# 如何设计一个优雅健壮的 Android WebView？

作者：quanguanzhou
原文链接：https://ld246.com/article/1519975549750
来源网站：链滴

# WebView实战操作

WebView在使用过程中会遇到各种各样的问题，下面针对几个在生产环境中使用的WebView可能出的问题进行探讨。

# WebView初始化

也许大部分的开发者针对要打开一个网页这一个Action，会停留在下面这段代码：

```
WebView webview = new WebView(context);
webview.loadUrl(url);
```

这应该是打开一个正常网页最简短的代码了。但大多数情况下，我们需要做一些额外的配置，例如缩支持、Cookie管理、密码存储、DOM存储等，这些配置大部分在WebSettings里，具体配置的内容上文中已有提及，本文不再具体讲解。

接下来，试想如果访问的网页返回的请求是30X，如使用http访问百度的链接（www.baidu.com）那么这时候页面就是空白一片，GG了。为什么呢？因为WebView只加载了第一个网页，接下来的事就不管了。为了解决这个问题，我们需要一个WebViewClient让系统帮我们处理重定向问题。

```
webview.setWebViewClient(new WebViewClient());
```

除了处理重定向，我们还可以覆写WebViewClient中的方法，方法有：

```
public boolean shouldOverrideUrlLoading(WebView view, String url)
public boolean shouldOverrideUrlLoading(WebView view, WebResourceRequest request)
public void onPageStarted(WebView view, String url, Bitmap favicon)
public void onPageFinished(WebView view, String url)
public void onLoadResource(WebView view, String url)
public void onPageCommitVisible(WebView view, String url)
public WebResourceResponse shouldInterceptRequest(WebView view, String url)
public WebResourceResponse shouldInterceptRequest(WebView view, WebResourceRequest
equest)
public void onTooManyRedirects(WebView view, Message cancelMsg, Message continueMsg)

public void onReceivedError(WebView view, int errorCode, String description, String failingUrl)

public void onReceivedError(WebView view, WebResourceRequest request, WebResourceErro
 error)
public void onReceivedHttpError(WebView view, WebResourceRequest request, WebResourc
Response errorResponse)
public void onFormResubmission(WebView view, Message dontResend, Message resend)
public void doUpdateVisitedHistory(WebView view, String url, boolean isReload)
public void onReceivedSslError(WebView view, SslErrorHandler handler, SslError error)
public void onReceivedClientCertRequest(WebView view, ClientCertRequest request)
public void onReceivedHttpAuthRequest(WebView view, HttpAuthHandler handler, String hos
, String realm)
public boolean shouldOverrideKeyEvent(WebView view, KeyEvent event)
public void onUnhandledKeyEvent(WebView view, KeyEvent event)
public void onScaleChanged(WebView view, float oldScale, float newScale)
public void onReceivedLoginRequest(WebView view, String realm, String account, String args)
```

```
public boolean onRenderProcessGone(WebView view, RenderProcessGoneDetail detail)
```

这些方法具体介绍可以参考文章《WebView使用详解（二）——WebViewClient与常用事件监听》
有几个方法是有必要覆写来处理一些客户端逻辑的，后面遇到会详细介绍。

另外，WebView的标题不是一成不变的，加载的网页不一样，标题也不一样。在WebView中，加载网页的标题会回调WebChromeClient.onReceivedTitle()方法，给开发者设置标题。因此，设置一个ebChromeClient也是有必要的。

```
webview.setWebChromeClient(new WebChromeClient());
```

同样，我们还可以覆写WebChromeClient中的方法，方法有：

```
public void onProgressChanged(WebView view, int newProgress)
public void onReceivedTitle(WebView view, String title)
public void onReceivedIcon(WebView view, Bitmap icon)
public void onReceivedTouchIconUrl(WebView view, String url, boolean precomposed)
public void onShowCustomView(View view, int requestedOrientation, CustomViewCallback cal
back)
public void onHideCustomView()
public boolean onCreateWindow(WebView view, boolean isDialog, boolean isUserGesture, M
ssage resultMsg)
public void onRequestFocus(WebView view)
public void onCloseWindow(WebView window)
public boolean onJsAlert(WebView view, String url, String message, JsResult result)
public boolean onJsConfirm(WebView view, String url, String message, JsResult result)
public boolean onJsPrompt(WebView view, String url, String message, String defaultValue, JsP
omptResult result)
public void onExceededDatabaseQuota(String url, String databaseIdentifier, long quota, long
stimatedDatabaseSize, long totalQuota, WebStorage.QuotaUpdater quotaUpdater)
public void onReachedMaxAppCacheSize(long requiredStorage, long quota, WebStorage.Quo
aUpdater quotaUpdater)
public void onGeolocationPermissionsShowPrompt(String origin, GeolocationPermissions.Cal
back callback)
public void onGeolocationPermissionsHidePrompt()
public void onPermissionRequest(PermissionRequest request)
public void onPermissionRequestCanceled(PermissionRequest request)
public boolean onJsTimeout()
public void onConsoleMessage(String message, int lineNumber, String sourceID)
public boolean onConsoleMessage(ConsoleMessage consoleMessage)
public Bitmap getDefaultVideoPoster()
public void getVisitedHistory(ValueCallback<String[]> callback)
public boolean onShowFileChooser(WebView webView, ValueCallback<Uri[]> filePathCallback
 FileChooserParams fileChooserParams)
public void openFileChooser(ValueCallback<Uri> uploadFile, String acceptType, String captur
)
public void setupAutoFill(Message msg)
```

这些方法具体介绍可以参考文章《WebView使用详解（三）——WebChromeClient与LoadData补》。除了接收标题以外，进度条的改变，WebView请求本地文件、请求地理位置权限等，都是通过WbChromeClient的回调实现的。

在初始化阶段，如果启用了Javascript，那么需要移除相关的安全漏洞，这在上一篇文章中也有所提。最后，在考拉KaolaWebView.init()方法中，执行了如下操作：

```
protected void init() {
    mContext = getContext();
    mWebJsManager = new WebJsManager();    // 初始化Js管理器
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT) {
        // 根据本地调试开关打开Chrome调试
       WebView.setWebContentsDebuggingEnabled(WebSwitchManager.isDebugEnable());
    }
    // WebSettings配置
    WebViewSettings.setDefaultWebSettings(this);
    // 获取deviceId列表，安全相关
    WebViewHelper.requestNeedDeviceIdUrlList(null);
    // 设置下载的监听器
    setDownloadListener(this);
    // 前端控制回退栈，默认回退1。
    mBackStep = 1;
    // 重定向保护，防止空白页
    mRedirectProtected = true;
    // 截图使用
    setDrawingCacheEnabled(true);
    // 初始化具体的Jsbridge类
    enableJsApiInternal();
    // 初始化WebCache，用于加载静态资源
    initWebCache();
    // 初始化WebChromeClient，覆写其中一部分方法
    super.setWebChromeClient(mChromeClient);
    // 初始化WebViewClient，覆写其中一部分方法
    super.setWebViewClient(mWebViewClient);
}
```

# WebView加载一个网页的过程中该做些什么？

如果说加载一个网页只需要调用WebView.loadUrl(url)这么简单，那肯定没程序员啥事儿了。往往事没有这么简单。加载网页是一个复杂的过程，在这个过程中，我们可能需要执行一些操作，包括：

1. 加载网页前，重置WebView状态以及与业务绑定的变量状态。WebView状态包括重定向状态(mouchByUser)、前端控制的回退栈(mBackStep)等，业务状态包括进度条、当前页的分享内容、分享钮的显示隐藏等。

2. 加载网页前，根据不同的域拼接本地客户端的参数，包括基本的机型信息、版本信息、登录信息以埋点使用的Refer信息等，有时候涉及交易、财产等还需要做额外的配置。

3. 开始执行页面加载操作时，会回调 WebViewClient.onPageStarted(webview, url, favicon)。此方法中，可以重置重定向保护的变量(mRedirectProtected)，当然也可以在页面加载前重置，由于史遗留代码问题，此处尚未省去优化。

