hotfix方案探究

一、背景

做过前端的人知道，前端的代码是实时更新生效的，而客户端则不然。尤其是iOS，在上线之前是需要经过APP审核的。当线上出现问题时，我们只能赶紧修复问题，然后等下个版本上线来解决这个版本的问题。这种修复方式无疑会出现两个问题：1.当前版本的问题其实还是没有修复，如果用户不升级还是存在问题的（当然可以强制更新）2.就是前面说的审核问题，并没有那么高的实时性。

二、热修复

顾名思义，就是能及时修复线上问题。这么强大的功能，往往也会引起安全性的质疑。因此苹果有一系列的措施：1.苹果不允许动态链接非系统动态库。（功能上封杀一部分热修复更新动态库方案，安卓是可以的。）2.苹果在审核时候不允许动态下发可执行代码或者脚本文件。这个第二点又很奇怪，因为通过苹果JavaScriptCore或WebKit执行的代码除外。（详见《iOS Developer Program Information》中 *Sample Apple Developer Program Requirements - 3.3.2* 条例）更琢磨不透的是JSPatch就是基于JavaScriptCore的但是还是被封了。（当时苹果发文说不允许使用JSPatch类方案。）当然此事之后，苹果的审核效率也大大提升（基本2天之内）。当然还有不少APP进行对JSPatch做代码混淆上线成功的。（主要是不能出现Patch字眼）热修复这几个敏感的字眼渐渐的在主流技术方案中淡出了。

三、技术方案

1.优化web新能

其实这个课题就非常大了，其主要意义不在于热修复而是跨平台开发从而提高开发效率。

React Native与Weex

文本之前就说过如果能像前端一样开发就没有这些问题了，这两个框架就是基于这个方向做的努力。把这两个库放在一起是因为这两个库非常的相像，其主要的原理，基于iOS内部的js引擎JavaScriptCore做转化，转化为原生的代码。在热修复方面，其部署在服务端可以随时修改下发js，所以就不成问题了。但是也有个巨大的弊端，就是它们只能修复用该框架开发的项目或模块。详见： 1.Weex 是如何在 iOS 客户端上跑起来的

2.React Native通信机制详解

Flutter

作为跨平台方案的新贵，flutter跟之前两个有很大的不同，这里只聊跟热修复相关的东西，现在其实可以说Flutter关于动态化的方案很少，与上面2个方案不同，flutter可执行文件是打包到项目中的。Android可以通过动态替换相关文件来实现动态化，iOS暂时不行。

上述方案其实都有比较大的开发成本，因为React Native与Weex都是js开发的而flutter则是Dart。

2.基于模块化

严格意义上来将，基于模块化的方案不是热“修复”，它只是把问题遮盖起来了。 这种方式的实现原理其实很简单，就是有后台下发需要加载的模块，在分别展示。当遇到某个模块出现问题时，修改下发的文件，剔除有问题的模块。

基于模块化方案一般用于新上的功能上，这样影响也比较小，衍生出来的方案就是A/BTest。

这种方案进行“修复”其实显而易见有几个致命的问题：1.需要所有的业务模块化，能动态上下线。2.不能修复问题，只能隐藏，对核心业务隐藏（如有重大问题）损失也会很大。

3.基于OC运行时

基于运行时的热修复方案相对来说可以算得上是比较正统的方案了，但是由于“安全”的问题，苹果对这块管控也比较严苛，处理不好会审核被拒。

我们先来简单的介绍下这个方案，这个方案可以细分为三个步骤：

1.后台的版本控制与下发相应的修复代码。

2.把修复代码转化成APP可执行的OC代码（或者直接调用OC代码）。

3.通过运行时处理，执行之前的代码修复bug。

其中第一个步骤相对来说比较简单，主要是做一个可以根据版本控制下发不同代码的平台。例如JSPatch有相应的平台JSPatch Platform。后面两个步骤不同的方案有不同的处理方式。

JSPatch

JSPatch可以说是热修复的标志性的词了。其主要的实现方式如下:

1.通过JavaScriptCore执行js代码。

2.在 OC 执行 JS 脚本前，通过正则把所有方法调用都改成调用 `__c()` 函数，再执行这个 JS 脚本，做到了类似 OC/Lua/Ruby 等的消息转发机制：

```
UIView.alloc().init()
->
UIView.__c('alloc')().__c('init')()
```

而这里的__c就是元函数

```
Object.defineProperty(Object.prototype, '__c', {value: function(methodName) {
  if (!this.__obj && !this.__clsName) return this[methodName].bind(this);
  var self = this
  return function(){
    var args = Array.prototype.slice.call(arguments)
    return _methodFunc(self.__obj, self.__clsName, methodName, args,
self.__isSuper)
  }
}})
```

`_methodFunc()` 就是把相关信息传给OC，OC用 Runtime 接口调用相应方法，返回结果值，这个调用就结束了。

至于方法替换则用了 OC 消息转发机制。

当调用一个 NSObject 对象不存在的方法时，并不会马上抛出异常，而是会经过多层转发，层层调用对象的 `-resolveInstanceMethod:`, `-forwardingTargetForSelector:`, `-methodSignatureForSelector:`, `-forwardInvocation:` 等方法，其中最后 `-forwardInvocation:` 是会有一个 NSInvocation 对象，这个 NSInvocation 对象保存了这个方法调用的所有信息，包括 Selector 名，参数和返回值类型，最重要的是有所有参数值，可以从这个 NSInvocation 对象里拿到调用的所有参数值。

