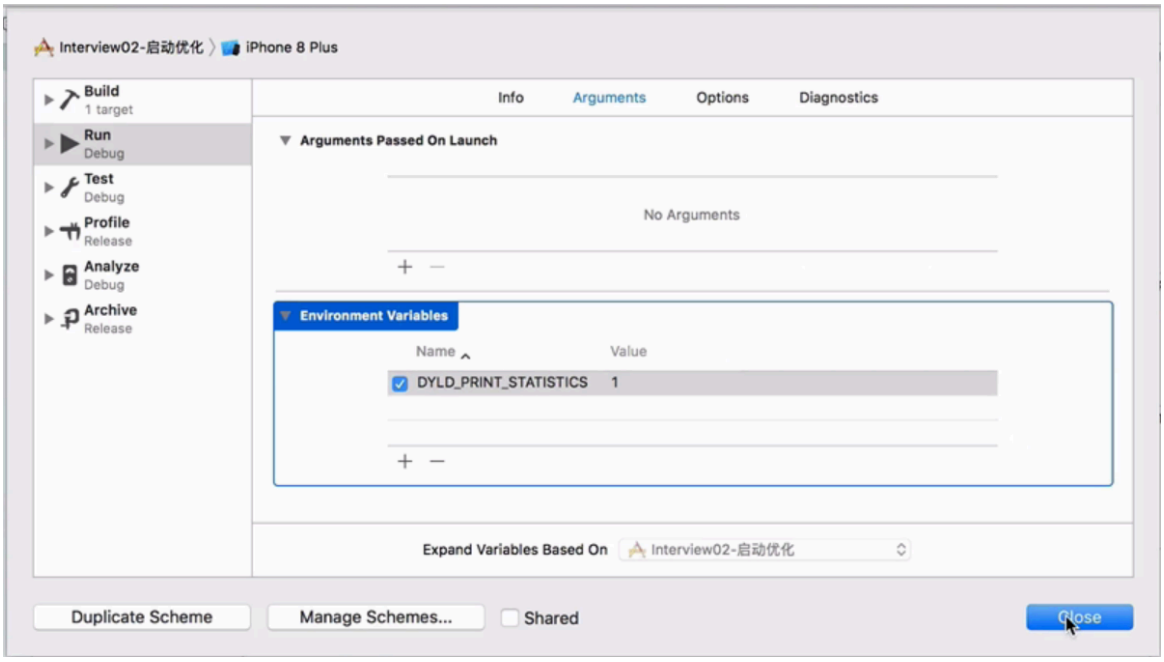
009-性能优化 启动时间

启动时间的概念

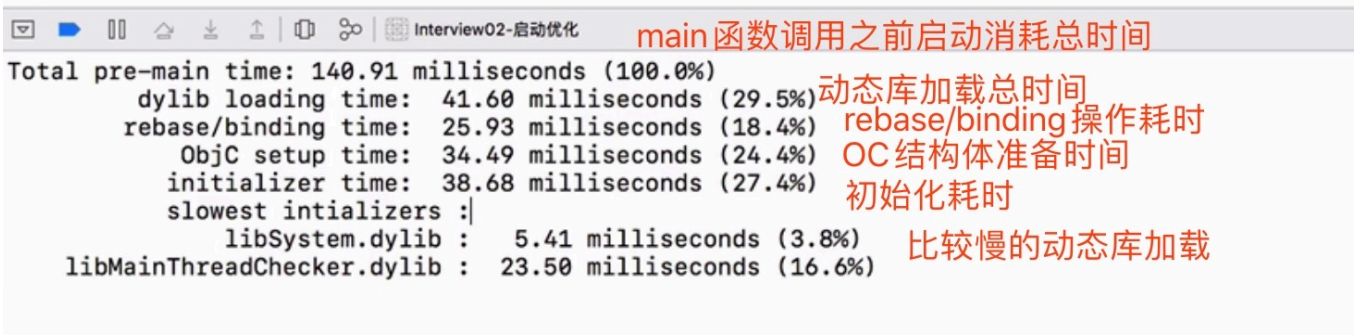
- APP的启动可以分为2种
  - 冷启动（Cold Launch）：从零开始启动APP
  - 热启动（Warm Launch）：APP已经在内存中，在后台存活着，再次点击图标启动APP
- APP启动时间的优化，主要是针对冷启动进行优化
- 通过添加环境变量可以打印出APP的启动时间分析（Edit scheme -> Run -> Arguments）
  - DYLD_PRINT_STATISTICS设置为1

