一、动态地生成将注入的JS代码  
  
JsCallJava在构造时，将要注入类的public且static方法拿出来，逐个生成方法的签名，依据方法签名先将方法缓存起来，同时结合方法名称与静态的HostApp-JS代码动态生成一段将要注入到webview中的字符串。

1. public JsCallJava (String injectedName, Class injectedCls) {
2. try {
3. mMethodsMap = new HashMap<String, Method>();
4. //获取自身声明的所有方法（包括public private protected）， getMethods会获得所有继承与非继承的方法
5. Method[] methods = injectedCls.getDeclaredMethods();
6. StringBuilder sb = new StringBuilder("javascript:(function(b){console.log(\"HostApp initialization begin\");var a={queue:[],callback:function(){var d=Array.prototype.slice.call(arguments,0);var c=d.shift();var e=d.shift();this.queue[c].apply(this,d);if(!e){delete this.queue[c]}}};");
7. for (Method method : methods) {
8. String sign;
9. if (method.getModifiers() != (Modifier.PUBLIC | Modifier.STATIC) || (sign = genJavaMethodSign(method)) == null) {
10. continue;
11. }
12. mMethodsMap.put(sign, method);
13. sb.append(String.format("a.%s=", method.getName()));
14. }
15. sb.append("function(){var f=Array.prototype.slice.call(arguments,0);if(f.length<1){throw\"HostApp call error, message:miss method name\"}var e=[];for(var h=1;h<f.length;h++){var c=f[h];var j=typeof c;e[e.length]=j;if(j==\"function\"){var d=a.queue.length;a.queue[d]=c;f[h]=d}}var g=JSON.parse(prompt(JSON.stringify({method:f.shift(),types:e,args:f})));if(g.code!=200){throw\"HostApp call error, code:\"+g.code+\", message:\"+g.result}return g.result};Object.getOwnPropertyNames(a).forEach(function(d){var c=a[d];if(typeof c===\"function\"&&d!==\"callback\"){a[d]=function(){return c.apply(a,[d].concat(Array.prototype.slice.call(arguments,0)))}}});b." + injectedName + "=a;console.log(\"HostApp initialization end\")})(window);");
16. mPreloadInterfaceJS = sb.toString();
17. } catch(Exception e){
18. Log.e(TAG, "init js error:" + e.getMessage());
19. }
20. }
21. private String genJavaMethodSign (Method method) {
22. String sign = method.getName();
23. Class[] argsTypes = method.getParameterTypes();
24. int len = argsTypes.length;
25. if (len < 1 || argsTypes[0] != WebView.class) {
26. Log.w(TAG, "method(" + sign + ") must use webview to be first parameter, will be pass");
27. return null;
28. }
29. for (int k = 1; k < len; k++) {
30. Class cls = argsTypes[k];
31. if (cls == String.class) {
32. sign += "\_S";
33. } else if (cls == int.class ||
34. cls == long.class ||
35. cls == float.class ||
36. cls == double.class) {
37. sign += "\_N";
38. } else if (cls == boolean.class) {
39. sign += "\_B";
40. } else if (cls == JSONObject.class) {
41. sign += "\_O";
42. } else if (cls == JsCallback.class) {
43. sign += "\_F";
44. } else {
45. sign += "\_P";
46. }
47. }
48. return sign;

从上面可以看出，类的各个方法名称被拼接到前后两段静态压缩的JS代码当中，那么这样生成的完整清晰的HostApp-JS片段是怎样的呢？ 我们假设HostJsScope类中目前只定义了toast、alert、getIMSI这三个公开静态方法，那么完整的片段就是下面这样：

