HttpURLConnection接入HTTPDNS SDK

以下代码片段摘自SDK使用Sample（HttpDnsSample目录），完整代码请参考使用Sample

HttpURLConnection当前没有提供注入DNS实现的API，只能通过替换URL的方式来接入HTTPDNS SDK

HttpURLConnection替换URL接入HTTPDNS SDK

如<<HTTPDNS Android接入文档>>所述，对于**需要使用到域名信息**的网络请求，我们需要针对HTTP和HTTPS请求分别进行兼容

示例代码如下：

```
// HttpUrlConnectionHelper.kt
internal object HttpUrlConnectionHelper {

    private const val HOST_HEADER_KEY = "Host"

    private val hostname2VerifierMap by lazy {
        Collections.synchronizedMap(CollectionCompat.createMap<String,
HostnameVerifier>())
    }
    private val hostname2SSLSocketFactoryMap by lazy {
        Collections.synchronizedMap(CollectionCompat.createMap<String,
SSLSocketFactory>())
    }

    // NOTE: HTTP/HTTPS兼容实现
    fun compat4ChangeHost(urlConnection: HttpURLConnection, hostname:
String) {
        // HTTP兼容实现
        urlConnection.setRequestProperty(HOST_HEADER_KEY, hostname)
        if (urlConnection is HttpsURLConnection) {
            // HTTPS兼容实现
```

```
                urlConnection.hostnameVerifier =
                    getHostnameVerifier(hostname)
                urlConnection.sslSocketFactory =
                    getSSLSocketFactory(hostname)
            }
        }

        private fun getHostnameVerifier(hostname: String): HostnameVerifier =
            hostname2VerifierMap[hostname] ?:
HostnameVerifier4Tls(hostname).also { hostname2VerifierMap[hostname] = it }

        private fun getSSLSocketFactory(hostname: String): SSLSocketFactory =
            hostname2SSLSocketFactoryMap[hostname]
                ?: SSLSocketFactory4SniHack(hostname).also {
hostname2SSLSocketFactoryMap[hostname] = it }
    }
```

以下区分HTTP/HTTPS进行具体分析

HTTP兼容

如<<HTTPDNS Android接入文档>>所述，对于**需要使用到域名信息**的HTTP请求，我们需要通过指定报文头中的Host字段来告知服务器域名信息

示例代码如下：

```
urlConnection.setRequestProperty(HOST_HEADER_KEY, hostname)
```

HTTPS兼容

如<<HTTPDNS Android接入文档>>所述，对于**需要使用到域名信息**的HTTPS请求，除去写入Host字段之外，在TLS握手过程中我们还需要通过SNI扩展来告知服务器域名信息

兼容性说明

由于HttpURLConnection API上没有提供相关的设置SNI扩展的接口，我们只能通过比较Hack的方式（基于源码流程寻找合适的注入相关逻辑的地方）来兼容HTTPS情况。这种实现方式依赖于网络库的具体实现，一旦网络库更新，则可能不再适用。从Android源码的提交日志上看，最新的Android源码上HttpURLConnection的实现底层应该是OkHttp 2.7.5版本，以下方案还可以正常使用

实现方案

在Java的SSL相关接口上，还有一个和域名信息相关的接口HostnameVerifier，为了兼容HTTPDNS的接入，除了添加SNI扩展之外，我们也要注入HostnameVerifier的接口实现

添加SNI扩展

参考OkHttp实现（AndroidPlatform类的configureTlsExtensions方法），我们可以通过反射调用SSLSocket实现上的setHostname方法，达到设置SNI扩展的目的

示例代码如下：

```
// SSLSocketFactory4SniHack.kt
internal class SSLSocketFactory4SniHack(private val hostname: String) :
SSLSocketFactory() {

    // ...

    companion object {

        private const val SET_HOSTNAME_METHOD_NAME = "setHostname"

        // ...

        private var _setHostnameMethod: Method? = null

        private fun setHostnameMethod(sslSocket: SSLSocket): Method? {
            if (null == _setHostnameMethod) {
                // NOTE: 如果无法成功反射setHostname，则SNI扩展无法成功设置
                // 这里直接消化异常，由网络库决定如何处理网络访问失败情况
                try {
                    _setHostnameMethod = sslSocket::class.java.getMethod(
                        SET_HOSTNAME_METHOD_NAME, String::class.java)
                } catch (e: NoSuchMethodException) {
                }
            }
            return _setHostnameMethod
        }

        // ...

    }
}
```

