Android部分的 WebView容器代码可独立成一个代码库单独维护，并打包成aar给主工程引用。

我们出产这么一个SDK，目的是给与整个公司的研发部门出具一份前端H5与原生容器的统一交互方案，并使用统一的Web容器，避免各个业务方自行开发容器造成资源浪费。当然，做的就是全平台，包括iOS，Android，Flutter，本文只讲解 Android的设计概要，其他平台部分细节可以类推。

# 目标

SDK初始化流程

阅读一个SDK我们通常会从它的Demo入手，让demo运行起来看具体效果。而在代码中，第一个进入视线的就是Application这个类的配置了。一个SDK的初始化代码，在调用方看来应该尽可能简单，而且对性能影响最小化，否则会影响到app的启动速度。

WebView的引入和常用设置

在国内市面上主流的web容器，通常是以X5为内核，本节将讲述X5的常用配置，H5原生混合开发中通常会出现一些莫名其妙的兼容问题，这种问题有的可以通过WebView的 配置来规避掉。

webView与H5的交互流程设计

作为一个H5业务模块的web容器，很多功能要借助原生能力来达成最佳的体验，js与native的交互，我们基本都是以 WebView的 addJavascriptInterface()方法来添加一个native对象作为交互媒介, 随着业务的迭代 js与native的交互协议数量可能会无限膨胀，我们需要对协议进行科学的设计来应对协议的扩展，避免代码变成屎山。另外，为了应对多各业务方在交互协议上的差异，还需要提供业务方自定义的扩展协议，以及 拦截原协议的入口。

本节除了包含 android侧的原生代码设计要素之外，还有jsBridge,即H5业务方需要引入的js桥文件，用于与native进行交互。

webView的容错容灾设计

WebView有自己的脾气，有时候出现的问题并不在我们的预料之内。出现问题的原因可能是 H5自己，也有可能是 WebView自身的配置。如果我们给SDK接入方一个监测容器状态的入口，这样出现问题，就不再每次都需要我们亲自处理，业务方能够自行处理一部分问题，由此来减轻我们SDK开发者的压力。做过SDK的人应该深有体会，当有几十上百个接入方在工作群对你口诛笔伐的时候，心情很沉重，然后一检查，问题并不在SDK本身，又是一阵无语。

# 任务分解

注：以下代码都是伪代码。变量名纯属虚构。

## SDK初始化流程

Application类中，SDK需要做的初始化只有一条:

class MyApp : Application() {

override fun onCreate() {

super.onCreate()

X5WebSDK.init(this)

}

}

WebSDK的init内部需要做的是 初始化X5，以及初始化WebView。

object X5WebSDK {

private val TAG = this.javaClass.simpleName

/\*\*

\* 初始化

\*/

fun init(

context: Context,

disableX5: Boolean = false,

) {

initX5(context, disableX5)

initWebView(context)

}

// 初始化 X5

private fun initX5(context: Context, disableX5: Boolean) {

val initStartTime = System.currentTimeMillis()

val map = HashMap<String, Any>()

map[TbsCoreSettings.TBS\_SETTINGS\_USE\_SPEEDY\_CLASSLOADER] = true //

map[TbsCoreSettings.TBS\_SETTINGS\_USE\_DEXLOADER\_SERVICE] = true // 为了解决首次加载X5内核的卡顿问题

QbSdk.initTbsSettings(map)

QbSdk.initX5Environment(context, object : QbSdk.PreInitCallback {

override fun onCoreInitFinished() {

val initEndTime = System.currentTimeMillis()

LogUtils.d("$TAG Init X5 onCoreInitFinished Cast: ${initEndTime - initStartTime}ms")

}

override fun onViewInitFinished(isX5: Boolean) {

val initEndTime = System.currentTimeMillis()

LogUtils.d("$TAG Init X5 onViewInitFinished isX5:$isX5 Cast: ${initEndTime - initStartTime}ms")

}

})

if (disableX5) disableX5(context)

val initEndTime = System.currentTimeMillis()

LogUtils.d("$TAG Init X5 Cast: ${initEndTime - initStartTime}ms")

}

// 禁用 X5

private fun disableX5(context: Context) {

LogUtils.d("$TAG disableX5")

val debugConfFile = File(

context.filesDir.path.substring(

0,

context.filesDir.path.lastIndexOf("/")

