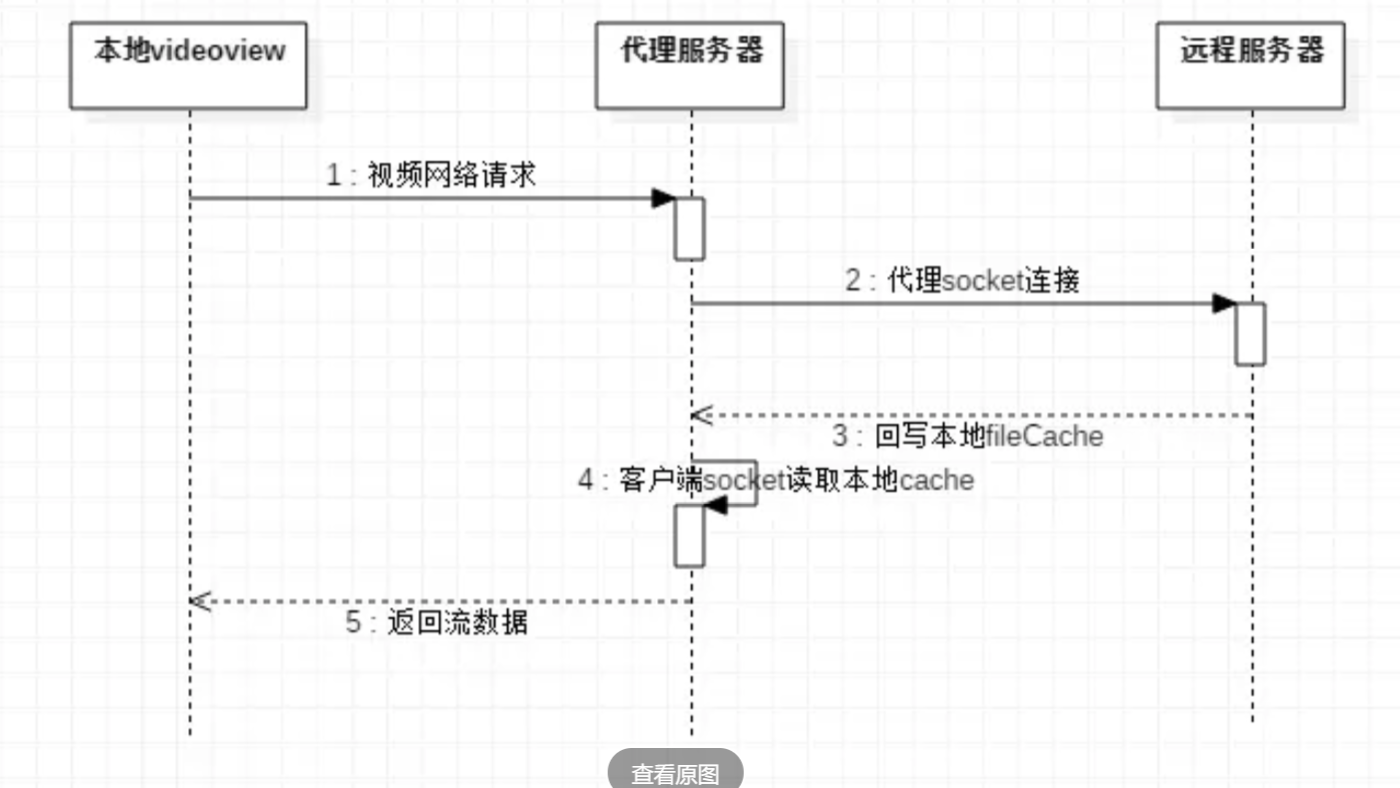
AndroidVideoCache-视频边播放边缓存的代理策略

# 1.视频现状

现在视频播放的需求越来越常见，就和16年上半年的直播一样，似乎不加个视频已经不是个正常的APP了，连微信朋友圈都支持上传小视频，更别谈以视频为本命的一系列APP。  
视频方面主要是两块，一个是视频录制，这个已经翻过一篇比较全的[文章](https://www.jianshu.com/p/e6c683d6dbbe)，再加上google开源的 [grafika](https://link.jianshu.com?t=https:/github.com/google/grafika) ，可以在踩坑时减少很多障碍，不过录制这块适配是大问题，需要不断调整。  
另一个方面就是视频播放，这方面的轮子比上面录制就多太多了，无论是google（开源良心）的 [ExoPlayer](https://link.jianshu.com?t=https:/github.com/google/ExoPlayer)，以及b站的 [ijkplayer](https://link.jianshu.com?t=https:/github.com/Bilibili/ijkplayer)，还是一些其他的，基本上满足了正常的需求。  
当然，今天这篇文章这两个并不是主角，在视频播放这种极其容易造成卡顿，跳帧等影响用户体验的需求上，如何优化体验是一件十分重要的事情。一般情况比较正常的就直接播放，一句设置数据源的代码了事。但是要为了用户体验考虑。纵观这些有视频功能的APP，主要分为两类，一种是直接下载然后再播放，比如微信，微信的小视频录制压缩比比较好，一个视频大概几百k，所以比较适合先全量下载，然后再播放的模式，另一种自然就是边播放边缓存，这是比较多的策略，大部分的视频都是比较大的，等全部下完，黄花菜都凉了。

# 2.基本原理

既然是采取边播放边缓存的策略，比较逗的方式就是一边正常的给videoview设置数据源，一边开一个线程去下载文件，下完后就可以使用本地缓存了，这个方式是比较逗的，相当于两份网络请求，大大的拖慢了用户体验。所以我们会想如何只有一份请求但是能够操作这些数据边读边写呢，这个就是 [AndroidVideoCache](https://link.jianshu.com?t=https:/github.com/danikula/AndroidVideoCache) 所做的事情。  
[AndroidVideoCache](https://link.jianshu.com?t=https:/github.com/danikula/AndroidVideoCache) 通过代理的策略实现一个中间层将我们的网络请求转移到本地实现的代理服务器上，这样我们真正请求的数据就会被代理拿到，这样代理一边向本地写入数据，一边根据我们需要的数据看是读网络数据还是读本地缓存数据再提供给我们，真正做到了数据的复用。  
这就和我们使用的抓包软件性质一样，上个原理图更清晰



代理服务器策略

## 2.1从使用开始

这里在如何使用上直接搬运作者自己的readme。  
首先AS用户一行代码在gradle中导包

dependencies {

compile 'com.danikula:videocache:2.6.4'

}