4. 加载页面的过程中，WebView会回调几个方法。

- WebChromeClient.onReceivedTitle(webview, title)，用来设置标题。需要注意的是，在部分ndroid系统版本中可能会回调多次这个方法，而且有时候回调的title是一个url，客户端可以针对这种况进行特殊处理，避免在标题栏显示不必要的链接。
- WebChromeClient.onProgressChanged(webview, progress)，根据这个回调，可以控制进

条的进度（包括显示与隐藏）。一般情况下，想要达到100%的进度需要的时间较长（特别是首次加），用户长时间等待进度条不消失必定会感到焦虑，影响体验。其实当progress达到80的时候，加载来的页面已经基本可用了。因此，可以投机取巧，达到80%以后便可以认为进度条到100%了，事实，国内厂商大部分都会提前隐藏进度条，让用户以为网页加载很快。

- WebViewClient.shouldInterceptRequest(webview, request)，无论是普通的页面请求(使用GT/POST)，还是页面中的异步请求，或者页面中的资源请求，都会回调这个方法，给开发一次拦截请的机会。在这个方法中，我们可以进行静态资源的拦截并使用缓存数据代替，也可以拦截页面，使用己的网络框架来请求数据。包括后面介绍的WebView免流方案，也和此方法有关。

- WebViewClient.shouldOverrideUrlLoading(webview, request)，如果遇到了重定向，或者击了页面中的a标签实现页面跳转，那么会回调这个方法。可以说这个是WebView里面最重要的回调一，后面WebView与Native页面交互一节将会详细介绍这个方法。

- WebViewClient.onReceived**Error(webview, handler, error)，加载页面的过程中发生了错，会回调这个方法。主要是http错误以及ssl错误。在这两个回调中，我们可以进行异常上报，监控异页面、过期页面，及时反馈给运营或前端修改。在处理ssl错误时，遇到不信任的证书可以进行特殊处，例如对域名进行判断，针对自己公司的域名“放行”，防止进入丑陋的错误证书页面。也可以与Chrme一样，弹出ssl证书疑问弹窗，给用户选择的余地。

5. 页面加载结束后，会回调 WebViewClient.onPageFinished(webview, url)。这时候可以根据回栈的情况判断是否显示关闭WebView按钮。通过mActivityWeb.canGoBackOrForward(-1)判断是可以回退。

# WebView与JavaScript交互——JsBridge

Android WebView与JavaScript的通信方案，目前业界已经有比较成熟的方案了。常见的有lzyzsd/Jsridge、pengwei1024/JsBridge等，详见此链接。

通常，Java调用js方法有两种：

- WebView.loadUrl("javascript:" + javascript);
- WebView.evaluateJavascript(javascript, callbacck);

第一种方式已经不推荐使用了，第二种方式不仅更方便，也提供了结果的回调，但仅支持API 19以后系统。

js调用Java的方法有四种，分别是：

- JavascriptInterface
- WebViewClient.shouldOverrideUrlLoading()
- WebChromeClient.onConsoleMessage()
- WebChromeClient.onJsPrompt()

这四种方式不再一一介绍，掘金上的这篇文章已经讲得很详细。

下面来介绍一下考拉使用的JsBridge方案。Java调用js方法不必多说，根据Android系统版本不同分调用第一个方法和第二个方法。在js调用Java方法上，考拉使用的是第四种方案，即侵入WebChromClient.onJsPrompt(webview, url, message, defaultValue, result)实现通信。

```
@Override
public boolean onJsPrompt(WebView view, String url, String message, String defaultValue,
        JsPromptResult result) {
```

```
    if (!ActivityUtils.activityIsAlive(mContext)) {//页面关闭后，直接返回
        try {
            result.cancel();
        } catch (Exception ignored) {
        }
        return true;
    }
    if (mJsApi != null && mJsApi.hijackJsPrompt(message)) {
        result.confirm();
        return true;
    }
    return super.onJsPrompt(view, url, message, defaultValue, result);
}
```

由于onJsPrompt方法不确定是在什么时候回调，官方文档也没有说明这个方法是在主线程调用还是步线程，因此判断一下Activity的生命周期是有必要的。js与Java的方法调用主要在mJsApi.hijackJsPrmpt(message)中。

```
public boolean hijackJsPrompt(String message) {
    if (TextUtils.isEmpty(message)) {
        return false;
    }

    boolean handle = message.startsWith(YIXIN_JSBRIDGE);

    if (handle) {
        call(message);
    }

    return handle;
}
```

首先判断该信息是否应该拦截，如果允许拦截的话，则取出js传过来的方法和参数，通过Handler把息抛给业务层处理。

```
private void call(String message) {
    // PREFIX
    message = message.substring(KaolaJsApi.YIXIN_JSBRIDGE.length());
    // BASE64
    message = new String(Base64.decode(message));

    JSONObject json = JSONObject.parseObject(message);
    String method = json.getString("method");
    String params = json.getString("params");
    String version = json.getString("jsonrpc");

    if ("2.0".equals(version)) {
        int id = json.containsKey("id") ? json.getIntValue("id") : -1;

        call(id, method, params);
    }
```

```
    callJS("window.jsonRPC.invokeFinish()");
}

private void call(int id, String method, String params) {
    Message msg = Message.obtain();
    msg.what = MsgWhat.JSCall;
    msg.obj = new KaolaJSMessage(id, method, params);
    // 通过handler把消息发出去，待接收方处理。
    if (handler != null) {
        handler.sendMessage(msg);
    }
}
```

jsbridge中，实现了一个存储jsbridge指令的队列CommandQueue，每次需要调用jsbridge时，只要入队即可。

```
function CommandQueue() {
    this.backQueue = [];
    this.queue = [];
};

CommandQueue.prototype.dequeue = function() {
    if(this.queue.length <=0 && this.backQueue.length > 0) {
        this.queue = this.backQueue.reverse();
        this.backQueue = [];
    }
    return this.queue.pop();
};

CommandQueue.prototype.enqueue = function(item) {
    this.backQueue.push(item);
};

Object.defineProperty(CommandQueue.prototype, 'length',
{get: function() {return this.queue.length + this.backQueue.length; }});

var commandQueue = new CommandQueue();
function filterObj(obj){
    for(var i in obj){
        if (obj.hasOwnProperty(i))
        {
            if(typeof obj[i] == 'string'){
                obj[i] = obj[i].replace(/[\uD800-\uDBFF][\uDC00-\uDFFF]/g, '');
            }
        }
    }
    return obj;
}

function _nativeExec(){
    var command = commandQueue.dequeue();
    if(command) {
        nativeReady = false;
```

```
        var jsoncommand = JSON.stringify(command);
        // 做了base64转换。
        var _temp = prompt(YIXIN_JSBRIDGE + base64encode(UTF8.encode(jsoncommand)),'');
        return true;
    } else {
        return false;
    }
}
```

前端真正需要调用Java方法时，执行window.WeiXinJSBridge.call方法。

```
function doCall(request, success_cb, error_cb) {
        if (jsonRPCIdTag in request && typeof success_cb !== 'undefined') {
            _callbacks[request.id] = { success_cb: success_cb, error_cb: error_cb };
        }
        commandQueue.enqueue(request);
        if(nativeReady) {
            _nativeExec();
        }
    }

    jsonRPC.call = function(method, params, success_cb, error_cb) {

        var request = {
            jsonrpc : jsonRPCVer,
            method  : method,
            params  : params,
            id      : _current_id++
        };
        doCall(request, success_cb, error_cb);
    };

    jsonRPC.notify = function(method, params) {
        var request = {
            jsonrpc : jsonRPCVer,
            method  : method,
            params  : params,
        };
        doCall(request, null, null);
    };

    jsonRPC.ready = function() {
        jsonRPC.nativeEvent.on('NativeReady', function(e) {
            nativeReady = false;
            if(!_nativeExec()) {
                nativeReady = true;
            }
        });
        jsonRPC.nativeEvent.Trigger('WeixinJSBridgeReady');
    };

    jsonRPC.invokeFinish = function() {
        nativeReady = true;
```

```
        _nativeExec();
    };

    jsonRPC.nativeEvent = {};

    jsonRPC.nativeEvent.Trigger = function(type, detail) {
        var ev = YixinEvent(type,detail);
        document.dispatchEvent(ev);
    };

    var nativeEvent = {};

    var doc = document;

    window.WeixinJSBridge = {};
    window.jsonRPC = jsonRPC;
    window.WeixinJSBridge.call = jsonRPC.notify;

})();
```

