如何编写 iOS Patch

写作背景

在iOS应用开发中，为什么催生了Patch 这种热更新的方式，对大家而言应该无需多说。在苹果未颁布对iOS应用使用下发热更新方式的禁令之前，在国内我们广泛使用了waxPatch, luaView, RubyMotion,JSPatch 作为动态更新的方案。尤其是JSPatch,因其利用了苹果开放的JSCore，并且采用OC语言的动态化技术，使得Patch的内核大为缩减；另一方面，iOS开发者可以采用简单的JS语法，按照熟悉的Cocoa Api方式编写Patch。除此之外，JSPatch 提供了JSPatch代码的转换工具。可以说，使得iOS开发者一键就生成的对应的Patch代码。

但是这种行径，却在今年初被苹果禁止了。在代码中使用了JSPatch开源提供的代码，都将被苹果扫描出来，并且予以警告。这样对于项目周期快，排期紧，容易出现问题的上架应用就不能采用热更新的方式来修复线上的问题了。再次期间，我们试图通过对JSPatch代码的类名，函数名，变量名进行了全文的替换，虽然躲过了一次苹果的审核，但是最终还是被苹果扫描出来并且发送了警告信。所以我们开始了自己编写Patch的计划。

方案的选择

1. 热更新的必要条件

如果要实现热更新，我们下发的Patch必须能够接入当前应用的执行环境，并且能够有执行Patch下发内容的执行环境。说白了，就是有以下两点构成。

- App 有一定的动态性，能够在运行期执行符合某些条件的代码。
- Patch使用语言可以App提供的一个执行环境中(context)被eval。

大家都知道，iOS开发使用的Objective-C语言属于动态语言，即有一条自己的消息执行与消息转发链，最终都是通过调用统一的几个例程完成代码的执行动作。因此我们利用Objective-C 语言提供的runtime机制可以轻松的保证第一点。

第二点的保证相对多元一些。就是我们要选择编写Patch的语言时，首先该语言的内核应该相对简单，其次需要在Objective-C端提供一个context 能够eval 该语言编写的代码，并且提供变量的相互访问功能(提供一个Objective-C 运行环境中，针对Patch使用语言的虚拟机)。

2. 对苹果审核的试探

为了实现自己的Patch，必须要满足上面提到的两个必要条件。但是自己的Patch 还需要符合苹果审查的规范，以至于不要像其他动态更新方案一样被苹果审核机制扫描出来。

关于第一个条件，runtime 机制是苹果API开放出来的。很多做动态代理的框架都有使用，如Aspect, ReactiveCocoa的部分代码。而使用这些框架的应用并不存在苹果审核被拒的问题。因此使用runtime机制提供的API 是不会有问题的。

第二个点就是要下发的Patch采用什么方案。我们知道JSCore是苹果提供的用于执行JS的动态库，可以在OC环境中eval JS代码。这就说明使用JSCore 的应用苹果应该不会禁止。另外就是是否苹果会审查通过JSCore 执行下发Js代码的应用。 但是苹果明确声明，类似使用RN,Weex 这样框架的应用也不会出现问题。 这就说明我们依旧可以利用JSCore 来执行远程下发的JS code。

因此我们还是选择采用远程下发JS语法的Patch代码， 利用runtime动态机制通过JSCore 加载Patch 以执行补丁的方式，实现我们的Patch。

对前辈JSPatch的思考

JSPatch 为什么会被拒绝？我们从几个方面来分析。首先从OC端代码进行考虑。

- JSContext 注册了太多的常量，JSPatch几乎在每一个功能的实现都向对应的JSContext 注册了一个回调，比如 `context[@"_OC_defineClass"]`, `context[@"_OC_callI"]` 等等。 这样的常量很容易就会被苹果采用关键字扫描出来。

- JSPatch 的runtime 代码都实现一个JPEngine 文件当中，有很多可以执行多个功能的超级函数，这样函数的调用关系(及时最后生成的obj)很容易被识别出来。

