

iOS 流量监控分析

2018-05-30

| 分类于 [iOS](#)， [iOS进阶](#) |

由于骑手不能随时处在有 WIFI 的状态，流量变成了很敏感的问题，为了精确到每个 API 的流量，进行针对性的优化，开始在我们的 APM 中添加流量监控功能。

本文将记录自己做流量监控方面的总结。其中包括了非常多的踩坑经验，和现有一些方案的缺陷分析，对我来说是一个非常有意义的过程。

干货预警😂请做好读大量代码的准备😂😂😂

一、资料收集

就目前来说，各家大厂基本都有自己的 APM（包括我们公司其实之前也有一套 APM，但是由于各个事业部的需求不同，尚不能完全满足物流平台的需要）但各家大厂目前开源的 APM 项目却不多，当然也可能是由于各家的业务场景差异比较大且对数据的后续处理不同。

所以本次在查阅资料阶段，没有太多的源码可选参考，但有不少文章。

以下是一些本次开发过程中参考的文章和开源库：

1. [iOS-Monitor-Platform](#)
2. [GodEye](#)

3. [NetworkEye](#)
4. [移动端性能监控方案Hertz](#)
5. [使用NSURLProtocol注意的一些问题](#)
6. [iOS 开发中使用 NSURLProtocol 拦截 HTTP 请求](#)
7. [获取NSURLResponse的HTTPVersion](#)

但以上这些资料对我们的需求都有不足之处：

1. Request 和 Response 记在同一条记录

在实际的网络请求中 Request 和 Response 不一定是成对的，如果网络断开、或者突然关闭进程，都会导致不成对现象，如果将 Request 和 Response 记录在同一条数据，将会对统计造成偏差

2. 上行流量记录不精准

主要的原因有三大类：

1. 直接忽略了 **Header** 和 **Line** 部分
2. 忽略了 **Cookie** 部分，实际上，臃肿的 Cookie 也是消耗流量的一部分
3. body 部分的字节大小计算直接使用了 `HTTPBody.length` 不够准确

3. 下行流量记录不精准

主要原因有：

1. 直接忽略了 **Header** 和 **Status-Line** 部分
2. body 部分的字节大小计算直接使用了 `expectedContentLength` 不够准确

3. 忽略了 **gzip** 压缩，在实际网络编程中，往往都使用了 gzip 来进行数据压缩，而系统提供的一些监听方法，返回的 NSData 实际是解压过的，如果直接统计字节数会造成大量偏差

后文将详细讲述。

二、需求

先简单罗列我们的需求：

- 1. Request 基本信息记录
- 2. 上行流量
- 3. Reponse 基本信息记录
- 4. 下行流量
- 5. 数据归类：按照 host 和 path 归类，一条记录记载改 host/path 的 **Request** 记录数，**Response** 记录数，**Reqeust** 总流量（上行流量），**Reponse** 总流量（下行流量）

我们的侧重点是流量统计，为了方便分析 APP 使用中哪些 API 消耗流量多。所以对上行、下行流量都需要尽量准确记录。

最终的数据库表展示：

logId	host	path	length	lineLength	headerLength	bodyLength	type	
C880426	mobile-gui01.dmc.ali.net	/collect/log	609	27	257	325	0	
5247673f	mobile-gui01.dmc.ali.net	/collect/log	236	15	221	0	1	
7DF2D2C	10.101.82.80	/report	1024	22	257	745	0	
953F128f	10.101.82.80	/report	112	15	93	4	1	
A878575f	knight.ali.net	/knight/sider/count	1135	33	1102	0	0	
6BBA973	knight.ali.net	/knight/sider/count	319	15	198	106	1	
CAC227A	knight.ali.net	/knight/work_status	1135	33	1102	0	0	
8A02B2A	knight.ali.net	/knight/work_status	375	15	198	162	1	
18844F2f	cityreport.ali.net	/track/addpoints	486	31	455	0	0	

type 字段表示的是『该条记录是 Request 还是 Response』，几个 length 分别记录了流量的各个细节，包括：总字节数、Line 字节数、Header 字节数、Body 字节数。

最后的界面展示类似于：



三、分析现有资料

现在分析一下上面收集到的资料有哪些不足之处。

[GodEye](#) | [NetworkEye](#):