除了明确设置SNI扩展的方法，我们还需要明确可以在哪个位置添加相关的设置SNI扩展代码 Java的SSL相关接口中，SSLSocketFactory这一抽象类用于创建SSLSocket。顾名思义，SSLSocket在Socket之上提供了对SSL/TLS协议的支持。我们可以通过重写SSLSocketFactory的createSocket(Socket s, String host, int port, boolean autoClose)方法，在创建SSLSocket之后，设置相应的SNI扩展内容HttpsURLConnection提供了setSSLSocketFactory方法，我们可以调用该方法注入SSLSocketFactory的自定义实现

示例代码如下：

```
// SSLSocketFactory4SniHack.kt
internal class SSLSocketFactory4SniHack(private val hostname: String) :
SSLSocketFactory() {
```

```
    override fun getDefaultCipherSuites(): Array<out String> =
        realSSLSocketFactory.defaultCipherSuites

    override fun getSupportedCipherSuites(): Array<out String> =
        realSSLSocketFactory.supportedCipherSuites

    override fun createSocket(host: String?, port: Int): Socket =
        realSSLSocketFactory.createSocket(host, port)

    override fun createSocket(host: String?, port: Int, localHost:
InetAddress?, localPort: Int): Socket =
        realSSLSocketFactory.createSocket(host, port, localHost, localPort)

    override fun createSocket(host: InetAddress?, port: Int): Socket =
        realSSLSocketFactory.createSocket(host, port)

    override fun createSocket(address: InetAddress?, port: Int,
localAddress: InetAddress?, localPort: Int): Socket =
        realSSLSocketFactory.createSocket(address, port, localAddress,
localPort)

    override fun createSocket(s: Socket?, host: String?, port: Int,
autoClose: Boolean): Socket {
        val sslSocket = realSSLSocketFactory.createSocket(s, host, port,
autoClose) as SSLSocket
        // NOTE: 如果无法成功setHostname, 则SNI扩展无法成功设置
        // 这里直接消化异常, 由网络库决定如何处理网络访问失败情况
        try {
            setHostnameMethod(sslSocket)
                ?.invoke(sslSocket, hostname)
        } catch (ignored: IllegalAccessException) {
        } catch (ignored: InvocationTargetException) {
        }
        // NOTE: 在OkHttp的实现上, 调用createSocket之后才会设置TLS的扩展信息, 会覆
盖我们设置的hostname信息, 这里我们直接建立连接, 保证设置生效
        if
(!HttpsURLConnection.getDefaultHostnameVerifier().verify(hostname,
sslSocket.session)) {
            throw SSLPeerUnverifiedException("Cannot verify
hostname:$hostname")
        }
        return sslSocket
    }

    companion object {

        private const val SET_HOSTNAME_METHOD_NAME = "setHostname"
```

```
        private val realSSLSocketFactory by lazy {
            newSslSocketFactory(platformTrustManager())
        }

        private var _setHostnameMethod: Method? = null

        private fun setHostnameMethod(sslSocket: SSLSocket): Method? {
            if (null == _setHostnameMethod) {
                // NOTE: 如果无法成功反射setHostname, 则SNI扩展无法成功设置
                // 这里直接消化异常, 由网络库决定如何处理网络访问失败情况
                try {
                    _setHostnameMethod = sslSocket::class.java.getMethod(
                        SET_HOSTNAME_METHOD_NAME, String::class.java)
                } catch (e: NoSuchMethodException) {
                }
            }
            return _setHostnameMethod
        }

        // copy from OkHttp

        private fun platformTrustManager() =
            try {
                val trustManagerFactory =

 TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm())
                trustManagerFactory.init(null as KeyStore?)
                val trustManagers = trustManagerFactory.trustManagers
                if (trustManagers.size != 1 || trustManagers[0] !is
X509TrustManager) {
                    throw IllegalStateException("Unexpected default trust
managers:${Arrays.toString(trustManagers)}")
                }
                trustManagers[0] as X509TrustManager
            } catch (e: GeneralSecurityException) {
                throw RuntimeException("No System TLS", e) // The system
has no TLS. Just give up.
            }

        private fun newSslSocketFactory(trustManager: X509TrustManager) =
            try {
                val sslContext = getSSLContext()
                sslContext.init(null, arrayOf<TrustManager>(trustManager),
null)
                sslContext.socketFactory
            } catch (e: GeneralSecurityException) {
                throw RuntimeException("No System TLS", e) // The system
has no TLS. Just give up.
            }
```

```
        private fun getSSLContext(): SSLContext {
            if (Build.VERSION.SDK_INT in 16..21) {
                try {
                    return SSLContext.getInstance("TLSv1.2")
                } catch (ignored: NoSuchAlgorithmException) {
                    // fallback to TLS
                }
            }

            try {
                return SSLContext.getInstance("TLS")
            } catch (e: NoSuchAlgorithmException) {
                throw IllegalStateException("No TLS provider", e)
            }
        }
    }
}
```

```
// 设置SSLSocketFactory
urlConnection.sslSocketFactory = getSSLSocketFactory(hostname)
```

