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

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSLog(@"Hello, World!");
    }
    return 0;
}
```

> iOS开发中，main函数是我们熟知的程序启动入口，但实际上并非真正意义上的入口，因为在我们运行程序，再到main方法被调用之间，程序已经做了许许多多的事情，比如我们熟知的runtime的初始化就发生在main函数调用前，还有程序动态库的加载链接也发生在这阶段，本文主要对从程序启动到main函数中发生的主要事情进行简单介绍。

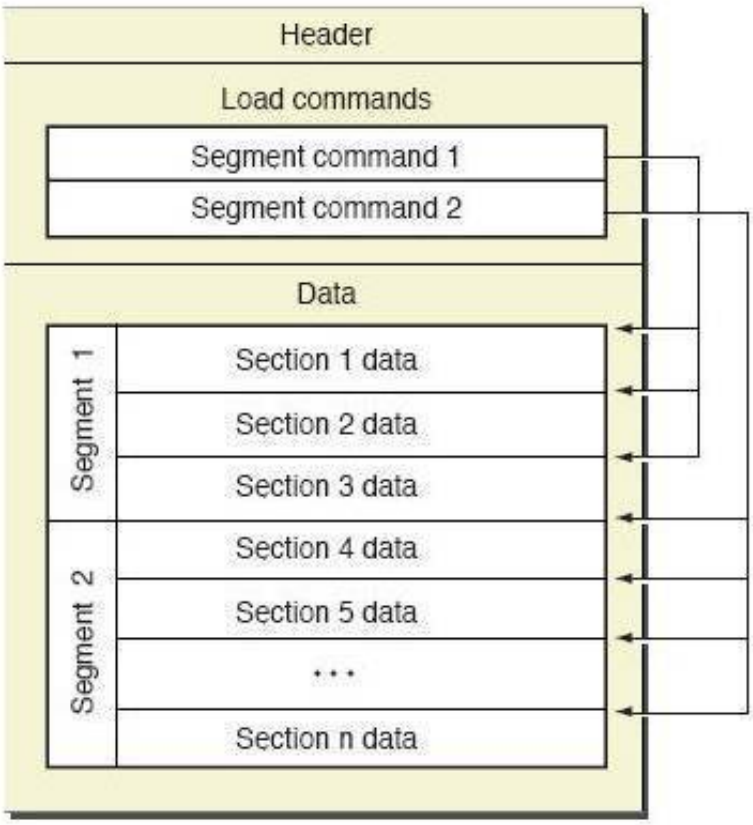
其实简单总结起来就是：

系统先读取App的可执行文件（Mach-O文件），从里面获得dyld的路径，然后加载dyld，dyld去初始化运行环境，开启缓存策略，加载程序相关依赖库(其中也包含我们的可执行文件)，并对这些库进行链接，最后调用每个依赖库的初始化方法，在这一步，runtime被初始化。当所有依赖库的初始化后，轮到最后一位(程序可执行文件)进行初始化，在这时runtime会对项目中所有类进行类结构初始化，然后调用所有的load方法。最后dyld返回main函数地址，main函数被调用，我们便来到了熟悉的程序入口。

下面我们将结合代码对整个过程进行分析：

dyld加载

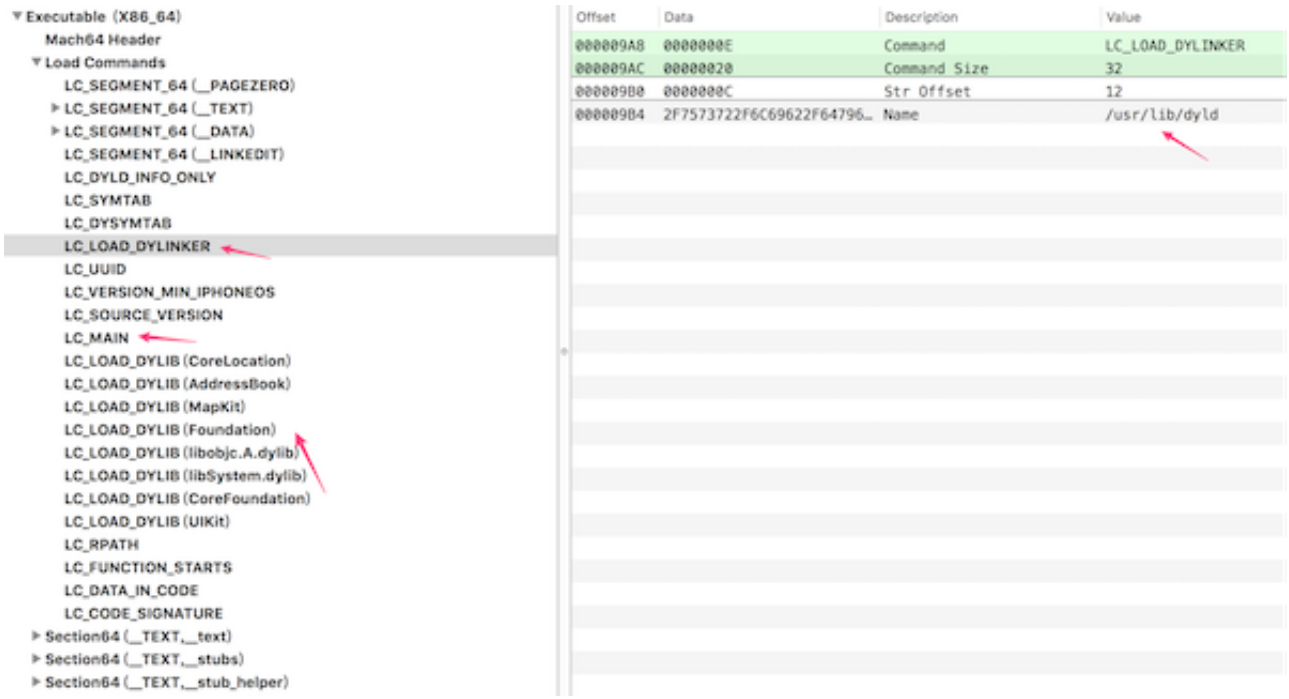
这里先说下Mach-O文件。
