今天就带大家一层层剥光Volley这位懵懂无知少女（14年出生，应该算无知少女）的外衣，带大家轻轻抚摸Volley每一寸肌肤，进入Volley的身体，为达到完美效果，开头录有激情小视频，观看时请自备纸巾，各位现在需要做的就是搬一把椅子，打开电脑，双击Android studio导入Volley源码，随我一起观看录制的激情小视频带着大家一起飞，观看激情小视频过程中，如能让你内心微微一颤有所收获，感受到满满的爱意，点赞或打赏都是最美的祝福。

### 源码分析相关面试题

* [Volley源码分析](http://www.jianshu.com/p/ec3dc92df581)
* [注解框架实现原理](http://www.jianshu.com/p/20da6d6389e1)
* [okhttp3.0源码分析](http://www.jianshu.com/p/9ed2c2f2a52c)
* [onSaveInstanceState源码分析](http://www.jianshu.com/p/cbf9c3557d64)
* [静默安装和源码编译](http://www.jianshu.com/p/2211a5b3c37f)

### Activity相关面试题

* [保存Activity的状态](http://www.jianshu.com/p/cbf9c3557d64)

### 与XMPP相关面试题

* [XMPP协议优缺点](http://www.jianshu.com/p/2c04ac3c526a)
* [极光消息推送原理](http://www.jianshu.com/p/d88dc66908cf)

### 与性能优化相关面试题

* [内存泄漏和内存溢出区别](http://www.jianshu.com/p/5dd645b05c76)
* [UI优化和线程池实现原理](http://www.jianshu.com/p/c22398f8587f)
* [代码优化](http://www.jianshu.com/p/ebd41eab90df)
* [内存性能分析](http://www.jianshu.com/p/2665c31b9c2f)
* [内存泄漏检测](http://www.jianshu.com/p/1514c7804a06)
* [App启动优化](http://www.jianshu.com/p/f0f73fefdd43)
* [与IPC机制相关面试题](http://www.jianshu.com/p/de4793a4c2d0)

### 与登录相关面试题

* [oauth认证协议原理](http://www.jianshu.com/p/2a6ecbf8d49d)
* [token产生的意义](http://www.jianshu.com/p/9b7ce2d6c195)
* [微信扫一扫实现原理](http://www.jianshu.com/p/a9d1f21bd5e0)

### 与开发相关面试题

* [迭代开发的时候如何向前兼容新旧接口](http://www.jianshu.com/p/cbecadec98de)
* [手把手教你如何解决as jar包冲突](http://www.jianshu.com/p/30fdc391289c)
* [context的原理分析](http://www.jianshu.com/p/2706c13a1769)
* [解决ViewPager.setCurrentItem中间很多页面切换方案](http://www.jianshu.com/p/38ab6d856b56)

### 与人事相关面试题

* [人事面试宝典](http://www.jianshu.com/p/d61b553ff8c9)

### 本文配套视频：

* [Volley内核分析配套视频一](https://v.qq.com/x/page/s05002geql6.html)
* [Volley内核分析配套视频二](https://v.qq.com/x/page/h05002uijux.html)
* [Volley内核分析配套视频三](https://v.qq.com/x/page/c05005gcs36.html)

通过一个小栗子开始咱们的源码分析

RequestQueue queue = Volley.newRequestQueue(this);  
 String url ="http://www.baidu.com";  
 StringRequest stringRequest = new StringRequest(Request.Method.GET,  
 url,  
 new Response.Listener<String>() {  
 @Override  
 public void onResponse(String response) {  
  
 }  
 }, new Response.ErrorListener() {  
  
 @Override  
 public void onErrorResponse(VolleyError error) {  
  
 }  
 });  
 queue.add(stringRequest);

### 第一部分：一行行分析

RequestQueue queue = Volley.newRequestQueue(this);

进入源码分析:

public static RequestQueue newRequestQueue(Context context) {  
 return newRequestQueue(context, (HttpStack)null);  
}

