cer文件，使用macOS的文件预览证书内容，效果如图4-8所示。



图4-8 使用macOS的文件预览证书内容

下面再来看看系统是如何实施代码签名验证的。内核加载解析Mach-O加载命令的函数是 `parse_machfile()`，位于内核代码 `XNU_SRC_Root/bsd/kern/mach_loader.c` 文件中，部分代码片段如下：

```
static load_return_t parse_machfile(
    struct vnode      *vp,
    vm_map_t         map,
    thread_t         thread,
    struct mach_header *header,
    off_t             file_offset,
    off_t             macho_size,
    int               depth,
    int64_t           aslr_offset,
    int64_t           dyld_aslr_offset,
    load_result_t     *result
)
{
    uint32_t          ncmds;
    struct load_command *lcp;
    struct dylinker_command *dlp = 0;
    integer_t          dlarchbits = 0;
    void *              control;
    load_return_t       ret = LOAD_SUCCESS;
    caddr_t             addr;
    void *              kl_addr;
    vm_size_t           size, kl_size;
    size_t              offset;
    size_t              oldoffset;
    int                 pass;
    proc_t              p = current_proc();
    int                 error;
    int resid=0;
    size_t              mach_header_sz = sizeof(struct mach_header);
    boolean_t           abi64;
    boolean_t           got_code_signatures = FALSE;
    int64_t             slide = 0;

    if (header->magic == MH_MAGIC_64 ||
        header->magic == MH_CIGAM_64) {
        mach_header_sz = sizeof(struct mach_header_64);
    }
    .....

    case LC_CODE_SIGNATURE:
        if (pass != 1)
            break;
```

```

ret = load_code_signature(
    (struct linkedit_data_command *) lcp,
    vp,
    file_offset,
    macho_size,
    header->cputype,
    result);
if (ret != LOAD_SUCCESS) {
    printf("proc %d: load code signature error %d "
        "for file \"%s\"\\n",
        p->p_pid, ret, vp->v_name);
    if (!lcs_enforcement(NULL))
        ret = LOAD_SUCCESS;

} else {
    got_code_signatures = TRUE;
}

if (got_code_signatures) {
    unsigned tainted = CS_VALIDATE_TAINTED;
    boolean_t valid = FALSE;
    struct cs_blob *blobs;
    vm_size_t off = 0;

    if (cs_debug > 10)
        printf("validating initial pages of %s\\n", vp->v_name);
    blobs = ubc_get_cs_blobs(vp);

    while (off < size && ret == LOAD_SUCCESS) {
        tainted = CS_VALIDATE_TAINTED;

        valid = cs_validate_page(blobs,
            NULL,
            file_offset + off,
            addr + off,
            &tainted);
        if (!valid || (tainted & CS_VALIDATE_TAINTED)) {
            if (cs_debug)
                printf("CODE SIGNING: %s[%d]: invalid initial page at offset %lld
                    validated:%d tainted:%d csflags:0x%x\\n",
                    vp->v_name, p->p_pid, (long long)(file_offset + off), valid, tainted,
                    result->csflags);
            if (cs_enforcement(NULL) ||
                (result->csflags & (CS_HARD|CS_KILL|CS_ENFORCEMENT))) {
                ret = LOAD_FAILURE;
            }
            result->csflags &= ~CS_VALID;
        }
        off += PAGE_SIZE;
    }
}
}
.....

```

整个代码签名的验证过程大致分为load_code_signature()与cs_validate_page()两步，前者负责加载代码签名，后者负责验证数据页面。

load_code_signature()在加载代码签名时，通过调用ubc_cs_blob_get()来获取特定CPU的cs_blob指针。ubc_cs_blob_get()第一次调用时，返回的cs_blob指针为空，会调用ubc_cs_blob_add()来加载与验证文件中的Blob信息；再次调用ubc_cs_blob_get()时就会返回内存中的cs_blob指针。当然并不是直接返回，系统会再次判断内存中的cs_blob指针是否损坏或遭到篡改，具体方法是调用ubc_cs_generation_check()做初步检查，之后调用ubc_cs_blob_revalidate()对Blob做重验证。

下面是load_code_signature()函数代码：

```
static load_return_t
load_code_signature(
    struct linkedit_data_command *lcp,
    struct vnode *vp,
    off_t macho_offset,
    off_t macho_size,
    cpu_type_t cputype,
    load_result_t *result)
{
    int ret;
    kern_return_t kr;
    vm_offset_t addr;
    int resid;
    struct cs_blob *blob;
    int error;
    vm_size_t blob_size;

    addr = 0;
    blob = NULL;

    if (lcp->cmdsiz != sizeof (struct linkedit_data_command) ||
        lcp->dataoff + lcp->datasiz > macho_size) {
        ret = LOAD_BADMACHO;
        goto out;
    }

    blob = ubc_cs_blob_get(vp, cputype, macho_offset);
    if (blob != NULL) {
        if (blob->csb_cpu_type == cputype &&
```

```

        blob->csb_base_offset == macho_offset &&
        blob->csb_mem_size == lcp->datasize) {
    if(0 != ubc_cs_generation_check(vp)) {
        if (0 != ubc_cs_blob_revalidate(vp, blob, 0)) {
            ret = LOAD_FAILURE;
            goto out;
        }
    }
    ret = LOAD_SUCCESS;
} else {
    ret = LOAD_BADMACHO;
}
goto out;
}

blob_size = lcp->datasize;
kr = ubc_cs_blob_allocate(&addr, &blob_size);
if (kr != KERN_SUCCESS) {
    ret = LOAD_NOSPACE;
    goto out;
}

resid = 0;
error = vn_rdwr(UIO_READ,
    vp,
    (caddr_t) addr,
    lcp->datasize,
    macho_offset + lcp->dataoff,
    UIO_SYSSPACE,
    0,
    kauth_cred_get(),
    &resid,
    current_proc());
if (error || resid != 0) {
    ret = LOAD_IOERROR;
    goto out;
}

if (ubc_cs_blob_add(vp,
    cputype,
    macho_offset,
    addr,
    lcp->datasize,
    0)) {
    ret = LOAD_FAILURE;
    goto out;
} else {
    addr = 0;
}

#ifdef CHECK_CS_VALIDATION_BITMAP
    ubc_cs_validation_bitmap_allocate( vp );
#endif

blob = ubc_cs_blob_get(vp, cputype, macho_offset);

```

```
    ret = LOAD_SUCCESS;
out:
    if (ret == LOAD_SUCCESS) {
        result->csflags |= blob->csb_flags;
        result->platform_binary = blob->csb_platform_binary;
        result->cs_end_offset = blob->csb_end_offset;
    }
    if (addr != 0) {
        ubc_cs_blob_deallocate(addr, blob_size);
        addr = 0;
    }

    return ret;
}
```

注意，前面提到的cs_blob指针，其实就是代码签名数据中的CS_SuperBlob指针类型。

ubc_cs_blob_add()的代码比较长，它主要做了以下3项工作。

(1) 调用cs_validate_csblob()验证cs_blob指针的合法性，cs_validate_csblob()会对CSMAGIC_EMBEDDED_SIGNATURE 与 CSMAGIC_CODEDIRECTORY 做相应的验证处理，包括调用cs_validate_codedirectory()验证CS_CodeDirectory结构体的合法性，以及调用cs_validate_blob()来验证CS_SuperBlob中每一个CS_GenericBlob是否合法有效。

(2) 调用mac_vnode_check_signature()验证Blob块的代码签名，也就是比较Blob块的SHA1哈希值与计算的值是否相同。

(3) 加载所有的代码签名Hash信息，填充cs_blobs字段，为下一步的内存页签名验证做准备。

ubc_cs_blob_revalidate()做的验证检查几乎与ubc_cs_blob_add()相同，但因为已经有了一些缓存信息，因此检查时会快一些。load_code_signature()完成以后，会调用ubc_get_cs_blobs()获取cs_blobs指针，最后调用cs_validate_page()，逐页验证文件中每一页数据的签名。

以上检查做完后，LC_CODE_SIGNATURE就处理完了。如果没有错误发生，就表示代码签名验证通过了。

讲完了代码签名，下面我们再讲讲代码加密。Mach-O程序如果使用了代码加密技术，在加载命令列表中会有一个LC_ENCRYPTION_INFO加载命令，它存储了Mach-O的加密信息。对于此加载命令，有iOS程序逆向经验的读者应该不会感到陌生。iOS系统由于安全机制的原因，会对App Store中上架的应用默认开启数据加密。

被加密的App文件，部分段的数据内容是经过加密的，而记录加密数据的关键就是LC_ENCRYPTION_INFO加载命令。分析人员要想对加密过的App进行逆向分析，必须先经过一次解密（俗称“砸壳”）操作。

LC_ENCRYPTION_INFO使用encryption_info_command结构体表示(LC_ENCRYPTION_INFO_64使用encryption_info_command_64表示)，定义如下：

```
struct encryption_info_command {
    uint32_t  cmd;
    uint32_t  cmdsize;
    uint32_t  cryptoff;
    uint32_t  cryptsize;
    uint32_t  cryptid;
};
```

cryptoff与cryptsize字段分别指明了加密数据的文件偏移与大小，cryptid指定了使用的加密系统。

聪明的安全研究人员根据Mach-O在内存中被加载完即解密完成的特点，开发出了针对iOS平

台App的代码解密工具dumpdecrypted^①，将内存中解密的数据写回原位置，并将cryptid置为0，来达到解密App的目的。

下面再来看看系统如何处理LC_ENCRYPTION_INFO，它的解析函数也是parse_machfile()，代

码片段如下：

```
static load_return_t parse_machfile(
    struct vnode      *vp,
    vm_map_t          map,
    thread_t           thread,
    struct mach_header *header,
    off_t              file_offset,
    off_t              macho_size,
    int                depth,
    int64_t             aslr_offset,
    int64_t             dyld_aslr_offset,
    load_result_t      *result
)
{
    .....

    #if CONFIG_CODE_DECRYPTION
    case LC_ENCRYPTION_INFO:
    case LC_ENCRYPTION_INFO_64:
        if (pass != 3)
            break;
        ret = set_code_unprotect(
            (struct encryption_info_command *) lcp,
            addr, map, slide, vp,
            header->cputype, header->cpusubtype);
        if (ret != LOAD_SUCCESS) {
            printf("proc %d: set_code_unprotect() error %d "
                "for file \"%s\"\\n",
                p->p_pid, ret, vp->v_name);
            if (ret == LOAD_DECRYPTFAIL) {
                proc_lock(p);
                p->p_lflag |= P_LTERM_DECRYPTFAIL;
                proc_unlock(p);
            }
            psignal(p, SIGKILL);
        }
        break;
    #endif

    default:
        ret = LOAD_SUCCESS;
        break;
}
```

① dumpdecrypted下载地址：<https://github.com/stefanesser/dumpdecrypted>。

.....

当系统内核被配置为启用代码解密（即定义了CONFIG_CODE_DECRYPTION）之后，parse_machfile()函数会解析LC_ENCRYPTION_INFO与LC_ENCRYPTION_INFO_64加载命令。

最后调用了set_code_unprotect()函数来对代码进行解密。该函数通过encryption_info_command中的cryptid来确定使用的加密系统，然后对代码进行内存解密。代码片段如下：

```
#if CONFIG_CODE_DECRYPTION

static load_return_t
set_code_unprotect(
    struct encryption_info_command *eip,
    caddr_t addr,
    vm_map_t map,
    int64_t slide,
    struct vnode *vp,
    cpu_type_t cputype,
    cpu_subtype_t cpusubtype)
{
    .....
    if (eip->cmdsize < sizeof(*eip)) return LOAD_BADMACHO;

    switch(eip->cryptid) {
        case 0:
            return LOAD_SUCCESS;
        case 1:
            cryptname = "com.apple.unfree";
            break;
        case 0x10:
            cryptname = "com.apple.null";
            break;
        default:
            return LOAD_BADMACHO;
    }

    if (map == VM_MAP_NULL) return (LOAD_SUCCESS);
    if (NULL == text_crypter_create) return LOAD_FAILURE;

    .....

    crypt_file_data_t crypt_data = {
        .filename = vpath,
        .cputype = cputype,
        .cpusubtype = cpusubtype};
    kr=text_crypter_create(&crypt_info, cryptname, (void*)&crypt_data);
    FREE_ZONE(vpath, MAXPATHLEN, M_NAMEI);
}
```



```

.....

offset = mach_header_sz;
uint32_t ncmds = header->ncmds;
while (ncmds--) {
    struct load_command *lcp = (struct load_command *) (addr + offset);
    offset += lcp->cmdsize;

    switch(lcp->cmd) {
        case LC_SEGMENT_64:
            seg64 = (struct segment_command_64 *) lcp;
            if ((seg64->fileoff <= eip->cryptoff) &&
                (seg64->fileoff+seg64->filesize >=
                 eip->cryptoff+eip->cryptsize)) {
                map_offset = seg64->vmaddr + eip->cryptoff - seg64->fileoff + slide;
                map_size = eip->cryptsize;
                goto remap_now;
            }
        case LC_SEGMENT:
            seg32 = (struct segment_command *) lcp;
            if ((seg32->fileoff <= eip->cryptoff) &&
                (seg32->fileoff+seg32->filesize >=
                 eip->cryptoff+eip->cryptsize)) {
                map_offset = seg32->vmaddr + eip->cryptoff - seg32->fileoff + slide;
                map_size = eip->cryptsize;
                goto remap_now;
            }
    }
}

return LOAD_BADMACHO;

remap_now:
kr = vm_map_apple_protected(map, map_offset, map_offset+map_size, &crypt_info);
if(kr) {
    printf("set_code_unprotect(): mapping failed with %x\n", kr);
    crypt_info.crypt_end(crypt_info.crypt_ops);
    return LOAD_PROTECT;
}

return LOAD_SUCCESS;
}

#endif

```

text_crypter_create() 是一个 text_crypter_create_hook_t 类型全局指针，在内核代码

osfink/kern/page_decrypt.c文件中通过text_crypter_create_hook_set()进行设置。

在填充完解密所需的信息crypt_info后，text_crypter_create()会再次计算需要重新解密映射到

内存的地址与大小，调用`vm_map_apple_protected()`进行解密操作。

由于内核的代码可以直接审阅，数据加密在macOS系统上显得意义不大。在目前最新的macOS 10.12系统上，苹果没有启用代码解密功能，`LC_ENCRYPTION_INFO`与`LC_ENCRYPTION_INFO_64`加载命令也就没那么常见了。

最后，可以使用`otool`命令行工具查看Mach-O文件的加载命令信息：

```
otool -l /usr/bin/python
/usr/bin/python:
Load command 0
  cmd LC_SEGMENT_64
  cmdsize 72
  segname __PAGEZERO
  vmaddr 0x0000000000000000
  vmsize 0x0000000100000000
  fileoff 0
  filesize 0
  maxprot 0x00000000
  initprot 0x00000000
  nsects 0
  flags 0x0
Load command 1
  cmd LC_SEGMENT_64
.....
Load command 15
  cmd LC_DATA_IN_CODE
  cmdsize 16
  dataoff 17776
  datasize 0
Load command 16
  cmd LC_CODE_SIGNATURE
  cmdsize 16
  dataoff 20528
  datasize 9344
```

也可以使用MachOView查看，效果如图4-9所示。

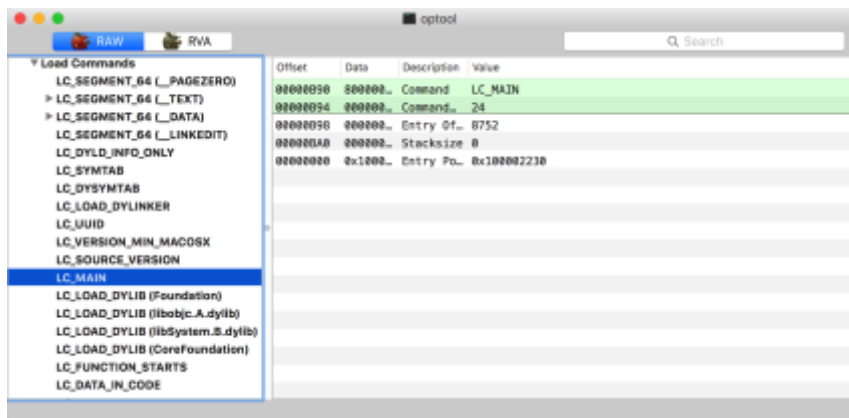


图4-9 使用MachOView查看加载命令信息

4.5.5 LC_SEGMENT

段加载命令LC_SEGMENT描述了32位Mach-O文件的段信息,使用segment_command结构体来

表示,它的定义如下:

```
struct segment_command {
    uint32_t  cmd;
    uint32_t  cmdsize;
    char      segname[16];
    uint32_t  vmaddr;
    uint32_t  vmsize;
    uint32_t  fileoff;
    uint32_t  filesize;
    vm_prot_t maxprot;
    vm_prot_t initprot;
    uint32_t  nsects;
    uint32_t  flags;
};
```

❑ segname字段是一个16字节大小的空间,用来存储段的名称。

❑ vmaddr字段指明了段要加载的虚拟内存地址。

❑ vmsize字段指明了段所占的虚拟内存的大小。

❑ fileoff字段指明了段数据所在文件中的偏移地址。

- ❑ filesize字段指明了段数据实际的大小。
- ❑ maxprot字段指明了页面所需要的最高内存保护。
- ❑ initprot字段指明了页面初始的内存保护。
- ❑ nsects字段指明了段所包含的节区。
- ❑ flags字段指明了段的标志信息。

与LC_SEGMENT对应的是LC_SEGMENT_64，它使用segment_command_64结构体表示，描述

了64位Mach-O文件的段的基本信息，定义如下：

```
struct segment_command_64 {
    uint32_t  cmd;
    uint32_t  cmdsize;
    char      segname[16];
    uint64_t  vmaddr;
    uint64_t  vmsize;
    uint64_t  fileoff;
    uint64_t  filesize;
    vm_prot_t  maxprot;
    vm_prot_t  initprot;
    uint32_t  nsects;
    uint32_t  flags;
};
```

所有的字段含义与32位的基本一致，下面我们重点讨论最后4个字段。

一个编译后可能执行的程序会分成多个段，不同类型的数据放入不同的段中。程序的代码被称作代码段，放入一个名为__TEXT的段中，代码段的maxprot字段在编译时被设置成VM_PROT_READ（可读）、VM_PROT_WRITE（可写）、VM_PROT_EXECUTE（可执行），initprot字段被设置成VM_PROT_READ与VM_PROT_EXECUTE，这样做是合理的。一个普通的应用程序，代码段部分通常是不可写的，有特殊需求的程序，如果要求代码段可写，必须在编译时设置它的

initprot字段为VM_PROT_WRITE。

nsects 字段指定了段加载命令包含几个节区，一个段可以包含0个或多个节区。如__PAGEZERO段就不包含任何节区，该段被称为空指针陷阱段，映射到虚拟内存空间的第一页，用于捕捉对NULL指针的引用。

