**WebView预加载优化**

在合适的时机 预加载 WebView 并存入 缓存池 中，等要用到时再直接从缓存池中取，从而缩短显示首屏页面的时间。

object WebViewCacheHolder {

private val webViewCacheStack = Stack<RobustWebView>()

private const val CACHED\_WEB\_VIEW\_MAX\_NUM = 4

private lateinit var application: Application

fun init(application: Application) {

this.application = application

prepareWebView()

}

fun prepareWebView() {

if (webViewCacheStack.size < CACHED\_WEB\_VIEW\_MAX\_NUM) {

Looper.myQueue().addIdleHandler {

log("WebViewCacheStack Size: " + webViewCacheStack.size)

if (webViewCacheStack.size < CACHED\_WEB\_VIEW\_MAX\_NUM) {

webViewCacheStack.push(createWebView(MutableContextWrapper(application)))

}

false

}

}

}

fun acquireWebViewInternal(context: Context): RobustWebView {

if (webViewCacheStack.isEmpty()) {

return createWebView(context)

}

val webView = webViewCacheStack.pop()

val contextWrapper = webView.context as MutableContextWrapper

contextWrapper.baseContext = context

return webView

}

private fun createWebView(context: Context): RobustWebView {

return RobustWebView(context)

}

}

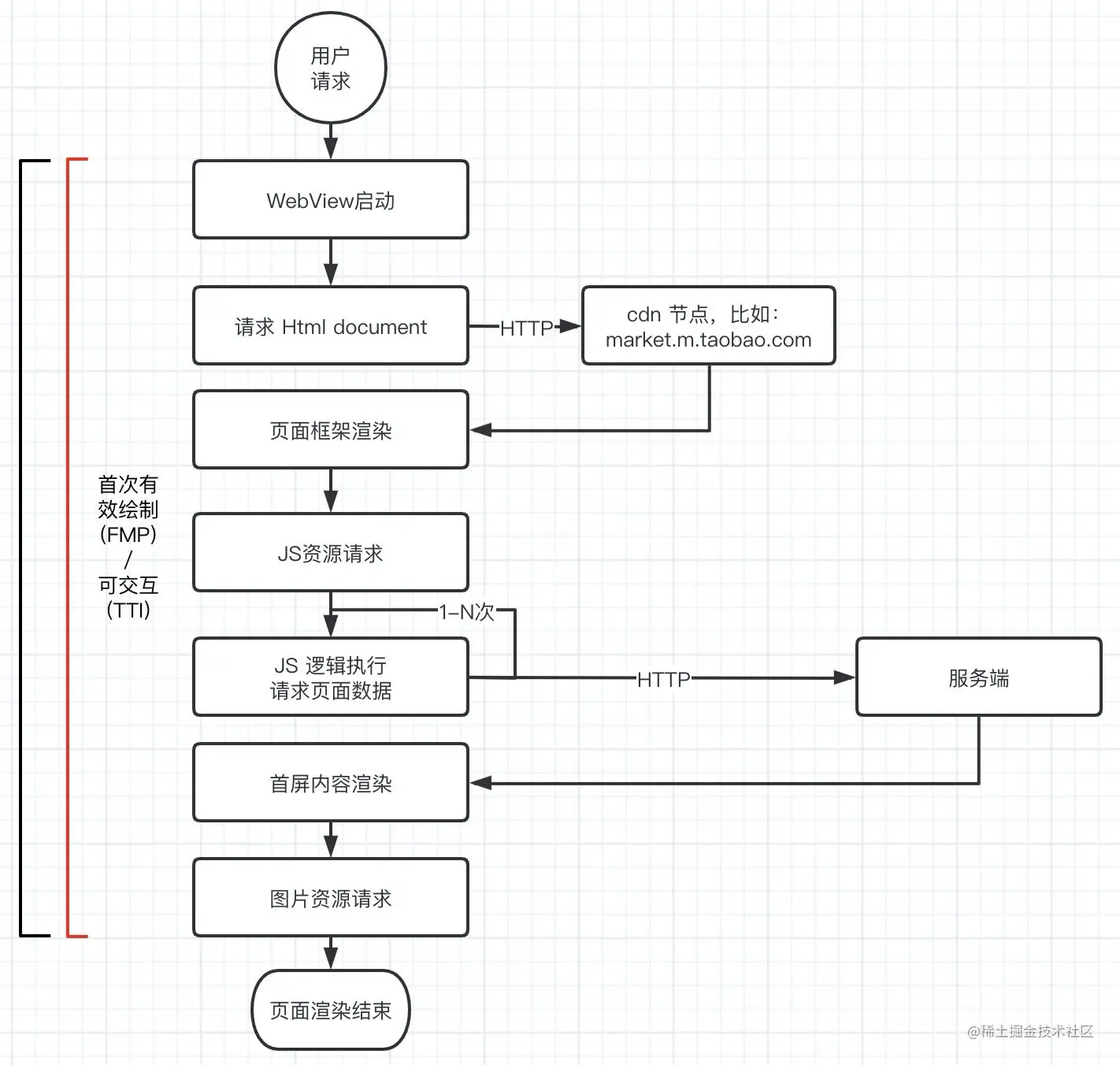
知识点：

