# HTTPDNS

## 最佳实践

# 最佳实践

# HTTPS（含SNI）业务场景“IP直连”方案说明

## 1. 背景说明

本文主要介绍HTTPS(含SNI)业务场景下在Android端和iOS端实现“IP直连”的通用解决方案。如果您是Android开发者，并且以OkHttp作为网络开发框架，由于OkHttp提供了自定义DNS服务接口可以优雅地实现IP直连。其方案相比通用方案更加简单且通用性更强，推荐您参考HttpDns+OkHttp最佳实践接入HttpDns。

### 1.1 HTTPS

发送HTTPS请求首先要进行SSL/TLS握手，握手过程大致如下：

1. 客户端发起握手请求，携带随机数、支持算法列表等参数。
2. 服务端收到请求，选择合适的算法，下发公钥证书和随机数。
3. 客户端对服务端证书进行校验，并发送随机数信息，该信息使用公钥加密。
4. 服务端通过私钥获取随机数信息。
5. 双方根据以上交互的信息生成session ticket，用作该连接后续数据传输的加密密钥。

上述过程中，和HTTPDNS有关的是第3步，客户端需要验证服务端下发的证书，验证过程有以下两个要点：

1. 客户端用本地保存的根证书解开证书链，确认服务端下发的证书是由可信任的机构颁发的。
2. 客户端需要检查证书的domain域和扩展域，看是否包含本次请求的host。

如果上述两点都校验通过，就证明当前的服务端是可信任的，否则就是不可信任，应当中断当前连接。

当客户端使用HTTPDNS解析域名时，请求URL中的host会被替换成HTTPDNS解析出来的IP，所以在证书验证的第2步，会出现domain不匹配的情况，导致SSL/TLS握手不成功。

### 1.2 SNI

SNI（Server Name Indication）是为了解决一个服务器使用多个域名和证书的SSL/TLS扩展。它的工作原理如下：

1. 在连接到服务器建立SSL链接之前先发送要访问站点的域名（Hostname）。
2. 服务器根据这个域名返回一个合适的证书。

目前，大多数操作系统和浏览器都已经很好地支持SNI扩展，OpenSSL 0.9.8也已经内置这一功能。

上述过程中，当客户端使用HTTPDNS解析域名时，请求URL中的host会被替换成HTTPDNS解析出来的IP，导致服务器获取到的域名为解析后的IP，无法找到匹配的证书，只能返回默认的证书或者不返回，所以会出现SSL/TLS握手不成功的错误。

> 比如当你需要通过HTTPS访问CDN资源时，CDN的站点往往服务了很多的域名，所以需要通过SNI指定具体的域名证书进行通信。

# 2. HTTPS场景（非SNI）解决方案

针对“domain不匹配”问题，可以采用如下方案解决：hook证书校验过程中第2步，将IP直接替换成原来的域名，再执行证书验证。

【注意】基于该方案发起网络请求，若报出SSL校验错误，比如iOS系统报错kCFStreamErrorDomainSSL, -9813; The certificate for this server is invalid，Android系统报错System.err: javax.net.ssl.SSLHandshakeException: java.security.cert.CertPathValidatorException: Trust anchor for certification path not found.，请检查应用场景是否为SNI（单IP多HTTPS域名）。

下面分别列出Android和iOS平台的示例代码。

## 2.1 Android示例

此示例针对HttpURLConnection接口。

```
try {
String url = "https://140.205.160.59/?sprefer=sypc00";
HttpsURLConnection connection = (HttpsURLConnection) new URL(url).openConnection();

connection.setRequestProperty("Host", "m.taobao.com");
connection.setHostnameVerifier(new HostnameVerifier() {

/*
* 关于这个接口的说明，官方有文档描述：
* This is an extended verification option that implementers can provide.
* It is to be used during a handshake if the URL's hostname does not match the
* peer's identification hostname.
*
* 使用HTTPDNS后URL里设置的hostname不是远程的主机名(如:m.taobao.com)，与证书颁发的域不匹配，
* Android HttpsURLConnection提供了回调接口让用户来处理这种定制化场景。
* 在确认HTTPDNS返回的源站IP与Session携带的IP信息一致后，您可以在回调方法中将待验证域名替换为原来的真实域名进行验证。
*
*/

@Override
public boolean verify(String hostname, SSLSession session) {
return HttpsURLConnection.getDefaultHostnameVerifier().verify("m.taobao.com", session);
```

```
return false;
}
});

connection.connect();
} catch (Exception e) {
e.printStackTrace();
} finally {
}
```

## 2.2 iOS示例

此示例针对NSURLSession/NSURLConnection接口。

```
- (BOOL)evaluateServerTrust:(SecTrustRef)serverTrust
forDomain:(NSString *)domain
{
/*
* 创建证书校验策略
*/
NSMutableArray *policies = [NSMutableArray array];
if (domain) {
[policies addObject:(__bridge_transfer id)SecPolicyCreateSSL(true, (__bridge CFStringRef)domain)];
} else {
[policies addObject:(__bridge_transfer id)SecPolicyCreateBasicX509()];
}

/*
* 绑定校验策略到服务端的证书上
*/
SecTrustSetPolicies(serverTrust, (__bridge CFArrayRef)policies);


/*
* 评估当前serverTrust是否可信任，
* 官方建议在result = kSecTrustResultUnspecified 或 kSecTrustResultProceed
* 的情况下serverTrust可以被验证通过，https://developer.apple.com/library/ios/technotes/tn2232/_index.html
* 关于SecTrustResultType的详细信息请参考SecTrust.h
*/
SecTrustResultType result;
SecTrustEvaluate(serverTrust, &result);

return (result == kSecTrustResultUnspecified || result == kSecTrustResultProceed);
}

/*
* NSURLConnection
*/
- (void)connection:(NSURLConnection *)connection
willSendRequestForAuthenticationChallenge:(NSURLAuthenticationChallenge *)challenge
{
if (!challenge) {
return;
}
```

```
/*
* URL里面的host在使用HTTPDNS的情况下被设置成了IP，此处从HTTP Header中获取真实域名
*/
NSString* host = [[self.request allHTTPHeaderFields] objectForKey:@"host"];
if (!host) {
host = self.request.URL.host;
}

/*
* 判断challenge的身份验证方法是否是NSURLAuthenticationMethodServerTrust（HTTPS模式下会进行该身份验证流程），
* 在没有配置身份验证方法的情况下进行默认的网络请求流程。
*/
if ([challenge.protectionSpace.authenticationMethod isEqualToString:NSURLAuthenticationMethodServerTrust])
{
if ([self evaluateServerTrust:challenge.protectionSpace.serverTrust forDomain:host]) {
/*
* 验证完以后，需要构造一个NSURLCredential发送给发起方
*/
NSURLCredential *credential = [NSURLCredential credentialForTrust:challenge.protectionSpace.serverTrust];
[[challenge sender] useCredential:credential forAuthenticationChallenge:challenge];
} else {
/*
* 验证失败，进入默认处理流程
*/
[[challenge sender] continueWithoutCredentialForAuthenticationChallenge:challenge];
}
} else {
/*
* 对于其他验证方法直接进行处理流程
*/
[[challenge sender] continueWithoutCredentialForAuthenticationChallenge:challenge];
}
}

////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////

/*
* NSURLSession
*/
- (void)URLSession:(NSURLSession *)session task:(NSURLSessionTask *)task
didReceiveChallenge:(NSURLAuthenticationChallenge *)challenge
completionHandler:(void (^)(NSURLSessionAuthChallengeDisposition disposition, NSURLCredential * __nullable
credential))completionHandler
{
if (!challenge) {
return;
}

NSURLSessionAuthChallengeDisposition disposition = NSURLSessionAuthChallengePerformDefaultHandling;
NSURLCredential *credential = nil;
```

```
/*
* 获取原始域名信息。
*/
NSString* host = [[self.request allHTTPHeaderFields] objectForKey:@"host"];
if (!host) {
host = self.request.URL.host;
}

if ([challenge.protectionSpace.authenticationMethod isEqualToString:NSURLAuthenticationMethodServerTrust]) {
if ([self evaluateServerTrust:challenge.protectionSpace.serverTrust forDomain:host]) {
disposition = NSURLSessionAuthChallengeUseCredential;
credential = [NSURLCredential credentialForTrust:challenge.protectionSpace.serverTrust];
} else {
disposition = NSURLSessionAuthChallengePerformDefaultHandling;
}
} else {
disposition = NSURLSessionAuthChallengePerformDefaultHandling;
}
// 对于其他的challenges直接使用默认的验证方案
completionHandler(disposition,credential);
}
```

# 3. HTTPS（SNI）场景方案

## 3.1 Android SNI场景

在HTTPDNS Android Demo中针对HttpsURLConnection接口，提供了在SNI业务场景下使用HTTPDNS的示例代码。

定制SSLSocketFactory，在createSocket时替换为HTTPDNS的IP，并进行SNI/HostNameVerify配置。

```
class TlsSniSocketFactory extends SSLSocketFactory {
private final String TAG = TlsSniSocketFactory.class.getSimpleName();
HostnameVerifier hostnameVerifier = HttpsURLConnection.getDefaultHostnameVerifier();
private HttpsURLConnection conn;

public TlsSniSocketFactory(HttpsURLConnection conn) {
this.conn = conn;
}

@Override
public Socket createSocket() throws IOException {
return null;
}

@Override
public Socket createSocket(String host, int port) throws IOException, UnknownHostException {
return null;
}

@Override
public Socket createSocket(String host, int port, InetAddress localHost, int localPort) throws IOException,
UnknownHostException {
```

```
return null;
}

@Override
public Socket createSocket(InetAddress host, int port) throws IOException {
return null;
}

@Override
public Socket createSocket(InetAddress address, int port, InetAddress localAddress, int localPort) throws
IOException {
return null;
}

// TLS layer

@Override
public String[] getDefaultCipherSuites() {
return new String[0];
}

@Override
public String[] getSupportedCipherSuites() {
return new String[0];
}

@Override
public Socket createSocket(Socket plainSocket, String host, int port, boolean autoClose) throws IOException {
String peerHost = this.conn.getRequestProperty("Host");
if (peerHost == null)
peerHost = host;
Log.i(TAG, "customized createSocket. host: " + peerHost);
InetAddress address = plainSocket.getInetAddress();
if (autoClose) {
// we don't need the plainSocket
plainSocket.close();
}
// create and connect SSL socket, but don't do hostname/certificate verification yet
SSLCertificateSocketFactory sslSocketFactory = (SSLCertificateSocketFactory)
SSLCertificateSocketFactory.getDefault(0);
SSLSocket ssl = (SSLSocket) sslSocketFactory.createSocket(address, port);

// enable TLSv1.1/1.2 if available
ssl.setEnabledProtocols(ssl.getSupportedProtocols());

// set up SNI before the handshake
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN_MR1) {
Log.i(TAG, "Setting SNI hostname");
sslSocketFactory.setHostname(ssl, peerHost);
} else {
Log.d(TAG, "No documented SNI support on Android <4.2, trying with reflection");
try {
java.lang.reflect.Method setHostnameMethod = ssl.getClass().getMethod("setHostname", String.class);
setHostnameMethod.invoke(ssl, peerHost);
} catch (Exception e) {
Log.w(TAG, "SNI not useable", e);
```

```
}
}

// verify hostname and certificate
SSLSession session = ssl.getSession();

if (!hostnameVerifier.verify(peerHost, session))
throw new SSLPeerUnverifiedException("Cannot verify hostname: " + peerHost);

Log.i(TAG, "Established " + session.getProtocol() + " connection with " + session.getPeerHost() +
" using " + session.getCipherSuite());

return ssl;
}
}
```

对于需要设置SNI的站点，通常需要重定向请求，示例中也给出了重定向请求的处理方法。

```
public void recursiveRequest(String path, String reffer) {
URL url = null;
try {
url = new URL(path);
conn = (HttpsURLConnection) url.openConnection();
String ip = httpdns.getIpByHostAsync(url.getHost());
if (ip != null) {
// 通过HTTPDNS获取IP成功，进行URL替换和HOST头设置
Log.d(TAG, "Get IP: " + ip + " for host: " + url.getHost() + " from HTTPDNS successfully!");
String newUrl = path.replaceFirst(url.getHost(), ip);
conn = (HttpsURLConnection) new URL(newUrl).openConnection();
// 设置HTTP请求头Host域
conn.setRequestProperty("Host", url.getHost());
}
conn.setConnectTimeout(30000);
conn.setReadTimeout(30000);
conn.setInstanceFollowRedirects(false);
TlsSniSocketFactory sslSocketFactory = new TlsSniSocketFactory(conn);
conn.setSSLSocketFactory(sslSocketFactory);
conn.setHostnameVerifier(new HostnameVerifier() {
/*
* 关于这个接口的说明，官方有文档描述：
* This is an extended verification option that implementers can provide.
* It is to be used during a handshake if the URL's hostname does not match the
* peer's identification hostname.
*
* 使用HTTPDNS后URL里设置的hostname不是远程的主机名(如:m.taobao.com)，与证书颁发的域不匹配，
* Android HttpsURLConnection提供了回调接口让用户来处理这种定制化场景。
* 在确认HTTPDNS返回的源站IP与Session携带的IP信息一致后，您可以在回调方法中将待验证域名替换为原来的真实域名进行验证。
*
*/
@Override
public boolean verify(String hostname, SSLSession session) {
String host = conn.getRequestProperty("Host");
if (null == host) {
host = conn.getURL().getHost();
```

```
}
return HttpsURLConnection.getDefaultHostnameVerifier().verify(host, session);
}
});
int code = conn.getResponseCode();// Network block
if (needRedirect(code)) {
//临时重定向和永久重定向location的大小写有区分
String location = conn.getHeaderField("Location");
if (location == null) {
location = conn.getHeaderField("location");
}
if (!(location.startsWith("http://") || location
.startsWith("https://"))) {
//某些时候会省略host，只返回后面的path，所以需要补全url
URL originalUrl = new URL(path);
location = originalUrl.getProtocol() + "://"
+ originalUrl.getHost() + location;
}
recursiveRequest(location, path);
} else {
// redirect finish.
DataInputStream dis = new DataInputStream(conn.getInputStream());
int len;
byte[] buff = new byte[4096];
StringBuilder response = new StringBuilder();
while ((len = dis.read(buff)) != -1) {
response.append(new String(buff, 0, len));
}
Log.d(TAG, "Response: " + response.toString());
}
} catch (MalformedURLException e) {
Log.w(TAG, "recursiveRequest MalformedURLException");
} catch (IOException e) {
Log.w(TAG, "recursiveRequest IOException");
} catch (Exception e) {
Log.w(TAG, "unknow exception");
} finally {
if (conn != null) {
conn.disconnect();
}
}
}

private boolean needRedirect(int code) {
return code >= 300 && code < 400;
}
```

