**iOS Web App 开发技术**

# 技术框架

1. JavaScript core framework

JavaScriptCore是webkit的一个重要组成部分，主要是对JS进行解析和提供执行环境。代码是开源的，可以下下来看看（[源码](https://github.com/phoboslab/JavaScriptCore-iOS)）。iOS7后苹果在iPhone平台推出，极大的方便了我们对js的操作。我们可以脱离webview直接运行我们的js。iOS7以前我们对JS的操作只有webview里面一个函数 stringByEvaluatingJavaScriptFromString，JS对OC的回调都是基于URL的拦截进行的操作。大家用得比较多的是[WebViewJavascriptBridge](https://github.com/marcuswestin/WebViewJavascriptBridge)和[EasyJSWebView](https://github.com/dukeland/EasyJSWebView)这两个开源库，很多混合都采用的这种方式。JavaScriptCore和我们相关的类不是很多，使用起来也非常简单。

JSContext

JS执行的环境，同时也通过JSVirtualMachine管理着所有对象的生命周期，每个JSValue都和JSContext相关联并且强引用context。

JSValue

JS对象在JSVirtualMachine中的一个强引用，其实就是Hybird对象。我们对JS的操作都是通过它。并且每个JSValue都是强引用一个context。同时，OC和JS对象之间的转换也是通过它，相应的类型转换如下：

Objective-C type | JavaScript type

--------------------+---------------------

nil | undefined

NSNull | null

NSString | string

NSNumber | number, boolean

NSDictionary | Object object

NSArray | Array object

NSDate | Date object

NSBlock (1) | Function object (1)

id (2) | Wrapper object (2)

Class (3) | Constructor object (3)

JSManagedValue

JS和OC对象的内存管理辅助对象。由于JS内存管理是垃圾回收，并且JS中的对象都是强引用，而OC是引用计数。如果双方相互引用，势必会造成循环引用，而导致内存泄露。我们可以用JSManagedValue保存JSValue来避免。

\_managedValue = [JSManagedValue managedValueWithValue:jsValue];

[[[JSContext currentContext] virtualMachine] addManagedReference:\_managedValue withOwner:self];

JSVirtualMachine

JS运行的虚拟机，有独立的堆空间和垃圾回收机制。

JSExport

一个协议，如果JS对象想直接调用OC对象里面的方法和属性，那么这个OC对象只要实现这个JSExport协议就可以了。

1. URL Loading system
2. WebKit

<http://www.cocoachina.com/ios/20150203/11089.html>

<http://www.cocoachina.com/ios/20150205/11108.html>

1. H5，JavaScript等

# 请求Request

1. 参数传递

a.将参数放入Request的header中，我们可以使用扩展URLProtocol协议，通过拦截网络请求，然后修改Request的请求header实现：

fileprivate let filterProtocolKey = "protocolKey"//防止循环请求的key  
  
@objc class YOURURLProtocol: URLProtocol, NSURLConnectionDelegate, NSURLConnectionDataDelegate {  
  
fileprivate var responseData:NSMutableData!  
fileprivate var connection:NSURLConnection!  
  
  
override class func canInit(with:URLRequest) -> Bool { // 是否拦截请求，再做处理  
var isBlockRequest = false  
if let url = with.url {  
if with.url!.host == ServerHost.host {  
if url.path.contains(YOURManagerURL.wxInfoPath) {  
isBlockRequest = false  
} else {  
if with.allHTTPHeaderFields != nil {  
if with.allHTTPHeaderFields!["native-uin"] != nil {  
let key = URLProtocol.property(forKey: filterProtocolKey, in: with)  
if key != nil {  
if key as! Bool == true {  
isBlockRequest = false  
}  
} else {  
isBlockRequest = true  
}  
  
} else {  
isBlockRequest = true  
}  
} else {  
isBlockRequest = false  
}  
}  
}  
}  
  
return isBlockRequest  
}

override class func canonicalRequest(for: URLRequest) ->URLRequest {  
let request = NSMutableURLRequest(url: `for`.url!, cachePolicy: `for`.cachePolicy, timeoutInterval: `for`.timeoutInterval)  
request.setValue(YOURManagerGlobalInfo.sharedInstance.user?.userID, forHTTPHeaderField: H5HeaderFiled.uin)  
request.setValue(YOURManagerGlobalInfo.sharedInstance.user?.userToken, forHTTPHeaderField: H5HeaderFiled.token)  
return request as URLRequest  
}

override class func requestIsCacheEquivalent(\_ a: URLRequest, to b: URLRequest) -> Bool {  
return super.requestIsCacheEquivalent(a, to: b)  
}  
  
override func startLoading() {  
let req = (self.request as NSURLRequest).mutableCopy()  
URLProtocol.setProperty(true, forKey: filterProtocolKey, in: req as! NSMutableURLRequest)  
self.connection = NSURLConnection(request: req as! URLRequest , delegate: self)  
self.connection.start()  
  
}  
  
override func stopLoading() {  
YOURManagerLog.log(info: "")  
if self.connection != nil {  
self.connection.cancel()  
self.connection = nil  
}  
}

func connection(\_ connection: NSURLConnection, didFailWithError error: Error) {  
self.client?.urlProtocol(self, didFailWithError: error)  
if self.responseData != nil {  
if self.responseData.length > 0 {  
self.client?.urlProtocol(self, didLoad: self.responseData as Data)  
}  
}  
  
self.connection = nil  
}  
  
func connection(\_ connection: NSURLConnection, didReceive response: URLResponse) {  
self.responseData = NSMutableData()  
self.client?.urlProtocol(self, didReceive: response, cacheStoragePolicy: .notAllowed)  
}  
  
func connection(\_ connection: NSURLConnection, didReceive data: Data) {  
if (data as NSData).length > 0 {  
if self.responseData == nil {  
self.responseData = NSMutableData()  
}  
self.responseData.append(data)  
self.client?.urlProtocol(self, didLoad: data)  
}  
}  
  
func connectionDidFinishLoading(\_ connection: NSURLConnection) {  
self.client?.urlProtocolDidFinishLoading(self)  
self.client?.urlProtocol(self, didLoad: self.responseData as Data)  
self.connection = nil  
}  
  
func connection(\_ connection: NSURLConnection, willCacheResponse cachedResponse: CachedURLResponse) -> CachedURLResponse? {  
if self.request.cachePolicy == .reloadIgnoringLocalCacheData {  
return nil  
} else {  
return cachedResponse  
}  
}  
}

b．使用JavaScript注入的方式，这个需要使用WKWebView

WKUserContentController \*contentCtrl = [[WKUserContentController alloc] init];  
WKUserScript \*userScript = [[WKUserScript alloc] initWithSource:@"var user='uwei';" injectionTime:WKUserScriptInjectionTimeAtDocumentStart forMainFrameOnly:YES];  
[contentCtrl addUserScript:userScript];  
WKWebViewConfiguration \*config = [[WKWebViewConfiguration alloc] init];  
config.userContentController = contentCtrl;  
self.testWebView = [[WKWebView alloc] initWithFrame:self.view.frame configuration:config];

