HTTPDNS

最佳实践

最佳实践

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。

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。

1. 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。

2. 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);

3. 实践

3.1 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);
}
```

3.2 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

3.2.1 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);
}
```

3.2.2 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());
}
}
```

3.2.3 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);
}
});
}
```

3.2.4 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;
}
}
```

3.2.5 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");
.....
}
```

3.2.6 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：

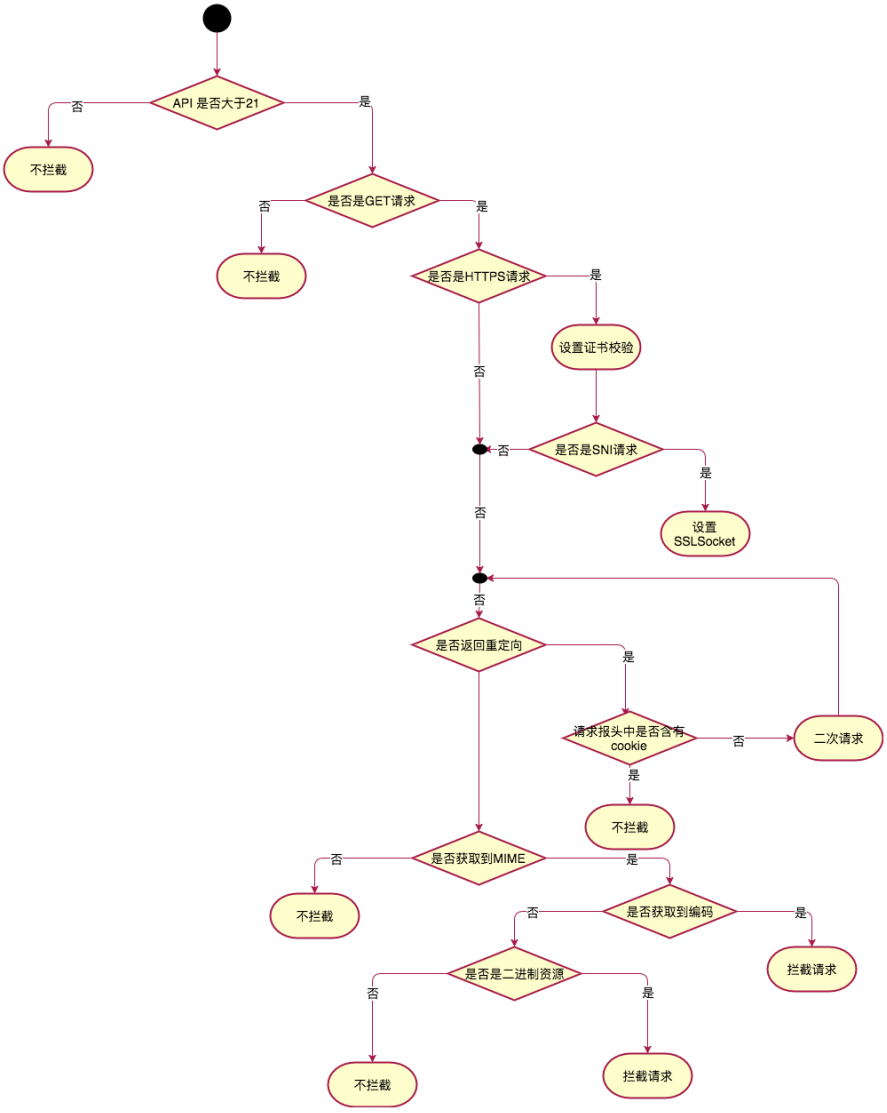
```
text/html;charset=utf-8
```

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;
}
}
```

4 总结

综上所述，在WebView场景下的请求拦截逻辑如下所示：



4.1 【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

4.2【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

4.3 完整代码

HTTPDNS+WebView最佳实践完整代码请参考：GithubDemo

请参考云栖社区文档HTTPDNS域名解析场景下如何使用Cookie？

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

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。