## 3.2 iOS SNI场景

SNI（单IP多HTTPS证书）场景下，iOS上层网络库NSURLConnection/NSURLSession没有提供接口进行SNI字段的配置，因此需要Socket层级的底层网络库例如CFNetwork，来实现IP直连网络请求适配方案。而基于CFNetwork的解决方案需要开发者考虑数据的收发、重定向、解码、缓存等问题（CFNetwork是非常底层的网络实现），希望开发者合理评估该场景的使用风险。

方案详情在这篇《HTTPS SNI 业务场景“IP直连”方案说明》 里进行了详细的讨论。

# 如何利用HTTPDNS降低DNS解析开销

## 1. 背景说明

移动场景下DNS的解析开销是整个网络请求延迟中不可忽视的一部分。一方面基于UDP的localDNS解析在高丢包率的移动网络环境下更容易出现解析超时的问题，另一方面在弱网环境下DNS解析所引入的动辄数百毫秒的网络延迟也大幅加重了整个业务请求的负担，直接影响用户的终极体验。

## 2. 解决方案

HTTPDNS在解决了传统域名劫持以及调度精确性的问题的同时，也提供了开发者更灵活的DNS管理方式。通过在客户端合理地应用HTTPDNS管理策略，我们甚至能够做到DNS解析0延迟，大幅提升弱网环境下的网络通讯效率。

DNS解析0延迟的主要思路包括：

- 构建客户端DNS缓存；

通过合理的DNS缓存，我们确保每次网络交互的DNS解析都是从内存中获取IP信息，从而大幅降低DNS解析开销。根据业务的不同，我们可以制订更丰富的缓存策略，如根据运营商缓存，可以在网络切换的场景下复用已缓存的不同运营商线路的域名IP信息，避免网络切换后进行链路重选择引入的DNS网络解析开销。另外，我们还可以引入IP本地化离线存储，在客户端重启时快速从本地读取域名IP信息，大幅提升首页载入效率。

- 热点域名预解析；

在客户端启动过程中，我们可以通过热点域名的预解析完成热点域名的缓存载入。当真正的业务请求发生时，直接由内存中读取目标域名的IP信息，避免传统DNS的网络开销。

- 懒更新策略；

绝大多数场景下业务域名的IP信息变更并不频繁，特别是在单次APP的使用周期内，域名解析获取的IP往往是相同的（特殊业务场景除外）。因此我们可以利用DNS懒更新策略来实现TTL过期后的DNS快速解析。所谓DNS懒更新策略即客户端不主动探测域名对应IP的TTL时间，当业务请求需要访问某个业务域名时，查询内存缓存并返回该业务域名对应的IP解析结果。如果IP解析结果的TTL已过期，则在后台进行异步DNS网络解析与缓存结果更新。通过上述策略，用户的所有DNS解析都在与内存交互，避免了网络交互引入的延迟。

### 2.1 Demo示例

我们在HTTPDNS Demo github中提供了Android/iOS SDK以及HTTPDNS API接口的使用例程，这里我们通

过使用Android SDK的例程演示如何实现0延迟的HTTPDNS服务。

```
public class NetworkRequestUsingHttpDNS {

    private static HttpDnsService httpdns;
    // 填入您的HTTPDNS accoutID信息，您可以从HTTPDNS控制台获取该信息
    private static String accountID = "100000";
    // 您的热点域名
    private static final String[] TEST_URL = {"http://www.aliyun.com", "http://www.taobao.com"};

    public static void main(final Context ctx) {
        try {
            // 设置APP Context和Account ID，并初始化HTTPDNS
            httpdns = HttpDns.getService(ctx, accountID);
            // DegradationFilter用于自定义降级逻辑
            // 通过实现shouldDegradeHttpDNS方法，可以根据需要，选择是否降级
            DegradationFilter filter = new DegradationFilter() {
                @Override
                public boolean shouldDegradeHttpDNS(String hostName) {
                    // 此处可以自定义降级逻辑，例如www.taobao.com不使用HttpDNS解析
                    // 参照HttpDNS API文档，当存在中间HTTP代理时，应选择降级，使用Local DNS
                    return hostName.equals("www.taobao.com") || detectIfProxyExist(ctx);
                }
            };
            // 将filter传进httpdns，解析时会回调shouldDegradeHttpDNS方法，判断是否降级
            httpdns.setDegradationFilter(filter);
            // 设置预解析域名列表，真正使用时，建议您将预解析操作放在APP启动函数中执行。预解析操作为异步行为，不会阻塞您的启动流程
            httpdns.setPreResolveHosts(new ArrayList<>(Arrays.asList("www.aliyun.com", "www.taobao.com")));
            // 允许返回过期的IP，通过设置允许返回过期的IP，配合异步查询接口，我们可以实现DNS懒更新策略
            httpdns.setExpiredIPEnabled(true);

            // 发送网络请求
            String originalUrl = "http://www.aliyun.com";
            URL url = new URL(originalUrl);
            HttpURLConnection conn = (HttpURLConnection) url.openConnection();
            // 异步接口获取IP，当IP TTL过期时，由于采用DNS懒更新策略，我们可以直接从内存获得最近的DNS解析结果，同时HTTPDNS SDK在后台自动更新对应域名的解析结果
            ip = httpdns.getIpByHostAsync(url.getHost());
            if (ip != null) {
                // 通过HTTPDNS获取IP成功，进行URL替换和HOST头设置
                Log.d("HTTPDNS Demo", "Get IP: " + ip + " for host: " + url.getHost() + " from HTTPDNS successfully!");
                String newUrl = originalUrl.replaceFirst(url.getHost(), ip);
                conn = (HttpURLConnection) new URL(newUrl).openConnection();
            }
            DataInputStream dis = new DataInputStream(conn.getInputStream());
            int len;
            byte[] buff = new byte[4096];
            StringBuilder response = new StringBuilder();
            while ((len = dis.read(buff)) != -1) {
                response.append(new String(buff, 0, len));
            }
            Log.e("HTTPDNS Demo", "Response: " + response.toString());

        } catch (Exception e) {
```

```
            e.printStackTrace();
        }
    }

    /**
     * 检测系统是否已经设置代理，请参考HttpDNS API文档。
     */
    public static boolean detectIfProxyExist(Context ctx) {
        boolean IS_ICS_OR_LATER = Build.VERSION.SDK_INT >= Build.VERSION_CODES.ICE_CREAM_SANDWICH;
        String proxyHost;
        int proxyPort;
        if (IS_ICS_OR_LATER) {
            proxyHost = System.getProperty("http.proxyHost");
            String port = System.getProperty("http.proxyPort");
            proxyPort = Integer.parseInt(port != null ? port : "-1");
        } else {
            proxyHost = android.net.Proxy.getHost(ctx);
            proxyPort = android.net.Proxy.getPort(ctx);
        }
        return proxyHost != null && proxyPort != -1;
    }
}
```

对于使用HTTPDNS API接口的开发者，您可以在客户端自己定制更高效，并且符合您需求的HTTPDNS管理逻辑。

# iOS 302等重定向业务场景“IP直连”方案说明

## 概述

302等 URL 重定向业务场景需要解决的问题：

302 等重定向状态码，如何正确执行跳转逻辑，要求跳转后，依然需要执行 IP 直连逻辑，多次 302，也能覆盖到。

302等 URL 重定向业务场景问题主要集中在 POST 请求上，解决方案的方向大致有几种：

- 将请求方式统一替换为 GET
- 解决 POST 请求时的重定向问题

将 URL 统一替换为 GET，这种方案在客户端这边是成本最低的，如果团队中达成一致是最好的。不过限制也是显而易见的。那么我们就着重讨论下如何解决 POST 请求时的重定向问题。

## POST 请求的重定向问题

对于 GET 请求，重定向问题较为简单，我们着重讨论下 POST 请求的重定向问题，看下不同状态码下的响应方

式。

下面介绍下重定向的类型以及解释：

| 重定向的类型 | 对应协议 | 解释 |
| --- | --- | --- |
| 300 Multiple Choices | HTTP 1.0 | 可选重定向，表示客户请求的资源已经被转向到另外的地址了，但是没有说明是否是永久重定向还是临时重定向。 |
| 301 Moved Permancently | HTTP 1.0 | 永久重定向，同上，但是这个状态会告知客户请求的资源已经永久性的存在在新的重定向的 URL 上。 |
| 302 Moved Temporarily | HTTP 1.0 | 临时重定向，在 HTTP1.1 中状态描述是 Found，这个和 300 一样，但是说明请求的资源临时被转移到新的 URL 上，在以后可能会再次变动或者此 URL 会正常请求客户的连接。 |
| 303 See Other | HTTP 1.1 | 类似于 301/302，不同之处在于，如果原来的请求是 POST，Location 头指定的重定向目标文档应该通过 GET 提取（HTTP 1.1新）。 |
| 304 Not Modified | HTTP 1.0 | 并不真的是重定向 - 它用来响应条件 GET 请求，避免下载已经存在于浏览器缓存中的数据。 |
| 305 Use Proxy | HTTP 1.0 | 客户请求的文档应该通过 Location 头所指明的代理服务器提取（HTTP 1.1新）。 |
| 306 | HTTP 1.0 | 已废弃，不再使用 |
| 307 Temporary Redirect | HTTP 1.1 | 和302（Found）相同。许多浏览器会错误地响应 302 应答进行重定向，即使原来的请求是 POST，即使它实际上只能在 POST 请求的应答是 303 时 才能重定向。由于这个原因，HTTP 1.1新增了 307，以便更加清楚地区分几个状态代码：当出现 303 应答时，浏览器可以跟随重定向的 GET 和 POST 请求；如果是 307 应答，则浏览器只能跟随对 GET 请求的重定向。（HTTP 1.1新） |

因为常见的重定向为 301、302、303 307，所以下面重点说说这几种重定向的处理方法。

HTTP1.0 文档中的 302（或301）状态码，原则上是要被废弃的，它在 HTTP1.1 被细分为了 303 和 307。不过 303 和 307 应用并不广泛，现在很多公司对 302（或301）处理实际上是 303。

总结起来就是：

| 协议 | 状态码 | 协议规定 | 实际情况 |
| --- | --- | --- | --- |
| HTTP1.0 | 302（或301） | 不建议使用 | 仍在大面积使用 |
| HTTP1.1 | 303 + 307 | 旧有302（或301）被细分，并建议使用的新的状态码 | 应用面积较小 |

这些新旧协议的主要差别集中在 POST 请求的重定向处理上：

对于301、302的location中包含的重定向url，如果请求method不是GET或者HEAD，那么浏览器是禁止自动重定向的，除非得到用户的确认，因为POST、PUT等请求是非冥等的（也就是再次请求时服务器的资源可能已经发生了变化）

另外注意307这种情况，表示的是 POST 不自动重定向为 GET ，需要询问访问当前 URL 的用户，是否需要重定向，进行手动重定向。

目前浏览器大都还把302当作303处理了（注意，303是HTTP1.1才加进来的，其实从HTTP1.0进化到HTTP1.1，浏览器什么都没动），它们获取到 HTTP 响应报文头部的 Location 字段信息，并发起一个 GET 请求。

我们可以根据业务需要，对不同的状态码做处理，比如可以对303状态码做如下处理，

- location 中包含重定向 URL 就重定向
- 如果是 POST 请求修改为 GET 请求，并清空 body。
- 清空 host 信息
- 重新发送网络请求

代码示例：

```
    NSString *location = self.response.headerFields[@"Location"];
if (location && location.length > 0) {
NSURL *url = [[NSURL alloc] initWithString:location];
NSMutableURLRequest *mRequest = [self.swizzleRequest mutableCopy];
mRequest.URL = url;
if ([[self.swizzleRequest.HTTPMethod lowercaseString] isEqualToString:@"post"]) {
// POST重定向为GET
mRequest.HTTPMethod = @"GET";
mRequest.HTTPBody = nil;
}
[mRequest setValue:nil forHTTPHeaderField:@"host"];
```

**相关的文章：**

- 《WKWebView 那些坑》
- 《HTTP状态码302、303和307的故事》
- HTTP 302 wiki
- RFC1945
- RFC2616

# Android Webview + HttpDns最佳实践

## 1. 说明

本文档为Android WebView场景下接入HttpDns的参考方案，提供的相关代码也为参考代码，非线上生产环境正式代码。建议您在仔细阅读本文档，进行合理评估后再进行接入。由于Android生态碎片化严重，各厂商也进行了不同程度的定制，建议您灰度接入，并监控线上异常。有问题欢迎您随时向我们反馈，方便我们及时优化。

## 2. 背景

阿里云HTTPDNS是避免dns劫持的一种有效手段，在许多特殊场景如HTTPS/SNI、okhttp等都有最佳实践，但在webview场景下却一直没完美的解决方案。

但这并不代表在WebView场景下我们完全无法使用HTTPDNS,事实上很多场景依然可以通过HTTPDNS进行IP直连，本文旨在给出Android端HTTPDNS+WebView最佳实践供用户参考。

## 3. 接口

```
void setWebViewClient (WebViewClient client);
```

WebView提供了setWebViewClient接口对网络请求进行拦截，通过重载WebViewClient中的shouldInterceptRequest方法，我们可以拦截到所有的网络请求：

```
public class WebViewClient{
// API < 21
public WebResourceResponse shouldInterceptRequest(WebView view,
String url) {
...
}

// API >= 21
public WebResourceResponse shouldInterceptRequest(WebView view,
WebResourceRequest request) {
...
}
......

}
```

shouldInterceptRequest有两个版本：

- API < 21: public WebResourceResponse shouldInterceptRequest(WebView view, String url);
- API >= 21 public WebResourceResponse shouldInterceptRequest(WebView view, WebResourceRequest request);

