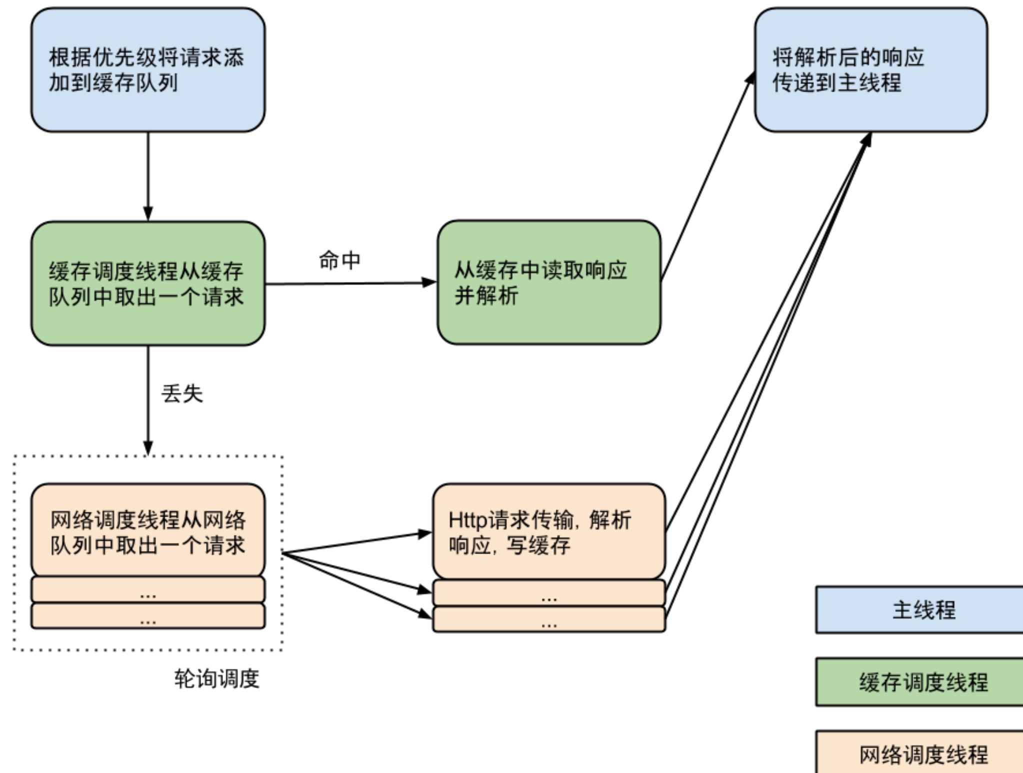
从源码解析 Volley

1. Volley 结构图



从上图可以看到 Volley 分为三个线程，分别是主线程、缓存调度线程、和网络调度线程，首先请求会加入缓存队列，如果发现可以找到相应的缓存结果就直接读取缓存并解析，然后回调给主线程；如果在缓存中没有找到结果，则将这条请求加入到网络队列中，然后发送 HTTP 请求，解析响应并写入缓存，并回调给主线程。

2. 从 RequestQueue 入手

我们都知道使用 Volley 之前首先要创建 RequestQueue：

```
RequestQueue mQueue = Volley.newRequestQueue(getApplicationContext());
```

这也是 volley 运作的入口，看看 newRequestQueue：

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

```
public static RequestQueue newRequestQueue(Context context, HttpStack stack) {
        return newRequestQueue(context, stack, -1);
    }
```

连续调用了两个重载函数，最终调用的是：

```
public static RequestQueue newRequestQueue(Context context, HttpStack stack, int
maxDiskCacheBytes) {
        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 var7) {
            ;
        }

        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;
        if(maxDiskCacheBytes <= -1) {
            queue1 = new RequestQueue(new DiskBasedCache(cacheDir), network1);
        } else {
            queue1 = new RequestQueue(new DiskBasedCache(cacheDir,
maxDiskCacheBytes), network1);
        }

        queue1.start();
        return queue1;
    }
```

可以看到如果android版本大于等于2.3则调用基于HttpURLConnection的HurlStack，否则就调用基于HttpClient的HttpClientStack。并创建了RequestQueue，调用了start()方法：

```
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();
      }

  }
```

CacheDispatcher 是缓存调度线程，并调用了 start()方法，在循环中调用了 NetworkDispatcher 的 start()方法，NetworkDispatcher 是网络调度线程，默认情况下 mDispatchers.length 为 4，默认开启了 4 个网络调度线程，也就是说有 5 个线程在后台运行并等待请求的到来。接下来我们创建各种的 Request，并调用 RequestQueue 的 add()方法：

```
public <T> Request<T> add(Request<T> request) {
        request.setRequestQueue(this);
        Set var2 = this.mCurrentRequests;
        synchronized(this.mCurrentRequests) {
            this.mCurrentRequests.add(request);
        }

        request.setSequence(this.getSequenceNumber());
        request.addMarker("add-to-queue");
        //如果不能缓存，则将请求添加到网络请求队列中
        if(!request.shouldCache()) {
            this.mNetworkQueue.add(request);
            return request;
        } else {
            Map var8 = this.mWaitingRequests;
            synchronized(this.mWaitingRequests) {
                String cacheKey = request.getCacheKey();

      //之前是否有执行相同的请求且还没有返回结果的，如果有的话将此请求加入
mWaitingRequests 队列，不再重复请求
                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);
            if(VolleyLog.DEBUG) {
                VolleyLog.v("Request for cacheKey=%s is in flight, putting on hold.", new Object[]{cacheKey});
            }
        } else {
 //没有的话就将请求加入缓存队列 mCacheQueue，同时加入 mWaitingRequests 中用来做下次同样请求来时的重复判断依据
            this.mWaitingRequests.put(cacheKey, (Object)null);
            this.mCacheQueue.add(request);
        }

        return request;
    }
  }
}
```