1、通过 IdleHandler 提交的任务只有在当前线程关联的 MessageQueue 为空的情况下才会被执行，因此通过 IdleHandler 来执行预创建可以保证不会影响到当前主线程任务。

2、MutableContextWrapper 是系统提供的 Context 包装类，其内部包含一个 baseContext，MutableContextWrapper 所有的内部方法都会交由 baseContext 来实现，且 MutableContextWrapper 允许外部替换它的 baseContext，因此我们可以在一开始的时候使用 Application 作为 baseContext，等到 WebView 和 Activity 进行实际绑定的时候再来替换。

需要注意，对 WebView 进行缓存采取的是用空间换时间的做法，需要考虑低端机型运存较小的情况。

**渲染优化**



这一整个过程需要完成多个网络请求和 IO 操作，WebView 在加载了基本的 HTML 和 CSS 文件后，再通过 JS 从服务端获取正文数据，拿到数据后还需要完成解析 JSON、构造 DOM、应用 CSS 样式等一系列耗时操作，最终才能由内核进行渲染上屏。

移动端的系统版本、处理器速度、运存大小是完全不受我们控制的，且极容易受网络波动的影响，网络链接的耗时是非常长且不可控的。如果 WebView 每次渲染都重复经历以上整个步骤，那用户的使用体验就是完全不可控的，此时可以尝试通过以下方法来进行优化。

**预置离线包**

精简并抽取公共的 JS 和 CSS 文件作为通用的页面模板，可以按业务类型来生成多套模板文件，每次打包时均预置最新的模板文件到客户端中，每套模板文件均有特定的版本号，App 在后台定时去静默更新。通过这种方式来避免每次使用都要去联网请求，从而缩短总耗时。

一般情况下，WebView 会在加载完主 HTML 之后才去加载 HTML 中的 JS 和 CSS 文件，先后需要进行多次 IO 操作，我们可以将 JS 和 CSS 还有一些图片都内联到一个文件中，这样加载模板时就只需要一次 IO 操作，也大大减少了因为 IO 加载冲突导致模板加载失败的问题。

**并行请求**

H5 在加载模板文件的同时，由 Native 端来请求正文数据，Native 端再通过 JS 将正文数据传给 H5，以此来实现并行请求从而缩短总耗时。

**预加载**

当模板和正文数据分离之后，由于 WebView 每次使用的都是同一个模板文件，因此我们并不需要在用户进入页面的时候才去加载模板，可以直接在预加载 WebView 的同时就让其预热加载模板，这样每次使用时仅需要将正文数据传给 H5，H5 收到数据后直接进行页面渲染即可。

