43-剖析使App具有动态化和热更新能力的方案

你好，我是戴铭。今天，我来和你聊聊iOS开发中的动态化和热更新方案。

热更新能力的初衷是，能够及时修复线上问题，减少Bug 对用户的伤害。而动态化的目的，除了修复线上问题外，还要能够灵活更新App 版本。

要实现动态化，就需要具备在运行时动态执行程序的能力。同时，实现了动态化，也就具备了热更新能力。通常情况下，实现动态化的方案有三种，分别是 JavaScriptCore 解释器方案、代码转译方案、自建解释器方案。接下来，我就和你详细说说这三种方案。

JavaScriptCore 解释器方案

iOS 系统内置的JavaScriptCore，是能够在 App 运行过程中解释执行脚本的解释器。

JavaScriptCore 提供了易用的原生语言接口，配合 iOS 运行时提供的方法替换能力，出现了使用 JavaScript 语言修复线上问题的 JSPatch，以及把 JavaScriptCore 作为前端和原生桥梁的 React Native 和 Weex开发框架。这些库，让 App 具有了动态化能力。

但是，对于原生开发者来说，只能解释执行 JavaScript 语言的解释器 JSPatch、React Native 等，我们用起来不是很顺手，还是更喜欢用原生语言来开发。那么，有没有办法能够解决语言栈的问题呢？

代码转译方案

DynamicCocoa 方案将 Objective-C 转换成 JavaScript 代码，然后下发动态执行。这样一来，原生开发者只要使用原生语言去开发调试即可，避免了使用 JavaScript 开发不畅的问题，也就解决了语言栈的问题。

当然，语言之间的转译过程需要解决语言差异的问题，比如 Objective-C 是强类型，而 JavaScript 是弱类型，这两种语言间的差异点就很多。但，好在 JavaScriptCore 解释执行完后，还会对应到原生代码上，所以我们只要做好各种情况的规则匹配，就可以解决这个问题。

手段上，语言转译可以使用现有的成熟工具，比如类 C 语言的转译，可以使用LLVM 套件中 Clang 提供的 LibTooling，通过重载 HandleTranslationUnit() 函数，使用 RecursiveASTVistor 来遍历 AST，获取代码的完整信息，然后转换成新语言的代码。

在这里，我无法穷尽两种编程语言间的转译，但是**如果你想要快速了解转译过程的话，最好的方法就是看一个实现的雏形。**

比如，我以前用 Swift 写过一个 Lisp 语言到 C 语言转译的雏形。你可以点击这个链接，查看具体的代码。通过这个代码，你能够了解到完成转译依次需要用到词法分析器、语法分析器、遍历器、转换器和代码生成器。它们的实现分别对应 LispToC 里的 JTokenizer.swif、JParser.swift、JTraverser.swift、JTransformer.swift和CodeGenerator.swift。

再比如，你可以查看SwiftRewrite项目的完整转译实现。SwiftRewriter 使用 Swift 开发，可以完成 Objective-C 到 Swift 的转换。

自建解释器方案

可以发现，我在前面提到的JSPatch、React Native等库，到最后能够具有动态性，用的都是系统内置的 JavaScriptCore 来解释执行 JavaScript 语言。

虽然直接使用内置的 JavaScriptCore 非常方便，但却限制了对性能的优化。比如，系统限制了第三方 App 对 JavaScriptCore JIT（即时编译）的使用。再比如，由于 JavaScript 使用的是弱类型，而类型推断只能在 LLInt 这一层进行，无法得到足够的优化。

再加上 JSContext 多线程的处理也没有原生多线程处理得高效、频繁的 JavaScriptCore 和原生间的切换、内存管理方式不一致带来的风险、线程管理不一致的风险、消息转发时的解析转换效率低下等等原因，使得 JavaScriptCore 作为解释器的方案，始终无法比拟原生。