NetworkEye 是 GodEye 的一部分，可以单独拆出来使用的网络监控库。

查阅两者的源码后发现，NetworkEye

1. 仅仅记录了 Reponse 的流量
2. 通过 expectedContentLength 记录是不准确的（后面将会说到）
3. 仅仅记录了总和，这对我们来说是无意义的，不能分析出哪条 API 流量使用多

```
//MARK: - NetworkEye
extension ConsoleController: NetworkEyeDelegate {
    /// god's network eye callback
    func networkEyeDidCatch(with request:URLRequest?,response:URLResponse?,data:Data?) {
        Store.shared.addNetworkByte(response?.expectedContentLength ?? 0)
        let model = NetworkRecordModel(request: request, response: response as? HTTPURLResponse, data: data)

        model.insert(complete: { [unowned self] (success:Bool) in
            self.addRecord(model: model)
        })
    }
}
```

[移动端性能监控方案Hertz](#):

美团的文章中展示几个代码片段：

```
1  - (void)connectionDidFinishLoading:(NSURLConnection *)connection
2  {
3      [self.client URLProtocolDidFinishLoading:self];
4
5      self.data = nil;
6      if (connection.originalRequest) {
7          WMNetworkUsageDataInfo *info = [[WMNetworkUsageDataInfo alloc]
8          self.connectionEndTime = [[NSDate date] timeIntervalSince1970]
9          info.responseSize = self.responseDataLength;
10         info.requestSize = connection.originalRequest.HTTPBody.length;
11         info.contentType = [WMNetworkUsageURLProtocol getContentTypeBy
```

```
12 [[WMNetworkMeter sharedInstance] setLastDataInfo:info];  
13 [[WMNetworkUsageManager sharedInstance] recordNetworkUsageDataInfo:  
14 }
```

在 `connectionDidFinishLoading` 中记录了整个网络请求结束的时间、`response` 数据大小、`request` 数据大小以及一些其他数据。

总体来说是比较详细的，但是这里并没有给出 `self.responseDataLength` 的具体逻辑，另外 `connection.originalRequest.HTTPBody.length` 仅仅是 Request body 的大小。

[iOS-Monitor-Platform](#):

这篇文章比较详细的介绍了整个 APM 制作的过程，贴出了很多代码段，应该说非常详细也极具参考价值。

在流量部分，也分别针对了上行流量、下行流量进行了区分，但其中：

1. 没有处理 gzip 压缩情况
2. 对 Header 计算大小的方式是 Dictionary 转 NSData，然而实际上头部并不是 Json 格式（这块我觉得很迷，因为作者特意展示了 HTTP 报文组成）

四、动手自己做

HTTP 报文

为了更好的让大家了解 HTTP 流量计算的一些关键信息，首先要了解 HTTP 报文的组成。

[Apple 文档](#)中对 NSURLProtocol 有非常详细的描述和使用介绍

An abstract class that handles the loading of protocol-specific URL data.

如果想更详细的了解 NSURLProtocol，也可以看大佐的[这篇文章](#)

在每一个 HTTP 请求开始时，URL 加载系统创建一个合适的 NSURLProtocol 对象处理对应的 URL 请求，而我们需要做的就是写一个继承自 NSURLProtocol 的类，并通过 - registerClass: 方法注册我们的协议类，然后 URL 加载系统就会在请求发出时使用我们创建的协议对象对该请求进行处理。

NSURLProtocol 是一个抽象类，需要做的第一步就是集成它，完成我们的自定义设置。

创建自己的 DMURLProtocol，为它添加几个属性并实现相关接口：

```
1 @interface DMURLProtocol() <NSURLConnectionDelegate, NSURLConnectionDat
2
3 @property (nonatomic, strong) NSURLConnection *connection;
4 @property (nonatomic, strong) NSURLRequest *dm_request;
5 @property (nonatomic, strong) NSURLResponse *dm_response;
6 @property (nonatomic, strong) NSMutableData *dm_data;
7
8 @end
```