Mach-O文件格式是 OS X 与 iOS 系统上的可执行文件格式，像我们编译过程产生的.O文件，以及程序的可执行文件，动态库等都是Mach-O文件。它的结构如下:



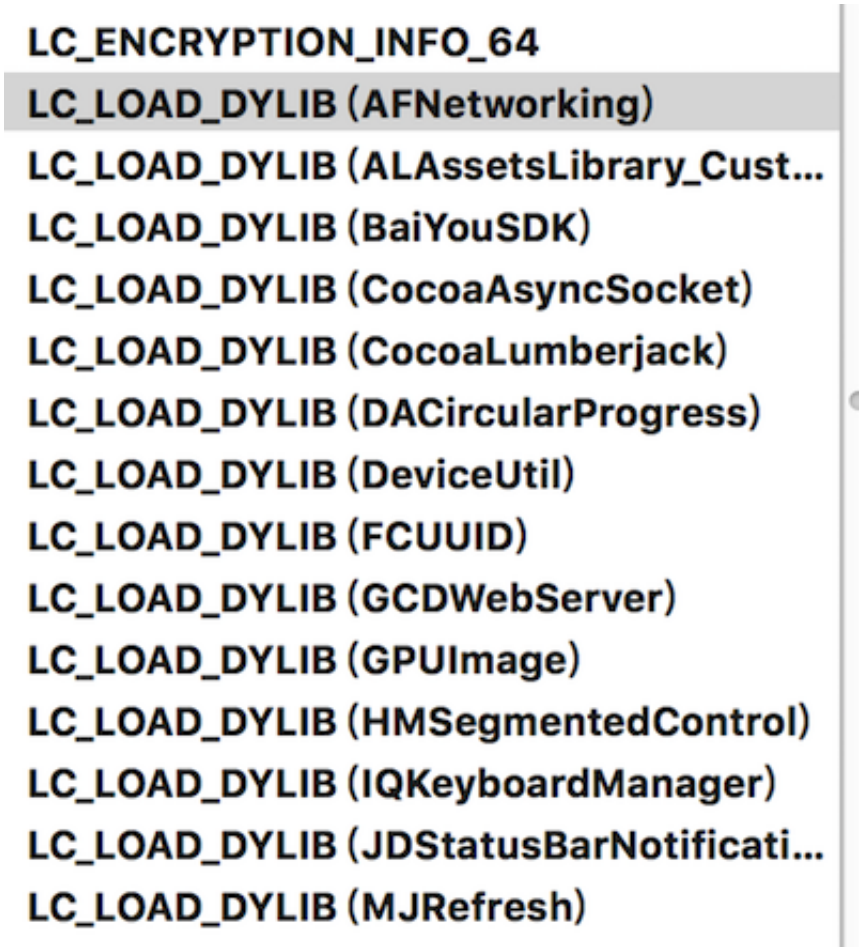
有如下几个部分组成：

1. Header：保存了一些基本信息，包括了该文件运行的平台、文件类型、LoadCommands的个数等等。
2. LoadCommands：可以理解为加载命令，在加载Mach-O文件时会使用这里的数据来确定内存的分布以及相关的加载命令。比如我们的main函数的加载地址，程序所需的dyld的文件路径，以及相关依赖库的文件路径。
3. Data： 这里包含了具体的代码、数据等等。

我们可以通过Mach-O文件查看器MachOView查看一个测试项目(这里放上地址)编译后的可执行文件内容:



这里可以看到，程序需要的dyld的路径在*LC_LOAD_DYLINKER*命令里，一般都是在/usr/lib/dyld路径下。这里的*LC_MAIN*指的是程序main函数加载地址，下面还有写*LC_LOAD_DYLIB*指向的都是程序依赖库加载信息，如果我们程序里使用到了AFNetworking，这里就会多一条名为*LC_LOAD_DYLIB(AFNetworking)*的命令，如下图



这里可以看到一些我们比较常用的三方库：AFNetworking,IQKeyboard等。

系统加载程序可执行文件后，通过分析文件来获得dyld所在路径来加载dyld，然后就将后面的事情甩给dyld了。

从dyld开始

dyld: (the dynamic link editor)动态链接器，其源码是开源的。