然后在全局初始化一个本地代理服务器，这里选择在Application的实现类中，至于这个类是干什么的，后面会详细分析

public class App extends Application {

private HttpProxyCacheServer proxy;

public static HttpProxyCacheServer getProxy(Context context) {

App app = (App) context.getApplicationContext();

return app.proxy == null ? (app.proxy = app.newProxy()) : app.proxy;

}

private HttpProxyCacheServer newProxy() {

return new HttpProxyCacheServer(this);

}

}

有了代理服务器，我们就可以使用了，把自己的网络视频url用提供的方法替换成另一个URL

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

HttpProxyCacheServer proxy = getProxy();

String proxyUrl = proxy.getProxyUrl(VIDEO\_URL);

videoView.setVideoPath(proxyUrl);

}

这样就已经可以正常使用了，当然这个库提供了更多的可以自定义的地方，比如缓存的文件最大大小，以及文件个数，缓存采取的是LruCache的方法，对于老文件在达到上限后会自动清理。

private HttpProxyCacheServer newProxy() {

return new HttpProxyCacheServer.Builder(this)

.maxCacheSize(1024 \* 1024 \* 1024) // 1 Gb for cache

.build();

}

private HttpProxyCacheServer newProxy() {

return new HttpProxyCacheServer.Builder(this)

.maxCacheFilesCount(20)

.build();

}

除了这个，还有一个就是生成的文件名，默认是使用的MD5方式生成key，考虑到一些业务逻辑，我们也可以继承一个 FileNameGenerator 来实现自己的策略

public class MyFileNameGenerator implements FileNameGenerator {

// Urls contain mutable parts (parameter 'sessionToken') and stable video's id (parameter 'videoId').

// e. g. http://example.com?videoId=abcqaz&sessionToken=xyz987

public String generate(String url) {

Uri uri = Uri.parse(url);

String videoId = uri.getQueryParameter("videoId");

return videoId + ".mp4";

}

}

...