) + "/app\_tbs/core\_private/debug.conf"

)

if (debugConfFile.exists()) {

LogUtils.d("$TAG disableX5 x32")

var inputStream: FileInputStream? = null

var outStream: FileOutputStream? = null

try {

inputStream = FileInputStream(debugConfFile)

outStream = FileOutputStream(debugConfFile)

val prop = Properties()

prop.load(inputStream)

prop.setProperty("setting\_forceUseSystemWebview", "true")

prop.setProperty("result\_systemWebviewForceUsed", "true")

prop.store(outStream, "update x5 core")

} catch (e: Exception) {

LogUtils.e(e.message.toString())

} finally {

inputStream?.close()

outStream?.close()

}

}

val debugConfFileX64 = File(

context.filesDir.path.substring(

0,

context.filesDir.path.lastIndexOf("/")

) + "/app\_tbs\_64/core\_private/debug.conf"

)

if (debugConfFileX64.exists()) {

LogUtils.d("$TAG disableX5 x64")

var inputStream: FileInputStream? = null

var outStream: FileOutputStream? = null

try {

inputStream = FileInputStream(debugConfFileX64)

outStream = FileOutputStream(debugConfFileX64)

val prop = Properties()

prop.load(inputStream)

prop.setProperty("setting\_forceUseSystemWebview", "true")

prop.setProperty("result\_systemWebviewForceUsed", "true")

prop.store(outStream, "update x5 core")

} catch (e: Exception) {

LogUtils.e(e.message.toString())

} finally {

inputStream?.close()

outStream?.close()

}

}

}

// 提前初始化 WebView

private fun initWebView(context: Context) {

val initStartTime = System.currentTimeMillis()

WebViewPool.init(context)

val initEndTime = System.currentTimeMillis()

LogUtils.d("$TAG Init WebView Cast: ${initEndTime - initStartTime}ms")

}

}

注意一下代码中的几个细节:

1. 初始化X5时，使用特殊的参数配置 TBS\_SETTINGS\_USE\_SPEEDY\_CLASSLOADER和 TBS\_SETTINGS\_USE\_DEXLOADER\_SERVICE，优化了X5的启动速度
2. 初始化X5时，传入disableX5这个bool值，让业务方可以控制是否使用X5内核
3. 初始化WebView对象时，使用了对象池，对WebView内核进行提前加载 WebViewPool.init(context), 它的作用主要是 提升打开H5时的加载速度。并且在后续使用webView对象时 只能从池子中去取。WebView的初始化其实也分为两种情况，一是 不带URL的，纯粹把内核提前加载，另外一个则是 带URL的，相当于预加载一个页面，提前加载H5的资源文件，在使用到的时候，再拿到这个WebView进行展示。对加载速度有一定提升。

WebViewPool 对象池参考代码

object WebViewPool {

private val TAG = this::class.java.simpleName

/\*\*

\* WebView 复用池

\*/

private var webViewPool: ArrayList<WebViewWrap> = arrayListOf()

/\*\*

\* WebView 预加载复用池

\*/

private var preWebViewPool: ArrayList<WebViewWrap> = arrayListOf()

/\*\*

\* webView 初始化

\* 最好放在application onCreate里

\*/

fun init(context: Context) {

buildPreWebView(context)

}

/\*\*

\* 获取webView

\*/

fun getWebView(activity: Activity, url: String): X5WebView {

val startTime = System.currentTimeMillis()

val wrap = checkWebView(activity, url)

if (wrap.webView.getInitUrl().isNotBlank()) {

// initUrl 不为空则代表预加载 URL 后的 WebView

webViewPool.add(wrap)

if (preWebViewPool.contains(wrap)) preWebViewPool.remove(wrap)

} else {

wrap.webView.doInit()

}

wrap.inUse = true

val contextWrapper = wrap.webView.context as MutableContextWrapper

contextWrapper.baseContext = activity

buildPreWebView(activity)

clearPrePool()

val endTime = System.currentTimeMillis()

LogUtils.d("$TAG Get WebView Cast:${endTime - startTime}ms")

return wrap.webView

}

/\*\*

\* 回收webView

\*/

fun recycleWebView(webView: X5WebView) {

webViewPool.forEach {

if (it.webView == webView && it.inUse) {

val contextWrapper = webView.context as MutableContextWrapper

contextWrapper.baseContext = webView.context.applicationContext

webView.release()

it.inUse = false

LogUtils.d("$TAG recycleWebView $it")

