Native 与 H5 交互的那些事

JavaScript (/lib/tag/JavaScript) 安卓开发 (/lib/tag/安卓开发) 2016-04-30 22:06:46 发布

Hybrid开发模式目前几乎每家公司都有涉及和使用，这种开发模式兼具良好的Native用户交互体验的优势与WebApp跨平台的优势，而这种模式，在Android中必然需要WebView作为载体来展示H5内容和进行交互，而WebView的各种安全性、兼容性的问题，我想大多数人与它友谊的小床已经翻了，特别是4.2版本之前的addjavascriptInterface接口引起的漏洞，可能导致恶意网页通过Js方法遍历刚刚通过addjavascriptInterface注入进来的类的所有方法从中获取到getClass方法，然后通过反射获取到Runtime对象，进而调用Runtime对象的exec方法执行一些操作，恶意的Js代码如下：

```
 function execute(cmdArgs) {
    for (var obj in window) {
        if ("getClass" in window[obj]) {
            alert(obj);
            return  window[obj].getClass().forName("java.lang.Runtime")
                 .getMethod("getRuntime",null).invoke(null,null).exec(cmdArgs);
        }
    }
}
```

为了避免这个漏洞，即需要限制Js代码能够调用到的Native方法，官方于是在从4.2开始的版本可以通过为可以被Js调用的方法添加 @JavascriptInterface 注解来解决，而之前的版本虽然不能通过这种方法解决，但是可以使用Js的 prompt 方法进行解决，只不过 需要和前端协商好一套公共的协议 ，除此之外，为了避免WebView加载任意url，也需要对url进行白名单检测，由于Android碎片化太严重，WebView也存在兼容性问题，WebView的内核也在4.4版本进行了改变，由webkit改为chromium，此外WebView还有一个非常明显的问题，就是内存泄露，根本原因就是Activity与WebView关联后，WebView内部的一些操作的执行在新线程中，这些时间无法确定，而可能导致WebView一直持有Activity的引用，不能回收。下面就谈谈怎样正确安全的让Native与H5交互

1. Native与H5怎样安全的进行交互？

要使得H5内的Js与Native之间安全的相互进行调用，我们除了可以通过添加 @JavascriptInterface 注解来解决（>=4.2），还有通过 prompt 的方式，不过如果使用官方的方式，这就需要对4.2以下做兼容了，这样使得我们一个app中有两套Js与Native交互的方式，这样极其不好维护，我们应该只需要一套Js与Native交互的方式，所以，我们借助Js中的 prompt 方法来实现 一套安全的Js与Native交互的JsBridge框架

1.1 Js与Native代码相互调用

Native Invoke Js:

我们知道如果Native需要调用Js中的方法，只需要使用 WebView:loadUrl(); 方法即可直接调用指定Js代码，如：

```
mWebView.loadUrl("javascript:setUserName('zhengxiaoyong');");
```

这样就直接调用了Js中的 setUserName 方法并把 zhengxiaoyong 这个名字传到这个方法中去了，接下来就是Js自己处理了

Js Invoke Native:

而如果Js要调用Native中的Java方法呢？这就需要我们自己实现了，因为我们不采取 JavascriptInterface 的方式，而采取prompt方式

对WebView熟悉的同学们应该都知道Js中对应的 window.alert() 、 window.confirm() 、 window.prompt() 这三个方法的调用在 WebChromeClient 中都有对应的回调方法，分别为：

onJsAlert() 、 onJsConfirm() 、 onJsPrompt() ，对于它们传入的 message ，都可以在相应的回调方法中接收到，所以，对于Js调Native方法，我们可以借助这个信道，和前端协定好一段特定规则的 message ，这个规则中应至少包含这些信息：