- 远程下发的JS文件的内容太容易被识别出来是用来做什么的。
  我们根据前辈JSPatch 猜测了这三点，然后逐个击破来搞定我们的Patch.
  JSPatch 的缺点。

- 后端代码的组织上，可能由于JSPatch 作为前辈，在源码阅读的过程中，我们可以看到它针对不同情况的问题所采用的补丁的痕迹，导致代码的阅读性稍微有些拗口。 由于我们是参考JSPatch 或者 Aspect 这样比较成熟的方案来做runtime 动态接入这块，我们可以整理自己的代码，使得看起来更规范一些。

- JSPatch 对Patch代码的预处理能力较弱。虽然采用了往原型链挂载原函数这样的思路解决了API调用的问题。 但是我们会发现针对下发Patch，在OC端还需要正则将JS函数的调用做一次正则替换。另外，Patch代码中对类型的预先处理做不了，也不能用ES6 这样的语法编写Patch。 比如对self 关键字和super 关键字的支持，就能看出略微生硬一些。 我们可以在这方面做些优化。

- JSContext 上注册的用于两端(JS 端 与 OC端)通信的常量过多。我们可以参照weex这种实现方式(weex 指令的传递主要通过在context 注册了define 和 bootstrap 用于命令分发)。注册有限的用于指令分发的context 常量，采用dispatch这样的方式，来完成JS环境与OC端环境的通信。

我们的Patch做了哪些事情

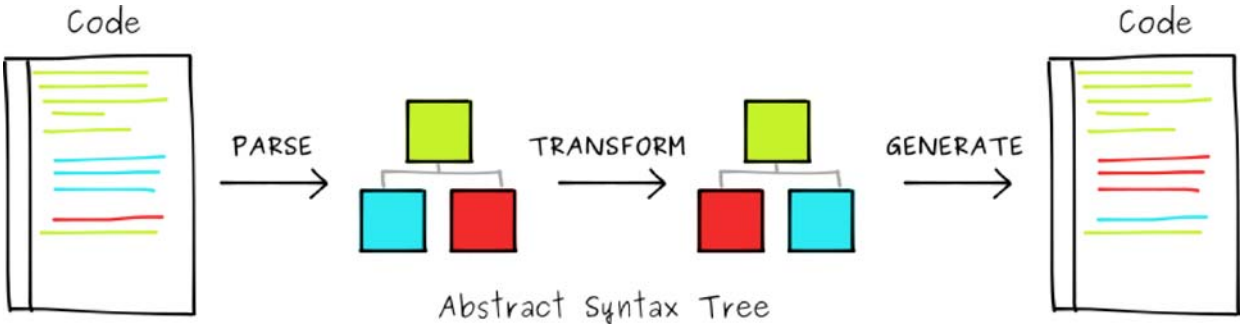
我们将自己的Patch 实现分为前端和后端。 Patch编写的部分称之为前端，主要增加了预处理环节，使得编译生成的Patch可以直接在任意支持context参数的JS环境中执行; 后端主要是Patch的动态接入，通过分发机制，利用runtime提供的能力插件化的实现各种功能。我们注册的context 只有三种类型，一种用于定义(比如定义要hook的类的方法，定义新属性，定义新block), 一种用于执行(比如执行OC方法的调用)，最后一种用于回调给前端环境。

后端的实现，主要利用了runtime机制，因此从技术上而言，除了对block的支持和addmethod方面做了相应的改进之外，其余的和JSPatch 实现的方式都差不太多。因此并不是我们今天讨论的重点。如果有需要的同学，可以使用我们后端的framework, 按照framework 指令接入规范接入自己的Patch, 这样可以避免大家重复造轮子带来的浪费。