ImageLoader: 用于辅助加载特定可执行文件格式的类，程序中对应实例可简称为image(如程序可执行文件，Framework库，bundle文件)。

dyld接手后得做很多事情，主要负责初始化程序环境，将可执行文件以及相应的依赖库与插入库加载进内存生成对应的ImageLoader类的image(镜像文件)对象，对这些image进行链接，调用各image的初始化方法等等(注:这里多数事情都是递归的，从底向上的方法调用)，其中runtime也是在这个过程中被初始化，这些事情大多数在dyld:_mian方法中被发生，我们可以看段简洁的代码：

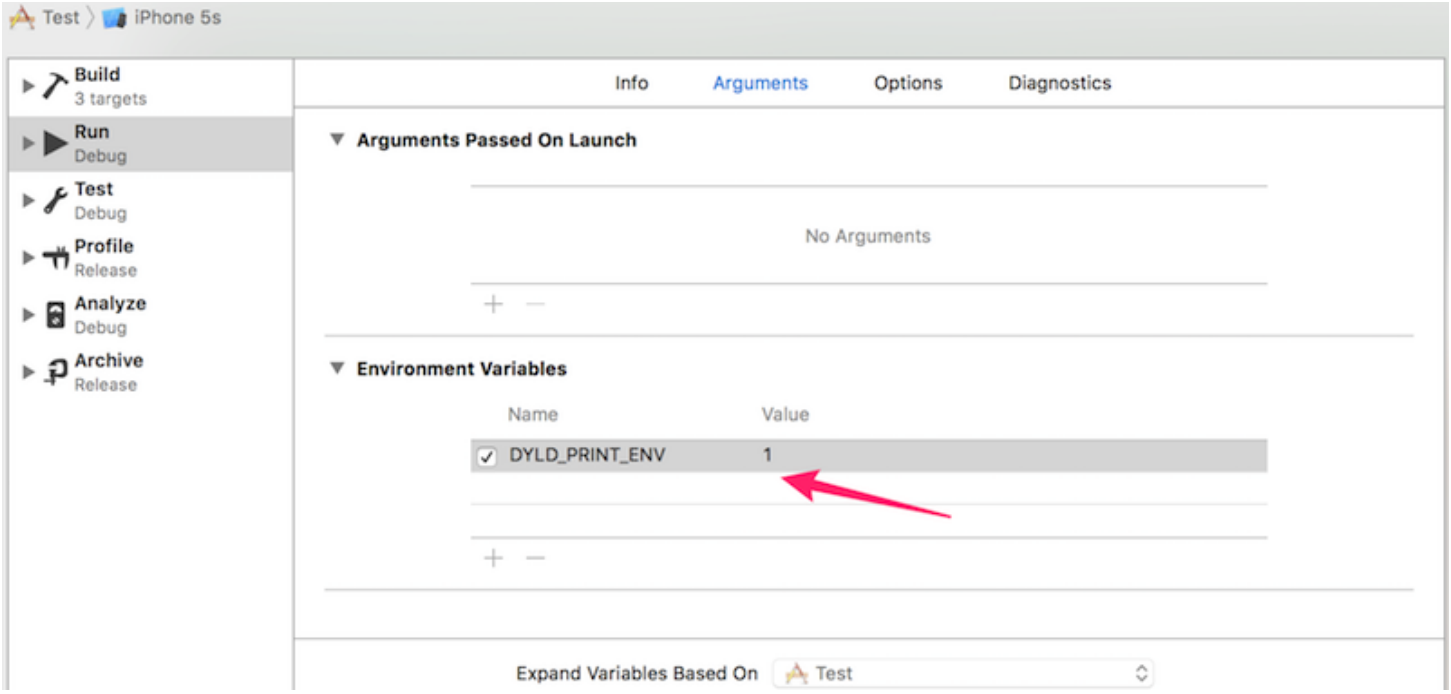
```
uintptr_t
_main(const macho_header* mainExecutableMH, uintptr_t mainExecutableSlide,
        int argc, const char* argv[], const char* envp[], const char* apple[],
        uintptr_t* startGlue)
{
        ........
        // instantiate ImageLoader for main executable
        sMainExecutable = instantiateFromLoadedImage(mainExecutableMH, mainExecutableSlide, sExecPath);
        ........
        // load any inserted libraries
        if  ( sEnv.DYLD_INSERT_LIBRARIES != NULL ) {