当一个段包含多个节区时，节区信息会以数组的形式存储在段加载命令后面。节区使用结构体section表示（64位使用section_64表示），定义如下：

```
struct section {
    char    sectname[16];
    char    segname[16];
    uint32_t addr;
    uint32_t size;
    uint32_t offset;
    uint32_t align;
    uint32_t reloff;
    uint32_t nreloc;
    uint32_t flags;
    uint32_t reserved1;
    uint32_t reserved2;
};
```

sectname字段表示节区的名称，segname字段表示节区所在的段名，addr与size指明了节区所在的内存地址与大小，offset指明了节区所在的文件偏移，align表示节区的内存对齐边界，reloff指明了重定位信息的文件偏移，nreloc表示重定位条目的数目，flags则是节区的一些标志属性。

段加载命令的最后一个字段flags存储了段的一些标志属性，取值如下：

```
#define SG_HIGHVM    0x1
#define SG_FVMLIB    0x2
#define SG_NORELOC    0x4
#define SG_PROTECTED_VERSION_1  0x8
```

值得关注的是SG_PROTECTED_VERSION_1，当段被设置了该标志位时，表示段是经过加密

的。在macOS版本10.6以前，系统使用AES算法进行段的加密与解密，10.6版本则使用了Blowfish加密算法。著名的iOS逆向工具class-dump^①提供了一个静态数据段解密工具deprotect，有兴趣的读者可以参看它的代码来了解段解密部分。

最后，使用MachOView工具查看系统python程序__TEXT段的信息，如图4-10所示。

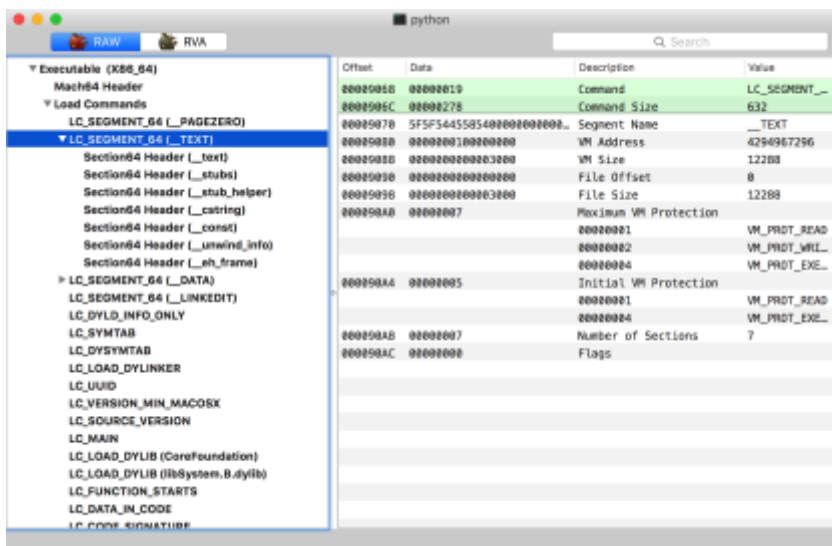


图4-10 MachOView查看__TEXT段的信息

4.6 动态库

Windows系统的动态库是DLL文件，Linux系统是so文件，macOS系统的动态库则使用dylib文件。dylib本质上是一个Mach-O格式的文件，它与普通的Mach-O执行文件使用几乎一样的结构，只是在文件类型上一个为MH_DYLIB，一个是MH_EXECUTE。

^① class-dump下载地址：<https://github.com/nygard/class-dump>。

在系统的/usr/lib目录下,存放了大量供系统和应用程序调用的动态库文件,使用file命令查看

系统动态库libobjc.dylib的信息,输出如下:

```
$ file /usr/lib/libobjc.dylib
/usr/lib/libobjc.dylib: Mach-O universal binary with 3 architectures
/usr/lib/libobjc.dylib (for architecture i386):  Mach-O dynamically linked shared library i386
/usr/lib/libobjc.dylib (for architecture x86_64):  Mach-O 64-bit dynamically linked shared library x86_64
/usr/lib/libobjc.dylib (for architecture x86_64h):  Mach-O 64-bit dynamically linked shared library x86_64
```

从输出信息可以看出,libobjc.dylib是一个通用的二进制文件,包含了3种CPU架构的Mach-O。

另外,可以使用Mach-O格式文件管理工具otool查看dylib的信息,如查看动态库的依赖库信息,

如下所示:

```
$ otool -L /usr/lib/libobjc.dylib
/usr/lib/libobjc.dylib:
    /usr/lib/libobjc.A.dylib (compatibility version 1.0.0, current version 228.0.0)
    /usr/lib/libauto.dylib (compatibility version 1.0.0, current version 1.0.0)
    /usr/lib/libc++abi.dylib (compatibility version 1.0.0, current version 125.0.0)
    /usr/lib/libc++.1.dylib (compatibility version 1.0.0, current version 120.1.0)
    /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 1225.0.0)
```

4.6.1 构建动态库

Xcode环境提供了创建动态库的工程模板,创建动态库的方法比较简单。在Xcode中选择

File→New→Project,在打开的工程模板选择对话框中,选择标签macOS→Framework & Library。

在右侧选择Library,点击Next按钮。在新页面中输入项目名称mylib,Type选择Dynamic,点击Next

按钮选择项目保存的路径,工程就创建好了。接下来修改工程文件内容:

```
#import <Foundation/Foundation.h>

@interface mylib : NSObject
-(void) hello;
@end

#import "mylib.h"

@implementation mylib
```

```

-(void) hello {
    NSLog(@"hello world");
}
@end

```

保存后点击菜单Product→Build，或者按快捷键command+B键就编译成功了。命令执行完后，就会生成mylib.dylib文件。

Xcode创建的项目是xcodeproj文件，可以使用Xcode提供的工具xcodesbuild在命令行下编译。

在命令行下切换到工程文件所在的目录，执行xcodesbuild会有如下输出：

4

```

$ xcodesbuild
=== BUILD TARGET mylib OF PROJECT mylib WITH THE DEFAULT CONFIGURATION (Release) ===

Check dependencies

Write auxiliary files
write-file
/Users/macbook/Documents/macbook/macbook/code/chapter4/mylib/build/mylib.build/Release/mylib.build/m
ylib-own-target-headers.hmap
write-file
/Users/macbook/Documents/macbook/macbook/code/chapter4/mylib/build/mylib.build/Release/mylib.build/m
ylib-all-non-framework-target-headers.hmap
.....
/bin/mkdir -p
/Users/macbook/Documents/macbook/macbook/code/chapter4/mylib/build/mylib.build/Release/mylib.build/O
bjects-normal/x86_64
write-file
/Users/macbook/Documents/macbook/macbook/code/chapter4/mylib/build/mylib.build/Release/mylib.build/O
bjects-normal/x86_64/mylib.LinkFileList

CompileC build/mylib.build/Release/mylib.build/Objects-normal/x86_64/mylib.o mylib/mylib.m normal x86_64
objective-c com.apple.compilers.llvm.clang.1_0.compiler
  cd /Users/macbook/Documents/macbook/macbook/code/chapter4/mylib
  export LANG=en_US.UTF-8
  /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/clang -x objective-c
-arch x86_64 -fmessage-length=94 -fdiagnostics-show-note-include-stack -fmacro-backtrace-limit=0
-fcolor-diagnostics -std=gnu99 -fobjc-arc -fmodules -gmodules -fmodules-prune-interval=86400
-fmodules-prune-after=345600
-fbuild-session-file=/var/folders/rd/mts0362j0n92rq0z1cnmdb580000gn/C/org.llvm.clang/ModuleCache/Sessio
n.modulevalidation -fmodules-validate-once-per-build-session -Wnon-modular-include-in-framework-module
-Werror=non-modular-include-in-framework-module -Wno-trigraphs -fpascal-strings -O5 -fno-common
-Wno-missing-field-initializers -Wno-missing-prototypes -Werror=return-type -Wunreachable-code
-Wno-implicit-atomic-properties -Werror=deprecated-objc-isa-usage -Werror=objc-root-class
-Wno-arc-repeated-use-of-weak -Wduplicate-method-match -Wno-missing-braces -Wparentheses -Wswitch
-Wunused-function -Wno-unused-label -Wno-unused-parameter -Wunused-variable -Wunused-value -Wempty-body
-Wconditional-uninitialized -Wno-unknown-pragmas -Wno-shadow -Wno-four-char-constants -Wno-conversion

```



```

-Wconstant-conversion -Wint-conversion -Wbool-conversion -Wenum-conversion -Wshorten-64-to-32
-Wpointer-sign -Wno-newline-eof -Wno-selector -Wno-strict-selector-match -Wundeclared-selector
-Wno-deprecated-implementations -DNS_BLOCK_ASSERTIONS=1 -DOBJC_OLD_DISPATCH_PROTOTYPES=0
-isysroot
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX10.11.sdk
-fasm-blocks -fstrict-aliasing -Wprotocol -Wdeprecated-declarations -mmacosx-version-min=10.11 -g
-Wno-sign-conversion -iquote
/Users/macbook/Documents/macbook/macbook/code/chapter4/mylib/build/mylib.build/Release/mylib.build/m
ylib-generated-files.hmap
-I/Users/macbook/Documents/macbook/macbook/code/chapter4/mylib/build/mylib.build/Release/mylib.build/m
ylib-own-target-headers.hmap
-I/Users/macbook/Documents/macbook/macbook/code/chapter4/mylib/build/mylib.build/Release/mylib.build/m
ylib-all-target-headers.hmap -iquote
/Users/macbook/Documents/macbook/macbook/code/chapter4/mylib/build/mylib.build/Release/mylib.build/m
ylib-project-headers.hmap
-I/Users/macbook/Documents/macbook/macbook/code/chapter4/mylib/build/Release/include
-I/Users/macbook/Documents/macbook/macbook/code/chapter4/mylib/build/mylib.build/Release/mylib.build/D
erivedSources/x86_64
-I/Users/macbook/Documents/macbook/macbook/code/chapter4/mylib/build/mylib.build/Release/mylib.build/D
erivedSources -F/Users/macbook/Documents/macbook/macbook/code/chapter4/mylib/build/Release -MMD
-MT dependencies -MF
/Users/macbook/Documents/macbook/macbook/code/chapter4/mylib/build/mylib.build/Release/mylib.build/O
bjects-normal/x86_64/mylib.d --serialize-diagnostics
/Users/macbook/Documents/macbook/macbook/code/chapter4/mylib/build/mylib.build/Release/mylib.build/O
bjects-normal/x86_64/mylib.dia -c
/Users/macbook/Documents/macbook/macbook/code/chapter4/mylib/mylib/mylib.m -o
/Users/macbook/Documents/macbook/macbook/code/chapter4/mylib/build/mylib.build/Release/mylib.build/O
bjects-normal/x86_64/mylib.o

```

```

Ld build/Release/libmylib.dylib normal x86_64
  cd /Users/macbook/Documents/macbook/macbook/code/chapter4/mylib
  export MACOSX_DEPLOYMENT_TARGET=10.11

```

```

/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/clang -arch x86_64
-dynamiclib -isysroot
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX10.11.sdk
-L/Users/macbook/Documents/macbook/macbook/code/chapter4/mylib/build/Release
-F/Users/macbook/Documents/macbook/macbook/code/chapter4/mylib/build/Release -filelist
/Users/macbook/Documents/macbook/macbook/code/chapter4/mylib/build/mylib.build/Release/mylib.build/O
bjects-normal/x86_64/mylib.LinkFileList -install_name /usr/local/lib/libmylib.dylib -mmacosx-version-min=10.11
-fobjc-arc -fobjc-link-runtime -single_module -compatibility_version 1 -current_version 1 -Xlinker -dependency_info
-Xlinker
/Users/macbook/Documents/macbook/macbook/code/chapter4/mylib/build/mylib.build/Release/mylib.build/O
bjects-normal/x86_64/mylib_dependency_info.dat -o
/Users/macbook/Documents/macbook/macbook/code/chapter4/mylib/build/Release/libmylib.dylib

```

```

GenerateDSYMFile build/Release/libmylib.dylib.dSYM build/Release/libmylib.dylib
  cd /Users/macbook/Documents/macbook/macbook/code/chapter4/mylib
  /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/dsymutil
/Users/macbook/Documents/macbook/macbook/code/chapter4/mylib/build/Release/libmylib.dylib -o
/Users/macbook/Documents/macbook/macbook/code/chapter4/mylib/build/Release/libmylib.dylib.dSYM

```

```

CodeSign build/Release/libmylib.dylib
  cd /Users/macbook/Documents/macbook/macbook/code/chapter4/mylib

```

```

export
CODESIGN_ALLOCATE = /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/codesign_allocate

Signing Identity:  "-"

/usr/bin/codesign --force --sign - --timestamp=none
/Users/macbook/Documents/macbook/macbook/code/chapter4/mylib/build/Release/libmylib.dylib

** BUILD SUCCEEDED **

```

从上面的日志中可以看出，整个编译过程分为：检查依赖（Check dependencies）、生成辅助文件（Write auxiliary files）、编译（CompileC）、链接（Ld）、生成调试符号（GenerateDSYMFile）、代码签名（CodeSign）6步。

编译代码时使用的编译器是Clang，这是苹果公司开发的用来替代GCC的现代化编译器。目前，该编译器也广泛用于Android、Linux平台。链接时使用Clang前端传入参数给链接器ld，链接完成后dylib动态库就编译成功了。生成调试符号这一步主要用于生成符号的调试信息，供调试器使用。最后一步是代码签名，在没有指定签名证书的情况下，Xcode默认使用adhoc签名。

对于编译好的动态库，其他程序可以通过头文件声明隐式调用，也可以像Linux系统那样，使用系统函数dlopen()、dlsym()手动调用。

4.6.2 dyld

动态库不能直接运行，需要通过系统的动态链接加载器加载到内存后执行。动态链接加载器在系统中以可执行文件形式存在，一般应用程序会在Mach-O文件部分指定一个LC_LOAD_DYLINKER的加载命令。此加载命令指定了dyld的路径，通常默认值是/usr/lib/dyld。系统内核在加载Mach-O文件时，会使用该路径指定的程序作为动态库的加载器来加载dylib。

为了优化程序启动，dyld在加载时启用了共享缓存（shared cache）技术。在进程启动时，共享缓存会被dyld映射到内存中，之后，当Mach-O映像加载时，dyld首先会检查该Mach-O映像与所需的动态库是否在共享缓存中，如果在，则直接将它在共享内存中的内存地址映射到进程的内存地址空间。在程序依赖的系统动态库很多的情况下，这样做会明显提升程序启动性能。

update_dyld_shared_cache程序确保了dyld的共享缓存是最新的，它会扫描/var/db/dyld/shared_region_roots/目录下的paths路径文件，这些paths文件包含了需要加入到共享缓存的Mach-O文件路径列表。update_dyld_shared_cache()会将这些Mach-O文件及其依赖的dylib逐个加到共享缓存中去。

共享缓存是以文件形式存放在/var/db/dyld/目录下的，生成共享缓存的update_dyld_shared_cache程序位于/usr/bin/目录下，该工具会为每种系统架构生成一个缓存文件以及对应的内存地址map表，如下所示：

```
ls -l /var/db/dyld/
total 1741296
-rw-r--r--  1 root  wheel  333085108 Apr 22 15:02 dyld_shared_cache_i386
-rw-r--r--  1 root  wheel    65378 Apr 22 15:02 dyld_shared_cache_i386.map
-rw-r--r--  1 root  wheel  558259294 Apr 25 16:18 dyld_shared_cache_x86_64h
-rw-r--r--  1 root  wheel   129633 Apr 25 16:18 dyld_shared_cache_x86_64h.map
drwxr-xr-x 10 root  wheel     340 Apr  7 09:19 shared_region_roots
```

可以使用dyld_shared_cache_util工具来查看生成的共享缓存，该工具位于dyld源代码中的launch-cache/dyld_shared_cache_util.cpp文件中，需要手动编译。另外，也可以使用dyld提供的两个函数dyld_shared_cache_extract_dylibs()与dyld_shared_cache_extract_dylibs_progress()解开cache文件，代码位于dyld源代码的launch-cache/dsc_extractor.cpp文件中。

`update_dyld_shared_cache`通常只在系统的安装器安装软件与系统更新时调用。当然，可以手动运行`sudo update_dyld_shared_cache`来更新共享缓存，新的共享缓存会在系统下次启动后自动更新。

4.6.3 动态库的加载

dyld是苹果操作系统的重要组成部分。令人兴奋的是，它是开源的，任何人都可以到苹果官网下载它的源代码^①，阅读理解它的运作方式，了解系统加载动态库的细节。

4

系统内核在加载动态库前，会加载dyld，然后调用执行`__dyld_start()`函数。该函数会执行`dyldbootstrap::start()`，后者又会执行`_main()`函数，dyld加载动态库的代码就是从`_main()`开始执行的。下面以dyld源代码的360.18版本为蓝本进行分析：

```
uintptr_t
_main(const macho_header* mainExecutableMH, uintptr_t mainExecutableSlide,
      int argc, const char* argv[], const char* envp[], const char* apple[],
      uintptr_t* startGlue)
{
    // 第一步，设置运行环境，处理环境变量
    uintptr_t result = 0;
    sMainExecutableMachHeader = mainExecutableMH;

    .....

    CRSetCrashLogMessage("dyld: launch started");

    .....

    setContext(mainExecutableMH, argc, argv, envp, apple);

    sExecPath = _simple_getenv(apple, "executable_path");

    if (!sExecPath) sExecPath = apple[0];

    .....
}
```

① dyld下载地址：<http://opensource.apple.com/tarballs/dyld>。

```
sExecShortName = ::strchr(sExecPath, '/');  
if ( sExecShortName != NULL )  
    ++sExecShortName;  
else  
    sExecShortName = sExecPath;  
sProcessIsRestricted = processRestricted(mainExecutableMH, &ignoreEnvironmentVariables,  
&sProcessRequiresLibraryValidation);
```

```

    if ( sProcessIsRestricted ) {
#ifdef SUPPORT_LC_DYLD_ENVIRONMENT
        checkLoadCommandEnvironmentVariables();
#endif
        pruneEnvironmentVariables(envp, &apple);
        setContext(mainExecutableMH, argc, argv, envp, apple);
    }
    else {
        if ( !ignoreEnvironmentVariables )
            checkEnvironmentVariables(envp);
        defaultUninitializedFallbackPaths(envp);
    }
    if ( sEnv.DYLD_PRINT_OPTS )
        printOptions(argv);
    if ( sEnv.DYLD_PRINT_ENV )
        printEnvironmentVariables(envp);
    getHostInfo(mainExecutableMH, mainExecutableSlide);

    .....

//第二步，初始化主程序
try {
    addDyldImageToUUIDList();
    CRSetCrashLogMessage(sLoadingCrashMessage);
    sMainExecutable = instantiateFromLoadedImage(mainExecutableMH, mainExecutableSlide,
        sExecPath);
    gLinkContext.mainExecutable = sMainExecutable;
    gLinkContext.processIsRestricted = sProcessIsRestricted;
    gLinkContext.processRequiresLibraryValidation = sProcessRequiresLibraryValidation;
    gLinkContext.mainExecutableCodeSigned = hasCodeSignatureLoadCommand(mainExecutableMH);

    .....

//第三步，加载共享缓存
    checkSharedRegionDisable();
#ifdef DYLD_SHARED_CACHE_SUPPORT
    if ( gLinkContext.sharedRegionMode != ImageLoader::kDontUseSharedRegion )
        mapSharedCache();
#endif

#ifdef SUPPORT_VERSIONED_PATHS
    checkVersionedPaths();
#endif

//第四步，加载插入的动态库
    if ( sEnv.DYLD_INSERT_LIBRARIES != NULL ) {
        for (const char* const* lib = sEnv.DYLD_INSERT_LIBRARIES; *lib != NULL; ++lib)
            loadInsertedDylib(*lib);
    }
    sInsertedDylibCount = sAllImages.size()-1;

//第五步，链接主程序
    gLinkContext.linkingMainExecutable = true;