canInitWithRequest & canonicalRequestForRequest:

```
1 static NSString *const DMHTTP = @"LPDHTTP";
```

```
1 + (BOOL)canInitWithRequest:(NSURLRequest *)request {  
2     if (![request.URL.scheme isEqualToString:@"http"]) {  
3         return NO;  
4     }  
5  
6     if ([NSURLProtocol propertyForKey:LPDHTTP inRequest:request] ) {  
7         return NO;  
8     }  
9     return YES;  
10 }
```

```
1 + (NSURLRequest *)canonicalRequestForRequest:(NSURLRequest *)request {  
2     NSMutableURLRequest *mutableReqeust = [request mutableCopy];  
3     [NSURLProtocol setProperty:@YES  
4         forKey:DMHTTP  
5         inRequest:mutableReqeust];  
6     return [mutableReqeust copy];  
7 }
```

startLoading:

```
1 - (void)startLoading {  
2     NSURLRequest *request = [[self class] canonicalRequestForRequest:self.request];  
3     self.connection = [[NSURLConnection alloc] initWithRequest:request];  
4     self.dm_request = request;  
5 }
```

didReceiveResponse:

```
1 - (void)connection:(NSURLConnection *)connection didReceiveResponse:(NSURLResponse *)response {  
2     [[self client] URLProtocol:self didReceiveResponse:response cacheStorage:nil];  
3     self.dm_response = response;  
4 }
```

didReceiveData:

```
1 - (void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data {
```

```
2 [self.client URLProtocol:self didLoadData:data];  
3 [self.dm_data appendData:data];  
4 }
```

以上部分是为了在单次 HTTP 请求中记录各个所需要属性。

记录 Response 信息

前面的代码实现了在网络请求过程中为 `dm_response` 和 `dm_data` 赋值，那么在 `stopLoading` 方法中，就可以分析 `dm_response` 和 `dm_data` 对象，获取下行流量等相关信息。

需要说明的是，如果需要获得非常精准的流量，一般来说只有通过 Socket 层获取是最准确的，因为可以获取包括握手、挥手的数据大小。当然，我们的目的是为了分析 App 的耗流量 API，所以仅从应用层去分析也基本满足了我们的需要。

上文中说到了报文的组成，那么按照报文所需要的内容获取。

Status Line

非常遗憾的是 `NSURLResponse` 没有接口能直接获取报文中的 Status Line，甚至连 HTTP Version 等组成 Status Line 内容的接口也没有。

最后，我通过转换到 `CFNetwork` 相关类，才拿到了 Status Line 的数据，这其中可能涉及到了读取私有 API

这里我为 `NSURLResponse` 添加了一个扩展：`NSURLResponse+DoggerMonitor`，并为其添加 `statusLineFromCF` 方法

```
1  typedef CFHTTPMessageRef (*DMURLResponseGetHTTPResponse)(CFURLRef resp
2
3  - (NSString *)statusLineFromCF {
4      NSURLResponse *response = self;
5      NSString *statusLine = @"";
6
7      NSString *funName = @"CFURLResponseGetHTTPResponse";
8      DMURLResponseGetHTTPResponse originURLResponseGetHTTPResponse =
9      dlsym(RTLD_DEFAULT, [funName UTF8String]);
10
11     SEL theSelector = NSSelectorFromString(@"_CFURLResponse");
12     if ([response respondsToSelector:theSelector] &&
13         NULL != originURLResponseGetHTTPResponse) {
14
15         CTypeRef cfResponse = CFBridgingRetain([response performSelec
16         if (NULL != cfResponse) {
17
18             CFHTTPMessageRef messageRef = originURLResponseGetHTTPResp
19             statusLine = (__bridge_transfer NSString *)CFHTTPMessageCc
20             CFRelease(cfResponse);
21         }
22     }
23     return statusLine;
24 }
```

通过调用私有 API `_CFURLResponse` 获得 `CTypeRef` 再转换成 `CFHTTPMessageRef`，获取 Status Line。

再将其转换成 `NSData` 计算字节大小：

```
1  - (NSUInteger)dm_getLineLength {
2      NSString *lineStr = @"";
3      if ([self isKindOfClass:[NSHTTPURLResponse class]]) {
4          NSHTTPURLResponse *httpResponse = (NSHTTPURLResponse *)self;
5          lineStr = [self statusLineFromCF];
6      }
7      NSData *lineData = [lineStr dataUsingEncoding:NSUTF8StringEncoding]
8      return lineData.length;
```

```
9 }
}
```