# 4. 实践

## 4.1 API < 21

当API < 21时，shouldInterceptRequest方法的版本为：

```
public WebResourceResponse shouldInterceptRequest(WebView view, String url)
```

此时仅能获取到请求URL，请求方法、头部信息以及body等均无法获取，强行拦截该请求可能无法能到正确响应。所以当API < 21时，不对请求进行拦截：

```
public WebResourceResponse shouldInterceptRequest(WebView view,
String url) {
return super.shouldInterceptRequest(view, url);
}
```

## 4.2 API >= 21

当API >= 21时，shouldInterceptRequest提供了新版：

```
public WebResourceResponse shouldInterceptRequest(WebView view, WebResourceRequest request)
```

其中WebResourceRequest结构为：

```
public interface WebResourceRequest {
Uri getUrl(); // 请求URL
boolean isForMainFrame(); // 是否由主MainFrame发出的请求
boolean hasGesture(); // 是否是由某种行为(如点击)触发
String getMethod(); // 请求方法
Map<String, String> getRequestHeaders(); // 头部信息
}
```

可以看到，在API >= 21时，在拦截请求时，可以获取到如下信息：

- 请求URL
- 请求方法：POST, GET...
- 请求头

### 4.2.1 仅拦截GET请求

由于WebResourceRequest并没有提供请求body信息，所以只能拦截GET请求，不能拦截POST:

```
public WebResourceResponse shouldInterceptRequest(WebView view, WebResourceRequest request) {
String scheme = request.getUrl().getScheme().trim();
String method = request.getMethod();
Map<String, String> headerFields = request.getRequestHeaders();
// 无法拦截body，拦截方案只能正常处理不带body的请求；
if ((scheme.equalsIgnoreCase("http") || scheme.equalsIgnoreCase("https"))
&& method.equalsIgnoreCase("get")) {
.....
} else {
return super.shouldInterceptRequest(view, reqeust);
}
```

#### 4.2.2 设置头部信息

```
public WebResourceResponse shouldInterceptRequest(WebView view, WebResourceRequest request) {
.....

URL url = new URL(request.getUrl().toString());
conn = (HttpURLConnection) url.openConnection();
// 接口获取IP
String ip = httpdns.getIpByHostAsync(url.getHost());
if (ip != null) {
// 通过HTTPDNS获取IP成功，进行URL替换和HOST头设置
Log.d(TAG, "Get IP: " + ip + " for host: " + url.getHost() + " from HTTPDNS successfully!");
String newUrl = path.replaceFirst(url.getHost(), ip);
conn = (HttpURLConnection) new URL(newUrl).openConnection();

// 添加原有头部信息
if (headers != null) {
for (Map.Entry<String, String> field : headers.entrySet()) {
conn.setRequestProperty(field.getKey(), field.getValue());
}
}
// 设置HTTP请求头Host域
conn.setRequestProperty("Host", url.getHost());
}
}
```

#### 4.2.3 HTTPS请求证书校验

如果拦截到的请求是HTTPS请求，需要进行证书校验：

```
if (conn instanceof HttpsURLConnection) {
final HttpsURLConnection httpsURLConnection = (HttpsURLConnection)conn;
// https场景，证书校验
httpsURLConnection.setHostnameVerifier(new HostnameVerifier() {
@Override
public boolean verify(String hostname, SSLSession session) {
String host = httpsURLConnection.getRequestProperty("Host");
if (null == host) {
host = httpsURLConnection.getURL().getHost();
}
```

```
return HttpsURLConnection.getDefaultHostnameVerifier().verify(host, session);
}
});
}
```

#### 4.2.4 SNI场景

如果请求涉及到SNI场景，需要自定义SSLSocket，对SNI场景不熟悉的用户可以参考SNI:

```
TlsSniSocketFactory sslSocketFactory = new TlsSniSocketFactory((HttpsURLConnection) conn);
// sni场景，创建SSLScocket
((HttpsURLConnection) conn).setSSLSocketFactory(sslSocketFactory);

.....
class TlsSniSocketFactory extends SSLSocketFactory {
private final String TAG = "TlsSniSocketFactory";
HostnameVerifier hostnameVerifier = HttpsURLConnection.getDefaultHostnameVerifier();
private HttpsURLConnection conn;

public TlsSniSocketFactory(HttpsURLConnection conn) {
this.conn = conn;
}
.....

@Override
public Socket createSocket(Socket plainSocket, String host, int port, boolean autoClose) throws IOException {
String peerHost = this.conn.getRequestProperty("Host");
if (peerHost == null)
peerHost = host;
Log.i(TAG, "customized createSocket. host: " + peerHost);
InetAddress address = plainSocket.getInetAddress();
if (autoClose) {
// we don't need the plainSocket
plainSocket.close();
}
// create and connect SSL socket, but don't do hostname/certificate verification yet
SSLCertificateSocketFactory sslSocketFactory = (SSLCertificateSocketFactory)
SSLCertificateSocketFactory.getDefault(0);
SSLSocket ssl = (SSLSocket) sslSocketFactory.createSocket(address, port);

// enable TLSv1.1/1.2 if available
ssl.setEnabledProtocols(ssl.getSupportedProtocols());

// set up SNI before the handshake
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN_MR1) {
Log.i(TAG, "Setting SNI hostname");
sslSocketFactory.setHostname(ssl, peerHost);
} else {
Log.d(TAG, "No documented SNI support on Android <4.2, trying with reflection");
try {
java.lang.reflect.Method setHostnameMethod = ssl.getClass().getMethod("setHostname", String.class);
setHostnameMethod.invoke(ssl, peerHost);
} catch (Exception e) {
Log.w(TAG, "SNI not useable", e);
}
```

```
}

// verify hostname and certificate
SSLSession session = ssl.getSession();

if (!hostnameVerifier.verify(peerHost, session))
throw new SSLPeerUnverifiedException("Cannot verify hostname: " + peerHost);

Log.i(TAG, "Established " + session.getProtocol() + " connection with " + session.getPeerHost() +
" using " + session.getCipherSuite());

return ssl;
}
}
```

#### 4.2.5 重定向

如果服务端返回重定向，此时需要判断原有请求中是否含有cookie：

- 如果原有请求报头含有cookie，因为cookie是以域名为粒度进行存储的，重定向后cookie会改变，且无法获取到新请求URL下的cookie，所以放弃拦截
- 如果不含cookie，重新发起二次请求

```
int code = conn.getResponseCode();
if (code >= 300 && code < 400) {
if (请求报头中含有cookie) {
// 不拦截
return super.shouldInterceptRequest(view, request);
}

//临时重定向和永久重定向location的大小写有区分
String location = conn.getHeaderField("Location");
if (location == null) {
location = conn.getHeaderField("location");
}
if (!(location.startsWith("http://") || location
.startsWith("https://"))) {
//某些时候会省略host，只返回后面的path，所以需要补全url
URL originalUrl = new URL(path);
location = originalUrl.getProtocol() + "://"
+ originalUrl.getHost() + location;
}
Log.e(TAG, "code:" + code + "; location:" + location + ";path" + path);

发起二次请求
} else {
// redirect finish.
Log.e(TAG, "redirect finish");
.....
}
```

#### 4.2.6 MIME&Encoding

如果拦截网络请求，需要返回一个WebResourceResponse：

```
public WebResourceResponse(String mimeType, String encoding, InputStream data) ;
```

创建WebResourceResponse对象需要提供：

- 请求的MIME类型
- 请求的编码
- 请求的输入流

其中请求输入流可以通过URLConnection.getInputStream()获取到，而MIME类型和encoding可以通过请求的ContentType获取到，即通过URLConnection.getContentType(),如：

```
text/html;charset=utf-8
```

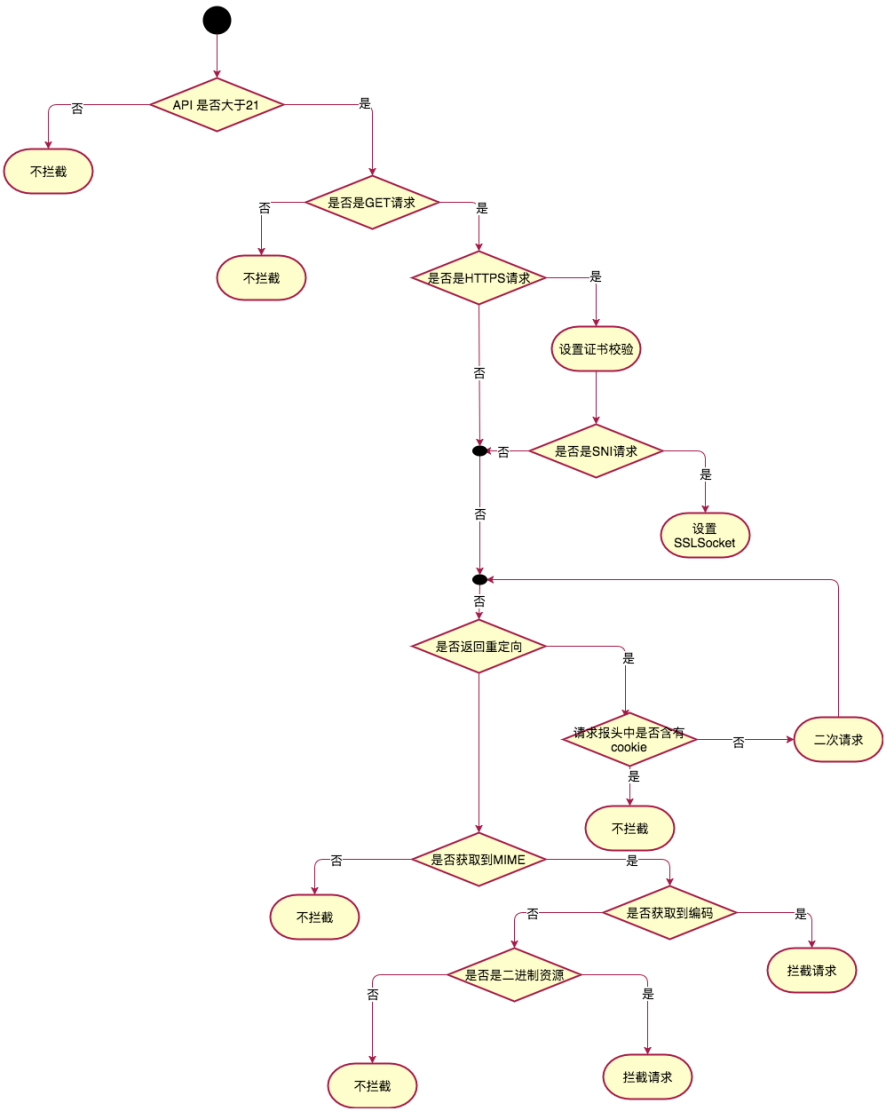
但并不是所有的请求都能得到完整的contentType信息，此时可以参考如下策略：

```
String contentType = conn.getContentType();
String mime = getMime(contentType);
String charset = getCharset(contentType);

// 无MIME类型的请求不拦截
if (TextUtils.isEmpty(mime)) {
return super.shouldInterceptRequest(view, request);
} else {
if (!TextUtils.isEmpty(charset)) {
// 如果同时获取到MIME和charset可以直接拦截
return new WebResourceResponse(mime, charset, connection.getInputStream());
} else {
//获取不到编码信息

// 二进制资源无需编码信息，可以进行拦截
if (isBinaryRes(mime)) {
Log.e(TAG, "binary resource for " + mime);
return new WebResourceResponse(mime, charset, connection.getInputStream());
} else {
// 非二进制资源需要编码信息，不拦截
Log.e(TAG, "non binary resource for " + mime);
return super.shouldInterceptRequest(view, request);
}
}
}

private boolean isBinaryRes(String mime) {
// 可进行扩展
if (mime.startsWith("image")
|| mime.startsWith("audio")
|| mime.startsWith("video")) {
return true;
} else {
return false;
}
```

```
}
```

# 5 总结

综上所述，在WebView场景下的请求拦截逻辑如下所示：




## 5.1 【不可用场景】

- API Level < 21的设备
- POST请求
- 无法获取到MIME类型的请求
- 无法获取到编码的非二进制文件请求

## 5.2【可用场景】

前提条件：

- API Level >= 21
- GET请求
- 可以获取到MIME类型以及编码信息请求或是可以获取到MIME类型的二进制文件请求

可用场景：

- 普通HTTP请求
- HTTPS请求
- SNI请求
- HTTP报头中不含cookie的重定向请求

### 5.3 完整代码

HTTPDNS+WebView最佳实践完整代码请参考：GithubDemo

# HTTPDNS域名解析场景下如何使用Cookie？

请参考云栖社区文档HTTPDNS域名解析场景下如何使用Cookie？

# HttpDns+OkHttp最佳实践

HTTPS（含SNI）业务场景“IP直连”方案说明一文介绍了利用HttpDns解析获得ip后进行ip直连的通用方法。但是如果您在Android端使用的网络框架是OkHttp,通过调用OkHttp提供的自定义Dns服务接口，可以更为优雅地使用HttpDns的服务。

OkHttp是一个处理网络请求的开源项目,是Android端最火热的轻量级框架,由移动支付Square公司贡献用于替代HttpUrlConnection和Apache HttpClient。随着OkHttp的不断成熟，越来越多的Android开发者使用OkHttp作为网络框架。