}

}

clearPool()

}

/\*\*

\* 清理预加载 WebViewPool

\*/

private fun clearPrePool() {

val noUseList = preWebViewPool.filter { !it.inUse }

if (noUseList.size > 1) {

val waitRemoveList = noUseList.subList(0, noUseList.size - 1)

waitRemoveList.forEach {

it.webView.removeAllViews()

it.webView.destroy()

}

preWebViewPool.removeAll(waitRemoveList.toSet())

System.gc()

LogUtils.d("$TAG clearPrePool $preWebViewPool")

}

}

/\*\*

\* 清理 WebViewPool

\*/

private fun clearPool() {

val noUseList = webViewPool.filter { !it.inUse }

if (noUseList.size > 1) {

val waitRemoveList = noUseList.subList(0, noUseList.size - 1)

waitRemoveList.forEach {

it.webView.removeAllViews()

it.webView.destroy()

}

webViewPool.removeAll(waitRemoveList.toSet())

System.gc()

LogUtils.d("$TAG clearPool $webViewPool")

}

}

/\*\*

\* 预热webView

\*/

private fun buildPreWebView(context: Context) {

Looper.myQueue().addIdleHandler {

val startTime = System.currentTimeMillis()

val webView = X5WebView(MutableContextWrapper(context.applicationContext))

webView.loadEmpty()

val wrap = WebViewWrap(webView, false)

webViewPool.add(wrap)

val endTime = System.currentTimeMillis()

LogUtils.d("$TAG buildPreWebView end cast:${endTime - startTime} webView:$wrap")

false

}

}

/\*\*

\* 创建带 url 的 webView

\*/

fun buildPreUrlWebView(context: Context, url: String) {

if (preWebViewPool.any { it.webView.getInitUrl() == url }) return

if (preWebViewPool.size > 1) {

val waitRemoveList = preWebViewPool.subList(0, preWebViewPool.size - 1)

preWebViewPool.removeAll(waitRemoveList.toSet())

}

val webView = X5WebView(MutableContextWrapper(context.applicationContext))

webView.doInit()

webView.loadUrl(url)

val wrap = WebViewWrap(webView, false)

preWebViewPool.add(wrap)

LogUtils.d("$TAG buildPreUrlWebView $wrap")

}

/\*\*

\* 创建webView

\*/

private fun buildWebView(context: Context): WebViewWrap {

val webView = X5WebView(MutableContextWrapper(context.applicationContext))

val wrap = WebViewWrap(webView, false)

webViewPool.add(wrap)

LogUtils.d("$TAG buildWebView $wrap")

return wrap

}

/\*\*

\* 检测 webView

\*/

private fun checkWebView(context: Context, url: String): WebViewWrap {

LogUtils.d("$TAG checkWebView url:$url")

preWebViewPool.reversed().forEach {

LogUtils.d("$TAG PrePool item:${it.webView.getInitUrl()}")

if (it.webView.getInitUrl() == url) {

LogUtils.d("$TAG Find WebView In PrePool $it")

return it

}

}

webViewPool.reversed().forEach {

if (!it.inUse) {

LogUtils.d("$TAG Find WebView In Pool $it")

return it

}

}

LogUtils.d("$TAG Not Find WebView In Pool")

return buildWebView(context)

}

}

class WebViewWrap(var webView: X5WebView, var inUse: Boolean)

## WebView的引入和常用设置

X5的引入过程，在腾讯官网有，这里不再赘述。

下面是一些细节：

setWebContentsDebuggingEnabled 此函数的作用，是允许在发布包中打开X5的调试模式。

通常X5内核调试H5页面，都是在debug模式运行时进行的，具体的方式是用X5WebView打开 [debugX5.qq,com](https://link.juejin.cn?target=http://debugX5.qq,com" \o "http://debugX5.qq,com" \t "/Users/zpw/Documents\\x/_blank) 进行一系列设置，然后就能在PC端看到H5页面的具体运行参数，包括网络，元素等。 将次函数的入参设置为 true，则可以在发布包中拥有相同的效果，不过此举有一定的风险，有可能导致关键信息泄露。所以如果是C端应用，建议采用特殊的方式控制此设置的开启关闭。