1. 响应Response
2. Web与Native通信

获取JSContext对象，然后将JS代码加载到context里面，最后取到这个函数对象，调用callWithArguments这个方法进行参数传值

self.context = [[JSContext alloc] init];//手动创建

在UIWebView加载网页的时候

获取JSContextx对象的方法是:

context = [webView valueForKeyPath:@"documentView.webView.mainFrame.javaScriptContext"];

1. JavaScript call native

JS调用OC有两个方法：block和JSExport protocol

Block的方式

//使用键值对的方式将block注册为JavaScript可用的方法

context[@"add"] = ^(NSInteger a, NSInteger b) {

NSLog(@"---%@", @(a + b));

};

[context evaluateScript:@"add(2,3)"];

JSExport protocol

//定义一个JSExport protocol

@protocol JSExportTest <JSExport>

- (NSInteger)add:(NSInteger)a b:(NSInteger)b;

@property (nonatomic, assign) NSInteger sum;

@end

//建一个对象去实现这个协议：

@interface JSProtocolObj : NSObject  
@end  
  
@implementation JSProtocolObj

@synthesize sum = \_sum;

//实现协议方法  
- (NSInteger)add:(NSInteger)a b:(NSInteger)b  
{  
return a+b;  
}  
//重写setter方法方便打印信息，  
- (void)setSum:(NSInteger)sum  
{  
NSLog(@"--%@", @(sum));  
\_sum = sum;  
}  
@end  
//测试  
@interface ViewController ()

@property (nonatomic, strong) JSProtocolObj \*obj;  
@property (nonatomic, strong) JSContext \*context;  
@end

@implementation ViewController

- (void)viewDidLoad {  
[super viewDidLoad];  
//创建context  
self.context = [[JSContext alloc] init];  
//设置异常处理  
self.context.exceptionHandler = ^(JSContext \*context, JSValue \*exception) {  
[JSContext currentContext].exception = exception;  
NSLog(@"exception:%@",exception);  
};  
//将obj添加到context中  
self.context[@"OCObj"] = self.obj;  
//JS里面调用Obj方法，并将结果赋值给Obj的sum属性  
[self.context evaluateScript:@"OCObj.sum = OCObj.addB(2,3)"];

}

@protocol JSExportTest

//用宏转换下，将JS函数名字指定为add；

JSExportAs(add, - (NSInteger)add:(NSInteger)a b:(NSInteger)b);

@property (nonatomic, assign) NSInteger sum;

@end

//调用

[self.context evaluateScript:@"OCObj.sum = OCObj.add(2,3)"];

可以自定义异常捕获，可以把context，异常block改为自己的：

self.context.exceptionHandler = ^(JSContext \*context, JSValue \*exception) {

[JSContext currentContext].exception = exception;

NSLog(@"exception:%@",exception);  
};

1. native call JavaScript

获取JavaScript函数的方法

NSString \*js = @"function add(a,b) {return a+b}";

JSValue \*n = [context evaluateScript:js];或者是

JSValue \*n = [context[@"add"] callWithArguments:@[@2, @3]];

NSLog(@"---%@", @([n toInt32]));//---5

1. 数据对象传递

新建一个类，使用上述的方式，实现JSExport Protocol，这种方式可以将一个Native的对象暴露给JavaScript

JSProtocolObj \*nativeObj = [JSProtocolObj new];

Context[“nativeObj”] = nativeObj;

然后就可以在JavaScript以自己的方式， 使用这个对象。

# 缓存

NSURLCache 为应用的 URL 请求提供了内存以及磁盘上的综合缓存机制，作为基础类库 URL 加载的一部分，任何通过 NSURLConnection 加载的请求都将被 NSURLCache 处理。  
网络缓存减少了需要向服务器发送请求的次数，同时也提升了离线或在低速网络中使用应用的体验。  
当一个请求完成下载来自服务器的回应，一个缓存的回应将在本地保存。下一次同一个请求再发起时，本地保存的回应就会马上返回，不需要连接服务器。NSURLCache  
会 自动 且 透明 地返回回应。  
为了好好利用 NSURLCache  
，你需要初始化并设置一个共享的 URL 缓存。在 iOS 中这项工作需要在 -application:didFinishLaunchingWithOptions:  
完成  
- (BOOL)application:(UIApplication \*)application didFinishLaunchingWithOptions:(NSDictionary \*)launchOptions{ NSURLCache \*URLCache = [[NSURLCache alloc] initWithMemoryCapacity:4 \* 1024 \* 1024   
diskCapacity:20 \* 1024 \* 1024  
diskPath:nil];  
[NSURLCache setSharedURLCache:URLCache];}  
缓存策略由请求（客户端）和回应（服务端）分别指定。理解这些策略以及它们如何相互影响，是为您的应用程序找到最佳行为的关键

无论采用哪种NSURLRequestCachePolicy，在请求的时候都会cache。而在下次请求（不包括[webview reload]，刷新会强制重新发请求）的时候，则会根据NSURLRequestCachePolicy的策略决定是否拿缓存。如果是NSURLRequestUseProtocolCachePolicy，还要根据对应的协议（http\https\ftp）的服务端提供的response来决定。

使用缓存的目的是为了使应用程序能更快速的响应用户输入，是程序高效的运行。有时候我们需要将远程web服务器获取的数据缓存起来，以空间换取时间，减少对同一个url多次请求，减轻服务器的压力，优化客户端网络，让用户体验更良好。  
背景：NSURLCache : 在iOS5以前，apple不支持磁盘缓存，在iOS5的时候，允许磁盘缓存，（NSURLCache 是根据NSURLRequest 来实现的）只支持http，在iOS6以后，支持http和https。  
缓存的实现说明：由于GET请求一般用来查询数据，POST请求一般是发大量数据给服务器处理（变动性比较大），因此一般只对GET请求进行缓存，而不对POST请求进行缓存。  
缓存原理：一个NSURLRequest对应一个NSCachedURLResponse  
缓存技术：把缓存的数据都保存到数据库中。  
NSURLCache的常见用法：  
（1）获得全局缓存对象（没必要手动创建）NSURLCache \*cache = [NSURLCache sharedURLCache];   
（2）设置内存缓存的最大容量（字节为单位，默认为512KB）- (void)setMemoryCapacity:(NSUInteger)memoryCapacity;  
（3）设置硬盘缓存的最大容量（字节为单位，默认为10M）- (void)setDiskCapacity:(NSUInteger)diskCapacity;  
（4）硬盘缓存的位置：沙盒/Library/Caches  
（5）取得某个请求的缓存- (NSCachedURLResponse \*)cachedResponseForRequest:(NSURLRequest \*)request;   
（6）清除某个请求的缓存- (void)removeCachedResponseForRequest:(NSURLRequest \*)request;  
（7）清除所有的缓存- (void)removeAllCachedResponses;  
  