HttpProxyCacheServer proxy = HttpProxyCacheServer.Builder(context)

.fileNameGenerator(new MyFileNameGenerator())

.build()

到这里基本上整个AndroidVideoCache的使用就没什么问题了。

**具体分析**

知道了怎么使用后，我们继续往下走，看看是怎么实现的，这里就不分析后面的那些LruCache这些缓存策略，生成key之类的逻辑了，和一般的网络请求里的都大同小异，我们直接看这个代码最有含金量的地方。  
前面在使用中，全局实例化过一个代理服务器，就先从这里开始

HttpProxyCacheServer.java

private static final String PROXY\_HOST = "127.0.0.1";

private final Object clientsLock = new Object();

private final ExecutorService socketProcessor = Executors.newFixedThreadPool(8);

private final Map<String, HttpProxyCacheServerClients> clientsMap = new ConcurrentHashMap<>();

private final ServerSocket serverSocket;

private final int port;

private final Thread waitConnectionThread;

private final Config config;

private final Pinger pinger;

private HttpProxyCacheServer(Config config) {

this.config = checkNotNull(config);

try {

InetAddress inetAddress = InetAddress.getByName(PROXY\_HOST);

this.serverSocket = new ServerSocket(0, 8, inetAddress);

this.port = serverSocket.getLocalPort();

CountDownLatch startSignal = new CountDownLatch(1);

this.waitConnectionThread = new Thread(new WaitRequestsRunnable(startSignal));

this.waitConnectionThread.start();

startSignal.await(); // freeze thread, wait for server starts

this.pinger = new Pinger(PROXY\_HOST, port);

LOG.info("Proxy cache server started. Is it alive? " + isAlive());

} catch (IOException | InterruptedException e) {

socketProcessor.shutdown();

throw new IllegalStateException("Error starting local proxy server", e);

}

}

这个构造函数一眼看过去就很清楚了，参数就是前面那些自定义配置，这里使用的是 127.0.0.1，这个就是localhost的ip也就是本地ip，创建了一个 ServerSocket ，随机分配了一个端口，这里通过 getLocalPort 拿到了这个服务器端口，后面用来通信。  
这里出现了一个线程 WaitRequestsRunnable 并且调用了 start 方法，继续跟进去看这个线程

@Override

public void run() {

startSignal.countDown();

waitForRequest();

}

信号量主要是为了保证这个run方法先执行，继续看这个 waitForRequest 方法

private void waitForRequest() {

try {

while (!Thread.currentThread().isInterrupted()) {

Socket socket = serverSocket.accept();

LOG.debug("Accept new socket " + socket);

socketProcessor.submit(new SocketProcessorRunnable(socket));

}

} catch (IOException e) {

onError(new ProxyCacheException("Error during waiting connection", e));

}

}

好了，到这里服务器socket一套比较清晰了，整理一下就是先构建一个全局的一个本地代理服务器 ServerSocket，指定一个随机端口，然后新开一个线程,在线程的 run 方法里，通过accept() 方法监听这个服务器socket的入站连接，accept() 方法会一直阻塞，直到有一个客户端尝试建立连接。  
现在有了服务器，然后就是客户端的socket，先从使用时代理替换url地方开始看

HttpProxyCacheServer proxy = getProxy();

String proxyUrl = proxy.getProxyUrl(VIDEO\_URL);

这里使用的是HttpProxyCacheServer 中的 getProxyUrl 方法

public String getProxyUrl(String url, boolean allowCachedFileUri) {

if (allowCachedFileUri && isCached(url)) {

File cacheFile = getCacheFile(url);

touchFileSafely(cacheFile);

return Uri.fromFile(cacheFile).toString();

}

return isAlive() ? appendToProxyUrl(url) : url;

}

整个策略就是如果本地已经缓存了，就直接那本地地址的Uri，并且touch一下文件，把时间更新后最新，因为后面LruCache是根据文件被访问的时间进行排序的，如果文件没有被缓存那么就会先走一下 isAlive() 方法， 这里会ping一下目标url，确保url是一个有效的，如果用户是通过代理访问的话，就会ping不通，这样就还是原生url，正常情况都会进入这个 appendToProxyUrl 方法里面。

private String appendToProxyUrl(String url) {

return String.format(Locale.US, "http://%s:%d/%s", PROXY\_HOST, port, ProxyCacheUtils.encode(url));