WebView是耗内存的大户，如果使用不当，内存泄露，应用会有明显的卡顿。在destory回调中，必须对使用到的资源进行释放。

class X5WebView : WebView {

// 设置初始化

init {

settings.let {

it.domStorageEnabled = true

it.allowFileAccess = true

it.setAppCacheEnabled(true)

it.databaseEnabled = true

it.domStorageEnabled = true

it.javaScriptEnabled = true

it.setAppCachePath(appCacheDirName)

it.useWideViewPort = true

it.setSupportZoom(false)

it.loadWithOverviewMode = true

it.textZoom = 100

CookieManager.getInstance().setAcceptThirdPartyCookies(this, true) // 设置允许接受第三方cookie

it.mixedContentMode = WebSettings.MIXED\_CONTENT\_ALWAYS\_ALLOW

}

setWebContentsDebuggingEnabled(true) // 设置允许在发布包内打开X5的调试模式

}

/\*\*

\* 业务参数初始化

\*/

fun doInit() {

val startTime = System.currentTimeMillis()

webChromeClient = mWebChromeClient // 自定义 WebChromeClient

webViewClient = mWebClient // 自定义 mWebClient

addJavascriptInterface(innerJavascriptInterface, BRIDGE\_NAME)

addJavascriptInterface(true, API\_FLAG)

clearHistory()

enableOfflinePackage = true

LogUtils.d("$TAG doInit cast:${System.currentTimeMillis() - startTime}")

}

fun release() {

loadEmpty()

javaScriptNamespaceInterfaces.clear()

removeJavascriptInterface(API\_FLAG)

removeJavascriptInterface(BRIDGE\_NAME)

webChromeClient = null

webViewClient = null

onLoadListener = null

callInfoList?.clear()

clearCache(false)

clearHistory()

if (parent != null) {

(parent as ViewGroup).removeView(this)

}

}

override fun destroy() {

LogUtils.e("$TAG destroy")

release()

super.destroy()

}

}

webView与H5的交互流程与SDK框架设计

H5与native的交互，都是通过webView作为媒介, 通常的方式，就是 利用 addJavascriptInterface 函数建立一个通信通道:

addJavascriptInterface(bridgeObj, 'bridge\_name')

bridgeObj对象中，能够被 js调用到的函数，都必须打上 @JavascriptInterface 标记，同时为了防止被混淆 ，也要加上 @Keep .

internal inner class BridgeObj {

/\*\*

\* 所有的js方法入口会进入call

\*/

@Keep

@JavascriptInterface

fun call(methodName: String, argStr: String): String {

return ""

}

}

在此配置之下，js可以通过代码：window.bridge\_name.call() 来调用到下面的call方法。

接下来就是设计的重点。当我们设计一套两端的通信协议时，要考虑的首先就是易用性，标准化，webView容器设计出来是要给众多业务方使用，先制定简单可行的标准流程, 可以增加以后工作的遍历。然后是可扩展性，保证业务的迭代过程中，我们开发人员自身的开发维护体验，不能让业务堆积起来让代码的后续维护困难重重。最后考虑的是稳定性，每一个原生能力应该相互独立，一方代码万一出现问题，将影响最小化。

达成这些目的，需要进行科学的代码框架设计。我们的思路如下：

js-native调用入口统一

有且仅有一个js-native的访问入口，也就是上述名为：bridge\_name 的native变量, 并且仅有一个 call 函数，作为调用的入口。其他特殊的参数，统一由 js调用时传入的 实参决定。比如，下面这种js-native的调用方式：

window.bridge\_name.call({'methodName':'XXXApi.getXXX','argStr':{'data1':''，'data1':''}}})

前面的 window.bridge\_name.call 始终保持一致，同时为了易用性，并且简化业务方的调用代码，还需对上面的调用方式进行二次封装。

命名空间分层 实现native接口分组隔离

注意观察上面这一句js代码，严格划分命名空间的话，会发现有三层，

1. bridge\_name（js-native交互的对象名），
2. methodName 的value前半部分 XXXApi
3. methodName 的value后半部分 getXXX