虽然通过引入前端技术栈和利用转译技术能够满足大部分动态化和热修复的需求，但一些对性能要求高的团队，还是会考虑使用性能更好的解释器。

如果想要不依赖系统解释器实现动态化和热修复，我们可以集成一个新的解释器，毕竟解释器也是用代码写出来的，使用开源解释器甚至是自己编写解释器，也不是不可以。

因此，腾讯公司曾公布的 OCS方案，自己实现了一个虚拟器 OCSVM 作为解释器，用来解释执行自定义的字节码指令集语言 OCScript，同时提供了将 Objective-C 转成 OCScript 基于 LLVM 定制的编译器 OCS。

腾讯公司自研一个解释器的好处，就是可以最大程度地提高动态化的执行效率，能够解释执行针对 iOS 运行时特性定制的字节码指令。这套定制的指令，不光有基本运算指令，还有内存操作、地址跳转、强类型转换指令。

OCSVM 解释执行 OCScript 指令，能够达到和原生媲美的稳定和高性能，完成运行时 App 的内存管理、解释执行、线程管理等各种任务。OCS 没有开源，所以你无法直接在工程中使用 OCS 方案，但是有些公司自己内部的动态化方案其实就是参考了这个方案。这些方案都没有开源，实现的难度也比较大。

因此，你想要在工程中使用高效的解释器，最好的方案就是，先找找看有没有其他的开源解释器能够满足需求。

这时，如果你仔细思考，一定会想到 LLVM。LLVM 作为标准的 iOS 编译器套件，对 iOS 开发语言的解析是最标准、最全面的。那么，LLVM 套件里面难道就没有提供一个解释器用来动态解释执行吗？

按理说，LLVM来实现这个功能是最合适不过了。其实 LLVM 里是有解释器的。

只不过，ExecutionEngine 里的 Interpreter，是专门用来解释 LLVM IR 的，缺少对 Objective-C 语法特性的支持，所以无法直接使用。除此之外，ExecutionEngine 里还有个 MCJIT，可以通过 JIT 来实现动态化，但因为iOS 系统的限制也无法使用。

其实，LLVM 之所以没有专门针对 iOS 做解释器，是因为 iOS 动态化在 LLVM 所有工作中的优先级并不高。

不过，好在 **GitHub 上有一个基于 LLVM 的 C++ 解释器 Cling，可以帮助我们学习怎样通过扩展 LLVM 来自制解释器。**

解释器分为解释执行 AST 和解释执行字节码两种，其中Cling 属于前者，而 LLVM 自带解释器属于后者。

从效率上来说，解释执行字节码的方案会更好一些，因为字节码可以在编译阶段进行优化，所以使用 LLVM IR 这种字节码，可以让你无需担心类似寄存器使用效率，以及不断重复计算相同值的问题。LLVM 通过优化器可以提高效率，生成紧凑的 IR。而这些优化都在编译时完成，也就提高了运行时的解释效率。

那么，LLVM 是怎么做到的呢？

LLVM IR 是 SSA（Static Single-Assignment，静态单赋值）形式的，LLVM IR 通过 mem2reg Pass 能够识别 alloca 模式，将局部变量变成 SSA value，这样就不再需要 alloca、load、store 了。

SSA 主要解决的是，多种数据流分析时种类多、难以维护的问题。它可以提供一种通用分析方法，把数据流和控制流都写在 LLVM IR 里。比如，LLVM IR 在循环体外生成一个 phi 指令，其中每个值仅分配一次，并且用特殊的 phi 节点合并多个可能的值，LLVM 的 mem2reg 传递将我们初始堆栈使用的代码，转成带有虚拟寄存器的 SSA。这样，LLVM 就能够更容易地分析和优化 IR 了。

LLVM 只是静态计算0和1地址，并且只用0和1处理虚拟寄存器。在高级编程语言中，一个函数可能就会有几十个变量要跟踪，虚拟寄存器计算量大后，如何有效使用虚拟寄存器就是一个很大的问题。SSA 形式的 LLVM IR 的 emitter 不用担心虚拟寄存器的使用效率，所有变量都会分配到堆栈里，由 LLVM 去优化。