            for (const char* const* lib = sEnv.DYLD_INSERT_LIBRARIES; *lib != NULL; ++lib)
                loadInsertedDylib(*lib);
        }
        ........
        link(sMainExecutable, sEnv.DYLD_BIND_AT_LAUNCH, true, ImageLoader::RPathChain(NULL, NULL));
        ........
        // link any inserted libraries
        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));
                ........
            }
            ........
        }
        ........
        // run all initializers
        initializeMainExecutable(); Å
        ........
    return result;
}
```

这里的_main函数是dyld的函数，并非我们程序里的main函数。

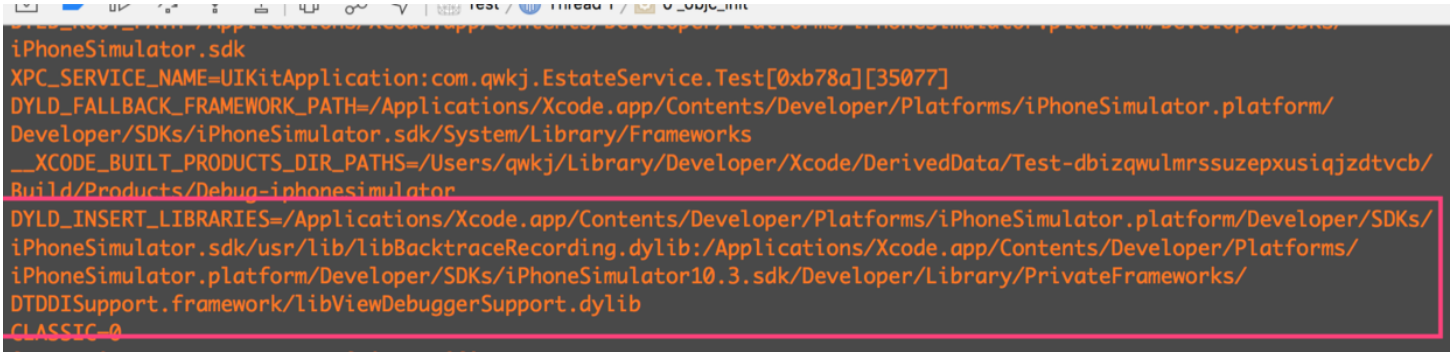
1.sMainExecutable = instantiateFromLoadedImage(....)与loadInsertedDylib(...)