xcode自带的启动时间打印

我们可以通过手动添加DYLD_PRINT_STATISTICS参数，在程序允许的时候打印程序的启动耗时。



我们通过添加DYLD_PRINT_STATISTICS参数，可以在下次启动时，获得以下打印信息



那么启动时间优化，也可以从这几个方面下手了：

- 动态库加载
- 方法重定向
- 结构体初始化

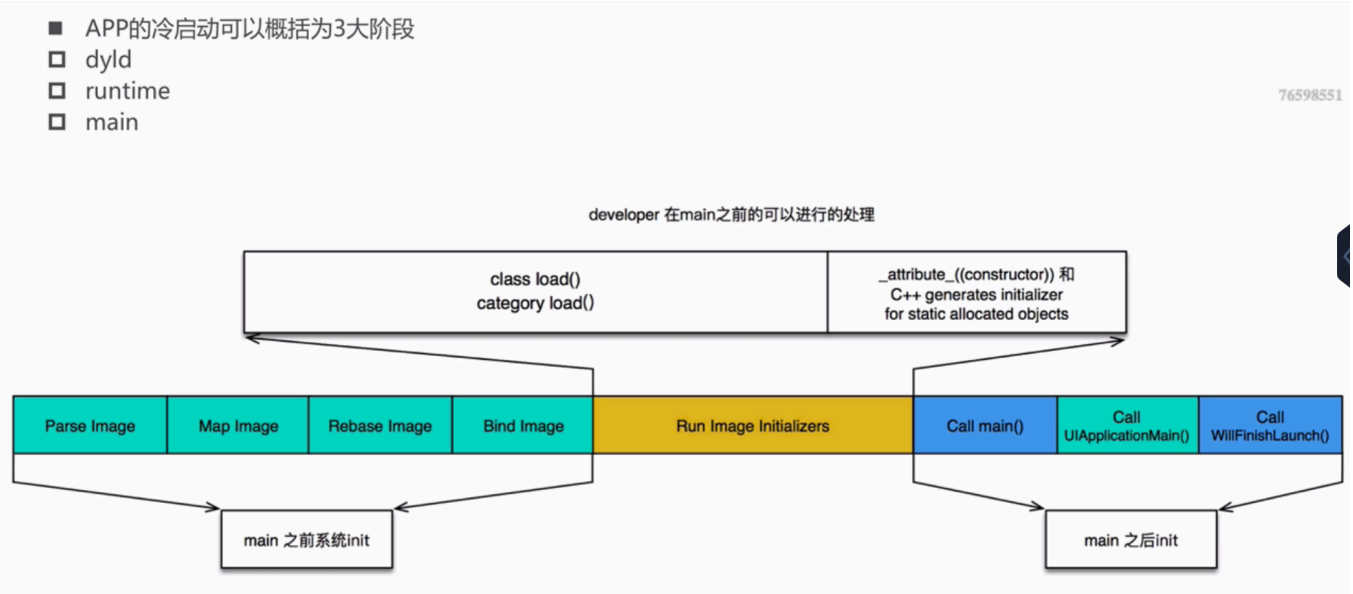
如果想获取更加相信的信息打印，可以添加DYLD_PRINT_STATISTICS_DETAILS参数，从而获得以下打印结果：

```
  total time: 306.30 milliseconds (100.0%)
  total images loaded:  218 (0 from dyld shared cache)
  total segments mapped: 646, into 92917 pages with 6012 pages pre-fetched
  total images loading time: 199.60 milliseconds (65.1%)
  total load time in ObjC:  32.32 milliseconds (10.5%)
  total debugger pause time: 160.41 milliseconds (52.3%)
  total dtrace DOF registration time:   0.18 milliseconds (0.0%)
  total rebase fixups:  2,114,436
  total rebase fixups time:  22.11 milliseconds (7.2%)
  total binding fixups: 244,873
  total binding fixups time:  19.66 milliseconds (6.4%)
  total weak binding fixups time:   0.41 milliseconds (0.1%)
  total redo shared cached bindings time:  19.48 milliseconds (6.3%)
  total bindings lazily fixed up: 0 of 0
  total time in initializers and ObjC +load:  31.99 milliseconds (10.4%)
                         libSystem.dylib :   2.24 milliseconds (0.7%)
              libBacktraceRecording.dylib :   2.43 milliseconds (0.7%)
                          CoreFoundation :   1.62 milliseconds (0.5%)
                              Foundation :   1.66 milliseconds (0.5%)
               libMainThreadChecker.dylib :  22.68 milliseconds (7.4%)
                   libLLVMContainer.dylib :   0.51 milliseconds (0.1%)
total symbol trie searches:    117378
total symbol table binary searches:    0
total images defining weak symbols:  25
total images using weak symbols:  63
```

一般启动时间应该保持在400ms以内。

冷启动的三个阶段

从上面的打印信息，我们可以将冷启动划分为以下三大阶段



- dyld阶段：加载可执行文件，加载动态库阶段
- runtime阶段：初始化OC结构（类，分类）

- main阶段：调用main函数

dyld阶段

- dyld（dynamic link editor），Apple的动态链接器，可以用来装载Mach-O文件（可执行文件、动态库等）

- 启动APP时，dyld所做的事情有
  - 装载APP的可执行文件，同时会递归加载所有依赖的动态库
  - 当dyld把可执行文件、动态库都装载完毕后，会通知Runtime进行下一步的处理

runtime阶段

- 启动APP时，runtime所做的事情有
  - 调用map_images进行可执行文件内容的解析和处理
  - 在load_images中调用call_load_methods，调用所有Class和Category的+load方法
  - 进行各种objc结构的初始化（注册Objc类 、初始化类对象等等）
  - 调用C++静态初始化器和__attribute__((constructor))修饰的函数

- 到此为止，可执行文件和动态库中所有的符号(Class，Protocol，Selector，IMP，...)都已经按格式成功加载到内存中，被runtime 所管理

main阶段

- 总结一下
  - APP的启动由dyld主导，将可执行文件加载到内存，顺便加载所有依赖的动态库
  - 并由runtime负责加载成objc定义的结构
  - 所有初始化工作结束后，dyld就会调用main函数
  - 接下来就是UIApplicationMain函数，AppDelegate的application:didFinishLaunchingWithOptions:方法

app的优化方案

在了解app整个启动过程之后，我们就可以针对各个启动阶段进行优化

- 按照不同的阶段
  - dyld
    - ✓ 减少动态库、合并一些动态库（定期清理不必要的动态库）
    - ✓ 减少Objc类、分类的数量、减少Selector数量（定期清理不必要的类、分类）
    - ✓ 减少C++虚函数数量
    - ✓ Swift尽量使用struct
  - runtime
    - ✓ 用+initialize方法和dispatch_once取代所有的__attribute__((constructor))、C++静态构造器、ObjC的+load
  - main
    - ✓ 在不影响用户体验的前提下，尽可能将一些操作延迟，不要全部都放在finishLaunching方法中