所调用Native方法所在类的类名

所调用Native的方法名

Js调用Native方法所传入的参数

所以基于这些信息，很容易想到使用http协议的格式来协定规则，如下格式：

scheme://host:port/path?query

对应的我们协定 prompt 传入 message 的格式为:

jsbridge://class:port/method?params

这样以来，前端和app端协商好后，以后前端需要通过Js调用Native方法来获取一些信息或功能，就只需要按照协议的格式把需要调用的类名、方法名、参数放入对应得位置即可，而我们会在 onJsPrompt 方法中接受到，所以我们根据与前端协定好的协议来进行解析，我们可以用一个 Uri 来包装这段协议，然后通过 Uri:getHost、getPath、getQuery 方法获取对应的类名，方法名，参数数据，最后通过反射来调用指定类中指定的方法

而此时会有人问： port 是用来干嘛的？params格式是KV还是什么格式？

当然，既然和前端协定好了协议的格式了，那么params肯定也是需要协定好的，可以用KV格式，也可以用一串Json字符串表示，为了解析方便，还是建议使用 Json格式

而 port 是用来干嘛的呢？

port 我们并不会直接操作它，它是由Js代码自动生成的，port的作用是为了标识Js中的回调 function ,当Js调用Native方法时，我们会得到本次调用的 port 号，我们需要在Native方法执行完毕后再把该 port 、执行的后结果、是否调用成功、调用失败的msg等信息通过调用Js的 onComplete 方法传入，这时候Js凭什么知道你本次返回的信息是哪次调用的结果呢？就是通过 port 号，因为在Js调用Native方法时我们会把自动生成的 port 号和此次回调的 function 绑定在一起，这样以来Native方法返回结果时把 port 也带过来，就知道是哪次回调该用哪个 function 方法来处理

自动生成 port 和绑定 function回调 的Js代码如下：

```
 generatePort: function () {
    return Math.floor(Math.random() * (1 << 50)) + '' + increase++;
},
//调用Native方法
callMethod: function (clazz, method, param, callback) {
    var port = PrivateMethod.generatePort();
    if (typeof callback !== 'function') {
        callback = null;
    }
    //绑定对应port的function回调函数
    PrivateMethod.registerCallback(port, callback);
    PrivateMethod.callNativeMethod(clazz, port, method, param);
},
onComplete: function (port, result) {
    //把Native返回的Json字符串转为JSONObject
    var resultJson = PrivateMethod.str2Json(result);
    //获取对应port的function回调函数
    var callback = PrivateMethod.getCallback(port).callback;
    PrivateMethod.unRegisterCallback(port);
    if (callback) {
        //执行回调
        callback && callback(resultJson);
    }
}
```

Js代码上已经注释的很清楚了，就不多解释了。

经过上面介绍，那么在Native方法执行完成后，当然就需要把结果返回给Js了，那么结果的格式又是什么呢？返回给Js方法又是什么呢？没错，还是需要和前端进行协定，建议数据的返回格式为Json字符串，基本格式为：

```
 resultData = {
    status: {
        code: 0,//0:成功，1:失败
        msg: '请求超时'//失败时候的提示，成功可为空
    },
    data: {}//数据，无数据可以为空
};
```

其中定义了一个 status ，这样的好处是无论在Native方法调用成功与否、Native方法是否有返回值，Js中都可以收到返回的信息，而这个Json字符串至少都会包含一个 status Json对象来描述Native方法调用的状况

阅读目录

而返回给Js的方法自然是上面的 onComplete 方法：

```
javascript:RainbowBridge.onComplete(port,resultData);
```

ps:RainbowBridge是我的JsBridge框架的名字

至此Js调用Native的流程就分析完成了，一切都看起来那么美妙，因为，我们自己实现一套 Js Invoke Native 的主要目的是让Js调用Native更加安全，同时也只维护一套 JsBridge 框架更加方便，那么这个安全性表现在哪里了？