}

比较直接，这里拼接出来一个带有127.0.0.1目标地址及端口并携带原url的新地址，这个请求的话就会被我们的服务器socket监听到，也就是前面的accept() 会继续往下走，这里接收到的socket就是我们所请求的客户端socket

socketProcessor.submit(new SocketProcessorRunnable(socket));

整个socket会被包裹成一个runnable，发配给线程池。这个 runnable 的 run 方法中所做的事情就是调用了一个方法

private void processSocket(Socket socket) {

try {

GetRequest request = GetRequest.read(socket.getInputStream());

LOG.debug("Request to cache proxy:" + request);

String url = ProxyCacheUtils.decode(request.uri);

if (pinger.isPingRequest(url)) {

pinger.responseToPing(socket);

} else {

HttpProxyCacheServerClients clients = getClients(url);

clients.processRequest(request, socket);

}

} catch (SocketException e) {

// There is no way to determine that client closed connection http://stackoverflow.com/a/10241044/999458

// So just to prevent log flooding don't log stacktrace

LOG.debug("Closing socket… Socket is closed by client.");

} catch (ProxyCacheException | IOException e) {

onError(new ProxyCacheException("Error processing request", e));

} finally {

releaseSocket(socket);

LOG.debug("Opened connections: " + getClientsCount());

}

}

前面ping的过程其实也被会这个socket监听并且走进来这一段，不过这个比较简单，就不分析了，我们直接看里面的 else 框内的代码，这里一个 getClients 就是一个ConcurrentHashMap，重复url返回的是同一个HttpProxyCacheServerClients ，

HttpProxyCacheServerClients clients = clientsMap.get(url);

if (clients == null) {

clients = new HttpProxyCacheServerClients(url, config);

clientsMap.put(url, clients);

}

return clients;

如果是第一次就会根据url构建出一个HttpProxyCacheServerClients并被put到ConcurrentHashMap中，真正的操作都在这个客户端的 processRequest 操作中，并且传递过去一个是request，这是一个GetRequest 对象，是一个url和rangeoffset以及partial的包装类，另一个就是客户端socket。

public void processRequest(GetRequest request, Socket socket) throws ProxyCacheException, IOException {

startProcessRequest();

try {

clientsCount.incrementAndGet();

proxyCache.processRequest(request, socket);

} finally {

finishProcessRequest();

}

}

这里 startProcessRequest 方法会得到一个HttpProxyCache 类

private synchronized void startProcessRequest() throws ProxyCacheException {

proxyCache = proxyCache == null ? newHttpProxyCache() : proxyCache;

}

private HttpProxyCache newHttpProxyCache() throws ProxyCacheException {

HttpUrlSource source = new HttpUrlSource(url, config.sourceInfoStorage);

FileCache cache = new FileCache(config.generateCacheFile(url), config.diskUsage);

HttpProxyCache httpProxyCache = new HttpProxyCache(source, cache);

httpProxyCache.registerCacheListener(uiCacheListener);

return httpProxyCache;

}

在这里，我们构建一个基于原生url的HttpUrlSource ，这个类负责持有url，并开启HttpURLConnection来获取一个InputStream，这样才能通过这个输入流读数据，同时也创建了一个本地的临时文件，一个以.download结尾的临时文件，这个文件在成功下载完后的 FileCache 类中的 complete 方法中被更名。  
我们构建了一个HttpProxyCache 类，也注册了一个CacheListener，这个listener可以用来回调进度。  
做完这一切之后，然后这个HttpProxyCache 对象就开始 processRequest ，

public void processRequest(GetRequest request, Socket socket) throws IOException, ProxyCacheException {

OutputStream out = new BufferedOutputStream(socket.getOutputStream());

String responseHeaders = newResponseHeaders(request);

out.write(responseHeaders.getBytes("UTF-8"));

long offset = request.rangeOffset;

if (isUseCache(request)) {

responseWithCache(out, offset);

} else {

responseWithoutCache(out, offset);