```

```

link(sMainExecutable, sEnv.DYLD_BIND_AT_LAUNCH, true, ImageLoader::RPathChain(NULL, NULL));
sMainExecutable->setNeverUnloadRecursive();
if ( sMainExecutable->forceFlat() ) {
    gLinkContext.bindFlat = true;
    gLinkContext.prebindUsage = ImageLoader::kUseNoPrebinding;
}

//第六步，链接插入的动态库
if ( sInsertedDylibCount > 0 ) {
    for(unsigned int i=0; i < sInsertedDylibCount; ++i) {
        ImageLoader* image = sAllImages[i+1];
        link(image, sEnv.DYLD_BIND_AT_LAUNCH, true, ImageLoader::RPathChain(NULL, NULL));
        image->setNeverUnloadRecursive();
    }
    for(unsigned int i=0; i < sInsertedDylibCount; ++i) {
        ImageLoader* image = sAllImages[i+1];
        image->registerInterposing();
    }
}

for (int i=sInsertedDylibCount+1; i < sAllImages.size(); ++i) {
    ImageLoader* image = sAllImages[i];
    if ( image->inSharedCache() )
        continue;
    image->registerInterposing();
}

for(int i=0; i < sImageRoots.size(); ++i) {
    sImageRoots[i]->applyInterposing(gLinkContext);
}

//第七步，执行弱符号绑定
gLinkContext.linkingMainExecutable = false;
sMainExecutable->weakBind(gLinkContext);

//第八步，执行初始化方法
CRSetCrashLogMessage("dyld: launch, running initializers");
#ifdef SUPPORT_OLD_CRT_INITIALIZATION
    if ( ! gRunInitializersOldWay )
        initializeMainExecutable();
#else
    initializeMainExecutable();
#endif

//第九步，查找入口点并返回
result = (uintptr_t)sMainExecutable->getThreadPC();
if ( result != 0 ) {
    if ( (gLibSystemHelpers != NULL) && (gLibSystemHelpers->version >= 9) )
        *startGlue = (uintptr_t)gLibSystemHelpers->startGlueToCallExit;
    else
        halt("libdyld.dylib support not present for LC_MAIN");
}
else {
    result = (uintptr_t)sMainExecutable->getMain();
}

```

```

        *startGlue = 0;
    }
}
catch(const char* message) {
    syncAllImages();
    halt(message);
}
catch(...) {
    dyld::log("dyld: launch failed\n");
}

CRSetCrashLogMessage(NULL);

return result;
}

```

整个方法的代码比较长，下面我们按功能分成9个步骤进行讲解。

1. 第一步，设置运行环境，处理环境变量

代码在开始时将传入的变量mainExecutableMH赋值给了sMainExecutableMachHeader。这是一个macho_header类型的变量，其结构体内容就是本章前面介绍的mach_header结构体，表示的是当前主程序的Mach-O头部信息。有了头部信息，加载器就可以从头开始遍历整个Mach-O文件的信息。

接着执行了setContext()，此方法设置了一个链接上下文，包括一些回调函数、参数与标志设置信息，代码片段如下所示：

```

static void setContext(const macho_header* mainExecutableMH, int argc, const char* argv[], const char*
    envp[], const char* apple[])
{
    gLinkContext.loadLibrary          = &libraryLocator;
    gLinkContext.terminationRecorder  = &terminationRecorder;
    gLinkContext.flatExportFinder     = &flatFindExportedSymbol;
    gLinkContext.coalescedExportFinder = &findCoalescedExportedSymbol;
    gLinkContext.getCoalescedImages   = &getCoalescedImages;

    .....

    gLinkContext.bindingOptions       = ImageLoader::kBindingNone;
    gLinkContext.argc                  = argc;
    gLinkContext.argv                  = argv;
}

```



```

gLinkContext.envp          = envp;
gLinkContext.apple        = apple;
gLinkContext.progname      = (argv[0] != NULL) ? basename(argv[0]) : "";
gLinkContext.programVars.mh = mainExecutableMH;
gLinkContext.programVars.NXArgcPtr = &gLinkContext.argc;
gLinkContext.programVars.NXArgvPtr = &gLinkContext.argv;
gLinkContext.programVars.envIRONPtr = &gLinkContext.envp;
gLinkContext.programVars.__prognamePtr = &gLinkContext.progname;
gLinkContext.mainExecutable = NULL;
gLinkContext.imageSuffix    = NULL;
gLinkContext.dynamicInterposeArray = NULL;
gLinkContext.dynamicInterposeCount = 0;
gLinkContext.prebindUsage    = ImageLoader::kUseAllPrebinding;
#if TARGET_IPHONE_SIMULATOR
gLinkContext.sharedRegionMode = ImageLoader::kDontUseSharedRegion;
#else
gLinkContext.sharedRegionMode = ImageLoader::kUseSharedRegion;
#endif
}

```

设置的回调函数都是dyld本模块实现的，如loadLibrary方法就是本模块的libraryLocator()方法，负责加载动态库。

设置完这些信息后，执行processRestricted()方法判断进程是否受限。代码如下：

```

static bool processRestricted(const macho_header* mainExecutableMH, bool* ignoreEnvVars, bool*
processRequiresLibraryValidation)
{
#if TARGET_IPHONE_SIMULATOR
gLinkContext.codeSigningEnforced = true;
#else
uint32_t flags;
if ( csops(0, CS_OPS_STATUS, &flags, sizeof(flags)) != -1 ) {
if (flags & CS_REQUIRE_LV)
*processRequiresLibraryValidation = true;

#if __MAC_OS_X_VERSION_MIN_REQUIRED
if ( flags & CS_ENFORCEMENT ) {
gLinkContext.codeSigningEnforced = true;
}
if ( ((flags & CS_RESTRICT) == CS_RESTRICT) && (csr_check(CSR_ALLOW_TASK_FOR_PID) != 0) ) {
sRestrictedReason = restrictedByEntitlements;
return true;
}
}
#else
if ((flags & CS_ENFORCEMENT) && !(flags & CS_GET_TASK_ALLOW)) {
*ignoreEnvVars = true;
}
gLinkContext.codeSigningEnforced = true;

```

```

#endif
}
#endif

if ( issetugid() ) {
    sRestrictedReason = restrictedBySetGUid;
    return true;
}

if ( hasRestrictedSegment(mainExecutableMH) ) {
    sRestrictedReason = restrictedBySegment;
    return true;
}
return false;
}

```

4

如果进程受限，会有以下3种可能。

- ❑ `restrictedByEntitlements`：在macOS系统上，在需要验证代码签名（Gatekeeper开启），且 `csr_check(CSR_ALLOW_TASK_FOR_PID)` 返回为true（表示Rootless开启了TASK_FOR_PID标志）时，进程才不会受限。在macOS 10.12系统上，Gatekeeper默认是开启的，并且Rootless是关闭了CSR_ALLOW_TASK_FOR_PID标志位的，这意味着在默认情况下，系统上运行的进程是受限的。
- ❑ `restrictedBySetGUid`：当进程的setuid与setgid位被设置时，进程会被设置成受限。这样做是出于安全考虑，受限后的进程无法访问DYLD_开头的环境变量。一种典型的系统攻击就是针对这种情况发生的：在macOS 10.10系统上，一个系统本地提权漏洞就是通过向DYLD_PRINT_TO_FILE环境变量传入拥有SUID权限的受限文件造成的，因为系统没做安全检测，这些文件有直接向系统创建与写入文件的权限。关于漏洞的具体细节可以参看：
https://www.sektioneins.de/en/blog/15-07-07-dyld_print_to_file_lpe.html。

- ❑ `restrictedBySegment`：段名受限。当Mach-O包含一个`__RESTRICT/__restrict`段时，进程会被设置成受限。

在进程受限后，会执行以下3个方法。

- ❑ `checkLoadCommandEnvironmentVariables()`：遍历Mach-O中所有的`LC_DYLD_ENVIRONMENT`加载命令，然后调用`processDyldEnvironmentVariable()`对不同的环境变量做相应的处理。
- ❑ `pruneEnvironmentVariables()`：删除进程的`LD_LIBRARY_PATH`与所有以“DYLD_”开头的环境变量，这样以后创建的子进程就不包含这些环境变量了。
- ❑ `setContext()`：重新设置链接上下文。这一步操作主要是由于环境变量发生了变化了，需要更新进程的`envp`与`apple`参数。

2. 第二步，初始化主程序

这一步主要执行了`instantiateFromLoadedImage()`，它的代码如下：

```
static ImageLoader* instantiateFromLoadedImage(const macho_header* mh, uintptr_t slide, const char* path)
{
    if ( isCompatibleMachO((const uint8_t*)mh, path) ) {
        ImageLoader* image = ImageLoaderMachO::instantiateMainExecutable(mh, slide, path,
            gLinkContext);
        addImage(image);
        return image;
    }

    throw "main executable not a known format";
}
```

`isCompatibleMachO()`主要检查Mach-O的头部的`cputype`与`cpusubtype`，从而判断程序与当前的

系统是否兼容。如果兼容就调用`InstantiateMainExecutable()`实例化主程序，代码如下：

```
ImageLoader* ImageLoaderMachO::InstantiateMainExecutable(const macho_header* mh, uintptr_t slide, const char*
path, const LinkContext& context)
{
    bool compressed;
    unsigned int segCount;
    unsigned int libCount;
    const linkedit_data_command* codeSigCmd;
    const encryption_info_command* encryptCmd;
    sniffLoadCommands(mh, path, false, &compressed, &segCount, &libCount, context, &codeSigCmd,
&encryptCmd);
    if ( compressed )
        return ImageLoaderMachOCompressed::InstantiateMainExecutable(mh, slide, path, segCount,
libCount, context);
    else
        #if SUPPORT_CLASSIC_MACHO
            return ImageLoaderMachOClassic::InstantiateMainExecutable(mh, slide, path, segCount, libCount,
context);
        #else
            throw "missing LC_DYLD_INFO load command";
        #endif
}
```

`sniffLoadCommands()`主要获取了加载命令中的如下信息。

❑ `compressed`：判断Mach-O是Compressed还是Classic类型。判断的依据是Mach-O是否包含

`LC_DYLD_INFO`或`LC_DYLD_INFO_ONLY`加载命令。这两个加载命令记录了Mach-O的动态

库加载信息，使用结构体`dyld_info_command`表示：

```
struct dyld_info_command {
    uint32_t cmd;
    uint32_t cmdsize;
    uint32_t rebase_off;
    uint32_t rebase_size;
    uint32_t bind_off;
    uint32_t bind_size;
    uint32_t weak_bind_off;
    uint32_t weak_bind_size;
    uint32_t lazy_bind_off;
    uint32_t lazy_bind_size;
    uint32_t export_off;
    uint32_t export_size;
};
```

- ❑ `rebase_off`与`rebase_size`存储了`rebase`（重设基址）相关信息。当Mach-O加载到内存中的地址不是指定的首选地址时，就需要对当前的映像数据重设基址。
- ❑ `bind_off`与`bind_size`存储了进程的符号绑定信息。当进程启动时，必须绑定这些符号。典型的有`dyld_stub_binder`，该符号被dyld用来做延迟绑定加载符号，一般动态库都包含该符号。
- ❑ `weak_bind_off`与`weak_bind_size`存储了进程的弱绑定符号信息。弱符号主要用于面向对象语言中的符号重载，典型的有C++中使用`new`创建对象，默认情况下会绑定`ibstdc++.dylib`，如果检测到某个映像使用弱符号引用重载了`new`符号，dyld会重新绑定该符号并调用重载的版本。
- ❑ `lazy_bind_off`与`lazy_bind_size`存储了进程的延迟绑定符号信息。有些符号在进程启动时不需要马上解析，它们会在第一次调用时被解析，这类符号叫延迟绑定符号（Lazy Symbol）。
- ❑ `export_off`与`export_size`存储了进程的导出符号绑定信息。导出符号可以被外部的Mach-O访问，通常动态库会导出一个或多个符号供外部使用，而可执行程序又导出`_main`与`_mh_execute_header`符号供dyld使用。
- ❑ `segCount`：段的数量。`sniffLoadCommands()`通过遍历所有的`LC_SEGMENT_COMMAND`加载命令来获取段的数量。
- ❑ `libCount`：需要加载的动态库的数量。Mach-O中包含的每一条`LC_LOAD_DYLIB`、`LC_LOAD_WEAK_DYLIB`、`LC_REEXPORT_DYLIB`、`LC_LOAD_UPWARD_DYLIB`加载命令，都表示需要加载一个动态库。

❑ codeSigCmd : 通过解析LC_CODE_SIGNATURE来获取代码签名的加载命令。

❑ encryptCmd : 通过LC_ENCRYPTION_INFO与LC_ENCRYPTION_INFO_64来获取段加密信息。

获取compressed后, 根据Mach-O是否compressed, 分别调用ImageLoaderMachOCompressed::

instantiateMainExecutable() 与 ImageLoaderMachOClassic::instantiateMainExecutable() 。

ImageLoaderMachOCompressed::instantiateMainExecutable()代码如下所示 :

```
ImageLoaderMachOCompressed* ImageLoaderMachOCompressed::instantiateMainExecutable(const
    macho_header* mh, uintptr_t slide, const char* path,
    unsigned int segCount, unsigned int libCount, const LinkContext& context)
{
    ImageLoaderMachOCompressed* image = ImageLoaderMachOCompressed::instantiateStart(mh, path,
        segCount, libCount);

    image->setSlide(slide);

    if ( slide != 0 )
        fgNextPIEDylibAddress = (uintptr_t)image->getEnd();

    image->disableCoverageCheck();
    image->instantiateFinish(context);
    image->setMapped(context);

    if ( context.verboseMapping ) {
        dyld::log("dyld: Main executable mapped %s\n", path);
        for(unsigned int i=0, e=image->segmentCount(); i < e; ++i) {
            const char* name = image->segName(i);
            if ( (strcmp(name, "__PAGEZERO") == 0) || (strcmp(name, "__UNIXSTACK") == 0) )
                dyld::log("%18s at 0x%08IX->0x%08IX\n", name, image->segPreferredLoadAddress(i),
                    image->segPreferredLoadAddress(i)+image->segSize(i));
            else
                dyld::log("%18s at 0x%08IX->0x%08IX\n", name, image->segActualLoadAddress(i),
                    image->segActualEndAddress(i));
        }
    }

    return image;
}
```

ImageLoaderMachOCompressed::instantiateStart() 使用主程序 Mach-O 信息构造了一个

ImageLoaderMachOCompressed对象。disableCoverageCheck()禁用覆盖率检查。instantiateFinish()调

用parseLoadCmds()解析其他所有的加载命令，后者会填充完ImageLoaderMachOCompressed的一些保护成员信息，最后调用setDyldInfo()设置动态库链接信息，然后调用setSymbolTableInfo()设置符号表信息。

instantiateFromLoadedImage()在调用完ImageLoaderMachO::instantiateMainExecutable()后，就会调用addImage()，代码如下：

```
static void addImage(ImageLoader* image)
{
    allImagesLock();
    sAllImages.push_back(image);
    allImagesUnlock();

    uintptr_t lastSegStart = 0;
    uintptr_t lastSegEnd = 0;
    for(unsigned int i=0, e=image->segmentCount(); i < e; ++i) {
        if ( image->segUnaccessible(i) )
            continue;
        uintptr_t start = image->segActualLoadAddress(i);
        uintptr_t end = image->segActualEndAddress(i);
        if ( start == lastSegEnd ) {
            lastSegEnd = end;
        }
        else {
            if ( lastSegEnd != 0 )
                addMappedRange(image, lastSegStart, lastSegEnd);
            lastSegStart = start;
            lastSegEnd = end;
        }
    }
    if ( lastSegEnd != 0 )
        addMappedRange(image, lastSegStart, lastSegEnd);

    if ( sEnv.DYLD_PRINT_LIBRARIES || (sEnv.DYLD_PRINT_LIBRARIES_POST_LAUNCH &&
        (sMainExecutable!=NULL) && sMainExecutable->isLinked()) ) {
        dyld::log("dyld: loaded: %s\n", image->getPath());
    }
}
```

这段代码将实例化的主程序添加到了全局主列表sAllImages中，最后调用addMappedRange()