详见：JSPatch实现原理详解

Aspect

Aspect其实严格上说不是一个热修复框架，它是一个轻量级的面向切面编程的库。它能允许你在每一个类和每一个实例中存在的方法里面加入任何代码。可以在以下切入点插入代码：before(在原始的方法前执行) / instead(替换原始的方法执行) / after(在原始的方法后执行,默认)。通过Runtime消息转发实现Hook。Aspects会自动的调用super方法，使用method swizzling起来会更加方便。它的实现原理相对来说复杂点，简单的说与KVO有点类似。就是说新注册一个对象来替换你之前的对象，然后通过消息转发机制到同一个地方进行检查处理。

详见：iOS 如何实现 Aspect Oriented Programming

那么有了第一步和第三步，第二步的实现就变成了关键点。在这方面，不同的人提出了不同的方案：

1.基于JavaScriptCore

这个方案就跟JSPatch有点类似了，但是又有不同，相比较JSPatch来说更简单。简单的说通过对JSContext事先设置好一些基础需要执行的方法，使用修复者再调用这些方法来达到热修复的目的。例如修复方法：

```
    JSContext *context = [self context];
    context[@"fixInstanceMethod"] = ^(NSString *className, NSString
*selectorName, AspectOptions options, JSValue *fixImpl) {
        [self fixMethodIsClass:NO className:className opthios:options
selector:selectorName fixImp:fixImpl];
    };
```

```
+ (void)fixMethodIsClass:(BOOL)isClassMethod
               className:(NSString *)className
                 opthios:(AspectOptions)options
                selector:(NSString *)selector
                  fixImp:(JSValue *)fixImp {
    if(!className){
        NSLog(@"WTCFIX hook className is nil.");
        return;
    }

    Class klass = NSClassFromString(className);
    if(isClassMethod){
        klass = object_getClass(klass);
```

```
    }

    if(!klass){
        NSLog(@"WTCFIX hook className: %@ is not exists.", className);
        return;
    }

    SEL sel = NSSelectorFromString(selector);

    NSError *error;
    [klass aspect_hookSelector:sel withOptions:options
usingBlock:^(id<AspectInfo> aspectInfo) {
        [fixImp callWithArguments:@[aspectInfo.instance,
aspectInfo.originalInvocation, aspectInfo.arguments]];
    } error:&error];

    if(error){
        NSLog(@"WTCFix hook error: %@", error);
    }
}
```

详见：基于Aspects的热修复(HotFix)

2.基于OCEval

OCEval这个框架就十分有意思了，它是一个Objective-C解释器, 能够像 `eval()` 一样动态执行OC代码。有了这个，加上Aspent简直热修复绝配呀，因为下发的直接是OC代码，所以学习成本也非常的低。

详见：OCEval

4.动态库

iOS 开发语言 Objective-C 天生动态，运行时都能随意替换方法，运行时加载动态库又是项很老的技术，只要我把增量的代码和资源打包到一个 framework 里，动态下发运行时加载，修 bug，加功能都不在话下。但是苹果把加载动态库的功能给封了，动态库必须跟随安装包一起签名才能被加载，无法通过别的途径签名后再下发,所以这条路是走不通。(Android是可以做到的。)

5.自己实现引擎

这个就比较厉害了，实现对，相对来说资料也比较少。

DynamicCocoa

这是一个滴滴之前说要开源的项目。其主要的功能是从编译阶段入手，通过 clang 把 OC 代码编译成自己定制的 JS 格式，再动态下发去执行，做到原生开发，动态运行。那么其实其实现原理也就很简单的主要是两大块：1.在Clang的基础上，实现了一个OC源码到JS代码的转换器。2.实现OC-JS互调引擎的DynamicCocoaSDK。当然最后的结果大家都知道，因为当时这点比较热苹果政策比较紧，最后也没有开源。

详见：滴滴 iOS 动态化方案 DynamicCocoa 的诞生与起航

手机QQ内部方案

据说手机QQ内部有类似的方案，他们通过 clang 把 OC 代码编译成自己定制的字节码动态下发，然后开发一个虚拟机去执行。当然，也没有对外公布。

6.其他

其实还有一些热修复方案有些相对较老，有些主要用在游戏里，这里就不再多述，例如lua+wax,xLua等。

四、现状

JSPatch平台最后更新与2019.01.30平台出了1.8.2SDK，未能保证审核通过。（之前1.8.0说是有较大该类被拒。）

滴滴 iOS 动态化方案 DynamicCocoa最后也未开源，讨论热修复的技术文档也销声匿迹。

苹果加快了审核进度，从之前的一个礼拜加快到基本2天出审核结果。

跨平台方案越来越热，Flutter成为客户端新贵。

手炒现阶段并无热修复相关方案，但是有模块动态下发与A/BTest方案。

五、总结

接下来我会从集成实现难度，修复功能性，修复代码编写，审核通过情况对上面提到的一些方案做一个对比

技术方案	跨平台方案	基于模块化	JSPatch	Aspect/jsCore	Aspect/OCEval	Aspect/自解析	自己实现引擎
实现难度	一般	简单	简单	一般	简单	困难	困难
集成难度	一般	困难	简单	简单	简单	简单	简单
修复性能	弱	弱	强	较强	强	强	强
编写难度	较难	简单	一般	一般	容易	容易	容易
审核情况	可过审	可过审	难过审	有风险	难过审	有点风险	有点风险

通过上面的表格我们可以清晰的看到每一种方案的优劣。基于现状我觉得有两条路相对来说可行。1.基于JSPatch做代码混淆，看看是否能通过审核。这条路风险较大。2.基于Aspect的方案，只要我们能解决中间解析的问题（要么用JavaScriptCore解析，要么使用OCEval估计要混淆代码不然难过审，要么自实现解析）。但在最后不得不说，在跨平台方案大火的当下是不是还有必要来做这个事情?