注意，上面的代码有所删减，若需要执行完整的jsbridge功能，还需要做一些额外的配置。例如告知端这段js代码已经注入成功的标记。

# 什么时候注入js合适?

如果做过WebView开发，并且需要和js交互的同学，大部分都会认为js在WebViewClient.onPageFinished()方法中注入最合适，此时dom树已经构建完成，页面已经完全展现出来[1][3]。但如果做过面加载速度的测试，会发现WebViewClient.onPageFinished()方法通常需要等待很久才会回调（首加载通常超过3s），这是因为WebView需要加载完一个网页里主文档和所有的资源才会回调这个方。能不能在WebViewClient.onPageStarted()中注入呢？答案是不确定。经过测试，有些机型可以，些机型不行。在WebViewClient.onPageStarted()中注入还有一个致命的问题——这个方法可能会调多次，会造成js代码的多次注入。

另一方面，从7.0开始，WebView加载js方式发生了一些小改变，官方建议把js注入的时机放在页面始加载之后。援引官方的文档[4]：

> Javascript run before page load
>
> Starting with apps targeting Android 7.0, the Javascript context will be reset when a new page is loaded. Currently, the context is carried over for the first page loaded in a new WebView ins ance.
>
> Developers looking to inject Javascript into the WebView should execute the script after the age has started to load.

在这篇文章中也提及了js注入的时机可以在多个回调里实现，包括：

- onLoadResource
- doUpdateVisitedHistory
- onPageStarted
- onPageFinished

- onReceivedTitle
- onProgressChanged

尽管文章作者已经做了测试证明以上时机注入是可行的，但他不能完全保证没有问题。事实也是，这回调里有多个是会回调多次的，不能保证一次注入成功。

WebViewClient.onPageStarted()太早，WebViewClient.onPageFinished()又太迟，究竟有没有比合适的注入时机呢？试试WebViewClient.onProgressChanged()？这个方法在dom树渲染的过程中回调多次，每次都会告诉我们当前加载的进度。这不正是告诉我们页面已经开始加载了吗？考拉正是用了WebViewClient.onProgressChanged()方法来注入js代码。

```
@Override
public void onProgressChanged(WebView view, int newProgress) {
    super.onProgressChanged(view, newProgress);
    if (null != mIWebViewClient) {
        mIWebViewClient.onProgressChanged(view, newProgress);
    }

    if (mCallProgressCallback && newProgress >= mProgressFinishThreshold) {
        DebugLog.d("WebView", "onProgressChanged: " + newProgress);
        mCallProgressCallback = false;
        // mJsApi不为null且允许注入js的情况下，开始注入js代码。
        if (mJsApi != null && WebJsManager.enableJs(view.getUrl())) {
            mJsApi.loadLocalJsCode();
        }
        if (mIWebViewClient != null) {
            mIWebViewClient.onPageFinished(view, newProgress);
        }
    }
}
```

可以看到，我们使用了mProgressFinishThreshold这个变量控制注入时机，这与前面提及的当progres达到80的时候，加载出来的页面已经基本可用了是相呼应的。

达到80%很容易，达到100%却很难。

正是因为这个原因，页面的进度加载到80%的时候，实际上dom树已经渲染得差不多了，表明WebVi w已经解析了标签，这时候注入一定是成功的。在WebViewClient.onProgressChanged()实现js注有几个需要注意的地方：

1. 上文提到的多次注入控制，我们使用了mCallProgressCallback变量控制
2. 重新加载一个URL之前，需要重置mCallProgressCallback，让重新加载后的页面再次注入js
3. 注入的进度阈值可以自由定制，理论上10%-100%都是合理的，我们使用了80%。

# H5页面、Weex页面与Native页面交互——KaolaRouter

H5页面、Weex页面与Native页面的交互是通过URL拦截实现的。在WebView中，WebViewClient.s ouldOverrideUrlLoading()方法能够获取到当前加载的URL，然后把URL传递给考拉路由框架，便可判断URL是否能够跳转到其他非H5页面，考拉路由框架在《考拉Android客户端路由总线设计》一文有详细介绍，但当时未引入Weex页面，关于如何整合三者的通信，后续文章会有详细介绍。

在WebViewClient.shouldOverrideUrlLoading()中，根据URL类型做了判断：

```
public boolean shouldOverrideUrlLoading(WebView view, String url) {
    if (StringUtils.isNotBlank(url) && url.equals("about:blank")) {   //js调用reload刷新页面时候，
别机型跳到空页面问题修复
        url = getUrl();
    }
    url = WebViewUtils.removeBlank(url);
    mCallProgressCallback = true;
    //允许启动第三方应用客户端
    if (WebViewUtils.canHandleUrl(url)) {
        boolean handleByCaller = false;
        // 如果不是用户触发的操作，就没有必要交给上层处理了，直接走url拦截规则。
        if (null != mIWebViewClient && isTouchByUser()) {
            // 先交给业务层拦截处理
            handleByCaller = mIWebViewClient.shouldOverrideUrlLoading(view, url);
        }
        if (!handleByCaller) {
            // 业务层不拦截，走通用路由总线规则
            handleByCaller = handleOverrideUrl(url);
        }
        mRedirectProtected = true;
        return handleByCaller || super.shouldOverrideUrlLoading(view, url);
    } else {
        try {
            notifyBeforeLoadUrl(url);
            // https://sumile.cn/archives/1223.html
            Intent intent = Intent.parseUri(url, Intent.URI_INTENT_SCHEME);
            intent.addCategory(Intent.CATEGORY_BROWSABLE);
            intent.setComponent(null);
            intent.setSelector(null);
            mContext.startActivity(intent);
            if (!mIsBlankPageRedirect) {
                back();
            }
        } catch (Exception e) {
            ExceptionUtils.printExceptionTrace(e);
        }
        return true;
    }
}

private boolean handleOverrideUrl(final String url) {
   RouterResult result =  WebActivityRouter.startFromWeb(
            new IntentBuilder(mContext, url).setRouterActivityResult(new RouterActivityResult() {
                @Override
                public void onActivityFound() {
                    if (!mIsBlankPageRedirect) {
                        // 路由拦截成功以后，为防止首次进入WebView产生白屏，因此加了保护机制
                        back();
                    }
                }

                @Override
```

原文链接：如何设计一个优雅健壮的 Android WebView?

```
            public void onActivityNotFound() {

            }
        }));
    return result.isSuccess();
}
```