由以上源码分析可知：

1）该方法有两个参数，第一个Context是上下文，第二个参数为null

public static RequestQueue newRequestQueue(Context context, HttpStack stack) {  
 File cacheDir = new File(context.getCacheDir(), "volley");  
 String userAgent = "volley/0";  
 try {  
 String network = context.getPackageName();  
 PackageInfo queue = context.getPackageManager().getPackageInfo(network, 0);  
 userAgent = network + "/" + queue.versionCode;  
 } catch (NameNotFoundException var6) {  
  
 }  
 if(stack == null) {  
 if(VERSION.SDK\_INT >= 9) {  
 stack = new HurlStack();  
 } else {  
 stack = new HttpClientStack(AndroidHttpClient.newInstance(userAgent));  
 }  
 }  
  
 BasicNetwork network1 = new BasicNetwork((HttpStack)stack);  
 RequestQueue queue1 = new RequestQueue(new DiskBasedCache(cacheDir), network1);  
 queue1.start();  
 return queue1;  
 }

由以上源码分析可知：

1）初始化缓存文件，名字叫volley。

2）获取到当前应用包名和包的基本信息，并且给userAgent 重新赋值。

3）stack 传递过来为null，所以直接走if判断里面。

4）在这个if中对版本号进行了判断，如果版本号大于等于9就使得HttpStack对象的实例为HurlStack，如果小于9则实例为HttpClientStack。至于这里为何进行版本号判断，实际代码中已经说明了，请看下文。

1. BasicNetwork是Network接口的实现，BasicNetwork实现了performRequest方法，其作用是根据传入的HttpStack对象来处理网络请求。紧接着new出一个RequestQueue对象，并调用它的start()方法进行启动，然后将RequestQueue返回。RequestQueue是根目录下的一个类，其作用是一个请求调度队列调度程序的线程池。这样newRequestQueue()的方法就执行结束了。

第四条如何分析出来请看如下代码：

public class HurlStack implements HttpStack   
public class HttpClientStack implements HttpStack

继承httpStack接口

public interface HttpStack {  
 HttpResponse performRequest(Request<?> var1, Map<String, String> var2) throws IOException, AuthFailureError;  
}

接口当中有如下方法，performRequest，大家有没有觉得奇怪都继承同一个方法，并且都实现一样的接口同样的方法，实际上子类实现方法不一样，代码如下：

#### HttpClientStack代码如下:

public HttpResponse performRequest(Request<?> request, Map<String, String> additionalHeaders) throws IOException, AuthFailureError {  
 HttpUriRequest httpRequest = createHttpRequest(request, additionalHeaders);  
 addHeaders(httpRequest, additionalHeaders);  
 addHeaders(httpRequest, request.getHeaders());  
 this.onPrepareRequest(httpRequest);  
 HttpParams httpParams = httpRequest.getParams();  
 int timeoutMs = request.getTimeoutMs();  
 HttpConnectionParams.setConnectionTimeout(httpParams, 5000);  
 HttpConnectionParams.setSoTimeout(httpParams, timeoutMs);  
 return this.mClient.execute(httpRequest);  
 }

#### HurlStack代码如下：

public HttpResponse performRequest(Request<?> request, Map<String, String> additionalHeaders) throws IOException, AuthFailureError {  
 ......  
  