1. (function(global){
2. console.log("HostApp initialization begin");
3. var hostApp = {
4. queue: [],
5. callback: function () {
6. var args = Array.prototype.slice.call(arguments, 0);
7. var index = args.shift();
8. var isPermanent = args.shift();
9. this.queue[index].apply(this, args);
10. if (!isPermanent) {
11. delete this.queue[index];
12. }
13. }
14. };
15. hostApp.toast = hostApp.alert = hostApp.getIMSI = function () {
16. var args = Array.prototype.slice.call(arguments, 0);
17. if (args.length < 1) {
18. throw "HostApp call error, message:miss method name";
19. }
20. var aTypes = [];
21. for (var i = 1;i < args.length;i++) {
22. var arg = args[i];
23. var type = typeof arg;
24. aTypes[aTypes.length] = type;
25. if (type == "function") {
26. var index = hostApp.queue.length;
27. hostApp.queue[index] = arg;
28. args[i] = index;
29. }
30. }
31. var res = JSON.parse(prompt(JSON.stringify({
32. method: args.shift(),
33. types: aTypes,
34. args: args
35. })));
36. if (res.code != 200) {
37. throw "HostApp call error, code:" + res.code + ", message:" + res.result;
38. }
39. return res.result;
40. };
41. //有时候，我们希望在该方法执行前插入一些其他的行为用来检查当前状态或是监测
42. //代码行为，这就要用到拦截（Interception）或者叫注入（Injection）技术了
43. /\*\*
44. \* Object.getOwnPropertyName 返回一个数组，内容是指定对象的所有属性
45. \*
46. \* 其后遍历这个数组，分别做以下处理：
47. \* 1. 备份原始属性；
48. \* 2. 检查属性是否为 function（即方法）；
49. \* 3. 若是重新定义该方法，做你需要做的事情，之后 apply 原来的方法体。
50. \*/
51. Object.getOwnPropertyNames(hostApp).forEach(function (property) {
52. var original = hostApp[property];
53. if (typeof original === 'function'&&property!=="callback") {
54. hostApp[property] = function () {
55. return original.apply(hostApp,  [property].concat(Array.prototype.slice.call(arguments, 0)));
56. };
57. }
58. });
59. global.HostApp = hostApp;
60. console.log("HostApp initialization end");
61. })(window);

其实在JsCallJava初始化时我们拼接的只是上面第15行 hostApp.toast = hostApp.alert = hostApp.getIMSI = function () 这段。目的是将所有JS层调用函数嫁接到一个匿名函数1中，而后利用拦截技术，遍历hostApp下所有的函数，拿出对应的函数名，然后将hostApp 下所有的函数调用嫁接到另一个匿名函数2，这样做的目的是hostApp下函数调用时首先执行匿名函数2，匿名函数2将对应的函数名作为第一个参数然后再 调用匿名函数1，这样匿名函数1中就能区分执行时调用来源。实现了JS层调用入口统一，返回出口统一的结构体系。  
  
二、HostApp JS片段注入时机  
  
步骤一说明了HostApp-JS片段的拼接方法，同时JS片段拼接是在JsCallJava初始化完成的，而JsCallJava初始化是在[实例](http://www.html5cn.org/portal.php?mod=list&catid=17" \t "_blank)化InjectedChromeClient对象时发起的。

1. public InjectedChromeClient (String injectedName, Class injectedCls) {
2. mJsCallJava = new JsCallJava(injectedName, injectedCls);
3. }

从步骤一的代码，我们知道JsCallJava拼接出来的JS代码暂时被存到mPreloadInterfaceJS字段中。那么我们何时把这段代码串注入到Webview的页面空间内呢？答案是页面加载进度变化的过程中。

1. @Override
2. public void onProgressChanged (WebView view, int newProgress) {
3. //为什么要在这里注入JS
4. //1 OnPageStarted中注入有可能全局注入不成功，导致页面脚本上所有接口任何时候都不可用
5. //2 OnPageFinished中注入，虽然最后都会全局注入成功，但是完成时间有可能太晚，当页面在初始化调用接口函数时会等待时间过长
6. //3 在进度变化时注入，刚好可以在上面两个问题中得到一个折中处理
7. //为什么是进度大于25%才进行注入，因为从测试看来只有进度大于这个数字页面才真正得到框架刷新加载，保证100%注入成功
8. if (newProgress <= 25) {
9. mIsInjectedJS = false;
10. } else if (!mIsInjectedJS) {
11. view.loadUrl(mJsCallJava.getPreloadInterfaceJS());
12. mIsInjectedJS = true;
13. Log.d(TAG, " inject js interface completely on progress " + newProgress);
14. }
15. super.onProgressChanged(view, newProgress);
16. }