OkHttp默认使用系统DNS服务InetAddress进行域名解析，但同时也暴露了自定义DNS服务的接口，通过该接口我们可以优雅地使用HttpDns。

## 1. 自定义DNS接口

OkHttp暴露了一个Dns接口，通过实现该接口，我们可以自定义Dns服务：

```
public class OkHttpDns implements Dns {
private static final Dns SYSTEM = Dns.SYSTEM;
```

```
HttpDnsService httpdns;//httpdns 解析服务
private static OkHttpDns instance = null;
private OkHttpDns(Context context) {
this.httpdns = HttpDns.getService(context, "account id");
}
public static OkHttpDns getInstance(Context context) {
if(instance == null) {
instance = new OkHttpDns(context);
}
return instance;
}
@Override
public List<InetAddress> lookup(String hostname) throws UnknownHostException {
//通过异步解析接口获取ip
String ip = httpdns.getIpByHostAsync(hostname);
if(ip != null) {
//如果ip不为null，直接使用该ip进行网络请求
List<InetAddress> inetAddresses = Arrays.asList(InetAddress.getAllByName(ip));
Log.e("OkHttpDns", "inetAddresses:" + inetAddresses);
return inetAddresses;
}
//如果返回null，走系统DNS服务解析域名
return Dns.SYSTEM.lookup(hostname);
}
}
```

## 2. 创建OkHttpClient

创建OkHttpClient对象，传入OkHttpDns对象代替默认Dns服务：

```
private void okhttpDnsRequest() {
OkHttpClient client = new OkHttpClient.Builder()
.dns(OkHttpDns.getInstance(getApplicationContext()))
.build();

Request request = new Request.Builder()
.url("http://www.aliyun.com")
.build();

Response response = null;
client.newCall(request).enqueue(new Callback() {
@Override
public void onFailure(Call call, IOException e) {
e.printStackTrace();
}

@Override
public void onResponse(Call call, Response response) throws IOException {
if (!response.isSuccessful()) throw new IOException("Unexpected code " + response);
DataInputStream dis = new DataInputStream(response.body().byteStream());
int len;
byte[] buff = new byte[4096];
StringBuilder result = new StringBuilder();
while ((len = dis.read(buff)) != -1) {
```

```
result.append(new String(buff, 0, len));
}
Log.d("OkHttpDns", "Response: " + result.toString());
}
});
}
```

以上就是OkHttp+HttpDns实现的全部代码。

## 3. 总结

相比于通用方案，OkHttp+HttpDns有以下两个主要优势：

- 实现简单，只需通过实现Dns接口即可接入HttpDns服务
- 通用性强，该方案在HTTPS,SNI以及设置Cookie等场景均适用。规避了证书校验，域名检查等环节

该实践对于Retrofit+OkHttp同样适用，将配置好的OkHttpClient作为
Retrofit.Builder::client(OkHttpClient)参数传入即可。

# iOS HTTPS SNI 业务场景“IP直连”方案说明

## 概述

SNI（单IP多HTTPS证书）场景下，iOS上层网络库 NSURLConnection/NSURLSession 没有提供接口进行 SNI 字段 配置，因此需要 Socket 层级的底层网络库例如 CFNetwork，来实现 IP 直连网络请求适配方案。而基于 CFNetwork 的解决方案需要开发者考虑数据的收发、重定向、解码、缓存等问题（CFNetwork是非常底层的网络实现）。

针对 SNI 场景的方案， Socket 层级的底层网络库，大致有两种：

- 基于 CFNetWork ，hook 证书校验步骤。
- 基于原生支持设置 SNI 字段的更底层的库，比如 libcurl。

下面将目前面临的一些挑战，以及应对策略介绍一下：

## 支持 Post 请求

使用 NSURLProtocol 拦截 NSURLSession 请求丢失 body，故有以下几种解决方法：

方案如下：

使用 HTTPBodyStream 获取 body，并赋值到 body 中，具体的代码如下，可以解决上面提到的问题：

```
//
// NSURLRequest+NSURLProtocolExtension.h
//
//
// Created by ElonChan on 28/07/2017.
// Copyright © 2017 ChenYilong. All rights reserved.
//

#import <Foundation/Foundation.h>

@interface NSURLRequest (NSURLProtocolExtension)

- (NSURLRequest *)httpdns_getPostRequestIncludeBody;

@end



//
// NSURLRequest+NSURLProtocolExtension.h
//
//
// Created by ElonChan on 28/07/2017.
// Copyright © 2017 ChenYilong. All rights reserved.
//

#import "NSURLRequest+NSURLProtocolExtension.h"

@implementation NSURLRequest (NSURLProtocolExtension)

- (NSURLRequest *)httpdns_getPostRequestIncludeBody {
return [[self httpdns_getMutablePostRequestIncludeBody] copy];
}

- (NSMutableURLRequest *)httpdns_getMutablePostRequestIncludeBody {
NSMutableURLRequest * req = [self mutableCopy];
if ([self.HTTPMethod isEqualToString:@"POST"]) {
if (!self.HTTPBody) {
NSInteger maxLength = 1024;
uint8_t d[maxLength];
NSInputStream *stream = self.HTTPBodyStream;
NSMutableData *data = [[NSMutableData alloc] init];
[stream open];
BOOL endOfStreamReached = NO;
//不能用 [stream hasBytesAvailable]) 判断，处理图片文件的时候这里的[stream hasBytesAvailable]会始终返回YES，导致在while里面死循环。
while (!endOfStreamReached) {
NSInteger bytesRead = [stream read:d maxLength:maxLength];
if (bytesRead == 0) { //文件读取到最后
endOfStreamReached = YES;
} else if (bytesRead == -1) { //文件读取错误
endOfStreamReached = YES;
} else if (stream.streamError == nil) {
[data appendBytes:(void *)d length:bytesRead];
}
```

```
}
req.HTTPBody = [data copy];
[stream close];
}

}
return req;
}

@end
```

使用方法：

在用于拦截请求的 NSURLProtocol 的子类中实现方法 +canonicalRequestForRequest: 并处理 request 对象：

```
+ (NSURLRequest *)canonicalRequestForRequest:(NSURLRequest *)request {
return [request httpdns_getPostRequestIncludeBody];
}
```

下面介绍下相关方法的作用：

```
 //NSURLProtocol.h

/*!
@method canInitWithRequest:
@abstract This method determines whether this protocol can handle
the given request.
@discussion A concrete subclass should inspect the given request and
determine whether or not the implementation can perform a load with
that request. This is an abstract method. Sublasses must provide an
implementation.
@param request A request to inspect.
@result YES if the protocol can handle the given request, NO if not.
*/
+ (BOOL)canInitWithRequest:(NSURLRequest *)request;

/*!
@method canonicalRequestForRequest:
@abstract This method returns a canonical version of the given
request.
@discussion It is up to each concrete protocol implementation to
define what "canonical" means. However, a protocol should
guarantee that the same input request always yields the same
canonical form. Special consideration should be given when
implementing this method since the canonical form of a request is
used to look up objects in the URL cache, a process which performs
equality checks between NSURLRequest objects.
<p>
This is an abstract method; sublasses must provide an
implementation.
@param request A request to make canonical.
@result The canonical form of the given request.
```

```
*/
+ (NSURLRequest *)canonicalRequestForRequest:(NSURLRequest *)request;
```

翻译下：

```
//NSURLProtocol.h
/*!
* @method:创建NSURLProtocol实例，NSURLProtocol注册之后，所有的NSURLConnection都会通过这个方法检查是否持有该Http请求。
@parma :
@return: YES：持有该Http请求NO：不持有该Http请求
*/
+ (BOOL)canInitWithRequest:(NSURLRequest *)request

/*!
* @method: NSURLProtocol抽象类必须要实现。通常情况下这里有一个最低的标准：即输入输出请求满足最基本的协议规范一致。因此这里简单的做法可以直接返回。一般情况下我们是不会去更改这个请求的。如果你想更改，比如给这个request添加一个title，组合成一个新的http请求。
@parma: 本地HttpRequest请求：request
@return:直接转发
*/

+ (NSURLRequest*)canonicalRequestForRequest:(NSURLRequest *)request
```

简单说：

- +[NSURLProtocol canInitWithRequest:] 负责筛选哪些网络请求需要被拦截
- +[NSURLProtocol canonicalRequestForRequest:] 负责对需要拦截的网络请求NSURLRequest 进行重新构造。

这里有一个注意点：+[NSURLProtocol canonicalRequestForRequest:] 的执行条件是 +[NSURLProtocol canInitWithRequest:] 返回值为 YES。

注意在拦截 NSURLSession 请求时，需要将用于拦截请求的 NSURLProtocol 的子类添加到 NSURLSessionConfiguration 中，用法如下：

```
    NSURLSessionConfiguration *configuration = [NSURLSessionConfiguration defaultSessionConfiguration];
NSArray *protocolArray = @[ [CUSTOMEURLProtocol class] ];
configuration.protocolClasses = protocolArray;
NSURLSession *session = [NSURLSession sessionWithConfiguration:configuration delegate:self
delegateQueue:[NSOperationQueue mainQueue]];
```

# 换用其他提供了SNI字段配置接口的更底层网络库

如果使用第三方网络库：curl， 中有一个 -resolve 方法可以实现使用指定 ip 访问 https 网站,iOS 中集成 curl 库，参考 curl文档 ；

另外有一点也可以注意下，它也是支持 IPv6 环境的，只需要你在 build 时添加上 --enable-ipv6 即可。

curl 支持指定 SNI 字段，设置 SNI 时我们需要构造的参数形如： {HTTPS域名}:443:{IP地址}

假设你要访问. www.example.org ，若IP为 127.0.0.1 ，那么通过这个方式来调用来设置 SNI 即可：

```
curl *—resolve 'www.example.org:443:127.0.0.1'
```

## iOS CURL 库

使用libcurl 来解决，libcurl / cURL 至少 7.18.1 （ 2008年3月30日 ） 在 SNI 支持下编译一个 SSL/TLS 工具包 ，curl 中有一个 --resolve 方法可以实现使用指定ip访问https网站。

在iOS实现中，代码如下

```
//{HTTPS域名}:443:{IP地址}
NSString *curlHost = ...;
_hosts_list = curl_slist_append(_hosts_list, curlHost.UTF8String);
curl_easy_setopt(_curl, CURLOPT_RESOLVE, _hosts_list);
```

其中 curlHost 形如：

{HTTPS域名}:443:{IP地址}

_hosts_list 是结构体类型hosts_list，可以设置多个IP与Host之间的映射关系。curl_easy_setopt方法中传入 CURLOPT_RESOLVE 将该映射设置到 HTTPS 请求中。

这样就可以达到设置SNI的目的。

## 参考链接：

- Apple - Communicating with HTTP Servers
- Apple - HTTPS Server Trust Evaluation - Server Name Failures
- Apple - HTTPS Server Trust Evaluation - Trusting One Specific Certificate
- 《HTTPDNS > 最佳实践 > HTTPS（含SNI）业务场景“IP直连”方案说明HTTPS（含SNI）业务场景“IP直连”方案说明》
- 《在 curl 中使用指定 ip 来进行请求 https》
- 《SNI: 实现多域名虚拟主机的SSL/TLS认证》

# iOS端WEEX + HTTPDNS 最佳实践

- WEEX + HTTPDNS iOS 解决方案 Stream网络请求 + HTTPDNS 新版本Weex SDK 实现旧版本 Weex SDK 实现<image> 网络请求 + HTTPDNS

# WEEX + HTTPDNS iOS 解决方案

由于WebView并未暴露处设置DNS的接口，因而在WebView场景下使用HTTPDNS存在很多无法限制，但如果接入WEEX,则可以较好地植入HTTPDNS，本文主要介绍在WEEX场景下接入HTTPDNS的方案细节。

在WEEX运行时环境下，所有的逻辑最终都会转换到Native Runtime中执行，网络请求也不例外。同时WEEX也提供了自定义相应实现的接口，通过重写网络请求适配器，我们可以较为简单地接入HTTPDNS。在WEEX运行环境中，主要有两种网络请求：

- 通过Stream进行的网络请求
- <image>标签指定的加载图片的网络请求

## Stream网络请求 + HTTPDNS

**新版本Weex SDK 实现**

下面以 weex iOS 0.17.0 版本为例：

Stream 网络请求在 iOS 端最终会通过 WXResourceRequestHandlerDefaultImpl 完成，同时 WEEX 也提供了相应的接口自定义网络请求适配器。具体的逻辑如下：

第一步：创建自定义网络请求适配器，实现 WXResourceRequestHandlerHttpDnsImpl 接口

```
#import <WeexSDK/WeexSDK.h>
#import "WXResourceRequestHandlerDefaultImpl.h"

@interface WXResourceRequestHandlerHttpDnsImpl :
WXResourceRequestHandlerDefaultImpl<WXResourceRequestHandler,NSURLSessionDelegate>

@end
```

第二步：在WEEX初始化时注册自定义网络适配器，替换默认适配器：

```
    [WXSDKEngine registerHandler:[WXResourceRequestHandlerHttpDnsImpl new]
withProtocol:@protocol(WXResourceRequestHandler)];
```

下面以 weex iOS 0.7.0 版本为例：

Stream网络请求在 iOS 端最终会通过WXNetworkDefaultImpl完成，同时WEEX也提供了相应的接口自定义网络请求适配器。具体的逻辑如下：

第一步：创建自定义网络请求适配器，实现 WXNetworkHttpDnsImpl 接口

```
#import <WeexSDK/WeexSDK.h>
#import "WXNetworkDefaultImpl.h"

@interface WXNetworkHttpDnsImpl : NSObject<WXNetworkProtocol, WXModuleProtocol ,
NSURLSessionDelegate>
```

```
@end
```

第二步：在WEEX初始化时注册自定义网络适配器，替换默认适配器：

```
    [WXSDKEngine registerHandler:[WXNetworkHttpDnsImpl new] withProtocol:@protocol(WXNetworkProtocol)];
```