这一步dyld将我们可执行文件以及插入的lib加载进内存,生成对应的image。
sMainExecutable对应着我们的可执行文件，里面包含了我们项目中所有新建的类。

InsertDylib一些插入的库，他们配置在全局的环境变量*sEnv*中，我们可以在项目中设置环境变量*DYLD_PRINT_ENV*为1来打印该*sEnv*的值。



运行程序Log如下：



可以看到插入的库为:*libBacktraceRecording.dylib*和*libViewDebuggerSupport*.
有时我们会在三方App的Mach-O文件中通过修改DYLD_INSERT_LIBRARIES的值来加入我们自己的动态库，从而注入代码，hook别人的App(相关资料)。

2.link（sMainExecutable,...)和link(image，....)
对上面生成的Image进行进行链接。其主要做的事有对image进行load(加载),rebase(基地址复位)，bind(外部符号绑定)，我们可以查看源码:

```
void ImageLoader::link(const LinkContext& context, bool forceLazysBound, bool preflightOnly, bool neverUnload, const RPathChain&
    loaderRPaths)
{
    //dyld::log("ImageLoader::link(%s) refCount=%d, neverUnload=%d\n", this->getPath(), fDlopenReferenceCount, fNeverUnload);

    ......

    this->recursiveLoadLibraries(context, preflightOnly, loaderRPaths);
    ......

    this->recursiveRebase(context);
    ......

    this->recursiveBind(context, forceLazysBound, neverUnload);
    ......
}
```

- recursiveLoadLibraries(context, preflightOnly, loaderRPaths)
  递归加载所有依赖库进内存。

- recursiveRebase(context)
  递归对自己以及依赖库进行复基位操作。在以前，程序每次加载其在内存中的堆栈基地址都是一样的，这意味着你的方法，变量等地址每次都一样的，这使得程序很不安全，后面就出现ASLR（Address space layout randomization,地址空间配置随机加载），程序每次启动后地址都会随机变化，这样程序里所有的代码地址都是错的，需要重新对代码地址进行计算修复才能正常访问。

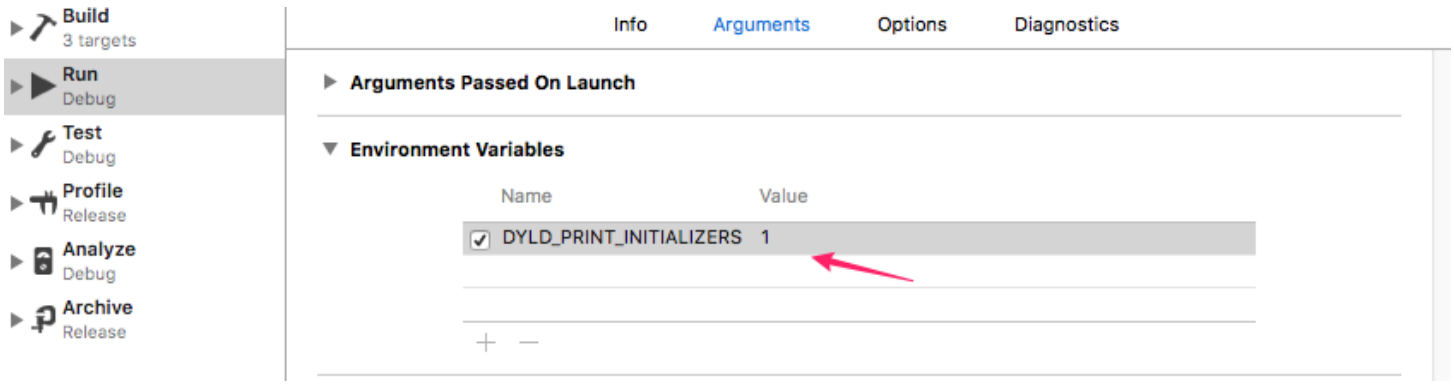
- recursiveBind(context, forceLazysBound, neverUnload)
  对库中所有nolazy的符号进行bind,一般的情况下多数符号都是lazybind的，他们在第一次使用的时候才进行bind。

3.initializeMainExecutable()

这一步主要是调用所有image的Initalizer方法进行初始化。这里的Initalizers方法并非名为Initalizers的方法，而是C++静态对象初始化构造器，__atribute__((constructor))进行修饰的方法，在LmageLoader类中initializer函数指针所指向该初始化方法的地址。

```
typedef void (*Initializer)(int argc, const char* argv[], const char* envp[], const char* apple[],
    const ProgramVars* vars);
typedef void (*Terminator)(void);
```