其实，我和你分享的OCS 和 Cling 解释器，都是基于 LLVM 扩展实现的。那么，**如果我们不用 LLVM 的话，应该怎么写解释器呢？**

要了解如何写解释器，就要先了解解释器的工作流程。

解释器首先将代码编译为字节码，然后执行字节码，对于使用频次多的代码才会使用 JIT 生成机器代码执行。因此，解释器编译的最初目标不是可执行的机器代码，而是专门用在解释器里解释执行的字节码。

因为编译器编译的机器代码是专门在编译时优化过的，所以解释器的优化就需要推迟到运行时再做。这时，就需要Tracing JIT来跟踪最热的循环优化，比如相同的循环调用超过一百万次，循环就会编译成优化的机器代码。浏览器的引擎，比如 JavaScriptCore、V8，都是基于字节码解释器加上 Tracing JIT 来解释执行 JavaScript 代码的。

其实，**JIT 技术就是在 App 运行时创建机器代码，同时执行这些机器代码。**编译过程，将高级语言转换成汇编语言，Assembler（汇编器）会将汇编语言转换成实际的机器代码。

仅基于字节码的解释器的实现，我们只需要做好解析工作，然后优化字节码和解释字节码的效率，对应上原生的基本方法执行，或者方法替换就可以实现动态化了。

但是，自己实现 JIT 就难多了，一方面编写代码和维护代码的成本都很高，另一方面还需要支持多 CPU 架构，如果搭载 iOS 系统的硬件 CPU 架构有了更新还要再去实现支持。所以，JIT 的标签和跳转都不对外提供调用。

那如果要想实现一个自制 JIT 的话，应该如何入手呢？

用 C++ 库实现的 JIT AsmJit，是一个完整的 JIT 和 AOT 的 Assembler，可以生成支持整个x86和x64架构指令集（从 MMX 到 AVX512）的机器代码。AsmJit的体积很小，在300KB 以内，并且没有外部依赖，非常适合用来实现自己的 JIT。使用 AsmJit 库后，我们再自己动手去为字节码编写 JIT 能力的解释器，就更容易了。

小结

今天这篇文章，我跟你分享了使 App 具有动态化和热更新能力的方案，其中包含了目前大多数项目在使用的 JavaScriptCore 解释器方案。

但由于 JavaScriptCore 方案更适合前端开发者，于是出现了对原生开发者更友好的代码转译方案，代码转译最终解释执行还是 JavaScriptCore，在效率上会受到种种限制。为了更好的性能，便有了在 App 内集成自建解释器的方案。

我觉得热更新用哪种方案问题都不大，毕竟只是修复代码。但是，动态化方案的选择，就要更慎重些了，毕竟整个业务都要用。

动态化方案的选择主要由团队人员自身情况决定，比如原生开发者居多时可以选择代码转译或自建解释器方案；前端开发者居多或者原生开发者有意转向前端开发时，可以选择 JavaScriptCore 方案。

另外，动态化方案本身，对大团队的意义会更加明显。因为大团队一般会根据业务分成若干小团队，由这些不同团队组成的超级大 App 每次发版，都会相互掣肘，而动态化就能够解决不同团队灵活发版的问题，让各个小团队按照自己的节奏来迭代业务。

课后作业

如果你负责的 App 出现了线上问题，你是采用什么方案来修复这个问题的呢？

感谢你的收听，欢迎你在评论区给我留言分享你的观点，也欢迎把它分享给更多的朋友一起阅读。

精选留言：

- Jeffrey 2019-06-20 08:45:30
  吹了2年了 都没有见过DynamicCocoa长啥样 什么时候准备开源 [5赞]

- Du 2019-06-20 10:56:02

是不是结课了？[2赞]

编辑回复2019-06-20 11:15:49

是的，周六就更新结束语了。但结束后，老师也会继续关注专栏，可能会再写些加餐文章，欢迎你继续关注哦