申请内存，更新主程序映像映射的内存区。做完这些工作，第二步初始化主程序就完成了。

3. 第三步，加载共享缓存

这一步主要执行mapSharedCache()来映射共享缓存。该函数先通过_shared_region_check_np()来检查缓存是否已经映射到了共享区域，如果已经映射，则更新缓存的slide与UUID，然后返回。反之，判断系统是否处于安全启动模式（safe-boot mode），如果是，则删除缓存文件并返回。在正常启动的情况下，调用openSharedCacheFile()打开缓存文件。该函数在sSharedCacheDir路径下，打开与系统当前CPU架构匹配的缓存文件，也就是/var/db/dyld/dyld_shared_cache_x86_64h，接着读取缓存文件的前8192字节，解析缓存头dyld_cache_header的信息，将解析好的缓存信息存入mappings变量，最后调用_shared_region_map_and_slide_np()完成真正的映射工作。部分代码片段如下：

```
static void mapSharedCache()
{
    uint64_t cacheBaseAddress = 0;
    if ( _shared_region_check_np(&cacheBaseAddress) == 0 ) {
        sSharedCache = (dyld_cache_header*)cacheBaseAddress;
#ifdef __x86_64__
        const char* magic = (sHaswell ? ARCH_CACHE_MAGIC_H : ARCH_CACHE_MAGIC);
#else
        const char* magic = ARCH_CACHE_MAGIC;
#endif
        if ( strcmp(sSharedCache->magic, magic) != 0 ) {
            sSharedCache = NULL;
            if ( gLinkContext.verboseMapping ) {
                dyld::log("dyld: existing shared cached in memory is not compatible\n");
                return;
            }
        }
        const dyld_cache_header* header = sSharedCache;
        .....
        if ( header->mappingOffset >= 0x68 ) {
            memcpy(dyld::gProcessInfo->sharedCacheUUID, header->uuid, 16);
        }
        .....
    }
    else {
#ifdef __i386__ || __x86_64__
        uint32_t safeBootValue = 0;
```



```

size_t    safeBootValueSize = sizeof(safeBootValue);
if ( ( sysctlbyname("kern.safeboot", &safeBootValue, &safeBootValueSize, NULL, 0) == 0 ) &&
    (safeBootValue != 0) ) {
    struct stat dyldCacheStatInfo;
    if ( my_stat(MACOSX_DYLD_SHARED_CACHE_DIR DYLD_SHARED_CACHE_BASE_NAME ARCH_NAME,
        &dyldCacheStatInfo) == 0 ) {
        struct timeval bootTimeValue;
        size_t bootTimeValueSize = sizeof(bootTimeValue);
        if ( ( sysctlbyname("kern.boottime", &bootTimeValue, &bootTimeValueSize, NULL, 0) ==
            0 ) && (bootTimeValue.tv_sec != 0) ) {
            if ( dyldCacheStatInfo.st_mtime < bootTimeValue.tv_sec ) {
                ::unlink(MACOSX_DYLD_SHARED_CACHE_DIR DYLD_SHARED_CACHE_BASE_NAME
                    ARCH_NAME);
                gLinkContext.sharedRegionMode = ImageLoader::kDontUseSharedRegion;
                return;
            }
        }
    }
}
}
}
#endif
int fd = openSharedCacheFile();
if ( fd != -1 ) {
    uint8_t firstPages[8192];
    if ( ::read(fd, firstPages, 8192) == 8192 ) {
        dyld_cache_header* header = (dyld_cache_header*)firstPages;
#ifdef __x86_64__
        const char* magic = (sHaswell ? ARCH_CACHE_MAGIC_H : ARCH_CACHE_MAGIC);
#else
        const char* magic = ARCH_CACHE_MAGIC;
#endif
        if ( strcmp(header->magic, magic) == 0 ) {
            const dyld_cache_mapping_info* const fileMappingsStart =
                (dyld_cache_mapping_info*)&firstPages[header->mappingOffset];
            const dyld_cache_mapping_info* const fileMappingsEnd =
                &fileMappingsStart[header->mappingCount];
            shared_file_mapping_np mappings[header->mappingCount+1]; // add room for
                code-sig

            .....

            if (_shared_region_map_and_slide_np(fd, mappingCount, mappings,
                codeSignatureMappingIndex, cacheSlide, slideInfo, slideInfoSize) == 0) {
                sSharedCache = (dyld_cache_header*)mappings[0].sfm_address;
                sSharedCacheSlide = cacheSlide;
                dyld::gProcessInfo->sharedCacheSlide = cacheSlide;
                .....
            }
            else {
#ifdef __IPHONE_OS_VERSION_MIN_REQUIRED
                throw "dyld shared cache could not be mapped";
#endif
            }
            if ( gLinkContext.verboseMapping )
                dyld::log("dyld: shared cached file could not be mapped\n");
        }
    }
}

```

```

    }
  }
  else {
    if ( gLinkContext.verboseMapping )
      dyld::log("dyld: shared cached file is invalid\n");
  }
}
else {
  if ( gLinkContext.verboseMapping )
    dyld::log("dyld: shared cached file cannot be read\n");
}
close(fd);
}
else {
  if ( gLinkContext.verboseMapping )
    dyld::log("dyld: shared cached file cannot be opened\n");
}
}
.....
}

```

共享缓存加载完毕后，接着进行动态库的版本化重载，这主要通过函数`checkVersionedPaths()`完成。该函数读取`DYLD_VERSIONED_LIBRARY_PATH`与`DYLD_VERSIONED_FRAMEWORK_PATH`环境变量，从这两个环境变量指定的路径中搜索动态库，如果路径中动态库的`current_version`字段比已经加载的dylib的版本新，就使用新版本的库替换掉旧版本的库。

4. 第四步，加载插入的动态库

这一步循环遍历`DYLD_INSERT_LIBRARIES`环境变量中指定的动态库列表，并调用`loadInserted- Dylib()`将其加载。该函数调用`load()`完成加载工作。`load()`会调用`loadPhase0()`尝试从文件加载，`loadPhase0()`会向下调用下一层Phase来查找动态库的路径，直到`loadPhase6()`。查找的顺序为`DYLD_ROOT_PATH`→`LD_LIBRARY_PATH`→`DYLD_FRAMEWORK_PATH`→原始路径→`DYLD_FALLBACK_LIBRARY_PATH`。找到后调用`ImageLoaderMachO::instantiateFromFile()`来实例化一个`ImageLoader`，之后调用`checkandAddImage()`验证映像并将其加入到全局映像列表中。如果

loadPhase0()返回为空，则表示在路径中没有找到动态库，就会尝试从共享缓存中查找。如果找到就调用ImageLoaderMachO:: instantiateFromCache()从缓存中加载，否则就抛出没找到映像的异常。部分代码片段如下：

```
ImageLoader* load(const char* path, const LoadContext& context)
{
    .....
    if ( context.useSearchPaths && ( gLinkContext.imageSuffix != NULL ) ) {
        if ( realpath(path, realPath) != NULL )
            path = realPath;
    }

    ImageLoader* image = loadPhase0(path, orgPath, context, NULL);
    if ( image != NULL ) {
        CRSetCrashLogMessage2(NULL);
        return image;
    }

    .....
    image = loadPhase0(path, orgPath, context, &exceptions);
    #if __IPHONE_OS_VERSION_MIN_REQUIRED && DYLD_SHARED_CACHE_SUPPORT
    && !TARGET_IPHONE_SIMULATOR
    if ( (image == NULL) && cacheablePath(path) && !context.dontLoad ) {
        .....
        if ( (myerr == ENOENT) || (myerr == 0) )
        {
            const macho_header* mhInCache;
            const char* pathInCache;
            long slidelnCache;
            if ( findInSharedCacheImage(resolvedPath, false, NULL, &mhInCache, &pathInCache,
                &slidelnCache) ) {
                struct stat stat_buf;
                bzero(&stat_buf, sizeof(stat_buf));
                try {
                    image = ImageLoaderMachO::instantiateFromCache(mhInCache, pathInCache,
                        slidelnCache, stat_buf, gLinkContext);
                    image = checkandAddImage(image, context);
                }
                catch (...) {
                    image = NULL;
                }
            }
        }
    }
    #endif
    .....
    else {
        const char* msgStart = "no suitable image found. Did find:";
```

```

.....
    throw (const char*)fullMsg;
}
}

```

5. 第五步，链接主程序

这一步执行link()完成主程序的链接操作。该函数调用了ImageLoader自身的link()函数，对实例化的主程序的动态数据进行修正，达到让进程可用的目的。比较典型的也就是主程序中的符号表修正操作，它的代码片段如下：

```

void ImageLoader::link(const LinkContext& context, bool forceLazysBound, bool preflightOnly, bool neverUnload, const
RPathChain& loaderRPaths)
{
    .....
    this->recursiveLoadLibraries(context, preflightOnly, loaderRPaths);

    .....

    context.clearAllDepths();
    this->recursiveUpdateDepth(context.imageCount());

    this->recursiveRebase(context);

    .....

    this->recursiveBind(context, forceLazysBound, neverUnload);

    if ( !context.linkingMainExecutable )
        this->weakBind(context);    //现在是链接主程序，这里现在不会执行

    .....

    std::vector<DOFInfo> dofs;
    this->recursiveGetDOFSections(context, dofs);
    context.registerDOFs(dofs);

    .....

    if ( !context.linkingMainExecutable && (fglInterposingTuples.size() != 0) ) {
        this->recursiveApplyInterposing(context);    //现在是链接主程序，这里现在不会执行
    }
    .....
}

```

recursiveLoadLibraries()采用递归的方式加载程序依赖的动态库，加载的方法是调用context的

loadLibrary指针方法。该方法在前面介绍过，是setContext()设置的libraryLocator()，该函数只是调用了load()来完成加载，load()加载动态库的过程在上一步已经分析过了。

接着调用recursiveUpdateDepth()对映像及其依赖库按列表方式进行排序。recursiveRebase()则对映像完成递归操作，该函数只是调用了虚函数doRebase()。doRebase()被ImageLoaderMachO重载，实际上只是将代码段设置成可写后调用了rebase()。在ImageLoaderMachOCompressed中，该函数读取映像动态链接信息的rebase_off与rebase_size来确定需要rebase的数据偏移与大小，然后逐个修正它们的地址信息。

recursiveBind()完成递归绑定符号表的操作。此处针对的是非延迟加载的符号表，核心是调用了doBind()。在ImageLoaderMachOCompressed中，该函数读取映像动态链接信息的bind_off与bind_size来确定需要绑定的数据偏移与大小，然后逐个进行绑定。绑定操作使用bindAt()函数，该函数调用resolve()解析完符号表后，会调用bindLocation()完成最终的绑定操作。需要绑定的符号信息有以下3种。

- ❑ BIND_TYPE_POINTER：需要绑定的是一个指针。直接将计算好的新值赋值即可。
- ❑ BIND_TYPE_TEXT_ABSOLUTE32：需要绑定的是一个32位的绝对地址。
- ❑ BIND_TYPE_TEXT_PCREL32：重定位符号。需要使用新值减掉需要修正的地址值来计算
出重定位值。

recursiveGetDOFSections()与registerDOFs()主要注册程序的DOF节区，供dtrace使用。

6. 第六步，链接插入的动态库

链接插入的动态库与链接主程序一样，都是使用link()。插入的动态库列表是前面调用addImage()保存到sAllImages中的，之后，循环获取每一个动态库的ImageLoader，调用link()对其进行链接。注意，sAllImages中保存的第一项是主程序的映像。

接下来调用每个映像的registerInterposing()方法，来注册动态库插入与调用applyInterposing()应用插入操作（也叫作符号地址替换）。registerInterposing()查找__DATA段的__interpose节区，找到需要应用插入操作的数据，然后做一些检查后，将要替换的符号与被替换的符号信息存入fgInterposingTuples列表中，供以后具体符号替换时查询。applyInterposing()调用了虚方法doInterpose()来做符号替换操作，在ImageLoaderMachO-Compressed中，实际是调用了eachBind()与eachLazyBind()，分别对常规符号和延迟加载符号进行应用插入操作。具体使用的是interposeAt()，该方法调用interposedAddress()在fgInterposingTuples中查找要替换的符号地址，找到后进行最终的符号地址替换。

7. 第七步，执行弱符号绑定

weakBind()函数执行弱符号绑定。首先通过调用context的getCoalescedImages()将sAllImages中所有含有弱符号的映像合并成一个列表，合并完成后调用initializeCoalliterator()对映像进行排序，排序完成后调用incrementCoalliterator()收集需要绑定的弱符号。incrementCoalliterator()是一个虚函数，在ImageLoaderMachOCompressed中，该函数读取映像动态链接信息的weak_bind_off与weak_bind_size来确定弱符号的数据偏移与大小，然后逐个计算它们的地址信息。之后调用

`getAddressCoallterator()`，按照映像的加载顺序在导出表中查找符号的地址，找到后调用 `updateUsesCoallterator()` 执行最终的绑定操作。执行绑定的是 `bindLocation()`，我们在前面讲过，此处不再赘述。

8. 第八步，执行初始化方法

执行初始化的方法是 `initializeMainExecutable()`。该函数主要执行 `runInitializers()`，它调用了 `ImageLoader` 的 `runInitializers()` 方法，最终迭代执行了 `ImageLoaderMachO` 的 `doInitialization()` 方法，后者主要调用 `doImageInit()` 和 `doModInitFunctions()`，执行映像与模块中设置为 `init` 的函数和静态初始化方法。代码如下：

```
bool ImageLoaderMachO::doInitialization(const LinkContext& context)
{
    CRSetCrashLogMessage2(this->getPath());

    doImageInit(context);
    doModInitFunctions(context);

    CRSetCrashLogMessage2(NULL);

    return (fHasDashInit || fHasInitializers);
}
```

9. 第九步，查找入口点并返回

这一步调用主程序映像的 `getThreadPC()` 函数来查找主程序的 `LC_MAIN` 加载命令，获取程序的入口。如果没找到，就调用 `getMain()` 到 `LC_UNIXTHREAD` 加载命令中去找，找到后就跳到入口点指定的地址并返回。

到这里，`dyld` 加载动态库的整个过程就完成了。

下面，我们再讨论一下延迟符号加载的技术细节。在所有拥有延迟加载符号的 Mach-O 文件

里，符号表中一定有一个dyld_stub_helper符号，它是延迟符号加载的关键。延迟绑定符号的修正工作就是由它完成的。绑定符号信息可以使用Xcode提供的命令行工具dyldinfo来查看，执行以下命令可以查看Python的绑定信息：

```
xcrun dyldinfo -bind /usr/bin/python
for arch i386:
bind information:
segment section      address      type  addend dylib      symbol
__DATA __cfstring    0x000040F0  pointer  0 CoreFoundation  __CFConstantStringClassReference
__DATA __cfstring    0x00004100  pointer  0 CoreFoundation  __CFConstantStringClassReference
__DATA __nl_symbol_ptr 0x00004010  pointer  0 CoreFoundation  _kCFAllocatorNull
__DATA __nl_symbol_ptr 0x00004008  pointer  0 libSystem      __stack_chk_guard
__DATA __nl_symbol_ptr 0x0000400C  pointer  0 libSystem      _environ
__DATA __nl_symbol_ptr 0x00004000  pointer  0 libSystem      dyld_stub_binder
bind information:
segment section      address      type  addend dylib      symbol
__DATA __cfstring    0x1000031D8  pointer  0 CoreFoundation  __CFConstantStringClassReference
__DATA __cfstring    0x1000031F8  pointer  0 CoreFoundation  __CFConstantStringClassReference
__DATA __got         0x100003010  pointer  0 CoreFoundation  _kCFAllocatorNull
__DATA __got         0x100003000  pointer  0 libSystem      __stack_chk_guard
__DATA __got         0x100003008  pointer  0 libSystem      _environ
__DATA __nl_symbol_ptr 0x100003018  pointer  0 libSystem      dyld_stub_binder
```

所有的延迟绑定符号都存储在_TEXT段的stubs节区。编译器在生成代码时创建的符号调用就生成在此节区中，该节区被称为“桩”节区。桩只是一小段临时使用的指令，在stubs中只是一条jmp跳转指令，跳转的地址位于__DATA段__la_symbol_ptr节区中，指向的是一段代码，类似于如下的语句：

```
push xxx
jmp yyy
```

xxx是符号在动态链接信息中延迟绑定符号数据的偏移值，yyy则是跳转到_TEXT段的

stub_helper节区头部。此处的代码通常如下所示：

```
lea    r11, qword [ds:zzz]
push   r11
jmp    qword [ds:imp___nl_symbol_ptr_dyld_stub_binder]
```


jmp跳转的地址是__DATA段中__nl_symbol_ptr节区，指向的是符号dyld_stub_binder()。该函数由dyld导出，实现位于dyld源代码的dyld_stub_binder.s文件中，调用dyld::fastBindLazySymbol()来绑定延迟加载的符号。而这是一个虚函数，实际上是调用ImageLoaderMachOCompressed的doBindFastLazySymbol()，该函数调用bindAt()解析并返回正确的符号地址，dyld_stub_binder()在最后跳转到符号地址去执行。这一步完成后，__DATA段__la_symbol_ptr节区中存储的符号地址就是修正后的地址了，下一次调用该符号时，就会直接跳转到真正的符号地址去执行，而不用dyld_stub_binder()来重新解析该符号了。

4.7 静态库

静态库与动态库都是Mach-O格式的文件，动态库使用.dylib作为文件的扩展名，静态库则使用.a作为文件的扩展名。在功能上，动态库通过动态链接的方式向其他程序提供接口，而静态库则将功能代码直接编译进目标Mach-O文件中。多个程序使用同一个动态库并不会增加目标文件的大小，使用静态库则会将每份功能代码都复制到目标文件中。从运行效率上来说，动态库需要在加载后做符号绑定操作，而静态库代码直接在目标程序中运行。因此，理论上来讲，使用静态库的运行效率比动态库要高一些。

4.7.1 构建静态库

Xcode提供了创建静态库的工程模板，创建静态库的方法与创建动态库几乎一样，唯一不同的是在项目设置时，Type选择Static。下面是与创建动态库一样的代码：

```
#import <Foundation/Foundation.h>
```

```
@interface mystaticlib : NSObject
-(void) hello;
@end
```

```
#import "mystaticlib.h"
```

```
@implementation mystaticlib
-(void) hello {
    NSLog(@"hello world");
}
@end
```