之后的网络请求都会通过WXNetworkHttpDnsImpl实现，所以只需要在WXNetworkHttpDnsImpl中植入HTTPDNS 逻辑即可，具体逻辑可以参考如下代码：

```
#import "WXResourceRequestHandlerHttpDnsImpl.h"
#import "WXThreadSafeMutableDictionary.h"
#import "WXAppConfiguration.h"
#import <AlicloudHttpDNS/AlicloudHttpDNS.h>

@interface WXResourceRequestHandlerHttpDnsImpl () <NSURLSessionDataDelegate>

@property (nonatomic, strong) NSMutableURLRequest *request;

@end

@implementation WXResourceRequestHandlerHttpDnsImpl {
NSURLSession *_session;
WXThreadSafeMutableDictionary<NSURLSessionDataTask *, id<WXResourceRequestDelegate>> *_delegates;
}

#pragma mark - WXResourceRequestHandler

- (void)sendRequest:(WXResourceRequest *)theRequest withDelegate:(id<WXResourceRequestDelegate>)delegate
{
self.request = [theRequest mutableCopy];
NSString *originalUrl = [theRequest.URL absoluteString];
NSString *originalHost = theRequest.URL.host;

// 初始化httpdns实例
HttpDnsService *httpdns = [HttpDnsService sharedInstance];
NSString *ip = [httpdns getIpByHostAsync:theRequest.URL.host];
if (ip) {
// 通过HTTPDNS获取IP成功，进行URL替换和HOST头设置
NSLog(@"Get IP(%@) for host(%@) from HTTPDNS Successfully!", ip, originalHost);
NSRange hostFirstRange = [originalUrl rangeOfString:originalHost];
if (NSNotFound != hostFirstRange.location) {
NSString *newUrl = [originalUrl stringByReplacingCharactersInRange:hostFirstRange withString:ip];
NSLog(@"New URL: %@", newUrl);
self.request.URL = [NSURL URLWithString:newUrl];
[self.request setValue:originalHost forHTTPHeaderField:@"host"];
}
}

if (!_session) {
NSURLSessionConfiguration *urlSessionConfig = [NSURLSessionConfiguration defaultSessionConfiguration];
if ([WXAppConfiguration customizeProtocolClasses].count > 0) {
NSArray *defaultProtocols = urlSessionConfig.protocolClasses;
```

```
urlSessionConfig.protocolClasses = [[WXAppConfiguration customizeProtocolClasses]
arrayByAddingObjectsFromArray:defaultProtocols];
}
_session = [NSURLSession sessionWithConfiguration:urlSessionConfig
delegate:self
delegateQueue:[NSOperationQueue mainQueue]];
_delegates = [WXThreadSafeMutableDictionary new];
}

NSURLSessionDataTask *task = [_session dataTaskWithRequest:theRequest];
theRequest.taskIdentifier = task;
[_delegates setObject:delegate forKey:task];
[task resume];
}

- (void)cancelRequest:(WXResourceRequest *)request
{
if ([request.taskIdentifier isKindOfClass:[NSURLSessionTask class]]) {
NSURLSessionTask *task = (NSURLSessionTask *)request.taskIdentifier;
[task cancel];
[_delegates removeObjectForKey:task];
}
}

#pragma mark - NSURLSessionTaskDelegate & NSURLSessionDataDelegate

- (void)URLSession:(NSURLSession *)session
task:(NSURLSessionTask *)task
didSendBodyData:(int64_t)bytesSent
totalBytesSent:(int64_t)totalBytesSent
totalBytesExpectedToSend:(int64_t)totalBytesExpectedToSend
{
id<WXResourceRequestDelegate> delegate = [_delegates objectForKey:task];
[delegate request:(WXResourceRequest *)task.originalRequest didSendData:bytesSent
totalBytesToBeSent:totalBytesExpectedToSend];
}

- (void)URLSession:(NSURLSession *)session dataTask:(NSURLSessionDataTask *)task
didReceiveResponse:(NSURLResponse *)response
completionHandler:(void (^)(NSURLSessionResponseDisposition))completionHandler
{
id<WXResourceRequestDelegate> delegate = [_delegates objectForKey:task];
[delegate request:(WXResourceRequest *)task.originalRequest didReceiveResponse:(WXResourceResponse
*)response];
completionHandler(NSURLSessionResponseAllow);
}

- (void)URLSession:(NSURLSession *)session dataTask:(NSURLSessionDataTask *)task didReceiveData:(NSData *)data
{
id<WXResourceRequestDelegate> delegate = [_delegates objectForKey:task];
[delegate request:(WXResourceRequest *)task.originalRequest didReceiveData:data];
}

- (void)URLSession:(NSURLSession *)session task:(NSURLSessionTask *)task didCompleteWithError:(NSError *)error
{
id<WXResourceRequestDelegate> delegate = [_delegates objectForKey:task];
```

```
if (error) {
[delegate request:(WXResourceRequest *)task.originalRequest didFailWithError:error];
}else {
[delegate requestDidFinishLoading:(WXResourceRequest *)task.originalRequest];
}
[_delegates removeObjectForKey:task];
}

#ifdef __IPHONE_10_0
- (void)URLSession:(NSURLSession *)session task:(NSURLSessionTask *)task
didFinishCollectingMetrics:(NSURLSessionTaskMetrics *)metrics
{
id<WXResourceRequestDelegate> delegate = [_delegates objectForKey:task];
[delegate request:(WXResourceRequest *)task.originalRequest didFinishCollectingMetrics:metrics];
}
#endif

- (BOOL)evaluateServerTrust:(SecTrustRef)serverTrust
forDomain:(NSString *)domain {
/*
* 创建证书校验策略
*/
NSMutableArray *policies = [NSMutableArray array];
if (domain) {
[policies addObject:(__bridge_transfer id) SecPolicyCreateSSL(true, (__bridge CFStringRef) domain)];
} else {
[policies addObject:(__bridge_transfer id) SecPolicyCreateBasicX509()];
}
/*
* 绑定校验策略到服务端的证书上
*/
SecTrustSetPolicies(serverTrust, (__bridge CFArrayRef) policies);
/*
* 评估当前serverTrust是否可信任，
* 官方建议在result = kSecTrustResultUnspecified 或 kSecTrustResultProceed
* 的情况下serverTrust可以被验证通过，https://developer.apple.com/library/ios/technotes/tn2232/_index.html
* 关于SecTrustResultType的详细信息请参考SecTrust.h
*/
SecTrustResultType result;
SecTrustEvaluate(serverTrust, &result);
return (result == kSecTrustResultUnspecified || result == kSecTrustResultProceed);
}

#pragma mark - NSURLSessionTaskDelegate
- (void)URLSession:(NSURLSession *)session task:(NSURLSessionTask *)task
didReceiveChallenge:(NSURLAuthenticationChallenge *)challenge completionHandler:(void
(^)(NSURLSessionAuthChallengeDisposition, NSURLCredential *_Nullable))completionHandler {
if (!challenge) {
return;
}
NSURLSessionAuthChallengeDisposition disposition = NSURLSessionAuthChallengePerformDefaultHandling;
NSURLCredential *credential = nil;
/*
* 获取原始域名信息。
*/
NSString *host = [[self.request allHTTPHeaderFields] objectForKey:@"host"];
```

```
if (!host) {
host = self.request.URL.host;
}
if ([challenge.protectionSpace.authenticationMethod isEqualToString:NSURLAuthenticationMethodServerTrust]) {
if ([self evaluateServerTrust:challenge.protectionSpace.serverTrust forDomain:host]) {
disposition = NSURLSessionAuthChallengeUseCredential;
credential = [NSURLCredential credentialForTrust:challenge.protectionSpace.serverTrust];
} else {
disposition = NSURLSessionAuthChallengePerformDefaultHandling;
}
} else {
disposition = NSURLSessionAuthChallengePerformDefaultHandling;
}
// 对于其他的challenges直接使用默认的验证方案
completionHandler(disposition, credential);
}

@end
```

**旧版本Weex SDK 实现**

以0.7.0为例子：

网络请求都会通过WXNetworkHttpDnsImpl实现，所以只需要在WXNetworkHttpDnsImpl中植入HTTPDNS 逻辑即可，具体逻辑可以参考如下代码：

```
#import "WXNetworkDefaultImpl.h"

@interface WXNetworkCallbackInfo : NSObject

@property (nonatomic, copy) void(^sendDataCallback)(int64_t, int64_t);
@property (nonatomic, copy) void(^responseCallback)(NSURLResponse *);
@property (nonatomic, copy) void(^receiveDataCallback)(NSData *);
@property (nonatomic, strong) NSMutableData *data;
@property (nonatomic, copy) void(^compeletionCallback)(NSData *, NSError *);

@end

@implementation WXNetworkCallbackInfo

@end

@implementation WXNetworkDefaultImpl
{
NSMutableDictionary *_callbacks;
NSURLSession *_session;
}

- (id)sendRequest:(NSURLRequest *)request withSendingData:(void (^)(int64_t, int64_t))sendDataCallback
withResponse:(void (^)(NSURLResponse *))responseCallback
withReceiveData:(void (^)(NSData *))receiveDataCallback
withCompeletion:(void (^)(NSData *, NSError *))compeletionCallback
{
WXNetworkCallbackInfo *info = [WXNetworkCallbackInfo new];
info.sendDataCallback = sendDataCallback;
```

```
info.responseCallback = responseCallback;
info.receiveDataCallback = receiveDataCallback;
info.compeletionCallback = compeletionCallback;

if (!_session) {
_session = [NSURLSession sessionWithConfiguration:[NSURLSessionConfiguration defaultSessionConfiguration]
delegate:self
delegateQueue:[NSOperationQueue mainQueue]];
}

NSURLSessionDataTask *task = [_session dataTaskWithRequest:request];
if (!_callbacks) {
_callbacks = [NSMutableDictionary dictionary];
}
[_callbacks setObject:info forKey:task];
[task resume];

return task;
}

- (void)URLSession:(NSURLSession *)session task:(NSURLSessionTask *)task
didSendBodyData:(int64_t)bytesSent
totalBytesSent:(int64_t)totalBytesSent
totalBytesExpectedToSend:(int64_t)totalBytesExpectedToSend
{
WXNetworkCallbackInfo *info = [_callbacks objectForKey:task];
if (info.sendDataCallback) {
info.sendDataCallback(totalBytesSent, totalBytesExpectedToSend);
}
}

- (void)URLSession:(NSURLSession *)session dataTask:(NSURLSessionDataTask *)task
didReceiveResponse:(NSURLResponse *)response
completionHandler:(void (^)(NSURLSessionResponseDisposition))completionHandler
{
WXNetworkCallbackInfo *info = [_callbacks objectForKey:task];
if (info.responseCallback) {
info.responseCallback(response);
}
completionHandler(NSURLSessionResponseAllow);
}

- (void)URLSession:(NSURLSession *)session dataTask:(NSURLSessionDataTask *)task didReceiveData:(NSData *)data
{
WXNetworkCallbackInfo *info = [_callbacks objectForKey:task];
if (info.receiveDataCallback) {
info.receiveDataCallback(data);
}

NSMutableData *mutableData = info.data;
if (!mutableData) {
mutableData = [NSMutableData new];
info.data = mutableData;
}

[mutableData appendData:data];
```

```
}

- (void)URLSession:(NSURLSession *)session task:(NSURLSessionTask *)task didCompleteWithError:(NSError *)error
{
WXNetworkCallbackInfo *info = [_callbacks objectForKey:task];
if (info.compeletionCallback) {
info.compeletionCallback(info.data, error);
}
[_callbacks removeObjectForKey:task];
}

@end
```

### <image> 网络请求 + HTTPDNS

WEEX并没有提供默认的图片适配器实现，所以用户必须自行实现才能完成图片请求逻辑，具体步骤分为以下几步：

第一步：自定义图片请求适配器，实现IWXImgLoaderAdapter接口

```
#import "WXImgLoaderProtocol.h"

@interface WXImgLoaderDefaultImpl : NSObject<WXImgLoaderProtocol, WXModuleProtocol>
@end
```

第二步：在WEEX初始化时注册该图片适配器：

```
[WXSDKEngine registerHandler:[WXImgLoaderDefaultImpl new] withProtocol:@protocol(WXImgLoaderProtocol)];
```

所以同WXNetworkHttpDnsImpl一样，我们只需在WXNetworkHttpDnsImpl植入HTTPDNS逻辑即可。

# 解析异常排查之 “会话追踪方案”

当您对解析结果或HTTPDNS服务质量产生如下疑问，希望排查问题时，请参考此“会话追踪方案”：

- 您认为解析出的IP不符合预期，以至于影响了业务域名的服务质量
- 您接入的APM系统显示，HTTPDNS服务质量有问题

**注意：** 对于有疑问的解析结果IP，需要先确定是否由HTTPDNS解析的结果：需要您提供从HTTPDNS接口取出IP的日志及具体的HTTPDNS接入代码。

**注意：** 由于技术限制，我们只提供3天内的解析问题排查。

**注意：** 由于此方案需要配合SDK的getSessionId方法，您至少需要升级至 Android ≥ 1.2.3, iOS ≥

1.6.20 才能应用此方案。

# 背景介绍

对于一个配置了分线路解析的域名，根据客户入网IP的不同，返回的IP是根据各线路的配置而不同。为了获得更好的服务质量，服务节点调度系统(GSLB)可能会频繁地变更上述的线路调度配置。例如CDN系统，可能每几个小时就会调整一次调度配置。如果客户端获取到了错误的调度结果(即解析结果)，一方面可能会导致服务质量受损，也就是慢或超时；一方面还有可能导致服务不可用。

为了定位上述问题，一般需要您提供『来源IP，解析时间，解析的域名，解析结果IP列表』来定位异常现场，以便观察是否有对应的调度结果记录。但这种定位异常现场的方案存在以下问题：

1. 客户端难以提供准确的入网IP
2. 即使提供了准确的入网IP，由于IP一般是共享的，不一定能定位到该次解析现场
3. 对于同一个网络环境，运营商可能会针对不同的目的IP，提供不同的入网IP
4. 在app的一次生命周期中，入网IP可能发生变化
5. 以及难以解释以上入网IP发生的变化原因

# 引入sessionId（会话唯一id）

sessionId参数在app启动时生成，在整个app生命周期中不会发生变化。在同一个app生命周期中的HTTPDNS解析请求均会携带同一个sessionId，服务器则会记录这个参数，并生成索引。与app异常日志上报中以单个设备id为索引条件类似，本方案以app的一次生命周期作为做索引条件。与问题常规排查方案中『来源IP-解析结果』组合的追踪方式相比，使用sessionId对app生命周期进行追踪，是一种更精准，更适用于app场景的问题排查方式。

