返回目录:全网各大厂 iOS 面试题-题集大全

如何优化 App 的启动耗时？

iOS 的 App 启动主要分为以下步骤：

- 打开 App，系统内核进行初始化跳转到 dyld 执行。这个过程包括这些步骤：
- 1）分配虚拟内存空间；
- 2）fork 进程；
- 3）加载 MachO （自身所有的可执行 MachO 文件的集合）到进程空间；
- 4）加载动态链接器 dyld 并将控制权交给 dyld 处理。在这个过程中内核会产生 ASLR(Address space layout randomization) 随机数值，这个值用于加载的 MachO 起始地址在内存中的偏移，随机的地址可防止 MachO 代码扫描并被 hack，提升安全性。通过 ASLR 虽然可随机化各内存区基地址，但无法将程序内的代码段和数据段随机化，如果绕过（bypass） ASLR 依然可进行篡改，就需要结合 PIE(Position Independent Executable) 共同使用。与之相似的还有 PIC(Position Independent Code)，位置无关代码，作用于共享库代码。
PIE/PIC 技术需要在编译阶段开启。顾名思义，PIC 可将程序代码装载到任意地址，这样就内部的指针不能靠固定的绝对地址访问，而通过相对地址指令如 adrp 来获取代码和数据。

- **进入 dyld 动态链接器，它负责将一个 App 处理为一个可运行的状态，包含：**

- **加载 MachO 的依赖库（这些依赖库也是 MachO 格式的文件）**。dyld 从可执行 MachO 文件的依赖开始, 递归加载所有依赖的动态库。 动态库包括：iOS 中用到的所有系统动态库：加载 OC runtime 方法的 libobjc，系统级别的 libSystem（例如 libdispatch(GCD) 和 libsystem_blocks(Block)）；其他 App 自己的动态库。根据 Apple 的描述，大部分 App 所加载的库在 100~400 个。不过 iOS 系统库已经被特殊优化过，如提前加入共享缓存，提前做好地址修正等。

- **Fix-ups（地址修正），包括 rebasing 和 binding 等**。ASLR + PIE 技术增强了程序的安全性，使得依赖固定地址进行攻击的方法失效，但也增加了程序自身的复杂度，MachO 文件的 rebase 和 bind info 等部分以及启动时的 fix-ups 地址修正阶段就是配合它而产生的。

- **ObjC 环境配置**。经过了 MachO 程序和依赖库的加载以及地址修正之后，dyld 所做的大部分事情已经完成了。在这一阶段，dyld 开始对主程序的依赖库进行初始化工作，而初始化的执行部分会回调到依赖库内部执行，如 ObjC 的运行时环境所在的 libobjc.A.dylib 以及 libdispatch.dylib 等。ObjC Setup 的过程，主要是对 ObjC 数据进行关联注册：
- 1）dyld 将主程序 MachO 基址指针和包含的 ObjC 相关类信息传递到 libobjc；

- 2）ObjC Runtime 从 __DATA 段中获取 ObjC 类信息，由于 ObjC 是动态语言，可以通过类名获取其实例，所以 Runtime 维护了一个映射所有类的全局类名表。当加载的数据包含了类的定义，类的名字就需要注册到全局表中；
- 3）获取 protocol、category 等类相关属性并与对应类进行关联；
- 4）ObjC 的调用都是基于 selector 的，所以需要对 selector 全局唯一性进行处理。以上步骤由 dyld 启动 libSystem.dylib 统一对基础库进行调用执行，这里面就包含了 libobjc 的 Runtime，同时 Runtime 会在 dyld 绑定回调，当 dyld 处理完相关数据后就会调用 ObjC Runtime 执行 Setup 工作。