分别保存为mystaticlib.h与mystaticlib.m，然后使用xcodebuild编译会有如下输出：

```
$ xcodebuild
=== BUILD TARGET mystaticlib OF PROJECT mystaticlib WITH THE DEFAULT CONFIGURATION (Release) ===

Check dependencies

Write auxiliary files
write-file
/Users/macbook/Documents/macbook/macbook/code/chapter4/mystaticlib/build/mystaticlib.build/Release/mys
taticlib.build/mystaticlib-generated-files.hmap
write-file
/Users/macbook/Documents/macbook/macbook/code/chapter4/mystaticlib/build/mystaticlib.build/Release/mys
taticlib.build/mystaticlib-all-target-headers.hmap
.....
/bin/mkdir -p
/Users/macbook/Documents/macbook/macbook/code/chapter4/mystaticlib/build/mystaticlib.build/Release/mys
taticlib.build/Objects-normal/x86_64
write-file
/Users/macbook/Documents/macbook/macbook/code/chapter4/mystaticlib/build/mystaticlib.build/Release/mys
taticlib.build/Objects-normal/x86_64/mystaticlib.LinkFileList

CompileC build/mystaticlib.build/Release/mystaticlib.build/Objects-normal/x86_64/mystaticlib.o
mystaticlib/mystaticlib.m normal x86_64 objective-c com.apple.compilers.llvm.clang.1_0.compiler
cd /Users/macbook/Documents/macbook/macbook/code/chapter4/mystaticlib
export LANG=en_US.US-ASCII
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/clang -x objective-c
-arch x86_64 -fmessage-length=94 -fdiagnostics-show-note-include-stack -fmacro-backtrace-limit=0
-fcolor-diagnostics -std=gnu99 -fobjc-arc -fmodules -gmodules -fmodules-prune-interval=86400
-fmodules-prune-after=345600
-fbuild-session-file=/var/folders/rd/mts0362j0n92rq0z1cnmdb580000gn/C/org.llvm.clang/ModuleCache/Sessi
on.modulevalidation -fmodules-validate-once-per-build-session -Wnon-modular-include-in-framework-module
-Werror=non-modular-include-in-framework-module -Wno-trigraphs -fpascal-strings -Os -fno-common
-Wno-missing-field-initializers -Wno-missing-prototypes -Werror=return-type -Wunreachable-code
-Wno-implicit-atomic-properties -Werror=deprecated-objc-isa-usage -Werror=objc-root-class
-Wno-arc-repeated-use-of-weak -Wduplicate-method-match -Wno-missing-braces -Wparentheses -Wswitch
-Wunused-function -Wno-unused-label -Wno-unused-parameter -Wunused-variable -Wunused-value -Wempty-body
-Wconditional-uninitialized -Wno-unknown-pragmas -Wno-shadow -Wno-four-char-constants -Wno-conversion
```

```

-Wconstant-conversion -Wint-conversion -Wbool-conversion -Wenum-conversion -Wshorten-64-to-32
-Wpointer-sign -Wno-newline-eof -Wno-selector -Wno-strict-selector-match -Wundeclared-selector
-Wno-deprecated-implementations -DNS_BLOCK_ASSERTIONS=1 -DOBJC_OLD_DISPATCH_PROTOTYPES=0
-isysroot
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX10.11.sdk
-fasm-blocks -fstrict-aliasing -Wprotocol -Wdeprecated-declarations -mmacosx-version-min=10.11 -g
-Wno-sign-conversion -iquote
/Users/macbook/Documents/macbook/macbook/code/chapter4/mystaticlib/build/mystaticlib.build/Release
/mystaticlib.build/mystaticlib-generated-files.hmap
-I/Users/macbook/Documents/macbook/macbook/code/chapter4/mystaticlib/build/mystaticlib.build/Release/m
ystaticlib.build/mystaticlib-own-target-headers.hmap
-I/Users/macbook/Documents/macbook/macbook/code/chapter4/mystaticlib/build/mystaticlib.build/Release/m
ystaticlib.build/mystaticlib-all-target-headers.hmap -iquote
/Users/macbook/Documents/macbook/macbook/code/chapter4/mystaticlib/build/mystaticlib.build/Release/mys
taticlib.build/mystaticlib-project-headers.hmap
-I/Users/macbook/Documents/macbook/macbook/code/chapter4/mystaticlib/build/Release/include
-I/Users/macbook/Documents/macbook/macbook/code/chapter4/mystaticlib/build/mystaticlib.build/Release/m
ystaticlib.build/DerivedSources/x86_64
-I/Users/macbook/Documents/macbook/macbook/code/chapter4/mystaticlib/build/mystaticlib.build/Release/m
ystaticlib.build/DerivedSources
-F/Users/macbook/Documents/macbook/macbook/code/chapter4/mystaticlib/build/Release -MMD -MT
dependencies -MF
/Users/macbook/Documents/macbook/macbook/code/chapter4/mystaticlib/build/mystaticlib.build/Release/mys
taticlib.build/Objects-normal/x86_64/mystaticlib.d --serialize-diagnostics
/Users/macbook/Documents/macbook/macbook/code/chapter4/mystaticlib/build/mystaticlib.build/Release/mys
taticlib.build/Objects-normal/x86_64/mystaticlib.dia -c
/Users/macbook/Documents/macbook/macbook/code/chapter4/mystaticlib/mystaticlib/mystaticlib.m -o
/Users/macbook/Documents/macbook/macbook/code/chapter4/mystaticlib/build/mystaticlib.build/Release/mys
taticlib.build/Objects-normal/x86_64/mystaticlib.o

Libtool build/Release/libmystaticlib.a normal x86_64
  cd /Users/macbook/Documents/macbook/macbook/code/chapter4/mystaticlib
  export MACOSX_DEPLOYMENT_TARGET=10.11
  /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/libtool -static
  -arch_only x86_64 -syslibroot
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX10.11.sdk
-L/Users/macbook/Documents/macbook/macbook/code/chapter4/mystaticlib/build/Release -filelist
/Users/macbook/Documents/macbook/macbook/code/chapter4/mystaticlib/build/mystaticlib.build/Release/mys
taticlib.build/Objects-normal/x86_64/mystaticlib.LinkFileList -o
/Users/macbook/Documents/macbook/macbook/code/chapter4/mystaticlib/build/Release/libmystaticlib.a

** BUILD SUCCEEDED **

```

整个编译过程分为：检查依赖（Check dependencies）、生成辅助文件（Write auxiliary files）、

编译（CompileC）、打包生成库（Libtool）4步。最后打包生成库的环节使用的是Libtool工具，该

工具除了生成静态库，也可以生成动态库。上一节生成动态库使用的链接器ld，它底层也是通过

libtool来生成动态库的。最后注意，静态库不需要签名，静态库中的代码最终会被插入到目标程

序中，由目标程序来签名。

4.7.2 静态库格式

上一节讲到的动态库文件格式就是标准的Mach-O文件。与Mach-O可执行文件不同的是，动态库在Mach-O头部指定文件类型为MH_DYLIB，可执行程序为MH_EXECUTE。而静态库文件不是标准的Mach-O文件，它的格式如下所示：

```
Start
Symtab Header
Symbol Table
String Table
Object Header 0
ObjName0.o
.....
Object Header N
ObjNameN.o
```

Start为静态库的开始，它是一个固定长度的签名值！<arch>\n，十六进制为21 3C 61 72 63 68 3E 0A。

Symtab Header为符号表头，描述了符号表的信息。它使用symtab_header结构体表示，具体定义如下：

```
struct symtab_header {
    char    name[16];      /* 名称 */
    char    timestamp[12]; /* 库创建的时间戳 */
    char    userid[6];     /* 用户id */
    char    groupid[4];    /* 组id */
    uint64_t mode;         /* 文件访问模式 */
    uint64_t size;         /* 符号表占总字节大小 */
    uint32_t endheader;    /* 头结束标志 */
    char    longname[20];  /* 符号表长名 */
};
```

Symbol Table为当前静态库导出的符号表。它使用symbol_table结构体表示，具体定义为：

```
struct symbol_table {
```

```

uint32_t    size;          /* 符号表占用的总字节数 */
symbol_info syminfo[0];    /* 符号信息，它的个数是 size / sizeof(symbol_info) */
};

struct symbol_info {
    uint32_t  symnameoff;    /* 符号名在字符串表数据中的偏移值 */
    uint32_t  objheaderoff; /* 符号所属的目标文件的文件头在文件中的偏移值 */
};

```

String Table为字符串表，该结构体存储的字符串信息供符号表使用，使用string_table结构体

表示，具体定义为：

```

struct string_table {
    uint32_t    size;          /* 字符串表占用的总字节数 */
    char        data[size];    /* 字符串数据 */
};

```

Object Header为目标文件的头，描述了接下来的目标文件的信息，使用object_header结构体

表示，具体定义为：

```

struct object_header {
    char        name[16];      /* 名称 */
    char        timestamp[12]; /* 目标文件创建的时间戳 */
    char        userid[6];     /* 用户id */
    char        groupid[4];    /* 组id */
    uint64_t    mode;          /* 文件访问模式 */
    uint64_t    size;          /* 符号表占总字节大小 */
    uint32_t    endheader;     /* 头结束标志 */
    char        longname[20];  /* 符号表长名 */
};

```

object_header结构体的布局与symtab_header基本一样。

在object_header结构体下面紧接着就是具体的目标文件内容了。目标文件是以.o结尾的

Mach-O格式的文件，它是由编译器生成的中间文件。目标文件在它的Mach-O头部被标识为MH_OBJECT类型的文件。

最后，可以使用MachOView查看本节生成的libmystaticlib.a的结构信息，效果如图4-11所示。

完目标文件后，可以将其打包进原来的库，或者直接生成新的静态库，执行以下命令：

```
$ ar rcs libmystaticlib_new.a *.o
```

同样没有输出信息，但ar已经将当前目录下所有的目标文件都成功打包进了libmystaticlib_new.a中。

4.8 框架

使用静态库与动态库固然方便，但无法解决引用外部库资源的问题，框架（framework）就是为了解决此问题而发明的。macOS系统大量地使用了框架技术。在System/Library/Frameworks目录下存放了系统为外部提供的框架，这些框架可以被外部应用程序调用。另外，System/Library/PrivateFrameworks目录里存放的是系统的私有框架些，供系统专用。Frameworks目录下存放的公用框架，通常在macOS的开发文档中会找到框架的介绍与使用文档，而私有框架则没有，因为私有框架提供的接口往往不太稳定、经常变动，而且对于一些比较底层的功能，苹果也不愿意暴露给开发者，因此使用私有框架的程序是不允许在App Store中发布的。

系统每次更新都会对框架做不少改动，废弃低版本的框架或者增加新的框架是常有的事，比如在macOS 10.11版本中增加了如下框架。

- ❑ Contacts.framework：联系人框架。使用现在面向对象的访问方式替换以前的地址簿框架。
- ❑ GameplayKit.framework：GameplayKit游戏框架。提供了构建游戏的基础技术，开发者可以使用GameplayKit框架，配合高级图像引擎SceneKit与SpriteKit，来开发完整的商业

化游戏。

- ❑ Metal.framework：Metal框架提供了基于GPU的3D图形渲染与分布式数据计算的接口，作用与OpenGL类似。
- ❑ MetalKit.framework：MetalKit框架包含了创建Metal程序的类与函数。
- ❑ ModelIO.framework：Model I/O框架提供了系统层面的3D模型与相关资源的解释功能接口。
- ❑ NetworkExtension.framework：网络扩展接口。用于配置与管理VPN。

安装了macOS的开发SDK后，SDK目录中会有一个框架目录，里面存放的是文档化的框架，可以直接用于程序开发。查看系统安装的SDK的路径可以执行如下命令：

```
$ xcrun --sdk macosx --show-sdk-path  
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX10.12.sdk
```

在SDK目录下，框架的目录为\$(SDK_PATH)/System/Library/Frameworks，私有框架的目录为\$(SDK_PATH)/System/Library/PrivateFrameworks。这些目录下的框架并不是真正的框架，只是指向系统框架的一个“索引框架”。拿Contacts.framework来说，该框架中有一个Headers目录，存放着框架暴露出去的接口的头文件，此目录在系统框架目录中是不存在的。另外，开发者自己创建的框架通常会包含此目录。Contacts.framework中有一个Contacts.tbd文件，它包含了该框架的一些信息，如下所示：

```
---  
archs:      [ x86_64 ]  
platform:   macosx  
install-name: /System/Library/Frameworks/Contacts.framework/Versions/A/Contacts  
current-version: 0.0  
compatibility-version: 0.0  
exports:
```



```

- archs:      [ x86_64 ]
symbols:      [ _CNAActivityAlertCallActivityKey, _CNAActivityAlertSoundKey,
               .....,
               _CNSocialProfileUsernameKey, _CNTelephonyServiceName,
               _CNWorkLabel ]
objc-classes: [ _CNAccount, _CNAActivityAlert, _CNCDCContactFetcher,
               .....,
               _CNPredicate, _CNPredicateValidator, _CNSaveRequest,
               _CNSocialProfile, _CNSuggestedSaveRequest, _CNTCC ]
objc-ivars:   [ _CNCDCContactFetcher._fetchRequestDescription, _CNCDCContactFetcher._
               persistenceContext,
               .....,
               _CNPostalAddress._subAdministrativeArea ]
.....

```

archs指明了框架支持的CPU架构，Contacts框架只支持x86_64，如果支持多个架构，会在此处列出来；platform指明了框架运行的系统平台；install-name指明了框架的路径，这才是框架的真正所在；current-version与compatibility-version指明了当前版本号与兼容版本号；symbols是框架向外提供的符号列表。objc-classes与objc-ivars提供了Objc类与变量。

4.8.1 构建框架

Xcode提供了构建框架的工程模板，方法与创建静态库和动态库一样，在选择项目模板时，选择Cocoa Framework，然后点击Next输入项目的名称即可。开发框架可以使用Swift或Objective-C语言，项目创建完成后，默认生成myframework.h头文件，它的代码如下：

```

#import <Cocoa/Cocoa.h>

FOUNDATION_EXPORT double myframeworkVersionNumber;

FOUNDATION_EXPORT const unsigned char myframeworkVersionString[];

```

将4.6节mylib项目的mylib.h与mylib.m文件添加到项目中，然后选择myframework项目，在Build Phases标签页中，展开Headers一项，将mylib.h设置为Public，如图4-12所示。

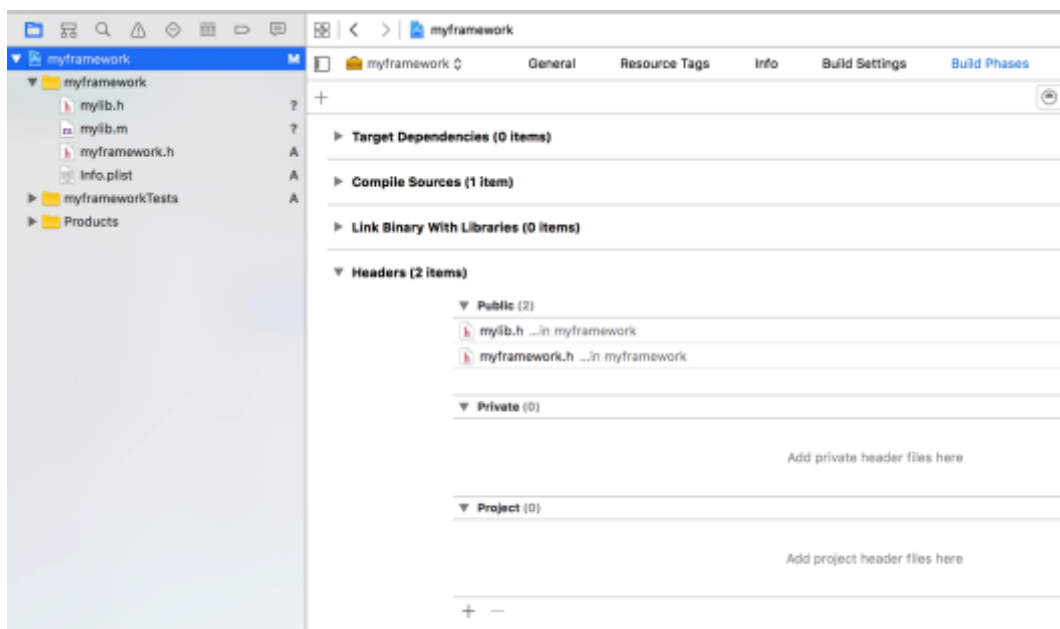


图4-12 将mylib.h设置为Public

接着在myframework.h中添加一行代码：`#import "mylib.h"`。

以后每次添加接口，都可以将头文件添加到myframework.h中。使用框架时，只需要包括myframework.h到项目中就可以了。以上操作完成后，点击Xcode菜单Product→Build，或者按键盘的command+B就会成功生成框架。

使用Swift创建框架的方法相同。Swift创建框架如果是给Swift调用，通常不需要导出头文件，只需要将类与方法都设置成public即可，编译器编译成功后会自动生成头文件。详见本节演示程序myframeworkswift代码。

4.8.2 框架的使用与安装

生成的框架可以直接拿来供程序使用，创建新的Cocoa Application工程myframeworktest，将上一节生成的myframework.framework直接拖入到项目中，会弹出如图4-13所示的对话框。

勾选“Copy items if needed”选项后，点击Finish完成框架的导入。接着选中“myframeworktest”项目，在General的“Embedded Binaries”选项中点击加号“+”，将myframework.framework导入。

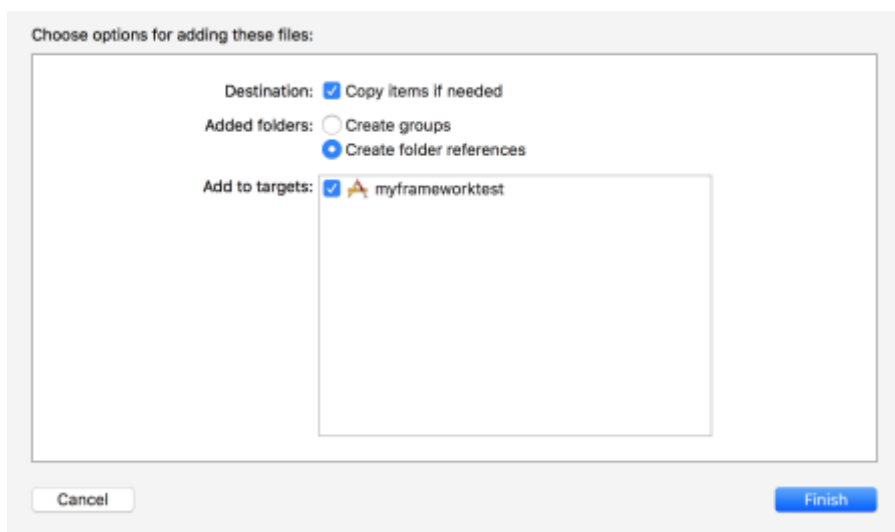


图4-13 框架选择对话框

框架导入完成后，为myframeworktest添加框架导入语句#import "myframework/myframework.h"，

然后在applicationDidFinishLaunching()方法中添加如下代码：

```
- (void)applicationDidFinishLaunching:(NSNotification *)aNotification {
    mylib* l = [[mylib alloc] init];
    [l hello];
}
```

保存代码后，点击Xcode界面上的运行按钮直接编译运行程序，成功的话就会在输出窗口输出“hello world”。

Swift版本的框架使用方法相同，此处不再展开，详见本节演示程序myframeworkswifttest的代码。

框架可以供自己单独使用，最终会嵌入到App程序包中，也可以安装到系统中，供其他程序使用。macOS系统允许开发人员将框架安装到~/Library/Frameworks目录下，以便第三方程序调用。

以下是安装了三星公司的Smart Switch程序后在系统中安装的框架。

```
$ ls -l ~/Library/Frameworks/
total 0
drwxr-xr-x  7 ... May  3 18:55 SamsungKiesFoundation.framework
drwxr-xr-x  6 ... Mar  6 06:02 SamsungKiesSerialPort.framework
```

4

4.9 pkg

不同的操作系统有专属的软件安装包格式。如Ubuntu系统上的deb安装包、Windows系统上的msi安装包等。macOS系统使用pkg作为软件安装包格式。

macOS上开发的大多数程序都不需要安装，它们是以.app结尾的Bundle包。通常以zip压缩包或者dmg镜像的方式进行发布。然而，一些App可能有特定的需求，比如，向系统配置面板写配置程序、安装屏幕保护程序、读写特定的目录与文件等。对于这些特殊的程序，就可以通过制作pkg安装程序来安装。当然，由于这些特殊性，pkg安装程序无法通过苹果官方商店来发布。

4.9.1 构建 pkg

pkg安装程序能够扩展程序安装内容以及读写特定目录的特性。来源于pkg支持的脚本特性，pkg安装程序允许开发人员在程序安装过程中，运行自己编写的Bash脚本程序。

苹果官方在低版本的Xcode工具中提供了用来制作pkg的PackageMaker。该工具没有直接包含在Xcode开发套件中，需要到苹果的开发者官网上下载。下载安装好该工具后，运行PackageMaker.app就可以制作pkg了。

本节将制作一个pkg安装包，完成以下目标：将上一节的myframeworktest程序安装到Applications目录下，将myframework.framework框架安装到~/Library/frameworks目录中。启动PackageMaker，点击菜单File→New，在弹出的对话框中，在Organization一栏输入机构信息，如“com.macbook”，点击OK返回程序主界面，点击File→Save，将工程保存为pkg_install.pmdoc。