缓存GET请求：  
　　要想对某个GET请求进行数据缓存，非常简单  
　　NSMutableURLRequest \*request = [NSMutableURLRequest requestWithURL:url];

// 设置缓存策略  
　　request.cachePolicy = NSURLRequestReturnCacheDataElseLoad;  
　　只要设置了缓存策略，系统会自动利用NSURLCache进行数据缓存  
iOS对NSURLRequest提供了7种缓存策略：（实际上能用的只有4种）  
NSURLRequestUseProtocolCachePolicy // 默认的缓存策略（取决于协议）  
NSURLRequestReloadIgnoringLocalCacheData // 忽略缓存，重新请求  
NSURLRequestReloadIgnoringLocalAndRemoteCacheData // 未实现  
NSURLRequestReloadIgnoringCacheData = NSURLRequestReloadIgnoringLocalCacheData // 忽略缓存，重新请求  
NSURLRequestReturnCacheDataElseLoad// 有缓存就用缓存，没有缓存就重新请求  
NSURLRequestReturnCacheDataDontLoad// 有缓存就用缓存，没有缓存就不发请求，当做请求出错处理（用于离线模式）  
NSURLRequestReloadRevalidatingCacheData // 未实现  
  
缓存的注意事项：  
  
缓存的设置需要根据具体的情况考虑，如果请求某个URL的返回数据：  
  
　　（1）经常更新：不能用缓存！比如股票、彩票数据  
  
　　（2）一成不变：果断用缓存  
  
　　（3）偶尔更新：可以定期更改缓存策略 或者 清除缓存  
  
提示：如果大量使用缓存，会越积越大，建议定期清除缓存  
  
NSURLCache的属性介绍：  
  
//获取当前应用的缓存管理对象  
+ (NSURLCache \*)sharedURLCache;  
//设置自定义的NSURLCache作为应用缓存管理对象  
+ (void)setSharedURLCache:(NSURLCache \*)cache;  
//初始化一个应用缓存对象  
/\*  
memoryCapacity 设置内存缓存容量  
diskCapacity 设置磁盘缓存容量  
path 磁盘缓存路径  
内容缓存会在应用程序退出后 清空 磁盘缓存不会  
\*/  
- (instancetype)initWithMemoryCapacity:(NSUInteger)memoryCapacity diskCapacity:(NSUInteger)diskCapacity diskPath:(nullable NSString \*)path;  
//获取某一请求的缓存  
- (nullable NSCachedURLResponse \*)cachedResponseForRequest:(NSURLRequest \*)request;  
//给请求设置指定的缓存  
- (void)storeCachedResponse:(NSCachedURLResponse \*)cachedResponse forRequest:(NSURLRequest \*)request;  
//移除某个请求的缓存  
- (void)removeCachedResponseForRequest:(NSURLRequest \*)request;  
//移除所有缓存数据  
- (void)removeAllCachedResponses;  
//移除某个时间起的缓存设置  
- (void)removeCachedResponsesSinceDate:(NSDate \*)date NS\_AVAILABLE(10\_10, 8\_0);  
//内存缓存容量大小  
@property NSUInteger memoryCapacity;  
//磁盘缓存容量大小  
@property NSUInteger diskCapacity;  
//当前已用内存容量  
@property (readonly) NSUInteger currentMemoryUsage;  
//当前已用磁盘容量  
@property (readonly) NSUInteger currentDiskUsage;  
  
与HTTP服务器进行交互的简单说明：  
Cache-Control头  
在第一次请求到服务器资源的时候，服务器需要使用Cache-Control这个响应头来指定缓存策略，它的格式如下：Cache-Control:max-age=xxxx，这个头指指明缓 存过期的时间  
Cache-Control头具有如下选项:  
public: 指示可被任何区缓存  
private  
no-cache: 指定该响应消息不能被缓存  
no-store: 指定不应该缓存  
max-age: 指定过期时间  
min-fresh:  
max-stable:  
Last-Modified/If-Modified-Since  
Last-Modified 是由服务器返回响应头，标识资源的最后修改时间.  
  
If-Modified-Since 则由客户端发送，标识客户端所记录的，资源的最后修改时间。服务器接收到带有该请求头的请求时，会使用该时间与资源的最后修改时间进行对比，如果发现资源未被修改过，则直接返回HTTP 304而不返回包体，告诉客户端直接使用本地的缓存。否则响应完整的消息内容。  
  
Etag/If-None-Match  
Etag 由服务器发送，告之当资源在服务器上的一个唯一标识符。  
  
客户端请求时，如果发现资源过期(使用Cache-Control的max-age)，发现资源具有Etag声明，这时请求服务器时则带上If-None-Match头，服务器收到后则与资源的标识进行对比，决定返回200或者304。  
文件缓存：借助ETag或Last-Modified判断文件缓存是否有效  
Last-Modified  
服务器的文件存贮，大多采用资源变动后就重新生成一个链接的做法。而且如果你的文件存储采用的是第三方的服务，比如七牛、青云等服务，则一定是如此。

这种做法虽然是推荐做法，但同时也不排除不同文件使用同一个链接。那么如果服务端的file更改了，本地已经有了缓存。如何更新缓存？  
这种情况下需要借助 ETag 或 Last-Modified 判断图片缓存是否有效。  
Last-Modified 顾名思义，是资源最后修改的时间戳，往往与缓存时间进行对比来判断缓存是否过期。

在浏览器第一次请求某一个URL时，服务器端的返回状态会是200，内容是你请求的资源，同时有一个Last-Modified的属性标记此文件在服务期端最后被修改的时间，格式类似这样：

Last-Modified: Fri, 12 May 2006 18:53:33 GMT  
客户端第二次请求此URL时，根据 HTTP 协议的规定，浏览器会向服务器传送 If-Modified-Since 报头，询问该时间之后文件是否有被修改过：

If-Modified-Since: Fri, 12 May 2006 18:53:33 GMT  
总结下来它的结构如下：

|  |  |
| --- | --- |
| 请求 HeaderValue | 响应 HeaderValue |
| Last-Modified | If-Modified-Since |

如果服务器端的资源没有变化，则自动返回 HTTP 304 （Not Changed.）状态码，内容为空，这样就节省了传输数据量。当服务器端代码发生改变或者重启服务器时，则重新发出资源，返回和第一次请求时类似。从而保证不向客户端重复发出资源，也保证当服务器有变化时，客户端能够得到最新的资源。