代码里写了注释，就不一一解释了。

# WebView下拉刷新实现

由于考拉使用的下拉刷新跟Material Design所使用的下拉刷新样式不一致，因此不能直接套用SwipefreshLayout。考拉使用的是一套改造过的Android-PullToRefresh，WebView的下拉刷新，正是承自PullToRefreshBase来实现的。

```
/**
 * 创建者：Square Xu
 * 日期：2017/2/23
 * 功能模块：webview下拉刷新组件
 */
public class PullToRefreshWebView extends PullToRefreshBase<KaolaWebview> {
    public PullToRefreshWebView(Context context) {
        super(context);
    }

    public PullToRefreshWebView(Context context, AttributeSet attrs) {
        super(context, attrs);
    }

    public PullToRefreshWebView(Context context, AttributeSet attrs, int defStyleAttr) {
        this(context, attrs);
    }

    public PullToRefreshWebView(Context context, Mode mode) {
        super(context, mode);
    }

    public PullToRefreshWebView(Context context, Mode mode, AnimationStyle animStyle) {
        super(context, mode, animStyle);
    }

    @Override
    public Orientation getPullToRefreshScrollDirection() {
        return Orientation.VERTICAL;
    }

    @Override
    protected KaolaWebview createRefreshableView(Context context, AttributeSet attrs) {
        KaolaWebview kaolaWebview = new KaolaWebview(context, attrs);
        //解决键盘弹起时候闪动的问题
        setGravity(AXIS_PULL_BEFORE);
        return kaolaWebview;
    }
```

```
    @Override
    protected boolean isReadyForPullEnd() {
        return false;
    }

    @Override
    protected boolean isReadyForPullStart() {
        return getRefreshableView().getScrollY() == 0;
    }
}
```

考拉使用了全屏模式实现沉浸式状态栏及滑动返回，全屏模式和WebView下拉刷新相结合对键盘的起产生了闪动效果，经过组内大神的研究与多次调试（感谢@俊俊），发现setGravity(AXIS_PULL_BEORE)能够解决闪动的问题。

# 如何处理加载错误(Http、SSL、Resource)?

对于WebView加载一个网页过程中所产生的错误回调，大致有三种：

● WebViewClient.onReceivedHttpError(webView, webResourceRequest, webResourceRespone)

任何HTTP请求产生的错误都会回调这个方法，包括主页面的html文档请求，iframe、图片等资源请。在这个回调中，由于混杂了很多请求，不适合用来展示加载错误的页面，而适合做监控报警。当某URL，或者某个资源收到大量报警时，说明页面或资源可能存在问题，这时候可以让相关运营及时响修改。

● WebViewClient.onReceivedSslError(webview, sslErrorHandler, sslError)

任何HTTPS请求，遇到SSL错误时都会回调这个方法。比较正确的做法是让用户选择是否信任这个网，这时候可以弹出信任选择框供用户选择（大部分正规浏览器是这么做的）。但人都是有私心的，何是遇到自家的网站时。我们可以让一些特定的网站，不管其证书是否存在问题，都让用户信任它。在一点上，分享一个小坑。考拉的SSL证书使用的是GeoTrust的GeoTrust SSL CA - G3，但是在某些机上，打开考拉的页面都会提示证书错误。这时候就不得不使用“绝招”——让考拉的所有二级域都是信任的。

```
@Override
public void onReceivedSslError(WebView view, SslErrorHandler handler, SslError error) {
    if (UrlUtils.isKaolaHost(getUrl())) {
        handler.proceed();
    } else {
        super.onReceivedSslError(view, handler, error);
    }
}
```

● WebViewClient.onReceivedError(webView, webResourceRequest, webResourceError)

只有在主页面加载出现错误时，才会回调这个方法。这正是展示加载错误页面最合适的方法。然鹅，果不管三七二十一直接展示错误页面的话，那很有可能会误判，给用户造成经常加载页面失败的错觉由于不同的WebView实现可能不一样，所以我们首先需要排除几种误判的例子：

1. 加载失败的url跟WebView里的url不是同一个url，排除；
2. errorCode=-1，表明是ERROR_UNKNOWN的错误，为了保证不误判，排除
3. failingUrl=null&errorCode=-12，由于错误的url是空而不是ERROR_BAD_URL，排除

```
@Override
public void onReceivedError(WebView view, int errorCode, String description, String failingUrl)
{
    super.onReceivedError(view, errorCode, description, failingUrl);

    // -12 == EventHandle.ERROR_BAD_URL, a hide return code inside android.net.http packag

    if ((failingUrl != null && !failingUrl.equals(view.getUrl()) && !failingUrl.equals(view.getOrigi
alUrl())) /* not subresource error*/
            || (failingUrl == null && errorCode != -12) /*not bad url*/
            || errorCode == -1) { //当 errorCode = -1 且错误信息为 net::ERR_CACHE_MISS
        return;
    }

    if (!TextUtils.isEmpty(failingUrl)) {
        if (failingUrl.equals(view.getUrl())) {
            if (null != mIWebViewClient) {
                mIWebViewClient.onReceivedError(view);
            }
        }
    }
}
```

# 如何操作cookie?

Cookie默认情况下是不需要做处理的，如果有特殊需求，如针对某个页面设置额外的Cookie字段，以通过代码来控制。下面列出几个有用的接口：

- 获取某个url下的所有Cookie： CookieManager.getInstance().getCookie(url)
- 判断WebView是否接受Cookie： CookieManager.getInstance().acceptCookie()
- 清除Session Cookie： CookieManager.getInstance().removeSessionCookies(ValueCallback allback)
- 清除所有Cookie： CookieManager.getInstance().removeAllCookies(ValueCallback callback)
- Cookie持久化： CookieManager.getInstance().flush()
- 针对某个主机设置Cookie： CookieManager.getInstance().setCookie(String url, String value)

下面是一个给考拉M站设置Cookie的例子：

```
public static void setBoundCookies() {
    CookieSyncManager.createInstance(HTApplication.getInstance());
    long expiredTime = System.currentTimeMillis() + 10 * 60 * 1000;
    CookieManager cookieManager = CookieManager.getInstance();
    cookieManager.setAcceptCookie(true);
    cookieManager.setCookie(NetConfig.KAOLA_M_HOST,
            String.format("Expires=%s; domain=.kaola.com; path=/", expiredTime));
```

```
    cookieManager.setCookie(NetConfig.KAOLA_M_HOST, "KAOLA_CLEAR_RELATION=1; doma
n=.kaola.com; path=/");
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
        CookieManager.getInstance().flush();
    } else {
        CookieSyncManager.getInstance().sync();
    }
}
```

# 如何调试WebView加载的页面?

在Android 4.4版本以后，可以使用Chrome开发者工具调试WebView内容[^5]。调试需要在代码里设打开调试开关。

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT) {
    WebView.setWebContentsDebuggingEnabled(true);
}
```

开启后，使用USB连接电脑，加载URL时，打开Chrome开发者工具，在浏览器输入

```
chrome://inspect
```

可以看到当前正在浏览的页面，点击inspect即可看到WebView加载的内容。

# WebView优化

除了上面提到的基本操作用来实现一个完整的浏览器功能外，WebView的加载速度、稳定性和安全是可以进一步加强和提高的。下面从几个方面介绍一下WebView的优化方案，这些方案可能并不是适用于所有场景，但思路是可以借鉴的。

# CandyWebCache

我们知道，在加载页面的过程中，js、css和图片资源占用了大量的流量，如果这些资源一开始就放在地，或者只需要下载一次，后面重复利用，岂不美哉。尽管WebView也有几套缓存方案[^6]，但是总而言效果不理想。基于自建缓存系统的思路，由网易杭研研发的CandyWebCache项目应运而生。CadyWebCache是一套支持离线缓存WebView资源并实时更新远程资源的解决方案，支持打母包时下当前最新的资源文件集成到apk中，也支持在线实时更新资源。在WebView中，我们需要拦截WebViewClient.shouldInterceptRequest()方法，检测缓存是否存在，存在则直接取本地缓存数据，减少网请求产生的流量。

```
@Override
public WebResourceResponse shouldInterceptRequest(WebView view, WebResourceRequest
equest) {
    if (WebSwitchManager.isWebCacheEnabled()) {
        try {
            WebResourceResponse resourceResponse = CandyWebCache.getsInstance().getResp
nse(view, request);
            return WebViewUtils.handleResponseHeader(resourceResponse);
        } catch (Throwable e) {
            ExceptionUtils.uploadCatchedException(e);
```