对于 Feed 流，可以通过一定策略去预加载正文数据，当用户点击查看详情时，最理想情况下就可以直接使用缓存的数据，避免受到网络的影响。

**延迟加载**

呈现首屏页面所需要的依赖项越多，就意味着用户需要的等待时间就越长，因此要尽可能地减少在首屏完成前执行的操作，对于一些非首屏必需的网络请求、 JS 调用、埋点上报等，都可以后置到首屏显示后再执行。

**页面静态直出**

并行请求正文数据虽然能够缩短总耗时，但还是需要完成解析 JSON、构造 DOM、应用 CSS 样式等一系列耗时操作，最终才能交由内核进行渲染上屏，此时 组装 HTML这个操作就显得比较耗时了。为了进一步缩短总耗时，可以改为由后端对正文数据和前端代码进行整合，直出首屏内容，直出后的 HTML 文件已经包含了首屏展现所需的内容和样式，无需进行二次加工，内核可以直接渲染。其它动态内容可以在渲染完首屏后再进行异步加载。

由于客户端可能向用户提供了控制 WebView 字体大小，夜间模式的选项，为了保证首屏渲染结果的准确性，服务端直出的 HTML 就需要预留一些占位符用于后续动态回填，客户端在 loadUrl 之前先利用正则匹配的方式查找这些占位字符，按照协议映射成端信息。经过客户端回填处理后的 HTML 内容就已经具备了展现首屏的所有条件。

**复用 WebView**

更进一步的做法就是可以尝试复用 WebView。由于 WebView 使用的模板文件已经是固定的了，因此我们可以在 WebView 预加载缓存池的基础上增加复用 WebView 的逻辑，当 WebView 使用完毕后可以将其正文数据全部清空并再次存入缓存池中，等下次需要时就可以直接注入新的正文数据进行复用了，从而减少了频繁创建 WebView 和预热模板文件带来的开销。

**视觉优化**

实现以上的优化方案后，页面的展现速度已经很快了，但在实际开发中还是会发现存在 Activity 切换过程中无法渲染 H5 页面的问题，产生视觉上的白屏现象，这可以通过开发者模式放慢动画时间来验证。

通过研究系统源码可以知道，在系统版本大于等于 4.3，小于等于 6.0 之间，ViewRootImpl 在处理 View 绘制的时候，会通过一个布尔变量 mDrawDuringWindowsAnimating 来控制 Window 在执行动画的过程中是否允许进行绘制，该字段默认为 false，我们可以利用反射的方式去手动修改这个属性，避免这个白屏效果。

这个方案基本也只适用于 Android 6.0 版本了，更低的系统版本也很少进行适配了。

/\*\*

\* 让 activity transition 动画过程中可以正常渲染页面

\*/

fun setDrawDuringWindowsAnimating(view: View) {

if (Build.VERSION.SDK\_INT > Build.VERSION\_CODES.M

|| Build.VERSION.SDK\_INT < Build.VERSION\_CODES.JELLY\_BEAN\_MR1

) {

//小于 4.3 和大于 6.0 时不存在此问题，无须处理

return

}

try {

val rootParent: ViewParent = view.rootView.parent

val method: Method = rootParent.javaClass

.getDeclaredMethod("setDrawDuringWindowsAnimating", Boolean::class.javaPrimitiveType)

method.isAccessible = true

method.invoke(rootParent, true)

} catch (e: Throwable) {

e.printStackTrace()

}

}

**Http 缓存策略**

在上一步的**渲染优化**中就涉及到了对网络请求的优化，包括 **减少网络请求次数、并行执行网络请求、网络请求预执行** 等。对于应用来说，网络请求是不可避免的，但我们可以通过设定**缓存策略**来避免重复执行网络请求，或者是可以用比较低的成本来完成非首次的网络请求，这就涉及到了和 Http 缓存相关的知识点。

WebView 一共支持以下四种缓存策略，默认使用的是 LOAD\_DEFAULT，该策略就属于 Http 缓存策略

LOAD\_CACHE\_ONLY：只使用本地缓存，不进行网络请求

LOAD\_NO\_CACHE：不使用本地缓存，只通过网络请求

LOAD\_CACHE\_ELSE\_NETWORK：只要本地有缓存就进行使用，否则就通过网络请求

LOAD\_DEFAULT：根据 Http 协议来决定是否进行网络请求

以请求网络上一个静态文件为例，查看其响应头，当中的 Cache-Control、Expires、Etag、Last-Modified 等信息就定义了具体的缓存策略。