判断方法用伪代码表示：

if ETagFromServer != ETagOnClient || LastModifiedFromServer != LastModifiedOnClient  
GetFromServer  
else  
GetFromCache  
之所以使用  
LastModifiedFromServer != LastModifiedOnClient  
而非使用：  
LastModifiedFromServer > LastModifiedOnClient  
原因是考虑到可能出现类似下面的情况：服务端可能对资源文件，废除其新版，回滚启用旧版本，此时的情况是：

LastModifiedFromServer <= LastModifiedOnClient  
但我们依然要更新本地缓存。

实例：  
/\*!  
@brief 如果本地缓存资源为最新，则使用使用本地缓存。如果服务器已经更新或本地无缓存则从服务器请求资源。  
@details  
步骤：  
1. 请求是可变的，缓存策略要每次都从服务器加载  
2. 每次得到响应后，需要记录住 LastModified  
3. 下次发送请求的同时，将LastModified一起发送给服务器（由服务器比较内容是否发生变化）  
@return 图片资源  
\*/  
- (void)getData:(GetDataCompletion)completion {  
NSURL \*url = [NSURL URLWithString:kLastModifiedImageURL];  
NSMutableURLRequest \*request = [NSMutableURLRequest requestWithURL:url cachePolicy:NSURLRequestReloadIgnoringCacheData timeoutInterval:15.0];  
  
// // 发送 etag  
// if (self.etag.length > 0) {  
// [request setValue:self.etag forHTTPHeaderField:@"If-None-Match"];  
// }  
// 发送 LastModified  
if (self.localLastModified.length > 0) {  
[request setValue:self.localLastModified forHTTPHeaderField:@"If-Modified-Since"];  
}  
  
[[[NSURLSession sharedSession] dataTaskWithRequest:request completionHandler:^(NSData \*data, NSURLResponse \*response, NSError \*error) {  
  
// NSLog(@"%@ %tu", response, data.length);  
// 类型转换（如果将父类设置给子类，需要强制转换）  
NSHTTPURLResponse \*httpResponse = (NSHTTPURLResponse \*)response;  
NSLog(@"statusCode == %@", @(httpResponse.statusCode));  
// 判断响应的状态码是否是 304 Not Modified （更多状态码含义解释： https://github.com/ChenYilong/iOSDevelopmentTips）  
if (httpResponse.statusCode == 304) {  
NSLog(@"加载本地缓存图片");  
// 如果是，使用本地缓存  
// 根据请求获取到`被缓存的响应`！  
NSCachedURLResponse \*cacheResponse = [[NSURLCache sharedURLCache] cachedResponseForRequest:request];  
// 拿到缓存的数据  
data = cacheResponse.data;  
}  
// 获取并且纪录 etag，区分大小写  
// self.etag = httpResponse.allHeaderFields[@"Etag"];  
// 获取并且纪录 LastModified  
self.localLastModified = httpResponse.allHeaderFields[@"Last-Modified"];  
// NSLog(@"%@", self.etag);  
NSLog(@"%@", self.localLastModified);  
dispatch\_async(dispatch\_get\_main\_queue(), ^{  
!completion ?: completion(data);  
});  
}] resume];  
}

ETag 是什么？

HTTP 协议规格说明定义ETag为“被请求变量的实体值” （参见 —— 章节 14.19）。 另一种说法是，ETag是一个可以与Web资源关联的记号（token）。它是一个 hash 值，用作 Request 缓存请求头，每一个资源文件都对应一个唯一的 ETag 值，  
服务器单独负责判断记号是什么及其含义，并在HTTP响应头中将其传送到客户端，以下是服务器端返回的格式：  
ETag: "50b1c1d4f775c61:df3"

客户端的查询更新格式是这样的：

If-None-Match: W/"50b1c1d4f775c61:df3"  
其中：

If-None-Match - 与响应头的 Etag 相对应，可以判断本地缓存数据是否发生变化  
如果ETag没改变，则返回状态304然后不返回，这也和Last-Modified一样。  
总结下来它的结构如下：

|  |  |
| --- | --- |
| 请求HeaderValue | 响应HeaderValue |
| ETag | If-None-Match |

ETag 是的功能与 Last-Modified 类似：服务端不会每次都会返回文件资源。客户端每次向服务端发送上次服务器返回的 ETag 值，服务器会根据客户端与服务端的 ETag 值是否相等，来决定是否返回 data，同时总是返回对应的 HTTP 状态码。客户端通过 HTTP 状态码来决定是否使用缓存。比如：服务端与客户端的 ETag 值相等，则 HTTP 状态码为 304，不返回 data。服务端文件一旦修改，服务端与客户端的 ETag 值不等，并且状态值会变为200，同时返回 data。  
  
因为修改资源文件后该值会立即变更。这也决定了 ETag 在断点下载时非常有用。  
比如 AFNetworking 在进行断点下载时，就是借助它来检验数据的。详见在 AFHTTPRequestOperation 类中的用法:  
  
//下载暂停时提供断点续传功能，修改请求的HTTP头，记录当前下载的文件位置，下次可以从这个位置开始下载。  
- (void)pause {  
unsigned long long offset = 0;  
if ([self.outputStream propertyForKey:NSStreamFileCurrentOffsetKey]) {  
offset = [[self.outputStream propertyForKey:NSStreamFileCurrentOffsetKey] unsignedLongLongValue];  
} else {  
offset = [[self.outputStream propertyForKey:NSStreamDataWrittenToMemoryStreamKey] length];  
}  
  
NSMutableURLRequest \*mutableURLRequest = [self.request mutableCopy];  
if ([self.response respondsToSelector:@selector(allHeaderFields)] && [[self.response allHeaderFields] valueForKey:@"ETag"]) {  
//若请求返回的头部有ETag，则续传时要带上这个ETag，  
//ETag用于放置文件的唯一标识，比如文件MD5值  
//续传时带上ETag服务端可以校验相对上次请求，文件有没有变化，  
//若有变化则返回200，回应新文件的全数据，若无变化则返回206续传。  
[mutableURLRequest setValue:[[self.response allHeaderFields] valueForKey:@"ETag"] forHTTPHeaderField:@"If-Range"];  
}  
//给当前request加Range头部，下次请求带上头部，可以从offset位置继续下载  
[mutableURLRequest setValue:[NSString stringWithFormat:@"bytes=%llu-", offset] forHTTPHeaderField:@"Range"];  
self.request = mutableURLRequest;  
  