将上一节的myframeworktest程序放到app目录下，并将app目录拖入PackageMaker的主界面，会自动为程序添加配置信息。点击Configuration可以设置一些安装时的配置信息，如图4-14所示。

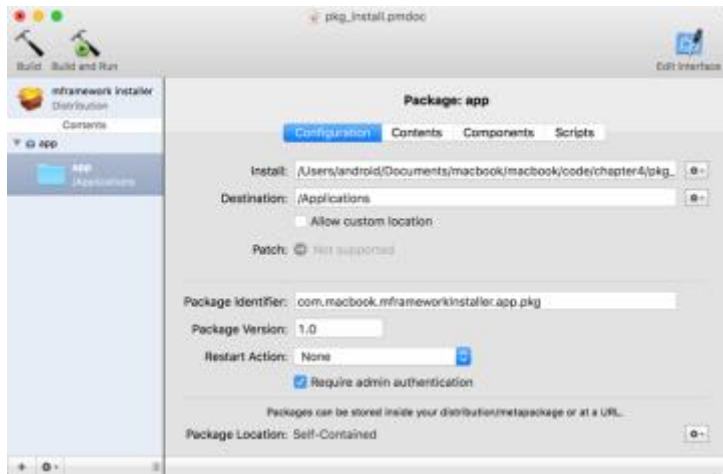


图4-14 设置安装时的配置信息

install指定要安装的程序路径，这里已经指定好了；Destination指定程序要安装的位置，默认

为/Applications目录；取消勾选“Allow custom location”选项，让程序只能安装到/Applications目录下；Package Identifier指定安装包的标识符，macOS记录安装过的pkg就是通过它来识别的，手动卸载pkg时需要用到它；Package Version指定安装包的版本，版本号是识别pkg版本升级的关键，为pkg指定升级脚本时需要用到；Restart Action指定pkg安装完成后，是否需要执行注销、关机或重启等操作；Require admin authentication复选框指定安装器需要管理员权限，如果为pkg指定的安装脚本需要管理员权限，就需要在此勾选上。

接下来，点击Contents标签，配置需要安装的内容。PackageMaker已经默认选择好了要安装的内容为myframeworktest.app，并且文件的读写与执行权限也自动设置好了，如图4-15所示。

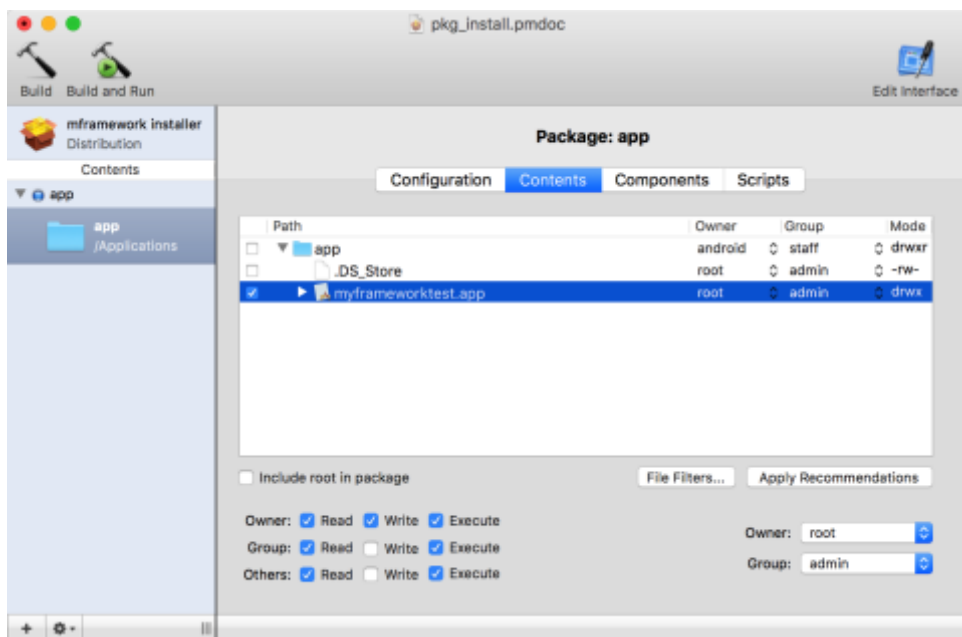


图4-15 读写与执行权限已自动设置

关于文件的读写权限，一个建议的设置如表4-1所示。

表4-1 Contents权限设置

	Owner	Group	Permissions
Applications	root	admin	rwXrwxr-X
System	root	admin	rwXrwxr-X
Library	root	admin	rwXrwxr-X
Extensions	root	admin	rwXrwxr-X

对于拖入Contents中的程序，macOS系统会在操作过的文件夹中生成一个隐藏的.DS_Store文件。如果系统开启了显示隐藏文件的选项，那么直接打包程序会在安装包中包含隐藏的.DS_Store文件。因此，在拖入Contents前需要将它们全部删除，使用如下命令即可：

```
find ./ -name ".DS_Store" -exec rm -f {} \;
```

点击Components标签，配置组件信息。取消“Allow Relocation”复选框，否则即使提示安装完成，在/Applications目录下也看不到安装后的程序，如图4-16所示。

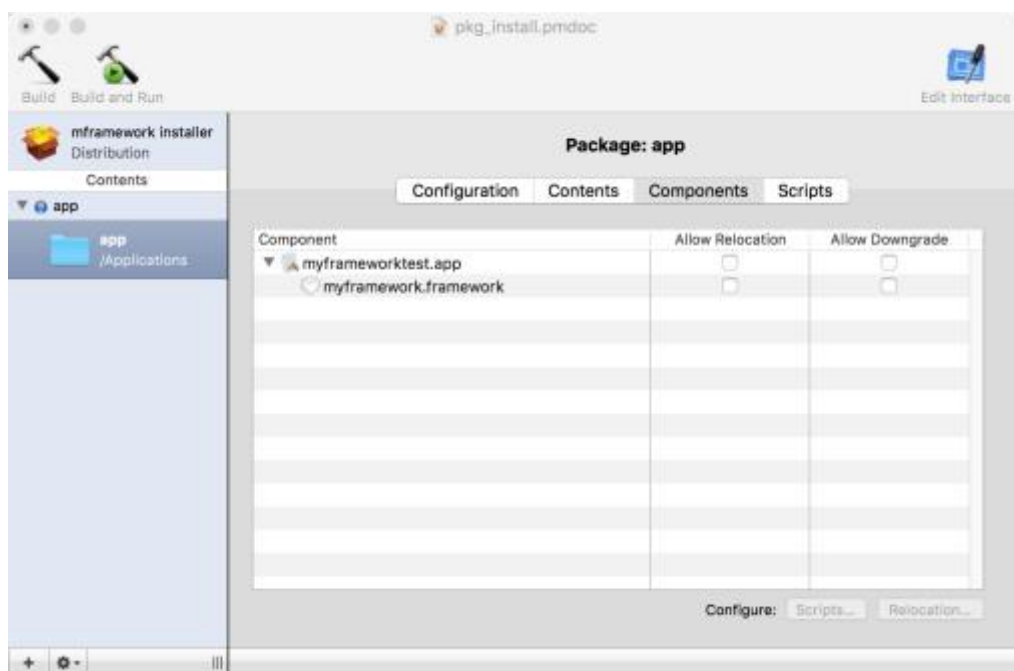


图4-16 配置组件信息

点击Scripts标签，配置运行安装器时需要执行的脚本。将编写好的脚本分别保存为preflight与postflight，然后将它们放到script目录下，执行以下命令为它们赋予可执行权限：

```
$ chmod a+x ./preinstall
$ chmod a+x ./postflight
```

将script目录直接拖入Scripts Directory旁的文本框中，此时，Preflight与Postflight脚本会自动设置完成，如图4-17所示。

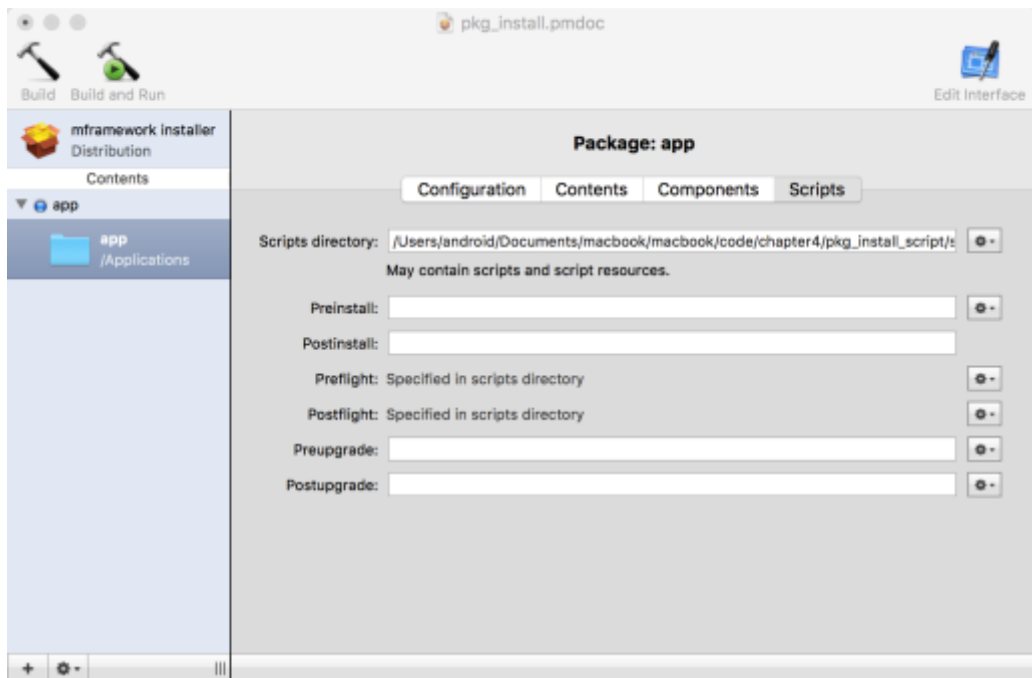


图4-17 自动设置脚本

可以设置的脚本有6个，下面按照它们的执行顺序——介绍。

- ❑ preflight：点击安装界面上的Install按钮时运行此脚本。该脚本在程序每次安装时都会运行。
- ❑ preinstall/preupgrade：针对单程序安装包，该脚本会在preflight脚本运行之后运行；针对多程序安装包，该脚本会在用户按下Install按钮后执行。preinstall与preupgrade的区别在于，preinstall只会在用户第一次安装该程序时执行，而preupgrade相反，只有之前安装过该程序，该脚本才会执行，因此，preupgrade常在软件升级时使用。区分程序是否为第一次安装是通过pkg安装器Installer.app完成的，Installer.app通过查看/private/var/db/receipts目录中

是否有以程序包名命名的pkg文件，如果存在，说明已经安装过，否则为第一次安装。

- ❑ postinstall/postupgrade：该脚本在程序安装完之后才运行。它们的区别与preinstall/preupgrade一样。
- ❑ postflight：该脚本在postinstall/postupgrade脚本之后运行。

PackageMaker支持Shell脚本与Perl脚本，此处编写的是Shell脚本，preinstall脚本的内容

4

如下：

```
#!/usr/bin/env bash

echo "Running myframeworktest.app preinstall script."
echo "Killing myframeworktest.app."
killall "myframeworktest"

echo "Finding old versions of myframeworktest."
mdfind -onlyin /Applications "kMDItemCFBundleIdentifier = 'fc.myframeworktest'" | xargs -l % rm -rf %

echo "Removed old versions of myframeworktest.app, if any."
echo "Ran myframeworktest.app preinstall script."

exit 0
```

这段脚本首先使用killall杀掉正在运行的myframeworktest.app进程，接着使用mdfind在/Applications目录下查找程序标识符为“fc.myframeworktest”的程序包路径，找到后使用rm -rf将其删除。

再看看postflight脚本的内容：

```
#!/usr/bin/env bash

echo "Running myframeworktest.app postinstall script."
echo "Installing myframework.framework."

rm -rf ~/Library/Frameworks/myframework.framework
mkdir ~/Library/Frameworks/myframework.framework
cp -r /Applications/myframeworktest.app/Contents/Frameworks/myframework.framework/*
```

```
~/Library/Frameworks/myframework.framework  
  
chmod -R 6777 ~/Library/Frameworks/myframework.framework  
echo "Ran myframeworktest.app postinstall script."  
  
exit 0
```

在该脚本运行时，myframeworktest.app程序包已经安装到了/Applications目录下，将myframework.framework复制到~/Library/Frameworks目录下，然后修改它的权限为任何人都可读可写可执行，执行完后调用exit 0退出脚本。

配置好要安装的内容与执行脚本后，点击Contents上面的图标，对pkg进行配置。点击Configuration，在Title旁的文本框中输入安装包的标题，例如“mframeworktest installer”；在User Sees处选择Easy Install Only（简单安装）即可；在Install Destination处勾选Volume selected by user。

点击Requirements标签，设置pkg运行的系统要求。点击界面左下角的加号“+”按钮，添加两条规则：一条是System OS Version(e.g. 10.x.x)，另一条是Target OS Version(e.g. 10.x.x)，都设置成 ≥ 10.6 ，如图4-18所示。

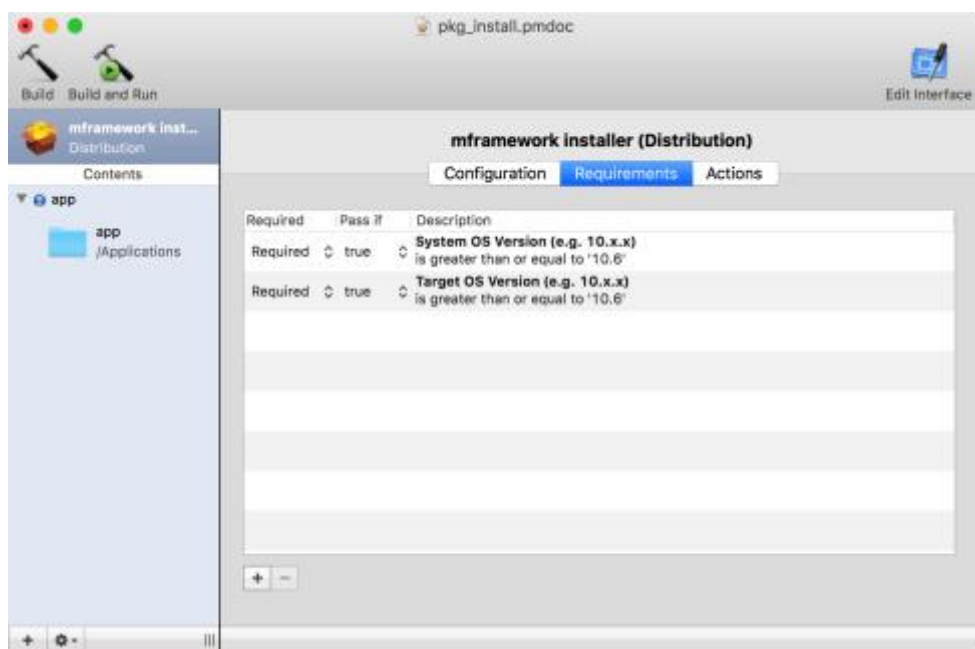


图4-18 添加规则

最后的Actions标签页不用管。配置完了后，点击界面右上角的Edit interface按钮，编辑安装程序的界面，包括：Background、Introduction、Read me、License与Finish up。每一项都是一个页面，内容可以选择系统默认的Default，也可以直接写一段文本嵌入进去，或者选择一个外部的rtf文档或html网页。图4-19所示为一段手写的Read Me。

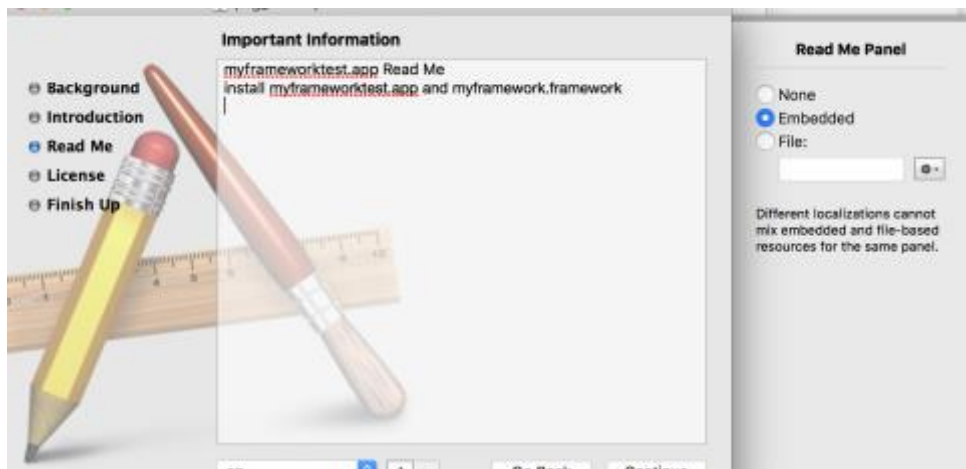


图4-19 Read Me

以上所有操作完成后，点击PackageMaker左上角的Build按钮进行构建，或者点击Build and Run按钮构建成功后直接运行。构建完成后会针对单程序安装包或多程序安装包生成一个pkg或mpkg文件，该文件是可以发布的产品，接下来只需要对其进行安装测试，没问题就可以发布了。

新版本的Xcode提供了命令行工具productbuild来打包制作pkg。本节PackageMaker操作的步骤可以通过执行以下命令完成：

```
$ productbuild --component app/myframeworktest.app /Applications --scripts script ~/Desktop/out.pkg
```

命令执行完后，就会在当前用户桌面上生成pkg文件，当然编译时可以指定--sign参数来为pkg签名。但pkg签名不使用codesign，如果创建pkg时没有对其进行签名，或者手动修改过pkg的内容，可以使用工具productsign来对pkg进行签名。

介绍了官方的pkg创建工具后，下面再来看看目前市面上常用的pkg制作工具。喜欢命令行编

译的开发人员一定会喜欢工具Luggage^①，它提供了一种自定义脚本的方式来编译构建pkg文件。

该工具的使用方法很简单，只需要将GitHub上的文件复制到/usr/local/share/luggage就完成了安装。

至于脚本如何编写，可以参考Luggage提供的样例^②。以编译样例中的fex程序为例，在命令行下

执行以下命令：

```
$ make

Usage

make clean - clean up work files.
make dmg   - roll a pkg, then stuff it into a dmg file.
make zip   - roll a pkg, then stuff it into a zip file.
make pkg   - roll a pkg.
make pkgls - list the bill of materials that will be generated by the pkg.

$ make pkg
Password:
make -f Makefile -e pack-fex

Disabling bundle relocation.
If you need to override permissions or ownerships, override modify_packageroot in your Makefile
Creating /tmp/the_luggage/Fex-20160902/payload/Fex-20160902.pkg with /usr/bin/pkgbuild.
sudo /usr/bin/pkgbuild --root /tmp/the_luggage/Fex-20160902/root \
    --component-plist /tmp/the_luggage/Fex-20160902/luggage.pkg.component.plist \
    --identifier com.huronhs.Fex \
    --filter "/CVSS$" --filter "/\..svn$" --filter "/\..cvsignore$" --filter "/\..cvspass$"
    --filter "/(\.?)?.DS_Store$" --filter "/\..git$" --filter "/\..gitignore$" \
    --scripts /tmp/the_luggage/Fex-20160902/scripts \
    --version 20160902 \
    --ownership preserve --quiet \
    /tmp/the_luggage/Fex-20160902/payload/Fex-20160902.pkg

$ ls
Fex-20160902.pkg      Makefile              fex
```

从输出中可以看出，除了构建pkg，Luggage还支持生成dmg与zip打包的程序，非常方便。

与Luggage类似的还有createOSXinstallPkg^③，使用方法也很简单，有兴趣的读者可以到GitHub

① Luggage下载地址：<https://github.com/unixorn/luggage>。

② Luggage样例地址：<https://github.com/unixorn/luggage-examples>。

③ createOSXinstallPkg下载地址：<https://github.com/munki/createOSXinstallPkg>。

上查看如何使用。

最后,还有一款强大且免费的pkg安装包制作工具Iceberg^①,该工具可以修改安装程序界面的背景图片。此处就不具体讨论它的用法了,有兴趣的读者可以去官网下载试用。

4.9.2 pkg 的安装与卸载

安装pkg很简单,只要双击pkg,或者双击mpkg,就会弹出安装向导,按照步骤不停点击Next,直到安装完成。在安装过程中,执行一些操作可能需要管理器权限,系统会弹出提示,要求用户输入管理员密码,按照操作输入密码即可。除了双击安装外,还可以使用命令行工具installer进行静默安装。执行以下命令可以安装上一节的pkg:

```
$ sudo installer -pkg ./myframework_installer.pkg -target LocalSystem
Password:
installer: Package name is myframework installer
installer: Upgrading at base path /
installer: The upgrade was successful.
```

pkg的卸载就没这么简单了!苹果公司没有提供直接卸载pkg的方法。上一节我们没有制作pkg格式的卸载程序,而是编写了一个简单的脚本,只需要双击运行它就可以卸载上一节制作的pkg

了。脚本的代码如下:

```
#!/usr/bin/env bash

if [ -d ~/Library/Frameworks/myframework.framework ]; then
    /bin/rm -rf ~/Library/Frameworks/myframework.framework
fi

if [ -d /Application/myframework.app ]; then
    /bin/rm -rf /Applications/myframeworktest.app
fi
```

① Iceberg下载地址: <http://s.sudre.free.fr/Software/Iceberg.html>。

echo done.