在示例代码中，我们拷贝了一部分OkHttp的代码实现，用于创建一个真正负责创建SSLSocket的SSLSocketFactory。在createSocket(Socket socket, String host, int port, boolean autoClose)方法中，先通过realSSLSocketFactory创建出一个SSLSocket，再通过setHostname的方式，设置SNI扩展

紧接着，我们还调用了SSLSocket的getSession方法，这里是为了抢先完成TLS握手的工作。之所以这样做，是因为在OkHttp的实现上，也利用URL上的Host信息，进行了SNI扩展的设置，为了避免我们设置的SNI扩展被OkHttp的设置所覆盖，我们必须抢先完成握手工作，一旦握手完成，后续的SNI扩展设置也就失去了意义

OkHttp进行TLS握手的相关代码实现（RealConnection类的connectTls方法，低版本上的实现大体相同）如下：

```
private void connectTls(ConnectionSpecSelector connectionSpecSelector)
throws IOException {
    // ...
    try {
        // Create the wrapper over the connected socket.
        sslSocket = (SSLSocket) sslSocketFactory.createSocket(
            rawSocket, address.url().host(), address.url().port(), true /*
autoClose */);

        // Configure the socket's ciphers, TLS versions, and extensions.
        ConnectionSpec connectionSpec =
connectionSpecSelector.configureSecureSocket(sslSocket);
        if (connectionSpec.supportsTlsExtensions()) {
            // 这里设置了SNI扩展
            // 为了避免SNI扩展设置被OkHttp覆盖，我们需要先于此处完成TLS握手工作
```

```
            Platform.get().configureTlsExtensions(
                sslSocket, address.url().host(), address.protocols());
        }

        // 这里OkHttp开始进行握手，如果之前已经成功握手，则startHandshake方法直接返
回
        // Force handshake. This can throw!
        sslSocket.startHandshake();

        // ...
    } catch (AssertionError e) {
        // ...
    } finally {
        // ...
    }
  }
```

实现HostnameVerifier接口

HostnameVerifer接口，在网络库中通常用于校验服务器证书中的域名信息是否和发起请求时的携带的Host信息一致。由于我们通过替换URL的方式进行HTTPDNS SDK的接入，在进行校验时，应该传入原本的域名信息

示例代码如下：

```
// HostnameVerifier4Tls.kt
internal class HostnameVerifier4Tls(private val hostname: String) :
HostnameVerifier {

    //  NOTE: 这里我们直接复用Default的HostnameVerifier的verify能力，来校验是否服
务器回传的证书和我们的目标域名相匹配
    private val realHostnameVerifier by lazy {
        HttpsURLConnection.getDefaultHostnameVerifier()
    }

    override fun verify(ip: String, session: SSLSession) =
        realHostnameVerifier.verify(hostname, session)
}
```

HttpsURLConnection提供了setHostnameVerifier方法，我们可以调用该方法注入HostnameVerifier的自定义实现

示例代码如下：

```
// 设置HostnameVerifier
urlConnection.hostnameVerifier = getHostnameVerifier(hostname)
```

注意事项

- HTTPS的兼容方式不适用于OkHttp 3.x版本
    - OkHttp 3.x版本上，调用OkHttpClient.Builder的sslSocketFactory方法设置SSLSocketFactory时，会反射获取传入的SSLSocketFactory实现中的context属性，而我们自定义的SSLSocketFactory实现中，是不包含这个属性的，这会导致应用crash
    - 实际上，在OkHttp 3.x版本上，已经将OkUrlFactory标注为deprecated，不建议业务再使用HttpURLConnection 这套API了
    - 业务如果利用OkUrlFactory将HttpURLConnection的底层实现替换成了OkHttp 3.x版本，请参照OkHttp接入HTTPDNS SDK接入方式进行接入
- 调用setHostnameVerifier和setSSLSocketFactory方法时，请勿使用匿名内部类方式实现
    - 匿名内部类的实现，会使得多次请求（同一域名）创建多个HostnameVerifier和SSLSocketFactory实例。
    - OkHttp内部实现了连接复用机制。OkHttp内部在判断当次请求是否可以复用连接池内已有的连接时，会将当前HostnameVerifier及SSLSocketFactory是否和已有连接的HostnameVerifier及SSLSocketFactory是否一致（equals）考虑进去（源码参见ConnectionPool类的get方法）。匿名内部类的实现方式会使得OkHttp的连接复用机制失效
    - 针对同一域名，应使用相同的HostnameVerifier及SSLSocketFactory