[super pause];  
}

NSURLConnection Demo

在 -connection:willCacheResponse: 中，cachedResponse 对象会根据 URL 连接返回的结果自动创建。因为 NSCachedURLResponse 没有可变部分，为了改变 cachedResponse 中的值必须构造一个新的对象，把改变过的值传入 –initWithResponse:data:userInfo:storagePolicy:，例如：  
- (NSCachedURLResponse \*)connection:(NSURLConnection \*)connection willCacheResponse:(NSCachedURLResponse \*)cachedResponse  
{ NSMutableDictionary \*mutableUserInfo = [[cachedResponse userInfo] mutableCopy];   
NSMutableData \*mutableData = [[cachedResponse data] mutableCopy];  
NSURLCacheStoragePolicy storagePolicy = NSURLCacheStorageAllowedInMemoryOnly;   
// ...  
return [[NSCachedURLResponse alloc] initWithResponse:[cachedResponse response]   
data:mutableData   
userInfo:mutableUserInfo  
storagePolicy:storagePolicy];  
}  
如果 -connection:willCacheResponse: 返回 nil，回应将不会缓存。  
- (NSCachedURLResponse \*)connection:(NSURLConnection \*)connection willCacheResponse:(NSCachedURLResponse \*)cachedResponse  
{  
return nil;  
}  
如果不实现此方法，NSURLConnection 就简单地使用本来要传入 -connection:willCacheResponse:的那个缓存对象，所以除非你需要改变一些值或者阻止缓存，否则这个代理方法不必实现

一般数据类型借助 Last-Modified 与 ETag 进行缓存  
以上的讨论是基于文件资源，那么对一般的网络请求是否也能应用？  
控制缓存过期时间，无非两种：设置一个过期时间；校验缓存与服务端一致性，只在不一致时才更新。  
一般情况下是不会对 api 层面做这种校验，只在有业务需求时才会考虑做，比如：  
数据更新频率较低，“万不得已不会更新”---只在服务器有更新时才更新，以此来保证2G 等恶略网络环境下，有较好的体验。比如网易新闻栏目，但相反微博列表、新闻列表就不适合。  
业务数据一致性要求高，数据更新后需要服务端立刻展示给用户。客户端显示的数据必须是服务端最新的数据  
有离线展示需求，必须实现缓存策略，保证弱网情况下的数据展示的速度。但不考虑使用缓存过期时间来控制缓存的有效性。  
尽量减少数据传输，节省用户流量  
一些建议：

如果是 file 文件类型，用 Last-Modified 就够了。即使 ETag 是首选，但此时两者效果一致。九成以上的需求，效果都一致。  
如果是一般的数据类型--基于查询的 get 请求，比如返回值是 data 或 string 类型的 json 返回值。那么 Last-Modified 服务端支持起来就会困难一点。因为比如  
你做了一个博客浏览 app ，查询最近的10条博客， 基于此时的业务考虑 Last-Modified 指的是10条中任意一个博客的更改。那么服务端需要在你发出请求后，遍历下10条数据，得到“10条中是否至少一个被修改了”。而且要保证每一条博客表数据都有一个类似于记录 Last-Modified 的字段，这显然不太现实。  
  
如果更新频率较高，比如最近微博列表、最近新闻列表，这些请求就不适合，更多的处理方式是添加一个接口，客户端将本地缓存的最后一条数据的的时间戳或 id 传给服务端，然后服务端会将新增的数据条数返回，没有新增则返回 nil 或 304

1. 网页缓存

使用NSURLCache这个类实现。原理就是大多数的网络请求都会先调用这个类中的cachedResponseForRequest:(NSURLRequest \*)request 这个方法，那我们只要重写这个类，就能达到本地缓存的目的

下面是大致的逻辑

1 判断请求中的request 是不是使用get方法，据资料显示一些本地请求的协议也会进到这个方法里面来，所以在第一部，要把不相关的请求排除掉。

2 判断缓存文件夹里面是否存在该文件，如果存在，继续判断文件是否过期，如果过期，则删除。如果文件没有过期，则提取文件，然后组成NSCacheURLResponse返回到方法当中。

3在有网络的情况下，如果文件夹中不存在该文件，则利用NSConnection这个类发网络请求，再把返回的data和response 数据本地化存储起来，然后组成NSCacheURLResponse返回到方法当中

1. 网页中的资源文件

使用NSURLProtocol实现webview的图片缓存。  
主要思路：  
1.过滤出非WebView发起的请求  
2.过滤出图片资源的请求,查找本地缓存中是否已经存在，如果已经存在则自定义Response，给到URLProtocol，否则进行3  
3.结合SD的做本地缓存  
4.NSOutputStream写入数据  
  
#import "YOURURLCacheProtocol.h"  
#import “SDWebImage/SDImageCache.h”

#import “UIImage+MultiFormat.h”

@interface YOURURLCacheProtocol (Private)

+ (BOOL)checkImageResourceWithURL:(NSURL\*)URL string:(NSString\*)urlString;  
- (BOOL)querySDFileDataWithURLString:(NSString\*)urlString;  
- (BOOL)storeImageData:(NSData \*)data;  
  
@end  
  
@interface YOURURLCacheProtocol ()  
@property (nonatomic, strong) NSURLConnection\* connection;  
@property (nonatomic, strong) NSOutputStream\* outputStream;  
@end  
  
@implementation YOURURLCacheProtocol  
+ (void)load {  
dispatch\_async(dispatch\_get\_global\_queue(0, 0), ^{  
[NSURLProtocol registerClass:self];  
});  
}  
  
+ (BOOL)canInitWithRequest:(NSURLRequest\*)request  
{  
NSURL \*url = request.URL;  
NSString \*urlStirng = request.URL.absoluteString.lowercaseString;  
  
///通过UA 来判断是否UIWebView发起的请求  
NSString\* UA = [request valueForHTTPHeaderField:@"User-Agent"];  
if ([UA containsString:@" AppleWebKit/"] == NO) {  
return NO;  
}  
  
if ([self checkImageResourceWithURL:url string:urlStirng]) {  
return YES;  
}  
return NO;  
}  
  
+ (NSURLRequest\*)canonicalRequestForRequest:(NSURLRequest\*)request  
{  
return request;  
}  
  
+ (BOOL) requestIsCacheEquivalent:(NSURLRequest \*)a toRequest:(NSURLRequest \*)b {  
return YES;  
}  
  
- (void)startLoading  
{  
NSString \*urlSting = self.request.URL.absoluteString;  
if (![self querySDFileDataWithURLString:urlSting]) {  
// 走网络请求  
[self startURLConnection];  
}  
}  
  