```
        }
    }
    return super.shouldInterceptRequest(view, request);
}

@Override
public WebResourceResponse shouldInterceptRequest(WebView view, String url) {
    if (WebSwitchManager.isWebCacheEnabled()) {
        try {
            WebResourceResponse resourceResponse = CandyWebCache.getsInstance().getResp
nse(view, url);
            return WebViewUtils.handleResponseHeader(resourceResponse);
        } catch (Throwable e) {
            ExceptionUtils.uploadCatchedException(e);
        }
    }
    return super.shouldInterceptRequest(view, url);
}
```

除了上述缓存方案外，腾讯的QQ会员团队也推出了开源的解决方案VasSonic，旨在提升H5的页面访体验，但最好由前后端一起配合改造。这套整体的解决方案有很多借鉴意义，考拉也在学习中。

# Https、HttpDns、CDN

将http请求切换为https请求，可以降低运营商网络劫持（js劫持、图片劫持等）的概率，特别是使用http2后，能够大幅提升web性能，减少网络延迟，减少请求的流量。

HttpDns，使用http协议向特定的DNS服务器进行域名解析请求，代替基于DNS协议向运营商的Local DNS发起解析请求，可以降低运营商DNS劫持带来的访问失败。目前在WebView上使用HttpDns尚在一定问题，网上也没有较好的解决方案（阿里云Android WebView+HttpDns最佳实践、腾讯云HtpDns SDK接入、webview接入HttpDNS实践），因此还在调研中。

另一方面，可以把静态资源部署到多路CDN，直接通过CDN地址访问，减少网络延迟，多路CDN保单个CDN大面积节点访问失败时可切换到备用的CDN上。

# WebView独立进程

WebView实例在Android7.0系统以后，已经可以选择运行在一个独立进程上^7；8.0以后默认就是行在独立的沙盒进程中^8，未来Google也在朝这个方向发展，具体的WebView历史可以参考上一篇章《如何设计一个优雅健壮的Android WebView？（上）》第一小节。

Android7.0系统以后，WebView相对来说是比较稳定的，无论承载WebView的容器是否在主进程，不需要担心WebView崩溃导致应用也跟着崩溃。然后7.0以下的系统就没有这么幸运了，特别是低版的WebView。考虑应用的稳定性，我们可以把7.0以下系统的WebView使用一个独立进程的Activity包装，这样即使WebView崩溃了，也只是WebView所在的进程发生了崩溃，主进程还是不受影响的。

```
public static Intent getWebViewIntent(Context context) {
    Intent intent;
    if (isWebInMainProcess()) {
        intent = new Intent(context, MainWebviewActivity.class);
    } else {
        intent = new Intent(context, WebviewActivity.class);
```

```
    }
    return intent;
}

public static boolean isWebInMainProcess() {
    return android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.N;
}
```

# WebView免流实现（待实现）

1. 全局代理
2. WebViewClient.shouldInterceptRequest()，IP替换

# WebView现状

Android系统的WebView发展历史可谓一波三折，系统WebView开发者肯定费劲心思才换取了今天局面——应用里的WebView和Chrome表现一致。对于Android初学者，或者刚要开始接触WebVie的开发来说，WebView是有点难以适应，甚至是有一些惧怕的。开源社区对于WebView的改造和包非常少，需要开发者查找大量资料去理解WebView。

# WebView Changelog

在Android4.4（API level 19）系统以前，Android使用了原生自带的Android Webkit内核，这个内对HTML5的支持不是很好，现在使用4.4以下机子的也不多了，就不对这个内核做过多介绍了，有兴可以看下这篇文章。

从Android4.4系统开始，Chromium内核取代了Webkit内核，正式地接管了WebView的渲染工作。hromium是一个开源的浏览器内核项目，基于Chromium开源项目修改实现的浏览器非常多，包括著名的Chrome浏览器，以及一众国内浏览器（360浏览器、QQ浏览器等）。其中Chromium在Andrid上面的实现是Android System WebView^1。

从Android5.0系统开始，WebView移植成了一个独立的apk，可以不依赖系统而独立存在和更新，们可以在系统->设置->Android System WebView看到WebView的当前版本。

从Android7.0系统开始，如果系统安装了Chrome (version>51)，那么Chrome将会直接为应用的WbView提供渲染，WebView版本会随着Chrome的更新而更新，用户也可以选择WebView的服务提方（在开发者选项->WebView Implementation里），WebView可以脱离应用，在一个独立的沙盒程中渲染页面（需要在开发者选项里打开）^2。

从Android8.0系统开始，默认开启WebView多进程模式，即WebView运行在独立的沙盒进程中^3。

# 为什么WebView那么难搞?

尽管应用开发者使用WebView和使用普通的View一样简单，只需要在xml里定义或者直接实例化出即可使用，但WebView是相当难搞的。为什么呢？以下有几个可能的因素。

- 繁杂的WebView配置

WebView在初始化的时候就提供了默认配置WebSettings，但是很多默认配置是不能够满足业务需的，还需要进行二次配置，例如考拉App在默认配置基础做了如下修改：

```
public static void setDefaultWebSettings(WebView webView) {
    WebSettings webSettings = webView.getSettings();
    //5.0以上开启混合模式加载
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
        webSettings.setMixedContentMode(WebSettings.MIXED_CONTENT_ALWAYS_ALLOW);
    }
    webSettings.setLoadWithOverviewMode(true);
    webSettings.setUseWideViewPort(true);
    //允许js代码
    webSettings.setJavaScriptEnabled(true);
    //允许SessionStorage/LocalStorage存储
    webSettings.setDomStorageEnabled(true);
    //禁用放缩
    webSettings.setDisplayZoomControls(false);
    webSettings.setBuiltInZoomControls(false);
    //禁用文字缩放
    webSettings.setTextZoom(100);
    //10M缓存，api 18后，系统自动管理。
    webSettings.setAppCacheMaxSize(10 * 1024 * 1024);
    //允许缓存，设置缓存位置
    webSettings.setAppCacheEnabled(true);
    webSettings.setAppCachePath(context.getDir("appcache", 0).getPath());
    //允许WebView使用File协议
    webSettings.setAllowFileAccess(true);
    //不保存密码
    webSettings.setSavePassword(false);
    //设置UA
    webSettings.setUserAgentString(webSettings.getUserAgentString() + " kaolaApp/" + AppUt
ls.getVersionName());
    //移除部分系统JavaScript接口
    KaolaWebViewSecurity.removeJavascriptInterfaces(webView);
    //自动加载图片
    webSettings.setLoadsImagesAutomatically(true);
}
```

除此之外，使用方还需要根据业务需求实现WebViewClient和WebChromeClient，这两个类所需要写的方法更多，用来实现标题定制、加载进度条控制、jsbridge交互、url拦截、错误处理（包括http资源、网络）等很多与业务相关的功能。

- 复杂的前端环境

如今，万维网的核心语言，超文本标记语言已经发展到了HTML5，随之而来的是html、css、js相应升级与更新。高版本的语法无法在低版本的内核上识别和渲染，业务上需要使用到新的特性时，开发得不面对后向兼容的问题。互联网的链接千千万万，使用哪些语言特性不是WebView能决定的，要WebView适配所有页面几乎是不可能的事情。

- 版本间差异

WebView不同的版本方法的实现是有可能不一样的，而前端一般情况下只会调用系统的api来实现功，这就会导致Android不同的系统、不同的WebView版本表现不一致的情况。一个典型的例子是下即将描述的WebView中的文件上传功能，当我们在Web页面上点击选择文件的控件(``)时，会产生不的回调方法。除了文件上传功能，版本间的差异还有很多很多，比如缓存机制的版本差异，js安全漏的屏蔽，cookie管理等。Google也在想办法解决这些差异给开发者带来的适配压力，例如Webkit内

到Chromium内核的切换对开发者是透明的，底层的API完全没有改变，这也是好的设计模式带来的处。

- 国内ROM、浏览器对WebView内核的魔改

国产手机的厂商基本在出厂时都自带了浏览器，查看系统应用时，发现并没有内置com.android.webview或者com.google.android.webview包，这些浏览器并不是简单地套了一层WebView的壳，而是接使用了Chromium内核，至于有没有魔改过内核源码，不得而知。国产出品的浏览器，如360浏览、QQ浏览器、UC浏览器，几乎都魔改了内核。值得一提的是，腾讯出品的X5内核，号称页面渲染流度高于原生内核，客户端减少了WebView带来坑的同时，增加了前端适配的难度，功能实现上需要更多地考虑。

- 需要一定的Web知识

如果仅仅会使用WebView.loadUrl()来加载一个网页而不了解底层到底发生了什么，那么url发生错误url中的某些内容加载不出来、url里的内容点击无效、支付宝支付浮层弹不起来、与前端无法沟通等问题就会接踵而至。要开发好一个功能完整的WebView，需要对Web知识（html、js、css）有一定解，知道loadUrl，WebView在后台请求这个url以后，服务器做了哪些响应，又下发了哪些资源，这资源的作用是怎么样的。

# 为什么Github上的WebView项目不适用?

从上面的链接可以看到，Github上面star过千的WebView项目主要是FinestWebView-Android和Android-AdvancedWebView。看过源码的话应该知道，第一个项目偏向于实现一个浏览器，第二个项提供的接口太少，并且一些坑并未填完。陆续看过几个别的开源实现，发现并不理想。后来想想，很不依赖于业务而单独实现一个WebView，特别是与前端约定了jsbridge接口，需要处理页面关闭、屏、url拦截、登录、分享等一系列功能，即便是接入了开源平台的WebView，也需要做大量的扩展有可能完全满足需求。与其如此，每个电商平台都有自己一套规则，基于电商的业务需求来自己扩展ebView是比较合理的。

# WebView踩坑历程

可以说，如果是初次接触WebView，不踩坑几乎是不可能的。笔者在接触到前人留下来的WebView码时，有些地方写的很trickey，如果不仔细阅读，或者翻阅资料，很有可能就会掉进坑里。下面介绍个曾经遇到过的坑。

# WebSettings.setJavaScriptEnabled

我相信99%的应用都会调用下面这句