我们知道之前原生的方式漏洞就是恶意Js代码可能会调用Native中的其它方法，那么答案出来了，如果需要让 Js Invoke Native 保证安全性，只需要限制我们通过反射可调用的方法，所以，在JsBridge框架中，我们需要对Js能调用的Native方法给予一定的规则，只有符合这些规则Js才能调用，而我的规则是：

1、Native方法包含public static void 这些修饰符（当然还可能有其它的，如：synchronized）

2、Native方法的参数数量和类型只能有这三个：WebView、JSONObject、JsCallback。为什么要传入这三个参数呢？

2.1、第一个参数是为了提供一个WebView对象，以便获取对应Context和执行WebView的一些方法

2.2、第二个参数就是Js中传入过来的参数，这个肯定要的

2.3、第三个参数就是当Native方法执行完毕后，把执行后的结果回调给Js对应的方法中

所以符合Js调用的Native方法格式为：

```
public static void ***(WebView webView, JSONObject data, JsCallback callback) {
        //get some info ...
        JsCallback.invokeJsCallback(callback, true, result, null);
}
```

判断Js调用的方法是否符合该格式的代码为，符合则存入一个Map中供Js调用：

```
private void putMethod(Class<?> clazz) {
    if (clazz == null)
        return;
    ArrayMap<String, Method> arrayMap = new ArrayMap<>();
    Method method;
    Method[] methods = clazz.getDeclaredMethods();
    int length = methods.length;
    for (int i = 0; i < length; i++) {
        method = methods[i];
        int methodModifiers = method.getModifiers();
        if ((methodModifiers & Modifier.PUBLIC) != 0 && (methodModifiers & Modifier.STATI
C) != 0 && method.getReturnType() == void.class) {
            Class<?>[] parameterTypes = method.getParameterTypes();
            if (parameterTypes != null && parameterTypes.length == 3) {
                if (WebView.class == parameterTypes[0] && JSONObject.class == parameterTy
pes[1] && JsCallback.class == parameterTypes[2]) {
                    arrayMap.put(method.getName(), method);
                }
            }
        }
    }
    mArrayMap.put(clazz.getSimpleName(), arrayMap);
}
```

对于有返回值的方法，并不需要设置它的返回值，因为方法的结果最后我们是通过 JsCallback.invokeJsCallback 来进行对Js层的回调，比如我贴一个符合该格式的Native方法：

```
public static void getOsSdk(WebView webView, JSONObject data, JsCallback callback) {
    JSONObject result = new JSONObject();
    try {
        result.put("os_sdk", Build.VERSION.SDK_INT);
    } catch (JSONException e) {
        e.printStackTrace();
    }
    JsCallback.invokeJsCallback(callback, true, result, null);
}
```