- (void)stopLoading  
{  
[self.connection cancel];  
}  
- (void)startURLConnection  
{  
// 防止递归调用  
NSMutableURLRequest \* request = [self.request mutableCopy];  
[NSURLProtocol setProperty:@(YES) forKey:@"protocolKey" inRequest:request];  
NSRunLoop\* runLoop = [NSRunLoop currentRunLoop];  
self.outputStream = [NSOutputStream outputStreamToMemory];  
self.connection = [NSURLConnection connectionWithRequest:request delegate:self];  
[self.connection scheduleInRunLoop:runLoop forMode:NSRunLoopCommonModes];  
[self.outputStream scheduleInRunLoop:runLoop forMode:NSRunLoopCommonModes];  
[self.outputStream open];  
[self.connection start];  
}  
  
- (void)connection:(NSURLConnection\*)connection didReceiveData:(NSData\*)data  
{  
NSUInteger length = [data length];  
while (YES) {  
NSInteger totalNumberOfBytesWritten = 0;  
if ([self.outputStream hasSpaceAvailable]) {  
const uint8\_t\* dataBuffer = (uint8\_t\*)[data bytes];  
  
NSInteger numberOfBytesWritten = 0;  
while (totalNumberOfBytesWritten < (NSInteger)length) {  
numberOfBytesWritten = [self.outputStream write:&dataBuffer[(NSUInteger)totalNumberOfBytesWritten] maxLength:(length - (NSUInteger)totalNumberOfBytesWritten)];  
if (numberOfBytesWritten == -1) {  
break;  
}  
totalNumberOfBytesWritten += numberOfBytesWritten;  
}  
  
break;  
}  
  
if (self.outputStream.streamError) {  
[self.connection cancel];  
[self connection:connection didFailWithError:self.outputStream.streamError];  
return;  
}  
}  
}  
  
- (void)connection:(NSURLConnection\*)connection didFailWithError:(NSError\*)error  
{  
NSData\* responseData = [self.outputStream propertyForKey:NSStreamDataWrittenToMemoryStreamKey];  
if (responseData.length > 0) {  
[[self client] URLProtocol:self didLoadData:responseData];  
}  
[[self client] URLProtocol:self didFailWithError:error];  
  
[self.outputStream close];  
self.outputStream = nil;  
self.connection = nil;  
  
}  
  
- (void)connection:(NSURLConnection\*)connection didReceiveResponse:(NSURLResponse\*)response  
{  
[[self client] URLProtocol:self didReceiveResponse:response cacheStoragePolicy:NSURLCacheStorageNotAllowed];  
}  
  
- (void)connectionDidFinishLoading:(NSURLConnection\*)connection  
{  
///请求回来的数据  
NSData\* responseData = [self.outputStream propertyForKey:NSStreamDataWrittenToMemoryStreamKey];  
[self storeImageData:responseData];  
  
  
///回调给webView  
[[self client] URLProtocol:self didLoadData:responseData];  
[[self client] URLProtocolDidFinishLoading:self];  
  
[self.outputStream close];  
self.outputStream = nil;  
self.connection = nil;  
}  
- (NSCachedURLResponse \*)connection:(NSURLConnection \*)connection willCacheResponse:(NSCachedURLResponse \*)cachedResponse  
{  
if (self.request.cachePolicy == NSURLRequestReloadIgnoringLocalCacheData) {  
return nil;  
}  
else {  
return cachedResponse;  
}  
}  
-(BOOL)connectionShouldUseCredentialStorage:(NSURLConnection \*)connection  
{  
return YES;  
}  
- (void)connection:(NSURLConnection \*)connection didReceiveAuthenticationChallenge:(NSURLAuthenticationChallenge \*)challenge  
{  
[[self client] URLProtocol:self didReceiveAuthenticationChallenge:challenge];  
}  
- (void)connection:(NSURLConnection \*)connection didCancelAuthenticationChallenge:(NSURLAuthenticationChallenge \*)challenge  
{  
[[self client] URLProtocol:self didCancelAuthenticationChallenge:challenge];  
}  
  
@end  
  
@implementation YOURURLCacheProtocol (Private)  
  
// 校验图片资源  
+ (BOOL)checkImageResourceWithURL:(NSURL\*)URL string:(NSString\*)urlString  
{  
static NSRegularExpression\* imageExpression;  
static dispatch\_once\_t onceToken;  
dispatch\_once(&onceToken, ^{  
  
NSArray\* imageExtension = @[ @"jpg", @"jpeg", @"gif", @"png", @"webp", @"bmp", @"tif" ];  
NSMutableString\* mutableString = [[NSMutableString alloc] init];  
for (NSString\* extension in imageExtension) {  
if (mutableString.length > 0) {  
[mutableString appendString:@"|"];  
}  
[mutableString appendString:@"\\."];  
[mutableString appendString:extension];  
}  
NSString\* pathExtension = [[NSString alloc] initWithFormat:@"((%@)\\b)|((%@)$)", mutableString, mutableString];  
imageExpression = [NSRegularExpression regularExpressionWithPattern:pathExtension options:NSRegularExpressionCaseInsensitive error:nil];  
});  
  
BOOL hasScriptExtension = ([imageExpression firstMatchInString:urlString options:0 range:NSMakeRange(0, urlString.length)] != nil);  
return hasScriptExtension;  
}  
  
// 查询本地缓存  
- (BOOL)querySDFileDataWithURLString:(NSString\*)urlString   
{  
NSString\* filePath = [[SDImageCache sharedImageCache] defaultCachePathForKey:urlString];  
if ([[NSFileManager defaultManager] fileExistsAtPath:filePath]) {  
///读取sd 的文件缓存  
NSData\* data = [NSData dataWithContentsOfFile:filePath];  
if (data.length > 0) {  
NSURLResponse\* response = [[NSHTTPURLResponse alloc] initWithURL:self.request.URL statusCode:200 HTTPVersion:@"HTTP\_1\_0" headerFields:nil];  
  
///回调给webView  
[[self client] URLProtocol:self didReceiveResponse:response cacheStoragePolicy:NSURLCacheStorageAllowedInMemoryOnly];  
[[self client] URLProtocol:self didLoadData:data];  
[[self client] URLProtocolDidFinishLoading:self];  
  
return YES;  
}  
}  
return NO;  
}  
// 图片本地缓存  
- (BOOL)storeImageData:(NSData \*)data {  
[[SDImageCache sharedImageCache] storeImage:[UIImage sd\_imageWithData:data] forKey:self.request.URL.absoluteString];  
return YES;  
}  
@end

方法二：

1、替换HTML所有的图片地址。  
2、JS取出HTML所有的图片。  
3、跟据滚动的offsetY判断需要加载的图片。  
4、使用OC下载图片并缓存起来转成imgB64。  
5、替换HTML的图片地址。

1、页面数据组成的方式  
首先获取详细的HTML代码并在最后加上JS，然后替换图片地址并设置一个新的属性，这一步是为了加快加载的速度和后续的JS处理。  
// 获取HMTL内容  
- (void)getHTML {  
  