通过引入sessionId，我们不再需要您提供客户端入网IP，改为根据sessionId获取准确的IP信息。同时可以追踪在app的一次生命周期中所有的HTTPDNS解析请求，可以观测到入网IP可能发生的变化，还可以根据net参数的变化解读客户入网IP变化的原因。

例如一个客户通过4G入网，接入使用的SIM卡可能是广东的运营商发行的，那么接入IP可能是广东区域的。当从HTTPDNS获取IP后，客户连接了其物理所在地的有线网络或wifi，可能导致接入IP立即变为该地区的IP，导致内容服务质量受到影响。这种场景在传统LocalDNS环境下和HTTPDNS下都会出现。

# 使用“会话追踪方案”

由于app更新升级过程是渐进式的，无法在短时间内部署最新的版本，我们建议您在接入HTTPDNS时就参考本方案，记录解析结果和对应的sessionId，以便后续有问题时，方便准确提供追查线索。

- SDK接入：我们会在app启动时生成sessionId，并提供了getSessionId方法获取生成的结果。您需要更新SDK版本来使用此功能（Android ≥ 1.2.3, iOS ≥ 1.6.20），但我们建议您升级至最新版本，以便解决一些已知的其他SDK问题。
- HTTP API方式接入（即非SDK接入）：您可以仿照SDK的逻辑生成相应参数

- 端日志上报调度服务质量问题时，需要额外记录sessionId参数

额外地，您也可以将sessionId携带至其他系统中：当使用HTTPDNS解析结果，请求该域名的服务时，可以在URL中额外携带sessionId参数，这样可以更加完整地观察客户使用场景。

在本方案中，每次SDK对HTTPDNS服务器的请求中，我们添加了如下3个参数（服务器会记录这些信息）：

- sid=<sessionId:[a-zA-Z0-9]{12}>，在SDK启动时生成，用于标记一次独立的app生命周期
- net=<4g|3g|2g|wifi|unknown>，用于标记请求发起时刻os层提供的网络情况；提供sessionId后，我们可以将当时记录的net信息反馈给您
- bssid=<wifi_bssid>，用于标记不同的wifi网络；提供sessionId后，我们可以将当时记录的bssid信息反馈给您
- 示例URL: http://203.107.1.33/100000/d?host=www.aliyun.com&sid=wInhNA3iM0PK&net=wifi&bssid=54e061553e79

# 排查场景：解析出的结果不符合预期

当出现解析出的结果不符合预期，您认为可能和HTTPDNS有关，需要排查时：

请您提交HTTPDNS工单，并提供以下信息

- 解析时的sessionId、解析时间、解析的域名、解析结果、预期解析结果
- 是否造成了实际业务服务质量下降（例如调度到了不正确地域的服务节点）？
- 是否造成了业务不可用的问题？
    - 如果发现这类问题，建议您尽早针对此类情况，考虑重新获取域名解析结果（HTTPDNS或LocalDNS）,并进行业务重试

我们会根据sessionId提供以下调查结果

1. 该解析结果是否由HTTPDNS服务提供
    - 根据端上策略不同，解析结果可能有三种不同的来源：HTTPDNS、LocalDNS、持久化缓存
2. 该次HTTPDNS解析对应的网络环境及入网IP
3. 客户在该次app生命周期内是否存在IP跨区域切换的情况

# 排查场景：APM显示HTTPDNS服务质量有问题

当您接入了APM组件（即线上版端性能分析工具），APM组件中显示HTTPDNS服务出现无法连接的问题时

请您提交HTTPDNS工单，并提供以下信息

- APM系统截图：连接失败或返回错误时的URL地址、出现的错误信息、以及问题出现的时间
    - URL中会包含sessionId、网络情况等信息
    - 虽然该请求可能没有到达服务器，但请求时的网络情况(net)会被添加到URL中，并由您接入的APM系统记录

我们会根据您提供的信息提供以下调查结果

1. 协助您分析该次失败请求的原因
2. 该次app生命周期内的相关请求记录，以便分析是否
    i. 判断是否存在网络切换的情况
    ii. 是否同一时间有成功的请求

# iOS WebView 中的 Cookie 处理业务场景“IP直连”方案说明

# WebView 中的 Cookie 处理业务场景“IP直连”方案说明

## 概述

本文将讨论下类似这样的问题：

- WKWebView 对于 Cookie 的管理一直是它的短板，那么 iOS11 是否有改进，如果有，如何利用这样的改进？
    - 采用 IP 直连方案后，服务端返回的 Cookie 里的 Domain 字段也会使用 IP 。如果 IP 是动态的，就有可能导致一些问题：由于许多 H5 业务都依赖于 Cookie 作登录态校验，而 WKWebView 上请求不会自动携带 Cookie。

## WKWebView 使用 NSURLProtocol 拦截请求无法获取 Cookie 信息

iOS11推出了新的 API WKHTTPCookieStore 可以用来拦截 WKWebView 的 Cookie 信息

用法示例如下：

```
   WKHTTPCookieStore *cookieStroe = self.webView.configuration.websiteDataStore.httpCookieStore;
//get cookies
[cookieStroe getAllCookies:^(NSArray<NSHTTPCookie *> * _Nonnull cookies) {
NSLog(@"All cookies %@",cookies);
}];

//set cookie
NSMutableDictionary *dict = [NSMutableDictionary dictionary];
```

```
dict[NSHTTPCookieName] = @"userid";
dict[NSHTTPCookieValue] = @"123";
dict[NSHTTPCookieDomain] = @"xxxx.com";
dict[NSHTTPCookiePath] = @"/";

NSHTTPCookie *cookie = [NSHTTPCookie cookieWithProperties:dict];
[cookieStroe setCookie:cookie completionHandler:^{
NSLog(@"set cookie");
}];

//delete cookie
[cookieStroe deleteCookie:cookie completionHandler:^{
NSLog(@"delete cookie");
}];
```

## 利用 iOS11 API WKHTTPCookieStore 解决 WKWebView 首次请求不携带 Cookie 的问题

问题说明：由于许多 H5 业务都依赖于 Cookie 作登录态校验，而 WKWebView 上请求不会自动携带 Cookie。比如，如果你在Native层面做了登陆操作，获取了Cookie信息，也使用 NSHTTPCookieStorage 存到了本地，但是使用 WKWebView 打开对应网页时，网页依然处于未登陆状态。如果是登陆也在 WebView 里做的，就不会有这个问题。

iOS11 的 API 可以解决该问题，只要是存在 WKHTTPCookieStore 里的 cookie，WKWebView 每次请求都会携带，存在 NSHTTPCookieStorage 的cookie，并不会每次都携带。于是会发生首次 WKWebView 请求不携带 Cookie 的问题。

解决方法：

在执行 -[WKWebView loadReques:] 前将 NSHTTPCookieStorage 中的内容复制到 WKHTTPCookieStore 中，以此来达到 WKWebView Cookie 注入的目的。示例代码如下：

```
    [self copyNSHTTPCookieStorageToWKHTTPCookieStoreWithCompletionHandler:^{
NSURL *url = [NSURL URLWithString:@"https://www.v2ex.com"];
NSURLRequest *request = [NSURLRequest requestWithURL:url];
[_webView loadRequest:request];
}];



- (void)copyNSHTTPCookieStorageToWKHTTPCookieStoreWithCompletionHandler:(nullable void
(^)())theCompletionHandler; {
NSArray *cookies = [[NSHTTPCookieStorage sharedHTTPCookieStorage] cookies];
WKHTTPCookieStore *cookieStroe = self.webView.configuration.websiteDataStore.httpCookieStore;
if (cookies.count == 0) {
!theCompletionHandler ?: theCompletionHandler();
return;
}
for (NSHTTPCookie *cookie in cookies) {
[cookieStroe setCookie:cookie completionHandler:^{
if ([[cookies lastObject] isEqual:cookie]) {
!theCompletionHandler ?: theCompletionHandler();
```

```
return;
}
}];
}
}
```

这个是 iOS11 的API，针对iOS11之前的系统，需要另外处理。

# 利用 iOS11 之前的 API 解决 WKWebView 首次请求不携带 Cookie 的问题

通过让所有 WKWebView 共享同一个 WKProcessPool 实例，可以实现多个 WKWebView 之间共享 Cookie（session Cookie and persistent Cookie）数据。不过 WKWebView WKProcessPool 实例在 app 杀进程重启后会被重置，导致 WKProcessPool 中的 Cookie、session Cookie 数据丢失，目前也无法实现 WKProcessPool 实例本地化保存。可以采取 cookie 放入 Header 的方法来做。

```
 WKWebView * webView = [WKWebView new];
NSMutableURLRequest * request = [NSMutableURLRequest requestWithURL:[NSURL
URLWithString:@"http://xxx.com/login"]];
[request addValue:@"skey=skeyValue" forHTTPHeaderField:@"Cookie"];
[webView loadRequest:request];
```

其中对于 skey=skeyValue 这个cookie值的获取，也可以统一通过domain获取，获取的方法，可以参照下面的工具类：

```
HTTPDNSCookieManager.h

#ifndef HTTPDNSCookieManager_h
#define HTTPDNSCookieManager_h

// URL匹配Cookie规则
typedef BOOL (^HTTPDNSCookieFilter)(NSHTTPCookie *, NSURL *);

@interface HTTPDNSCookieManager : NSObject

+ (instancetype)sharedInstance;

/**
指定URL匹配Cookie策略

@param filter 匹配器
*/
- (void)setCookieFilter:(HTTPDNSCookieFilter)filter;

/**
处理HTTP Reponse携带的Cookie并存储

@param headerFields HTTP Header Fields
@param URL 根据匹配策略获取查找URL关联的Cookie
@return 返回添加到存储的Cookie
```

```
*/
- (NSArray<NSHTTPCookie *> *)handleHeaderFields:(NSDictionary *)headerFields forURL:(NSURL *)URL;

/**
匹配本地Cookie存储，获取对应URL的request cookie字符串

@param URL 根据匹配策略指定查找URL关联的Cookie
@return 返回对应URL的request Cookie字符串
*/
- (NSString *)getRequestCookieHeaderForURL:(NSURL *)URL;

/**
删除存储cookie

@param URL 根据匹配策略查找URL关联的cookie
@return 返回成功删除cookie数
*/
- (NSInteger)deleteCookieForURL:(NSURL *)URL;

@end

#endif /* HTTPDNSCookieManager_h */

HTTPDNSCookieManager.m
#import <Foundation/Foundation.h>
#import "HTTPDNSCookieManager.h"

@implementation HTTPDNSCookieManager
{
HTTPDNSCookieFilter cookieFilter;
}

- (instancetype)init {
if (self = [super init]) {
/**
此处设置的Cookie和URL匹配策略比较简单，检查URL.host是否包含Cookie的domain字段
通过调用setCookieFilter接口设定Cookie匹配策略，
比如可以设定Cookie的domain字段和URL.host的后缀匹配 | URL是否符合Cookie的path设定
细节匹配规则可参考RFC 2965 3.3节
*/
cookieFilter = ^BOOL(NSHTTPCookie *cookie, NSURL *URL) {
if ([URL.host containsString:cookie.domain]) {
return YES;
}
return NO;
};
}
return self;
}

+ (instancetype)sharedInstance {
static id singletonInstance = nil;
static dispatch_once_t onceToken;
dispatch_once(&onceToken, ^{
if (!singletonInstance) {
singletonInstance = [[super allocWithZone:NULL] init];
```

```
}
});
return singletonInstance;
}

+ (id)allocWithZone:(struct _NSZone *)zone {
return [self sharedInstance];
}

- (id)copyWithZone:(struct _NSZone *)zone {
return self;
}

- (void)setCookieFilter:(HTTPDNSCookieFilter)filter {
if (filter != nil) {
cookieFilter = filter;
}
}

- (NSArray<NSHTTPCookie *> *)handleHeaderFields:(NSDictionary *)headerFields forURL:(NSURL *)URL {
NSArray *cookieArray = [NSHTTPCookie cookiesWithResponseHeaderFields:headerFields forURL:URL];
if (cookieArray != nil) {
NSHTTPCookieStorage *cookieStorage = [NSHTTPCookieStorage sharedHTTPCookieStorage];
for (NSHTTPCookie *cookie in cookieArray) {
if (cookieFilter(cookie, URL)) {
NSLog(@"Add a cookie: %@", cookie);
[cookieStorage setCookie:cookie];
}
}
}
return cookieArray;
}

- (NSString *)getRequestCookieHeaderForURL:(NSURL *)URL {
NSArray *cookieArray = [self searchAppropriateCookies:URL];
if (cookieArray != nil && cookieArray.count > 0) {
NSDictionary *cookieDic = [NSHTTPCookie requestHeaderFieldsWithCookies:cookieArray];
if ([cookieDic objectForKey:@"Cookie"]) {
return cookieDic[@"Cookie"];
}
}
return nil;
}

- (NSArray *)searchAppropriateCookies:(NSURL *)URL {
NSMutableArray *cookieArray = [NSMutableArray array];
NSHTTPCookieStorage *cookieStorage = [NSHTTPCookieStorage sharedHTTPCookieStorage];
for (NSHTTPCookie *cookie in [cookieStorage cookies]) {
if (cookieFilter(cookie, URL)) {
NSLog(@"Search an appropriate cookie: %@", cookie);
[cookieArray addObject:cookie];
}
}
return cookieArray;
}
```

```
- (NSInteger)deleteCookieForURL:(NSURL *)URL {
int delCount = 0;
NSHTTPCookieStorage *cookieStorage = [NSHTTPCookieStorage sharedHTTPCookieStorage];
for (NSHTTPCookie *cookie in [cookieStorage cookies]) {
if (cookieFilter(cookie, URL)) {
NSLog(@"Delete a cookie: %@", cookie);
[cookieStorage deleteCookie:cookie];
delCount++;
}
}
return delCount;
}

@end
```

使用方法示例：

发送请求

```
 WKWebView * webView = [WKWebView new];
NSMutableURLRequest * request = [NSMutableURLRequest requestWithURL:[NSURL
URLWithString:@"http://xxx.com/login"]];
NSString *value = [[HTTPDNSCookieManager sharedInstance] getRequestCookieHeaderForURL:url];
[request setValue:value forHTTPHeaderField:@"Cookie"];
[webView loadRequest:request];
```