三层空间都解析出来之后，才能最终确定调用了 哪一个native方法。第一层是为了入口统一，那么后面两层则是为了业务隔离。对比一下，如果后面两层合一，调用方式变为：window.bridge\_name.call({'methodName':'getXXX','argStr':{'data1':''，'data1':''}}}), 那么所有的 native方法将会挤在一个文件中，随着业务的膨胀，native接口越来越多的话，维护难度会越来越大。增加第二层命名空间，对native接口进行分组隔离管理，各组互不干扰。

native解析命名空间分发执行api

在命名空间分层的基础之上，native接收到 第一层 XXXApi，与第二层 getXXX 之后，将第一层解析为 类名，第二层解析为 函数名。而native层的api代码 XXXApi 为一个整体类，内部包含多个 getXXX 文件。

class XXXApi {

@JavascriptInterface

fun getXXX1(callBack: CallBack<Map<String, Any>>) {

val resp = hashMapOf<String, Any>()

resp["brand"] = Build.BRAND

callBack.complete(ResultHelper.success(resp))

}

@JavascriptInterface

fun getXXX2(callBack: CallBack<Unit>) {

callBack.complete(ResultHelper.success(Unit))

}

}

具体的调用方式为，提前将上面提取出来的 XXXAPI 与 真实的全类名做**一个映射**，通过XXXAPI找到全类名，反射取得该对象，并且执行该对象的 getXXX 方法。

统筹执行 同步函数和异步函数

js调用native过程，有可能是能够立即获取结果的同步函数，也有可能是需要跳转某个新页面，经过处理之后才能拿到结果的异步函数。将两种流程统一按照 异步回调的方式 将执行结果通知js，可以极大的简化处理过程。

以网络请求为例，如果js想借助native来执行网络请求，并且拿到执行的结果，那么必然是异步过程。

那么在反射执行的时候，先将这个 回调函数对象 创建出来，并且在执行反射方法时，设置成其中一个参数.

以下是参考代码，request方法的第一个参数params 是 原来js传过来的参数，第二个callBack则是 反射执行时创建的 回调对象。

class XXXNetworkApi {

// 同步过程

@JavascriptInterface

fun getNetworkType(params: JsonObject, callBack: CallBack<Any>) {

val type = when (NetworkUtils.getNetworkType()) {

NetworkUtils.NetworkType.NETWORK\_NO -> "none"

NetworkUtils.NetworkType.NETWORK\_2G,

NetworkUtils.NetworkType.NETWORK\_3G,

NetworkUtils.NetworkType.NETWORK\_4G,

NetworkUtils.NetworkType.NETWORK\_5G -> "cellular"

NetworkUtils.NetworkType.NETWORK\_WIFI -> "wifi"

else -> "unknown"

}

val resp = hashMapOf<String, Any>()

resp["type"] = type

callBack.complete(ResultHelper.success(resp))

}

// 异步过程

@JavascriptInterface

fun request(params: JsonObject, callBack: CallBack<Any>) {

val url = params.get("url")?.asString

val method = params.get("method")?.asString ?: "POST"

val headers = params.get("headers")?.asJsonObject

val requestParams = params.get("params")?.asJsonObject

val timeout = params.get("timeout")?.asInt ?: 30

if (url.isNullOrEmpty()) {

callBack.complete(ResultHelper.fail(msg = "调用失败，url 为空"))

return

}

val requestHeaders = headers.toString().toJsonObject() ?: mapOf<String, String>()

if (method == "Get") {

HttpUtils.request("GET", requestHeaders,

JSONObject(requestParams.toString()),

url,

timeout,

object : Callback {

override fun onFailure(call: Call, e: IOException) {

callBack.complete(ResultHelper.fail("请求失败：$e"))

}

override fun onResponse(call: Call, response: Response) {

callBack.complete(

ResultHelper.success(

(response.body()?.string() ?: "").toJsonObject()

)

)

}

})

} else {

HttpUtils.request(

"POST",

requestHeaders,

JSONObject(requestParams.toString()),

url,

timeout,

object : Callback {

override fun onFailure(call: Call, e: IOException) {

callBack.complete(ResultHelper.fail("请求失败：$e"))

}

override fun onResponse(call: Call, response: Response) {

callBack.complete(

ResultHelper.success(

(response.body()?.string() ?: "").toJsonObject()

)

)

}

})

}

}

}

上面的伪代码中，给出了一个同步过程，一个异步过程的，两者都是在执行完毕之后，直接调用了 callback对象的 complete 方法来告知， complete 内部执行的则是 native调用js。