我们可以在程序中设置环境变量*DYLD_PRINT_INITIALIZERS*为1来打印出程序的各种依赖库的initializer方法：



运行程序，系统Log打印如下:

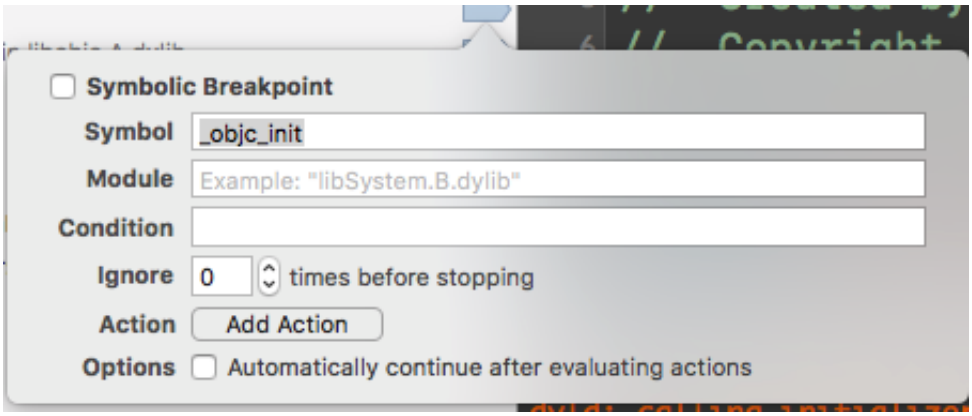
```
dyld: calling initializer function 0x104086a2d in /Applications/Xcode.app/Contents/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator.sdk/usr/lib/
libSystem.dylib
dyld: calling -init function 0x103159a95 in /Applications/Xcode.app/Contents/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator.sdk/usr/lib/
libBacktraceRecording.dylib
dyld: calling initializer function 0x106d2c325 in /Applications/Xcode.app/Contents/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator.sdk/usr/lib/libc++.
1.dylib
dyld: calling -init function 0x10408d2f0 in /Applications/Xcode.app/Contents/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator.sdk/System/Library/
Frameworks/CoreFoundation.framework/CoreFoundation
dyld: calling initializer function 0x1042223c0 in /Applications/Xcode.app/Contents/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator.sdk/System/Library/
Frameworks/CoreFoundation.framework/CoreFoundation
dyld: calling initializer function 0x1083438e6 in /Applications/Xcode.app/Contents/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator.sdk/usr/lib/
```

(由于打印的比较长，这样就截取开头的log)可以看到每个依赖库对应着一个初始化方法，名称各有不同。
这里最开始调用的libSystem.dylib的initializer function比较特殊，因为runtime初始化就在这一阶段，而这个方法其实很简单，我们可以在这里看到init.c源码，主要方法如下：

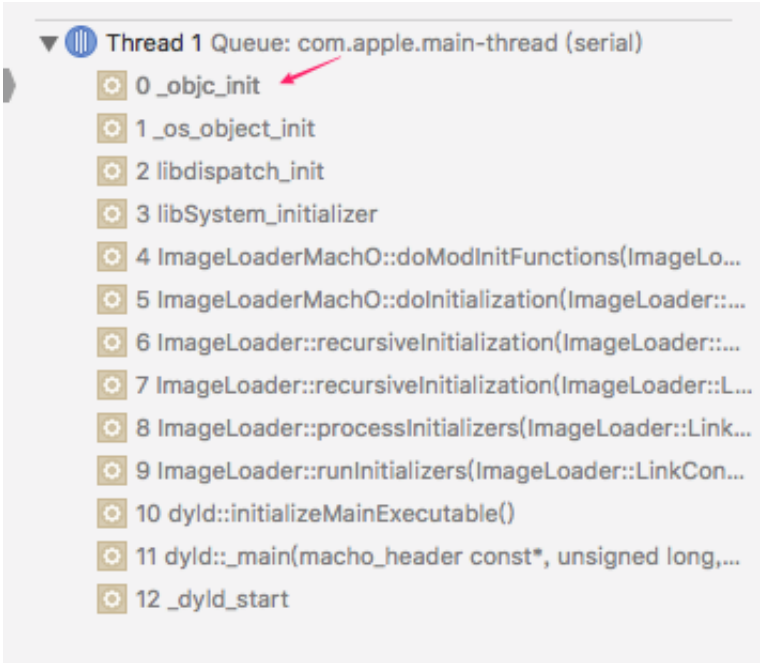
```
/*
 * libsyscall_initializer() initializes all of libSystem.dylib <rdar://problem/4892197>
 */
static __attribute__((constructor))
void libSystem_initializer(int argc, const char* argv[], const char* envp[], const char* apple[], const
    struct ProgramVars* vars)
{
    _libkernel_functions_t libkernel_funcs = {
        .get_reply_port = _mig_get_reply_port,
        .set_reply_port = _mig_set_reply_port,
        .get_errno = __error,