从上面我们可以看出，注入的时机是准确把握在进度大于25%时。如果在OnPageFinished注入，页面document.ready的初始回调会等待时间过长，详细的原因我们会在后面讲到。  
  
三、页面调用Java方法执行的过程  
  
OK，上面两步解决了动态生成与成功注入的两大问题，接下来就要处理JS具体的调用过程。上面，我们知道页面调用Java方法时，匿名js函数在拼 接好参数后prompt json数据。prompt消息被Java层的WebChromeClient.onJsPrompt拦截到。

1. @Override
2. public boolean onJsPrompt(WebView view, String url, String message, String defaultValue, JsPromptResult result) {
3. result.confirm(mJsCallJava.call(view, message));
4. return true;
5. }

而JsCallJava.call的具体实现如下。

1. public String call(WebView webView, String jsonStr) {
2. if (!TextUtils.isEmpty(jsonStr)) {
3. try {
4. JSONObject callJson = new JSONObject(jsonStr);
5. String methodName = callJson.getString("method");
6. JSONArray argsTypes = callJson.getJSONArray("types");
7. JSONArray argsVals = callJson.getJSONArray("args");
8. String sign = methodName;
9. int len = argsTypes.length();
10. Object[] values = new Object[len + 1];
11. int numIndex = 0;
12. String currType;
13. values[0] = webView;
14. for (int k = 0; k < len; k++) {
15. currType = argsTypes.optString(k);
16. if ("string".equals(currType)) {
17. sign += "\_S";
18. values[k + 1] = argsVals.isNull(k) ? null : argsVals.getString(k);
19. } else if ("number".equals(currType)) {
20. sign += "\_N";
21. numIndex = numIndex \* 10 + k + 1;
22. } else if ("boolean".equals(currType)) {
23. sign += "\_B";
24. values[k + 1] = argsVals.getBoolean(k);
25. } else if ("object".equals(currType)) {
26. sign += "\_O";
27. values[k + 1] = argsVals.isNull(k) ? null : argsVals.getJSONObject(k);
28. } else if ("function".equals(currType)) {
29. sign += "\_F";
30. values[k + 1] = new JsCallback(webView, argsVals.getInt(k));
31. } else {
32. sign += "\_P";
33. }
34. }
35. Method currMethod = mMethodsMap.get(sign);
36. // 方法匹配失败
37. if (currMethod == null) {
38. return getReturn(jsonStr, 500, "not found method(" + methodName + ") with valid parameters");
39. }
40. // 数字类型细分匹配
41. if (numIndex > 0) {
42. Class[] methodTypes = currMethod.getParameterTypes();
43. int currIndex;
44. Class currCls;
45. while (numIndex > 0) {
46. currIndex = numIndex - numIndex / 10 \* 10;
47. currCls = methodTypes[currIndex];
48. if (currCls == int.class) {
49. values[currIndex] = argsVals.getInt(currIndex - 1);
50. } else if (currCls == long.class) {
51. //WARN: argsJson.getLong(k + defValue) will return a bigger incorrect number
52. values[currIndex] = Long.parseLong(argsVals.getString(currIndex - 1));
53. } else {
54. values[currIndex] = argsVals.getDouble(currIndex - 1);
55. }
56. numIndex /= 10;
57. }
58. }
59. return getReturn(jsonStr, 200, currMethod.invoke(null, values));
60. } catch (Exception e) {
61. //优先返回详细的错误信息
62. if (e.getCause() != null) {
63. return getReturn(jsonStr, 500, "method execute error:" + e.getCause().getMessage());
64. }
65. return getReturn(jsonStr, 500, "method execute error:" + e.getMessage());
66. }
67. } else {
68. return getReturn(jsonStr, 500, "call data empty");
69. }
70. }