```
WebSettings.setJavaScriptEnabled(true);
```

在Android 4.3版本调用WebSettings.setJavaScriptEnabled()方法时会调用一下reload方法，同时回调多次WebChromeClient.onJsPrompt()。如果有业务逻辑依赖于这两个方法，就需要注意判断调多次是否会带来影响了。

同时，如果启用了JavaScript，务必做好安全措施，防止远程执行漏洞^5。

```
@TargetApi(11)
private static final void removeJavascriptInterfaces(WebView webView) {
    try {
```

```
        if (Build.VERSION.SDK_INT >= 11 && Build.VERSION.SDK_INT < 17) {
            webView.removeJavascriptInterface("searchBoxJavaBridge_");
            webView.removeJavascriptInterface("accessibility");
            webView.removeJavascriptInterface("accessibilityTraversal");
        }
    } catch (Throwable tr) {
        tr.printStackTrace();
    }
}
```

# 301/302重定向问题

WebView的301/302重定向问题，绝对在踩坑排行榜里名列前茅。。。随便搜了几个解决方案，要不能满足业务需求，要么清一色没有彻底解决问题。

stackoverflow.com/questions/4... blog.csdn.net/jdsjlzx/art... www.cnblogs.com/pedro-neer/... ww.jianshu.com/p/c01769aba...

# 301/302业务场景及白屏问题

先来分析一下业务场景。对于需要对url进行拦截以及在url中需要拼接特定参数的WebView来说，30和302发生的情景主要有以下几种：

- 首次进入，有重定向，然后直接加载H5页面，如http跳转https
- 首次进入，有重定向，然后跳转到native页面，如扫一扫短链，然后跳转到native
- 二次加载，有重定向，跳转到native页面
- 对于考拉业务来说，还有类似登录后跳转到某个页面的需求。如我的拼团，未登录状态下点击我的团跳转到登录页面，登录完成后再加载我的拼团页。

第一种情况属于正常情况，暂时没遇到什么坑。

第二种情况，会遇到**WebView空白页问题**，属于原始url不能拦截到native页面，但301/302后的url截到native页面的情况，当遇到这种情况时，需要把WebView对应的Activity结束，否则当用户从拦后的页面返回上一个页面时，是一个WebView空白页。

第三种情况，也会遇到**WebView空白页问题**，原因在于加载的第一个页面发生了重定向到了第二个面，第二个页面被客户端拦截跳转到native页面，那么WebView就停留在第一个页面的状态了，第个页面显然是空白页。

第四种情况，会遇到**无限加载登录页面的问题**。考拉的登录链接是类似下面这种格式：

https://m.kaola.com/login.html?target=登录后跳转的url

如果登录成功后还重新加载这个url，那么就会循环跳转到登录页面。第四点解决起来比较简单，登录功以后拿到target后的跳转url再重新加载即可。

# 301/302回退栈问题

无论是哪种重定向场景，都不可避免地会遇到回退栈的处理问题，如果处理不当，用户按返回键的时

不一定能回到重定向之前的那个页面。很多开发者在覆写WebViewClient.shouldOverrideUrlLoadin()方法时，会简单地使用以下方式粗暴处理：

```
WebView.setWebViewClient(new WebViewClient() {
    @Override
    public boolean shouldOverrideUrlLoading(WebView view, String url) {
        view.loadUrl(url);
        return true;
    }
    ...
)
```

这种方法最致命的弱点就是如果不经过特殊处理，那么按返回键是没有效果的，还会停留在302之前页面。现有的解决方案无非就几种：

1. 手动管理回退栈，遇到重定向时回退两次 ^6。
2. 通过 HitTestResult判断是否是重定向，从而决定是否自己加载url^7 ^8。
3. 通过设置标记位，在 onPageStarted和onPageFinished分别标记变量避免重定向^9。

可以说，这几种解决方案都不是完美的，都有缺陷。

# 301/302较优解决方案

## 解决301/302回退栈问题

能否结合上面的几种方案，来更加准确地判断301/302的情况呢？下面说一下本文的解决思路。在提解决方案之前，我们需要了解一下shouldOverrideUrlLoading方法的返回值代表什么意思。

Give the host application a chance to take over the control when a new url is about to be loaded in the current WebView. If WebViewClient is not provided, by default WebView will ask Activity Manager to choose the proper handler for the url. If WebViewClient is provided, **return true means the host application handles the url, while return false means the current Webiew handles the url.**

简单地说，就是返回true，那么url就已经由客户端处理了，WebView就不管了，如果返回false，那当前的WebView实现就会去处理这个url。

WebView能否知道某个url是不是301/302呢？当然知道，WebView能够拿到url的请求信息和响应息，根据header里的code很轻松就可以实现，事实正是如此，交给WebView来处理重定向(return false)，这时候按返回键，是可以正常地回到重定向之前的那个页面的。（PS：从上面的章节可知，WeView在5.0以后是一个独立的apk，可以单独升级，新版本的WebView实现肯定处理了重定向问题）

但是，业务对url拦截有需求，肯定不能把所有的情况都交给系统WebView处理。为了解决url拦截问，本文引入了另一种思想——通过用户的touch事件来判断重定向。下面通过代码来说明。

```
/**
 * WebView基础类，处理一些基础的公有操作
 *
 * @author xingli
 * @time 2017-12-06
 */
```

```
public class BaseWebView extends WebView {

    private boolean mTouchByUser;

    public BaseWebView(Context context) {
        super(context);
    }

    public BaseWebView(Context context, AttributeSet attrs) {
        super(context, attrs);
    }

    public BaseWebView(Context context, AttributeSet attrs, int defStyleAttr) {
        super(context, attrs, defStyleAttr);
    }

    @Override
    public final void loadUrl(String url, Map<String, String> additionalHttpHeaders) {
        super.loadUrl(url, additionalHttpHeaders);
        resetAllStateInternal(url);
    }