- **执行各模块初始化器**。从这一步就开始接近上（业务）层：
- 1）通过 ObjC Runtime 在 dyld 注册的通知，当 MachO 镜像准备完毕后，dyld 会回调到 ObjC 中执行 +load() 方法，包括以下步骤：a）获取所有 non-lazy class 列表；b)按继承以及 category 的顺序将类排入待加载列表；c）对待加载列表中的类进行方法判断并调用 +load() 方法。
- 2）执行 C/C++ 初始化构造器，如通过 attribute((constructor)) 注解的函数。
- 3）如果包含 C++，则 dyld 同样会回调到 libc++ 库中对全局静态变量、隐式初始化等进行调用。

- **查找并跳转到 main() 函数入口**。到了最后，dyld 回到 Load command，找到 LC_MAIN，拿到 entryoff 再加上 MachO 在内存的加载首地址（首地址就是内核传来的 slide 偏移）就得到了 main() 的入口地址，从而进入我们显式的程序逻辑。

进入 main() -> UIApplicationMain -> 初始化回调 -> 显示 UI。

iOS 的 App 启动时长大概可以这样计算：

t(App 总启动时间) = t1(main 调用之前的加载时间) + t2(main 调用之后的加载时间)。

t1 = 系统 dylib(动态链接库)和自身 App 可执行文件的加载。

t2 = main 方法执行之后到 AppDelegate 类中的 application:didFinishLaunchingWithOptions:方法执行结束前这段时间，主要是构建第一个界面，并完成渲染展示。

在 t1 阶段加快 App 启动的建议： * 尽量使用静态库，减少动态库的使用，动态链接比较耗时。 * 如果要用动态库，尽量将多个 dylib 动态库合并成一个。 * 尽量避免对系统库使用 optional linking，如果 App 用到的系统库在你所有支持的系统版本上都有，就设置为 required，因为 optional 会有些额外的检查。 * 减少 Objective-C Class、Selector、Category 的数量。可以合并或者删减一些 OC 类。 * 删减一些无用的静态变量，删减没有被调用到或者已经废弃的方法。 * 将不必须在 +load 中做的事情尽量挪到+initialize 中，+initialize 是在第一次初始化这个类之前被调用，+load 在加载类的时候就被调用。尽量将+load 里的代码延后调用。

* 尽量不要用 C++ 虚函数，创建虚函数表有开销。 * 不要使用 `__atribute__((constructor))`将方法显式标记为初始化器，而是让初始化方法调用时才执行。比如使用 `dispatch_once()`，`pthread_once()`或 `std::once()`。 * 在初始化方法中不调用 `dlopen()`，`dlopen()`有性能和死锁的可能性。 * 在初始化方法中不创建线程。

在 t2 阶段加快 App 启动的建议： * 尽量不要使用 xib/storyboard，而是用纯代码作为首页 UI。 * 如果要用 xib/storyboard，不要在 xib/storyboard 中存放太多的视图。 * 对 `application:didFinishLaunchingWithOptions:`里的任务尽量延迟加载或懒加载。 * 不要在 NSUserDefaults 中存放太多的数据，NSUserDefaults 是一个 plist 文件，plist 文件被反序列化一次。 * 避免在启动时打印过多的 log。 * 少用 NSLog，因为每一次 NSLog 的调用都会创建一个新的 NSCalendar 实例。 * 每一段 SQLite 语句都是一个段被编译的程序，调用 sqlite3_prepare 将编译 SQLite 查询到字节码，使用 `sqlite_bind_int` 绑定参数到 SQLite 语句。 * 为了防止使用 GCD 创建过多的线程，解决方法是创建串行队列, 或者使用带有最大并发数限制的 NSOperationQueue。 * 线程安全：UIKit 只能在主线程执行，除了 UIGraphics、UIBezierPath 之外，UIImage、CG、CA、Foundation 都不能从两个线程同时访问。 * 不要在主线程执行磁盘、网络、Lock 或者 dispatch_sync、发送消息给其他线程等操作。

返回目录:全网各大厂 iOS 面试题-题集大全