按module 维护多个API实例

通常，做一个webView容器，native方法经过上面第2节的分组之后，可以在一个module之内完成所有的代码。但是考虑到两个问题，其一，某一些native的调用过程会引用到体积比较大的第三方SDK，如果强行引入的话，对于不需要用到该SDK的业务方，是一个包体积的不必要的扩大。其二，多种业务挤压在一起，造成module会无线膨胀，容易发生耦合，管理困难。

我们采取的方式是，

抽离native api的特征，提取成接口，接口下沉，放置在一个module中，

每一个具体的业务module，都依赖这个下沉的module，来编辑自己的业务module，并且每一个module中api类，打上 @AutoService 标记

在webView初始化时，利用ServiceLoader类（android自带，无需引入依赖）, 抓取运行时所有标记了 注解@AutoService的class，这样便能将所有的api的class都保存到 第三节提到的映射中。

在所有 xxxApi module都并行独立之后，我们依赖多个 module的方式时如下写法:

dependencies {

implementation 'androidx.camera:camera-camera2:1.0.0-rc05'

// 框架层

implementation project(':baseApiModule')

// 模块化业务api

implementation project(':xxxApiModule1')

implementation project(':xxxApiModule2')

implementation project(':xxxApiModule3')

implementation project(':xxxApiModule4')

implementation project(':xxxApiModule1...')

}

我们可以根据每个业务方的实际需要来引入不同的业务module，而不是一股脑全依赖进去。

webView容器的容灾方案设计

容器正常运行时自然皆大欢喜，但是出现问题，首先应该做的就是业务方自查，我们容器SDK开发人员最好是给业务方一个明确的排查方案，这是为了业务方的体验，也是为了我们自己的工作体验（老板舒服了，我们才不会被喷）。

在我们的实践过程中，发现了一些坑，这里把解决方案列出来：

WebViewClient中有 一个 shouldInterceptRequest 函数 支持使用离线资源，配合我们自建的离线包更新机制，可以极大的加快H5的加载速度，但是，常规出现了诡异的问题，常规的H5访问线上资源能够正常，但是使用离线js资源则会出现网络请求跨域的问题，我们可以在 shouldInterceptRequest的return的WebResourceResponse中加入 跨域配置来避免此类问题。

伪代码如下:

override fun shouldInterceptRequest(

webView: WebView?,

request: WebResourceRequest

): WebResourceResponse? {

val exs = OfflinePackage.findOfflineResource(request.url.toString()) // 走离线包逻辑

// ....

return WebResourceResponse(

exs.mimeType,

exs.encoding,

200,

"ok",

addCorsHeader(hashMapOf()),

FileInputStream(exs.resourcePath)

)

}

// 添加跨域参数允许跨域

private fun addCorsHeader(originHeader: HashMap<String, String>): HashMap<String, String> {

originHeader["Access-Control-Allow-Origin"] = "\*"

originHeader["Access-Control-Allow-Headers"] = "\*"

originHeader["Access-Control-Allow-Credentials"] = "true"

return originHeader

}

WebViewClient中的另一个方法 onRenderProcessGone 处理一个WebView对象的渲染程序消失的情况，要么是因为系统杀死了渲染器以回收急需的内存，要么是因为渲染程序本身崩溃了，通过使用这个API，可以让您的应用程序继续执行，即使渲染过程已经消失了。参考代码如下

override fun onRenderProcessGone(view: WebView?, detail: RenderProcessGoneDetail): Boolean {

LogUtils.d("$TAG onRenderProcessGone start")

if (Build.VERSION.SDK\_INT < Build.VERSION\_CODES.O) return false

super.onRenderProcessGone(view, detail)

if (!detail.didCrash()) {

LogUtils.d("$TAG onRenderProcessGone did no crash")

if (context != null) {

if (context is Activity) {

(context as Activity).finish()

}

if (context is MutableContextWrapper && (context as MutableContextWrapper).baseContext is Activity) {

((context as MutableContextWrapper).baseContext as Activity).finish()