这里主要探讨一下前端的实现方式，可能为我们以后遇到类似问题会带来一些帮助。在正式介绍Patch前端的实现方式之前，我们先回顾一下我们所使用的技术：

Babel transform

babel 在处理代码的过程中有3个核心的工具。



如上图所示，使用Babel对编写的源代码进行转换的时候，主要有三个流程：

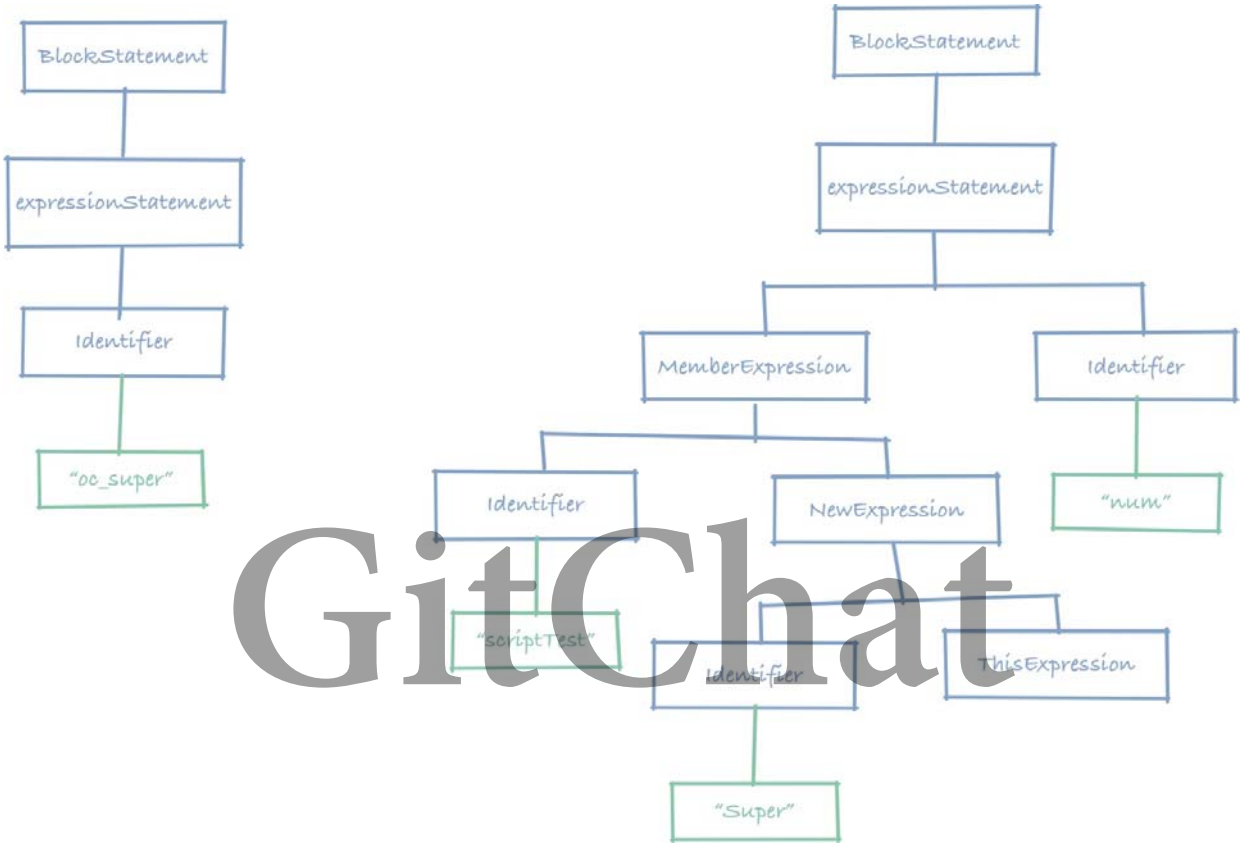
首先，使用Babylon工具对输入的源代码进行语法分析，生成抽象语法树(AST)；生成抽象语法树后，可以使用babel-traverse 浏览、修改相应的AST 节点的内容。 最后通过babel-generator模块转换AST为最终格式的javascript代码。

在我们的工程中会以oc_super表达式的转换为例，简述babel-tramsform在AST转换中的使用。

oc_super 在Patch语法中表示调用对应Objective-C端函数的父类方法（由于super 在 ES 6 中是关键字，所以增加了 oc_ 前缀加以区分）; 首先看一下使用 oc_super 表达式在编译