}

}

这里我们用传过来的那个客户端socket，拿到一个OutputStream输出流，这样我们就能往里面写数据了，如果不用缓存就走常规逻辑，这里我们只看走缓存的行为。

private void responseWithCache(OutputStream out, long offset) throws ProxyCacheException, IOException {

byte[] buffer = new byte[DEFAULT\_BUFFER\_SIZE];

int readBytes;

while ((readBytes = read(buffer, offset, buffer.length)) != -1) {

out.write(buffer, 0, readBytes);

offset += readBytes;

}

out.flush();

}

构造一个8 \* 1024字节的buffer，这里的read方法，实际上是调用的父类ProxyCache的实现

public int read(byte[] buffer, long offset, int length) throws ProxyCacheException {

ProxyCacheUtils.assertBuffer(buffer, offset, length);

while (!cache.isCompleted() && cache.available() < (offset + length) && !stopped) {

readSourceAsync();

waitForSourceData();

checkReadSourceErrorsCount();

}

int read = cache.read(buffer, offset, length);

if (cache.isCompleted() && percentsAvailable != 100) {

percentsAvailable = 100;

onCachePercentsAvailableChanged(100);

}

return read;

}

在while循环里面，开启了一个新的线程sourceReaderThread，其中封装了一个SourceReaderRunnable的Runnable，这个异步线程用来给cache，也就是本地文件写数据，同时还更新一下当前的缓存进度

int sourceAvailable = -1;

int offset = 0;

try {

offset = cache.available();

source.open(offset);

sourceAvailable = source.length();

byte[] buffer = new byte[ProxyCacheUtils.DEFAULT\_BUFFER\_SIZE];

int readBytes;

while ((readBytes = source.read(buffer)) != -1) {

synchronized (stopLock) {

if (isStopped()) {

return;

}

cache.append(buffer, readBytes);

}

offset += readBytes;

notifyNewCacheDataAvailable(offset, sourceAvailable);

}

tryComplete();

onSourceRead();

同时我们的另一个线程也会从cache中去读数据，在缓存结束后同样也会发送一个通知通知自己已经缓存完了，回调由外界控制。  
以上差不多就是总体代码，这里我们在请求远程URL时将文件写到本地fileCache中，然后读数据从本地读取，写入到客户端socket里面，服务器Socket主要还是一个代理的作用，从中间拦截掉网络请求，然后实现对socket的读取和写入。

**后记**

整个分析为了节约篇幅，尽量的是描述一些其中比较重要的片段，源码文件还是比较多的，这里不能详述，对这种代理方式感兴趣的可以在自己详细阅读一下源码，毕竟源码面前，了无秘密。  
这个项目用起来有一点问题，是因为如果我们的APP设置了代理，那么这个socket方式拿url就会出问题，因为我们拿到的也是一个代理url，所以在开发时需要考虑代理用户提供兼容性处理。  
另外这种本地代理服务器的策略也能为我们提供一些不一样的思路，既然视频可行那么音频文件呢，进而推导到普通的网络请求，json文件。基于这样一套思路，在其基础上甚至能够实现一套离线缓存加载的策略，当然这取决于我们自身的服务器架构，服务端URL策略。

总结：

1. 视频播放的开源有哪些

(1) b站的 [ijkplayer](https://link.jianshu.com/?t=https://github.com/Bilibili/ijkplayer) (2) google开源的 [grafika](https://link.jianshu.com/?t=https://github.com/google/grafika" \t "_blank)

1. 为什么会有视频缓存

视频播放的地址分为两种 1.本地视频 2.在线视频

在线视频很可能因为网络不好导致卡顿，跳帧的情况

1. 解决的办法
2. 全部下载下来，然后再播放，例如：微信小视频，这种适用于视频比较小，但是如果文件特别大会等待很长时间
3. 一边缓存一边播放

例如AndroidVideoCache:创建代理代理服务器，将真正的url地址交给代理服務器，通知代理服务器返回一个本地地址，本地播放器向代理服务器发出请求，代理服务器请求正确的服务器，然后对请求到的数据写入到本地缓存文件中，如果本地缓存中有数据有数据，代理服务器把数据返回给播放器，利用缓存，从而保证视频的完整性以及流畅性

作者：juexingzhe  
链接：https://www.jianshu.com/p/53c4a6c9bd07  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

链接：<https://www.jianshu.com/p/4745de02dcdc>

<https://www.jianshu.com/p/53c4a6c9bd07>