**Cache-Control、Expires**

Cache-Control 是 Http 1.1 中新增加的一个用来定义资源缓存策略的报文头，它由一些定义一个响应资源应该何时被缓存、如何被缓存以及缓存多长时间的指令组成，可选值有很多种：no-cache、no-store、only-if-cached、max-age 等。

Expires 是 Http 1.0 中规定的字段，含义和 Cache-Control 类似，但由于 Expires 可能会因为客户端和服务端的时间不一致造成缓存失效，因此现在主要使用的是 Cache-Control，在优先级上也是 Cache-Control 更高。

Cache-Control 也是一个通用的 Http 报文头字段，它可以分别在请求头和响应头中使用，具有不同的含义，以 max-age 为例：

请求头：客户端用于告知服务端，希望接收一个有效期不大于 max-age 的资源。

响应头：服务端用于告知客户端，该资源在请求发起后的 max-age 时间内均是有效的，可以直接使用本地缓存。

如果在 WebView 中使用了 LOAD\_DEFAULT 的话，就会遵循此 Http 缓存策略，在有效期内 WebView 会直接使用本地缓存。

**ETag、Last-Modified**

Cache-Control 避免了 WebView 在有效期内去重复请求资源，有效期过了后 WebView 就还是需要重新去请求网络，但此时服务端的资源也许并没有发生变化，WebView 依然可以使用本地缓存，此时客户端就需要依靠 ETag 和 Last-Modified 这两个报文头来向服务器确认该资源是否可以继续使用。

在第一次请求资源的时候，响应头中就包含了 ETag 和 Last-Modified，这两个报文头就用来唯一标识该资源文件。

ETag：用于作为资源的唯一标识信息

Last-Modified：用于记录资源的最后一次修改时间

等客户端判断到 max-age 已过期后，就会携带这两个报文头去执行网络请求，服务端就通过这两个标识符来判断客户端的缓存资源是否可以继续使用。

在有效期过后，客户端会在 If-None-Match 请求头中携带上第一次网络请求时拿到的 ETag 值。实际上 ETag 和 Last-Modified 可以只使用一个，以下就只使用到了 ETag；如果要传递 Last-Modified 的话，对应的请求头就是 If-Modified-Since。

如果服务端判断出资源已过期，就会返回新的资源文件，此时就相当于在第一次请求资源文件，后续操作就和一开始保持一致；如果服务端判断资源还未过期，则会返回一个 304 状态码，告知客户端可以继续使用本地缓存，客户端同时更新 max-age 值，重复一开始的的缓存失效规则，这样客户端就可以用极低的成本来完成本次网络请求，这在请求的资源文件比较大的时候特别有用。

但 Http 缓存策略也存在一些问题需要注意，即如何保证**用户在资源更新了时能马上感知到且重新下载最新资源**。假设服务端在资源有效期内更新了资源内容，此时由于客户端还处于 max-age 阶段，无法马上感知到资源已更新，从而造成更新不及时。一种比较好的解决方案就是：要求服务端在每次更新资源文件时都为其生成一个新的名字，可以用 hash 值或者随机数命名，而资源文件依托的主文件在每次发版时都引用最新的资源文件路径，从而保证客户端能够马上就感知到资源已更新，从而保证及时更新。而且，通过这种方案，既可以为资源文件设定一个非常大的 max-age 值，尽量让客户端只使用本地缓存，又可以保证每次发版时客户端都能及时更新。

所以说，通过合理地设定 Http 缓存策略，一方面能够很明显地减少服务器网络带宽消耗、降低服务器的压力和开销，另一方面也可以减少客户端网络延迟的情况、避免重复请求资源文件、加快页面的打开速度，毕竟加载本地缓存文件的开销怎样都要比从网络上加载低得多。