前后的差别。

针对于oc_super 表达式，转换前后的抽象语法树如下所示：



针对oc_super 我们做了如下工作：

1. 遵照babel transform 规范，配置替换oc_super 表达式的babel-plugin
2. 编写visitor 。先将 “oc_super;” 表达式替换成 “oc_super();” callExpression 形式

```
Identifier(path) {
    let token = "oc_super";
    let parentPath = path.parentPath;
    let node = path.node;
    if (node.name == token){
     if(!parentPath.isCallExpression()){
       path.replaceWith(t.callExpression(t.identifier(token),
[]));
    }
    }
  },
```

3. 编写 callExpression(path) 逻辑，将”oc_super()” 替换成 “new Super(this)” 表达式。

```
 const OCSuperTemplate = template(`new Super(INSTANCE)`);
  CallExpression(path){
   ...
     const caller = OCSuperTemplate({INSTANCE:t.thisExpression()});
      path.node.callee =
t.memberExpression(caller.expression,t.identifier(funcName));
       path.node.arguments = args;
}
```

Flow

在Objective-C 语言中，对于不同的数据类型都会有相应的类型编码。在Patch的编写中会遇到到JS语法的数据类型与Objective-C 中数据类型的转换。如果在编写patch中能够将类型信息表示出来，并且通过前端预处理将对应的类型转换为Objective-C 对应数据类型的类型编码则会有助于提高性能。 为了使得前端支持静态类型的语义表达，在patch前端借助了Flow。

Flow 是由Facebook 开源的一个JS 静态类型检查工具，babel 集成了babel-preset-flow 插件，可以用于对使用Flow 做静态类型检查的代码进行转换。在构建的AST中可以获取相应节点的类型信息，以至于我们可以根据该信息来转换Objective-C的类型编码。 使用方式如下：

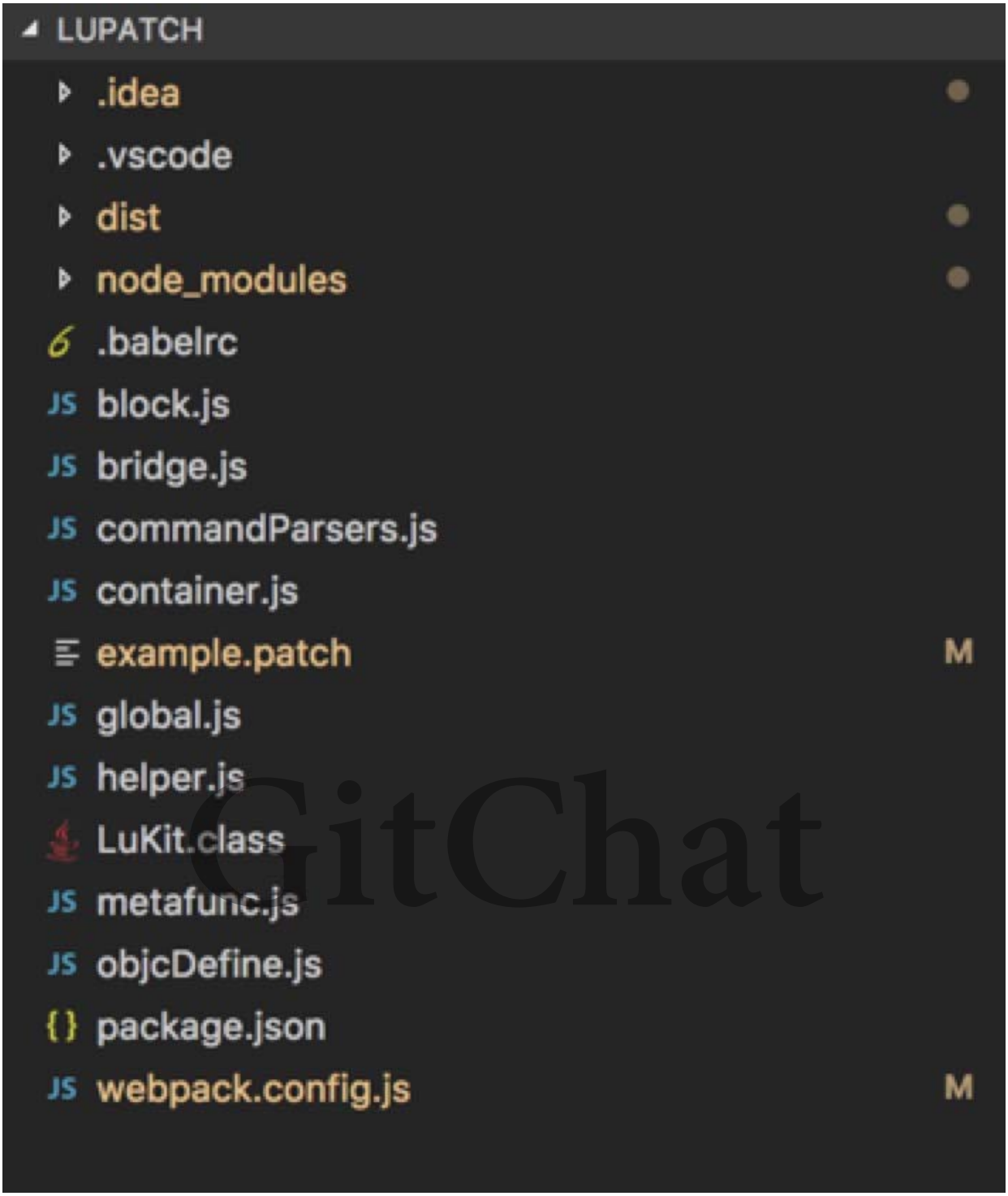

```
require("babel-core").transform("code", {
  presets: ["flow"],
  plugins: ["patch-semantics"]
});
```