 URL parsedUrl1 = new URL(url);  
 HttpURLConnection connection = this.openConnection(parsedUrl1, request);  
  
 .......  
 return response;  
 }  
 }

由以上得知HttpClientStack走的是httpClient，HurlStack走的是HttpURLConnection ，谷歌在高版本当中已经去掉了httpClient，所以这里必须做版本号的判断。

继续源码分析，现在再来看下RequestQueue队列的start方法，如下所示：

public void start() {  
 this.stop();  
 this.mCacheDispatcher = new CacheDispatcher(this.mCacheQueue, this.mNetworkQueue, this.mCache, this.mDelivery);  
 this.mCacheDispatcher.start();  
  
for(int i = 0; i < this.mDispatchers.length; ++i) {  
 NetworkDispatcher networkDispatcher = new NetworkDispatcher(this.mNetworkQueue, this.mNetwork, this.mCache, this.mDelivery);  
 this.mDispatchers[i] = networkDispatcher;  
 networkDispatcher.start();  
 }  
}

以上源码可知：

1）创建了一个CacheDispatcher的实例，然后调用了它的start()方法

2）for循环里去创建NetworkDispatcher的实例，并分别调用它们的start()方法

3）这里的CacheDispatcher和NetworkDispatcher都是继承自Thread的，而默认情况下for循环会执行四次，也就是说当调用了Volley.newRequestQueue(context)之后，就会有五个线程一直在后台运行，不断等待网络请求的到来，其中一个CacheDispatcher是缓存线程，四个NetworkDispatcher是网络请求线程。

第三条如何分析出来for循环会执行四次 , 请看如下代码分析：

public RequestQueue(Cache cache, Network network, int threadPoolSize, ResponseDelivery delivery) {  
 ......  
 this.mDispatchers = new NetworkDispatcher[threadPoolSize];  
 ......  
 }

threadPoolSize大小是4，如何看出来请看如下代码：

public RequestQueue(Cache cache, Network network, int threadPoolSize) {  
 this(cache, network, threadPoolSize, new ExecutorDelivery(new Handler(Looper.getMainLooper())));  
 }  
  
 public RequestQueue(Cache cache, Network network) {  
 this(cache, network, 4);  
 }

通过构造方法传递，默认是4，如上已经把第一行代码分析完毕，进入第二部分。

RequestQueue queue = Volley.newRequestQueue(this);

### 第二部分：添加请求到队列

StringRequest stringRequest = new StringRequest(Request.Method.GET,  
 url,  
new Response.Listener<String>() {  
@Override  
public void onResponse(String response) {  
  
 }  
}, new Response.ErrorListener() {  
  
 @Override  
 public void onErrorResponse(VolleyError error) {  
  
 }  
 });  
 queue.add(stringRequest);

调用RequestQueue的add()方法将Request传入就可以完成网络请求操作了。也就是说add()方法的内部是核心代码了。现在看下RequestQueue的add方法，具体如下：

public Request add(Request request) {  
 request.setRequestQueue(this);  
 Set var2 = this.mCurrentRequests;  
 synchronized(this.mCurrentRequests) {  
 this.mCurrentRequests.add(request);  
 }  
  
 request.setSequence(this.getSequenceNumber());  
 request.addMarker("add-to-queue");  
 if(!request.shouldCache()) {  
 this.mNetworkQueue.add(request);  
 return request;  
 } else {  
 Map var7 = this.mWaitingRequests;  
 synchronized(this.mWaitingRequests) {  
 String cacheKey = request.getCacheKey();  
 if(this.mWaitingRequests.containsKey(cacheKey)) {  
 Object stagedRequests = (Queue)this.mWaitingRequests.get(cacheKey);  
 if(stagedRequests == null) {  
 stagedRequests = new LinkedList();  
 }  
 ((Queue)stagedRequests).add(request);  
 this.mWaitingRequests.put(cacheKey, stagedRequests);  
 }  
 } else {  
 this.mWaitingRequests.put(cacheKey, (Object)null);  
 this.mCacheQueue.add(request);  
 }  
  
 return request;  
  
 }

通过以上源码可知：

1）Request是所有请求的基类，是一个抽象类。request.setRequestQueue(this);的作用就是将请求Request关联到当前RequestQueue。然后同步操作将当前Request添加到RequestQueue对象的mCurrentRequests HashSet中做记录。

2）通过request.setSequence(getSequenceNumber());得到当前RequestQueue中请求的个数，然后关联到当前Request。