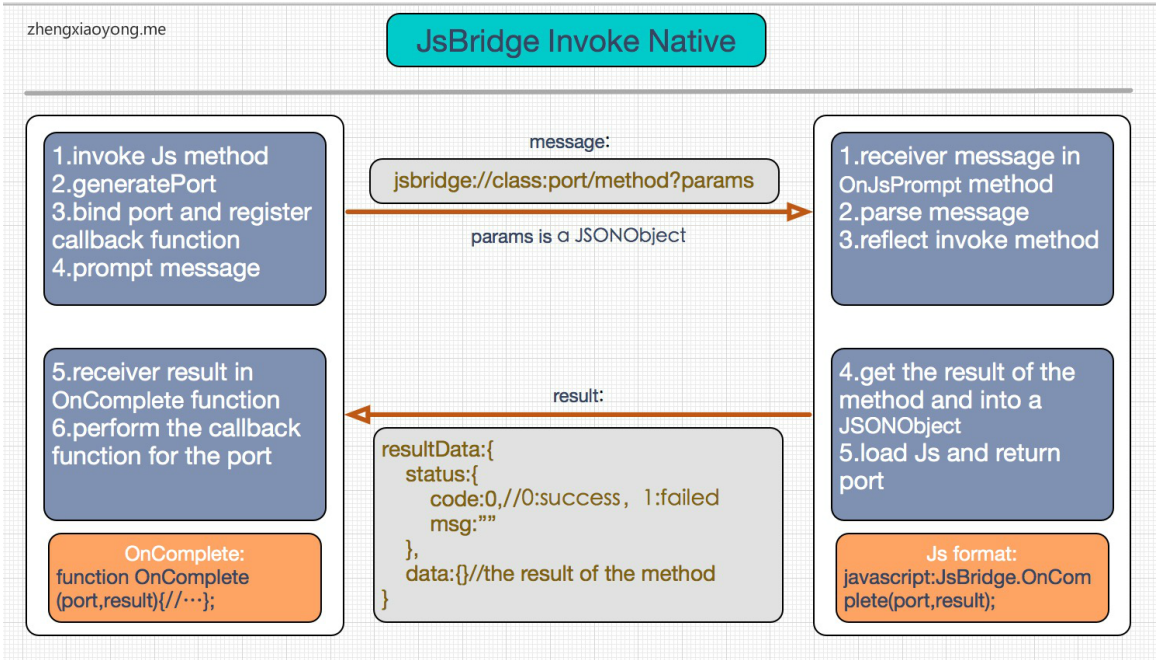
Js调Native代码执行耗时操作情况处理

一般情况下，比如我们通过Js调用Native方法来获取AppName、OsSDK版本、IMSI号、用户信息等都不会有问题，但是，假如该Native方法需要执行一些耗时操作，如：IO、sp、Bitmap Decode、SQLite等，这时为了保护UI的流畅性，我们需要让这些操作执行在异步线程中，待执行完毕再把结果回调给Js，而我们可以提供一个线程池来专门处理这些耗时操作，如：

```
 public static void doAsync(WebView webView, JSONObject data, final JsCallback callback)
{
    AsyncTaskExecutor.runOnAsyncThread(new Runnable() {
        @Override
        public void run() {
            //IO、sp、Bitmap Decode、SQLite
            JsCallback.invokeJsCallback(callback, true, result, null);
        }
    });
}
```

【注】：对于WebView，它的方法的调用只能在主线程中调用，当设计到WebView的方法调用时，切记不可以放在异步线程中调用，否则就GG了.

Js调Native流程图



JsBridge效果图

RainbowBridge: github地址

(https://github.com/Sunzxyong/RainbowBridge)

1.2 白名单Check

上面我们介绍了JsBridge的基本原理，实现了Js与Native相互调用，而且还避免了恶意Js代码调用Native方法的安全问题，通过这样我们保证了Js调用Native方法的安全性，即Js不能随意调用任意Native方法，不过，对于WebView容器来说，它并不关心所加载的url是Js代码还是网页地址，它所做的工作就是执行我们传入的url，而WebView加载url的方式有两种：get和post，方式如下：

```
 mWebView.loadUrl(url);//get
mWebView.postUrl(url,data);//post
```

对于这两种方式，也有不同的应用点，一般get方式用于查，也就是传入的数据不那么重要，比如：商品列表页、商品详情页等，这些传入的数据只是一些商品类的信息。而post方式一般用于改，post传入的数据往往是比较私密的，比如：订单界面、购物车界面等，这些界面只有在把用户的信息post给服务器后，服务器才能正确的返回相应的信息显示在界面上。所以，对于post方式涉及到用户的私密信息，我们总不能给一个url就把私密数据往这个url里面发吧，当然不可能的，这涉及到安全问题，那么就需要一个白名单机制来检查url是否是我们自己的，是我们自己的那么即可以post数据，不是我们自己的那就不post数据，而白名单的定义通常可以以我们自己的域名来判断，搞一个正则表达式，所以我们可以重写WebView的 postUrl 方法：

```
 @Override
public void postUrl(String url, byte[] postData) {
    if (JsBridgeUrlCheckUtil.isTrustUrl(url)) {
        super.postUrl(url, postData);
    } else {
        super.postUrl(url, null);
    }
}
```