// 读取服务端的HTML  
NSString \*urlStr = @"http://www.crazysurfboy.com/app-1000.html";  
contentHtml = [NSString stringWithContentsOfURL:[NSURL URLWithString:urlStr] encoding: NSUTF8StringEncoding error:nil];  
  
// 读取本地JS文件，把JS加到最后面  
NSString \*jsPath = [[NSBundle mainBundle] pathForResource:@"DetalJavascript" ofType:@"html"];  
NSString\* jsHtml = [NSString stringWithContentsOfFile:jsPath encoding:NSUTF8StringEncoding error:nil];  
  
// 默认加载的图片  
NSString \*imageHTMLString = @"src=\"http://www.crazysurfboy.com/uploads/default\_image\_logo.png\" asrc";  
contentHtml = [contentHtml stringByReplacingOccurrencesOfString:@"src" withString:imageHTMLString];  
contentHtml = [contentHtml stringByAppendingString:jsHtml];  
  
[self.htmlWebView loadHTMLString:contentHtml baseURL:nil];  
}  
通过产品id获取整个线路的HMTL加载到 UIWebView 再通过JS的方法得到HTML的总高度。此时headerView已添加到了 detailScrollView 占300高度，此时只需要把UIWebView添加进去设置好y值，然后添加到 detailScrollView，再重新设置 detailScrollView.contentSize

2、WebViewJavascriptBridge  
参考资料：WebViewJavascriptBridge使用  
以往简单的操作通常用webView:shouldStartLoadWithRequest:navigationType: 和 stringByEvaluatingJavaScriptFromString: 进行JS和OC互相调用。对于相对复杂的操作也许就没有这么方便了。所以我们会使用开源库 WebViewJavascriptBridge  
GitHub:WebViewJavascriptBridge  
OC端：  
@property WebViewJavascriptBridge\* bridge;  
在viewDidLoad 初始化代码如下，在初始化中直接包含了一个用于接收JS的回调：  
// 这一段初始化都是一样的  
self.bridge = [WebViewJavascriptBridge bridgeForWebView:self.htmlWebView webViewDelegate:self handler:^(id data, WVJBResponseCallback responseCallback) {  
DLog(@"ObjC received message from JS: %@", data);  
responseCallback(@"Response for message from ObjC");  
}];  
  
// 这一段是将从JS返回的图片数组到OC中下载并缓存起来  
[self.bridge registerHandler:@"ImageURLObjcCallback" handler:^(id data, WVJBResponseCallback responseCallback) {  
[self downloadAllImagesInNative:data];  
}];

JS端：  
// 以下是固定写法，你自己的JS文件中必须包含如下代码  
function connectWebViewJavascriptBridge(callback) {  
if (window.WebViewJavascriptBridge) {  
callback(WebViewJavascriptBridge)  
} else {  
document.addEventListener('WebViewJavascriptBridgeReady', function() {  
callback(WebViewJavascriptBridge)  
}, false)  
}  
}  
  
// 注册相关的回调  
connectWebViewJavascriptBridge(function(bridge) {  
bridge.init(function(message, responseCallback) {  
log('JS got a message', message)  
var data = { 'Javascript Responds':'Wee!' }  
log('JS responding with', data)  
responseCallback(data)  
})  
  
// 查找可见区域内的图片URL  
bridge.registerHandler('findVisibleImageUrlHandler', function(data, responseCallback) {  
findVisibleImageUrl(data.contentOffsetY, data.offsetHeight);  
})  
  
// 把URL换成cache的URL  
bridge.registerHandler('imagesDownloadCompleteHandler', function(data, responseCallback) {  
imagesDownloadComplete(data[0], data[1]);  
})  
  
// 查找可见区域内的图片URL  
bridge.registerHandler('ScrollViewDidScrollHandler', function(data, responseCallback) {  
//alert("aaaa");  
getScrollHeight(data.contentOffsetY);  
})  
})   
  
  
// 这一段其实是网上常用的图片延迟加载方法  
// 就算你不使用图片缓存，只是简单的把图片地址替换回去同样可以加快很多的速度  
var v = {  
eleGroup: null, // 图片数组  
limitHeight: 0, // 偏移量  
imageUrlsArray: null, // url数组  
willCompleteImageArray: null, // 可见高度也就是读取图片高度，如果你内容不会翻到好几页，大可以不需要。  
isGetScrollHeight:false  
}  
  
  
/\*\*  
\* 获取所有img标签URL的数据  
\*/  
function getAllImageURL() {  
  
// 获取图片标签并转化为数组  
v.eleGroup = document.querySelectorAll("img");  
v.eleGroup = Array.prototype.slice.call(v.eleGroup, 0);  
v.replaceImageArray =   
v.imageUrlsArray = new Array();  
  
// 把URL塞到数组  
v.eleGroup.forEach(function(image) {  
var esrc = image.getAttribute("asrc");  
v.imageUrlsArray.push(esrc);  
});  
return  
}  
  
  
/\*\*  
\* 找出两倍屏幕大小的可见的图片的对象  
\* 由于我们内容往往很长，以用户滚动的距离去加载图片，这一步可以减轻服务器负担也可以为用户省流量。  
\*  
\* @param float scrollView.contentOffset.y  
\* @param float 偏移高度 - 原生 + UIWebView  
\*   
\*/  
function findVisibleImageUrl(contentOffsetY, offsetHeight) {  
  
// 初始化  
offsetHeight = Number(offsetHeight);  
contentOffsetY = Number(contentOffsetY);  
var tempImageUrlsArray = new Array();  
var sedImageUrlsArray = new Array();  
var tempEleGroup = new Array();  
  
// 滚动到UIWebView才开始查找图片位置  
if ( (v.limitHeight == 0) || (contentOffsetY > v.limitHeight)) {  
v.limitHeight = Number(v.limitHeight) + offsetHeight;  
}  
else {  
return;  
}  
  
// 根据仿移量，定位图像是否在预设的可见区域，并且减去已经显示过的图片，减小循环次数  
for (var i = 0, j = v.eleGroup.length; i < j; i++) {  
  
// 判断位置并添加到数组，传回到OC  
if (v.eleGroup[i].offsetTop <= v.limitHeight) {  
console.log("contentOffsetY:" + contentOffsetY + " v.limitHeight:" + v.limitHeight + " v.eleGroup[i].offsetTop:" + v.eleGroup[i].offsetTop);  
sedImageUrlsArray.push(v.imageUrlsArray[i]);  
}  
else {  
tempImageUrlsArray.push(v.imageUrlsArray[i]);  
tempEleGroup.push(v.eleGroup[i]);  
}  
}  
  
v.imageUrlsArray = tempImageUrlsArray;  
v.eleGroup = tempEleGroup;  
  