Header

通过 `httpResponse.allHeaderFields` 拿到 Header 字典，再拼接成报文的 `key: value` 格式，转换成 NSData 计算大小：

```
1  - (NSUInteger)dm_getHeadersLength {
2      NSUInteger headersLength = 0;
3      if ([self isKindOfClass:[NSHTTPURLResponse class]]) {
4          NSHTTPURLResponse *httpResponse = (NSHTTPURLResponse *)self;
5          NSDictionary<NSString *, NSString *> *headerFields = httpResponse.allHeaderFields;
6          NSString *headerStr = @"";
7          for (NSString *key in headerFields.allKeys) {
8              headerStr = [headerStr stringByAppendingString:key];
9              headerStr = [headerStr stringByAppendingString:@": "];
10             if ([headerFields objectForKey:key]) {
11                 headerStr = [headerStr stringByAppendingString:headerFields[key]];
12             }
13             headerStr = [headerStr stringByAppendingString:@"\n"];
14         }
15         NSData *headerData = [headerStr dataUsingEncoding:NSUTF8StringEncoding];
16         headersLength = headerData.length;
17     }
18     return headersLength;
19 }
```

Body

对于 Body 的计算，上文看到有些文章里采用的 `expectedContentLength` 或者去 `NSURLResponse` 对象的 `allHeaderFields` 中获取 `Content-Length` 值，其实都不够准确。

首先 API 文档中对 `expectedContentLength` 也有介绍是不准确的：

```
/*!
 * @abstract Returns the expected content length of the receiver.
 * @discussion Some protocol implementations report a content length
 * as part of delivering load metadata, but not all protocols
 * guarantee the amount of data that will be delivered in actuality.
 * Hence, this method returns an expected amount. Clients should use
 * this value as an advisory, and should be prepared to deal with
 * either more or less data.
 * @result The expected content length of the receiver, or -1 if
 * there is no expectation that can be arrived at regarding expected
 * content length.
 */
```

其次，HTTP 1.1 标准里也有介绍 Content-Length 字段不一定是每个 Response 都带有的，最重要的是，**Content-Length** 只是表示 **Body** 部分的大小。

我的方式是，在前面代码中有写到，在 `didReceiveData` 中对 `dm_data` 进行了赋值

`didReceiveData:`

```
1 - (void)connection:(NSURLConnection *)connection didReceiveData:(NSData*)data {
2     [self.client URLProtocol:self didLoadData:data];
3     [self.dm_data appendData:data];
4 }
```

那么在 `stopLoading` 方法中，就可以拿到本次网络请求接收到的数据。

但需要注意对 `gzip` 情况进行区别分析。我们知道 HTTP 请求中，客户端在发送请求的时候会带上 **Accept-Encoding**，这个字段的值将会告知服务器客户端能够理解的内容压缩算法。而服务器进行相应时，会在 Response 中添加 **Content-Encoding** 告知客户端选中的压缩算法。

所以，我们在 `stopLoading` 中获取 `Content-Encoding`，如

果使用了 gzip, 则模拟一次 gzip 压缩, 再计算字节大小:

```
1  - (void)stopLoading {
2      [self.connection cancel];
3
4      DMNetworkTrafficLog *model = [[DMNetworkTrafficLog alloc] init];
5      model.path = self.request.URL.path;
6      model.host = self.request.URL.host;
7      model.type = DMNetworkTrafficDataTypeResponse;
8      model.lineLength = [self.dm_response dm_getLineLength];
9      model.headerLength = [self.dm_response dm_getHeadersLength];
10     if ([self.dm_response isKindOfClass:[NSHTTPURLResponse class]]) {
11         NSHTTPURLResponse *httpResponse = (NSHTTPURLResponse *)self.dm
12         NSData *data = self.dm_data;
13         if ([[httpResponse.allHeaderFields objectForKey:@"Content-Enc
14
15             data = [self.dm_data gzippedData];
16         }
17         model.bodyLength = data.length;
18     }
19     model.length = model.lineLength + model.headerLength + model.bodyL
20     [model settingOccurTime];
21     [[DMDataManager defaultManager] addNetworkTrafficLog:model];
22 }
```