**拦截请求与共享缓存**

如今的 WebView 页面往往是图文混排的，图片是资讯类应用的重要表现形式，WebView 获取图片资源的传统方案有以下两种：

H5 端自己通过网络请求去下载资源。优点：实现简单，各端之间可以只专注自己的业务。缺点：两端之间的无法共享缓存，造成资源重复请求，流量浪费

H5 端通过调用 Native 的图片下载和缓存能力来获取资源。优点：可以实现两端之间的缓存共享。缺点：需要由 H5 端来主动触发 Native 执行，时机较为延迟，且需要通过多次 JS 调用完成资源传递，存在效率问题

以上两种方案都存在着一些缺点，要么是无法共享缓存，要么是存在效率问题，这里就再介绍一种改进方案：

实际上，WebViewClient 提供了一个 shouldInterceptRequest 方法用于支持外部去拦截请求，WebView 每次在请求网络资源时都会回调该方法，方法入参就包含了 Url，Header 等请求参数，返回值 WebResourceResponse 即代表获取到的资源对象，默认是返回 null，即由浏览器内核自己去完成网络请求。

我们可以通过该方法来主动拦截并完成图片的加载操作，这样我们既可以使得两端的资源文件得以共享，也避免了多次 JS 调用带来的效率问题。

大致实现就如下所示，这里通过 OkHttp 来代理实现网络请求。

object WebViewInterceptRequestProxy {

private lateinit var application: Application

private val webViewResourceCacheDir by lazy {

File(application.cacheDir, "RobustWebView")

}

private val okHttpClient by lazy {

OkHttpClient.Builder().cache(Cache(webViewResourceCacheDir, 100L \* 1024 \* 1024))

.followRedirects(false)

.followSslRedirects(false)

.addNetworkInterceptor(

ChuckerInterceptor.Builder(application)

.collector(ChuckerCollector(application))

.maxContentLength(250000L)

.alwaysReadResponseBody(true)

.build()

)

.build()

}

fun init(application: Application) {

this.application = application

}

fun shouldInterceptRequest(webResourceRequest: WebResourceRequest?): WebResourceResponse? {

if (webResourceRequest == null || webResourceRequest.isForMainFrame) {

return null

}

val url = webResourceRequest.url ?: return null

if (isHttpUrl(url)) {

return getHttpResource(url.toString(), webResourceRequest)

}

return null

}

private fun isHttpUrl(url: Uri): Boolean {

val scheme = url.scheme

log("url: $url")

log("scheme: $scheme")

if (scheme == "http" || scheme == "https") {

return true

}

return false

}

private fun getHttpResource(

url: String,

webResourceRequest: WebResourceRequest

): WebResourceResponse? {

val method = webResourceRequest.method

if (method.equals("GET", true)) {

try {

val requestBuilder =

Request.Builder().url(url).method(webResourceRequest.method, null)

val requestHeaders = webResourceRequest.requestHeaders

if (!requestHeaders.isNullOrEmpty()) {

var requestHeadersLog = ""

requestHeaders.forEach {

requestBuilder.addHeader(it.key, it.value)

requestHeadersLog = it.key + " : " + it.value + "\n" + requestHeadersLog

}

log("requestHeaders: $requestHeadersLog")

}

val response = okHttpClient.newCall(requestBuilder.build())

.execute()

val body = response.body

if (body != null) {

val mimeType = response.header(

"content-type", body.contentType()?.type

).apply {

log(this)

}

val encoding = response.header(

"content-encoding",

"utf-8"

).apply {

log(this)

}

val responseHeaders = mutableMapOf<String, String>()

var responseHeadersLog = ""

for (header in response.headers) {

responseHeaders[header.first] = header.second

responseHeadersLog =

header.first + " : " + header.second + "\n" + responseHeadersLog

}

log("responseHeadersLog: $responseHeadersLog")

var message = response.message

val code = response.code

if (code == 200 && message.isBlank()) {

message = "OK"