这是一个完整的解析匹配过程，会依据js层传入的方法名、参数类型列表再次生成方法签名，与之前初始化构造好的缓存对象中的方法匹配。匹配成功后则 判断js调用参数类型中是否有number类型，如果有依据Java层方法的定义决定是取int、long还是double类型的值。最后使用调用值列表 和方法对象反射执行，返回函数执行的结果。这里有几点需要注意：

* 方法反射执行时会将当前WebView的实例放到第一个参数，方便在HostJsScope静态方法依据Context拿到一些相关上下文信息；
* 注入类（如HostJsScope）静态方法的参数定义可使用的类型有int/long/double、String、boolean、 JSONObject、JsCallback，对应于js层传入的类型为number、string、boolean、object、function， 注意number数字过大时（如时间戳），可能需要先转为string类型（Java方法中参数也须定义为String），避免精度丢失；
* Java方法的返回值可以是void 或 能转为字符串的类型（如int、long、String、double、float等）或 可序列化的自定义类型；
* 如果执行失败或找不到调用方法时，Java层会将异常信息传递到JS层， JS匿名函数中会throw抛出错误；

四、HostApp在页面的使用  
  
有了上面的准备工作，现在我们在页面中就可以很方便地使用HostApp了，而不需要加载任何依赖文件。如li标签的点击：

1. <ul class="entry">
2. <li onclick="HostApp.alert('HostApp.alert');">HostApp.alert</li>
3. <li onclick="HostApp.toast('HostApp.toast');">HostApp.toast</li>
4. <li onclick="HostApp.testLossTime(new Date().getTime() + '');">HostApp.testLossTime</li> <!-- 时间戳长整型调用前先转换为string -->
5. <li onclick="HostApp.toast(HostApp.getIMSI());">HostApp.getIMSI</li>
6. </ul>

但同时有一种业务情景时，页面初始加载完备时就应立即触发的调用，如果我们这样写：

1. document.addEventListener('DOMContentLoaded', function() {
2. HostApp.toast('document ready now');;
3. }, false);

那么HostApp的调用极有可能不成功，因为端注入HostApp-JS片段的时机可能在document.ready前也可能在其后。那么如何解决这个矛盾的问题呢？  
  
如果document.ready的时候HostApp JS已经注入成功，这种情况OK没有问题。当document.ready的时候HostApp JS还未开始注入，这种情景下我们的js脚本层就需要做出变动，即轮询状态，直到端注入成功或者超时（1.5s），再发生回调。具体实现如下（下面的是以 zepto.js的$.ready()函数改造为例）。

1. //针对DOM的一些操作
2. // Define methods that will be available on all
3. // Zepto collections
4. $.fn = {
5. //DOM Ready
6. ready: function(callback, jumpHostAppInject) {
7. var originCb = callback;
8. var mcounter = 0;
9. //尝试等待(1500ms超时)让端注入HostApp Js
10. callback = function () {
11. if(!window.HostApp && mcounter++ < 150)setTimeout(callback, 10);else originCb($);
12. };
13. //是否跳过等待HostApp的注入
14. if (jumpHostAppInject) {
15. callback = originCb;
16. }
17. if (readyRE.test(document.readyState)) callback($); else document.addEventListener('DOMContentLoaded', function() {
18. callback($)
19. }, false);
20. return this
21. },
22. ...
23. ...
24. };

这样的机制也就解释了为什么不把Java层的JS注入放在OnPageFinish了，如果那样页面轮询的次数就会上升，等待的时间就会变长，而且有可能会超时。好了，有了上面的改动，页面初始加载完备时需要立即触发HostApp的调用，如下：

1. <script type="text/javascript">
2. $(function () {
3. HostApp.alert("HostApp ready now");
4. });
5. </script>