这里 gzippedData 参考[这个库的内容](#)

```
[[DMDataManager defaultManager]
```

addNetworkTrafficLog:model]; 是调用持久化层的代码将数据落库。

记录 Request 信息

Line

很遗憾, 对于 `NSURLRequest` 我没有像 `NSURLReponse` 一样幸运的找到私有接口将其转换成 `CFNetwork` 相关数据, 但是我们很清楚 HTTP 请求报文 Line 部分的组成, 所以

我们可以添加一个方法，获取一个经验 Line。

同样为 NSURLReques 添加一个扩

展：NSURLRequest+DoggerMonitor

```
1 - (NSUInteger)dgm_getLineLength {
2     NSString *lineStr = [NSString stringWithFormat:@"%s", self.
3     NSData *lineData = [lineStr dataUsingEncoding:NSUTF8StringEncoding]
4     return lineData.length;
5 }
```

Header

Header 这里有一个非常大的坑。

`request.allHTTPHeaderFields` 拿到的头部数据是有很多缺失的，这块跟业内朋友交流的时候，发现很多人都没有留意到这个问题。

缺失的部分不仅仅是上面一篇文章中说到的 Cookie。

如果通过 Charles 抓包，可以看到，会缺失包括但不限于以下字段：

1. Accept
2. Connection
3. Host

这个问题非常的迷，同时由于无法转换到 CFNetwork 层，所以一直拿不到准确的 Header 数据。

最后，我在 so 上也找到了两个相关问题，供大家参考

NSURLRequest: where an app can find the default headers for HTTP request?

NSMutableURLRequest, cant access all request headers sent out from within my iPhone program

两个问题的回答基本表明了，如果你是通过 CFNetwork 来发起请求的，才可以拿到完整的 Header 数据。

所以这块只能拿到大部分的 Header，但是基本上缺失的都固定是那几个字段，对我们流量统计的精确度影响不是很大。

那么主要就针对 cookie 部分进行补全：

```
1  - (NSUInteger)dgm_getHeadersLengthWithCookie {
2      NSUInteger headersLength = 0;
3
4      NSDictionary<NSString *, NSString *> *headerFields = self.allHTTPHeaderFields;
5      NSDictionary<NSString *, NSString *> *cookiesHeader = [self dgm_getCookiesHeader];
6
7
8      if (cookiesHeader.count) {
9          NSMutableDictionary *headerFieldsWithCookies = [NSMutableDictionary dictionary];
10         [headerFieldsWithCookies addEntriesFromDictionary:cookiesHeader];
11         headerFields = [headerFieldsWithCookies copy];
12     }
13     NSLog(@"%@", headerFields);
14     NSString *headerStr = @"";
15
16     for (NSString *key in headerFields.allKeys) {
17         headerStr = [headerStr stringByAppendingString:key];
18         headerStr = [headerStr stringByAppendingString:@": "];
19         if ([headerFields objectForKey:key]) {
20             headerStr = [headerStr stringByAppendingString:headerFields[key]];
21         }
22         headerStr = [headerStr stringByAppendingString:@"\n"];
23     }
24     NSData *headerData = [headerStr dataUsingEncoding:NSUTF8StringEncoding];
25     headersLength = headerData.length;
26     return headersLength;
27 }
```

```
1 - (NSDictionary<NSString *, NSString *> *)dgm_getCookies {
2     NSDictionary<NSString *, NSString *> *cookiesHeader;
3     NSHTTPCookieStorage *cookieStorage = [NSHTTPCookieStorage sharedHTT
4     NSArray<NSHTTPCookie *> *cookies = [cookieStorage cookiesForURL:self
5     if (cookies.count) {
6         cookiesHeader = [NSHTTPCookie requestHeaderFieldsWithCookies:cc
7     }
8     return cookiesHeader;
9 }
```

body

最后是 **body** 部分，这里也有个坑。通过 **NSURLConnection** 发出的网络请求 **request.HTTPBody** 拿到的是 **nil**。