对于没有提供卸载程序的pkg，就只能手动卸载，或者使用第三方工具，例如UninstallPKG^①，这是一款收费软件，安装运行后，它会收集系统中安装的所有pkg软件，然后以列表形式展示出来，如图4-20所示。

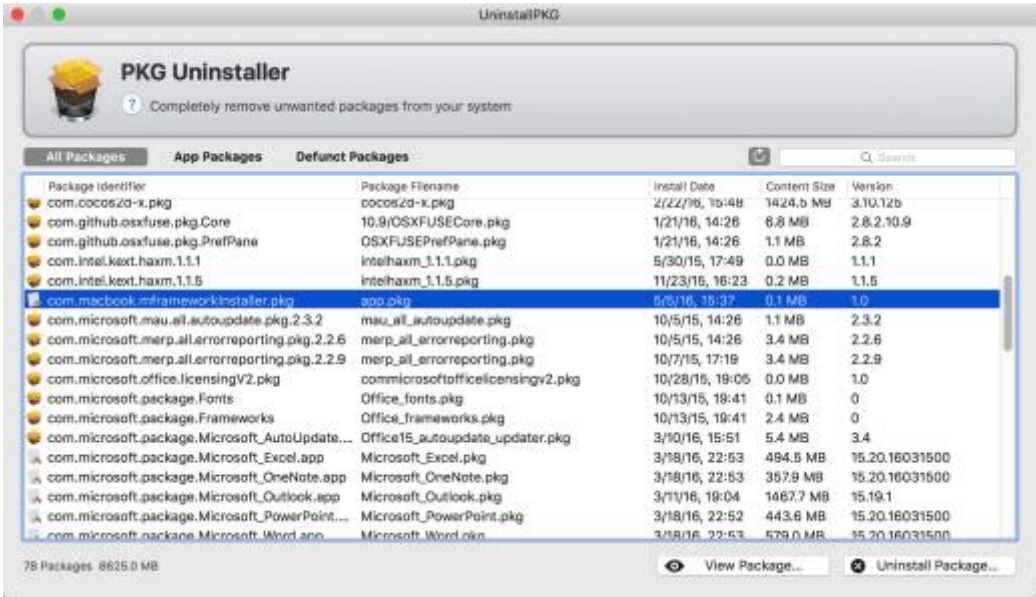


图4-20 系统中安装的所有pkg软件

点击View Package...按钮，可以查看pkg在系统中写入了哪些文件内容，点击Uninstall Package...，可以直接卸载pkg。

UninstallPKG是如何收集与卸载系统中安装的pkg的呢？其实原理很简单。它读取/private/var/db/receipts下的pkg列表，然后使用lsbom查看这些pkg文件的bom信息。找到bom文件中保存的文件列表，将它们列举出来，卸载的时候将它们全部删除即可。执行如下命令列表就可

以查看上一节pkg的信息：

```
$ cd /private/var/db/receipts
$ ls | grep macbook
com.macbook.myframeworkInstaller.pkg.bom
com.macbook.myframeworkInstaller.pkg.plist
$ lsbom -pf ./com.macbook.myframeworkInstaller.pkg.bom
.
./myframeworktest.app
./myframeworktest.app/Contents
./myframeworktest.app/Contents/Frameworks
./myframeworktest.app/Contents/Frameworks/myframework.framework
./myframeworktest.app/Contents/Frameworks/myframework.framework/Resources
./myframeworktest.app/Contents/Frameworks/myframework.framework/Versions
./myframeworktest.app/Contents/Frameworks/myframework.framework/Versions/A
./myframeworktest.app/Contents/Frameworks/myframework.framework/Versions/A/Resources
./myframeworktest.app/Contents/Frameworks/myframework.framework/Versions/A/Resources/Info.plist
./myframeworktest.app/Contents/Frameworks/myframework.framework/Versions/A/_CodeSignature
./myframeworktest.app/Contents/Frameworks/myframework.framework/Versions/A/_CodeSignature/CodeResources
./myframeworktest.app/Contents/Frameworks/myframework.framework/Versions/A/myframework
./myframeworktest.app/Contents/Frameworks/myframework.framework/Versions/Current
./myframeworktest.app/Contents/Frameworks/myframework.framework/myframework
./myframeworktest.app/Contents/Info.plist
./myframeworktest.app/Contents/MacOS
./myframeworktest.app/Contents/MacOS/myframeworktest
./myframeworktest.app/Contents/PkgInfo
./myframeworktest.app/Contents/Resources
./myframeworktest.app/Contents/Resources/Base.lproj
./myframeworktest.app/Contents/Resources/Base.lproj/MainMenu.nib
./myframeworktest.app/Contents/_CodeSignature
./myframeworktest.app/Contents/_CodeSignature/CodeResources
```

观察脚本中执行的命令，可以看出，在~/Library/Frameworks目录中安装的myframework.

framework并没有列出来，而只有在Contents中指定的内容。上面查看bom信息使用的是lsbom命令，

其实，查看pkg中的内容还有一种更简单的方法：双击运行pkg后，不要点击Continue按钮，而是

点击菜单File→Show Files，pkg中包含的文件内容就一目了然了，如图4-21所示。

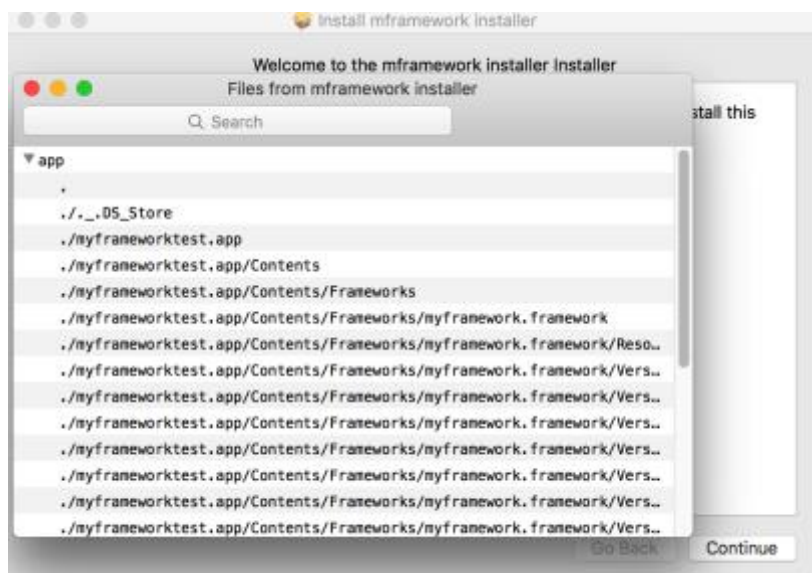


图4-21 pkg中包含的文件内容

除了到/private/var/db/receipts目录手动读取pkg列表,还可以使用pkg管理工具pkgutil来查看系统中安装的pkg信息,不过只有查看功能,不能卸载。执行如下命令可以查看上一节安装的pkg信息,效果与上面一样:

```
$ pkgutil --pkgs | grep -i com.macbook
com.macbook.myframeworkInstaller.pkg
$ pkgutil --files com.macbook.myframeworkInstaller.pkg
myframeworktest.app
myframeworktest.app/Contents
myframeworktest.app/Contents/Frameworks
myframeworktest.app/Contents/Frameworks/myframework.framework
myframeworktest.app/Contents/Frameworks/myframework.framework/Resources
myframeworktest.app/Contents/Frameworks/myframework.framework/Versions
myframeworktest.app/Contents/Frameworks/myframework.framework/Versions/A
myframeworktest.app/Contents/Frameworks/myframework.framework/Versions/A/Resources
myframeworktest.app/Contents/Frameworks/myframework.framework/Versions/A/Resources/Info.plist
myframeworktest.app/Contents/Frameworks/myframework.framework/Versions/A/_CodeSignature
myframeworktest.app/Contents/Frameworks/myframework.framework/Versions/A/_CodeSignature/CodeResources
myframeworktest.app/Contents/Frameworks/myframework.framework/Versions/A/myframework
myframeworktest.app/Contents/Frameworks/myframework.framework/Versions/Current
myframeworktest.app/Contents/Frameworks/myframework.framework/myframework
myframeworktest.app/Contents/Info.plist
myframeworktest.app/Contents/MacOS
myframeworktest.app/Contents/MacOS/myframeworktest
```

```
myframeworktest.app/Contents/PkgInfo
myframeworktest.app/Contents/Resources
myframeworktest.app/Contents/Resources/Base.lproj
myframeworktest.app/Contents/Resources/Base.lproj/MainMenu.nib
myframeworktest.app/Contents/_CodeSignature
myframeworktest.app/Contents/_CodeSignature/CodeResources
```

4.9.3 pkg 文件格式

pkg分为pkg与mpkg，前者是针对单程序安装，后者是针对多程序安装，它包含一个或多个子包（sub package）。pkg本身又有两种格式：一种与Bundle一样，有着特定组织结构的目录，上一节生成的pkg的安装包就是这种格式；还有一种是xar格式的文件。下面分别对这两种格式的安装包进行分析。

首先看myframework_installer.mpkg，使用tree命令（系统默认没有此命令，可以使用brew install tree进行安装）查看它的目录结构，如下所示：

```
$ tree ./myframework_installer.mpkg/
./myframework_installer.mpkg/
├── Contents
│   ├── Packages
│   │   └── app.pkg
│   │       └── Contents
│   │           ├── Archive.bom
│   │           ├── Archive.pax.gz
│   │           ├── Info.plist
│   │           ├── PkgInfo
│   │           └── Resources
│   │               ├── en.lproj
│   │               │   └── Description.plist
│   │               ├── package_version
│   │               ├── postflight
│   │               └── preflight
│   ├── Resources
│   │   └── en.lproj
│   └── distribution.dist
```

8 directories, 9 files

对于外层的mpkg，它的Packages目录下存放的是pkg文件列表，也就是子包列表；Resources

目录存放了pkg用到的资源，如本地化资源、图像、rtf文档、pdf文档等；distribution.dist文件是一个

xml文档，包含了要安装的子包、运行时脚本等信息。对于当前的mpkg，它的内容如下所示：

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<installer-script minSpecVersion="1.000000" authoringTool="com.apple.PackageMaker"
authoringToolVersion="3.0.6" authoringToolBuild="201">
  <title>myframework installer</title>
  <options customize="never" allow-external-scripts="no" rootVolumeOnly="false" />
  <installation-check script="pm_install_check();" />
  <volume-check script="pm_volume_check();" />
  <script>
function pm_volume_check() {
  if(! (my.target.systemVersion & & /* &gt;= */
    system.compareVersions(my.target.systemVersion.ProductVersion, '10.6') &gt;= 0)) {
    my.result.title = 'Failure';
    my.result.message = 'Installation cannot proceed, as not all requirements were met.';
    my.result.type = 'Fatal';
    return false;
  }
  return true;
}

function pm_install_check() {
  if(! (/* &gt;= */ system.compareVersions(system.version.ProductVersion, '10.6') &gt;= 0)) {
    my.result.title = 'Failure';
    my.result.message = 'Installation cannot proceed, as not all requirements were met.';
    my.result.type = 'Fatal';
    return false;
  }
  return true;
}
  </script>
  <choices-outline>
    <line choice="choice0" />
  </choices-outline>
  <choice id="choice0" title="app">
    <pkg-ref id="com.macbook.myframeworkInstaller.pkg" />
  </choice>
  <pkg-ref id="com.macbook.myframeworkInstaller.pkg" installKBytes="108" version="1.0"
    auth="Root">file:./Contents/Packages/app.pkg</pkg-ref>
</installer-script>
```

pm_install_check()与pm_volume_check()分别做安装时检查与卷标检查，下面的choices- outline

部分指定了安装时使用的choice，也就是选择执行哪个子包。对于当前的mpkg包，它只有一个pkg，

choice的id为choice0，指向的路径是file:./Contents/Packages/app.pkg。

app.pkg是要安装的子包，是一个pkg格式的Bundle结构的目录，它包含一个Contents子目录，里面有4个文件与1个目录Resources，下面分别进行介绍。

❑ Archive.bom：bom信息。存放的是要安装写入的文件列表，可以使用lsbom -pf命令查看，效果与上一节讲到的一样。

❑ Archive.pax.gz：使用pax格式打包后再使用gzip压缩的压缩包，它的内容就是要安装的内容，此处就是myframeworktest.app程序。可以执行以下命令进行解压：

```
$ cd ./myframework_installer.mpkg/Contents/Packages/app.pkg/Contents/
$ gunzip -d ./Archive.pax.gz
$ pax -rvf ./Archive.pax
.
./myframeworktest.app
./myframeworktest.app/Contents
./myframeworktest.app/Contents/_CodeSignature
./myframeworktest.app/Contents/_CodeSignature/CodeResources
./myframeworktest.app/Contents/Frameworks
./myframeworktest.app/Contents/Frameworks/myframework.framework
./myframeworktest.app/Contents/Frameworks/myframework.framework/myframework
./myframeworktest.app/Contents/Frameworks/myframework.framework/Resources
./myframeworktest.app/Contents/Frameworks/myframework.framework/Versions
./myframeworktest.app/Contents/Frameworks/myframework.framework/Versions/A
./myframeworktest.app/Contents/Frameworks/myframework.framework/Versions/A/_CodeSignature
./myframeworktest.app/Contents/Frameworks/myframework.framework/Versions/A/_CodeSignature
/CodeResources
./myframeworktest.app/Contents/Frameworks/myframework.framework/Versions/A/myframework
./myframeworktest.app/Contents/Frameworks/myframework.framework/Versions/A/Resources
./myframeworktest.app/Contents/Frameworks/myframework.framework/Versions/A/Resources/Info.plist
./myframeworktest.app/Contents/Frameworks/myframework.framework/Versions/Current
./myframeworktest.app/Contents/Info.plist
./myframeworktest.app/Contents/MacOS
./myframeworktest.app/Contents/MacOS/myframeworktest
./myframeworktest.app/Contents/PkgInfo
./myframeworktest.app/Contents/Resources
./myframeworktest.app/Contents/Resources/Base.lproj
./myframeworktest.app/Contents/Resources/Base.lproj/MainMenu.nib
```

❑ Info.plist：pkg包的信息。CFBundleIdentifier为pkg的标识；IFMajorVersion与IFMinorVersion

分别为pkg的主版本与子版本号；IFPkgFlagInstalledSize为pkg安装后所需要占用的字节大小。

- ❑ PkgInfo：8字节的标识。表明是一个pkg文件。

Resources目录除了包含资源文件外，还包含了以下文件。

- ❑ package_version：包版本文件。即使用PackageManager制作pkg时设置的版本号。
- ❑ postflight/preflight：pkg要执行的脚本文件。上一节中讲过，此处是未经过加密明文存放的。
- ❑ 另外一种是xar格式的文件，可以使用如下命令查看myframework_installer.pkg文件的格式：

```
$ file ./myframework_installer.pkg
./myframework_installer.pkg: xar archive - version 1
```

- ❑ xar是压缩的可扩展归档格式，可以使用xar命令对其进行解压，如下所示：

```
$ xar -xvf ./myframework_installer.pkg
Distribution
app.pkg/PackageInfo
app.pkg/Bom
app.pkg/Payload
app.pkg/Scripts
app.pkg
Resources/en.lproj
Resources
```

Distribution与前面讨论的distribution.dist文件基本一样，Resources目录也与前面的一样，下面我们主要看app.pkg，它包含以下4个文件。

- ❑ Bom：bom信息，存放要安装写入的文件列表。可以使用lsbom -pf命令查看，效果与上一节讲到的一样。

□ PackageInfo：文本文件，包含了包的信息。可以使用cat命令查看它的内容：

```
$ cat app.pkg/PackagelInfo
```

```
<pkg-info format-version="2" identifier="com.macbook.myframeworkInstaller.pkg" version="1.0"
install-location="/Applications" auth="root">
  <payload installKBytes="108" numberOfFiles="25"/>
  <scripts>
    <preinstall file="./preflight"/>
    <postinstall file="./postflight"/>
  </scripts>
  <bundle id="fc.myframeworktest" CFBundleIdentifier="fc.myframeworktest"
    path="./myframeworktest.app" CFBundleVersion="1">
    <bundle id="fc.myframework" CFBundleIdentifier="fc.myframework"
      path="./Contents/Frameworks/myframework.framework" CFBundleVersion="1"/>
  </bundle>
  <bundle-version>
    <bundle id="fc.myframeworktest"/>
    <bundle id="fc.myframework"/>
  </bundle-version>
```

□ Payload：经过gzip压缩过的数据内容，本处为要安装的myframework.app，可以使用如下

命令进行解压。解压成功后会在当前目录下生成myframework.app。