}

val resourceResponse =

WebResourceResponse(mimeType, encoding, body.byteStream())

resourceResponse.responseHeaders = responseHeaders

resourceResponse.setStatusCodeAndReasonPhrase(code, message)

return resourceResponse

}

} catch (e: Throwable) {

log("Throwable: $e")

}

}

return null

}

private fun getAssetsImage(url: String): WebResourceResponse? {

if (url.contains(".jpg")) {

try {

val inputStream = application.assets.open("ic\_launcher.webp")

return WebResourceResponse(

"image/webp",

"utf-8", inputStream

)

} catch (e: Throwable) {

log("Throwable: $e")

}

}

return null

}

}

采用此方案的好处有：

通过 OkHttp 本身的 Cache 功能来实现资源缓存，并不局限于特定的文件类型，可以用于图片、HTML、JS、CSS 等多种类型。

OkHttp 是完全遵循 Http 协议的，我们可以在这基础上来自由扩展 Http 缓存策略。

解耦了客户端和前端代码，由客户端充当 Server 的角色，对于前端来说是完全无感知的，用比较低的成本就实现了两端缓存共享。

WebView 自带的缓存机制允许的最大缓存空间是比较小的，此方案相当于突破了 WebView 的最大缓存容量限制。

如果移动端已经预置了离线包，那么就可以通过此方案判断离线包是否已经包含目标文件，存在的话直接使用，否则才联网请求，参照上述的 getAssetsImage方法。

需要注意，以上只是一份示例代码，并不能直接用于生产环境，读者需要根据具体业务去进行扩展。Github 上也有一个通过此方案实现了 WebView 缓存复用的开源库，读者可以去借鉴其思路：[CacheWebView](https://link.juejin.cn?target=https%3A%2F%2Fgithub.com%2Fyale8848%2FCacheWebView)。

**DNS 优化**

DNS 也即域名解析，指代的是将域名转换为具体的 IP 地址的过程。DNS 会在系统级别进行缓存，如果已经解析过某域名，那么在下次使用时就可以直接去访问已知的 IP 地址，而不用先发起 DNS 再访问 IP 地址。

如果 WebView 访问的主域名和客户端的不一致，那么 WebView 在首次访问线上资源时，就需要先完成域名解析才能开始资源请求，这个过程就需要多耗费几十毫秒的时间。因此最好就是保持客户端整体 API 地址、资源文件地址、WebView 线上地址的主域名都是一致的。

**CDN 加速**

CDN 的全称是 Content Delivery Network，即内容分发网络。CDN 是构建在现有网络基础之上的智能虚拟网络，依靠部署在各地的边缘服务器，通过中心平台的负载均衡、内容分发、调度等功能模块，使用户就近获取所需内容，降低网络拥塞，提高用户访问响应速度和命中率。

通过将 JS、CSS、图片、视频等静态类型文件托管到 CDN，当用户加载网页时，就可以从地理位置上最接近它们的服务器接收这些文件，解决了远距离访问和不同网络带宽线路访问造成的网络延迟情况。

**白屏检测**

在正常情况下，完成上述的优化措施后用户基本是可以秒开 H5 页面的了。但异常情况总是会有的，用户的网络环境和系统环境千差万别，甚至 WebView 也可能发生内部崩溃。当发生问题时，用户看到的可能就直接只是一个白屏页面了，所以进一步的优化手段就是需要去检测是否发生白屏以及相应的应对措施。

检测白屏最直观的方案就是对 WebView 进行截图，遍历截图的像素点的颜色值，如果非白屏颜色的颜色点超过一定的阈值，就可以认为不是白屏。字节跳动技术团队的做法是：通过 View.getDrawingCache()方法去获取包含 WebView 视图的 Bitmap 对象，然后把截图缩小到原图的 1/6，遍历检测图片的像素点，当非白色的像素点大于 5% 的时候就可以认为是非白屏的情况，可以相对高效且准确地判断出是否发生了白屏。

当检测到白屏后，如果发现怎么重试也无法成功，那就只能进行降级处理了，放弃上述的优化措施，直接加载线上的详情页，优先保证用户体验。