需要转而通过 **HTTPBodyStream** 读取 **stream** 来获取 **request** 的 **Body** 大小。

```
1 - (NSUInteger)dgm_getBodyLength {
2     NSDictionary<NSString *, NSString *> *headerFields = self.allHTTPH
3     NSUInteger bodyLength = [self.HTTPBody length];
4
5     if ([headerFields objectForKey:@"Content-Encoding"]) {
6         NSData *bodyData;
7         if (self.HTTPBody == nil) {
8             uint8_t d[1024] = {0};
9             NSInputStream *stream = self.HTTPBodyStream;
10            NSMutableData *data = [[NSMutableData alloc] init];
11            [stream open];
12            while ([stream hasBytesAvailable]) {
13                NSInteger len = [stream read:d maxLength:1024];
14                if (len > 0 && stream.streamError == nil) {
15                    [data appendBytes:(void *)d length:len];
16                }
17            }
18            bodyData = [data copy];
19            [stream close];
20        } else {
21            bodyData = self.HTTPBody;
22        }
23        bodyLength = [[bodyData gzippedData] length];
24    }
```

```
24     }
25
26     return bodyLength;
27 }
```

落库

最后在 DMURLProtocol 的 - (nullable NSURLRequest *)connection:(NSURLConnection *)connection willSendRequest:(NSURLRequest *)request redirectResponse:(nullable NSURLResponse *)response; 方法中对 request 调用报文各个部分大小方法后落库：

```
1  -(NSURLRequest *)connection:(NSURLConnection *)connection willSendReq
2      if (response != nil) {
3          self.dm_response = response;
4          [self.client URLProtocol:self wasRedirectedToRequest:request r
5      }
6
7      DMNetworkTrafficLog *model = [[DMNetworkTrafficLog alloc] init];
8      model.path = request.URL.path;
9      model.host = request.URL.host;
10     model.type = DMNetworkTrafficDataTypeRequest;
11     model.lineLength = [connection.currentRequest dgm_getLineLength];
12     model.headerLength = [connection.currentRequest dgm_getHeadersLeng
13     model.bodyLength = [connection.currentRequest dgm_getBodyLength];
14     model.length = model.lineLength + model.headerLength + model.bodyL
15     [model settingOccurTime];
16     [[DMDataManager defaultManager] addNetworkTrafficLog:model];
17     return request;
18 }
```

针对 NSURLSession 的处理

直接使用 DMURLProtocol 并 registerClass 并不能完整的拦截所有网络请求，因为通过 NSURLSession 的 sharedSession 发出的请求是无法被 NSURLProtocol 代理

的。

我们需要让 [NSURLSessionConfiguration defaultSessionConfiguration].protocolClasses 的属性中也设置我们的 DMURLProtocol，这里通过 swizzle，置换 protocolClasses 的 get 方法：

编写一个 DMURLSessionConfiguration

```
1  #import <Foundation/Foundation.h>
2
3  @interface DMURLSessionConfiguration : NSObject
4
5  @property (nonatomic,assign) BOOL isSwizzle;
6  + (DMURLSessionConfiguration *)defaultConfiguration;
7  - (void)load;
8  - (void)unload;
9
10 @end
```

```
1  #import "DMURLSessionConfiguration.h"
2  #import <objc/runtime.h>
3  #import "DMURLProtocol.h"
4  #import "DMNetworkTrafficManager.h"
5
6  @implementation DMURLSessionConfiguration
7
8  + (DMURLSessionConfiguration *)defaultConfiguration {
9      static DMURLSessionConfiguration *staticConfiguration;
10     static dispatch_once_t onceToken;
11     dispatch_once(&onceToken, ^{
12         staticConfiguration=[[DMURLSessionConfiguration alloc] init];
13     });
14     return staticConfiguration;
15
16 }
17
18 - (instancetype)init {
19     self = [super init];
20     if (self) {
21         self.isSwizzle = NO;
22     }
23     return self;
```

```

24 }
25
26 - (void)load {
27     self.isSwizzle = YES;
28     Class cls = NSClassFromString(@"__NSCFURLSessionConfiguration") ?:
29     [self swizzleSelector:@selector(protocolClasses) fromClass:cls to(
30
31 }
32
33 - (void)unload {
34     self.isSwizzle=NO;
35     Class cls = NSClassFromString(@"__NSCFURLSessionConfiguration") ?:
36     [self swizzleSelector:@selector(protocolClasses) fromClass:cls to(
37 }
38
39 - (void)swizzleSelector:(SEL)selector fromClass:(Class)original toClass:
40     Method originalMethod = class_getInstanceMethod(original, selector);
41     Method stubMethod = class_getInstanceMethod(stub, selector);
42     if (!originalMethod || !stubMethod) {
43         [NSException raise:NSInternalInconsistencyException format:@"%C
44     }
45     method_exchangeImplementations(originalMethod, stubMethod);
46 }
47
48 - (NSArray *)protocolClasses {
49
50     return [DMNetworkTrafficManager manager].protocolClasses;
51 }
52
53 @end