}

}

return true

}

LogUtils.d("$TAG onRenderProcessGone did crash")

return false

}

H5的加载过程中有时候会由于网络问题等原因 出现白屏的情况，我们必须制定一种白屏监测机制来优化这种异常体验。

白瓶检测工具的参考代码如下：

基本原理，利用webView自身的截图函数 snap，取得当前的截图bitmap，然后逐个监测像素点，如果白点数量超过了一定比例，则认定是白屏。

/\*\*\*

\* WebView白屏监测工具

\*/

object BlankCheckUtil {

private val TAG = this.javaClass.simpleName

private var config = BlankCheckConfig()

private var timer: Timer? = null

/\*\*

\* 设置白屏检测配置

\*/

fun setConfig(config: BlankCheckConfig) {

if (config.checkRate < 0 || config.checkRate > 100) {

throw Throwable(message = "checkRate range is 0 - 100")

}

if (config.scaleRatio < 0 || config.scaleRatio > 100) {

throw Throwable(message = "scaleRatio range is 0 - 100")

}

this.config = config

}

/\*\*

\* 开始检测

\*/

fun start(webView: X5WebView, callback: (isBlank: Boolean) -> Unit) {

LogUtils.d("$TAG Start WebView:$webView config:$config")

// 延迟500ms执行

timer?.cancel()

timer = Timer()

timer?.schedule(BlankCheckTask(webView, callback), 500)

}

private class BlankCheckTask(

private val webView: X5WebView,

private val callback: (isBlank: Boolean) -> Unit

) : TimerTask() {

override fun run() {

LogUtils.d("$TAG BlankCheckTask WebView:$webView}")

webView.post {

val baseContext = if (webView.context is MutableContextWrapper) {

(webView.context as MutableContextWrapper).baseContext

} else {

webView.context

}

if (baseContext is Activity && !baseContext.isDestroyed && !baseContext.isFinishing) {

val startTime = System.currentTimeMillis()

val bitmap =

webView.snapShot(config.scaleRatio, Bitmap.Config.RGB\_565) ?: return@post

val isBlank = check(bitmap)

bitmap.recycle()

callback.invoke(isBlank)

val endTime = System.currentTimeMillis()

LogUtils.d("$TAG BlankCheckTask Check End IsBlank:$isBlank Cast${endTime - startTime}ms WebView:$webView")

}

}

}

}

/\*\*

\* 停止检测

\*/

fun stop() {

timer?.cancel()

}

/\*\*

\* 检测

\*/

private fun check(bitmap: Bitmap): Boolean {

LogUtils.d("$TAG Check")

//白点计数

var whitePixelCount = 0f

val width = bitmap.width

val height = bitmap.height

for (x in 0 until width) {

for (y in 0 until height) {

if (bitmap.getPixel(x, y) == -1) {

//表示是白色

whitePixelCount++

}

}

}

LogUtils.d("$TAG width:$width height:$height whitePixelCount:$whitePixelCount")

val rate = whitePixelCount / (width \* height) \* 100

//这里可以对比设定的上限，然后做处理

LogUtils.d("$TAG Check End White Rate:$rate%")

return rate > config.checkRate

}

}

/\*\*

\* @param scaleRatio 截图缩放比例 (默认10%，取值范围 0-100)

\* @param checkRate 白色像素点检测比例 (默认99.9%，取值范围 0-100)

\*/

class BlankCheckConfig(val scaleRatio: Int = 10, val checkRate: Double = 99.9) {

override fun toString(): String {

return "BlankCheckConfig:scaleRatio=$scaleRatio, checkRate=$checkRate"

}

}

监测的时机，通常放在 onLoadFinish时，如果加载进度超过了一定的值（99%），则执行白屏监测，如果监测结果是认定为白屏，则执行刷新逻辑 reload.

WebViewClient中有 一个 onReceivedError 可以侦测出大部分的webView相关的异常，为了减轻后期业务方自己犯错反而来过问我们的麻烦，我们选择把这个函数重写，并且在出现此类问题时，将报错信息直接回传给H5，让他们能够先自查。

一些原生能力需要用到系统权限，比如 相册相机等。这些申请的动作，一定给一个权限申请结果的回调函数给业务方，我们当时要进行隐私合规的整改，权限的申请结果都需要上报到后台，如果在SDK内部，就无法保证业务方统一上报了。

# 总结

设计一套WebView容器SDK需要对androidWebView的基本配置有了解，X5基本上在api层面没有大改，改动的只是内核，开发人员基本感知不到。上面的思路，基本上涵盖了一个容器SDK从0到1的所有过程，能够在保证功能的同时让代码尽可能保持优雅。