// 把显示区域的图像数组返回去OC  
bridgeImageURLCallback(sedImageUrlsArray);  
}  
  
  
/\*\*  
\* 发送收到的URL到OC  
\*  
\* @param str 图像的地址  
\*/  
function bridgeImageURLCallback(imageURL) {  
  
connectWebViewJavascriptBridge(function(bridge) {  
  
bridge.callHandler('ImageURLObjcCallback', imageURL, function(response) {  
//log('JS got response', response)  
});  
  
})  
}  
/\*\*  
\* 将图片的URL 换成 cache 的URL  
\*  
\* @param dic 要修改的字典  
\*   
\* @return NSInter类型的时间戳  
\*/  
function imagesDownloadComplete(pOldUrl, pNewUrl) {   
  
// 读取未替换的图片  
var tempImageUrlsArray = new Array();  
if (v.willCompleteImageArray == null) {  
v.willCompleteImageArray = document.querySelectorAll("img");  
v.willCompleteImageArray = Array.prototype.slice.call(v.willCompleteImageArray, 0);  
}   
  
// 查找并替换图像(Base64String)  
for (var i = 0, j = v.willCompleteImageArray.length; i < j; i++) {  
if (v.willCompleteImageArray[i].getAttribute("asrc") == pOldUrl || v.willCompleteImageArray[i].getAttribute("asrc") == decodeURIComponent(pOldUrl)) {  
v.willCompleteImageArray[i].src = pNewUrl;  
}  
else {  
tempImageUrlsArray.push(v.willCompleteImageArray[i]);  
}  
}  
  
v.willCompleteImageArray = tempImageUrlsArray;  
}  
  
// 加载完HTML，获取所有的图像  
getAllImageURL();

3、OC与JS交互  
UIWebView加载完HMTL会调用 webViewDidFinishLoad: 告诉我们已经加载完毕。而此时所有的图片都是我们设置的默认图片，加载起来相当的快。  
// Sent after a web view finishes loading a frame.  
- (void)webViewDidFinishLoad:(UIWebView \*)webView {  
  
// offsetHeight 如果设为1000，那么当contentOffsetY为1001的时候，offsetHeight为 2001  
[self.bridge callHandler:@"findVisibleImageUrlHandler" data:@{ @"contentOffsetY":@"1", @"offsetHeight":@"1000" }];  
}  
由于我禁用了UIWebView的滚动，所以必须要手动的传回给JS  
- (void)scrollViewDidScroll:(UIScrollView \*)scrollView {  
  
NSString \*contentOffsetY = [NSString stringWithFormat:@"%.f", scrollView.contentOffset.y];  
[self.bridge callHandler:@"findVisibleImageUrlHandler" data:@{ @"contentOffsetY":contentOffsetY, @"offsetHeight":@"1000" }];  
当bridgeImageURLCallback这个JS被调后呢，会调用以下方法：  
// 使用OC把URL下载到本地  
-(void)downloadAllImagesInNative:(NSString \*)imageURL {  
  
SDWebImageManager \*manager = [SDWebImageManager sharedManager];  
  
  
NSString \*\_url = imageURL;  
  
[manager downloadImageWithURL:[NSURL URLWithString:\_url] options:SDWebImageHighPriority progress:nil completed:^(UIImage \*image, NSError \*error, SDImageCacheType cacheType, BOOL finished, NSURL \*imageURL) {  
  
if (image) {  
  
dispatch\_async(dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_DEFAULT, 0), ^{  
  
NSString \*imgB64 = [UIImageJPEGRepresentation(image, 1.0) base64EncodedStringWithOptions:NSDataBase64Encoding64CharacterLineLength];  
  
// 把图片在磁盘中的地址传回给JS  
NSString \*key = [manager cacheKeyForURL:imageURL];  
//DLog(@"key:%@", key);  
  
NSString \*source = [NSString stringWithFormat:@"data:image/png;base64,%@", imgB64];  
[self.bridge callHandler:@"imagesDownloadCompleteHandler" data:@[key,source]];  
  
});  
  
}  
  
}];  
  
}  
从SDWebImageManager 缓存的图片转成imgB64就能实现UIWebView缓存。不过同时也带来了一个问题，因为imgB64转成字符它还是那么大，如果一个页面的图片同时塞进HTML，那么这个HTML可能会有几十M，所以如何使用还是看情况。在不能改动版面的情况下，我现在并没有做SDWebImageManager缓存，仅仅是重新的替换URL，但也能加快加载的速度也在寻找解决的方案。

# Cookie

Cookie的读取  
NSArray \*cookies = [[NSHTTPCookieStorage sharedHTTPCookieStorage] cookies];   
NSHTTPCookie \*cookie;  
for (id c in cookies){  
if ([c isKindOfClass:[NSHTTPCookie class]]) {  
cookie=(NSHTTPCookie \*)c;  
NSLog(@"%@: %@", cookie.name, cookie.value); }  
}

Cookie的赋值

NSHTTPCookieValue，NSHTTPCookieName，NSHTTPCookieDomain，NSHTTPCookiePath是必选的

NSMutableDictionary \*cookieProperties = [NSMutableDictionary dictionary];

[cookieProperties setObject:@"test" forKey:NSHTTPCookieValue];

[cookieProperties setObject:@"cookiesid" forKey:NSHTTPCookieName];

[cookieProperties setValue:@".baidu.com" forKey:NSHTTPCookieDomain];

[cookieProperties setValue:@"/" forKey:NSHTTPCookiePath];

NSHTTPCookie \*cookie = [[NSHTTPCookie alloc] initWithProperties:cookieProperties];

[[NSHTTPCookieStorage sharedHTTPCookieStorage] setCookie:cookie];

request还可以这样设置cookie

[request setHTTPShouldHandleCookies:YES];

[request setValue:[NSString stringWithFormat:@"%@=%@", [cookie name], [cookie value]] forHTTPHeaderField:@"Cookie"];

注：如果提供的属性是无效的，初始化Cookie对象为nil。

# 内存管理

Native使用的ARC，JS使用的是垃圾回收机制，并且所有的引用是都强引用，不过JS的循环引用，垃圾回收会帮它们打破。JavaScriptCore里面提供的API，正常情况下，Native和JS对象之间内存管理都无需我们去关心  
1、不要在block里面直接使用context，或者使用外部的JSValue对象  
2、Native对象不要用属性直接保存JSValue对象，容易循环引用

3、不要在不同的 JSVirtualMachine 之间传递JS对象。一个 JSVirtualMachine可以运行多个context，由于都是在同一个堆内存和同一个垃圾回收下，所以相互之间传值是没问题的。但是如果在不同的 JSVirtualMachine传值，垃圾回收就不知道他们之间的关系了，可能会引起异常