```

这样，我们写好了方法置换，在执行过该类单例的 `load` 方法后，`[NSURLSessionConfiguration defaultSessionConfiguration].protocolClasses` 拿到的将会是我们设置好的 `protocolClasses`。

如此，我们再为 `DMURLProtocol` 添加 `start` 和 `stop` 方法，用于启动网络监控和停止网络监控：

```

1 + (void)start {
2     DMURLSessionConfiguration *sessionConfiguration = [DMURLSessionCor
3     for (id protocolClass in [DMNetworkTrafficManager manager].protoc
4         [NSURLProtocol registerClass:protocolClass];

```

```

5     }
6     if (![sessionConfiguration isSwizzle]) {
7
8         [sessionConfiguration load];
9     }
10 }
11
12 + (void)end {
13     DMURLSessionConfiguration *sessionConfiguration = [DMURLSessionCor
14     [NSURLProtocol unregisterClass:[DMURLProtocol class]];
15     if ([sessionConfiguration isSwizzle]) {
16
17         [sessionConfiguration unload];
18     }
19 }

```

到此，基本完成了整个网络流量监控。

再提供一个 Manger 方便使用者调用：

```

1  #import <Foundation/Foundation.h>
2
3  @class DMNetworkLog;
4  @interface DMNetworkTrafficManager : NSObject
5
6
7  @property (nonatomic, strong) NSArray *protocolClasses;
8
9
10
11 + (DMNetworkTrafficManager *)manager;
12
13
14 + (void)startWithProtocolClasses:(NSArray *)protocolClasses;
15
16 + (void)start;
17
18 + (void)end;
19
20 @end

```

```

1  #import "DMNetworkTrafficManager.h"
2  #import "DMURLProtocol.h"
3
4  @interface DMNetworkTrafficManager ()
5

```

```

6  @end
7
8  @implementation DMNetworkTrafficManager
9
10 #pragma mark - Public
11
12 + (DMNetworkTrafficManager *)manager {
13     static DMNetworkTrafficManager *manager;
14     static dispatch_once_t onceToken;
15     dispatch_once(&onceToken, ^{
16         manager=[[DMNetworkTrafficManager alloc] init];
17     });
18     return manager;
19 }
20
21 + (void)startWithProtocolClasses:(NSArray *)protocolClasses {
22     [self manager].protocolClasses = protocolClasses;
23     [DMURLProtocol start];
24 }
25
26 + (void)start {
27     [self manager].protocolClasses = @[[DMURLProtocol class]];
28     [DMURLProtocol start];
29 }
30
31 + (void)end {
32     [DMURLProtocol end];
33 }
34
35 @end

```

五、代码

本文中贴出了比较多的代码，为了便于大家整体观看，可以到 [这里](#) 来阅读。

由于其中包含了一些数据操作的内容不需要关心，所以我直接省略了，虽然没有 Demo，但我相信大家都是能理解整个监控结构的。

六、Other

如果你的 APP 从 iOS 9 支持，可以使用 [NetworkExtension](#)，通过 NetworkExtension 可以通过 VPN 的形式接管整个网络请求，省掉了上面所有的烦恼。

有什么问题都可以在博文后面留言，或者微博上私信我，或者邮件我 coderfish@163.com。

博主是 iOS 妹子一枚。

希望大家一起进步。

我的微博：[周小鱼](#)