3）request.addMarker("add-to-queue");添加调试的队列标记。

4）if (!request.shouldCache())判断当前的请求是否可以缓存，如果不能缓存则直接通过mNetworkQueue.add(request);将这条请求加入网络请求队列，然后返回request；如果可以缓存的话则在通过同步操作将这条请求加入缓存队列。

第二部分分析完毕，进入第三部分。

### 第三部分：CacheDispatcher中的run()方法，代码如下所示：

public void run() {  
 if(DEBUG) {  
 VolleyLog.v("start new dispatcher", new Object[0]);  
 }  
  
 Process.setThreadPriority(10);  
 this.mCache.initialize();  
  
  
 while(true) {  
 final Request e = (Request)this.mCacheQueue.take();  
 e.addMarker("cache-queue-take");  
 if(e.isCanceled()) {  
 e.finish("cache-discard-canceled");  
 } else {  
 Entry entry = this.mCache.get(e.getCacheKey());  
 if(entry == null) {  
 e.addMarker("cache-miss");  
 this.mNetworkQueue.put(e);  
 } else if(entry.isExpired()) {  
 e.addMarker("cache-hit-expired");  
 e.setCacheEntry(entry);  
 this.mNetworkQueue.put(e);  
 } else {  
 e.addMarker("cache-hit");  
 Response response = e.parseNetworkResponse(new NetworkResponse(entry.data, entry.responseHeaders));  
 e.addMarker("cache-hit-parsed");  
 if(entry.refreshNeeded()) {  
 e.addMarker("cache-hit-refresh-needed");  
 e.setCacheEntry(entry);  
 response.intermediate = true;  
 this.mDelivery.postResponse(e, response,   
 new Runnable() {  
 public void run() {  
 CacheDispatcher.this.mNetworkQueue.put(e);  
 }

### 以上源码可知：

1）首先通过Process.setThreadPriority设置线程优先级

2）mCache.initialize(); 初始化缓存块

3）while(true)循环，表示它一直在等待缓存队列的新请求的出现

4）接着，先判断这个请求是否有对应的缓存结果，如果没有则直接添加到网络请求队列

5）再判断这个缓存结果是否过期了，如果过期则同样地添加到网络请求队列

1. 接下来便是对缓存结果的处理了，我们可以看到，先是把缓存结果包装成NetworkResponse类，然后调用了Request的parseNetworkResponse;

### 小结

CacheDispatcher线程主要对请求进行判断，是否已经有缓存，是否已经过期，根据需要放进网络请求队列。同时对相应结果进行包装、处理，然后交由ExecutorDelivery处理。这里以一张流程图显示它的完整工作流程：



img

### 第四部分：NetworkDispatcher代码如下所示：

public void run() {  
 Process.setThreadPriority(10);  
 while(true) {  
 Request request;  
 while(true) {  
 try {  
 request = (Request)this.mQueue.take();  
 break;  
 } catch (InterruptedException var4) {  
  
 }  
 try {  
 request.addMarker("network-queue-take");  
 if(request.isCanceled()) {  
 request.finish("network-discard-cancelled");  
 } else {  
 ......  
 NetworkResponse e = this.mNetwork.performRequest(request);  
 request.addMarker("network-http-complete");  
 if(e.notModified && request.hasHadResponseDelivered()) {  
 request.finish("not-modified");  
 } else {  
 Response response = request.parseNetworkResponse(e);  
 request.addMarker("network-parse-complete");  
 if(request.shouldCache() && response.cacheEntry != null) {  
 this.mCache.put(request.getCacheKey(), response.cacheEntry);  
 request.addMarker("network-cache-written");  
 }  
 request.markDelivered();  
 this.mDelivery.postResponse(request, response);  
 }