Webpack

webpack 是一个现代JavaScript 的应用模块打包器。我们采用webpack 用于构建我们patch的输出文件。为了实现patch语法的转换，我们参考了webpack的loader 规范，编写了patch-loader。 具体的loader 编写，网上有很多教程，由于篇幅有限，再这里就不做展开。

下面，主要介绍Patch 前端的设计与实现。

首先看一下前端工程目录的组织结构：



patch 代码的内容放在以 .patch为后缀的文件中。 objcDefine.js 封装了与后端通信的接口，使用patch时需要引入(require)该文件。

其次看一下webpack.config.js 的配置内容：

```
module.exports = {
  entry: "./example.patch", // string | object | array
  output: {
   path: path.resolve(__dirname, "dist"), // string
    filename: "20171116/bundle.js", // string
  },

  module: {
    rules: [
      {
        test: /\.jsx?$/,
        include: [
          path.resolve(__dirname, "./")
        ],
        loader: "babel-loader",
      },
      {
        test: /\.patch$/,
        include: [
          path.resolve(__dirname, "./")
        ],
        loader: "babel-loader!patch-loader",
      },
       {
        test: /\.class$/,
         include: [
          path.resolve(__dirname, "./")
        ],
        loader: 'raw-loader!babel-loader!patch-loader'
      }
    ],
  },
    plugins:[new UglifyJSPlugin(),new PatchDigestPlugin()]
}
```


1. 首先配置打包的入口文件，由于本工程只有一个patch文件，所以配置为example.patch

2. webpack loader 配置： 针对工程中的 .js 文件，配置babel-loader 做chunk的转换；针对工程中的.patch文件，使用patch-loader 进行转换。

3. webpack plugin配置： UglifyJSPlugin 用于对生成的bundle.js 进行混淆压缩处理；PatchDigestPlugin是用来计算Patch文件的md5值，用于remote端接口配置的参数。

部分关键字处理举例：

```
import {requireCocoa} from './objcDefine.js'
import LuKit from './LuKit.class'
const {LuViewControllerTest,LuScriptTest,UITableView}= requireCocoa('LuViewControllerTest','LuScriptTest','UITableView');

LuSubScriptTest.renewInstanceMethods({scriptTest:function(num){
    var a = oc_super;
    self.aa = oc_super.fetch_value();
    let c ;
    c.a.d = c.a.b;
  return original() ;},
   tableView_numberOfRowsInSection:function(tableView,section){
       return original() + 400;
   }
});
```