这样就对不是我们自己的url进行了拦截，不把数据发送到不是我们自己的服务器中

至此，白名单的Check还没有完成，因为这只是对WebView加载Url时候做的检查，而在WebView内各中链接的跳转、其中有些url还可能被运营商劫持注入了广告，这就有可能在WebView容器内的跳转到某些界面后，该界面的url并不是我们自己的，但是它里面有Js代码调用Native方法来获取一些数据，虽然说Js并不能随便调我们的Native方

法，但是有些我们指定可以被调用的Native方法可能有一些获取设备信息、读取文件、获取用户信息等方法，所以，我们也应该在Js调用Native方法时做一层白名单Check，这样才能保证我们的信息安全

所以，白名单检测需要在两个地方进行检测：

1、WebView:postUrl()前检测url的合法性2、Js调用Native方法前检测当前界面url的合法性

具体代码如下：

```
 @Override
public void postUrl(String url, byte[] postData) {
    if (JsBridgeUrlCheckUtil.isTrustUrl(url)) {
        super.postUrl(url, postData);
    } else {
        super.postUrl(url, null);
    }
}

 /**
 * @param webView WebView
 * @param message rainbow://class:port/method?params
 */
public void call(WebView webView, String message) {
    if (webView == null || TextUtils.isEmpty(message))
        return;
    if (JsBridgeUrlCheckUtil.isTrustUrl(webView.getUrl())) {
        parseMessage(message);
        invokeNativeMethod(webView);
    }
}
```

1.3 移除默认内置接口

WebView内置默认也注入了一些接口，如下：

```
 //移除默认内置接口,防止远程代码执行漏洞攻击
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
    mWebView.removeJavascriptInterface("searchBoxJavaBridge_");
    mWebView.removeJavascriptInterface("accessibility");
    mWebView.removeJavascriptInterface("accessibilityTraversal");
}
```

这些接口虽然不会影响用prompt方式实现的Js与Native交互，但是在使用addJavascriptInterface方式时，有可能有安全问题，最好移除

2. WebView相关

2.1 WebView的配置

下面给出WebView的通用配置：

```
 WebSettings webSettings = mWebView.getSettings();
webSettings.setJavaScriptEnabled(true);
webSettings.setJavaScriptCanOpenWindowsAutomatically(true);
webSettings.setSupportZoom(false);
webSettings.setBuiltInZoomControls(false);
webSettings.setAllowFileAccess(true);
webSettings.setDatabaseEnabled(true);
webSettings.setDomStorageEnabled(true);
webSettings.setGeolocationEnabled(true);
webSettings.setAppCacheEnabled(true);
webSettings.setAppCachePath(getApplicationContext().getCacheDir().getPath());
webSettings.setDefaultTextEncodingName("UTF-8");
//屏幕自适应
webSettings.setUseWideViewPort(true);
webSettings.setLoadWithOverviewMode(true);
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT) {
    webSettings.setCacheMode(WebSettings.LOAD_CACHE_ELSE_NETWORK);
} else {
    webSettings.setCacheMode(WebSettings.LOAD_DEFAULT);
}
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
    webSettings.setDisplayZoomControls(false);
}
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT) {
    webSettings.setLoadsImagesAutomatically(true);
} else {
    webSettings.setLoadsImagesAutomatically(false);
}

mWebView.setScrollBarStyle(WDWebView.SCROLLBARS_INSIDE_OVERLAY);
mWebView.setHorizontalScrollBarEnabled(false);
mWebView.setHorizontalFadingEdgeEnabled(false);
mWebView.setVerticalFadingEdgeEnabled(false);
```