    @Override
    public void loadUrl(String url) {
        super.loadUrl(url);
        resetAllStateInternal(url);
    }

    @Override
    public final void postUrl(String url, byte[] postData) {
        super.postUrl(url, postData);
        resetAllStateInternal(url);
    }

    @Override
    public final void loadData(String data, String mimeType, String encoding) {
        super.loadData(data, mimeType, encoding);
        resetAllStateInternal(getUrl());
    }

    @Override
    public final void loadDataWithBaseURL(String baseUrl, String data, String mimeType, String
encoding,
            String historyUrl) {
        super.loadDataWithBaseURL(baseUrl, data, mimeType, encoding, historyUrl);
        resetAllStateInternal(getUrl());
    }

    @Override
    public void reload() {
        super.reload();
        resetAllStateInternal(getUrl());
    }
```

```
    public boolean isTouchByUser() {
        return mTouchByUser;
    }

    private void resetAllStateInternal(String url) {
        if (!TextUtils.isEmpty(url) && url.startsWith("javascript:")) {
            return;
        }
        resetAllState();
    }

    // 加载url时重置touch状态
    protected void resetAllState() {
        mTouchByUser = false;
    }

    @Override
    public boolean onTouchEvent(MotionEvent event) {
        switch (event.getAction()) {
            case MotionEvent.ACTION_DOWN:
                //用户按下到下一个链接加载之前，置为true
                mTouchByUser = true;
                break;
        }
        return super.onTouchEvent(event);
    }

    @Override
    public void setWebViewClient(final WebViewClient client) {
        super.setWebViewClient(new WebViewClient() {
            @Override
            public boolean shouldOverrideUrlLoading(WebView view, String url) {
                boolean handleByChild = null != client && client.shouldOverrideUrlLoading(view, ur
);
                  if (handleByChild) {
                    // 开放client接口给上层业务调用，如果返回true，表示业务已处理。
                   return true;
                  } else if (!isTouchByUser()) {
                    // 如果业务没有处理，并且在加载过程中用户没有再次触摸屏幕，认为是301/302事件
直接交由系统处理。
                   return super.shouldOverrideUrlLoading(view, url);
                } else {
                    //否则，属于二次加载某个链接的情况，为了解决拼接参数丢失问题，重新调用loadUrl
法添加固有参数。
                    loadUrl(url);
                    return true;
                }
            }

            @RequiresApi(api = Build.VERSION_CODES.N)
            @Override
            public boolean shouldOverrideUrlLoading(WebView view, WebResourceRequest requ
st) {
                boolean handleByChild = null != client && client.shouldOverrideUrlLoading(view, r
```

```
quest);

                if (handleByChild) {
                    return true;
                } else if (!isTouchByUser()) {
                    return super.shouldOverrideUrlLoading(view, request);
                } else {
                    loadUrl(request.getUrl().toString());
                    return true;
                }
            }
        });
    }
}
```

上述代码解决了正常情况下的回退栈问题。

## 解决业务白屏问题

为了解决白屏问题，考拉目前的解决思路和上面的回退栈问题思路有些类似，通过监听touch事件分以及onPageFinished事件来判断是否产生白屏，代码如下：

```
public class KaolaWebview extends BaseWebView implements DownloadListener, Lifeful, OnA
tivityResultListener {

    private boolean mIsBlankPageRedirect;  //是否因重定向导致的空白页面。

    public KaolaWebview(Context context) {
        super(context);
        init();
    }

    public KaolaWebview(Context context, AttributeSet attrs) {
        super(context, attrs);
        init();
    }

    public KaolaWebview(Context context, AttributeSet attrs, int defStyleAttr) {
        super(context, attrs, defStyleAttr);
        init();
    }

    protected void back() {
        if (mBackStep < 1) {
            mJsApi.trigger2("kaolaGoback");
        } else {
            realBack();
        }
    }

    @Override
    public boolean dispatchTouchEvent(MotionEvent ev) {
        if (ev.getAction() == MotionEvent.ACTION_UP) {
```

```
            mIsBlankPageRedirect = true;
        }
        return super.dispatchTouchEvent(ev);
    }

    private WebViewClient mWebViewClient = new WebViewClient() {
        @Override
        public boolean shouldOverrideUrlLoading(WebView view, String url) {
            url = WebViewUtils.removeBlank(url);
            //允许启动第三方应用客户端
            if (WebViewUtils.canHandleUrl(url)) {
                boolean handleByCaller = false;
                // 如果不是用户触发的操作，就没有必要交给上层处理了，直接走url拦截规则。
                if (null != mIWebViewClient && isTouchByUser()) {
                    handleByCaller = mIWebViewClient.shouldOverrideUrlLoading(view, url);
                }
                if (!handleByCaller) {
                    handleByCaller = handleOverrideUrl(url);
                }
                return handleByCaller || super.shouldOverrideUrlLoading(view, url);
            } else {
                try {
                    notifyBeforeLoadUrl(url);
                    Intent intent = Intent.parseUri(url, Intent.URI_INTENT_SCHEME);
                    intent.addCategory(Intent.CATEGORY_BROWSABLE);
                    mContext.startActivity(intent);
                    if (!mIsBlankPageRedirect) {
                        // 如果遇到白屏问题，手动后退
                        back();
                    }
                } catch (Exception e) {
                    ExceptionUtils.printExceptionTrace(e);
                }
                return true;
            }
        }

        @RequiresApi(Build.VERSION_CODES.LOLLIPOP)
        @Override
        public boolean shouldOverrideUrlLoading(WebView view, WebResourceRequest request)
{
            return shouldOverrideUrlLoading(view, request.getUrl().toString());
        }

        private boolean handleOverrideUrl(final String url) {
           RouterResult result =  WebActivityRouter.startFromWeb(
                    new IntentBuilder(mContext, url).setRouterActivityResult(new RouterActivityResul
() {
                        @Override
                        public void onActivityFound() {
                            if (!mIsBlankPageRedirect) {
                                // 路由已经拦截到跳转到native页面，但此时可能发生了
                                // 301/302跳转，那么执行后退动作，防止白屏。
                                back();
```

```
                    }
                }

                @Override
                public void onActivityNotFound() {
                    if (mIWebViewClient != null) {
                        mIWebViewClient.onActivityNotFound();
                    }
                }
            }));
        return result.isSuccess();
    }
};

@Override
public void onPageFinished(WebView view, String url) {
    mIsBlankPageRedirect = true;
    if (null != mIWebViewClient) {
        mIWebViewClient.onPageReallyFinish(view, url);
    }
    super.onPageFinished(view, url);
}
}
```

本来上面的两个问题可以用同一个变量控制解决的，但由于历史代码遗留问题，目前还没有时间优化试，这也是代码暂不公布的原因之一（代码太丑陋:()。

# url参数拼接问题

一般情况下，WebView会拼接一些本地参数作为识别码传给前端，如app版本号，网络状态等，例如要加载的url是

http://m.kaola.com?platform=android

假设我们拼接appVersion和network，则拼接后url变成：

http://m.kaola.com?platform=android&appVersion=3.10.0&network=4g

使用WebView.loadUrl()加载上面拼接好的url，随意点击这个页面上的某个链接跳转到别的页面，本拼接的参数是不会自动带过去的。如果需要前端处理参数问题，那么如果是同域，可以通过cookie传。非同域的话，还是需要客户端拼接参数带过去。

# 部分机型没有WebView，应用直接崩溃

在Crash平台上面发现有部分机型会存在下面这个崩溃，这些机型都是7.0系统及以上的。

```
android.util.AndroidRuntimeException: android.webkit.WebViewFactory$MissingWebViewPac
ageException: Failed to load WebView provider: No WebView installed
at android.webkit.WebViewFactory.getProviderClass(WebViewFactory.java:371)
at android.webkit.WebViewFactory.getProvider(WebViewFactory.java:194)
```