上图为example.patch文件中的代码片段，为了尽量符合Js的语法，我们对很多关键字和调用方式做了预处理。 这里主要使用了babel 提供的babel.transform, 遍历整个语法树，对相应的节点做替换。可以参考Babylon的使用方式做这些语法处理。部分关键字的处理如下图所示。

oc 语法	jsPatch	luPatch
函数调用	__c()元函数 UIView.caller(a,b,c) UIView.__c('caller')(a,b,c) 1.每次调用都返回一个闭包，在__c的调用中获取函数名称在闭包的调用中传入参数 2.oc 获取到Patch后进行全文的正则替换	利用patch-loader 做转换 UIView.caller(a,b,c) UIView.__forward('caller',a,b,c) 1. patch-loder 遍历文件的语法树，对所有的callExpression 进行处理 2. babel-loader 暴露的visitor接口非常方便

self	self 作为全局变量，在方法定义的位置执行了如下操作:
	var lastSelf = global.self
	global.self = args[0]
	//执行函数调用
	global.self = lastSelf
		self 的语义和this 相同，那么将self 这个identifier 直替换成 thisExpression()

super	1. self.super().viewDidLoad()
	2. super: function(){
	var slf = this
	/****/
	return {__obj: slf.__obj,
	__clsName: slf.__clsName
	__isSuper: 1}
	}
	3. 在callSelector 中根据isSuper 标识区分
		super 在es6同样是关键字，这里声明super 为oc_super (以后考虑进行兼容性改造)
		将super.call 与 instance.call 分开
		1. 定义了Super 对象，封装super的call_i 和 call_c 的方法
		class Super {
		constructor(instance){
		this.instance = instance
		}
		}
		2. 将oc_super的调用转成了 new Super(this).#nearestFunction(#arguments)

除此之外，我们将Js 与 Objective-C 类型转换的工作也放到预编译阶段处理。在写Js代码的时候，需要携带声明参数的类型。这样编译阶段就将类型编码处理成对应的字符串，直接通过命令传递到后端进行处理。

如 `a.request((num1:double, num2:double):double => { return num1 + num2; })` request 是一个高阶函数类型，参数为两个double 型参 一个double 类型返回值的函数类型。该函数在发生调用前会转换成double (^)(double, double) 类型的 Objective-C block。另外，为了方便Patch的书写，我们对容器类型和属性的存取也做了相应的语法转换。使用方式如下图。

容器使用示例

```
// OC 端对应的代码
@interface LuScriptTest : NSObject<UITableViewDelegate>
    @property (nonatomic,strong) NSMutableDictionary *dic;
    @property (nonatomic,strong) NSMutableArray* arr;
@end
// Patch 代码
 let test = LuScriptTest.alloc().init();
 test.arr = [1,2,3,4]; //赋值
 test.arr[2]= 10; //修改
 const val = test.arr[3]; //读取
 test.arr.push(20)// js的容器API
 test.arr.addObject(30)//OC端的容器API
 test.arr.insertObject_atIndex("inserted",2)//OC端的容器API
 test.arr.getOrigin();//test.arr 作为参数传递到OC端需要这样调用 [patch-loader 没有为参数进行转码]

 test.dic ={name:'刘大铜',sex:'male',age: 27}
 test.dic.age = 28 //修改
 test.dic.company = 'CCTV' //新增
 test.dic.age = null // 删除age字段
 test.dic.setObject_forKey('刘大银',name) //调用OC端容器API
 test.dic.getOrigin();//同test.arr的API调用含义
```

开发人员如何快速编写自己的Patch

如果打算接入我们后端的framework, 你只需要根据自己的需要编写符合个人语法习惯的Patch-Loader，利用webpack 环境就可以快速的构建自己的Patch了。你可以利用npm提供的各种包，webpack提供的各种插件，写出自己花哨的代码。最后通过webpack build一条指令，就快速生成了（目前而言苹果不会审核出来）Patch。