接收处理请求：

```
    NSURLSessionTask *task = [session dataTaskWithRequest:request completionHandler:^(NSData *data,
NSURLResponse *response, NSError *error) {
if (!error) {
NSHTTPURLResponse *httpResponse = (NSHTTPURLResponse *)response;
// 解析 HTTP Response Header，存储cookie
[[HTTPDNSCookieManager sharedInstance] handleHeaderFields:[httpResponse allHeaderFields] forURL:url];
}
}];
[task resume];
```

通过 document.cookie 设置 Cookie 解决后续页面(同域)Ajax、iframe 请求的 Cookie 问题；

```
WKUserContentController* userContentController = [WKUserContentController new];
WKUserScript * cookieScript = [[WKUserScript alloc] initWithSource: @"document.cookie = 'skey=skeyValue';"
injectionTime:WKUserScriptInjectionTimeAtDocumentStart forMainFrameOnly:NO];
[userContentController addUserScript:cookieScript];
```

# Cookie包含动态 IP 导致登陆失效问题

关于COOKIE失效的问题，假如客户端登录 session 存在 COOKIE，此时这个域名配置了多个IP，使用域名访问会读对应域名的COOKIE，使用IP访问则去读对应IP的COOKIE，假如前后两次使用同一个域名配置的不同

IP访问，会导致COOKIE的登录session失效，

如果APP里面的webview页面需要用到系统COOKIE存的登录session，之前APP所有本地网络请求使用域名访问，是可以共用COOKIE的登录session的，但现在本地网络请求使用httpdns后改用IP访问，导致还使用域名访问的webview读不到系统COOKIE存的登录session了（系统COOKIE对应IP了）。IP直连后，服务端返回Cookie包含动态 IP 导致登陆失效。

使用IP访问后，服务端返回的cookie也是IP。导致可能使用对应的域名访问，无法使用本地cookie，或者使用隶属于同一个域名的不同IP去访问，cookie也对不上，导致登陆失效，是吧。

我这边的思路是这样的，

- 应该得干预cookie的存储，基于域名。
- 根源上，api域名返回单IP

第二种思路将失去DNS调度特性，故不考虑。第一种思路更为可行。

## 基于 iOS11 API WKHTTPCookieStore 来解决 WKWebView 的 Cookie 管理问题

当每次服务端返回cookie后，在存储前都进行下改造，使用域名替换下IP。之后虽然每次网络请求都是使用IP访问，但是host我们都手动改为了域名，这样本地存储的 cookie 也就能对得上了。

代码演示：

在网络请求成功后，或者加载网页成功后，主动将本地的 domain 字段为 IP 的 Cookie 替换 IP 为 host 域名地址。

```
- (void)updateWKHTTPCookieStoreDomainFromIP:(NSString *)IP toHost:(NSString *)host {
WKHTTPCookieStore *cookieStroe = self.webView.configuration.websiteDataStore.httpCookieStore;
[cookieStroe getAllCookies:^(NSArray<NSHTTPCookie *> * _Nonnull cookies) {
[[cookies copy] enumerateObjectsUsingBlock:^(NSHTTPCookie * _Nonnull cookie, NSUInteger idx, BOOL * _Nonnull
stop) {
if ([cookie.domain isEqualToString:IP]) {
NSMutableDictionary<NSHTTPCookiePropertyKey, id> *dict = [NSMutableDictionary
dictionaryWithDictionary:cookie.properties];
dict[NSHTTPCookieDomain] = host;
NSHTTPCookie *newCookie = [NSHTTPCookie cookieWithProperties:[dict copy]];
[cookieStroe setCookie:newCookie completionHandler:^{
[self logCookies];
[cookieStroe deleteCookie:cookie
completionHandler:^{
[self logCookies];
}];
}];
}
}];
}];
}
```

iOS11中也提供了对应的 API 供我们来处理替换 Cookie 的时机，那就是下面的API：

```
@protocol WKHTTPCookieStoreObserver <NSObject>
@optional
- (void)cookiesDidChangeInCookieStore:(WKHTTPCookieStore *)cookieStore;
@end


//WKHTTPCookieStore
/*! @abstract Adds a WKHTTPCookieStoreObserver object with the cookie store.
@param observer The observer object to add.
@discussion The observer is not retained by the receiver. It is your responsibility
to unregister the observer before it becomes invalid.
*/
- (void)addObserver:(id<WKHTTPCookieStoreObserver>)observer;

/*! @abstract Removes a WKHTTPCookieStoreObserver object from the cookie store.
@param observer The observer to remove.
*/
- (void)removeObserver:(id<WKHTTPCookieStoreObserver>)observer;
```

用法如下：

```
@interface WebViewController ()<WKHTTPCookieStoreObserver>
- (void)viewDidLoad {
[super viewDidLoad];
[NSURLProtocol registerClass:[WebViewURLProtocol class]];
NSHTTPCookieStorage *cookieStorage = [NSHTTPCookieStorage sharedHTTPCookieStorage];
[cookieStorage setCookieAcceptPolicy:NSHTTPCookieAcceptPolicyAlways];
WKHTTPCookieStore *cookieStroe = self.webView.configuration.websiteDataStore.httpCookieStore;
[cookieStroe addObserver:self];

[self.view addSubview:self.webView];
//... ...
}

#pragma mark -
#pragma mark - WKHTTPCookieStoreObserver Delegate Method

- (void)cookiesDidChangeInCookieStore:(WKHTTPCookieStore *)cookieStore {
[self updateWKHTTPCookieStoreDomainFromIP:CYLIP toHost:CYLHOST];
}
```

-updateWKHTTPCookieStoreDomainFromIP 方法的实现，在上文已经给出。

这个方案需要客户端维护一个IP —> HOST的映射关系，需要能从 IP 反向查找到 HOST，这个维护成本还时挺高的。下面介绍下，更通用的方法，也是iOS11 之前的处理方法：

## iOS11 之前的处理方法：NSURLProtocal拦截后，手动管理 Cookie 的存储：

步骤：做 IP 替换时将原始 URL 保存到 Header 中

```
+ (NSURLRequest *)canonicalRequestForRequest:(NSURLRequest *)request {
NSMutableURLRequest *mutableReq = [request mutableCopy];
NSString *originalUrl = mutableReq.URL.absoluteString;
NSURL *url = [NSURL URLWithString:originalUrl];
// 异步接口获取IP地址
NSString *ip = [[HttpDnsService sharedInstance] getIpByHostAsync:url.host];
if (ip) {
NSRange hostFirstRange = [originalUrl rangeOfString:url.host];
if (NSNotFound != hostFirstRange.location) {
NSString *newUrl = [originalUrl stringByReplacingCharactersInRange:hostFirstRange withString:ip];
mutableReq.URL = [NSURL URLWithString:newUrl];
[mutableReq setValue:url.host forHTTPHeaderField:@"host"];
// 添加originalUrl保存原始URL
[mutableReq addValue:originalUrl forHTTPHeaderField:@"originalUrl"];
}
}
NSURLRequest *postRequestIncludeBody = [mutableReq cyl_getPostRequestIncludeBody];
return postRequestIncludeBody;
}
```

然后获取到数据后，手动管理 Cookie：

```
- (void)handleCookiesFromResponse:(NSURLResponse *)response {
NSString *originalURLString = [self.request valueForHTTPHeaderField:@"originalUrl"];
if ([response isKindOfClass:[NSHTTPURLResponse class]]) {
NSHTTPURLResponse *httpResponse = (NSHTTPURLResponse *)response;
NSDictionary<NSString *, NSString *> *allHeaderFields = httpResponse.allHeaderFields;
if (originalURLString && originalURLString.length > 0) {
NSArray *cookies = [NSHTTPCookie cookiesWithResponseHeaderFields:allHeaderFields forURL: [[NSURL alloc]
initWithString:originalURLString]];
if (cookies && cookies.count > 0) {
NSURL *originalURL = [NSURL URLWithString:originalURLString];
[[NSHTTPCookieStorage sharedHTTPCookieStorage] setCookies:cookies forURL:originalURL mainDocumentURL:nil];
}
}
}
}

- (void)URLSession:(NSURLSession *)session task:(NSURLSessionTask *)task
willPerformHTTPRedirection:(NSHTTPURLResponse *)response newRequest:(NSURLRequest *)newRequest
completionHandler:(void (^)(NSURLRequest *))completionHandler {
NSString *location = response.allHeaderFields[@"Location"];
NSURL *url = [[NSURL alloc] initWithString:location];
NSMutableURLRequest *mRequest = [newRequest mutableCopy];
mRequest.URL = url;
if (location && location.length > 0) {
if ([[newRequest.HTTPMethod lowercaseString] isEqualToString:@"post"]) {
// POST重定向为GET
mRequest.HTTPMethod = @"GET";
mRequest.HTTPBody = nil;
}
[mRequest setValue:nil forHTTPHeaderField:@"host"];
// 在这里为 request 添加 cookie 信息。
[self handleCookiesFromResponse:response];
```

```
[XXXURLProtocol removePropertyForKey:XXXURLProtocolHandledKey inRequest:mRequest];
completionHandler(mRequest);
} else{
completionHandler(mRequest);
}
}
```

发送请求前，向请求中添加Cookie信息：

```
+ (void)handleCookieWithRequest:(NSMutableURLRequest *)request {
NSString* originalURLString = [request valueForHTTPHeaderField:@"originalUrl"];
if (!originalURLString || originalURLString.length == 0) {
return;
}
NSArray *cookies = [[NSHTTPCookieStorage sharedHTTPCookieStorage] cookies];
if (cookies && cookies.count >0) {
NSDictionary *cookieHeaders = [NSHTTPCookie requestHeaderFieldsWithCookies:cookies];
NSString *cookieString = [cookieHeaders objectForKey:@"Cookie"];
[request addValue:cookieString forHTTPHeaderField:@"Cookie"];
}
}

+ (NSURLRequest *)canonicalRequestForRequest:(NSURLRequest *)request {
NSMutableURLRequest *mutableReq = [request mutableCopy];
//...
[self handleCookieWithRequest:mutableReq];
return [mutableReq copy];
}
```

**相关的文章：**

- Apple Docs > URL Session Programming Guide > Cookies and Custom Protocols
    - 《WKWebView 那些坑》
    - 《HTTPDNS域名解析场景下如何使用Cookie？》

# iOS WebView 业务场景“IP直连”方案说明

## 概述

本文主要介绍，防 DNS 污染方案在 WebView 场景下所遇到的一些问题，及解决方案。

### WKWebView 无法使用 NSURLProtocol 拦截请求

针对该问题方案如下：

1. 换用 UIWebView
2. 使用私有API进行注册拦截

换用 UIWebView 方案不做赘述，说明下使用私有API进行注册拦截的方法：

```
//注册自己的protocol
[NSURLProtocol registerClass:[CustomProtocol class]];

//创建WKWebview
WKWebViewConfiguration * config = [[WKWebViewConfiguration alloc] init];
WKWebView * wkWebView = [[WKWebView alloc] initWithFrame:CGRectMake(0, 0, [UIScreen
mainScreen].bounds.size.width, [UIScreen mainScreen].bounds.size.height) configuration:config];
[wkWebView loadRequest:webViewReq];
[self.view addSubview:wkWebView];

//注册scheme
Class cls = NSClassFromString(@"WKBrowsingContextController");
SEL sel = NSSelectorFromString(@"registerSchemeForCustomProtocol:");
if ([cls respondsToSelector:sel]) {
// 通过http和https的请求，同理可通过其他的Scheme 但是要满足ULR Loading System
[cls performSelector:sel withObject:@"http"];
[cls performSelector:sel withObject:@"https"];
}
```

使用私有 API 的另一风险是兼容性问题，比如上面的 browsingContextController 就只能在 iOS 8.4 以后才能用，反注册 scheme 的方法 unregisterSchemeForCustomProtocol: 也是在 iOS 8.4 以后才被添加进来的，要支持 iOS 8.0 ~ 8.3 机型的话，只能通过动态生成字符串的方式拿到 WKBrowsingContextController，而且还不能反注册，不过这些问题都不大。至于向后兼容，这个也不用太担心，因为 iOS 发布新版本之前都会有开发者预览版的，那个时候再测一下也不迟。对于本文的例子来说，如果将来哪个 iOS 版本移除了这个 API，那很可能是因为官方提供了完整的解决方案，到那时候自然也不需要本文介绍的方法了。

注意避免执行太晚，如果在 - (void)viewDidLoad 中注册，可能会因为注册太晚，引发问题。建议在 +load 方法中执行。

然后同样会遇到 《HTTPS SNI 业务场景“IP直连”方案说明》 里提到的各种 NSURLProtocol 相关的问题，可以参照里面的方法解决。

# Cookie相关问题

单独成篇：《WebView 中的 Cookie 处理业务场景“IP直连”方案说明》。

# 302重定向问题

上面提到的 Cookie 方案无法解决302请求的 Cookie 问题，比如，第一个请求是 http://www.a.com ，我们通过在 request header 里带上 Cookie 解决该请求的 Cookie 问题，接着页面302跳转到 http://www.b.com ，这个时候 http://www.b.com 这个请求就可能因为没有携带 cookie 而无法访问。当然，由于每一次页面跳转前都会调用回调函数：

```
 - (void)webView:(WKWebView *)webView decidePolicyForNavigationAction:(WKNavigationAction
*)navigationAction decisionHandler:(void (^)(WKNavigationActionPolicy))decisionHandler;
```

可以在该回调函数里拦截302请求，copy request，在 request header 中带上 cookie 并重新 loadRequest。不过这种方法依然解决不了页面 iframe 跨域请求的 Cookie 问题，毕竟 -[WKWebView loadRequest:] 只适合加载 mainFrame 请求。

## 参考链接

**相关的库：**

- GitHub：WebViewProxy
- GitHub：NSURLProtocol-WebKitSupport
- GitHub：happy-dns-objc
- Chrome For iOS