其中有一项配置，是在4.4以上版本时设置网页内图片可以自动加载，而4.4以下版本则不可自动加载，原因是4.4WebView内核的改变，使得WebView的性能更优，所以在4.4以下版本不让图片自动加载，而是先让WebView加载网页的其它静态资源：js、css、文本等等，待网页把这些静态资源加载完成后，在 onPageFinished 方法中再把图片自动加载打开让网页加载图片：

```
 @Override
public void onPageFinished(WebView view, String url) {
    super.onPageFinished(view, url);
    if (!mWebView.getSettings().getLoadsImagesAutomatically()) {
        mWebView.getSettings().setLoadsImagesAutomatically(true);
    }
}
```

2.2 WebView的独立进程

通常来说，WebView的使用会带来诸多问题，内存泄露就是最常见的问题，为了避免WebView内存泄露，目前最流行的有两种做法：

1、独立进程，简单暴力，不过可能涉及到进程间通信2、动态添加WebView，对传入WebView中使用的Context使用弱引用，动态添加WebView意思在布局创建个ViewGroup用来放置WebView，Activity创建时add进来，在Activity停止时remove掉

个人推荐独立进程，好处主要有两点，一是在WebViewActivity使用完毕后直接干掉该进程，防止了内存泄露，二是为我们的app主进程减少了额外的内存占用量

使用独立进程还需注意一点，这个进程中在有多个WebViewActivity，不能在Activity销毁时就干掉进程，不然其它Activity也会蹦了，此时应该在该进程创建一个Activity的维护集合，集合为空时即可干掉进程

关于WebView的销毁，如下：

```
 private void destroyWebView(WebView webView) {
    if (webView == null)
        return;
    webView.stopLoading();
    ViewParent viewParent = webView.getParent();
    if (viewParent != null && viewParent instanceof ViewGroup)
        ((ViewGroup) viewParent).removeView(webView);
    webView.removeAllViews();
    webView.destroy();
    webView = null;
}
```

2.3 WebView的兼容性

2.3.1 不同版本硬件加速的问题

```
 if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.ICE_CREAM_SANDWICH_MR1 && shouldOpenHar
dware()) {
    mWebView.setLayerType(View.LAYER_TYPE_HARDWARE, null);
}
public static boolean shouldOpenHardware () {
    if ("samsung".equalsIgnoreCase(Build.BRAND))
        return false;
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP)
        return true;
    return true;
}
```

2.3.2 不同设备点击WebView输入框键盘的不弹起

```
 mWebView.setOnTouchListener(new View.OnTouchListener() {
    @Override
    public boolean onTouch(View v, MotionEvent event) {
        try {
            if (mWebView != null)
                mWebView.requestFocus();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return false;
    }
});
```

2.3.3 三星手机硬件加速关闭后导致H5弹出的对话框出现不消失情况

```
 String brand = android.os.Build.BRAND;
if ("samsung".equalsIgnoreCase(brand) && Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOL
LIPOP) {
    getWindow().setFlags(
            WindowManager.LayoutParams.FLAG_HARDWARE_ACCELERATED,
            WindowManager.LayoutParams.FLAG_HARDWARE_ACCELERATED);
}
```

2.3.4 不同版本shouldOverrideUrlLoading的回调时机

对于 shouldOverrideUrlLoading 的加载时机，有些同学经常与 onProgressChanged 这个方法的加载时机混淆，这两个方法有两点不同：

1、 shouldOverrideUrlLoading 只会走Get方式的请求，Post方式的请求将不会回调这个方法，而 onProgressChanged 对Get和Post都会走

2、 shouldOverrideUrlLoading 都知道在WebView内部点击链接（Get）会触发，它在Get请求打开界面时也会触发， shouldOverrideUrlLoading 还有一点特殊，就是在按返回键返回到上一个页面时时不会触发的，而 onProgressChanged 在只要界面更新了都会触发

对于 shouldOverrideUrlLoading 的返回值，返回true为剥夺WebView对该此请求的控制权，交给应用自己处理，所以WebView也不会加载该url了，返回false为WebView自己处理