### 由以上代码分析可知：

1）通过mNetwork.performRequest(request);代码来发送网络请求

2）而Network是一个接口，这里具体的实现之前已经分析是BasicNetwork，所以先看下它的performRequest()方法，如下所示：

public NetworkResponse performRequest(Request<?> request) throws VolleyError {  
 long requestStart = SystemClock.elapsedRealtime();  
 while(true) {  
 HttpResponse httpResponse = null;  
 Object responseContents = null;  
 HashMap responseHeaders = new HashMap();  
 try {  
 HashMap e = new HashMap();  
 this.addCacheHeaders(e, request.getCacheEntry());  
 httpResponse = this.mHttpStack.performRequest(request, e);  
 ......  
 byte[] responseContents1;  
 if(httpResponse.getEntity() != null) {  
 responseContents1 = this.entityToBytes(httpResponse.getEntity());  
 } else {  
 responseContents1 = new byte[0];  
 }  
  
 long requestLifetime = SystemClock.elapsedRealtime() - requestStart;  
 this.logSlowRequests(requestLifetime, request, responseContents1, statusCode2);  
 if(networkResponse1 >= 200 && networkResponse1 <= 299) {  
 return new NetworkResponse(networkResponse1, responseContents1, responseHeaders1, false);  
 }  
}

### 由上述代码可知：

1）httpResponse = this.mHttpStack.performRequest(request, e)该方法返回了httpResponse

2）把httpResponse 交给 new NetworkResponse对象进行处理，封装成NetworkResponse对象并返回

3）在NetworkDispatcher#run()方法获取返回的NetworkResponse对象后，对响应解析

4）在解析完了NetworkResponse中的数据之后，又会调用ExecutorDelivery（ResponseDelivery接口的实现类）的postResponse()方法来回调解析出的数据，具体代码如下所示：

this.mDelivery.postResponse(request, response);

public void postResponse(Request<?> request, Response<?> response, Runnable runnable) {  
 request.markDelivered();  
 request.addMarker("post-response");  
 this.mResponsePoster.execute(new ExecutorDelivery.ResponseDeliveryRunnable(request, response, runnable));  
}

这里可以看见在mResponsePoster的execute()方法中传入了一个ResponseDeliveryRunnable对象，就可以保证该对象中的run()方法就是在主线程当中运行的了，我们看下run()方法中的代码是什么样的：

if(this.mResponse.isSuccess()) {  
 this.mRequest.deliverResponse(this.mResponse.result);  
 } else {  
 this.mRequest.deliverError(this.mResponse.error);  
}

### 以上代码可知

mRequest的deliverResponse或者deliverError将反馈发送到回调到UI线程。这也是你重写实现的接口方法，到目前为止，关于Volley的源码解析完毕。

### 总结

1）当一个RequestQueue被成功申请后会开启一个CacheDispatcher和4个默认的NetworkDispatcher。

2）CacheDispatcher缓存调度器最为第一层缓冲，开始工作后阻塞的从缓存序列mCacheQueue中取得请求；对于已经取消的请求，标记为跳过并结束这个请求；新的或者过期的请求，直接放入mNetworkQueue中由N个NetworkDispatcher进行处理；已获得缓存信息（网络应答）却没有过期的请求，由Request的parseNetworkResponse进行解析，从而确定此应答是否成功。然后将请求和应答交由Delivery分发者进行处理，如果需要更新缓存那么该请求还会被放入mNetworkQueue中。

3）将请求Request add到RequestQueue后对于不需要缓存的请求（需要额外设置，默认是需要缓存）直接丢入mNetworkQueue交给N个NetworkDispatcher处理；对于需要缓存的，新的请求加到mCacheQueue中给CacheDispatcher处理；需要缓存，但是缓存列表中已经存在了相同URL的请求，放在mWaitingQueue中做暂时处理，等待之前请求完毕后，再重新添加到mCacheQueue中。

4）网络请求调度器NetworkDispatcher作为网络请求真实发生的地方，对消息交给BasicNetwork进行处理，同样的，请求和结果都交由Delivery分发者进行处理。