**相关的文章：**

- 《NSURLProtocol对WKWebView的处理》
- 《可能是最全的iOS端HttpDns集成方案》
- 《WKWebView 那些坑》
- 《让 WKWebView 支持 NSURLProtocol》
- 《WKWebView的使用和各种坑的解决方法（OC + Swift）》

**可以参考的Demo：**

- HybirdWKWebVIew
- 《WWDC 2017-WKWebView 新功能》

# WebView业务场景“IP直连”方案说明

## 1. 背景说明

在App WebView加载网络请求场景下，Android/iOS系统可基于系统API进行网络请求拦截，并实现自定义逻辑注入，如使用HTTPDNS进行基于IP的直连请求。但iOS系统在Webview场景下拦截网络请求后，需要自行接管基于IP的网络请求的发送、数据接收、页面重定向、页面解码、Cookie、缓存等逻辑；Android除了上述iOS遇到的问题外，不同版本的ROM的网络请求拦截能力还存在差异，比如低版本Android ROM基于系统API拦截的网络请求会丢失请求方法、Body信息等。综合来看，Webview场景下HTTPDNS的使用（IP直连进行网络请求）门槛比较高，移动操作系统针对这个场景的支持粒度很粗，且存在一些缺陷，需要开发者具备较强的网络/OS Framework的代码级掌控能力来规避和优化上述问题。

# 2. 方案概述

## 2.1 Android

- 通过以下API对WebView进行配置；

```
void setWebViewClient (WebViewClient client);
```

- 通过重写WebViewClient中以下方法拦截WebView的网络请求，获取请求URL.host进行HTTPDNS域名解析，解析完成后处理方式和普通网络请求一致，替换URL.host字段，设置HTTP Header Host域，最后返回新请求对应的WebResourceResponse。

```
/**
* 拦截WebView网络请求（Android API < 21）
* 只能拦截网络请求的URL，请求方法、请求内容等无法拦截
*/
public WebResourceResponse shouldInterceptRequest(WebView view,
String url);

/**
* 拦截WebView网络请求（Android API >= 21）
* 通过解析WebResourceRequest对象获取网络请求相关信息
*/
public WebResourceResponse shouldInterceptRequest(WebView view,
WebResourceRequest request);
```

- 使用方式参考Android WebClient API。

## 2.2 iOS

- 基于NSURLProtocol可拦截iOS系统上基于上层网络库NSURLConnection/NSURLSession发出的网络请求，WebView发出的请求同样包含在内；
- 通过以下接口注册自定义NSURLProtocol，用于拦截WebView上层网络请求，并创建新的网络请求接管数据发送、接收、重定向等处理逻辑，将结果反馈给原始请求。

```
[NSURLProtocol registerClass:[CustomProtocol class]];
```

- 自定义NSURLProtocol处理过程概述：
    - 在canInitWithRequest中过滤要需要做HTTPDNS域名解析的请求；
    - 请求拦截后，做HTTPDNS域名解析；
    - 解析完成后，同普通请求一样，替换URL.host字段，替换HTTP Header Host域，并接管该请求的数据发送、接收、重定向等处理；
    - 通过NSURLProtocolClient的接口，将请求处理结果反馈到WebView原始请求。
- NSURLProtocol使用参考Apple NSURLProtocol API，苹果官方示例代码参考Apple Sample Code - CustomHTTPProtocol。

# Android端WEEX + HTTPDNS 最佳实践

由于WebView并未暴露处设置DNS的接口，因而在WebView场景下使用HttpDns存在很多无法限制，但如果接入WEEX,则可以较好地植入HTTPDNS，本文主要介绍在WEEX场景下接入HTTPDNS的方案细节。

在WEEX运行时环境下，所有的逻辑最终都会转换到Native Runtime中执行，网络请求也不例外。同时WEEX也提供了自定义相应实现的接口，通过重写网络请求适配器，我们可以较为简单地接入HTTPDNS。在WEEX运行环境中，主要有两种网络请求：

- 通过Stream进行的网络请求
- 标签指定的加载图片的网络请求

## 1 Stream网络请求 + HTTPDNS

Stream网络请求在Android端最终会通过DefaultWXHttpAdapter完成，同时WEEX也提供了相应的接口自定义网络请求适配器。具体的逻辑如下：

第一步：创建自定义网络请求适配器，实现IWXHttpAdapter接口

```
public class WXHttpdnsAdatper implements IWXHttpAdapter {
@Override
public void sendRequest(final WXRequest request, final OnHttpListener listener) {
......
}
}
```

该接口需要实现sendRequest方法，该方法传入一个WXRequest对象的实例，该对象提供了以下信息：

```
public class WXRequest {
// The request parameter
public Map<String, String> paramMap;
// The request URL
public String url;
// The request method
public String method;
// The request body
public String body;
// The request time out
public int timeoutMs = WXRequest.DEFAULT_TIMEOUT_MS;
// The default timeout
public static final int DEFAULT_TIMEOUT_MS = 3000;
}
```

通过该对象我们可以获取到请求报头、URL、方法以及body。

第二步：在WEEX初始化时注册自定义网络适配器，替换默认适配器：

```
InitConfig config=new InitConfig.Builder()
.setHttpAdapter(new WXHttpdnsAdatper()) // 注册自定义网络请求适配器
.....
.build();
WXSDKEngine.initialize(this,config);
```

之后左右的网络请求都会通过WXHttpdnsAdatper实现，所以只需要在WXHttpdnsAdapter中植入HTTPDNS逻辑即可，具体逻辑可以参考如下代码：

```
public class WXHttpdnsAdatper implements IWXHttpAdapter {
.....
private void execute(Runnable runnable){
if(mExecutorService==null){
mExecutorService = Executors.newFixedThreadPool(3);
}
mExecutorService.execute(runnable);
}

@Override
public void sendRequest(final WXRequest request, final OnHttpListener listener) {
if (listener != null) {
listener.onHttpStart();
}

Log.e(TAG, "URL:" + request.url);
execute(new Runnable() {
@Override
public void run() {
WXResponse response = new WXResponse();
WXHttpdnsAdatper.IEventReporterDelegate reporter = getEventReporterDelegate();
try {
// 创建连接
HttpURLConnection connection = openConnection(request, listener);
reporter.preConnect(connection, request.body);
.....
} catch (IOException |IllegalArgumentException e) {
.....
}
}
});
}

private HttpURLConnection openConnection(WXRequest request, OnHttpListener listener) throws IOException {
URL url = new URL(request.url);
// 创建一个植入HTTPDNS的连接
HttpURLConnection connection = openHttpDnsConnection(request, request.url, listener, null);
return connection;
}



private HttpURLConnection openHttpDnsConnection(WXRequest request, String path, OnHttpListener listener,
```

```
String reffer) {
HttpURLConnection conn = null;
URL url = null;
try {
url = new URL(path);
conn = (HttpURLConnection) url.openConnection();
// 创建一个接入httpdns的连接
HttpURLConnection tmpConn = httpDnsConnection(url, path);
......

int code = conn.getResponseCode();// Network block
Log.e(TAG, "code:" + code);
// SNI场景下通常涉及重定向，重新建立新连接
if (needRedirect(code)) {
Log.e(TAG, "need redirect");
String location = conn.getHeaderField("Location");
if (location == null) {
location = conn.getHeaderField("location");
}

if (location != null) {
if (!(location.startsWith("http://") || location
.startsWith("https://"))) {
//某些时候会省略host，只返回后面的path，所以需要补全url
URL originalUrl = new URL(path);
location = originalUrl.getProtocol() + "://"
+ originalUrl.getHost() + location;
}
Log.e(TAG, "code:" + code + "; location:" + location + "; path" + path);
return openHttpDnsConnection(request, location, listener, path);
} else {
return conn;
}
} else {
// redirect finish.
Log.e(TAG, "redirect finish");
return conn;
}
}
......
return conn;
}

private HttpURLConnection httpDnsConnection(URL url, String path) {
HttpURLConnection conn = null;
// 通过HTTPDNS SDK接口获取IP
String ip = HttpDnsManager.getInstance().getHttpDnsService().getIpByHostAsync(url.getHost());
if (ip != null) {
// 通过HTTPDNS获取IP成功，进行URL替换和HOST头设置
Log.d(TAG, "Get IP: " + ip + " for host: " + url.getHost() + " from HTTPDNS successfully!");
String newUrl = path.replaceFirst(url.getHost(), ip);
try {
conn = (HttpURLConnection) new URL(newUrl).openConnection();
} catch (IOException e) {
return null;
}
```

```
// 设置HTTP请求头Host域
conn.setRequestProperty("Host", url.getHost());

// HTTPS场景
if (conn instanceof HttpsURLConnection) {
final HttpsURLConnection httpsURLConnection = (HttpsURLConnection)conn;
WXTlsSniSocketFactory sslSocketFactory = new WXTlsSniSocketFactory((HttpsURLConnection) conn);

// SNI场景，创建SSLScocket解决SNI场景下的证书问题
conn.setInstanceFollowRedirects(false);
httpsURLConnection.setSSLSocketFactory(sslSocketFactory);
// HTTPS场景，证书校验
httpsURLConnection.setHostnameVerifier(new HostnameVerifier() {
@Override
public boolean verify(String hostname, SSLSession session) {
String host = httpsURLConnection.getRequestProperty("Host");
Log.e(TAG, "verify host:" + host);
if (null == host) {
host = httpsURLConnection.getURL().getHost();
}
return HttpsURLConnection.getDefaultHostnameVerifier().verify(host, session);
}
});
}
} else {
Log.e(TAG, "no corresponding ip found, return null");
return null;
}

return conn;
}
}
```

## 1.2 <image>网络请求 + HTTPDNS

WEEX并没有提供默认的图片适配器实现，所以用户必须自行实现才能完成图片请求逻辑，具体步骤分为以下几步：

第一步：自定义图片请求适配器，实现IWXImgLoaderAdapter接口

```
public class HttpDnsImageAdapter implements IWXImgLoaderAdapter {
@Override
public void setImage(final String url, final ImageView view, WXImageQuality quality, final WXImageStrategy
strategy) {
......
}
}
```

第二步：在WEEX初始化时注册该图片适配器：

```
    private void initWeex() {
InitConfig config=new InitConfig.Builder()
```

```
.setImgAdapter(new HttpDnsImageAdapter())
.build();
WXSDKEngine.initialize(this,config);
.....
}
```

所以同WXHttpdnsAdatper一样，我们只需在HttpDnsImageAdapter植入HTTPDNS逻辑即可。具体代码可参考：

```
/**
* Created by liyazhou on 2017/10/22.
*/

public class WXHttpDnsImageAdapter implements IWXImgLoaderAdapter {

@Override
public void setImage(final String url, final ImageView view, WXImageQuality quality, final WXImageStrategy
strategy) {
Log.e(TAG, "img url:" + url);
execute(new Runnable() {
@Override
public void run() {
.....
HttpURLConnection conn = null;
try {
conn = createConnection(url);
....
// 将得到的数据转化成InputStream
InputStream is = conn.getInputStream();
// 将InputStream转换成Bitmap
final Bitmap bitmap = BitmapFactory.decodeStream(is);
WXSDKManager.getInstance().postOnUiThread(new Runnable() {
@Override
public void run() {
view.setImageBitmap(bitmap);
}
}, 0);
.....
}
});
}

protected HttpURLConnection createConnection(String originalUrl) throws IOException {
mHttpDnsService = HttpDnsManager.getInstance().getHttpDnsService();
if (mHttpDnsService == null) {
URL url = new URL(originalUrl);
return (HttpURLConnection) url.openConnection();
} else {
return httpDnsRequest(originalUrl, null);
}
}

private HttpURLConnection httpDnsRequest(String path, String reffer) {
HttpURLConnection httpDnsConn = null;
HttpURLConnection originalConn = null;
```

```
URL url = null;
try {
url = new URL(path);
originalConn = (HttpURLConnection) url.openConnection();
// 异步接口获取IP
String ip = HttpDnsManager.getInstance().getHttpDnsService().getIpByHostAsync(url.getHost());
if (ip != null) {
// 通过HTTPDNS获取IP成功，进行URL替换和HOST头设置
Log.d(TAG, "Get IP: " + ip + " for host: " + url.getHost() + " from HTTPDNS successfully!");
String newUrl = path.replaceFirst(url.getHost(), ip);
httpDnsConn = (HttpURLConnection) new URL(newUrl).openConnection();

// 设置HTTP请求头Host域
httpDnsConn.setRequestProperty("Host", url.getHost());

// HTTPS场景
if (httpDnsConn instanceof HttpsURLConnection) {
final HttpsURLConnection httpsURLConnection = (HttpsURLConnection)httpDnsConn;
WXTlsSniSocketFactory sslSocketFactory = new WXTlsSniSocketFactory((HttpsURLConnection) httpDnsConn);

// sni场景，创建SSLScocket解决SNI场景下的证书问题
httpsURLConnection.setSSLSocketFactory(sslSocketFactory);
// https场景，证书校验
httpsURLConnection.setHostnameVerifier(new HostnameVerifier() {
@Override
public boolean verify(String hostname, SSLSession session) {
String host = httpsURLConnection.getRequestProperty("Host");
if (null == host) {
host = httpsURLConnection.getURL().getHost();
}
return HttpsURLConnection.getDefaultHostnameVerifier().verify(host, session);
}
});
}
} else {
return originalConn;
}

......

int code = httpDnsConn.getResponseCode();// Network block
if (needRedirect(code)) {
String location = httpDnsConn.getHeaderField("Location");
if (location == null) {
location = httpDnsConn.getHeaderField("location");
}

if (location != null) {
if (!(location.startsWith("http://") || location
.startsWith("https://"))) {
//某些时候会省略host，只返回后面的path，所以需要补全url
URL originalUrl = new URL(path);
location = originalUrl.getProtocol() + "://"
+ originalUrl.getHost() + location;
}
Log.e(TAG, "code:" + code + "; location:" + location + "; path" + path);
```

```
return httpDnsRequest(location, path);
} else {
return originalConn;
}
}
return originalConn;
}
}
```

上述方案详细代码建：WeexAndroid