对于 shouldOverrideUrlLoading 的调用时机，也会有不同，在3.0以上是会正常调用的，而在3.0以下，并不是每次都会调用，可以在 onPageStarted 方法中做处理，也没必要了，现在应该都适配4.0以上了

2.3.5 页面重定向导致WebView:goBack()无效的处理

像一些界面有重定向，比如：淘宝等，需要按多次（>1）才能正常返回，一般都是二次，所以可以把那些具有重定向的界面存入一个集合中，在拦截返回事件中这样处理：

```
 @Override
public void onBackPressed() {
    if (mWebView == null)
        return;
    WebBackForwardList backForwardList = mWebView.copyBackForwardList();
    if (backForwardList != null && backForwardList.getSize() != 0) {
        int currentIndex = backForwardList.getCurrentIndex();
        WebHistoryItem historyItem = backForwardList.getItemAtIndex(currentIndex - 1);
        if (historyItem != null) {
            String backPageUrl = historyItem.getUrl();
            if (TextUtils.isEmpty(backPageUrl))
                return;
            int size = REDIRECT_URL.size();
            for (int i = 0; i < size; i++) {
                if (backPageUrl.contains(REDIRECT_URL.get(i)))
                    mWebView.goBack();
            }
        }
    }
    if (mWebView.canGoBack()) {
        mWebView.goBack();
    } else {
        this.finish();
    }
}
```

这里处理是在按返回键时，如果上一个界面是重定向界面，则直接调用goBack，或者也可以finish当前Activity

2.3.6 WebView无法加载不信任网页SSL错误的处理

有时我们的WebView会加载一些不信任的网页，这时候默认的处理是WebView停止加载了，而那些不信任的网页都不是由CA机构信任的，这时候你可以选择继续加载或者让手机内的浏览器来加载：

```
 @Override
public void onReceivedSslError(WebView view, final SslErrorHandler handler, SslError erro
r) {
    //继续加载
    handler.proceed();
    //或者其它处理 ...
}
```

2.3.7 自定义WebView加载出错界面

出错的界面的显示，可以在这个方法中控制：

```
 @Override
public void onReceivedError(WebView view, WebResourceRequest request, WebResourceError er
ror) {
    super.onReceivedError(view, request, error);
}
```

你可以重新加载一段Html专门用来显示错误界面，或者用布局显示一个出错的View，这时候需要把出错的WebView内容清除，可以使用：

```
 @Override
public void onReceivedError(WebView view, WebResourceRequest request, WebResourceError er
ror) {
    super.onReceivedError(view, request, error);
    view.loadDataWithBaseURL(null,"","text/html","UTF-8",null);
    errorView.setVisibility(View.VISIBLE);
}
```

2.3.8 获取位置权限的处理

如果在WebView中有获取地理位置的请求，那么可以直接在代码中默认处理了，没必要弹出一个框框让用户每次都确认：

```
 @Override
public void onGeolocationPermissionsShowPrompt(String origin, GeolocationPermissions.Call
back callback) {
    super.onGeolocationPermissionsShowPrompt(origin, callback);
    callback.invoke(origin, true, false);
}
```

2.4 打造一个通用的WebViewActivity界面

一个通用的WebViewActivity当然是样式和WebView内部处理的策略都统一样，这里只对样式进行说明，因为WebView内部的处理各个公司都不一样，但应该都需要包含这么几点吧：