```
$ cat ./Payload | cpio -i
3 blocks
```

□ Scripts：经过gzip压缩的脚本。可以使用如下命令进行解压。解压成功后会在当前目录下

生成未加密的preflight与postflight脚本。

```
$ cd app.pkg
$ cat ./Scripts | cpio -i
181 blocks
```

4.9.4 破解 pkg

对pkg格式有一定了解后，修改或破解pkg就不难了。破解pkg无非有以下3种情况。

□ 资源的替换或修改：针对文件夹类型的pkg，未加密，可直接进行修改替换；针对xar类型

的pkg，需要先解压xar，然后替换或修改完资源后，重新压缩xar。

- 安装脚本的替换或修改：针对文件夹类型的pkg，未加密，可直接进行修改替换；针对xar类型的pkg，需要先解压xar，再解压Scripts，然后替换或修改完脚本后，重新压缩Scripts，最后重新压缩xar。
- 安装内容的替换或修改：针对文件夹类型的pkg，未加密，但需要先对Archive.pax.gz进行解包，修改完后，需要重新打包回去；针对xar类型的pkg，需要先解压xar，然后解压Payload，替换或修改完数据后，重新压缩Payload，最后重新压缩xar。

以上步骤是操作思路，在实际分析过程中，使用工具来做一些辅助工作可以大大提高效率。在拿到pkg后，首先快速浏览pkg文件，简单分析出pkg的行为与可能要做的操作。推荐一款工具：Suspicious Package^①，此工具提供了快速浏览插件，安装完成后，在要操作的pkg上按下空格，可以快速查看pkg，检索要安装的软件内容，如图4-22所示。

^① Suspicious Package 下载地址：<http://www.mothersruin.com/software/SuspiciousPackage>。

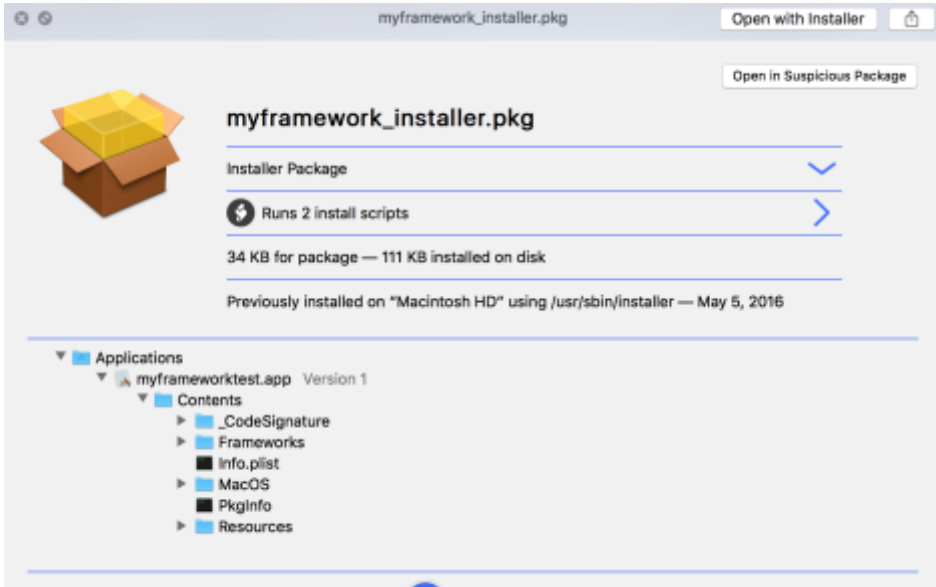


图4-22 查看pkg

还可以查看要执行的脚本内容，如图4-23所示。

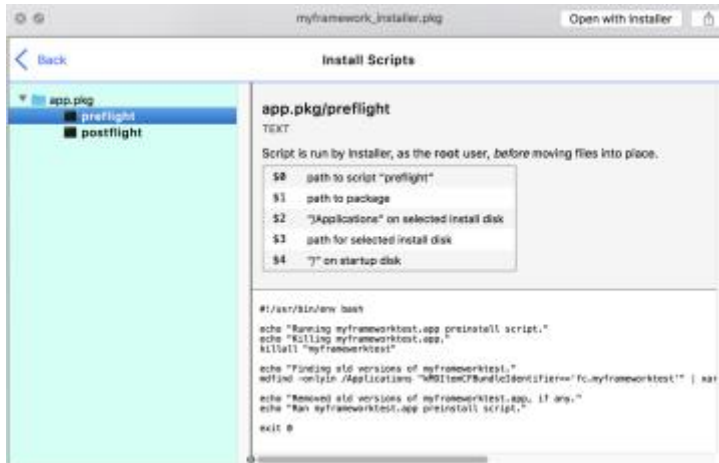


图4-23 查看要执行的脚本内容

对pkg有了初步了解，找到需要操作的地方后，下一步就是提取数据内容了。Suspicious

Package支持数据提取，使用Suspicious Package打开pkg文件后，在主界面的All Files列就可以查看所有文件，可以选中要导出的文件，直接拖出到Finder，或者点击Action→Export，都可以将文件导出，操作效果如图4-24所示。

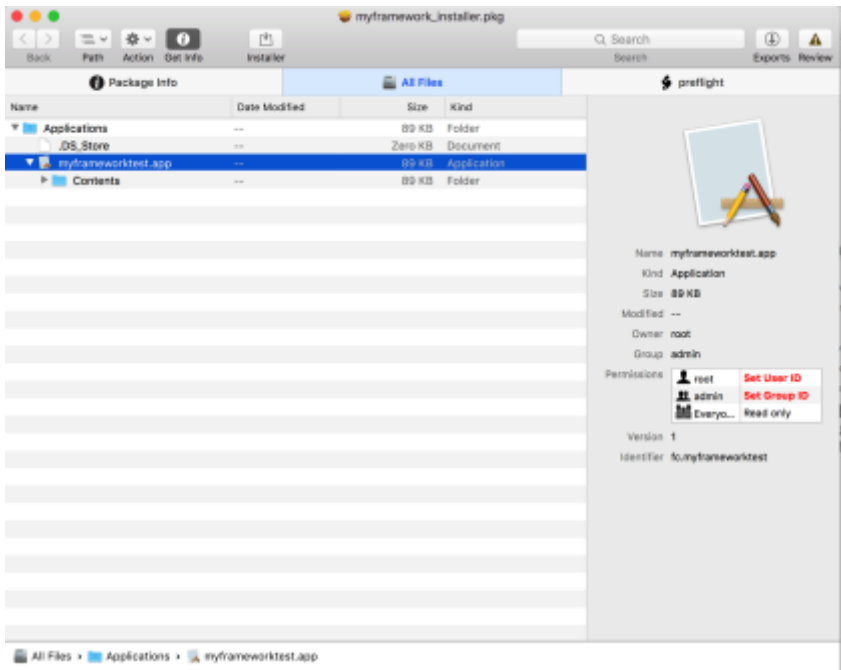


图4-24 文件导出

除了Suspicious Package，还有另外一款更强大的工具：Pacifist^①，这款工具支持多种文件数据的提取，其中就包括pkg，如图4-25所示。

^① Pacifist下载地址：<http://www.charlessoft.com>。



图4-25 Pacifist

选中要提取的文件，右键选择“Extract to Custom Location...”，或者直接拖到要保存的文件夹中，都可以将文件提取出来。

将提取出来的文件，分析完成并修改好了后，就要打包回去了。Pacifist不支持将数据打包回去，如果修改的是Payload，可以使用如下命令将app目录下的myframeworktest.app打包回去：

```
find app/* | cpio -o > ./Payload
```

如果是脚本文件，也可以如法炮制。最后就是将修改好的Payload或Scripts重新打包回去，可以执行xar cvf命令来操作，这里推荐另一款图形化工具：Flat Package Editor，该工具是苹果官方

提供的，可以对pkg直接进行增、删、改操作。打开要操作的pkg，将修改好的Payload或Scripts拖回去，然后点击菜单File→Save就保存成功了，如图4-26所示。

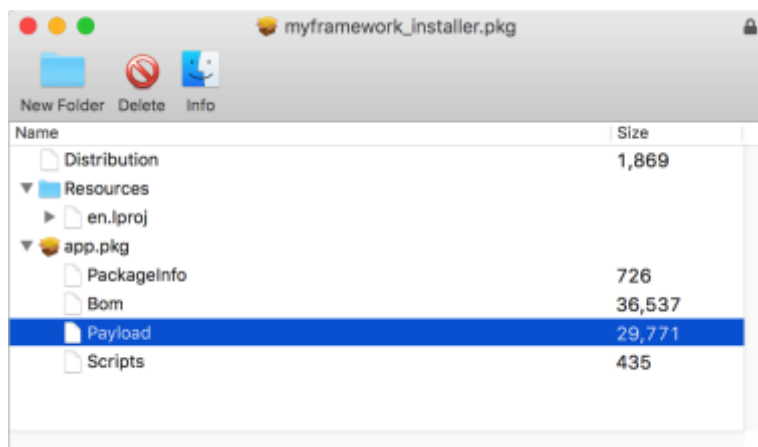


图4-26 使用Flat Package Editor进行打包

操作完成后，pkg就算修改好了，接下来测试安装没问题就算破解完成了。

4.10 dmɡ

dmɡ是苹果电脑上专用的磁盘镜像（disk image）文件，类似于Windows平台上的iso镜像。dmɡ类似于一个压缩文档，支持压缩与加密，将程序与文档打包成dmɡ是一种比较流行的软件发布形式。

4.10.1 构建 dmɡ

苹果官方系统自带的磁盘管理工具Disk Utility可以很方便地构建dmɡ文件，最简单的方法就是启动/Applications/Utilities/Disk Utility，点击菜单File→New Image→Image From Folder...，从文件夹创建镜像。选择上一节的app目录，如图4-27所示。

在Save As处输入要保存的文件名，在Encryption处选择是否进行加密，none表示不加密，128-bit AES encryption是macOS 10.3以前之前支持的128位的AES加密，256-bit AES encryption则是macOS 10.5以后才开始支持的256位AES加密。在选择任意一种加密方式后，会弹出输入密码的对话框，提示输入的密码不是AES算法的加密key，只是一个用户自己设置的密码。设置好密码后，在Image Format处设置镜像的格式，read-only表示创建只读的镜像，compressed表示对镜像进行压缩，read/write表示镜像可读可写，DVD/CD master表示创建DVD镜像，hybrid image表示创建混合镜像。选择好选项后，点击Save按钮，dmg就创建成功了。

除了使用图形界面创建dmg外，还可以使用命令行工具hdiutil来创建，例如为app目录下的myframeworktest.app创建一个AES128加密、密码为abc123的dmg镜像，只需要执行如下命令即可：

```
$ hdiutil create -fs HFS+ -volname myframework -srcfolder ../pkg_install_script/app -encryption AES-128 -stdinpass  
-o myframeworktest_cmd.dmg  
Enter disk image passphrase: //此处输入密码123  
..  
created: /Users/macbook/code/chapter4/dmg/myframeworktest_cmd.dmg
```

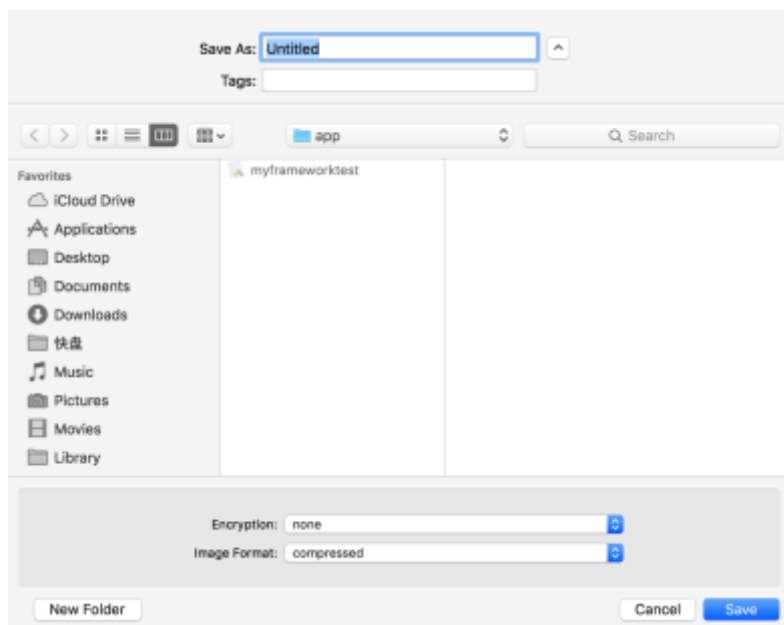


图4-27 选择app目录

如果觉得从文件夹中创建的dmg不够个性化，完全可以使用Disk Utility创建自定义的dmg，自定义的dmg包括为dmg指定图标、背景图片以及dmg文件的显示方式及大小。只需要打开Disk Utility，点击菜单File→New Image→Blank Image...，创建一个空白的镜像，在保存对话框中设置镜像的大小、加密方式、分区格式及镜像格式。需要注意的是，此处镜像格式需要选择read/write disk image。创建成功后，打开镜像，将app目录中的文件复制进去。如果需要更换背景图片，只需要将背景图片复制到镜像中，使用chflags命令设置成隐藏格式，或者放一个点“.”结尾的目录（默认会隐藏显示）。在镜像上点击右键，在弹出的菜单中选择Get Info，然后设置背景图片即可。操作完后，点击Disk Utility菜单的Images→Convert...，选择操作后的dmg镜像，将该镜像压缩保存就可以发布了。

除了使用官网的Disk Utility外，也可以使用上一节介绍的Luggage工具。编译脚本后，执行make dmg来生成dmg文件。最后，还有一些第三方工具也可以用来创建dmg镜像，比较知名的有DropDMG^①，从软件的名称上就可以判断，它支持通过文件拖放来快速创建dmg镜像。有兴趣的读者可以试试，使用比较简单，此处不再赘述。

4.10.2 管理 dmg

dmg文件格式不是开放的，要想探索它的文件格式，可以使用逆向工具hidutil来处理dmg的部分代码。在使用dmg的过程中，一种可能遇到的典型场景是将dmg转换格式后，在Windows或Linux平台上使用。针对早期版本的dmg，网上有第三方工具dmg2img^②，可以很方便地将dmg转换成可以在Linux系统上挂载的镜像。还有一个工具dmg2iso^③，可以将dmg转换成Windows平台上使用的iso镜像。实际上该工具的底层是调用了hdiutil。

其实，使用hdiutil来管理dmg已经足够了。它提供了查看、创建、转换dmg等功能。例如，查看myframeworktest.dmg的信息可以执行如下命令：

```
$ hdiutil imageinfo myframeworktest.dmg
Format Description: UDIF read-only compressed (zlib)
Class Name: CUDIFDiskImage
Checksum Type: CRC32
Size Information:
  Compressed Ratio: 0.022532451628704386
  Total Empty Bytes: 500224
  Sector Count: 5060
  Total Bytes: 2590720
  CUDIFEncoding-bytes-wasted: 7963
  Total Non-Empty Bytes: 2090496
```

① DropDMG下载地址：<http://c-command.com/dropdmg>。

② dmg2img下载地址：<https://github.com/Lekensteyn/dmg2img>。

③ dmg2iso下载地址：<https://sourceforge.net/projects/dmg2iso>。

```

    CUDIFEncoding-bytes-in-use: 47410
    Compressed Bytes: 47410
    CUDIFEncoding-bytes-total: 55373
Checksum Value: $404B6F25
Segments:
.....
-1:
    Name: Protective Master Boot Record (MBR : 0)
    Partition Number: -1
    Checksum Type: CRC32
    Checksum Value: $0492F534
2:
    Name: (Apple_Free : 3)
    Partition Number: 2
    Checksum Type: CRC32
    Checksum Value: $00000000
Format: UDZO
Backing Store Information:
.....
partitions:
    partition-scheme: GUID
    block-size: 512
    partitions:
        0:
            partition-name: Protective Master Boot Record
            partition-start: 0
            partition-synthesized: true
            partition-length: 1
            partition-hint: MBR
        .....
        7:
            partition-name: GPT Header
            partition-start: 5059
            partition-synthesized: true
            partition-length: 1
            partition-hint: Backup GPT Header
    burnable: false
    udif-ordered-chunks: false
Properties:
    Encrypted: false
    Kernel Compatible: true
    Checksummed: true
    Software License Agreement: false
    Partitioned: false
    Compressed: true
Resize limits (per hdiutil resize -limits):
    min    cur    max
5060    5060    5060

```

将myframeworktest_cmd.dmg的密码abc123更改为123abc只需执行如下命令：

```

$ hdiutil chpass ./myframeworktest_cmd.dmg
Enter password to access "myframeworktest_cmd.dmg":      //abc123
Enter a new password to secure "myframeworktest_cmd.dmg": //123abc

```


Re-enter new password: //123abc

将myframeworktest.dmg转换成iso格式可以执行如下命令：

```
$ hdiutil convert ./myframeworktest.dmg -format UDTO -o ./myframeworktest.cdr
Reading Protective Master Boot Record (MBR : 0)...
Reading GPT Header (Primary GPT Header : 1)...
Reading GPT Partition Data (Primary GPT Table : 2)...
Reading (Apple_Free : 3)...
Reading disk image (Apple_HFS : 4)...
.....
Reading (Apple_Free : 5)...
Reading GPT Partition Data (Backup GPT Table : 6)...
.....
Reading GPT Header (Backup GPT Header : 7)...
.....
Elapsed Time: 7.868ms
Speed: 314.0Mbytes/sec
Savings: 0.0%
created: /Users/.../code/chapter4/dmg/myframeworktest.cdr
$ mv ./myframeworktest.cdr ./myframeworktest.iso
```

另外，DropDMG也提供了方便的dmg管理功能。例如，在文件夹上点击右键，在弹出的菜单中选择Services→DropDMG:Use Current Configuration，DropDMG就会使用当前默认的配置为文件夹在当前目录创建一个dmg。或者可以在dmg上点击右键，选择DropDMG:Ask for Options来对dmg做一些修改，例如设置图标、修改密码、更改格式等。

4.11 本章小结

macOS系统中包含了形形色色的文件，了解这些文件的内幕对于探索系统的底层运作方式有着重要作用。本章主要探讨了macOS系统中常见的文件格式，并详细讲解了Mach-O的文件格式，分析了dyld加载dylib动态库过程，最后分析讲解了软件安装包pkg与镜像dmg文件。在讲解的过程中，还直接或间接地介绍了大量的第三方工具，这些工具对于分析文件格式是极其有用的，熟练地掌握它们的使用方法，可以有效地提高学习文件格式的效率。

最后，本章中讨论的文件格式只是macOS系统中极少的一部分，限于篇幅，还有很多文件格式没有涉及，比如系统Quicklook插件、Service插件、Internet插件、Xcode插件、内核kext扩展、屏幕保护程序、系统面板等，希望有兴趣的读者可以自行探索。