        .set_errno = cthread_set_errno_self,
        .dlsym = dlsym,
    };

    _libkernel_init(libkernel_funcs);

    bootstrap_init();
    mach_init();
    pthread_init();
    __libc_init(vars, libSystem_atfork_prepare, libSystem_atfork_parent, libSystem_atfork_child, apple);
    __keymgr_initializer();
    _dyld_initializer();
    libdispatch_init();
    _libxpc_initializer();

    __stack_logging_early_finished();

    /* <rdar://problem/11588042>
     * C99 standard has the following in section 7.5(3):
     * "The value of errno is zero at program startup, but is never set
     * to zero by any library function."
     */
    errno = 0;
}
```

其中libdispatch_init里调用了到了runtime初始化方法_objc_init.我们可以、在程序中打个符号断

点来验证:



运行程序，然后断点命中，我们来看下调用栈:



这里可以看到_objc_init调用的顺序，先libSystem_initializer调用libdispatch_init再到_objc_init初始化runtime。

runtime初始化后不会闲着，在_objc_init中注册了几个通知，从dyld这里接手了几个活，其中包括负责初始化相应依赖库里的类结构，调用依赖库里所有的laod方法。

就拿sMainExcuatable来说，它的initializer方法是最后调用的，当initializer方法被调用前dyld会通知runtime进行类结构初始化，然后再通知调用load方法，这些目前还发生在main函数前，但由于lazy bind机制，依赖库多数都是在使用时才进行bind，所以这些依赖库的类结构初始化都是发生在程序里第一次使用到该依赖库时才进行的。

main函数被调用

当所有的依赖库库的Initializer都调用完后，dyld::main函数会返回程序的main函数地址，main函

数被调用，从而代码来到了我们熟悉的程序入口。

```
11 int main(int argc, const char * argv[]) {
12     @autoreleasepool {
13         NSLog(@"Hello, World!");
14     }
15     return 0;
16 }
```

参考资料

1.Mach-O 可执行文件
2.dylib动态库加载过程分析
3.iOS 程序 main 函数之前发生了什么
4.今日头条iOS客户端启动速度优化
5.App 启动时间：过去，现在和未来
6.优化 App 的启动时间
7.dyld在hook方面的小东西

结语

这里只是简单了概括了从程序启动->dyld加载依赖库->runtime初始化->main 的过程。但这阶段还有很多事情未讲，如果想深入了解还得结合源码来学习，这里我已经将dyld和runtime源码都放在这了，大家可直接下载，也可以从opensource-apple下载。

再唠嗑会

dyld源码前前后后读个大概懂，花了我3个多礼拜的空闲时间，由于C和C++基础并不是很好，所以特意跑回学校买了几本书补了下基础，不过读源码的这段时间还是挺累的。

为什么要去读源码，主要是看别人的文章时并不能很好解决我的某些疑问，而且只有真正去认识源码，去亲身体会才能加深对它的理解。

学习的旅途虽然颇累，但一路下来收获颇多。加油！

前行路，路漫漫，一人一酒似逍遥。

喜欢的话点个喜欢呗