1、白名单检测

2、Url的跳转

3、出错的处理

4、…

一个WebViewActivity界面，最主要的就是Toolbar标题栏的设计了，因为不同的app的WebViewActivity界面Toolbar上有不同的icon和操作，比如：分享按钮、刷新按钮、更多按钮，都不一样，既然需要通用，即可让调用者传入某个参数来动态改变这些东西吧，比如传一个ToolbarStyle来标识此WebViewActivity的风格是什么样的，背景色、字体颜色、图标等，包括点击时的动画效果，作为通用的界面，必须是让调用者简单操作，不可能调用时传入一个图标id还是一个Drawable，所以，主要需要用到tint，来对字体、图标的颜色动态改变，代码如下：

```
 public static ColorStateList createColorStateList(int normal, int pressed) {
    int[] colors = new int[]{normal, pressed};
    int[][] states = new int[2][];
    states[0] = new int[]{-android.R.attr.state_pressed};
    states[1] = new int[]{android.R.attr.state_pressed};
    return new ColorStateList(states, colors);
}

public static Drawable tintDrawable(Drawable drawable, int color) {
    final Drawable tintDrawable = DrawableCompat.wrap(drawable.mutate());
    ColorStateList colorStateList = ColorStateList.valueOf(color);
    DrawableCompat.setTintMode(tintDrawable, PorterDuff.Mode.SRC_IN);
    DrawableCompat.setTintList(tintDrawable, colorStateList);
    return tintDrawable;
}
```

3. H5与Native界面互相唤起

对于H5界面，有些操作往往是需要唤起Native界面的，比如：H5中的登录按钮，点击后往往唤起Native的登录界面来进行登录，而不是直接在H5登录，这样一个app就只需要一套登录了，而我们所做的便是拦截登录按钮的url：

```
 @Override
public boolean shouldOverrideUrlLoading(WebView view, String url) {
    parserURL(url); //解析url,如果符合跳转native界面的url规则，则跳转native界面
    return super.shouldOverrideUrlLoading(view, url);
}
```

这个规则我们可以在Native的Activity的 intent-filter 中的 data 来定义，如下：

```
 <activity android:name=".LoginActivity">
    <intent-filter>
        <action android:name="android.intent.action.VIEW"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <data
 android:host="native"
 android:path="/login"
 android:scheme="activity"/>
    </intent-filter>
</activity>
```

解析url过程是判断scheme、host、path的是否有完全与之匹配的，有则唤起

而Native唤H5，其实也是一个url的解析过程，只不过需要配置WebViewActivity的 intent-filter 的 data ，WebViewActivity的scheme配置为http和https

startActivity VS UrlRouter

上面说到了H5与Native互相调起，其实这个可以在app内做成一套界面跳转的方式，摒弃startActivity，为什么原生的跳转方式不佳？

1、因为原生的跳转需要确定该Activity是已经存在的，否则编译将报错，这样带来的问题是不利于协同开发，如：A、B同学分别正在开发项目的两个不同的模块，此时B刚好需要跳A同学的某一个界面，如商品列表页跳商品详情页，这时候B就必须写个TODO，待B完成该模块后再写了。而通过url跳转，只需要传入一串url即可

2、原生的跳转Activity与目标Activity是耦合的，跳转Activity完全依赖于目标Activity

3、原生的跳转方式不利于管理所传递来的参数，获取参数时需要在跳转Activity的地方确定传递了几个参数、什么类型的参数，这样以来跳转的方式多了，就比较混乱了。当然一个原生跳转良好的设计是在目的Activity实现一个静态的start方法，其它界面要跳直接调用即可

4、最后一个就是在有参数传递的情况下，每次跳转都要写好多代码啊

而UrlRouter框架的实现原理，一种实现是可以维护一套Activity与url的映射表，这种方式还是没有摆脱不利于协同开发这个毛病，另外一种是通过一串指定规则的url与manifest中配置的data匹配，具体跳转则是通过 intent.setData() 来设置跳转的url，这种方式比较好，不过需要处理下匹配到多个Activity时优先选择的问题

JsBridge地址： RainbowBridge (https://github.com/Sunzxyong/RainbowBridge)

来自： http://zhengxiaoyong.me/2016/04/20/Native与H5交互的那些事/ (http://zhengxiaoyong.me/2016/04/20/Native与H5交互的那些事/)