```
at android.webkit.WebView.getFactory(WebView.java:2325)
at android.webkit.WebView.ensureProviderCreated(WebView.java:2320)
at android.webkit.WebView.setOverScrollMode(WebView.java:2379)
at android.view.View.(View.java:4015)
at android.view.View.(View.java:4132)
at android.view.ViewGroup.(ViewGroup.java:578)
at android.widget.AbsoluteLayout.(AbsoluteLayout.java:55)
at android.webkit.WebView.(WebView.java:627)
at android.webkit.WebView.(WebView.java:572)
at android.webkit.WebView.(WebView.java:555)
at android.webkit.WebView.(WebView.java:542)
at com.kaola.modules.webview.BaseWebView.void (android.content.Context)(Unknown Sourc
)
```

经过测试发现，普通用户是没有办法卸载WebView的（即使能卸载，也只是把更新卸载了，原始版的WebView还是存在的），所以理论上不会存在异常……但既然发生并且上传上来了，那么就需要细分析一下原因了。跟着代码WebViewFactory.getProvider()走，

```
static WebViewFactoryProvider getProvider() {
    synchronized (sProviderLock) {
        // For now the main purpose of this function (and the factory abstraction) is to keep
        // us honest and minimize usage of WebView internals when binding the proxy.
        if (sProviderInstance != null) return sProviderInstance;

        final int uid = android.os.Process.myUid();
        if (uid == android.os.Process.ROOT_UID || uid == android.os.Process.SYSTEM_UID
                || uid == android.os.Process.PHONE_UID || uid == android.os.Process.NFC_UID
                || uid == android.os.Process.BLUETOOTH_UID) {
            throw new UnsupportedOperationException(
                    "For security reasons, WebView is not allowed in privileged processes");
        }

        StrictMode.ThreadPolicy oldPolicy = StrictMode.allowThreadDiskReads();
        Trace.traceBegin(Trace.TRACE_TAG_WEBVIEW, "WebViewFactory.getProvider()");
        try {
            Class<WebViewFactoryProvider> providerClass = getProviderClass();
            Method staticFactory = null;
            try {
                staticFactory = providerClass.getMethod(
                        CHROMIUM_WEBVIEW_FACTORY_METHOD, WebViewDelegate.class);
            } catch (Exception e) {
                if (DEBUG) {
                    Log.w(LOGTAG, "error instantiating provider with static factory method", e);
                }
            }

            Trace.traceBegin(Trace.TRACE_TAG_WEBVIEW, "WebViewFactoryProvider invocation");
            try {
                sProviderInstance = (WebViewFactoryProvider)
                        staticFactory.invoke(null, new WebViewDelegate());
                if (DEBUG) Log.v(LOGTAG, "Loaded provider: " + sProviderInstance);
                return sProviderInstance;
            } catch (Exception e) {
```

```
                Log.e(LOGTAG, "error instantiating provider", e);
                throw new AndroidRuntimeException(e);
            } finally {
                Trace.traceEnd(Trace.TRACE_TAG_WEBVIEW);
            }
        } finally {
            Trace.traceEnd(Trace.TRACE_TAG_WEBVIEW);
            StrictMode.setThreadPolicy(oldPolicy);
        }
    }
}
```

可以看到，获取WebView的实例，就是先拿到WebViewFactoryProvider这个工厂类，通过WebViewFactoryProvider工厂类里的静态方法CHROMIUM_WEBVIEW_FACTORY_METHOD创建一个WebViewFactoryProvider，接着，调用WebViewFactoryProvider.createWebView()创建一个WebViewProvider（相当于WebView的代理类），后面WebView的方法都是通过代理类来实现的。

在第一步获取WebVIewFactoryProvider类的过程中，

```
private static Class<WebViewFactoryProvider> getProviderClass() {
    Context webViewContext = null;
    Application initialApplication = AppGlobals.getInitialApplication();

    try {
        //获取WebView上下文并设置provider
        webViewContext = getWebViewContextAndSetProvider();
    } finally {
        Trace.traceEnd(Trace.TRACE_TAG_WEBVIEW);
    }
     代码省略...
    }
}

private static Context getWebViewContextAndSetProvider() {
    Application initialApplication = AppGlobals.getInitialApplication();
    WebViewProviderResponse response = null;
    Trace.traceBegin(Trace.TRACE_TAG_WEBVIEW,
            "WebViewUpdateService.waitForAndGetProvider()");
    try {
        response = getUpdateService().waitForAndGetProvider();
    } finally {
        Trace.traceEnd(Trace.TRACE_TAG_WEBVIEW);
    }
    if (response.status != LIBLOAD_SUCCESS
            && response.status != LIBLOAD_FAILED_WAITING_FOR_RELRO) {
        // 崩溃就发生在这里。
        throw new MissingWebViewPackageException("Failed to load WebView provider: "
                + getWebViewPreparationErrorReason(response.status));
    }
}
```

可以发现，在与WebView包通信的过程中，so库并没有加载成功，最后代码到了native层，没有继

跟下去了。

对于这种问题，解决方案有两种，一种是判断包名，如果检测到系统包名里不包含com.google.android.webview或者com.android.webview，则认为用户手机里的WebView不可用；另外一种是通过try/atch判断WebView实例化是否成功，如果抛出了WebViewFactory$MissingWebViewPackageException异常，则认为用户的WebView不可用。

需要说明的是，第一种解决方案是不可靠的，因为国内的厂商基于Chromium的WebView实现有很种，很有可能包名就被换了，比如MiWebView，包名是com.mi.webkit.core。

# WebView中的POST请求

在WebView中，如果前端使用POST方式向后端发起一个请求，那么这个请求是不会走到WebViewClient.shouldOverrideUrlLoading()方法里的^10。网上有一些解决方案，例如android-post-webview通过js判断是否是post请求，如果是的话，在WebViewClient.shouldInterceptRequest()方法里自建立连接，并拿到对应的页面信息，返回给WebResourceResponse。总之，尽量避免Web页面使用OST请求，否则会带来很大不必要的麻烦。

# WebView文件上传功能

WebView中的文件上传功能，当我们在Web页面上点击选择文件的控件(``)时，会产生不同的回调方法:4

void openFileChooser(ValueCallback uploadMsg) works on Android 2.2 (API level 8) up to Android 2.3 (API level 10)

openFileChooser(ValueCallback uploadMsg, String acceptType) works on Android 3.0 (API level 11) up to Android 4.0 (API level 15)

openFileChooser(ValueCallback uploadMsg, String acceptType, String capture) works on Android 4.1 (API level 16) up to Android 4.3 (API level 18)

onShowFileChooser(WebView webView, ValueCallback filePathCallback, WebChromeClient.FileChooserParams fileChooserParams) works on Android 5.0 (API level 21) and above

最坑的点是在Android4.4系统上没有回调，这将导致功能的不完整，需要前端去做兼容。解决方案就和前端另外约定一个jsbridge来解决此类问题。

# 总结

限于篇幅，《如何设计一个优雅健壮的Android WebView？（上）》先介绍到这里。本文介绍了目Android里的WebView现状，以及由于现状的不可改变导致遗留下的一些坑。所幸，世界上没有什代码问题是一个程序员不能解决的，如果有，那就用两个程序员解决。既然我们已经把前人留下的一坑填了，那么是时候构造一个可以用于生产环境的WebView了！《如何设计一个优雅健壮的Android WebView？（下）》将会介绍如何打造WebView的实战操作，以及为了用户更好的体验，提出的一WebView优化策略，敬请期待。

# 参考链接


1. developer.chrome.com/multidevice...
2. developer.android.com/about/versi...

3. developer.android.com/about/versi...
4. stackoverflow.com/questions/3...
5. blog.csdn.net/self_study/...
6. qbeenslee.com/article/and...
7. juejin.im/entry/59775...
8. www.cnblogs.com/zimengfang/...
9. blog.csdn.net/dg_summer/a...
10. issuetracker.google.com/issues/3691...