通过判断 request.shouldCache()，来判断是否可以缓存，默认是可以缓存的，如果不能缓存，则将请求添加到网络请求队列中，如果能缓存就判断之前是否有执行相同的请求且还没有返回结果的，如果有的话将此请求加入 mWaitingRequests 队列，不再重复请求；没有的话就将请求加入缓存队列 mCacheQueue，同时加入 mWaitingRequests 中用来做下次同样请求来时的重复判断依据。

从上面可以看出 RequestQueue 的 add()方法并没有做什么请求网络或者对缓存进行操作。当将请求添加到网络请求队列或者缓存队列时，这时在后台的网络调

度线程和缓存调度线程轮询各自的请求队列发现有请求任务则开始执行，我们先看看缓存调度线程。

3.CacheDispatcher 缓存调度线程

CacheDispatcher 的 run()方法：

```
public void run() {
    if(DEBUG) {
        VolleyLog.v("start new dispatcher", new Object[0]);
    }
    //线程优先级设置为最高级别
    Process.setThreadPriority(10);
    this.mCache.initialize();

    while(true) {
        while(true) {
            while(true) {
                while(true) {
                    try {
                    //获取缓存队列中的一个请求
                        final Request e = (Request)this.mCacheQueue.take();
                        e.addMarker("cache-queue-take");
                        //如果请求取消了则将请求停止掉
                        if(e.isCanceled()) {
                            e.finish("cache-discard-canceled");
                        } else {
                        //查看是否有缓存的响应
                            Entry entry = this.mCache.get(e.getCacheKey());
                            //如果缓存响应为空，则将请求加入网络请求队列
                            if(entry == null) {
                                e.addMarker("cache-miss");
```

```
                this.mNetworkQueue.put(e);
            //判断缓存响应是否过期
            } else if(!entry.isExpired()) {
                e.addMarker("cache-hit");
                //对数据进行解析并回调给主线程
                Response response = e.parseNetworkResponse(new
NetworkResponse(entry.data, entry.responseHeaders));
                e.addMarker("cache-hit-parsed");
                if(!entry.refreshNeeded()) {
                    this.mDelivery.postResponse(e, response);
                } else {
                    e.addMarker("cache-hit-refresh-needed");
                    e.setCacheEntry(entry);
                    response.intermediate = true;
                    this.mDelivery.postResponse(e, response, new
Runnable() {
                        public void run() {
                            try {

CacheDispatcher.this.mNetworkQueue.put(e);
                            } catch (InterruptedException var2) {
                                ;
                            }

                        }
                    });
                }
            } else {
                e.addMarker("cache-hit-expired");
```

```
                        e.setCacheEntry(entry);
                        this.mNetworkQueue.put(e);
                    }
                }
            } catch (InterruptedException var4) {
                if(this.mQuit) {
                    return;
                }
            }
        }
    }
  }
}

static {
    DEBUG = VolleyLog.DEBUG;
}
```

看到四个while循环有些晕吧，让我们挑重点的说，首先从缓存队列取出请求，判断是否请求是否被取消了，如果没有则判断该请求是否有缓存的响应，如果有并且没有过期则对缓存响应进行解析并回调给主线程。接下来看看网络调度线程。

4.NetworkDispatcher 网络调度线程

NetworkDispatcher的run()方法：

```
public void run() {
    Process.setThreadPriority(10);

    while(true) {
```

```
long startTimeMs;
Request request;
while(true) {
    startTimeMs = SystemClock.elapsedRealtime();

    try {
    //从队列中取出请求
        request = (Request)this.mQueue.take();
        break;
    } catch (InterruptedException var6) {
        if(this.mQuit) {
            return;
        }
    }
}

try {
    request.addMarker("network-queue-take");
    if(request.isCanceled()) {
        request.finish("network-discard-cancelled");
    } else {
        this.addTrafficStatsTag(request);
        //请求网络
        NetworkResponse e = this.mNetwork.performRequest(request);
        request.addMarker("network-http-complete");
        if(e.notModified && request.hasHadResponseDelivered()) {
            request.finish("not-modified");
        } else {
            Response volleyError1 = request.parseNetworkResponse(e);
```

```
                        request.addMarker("network-parse-complete");
                        if(request.shouldCache() && volleyError1.cacheEntry != null) {
                            //将响应结果存入缓存
                            this.mCache.put(request.getCacheKey(),
volleyError1.cacheEntry);
                            request.addMarker("network-cache-written");
                        }

                        request.markDelivered();
                        this.mDelivery.postResponse(request, volleyError1);
                    }
                }
            } catch (VolleyError var7) {
                var7.setNetworkTimeMs(SystemClock.elapsedRealtime() - startTimeMs);
                this.parseAndDeliverNetworkError(request, var7);
            } catch (Exception var8) {
                VolleyLog.e(var8, "Unhandled exception %s", new
Object[]{var8.toString()});
                VolleyError volleyError = new VolleyError(var8);
                volleyError.setNetworkTimeMs(SystemClock.elapsedRealtime() -
startTimeMs);
                this.mDelivery.postError(request, volleyError);
            }
        }
    }
```

网络调度线程也是从队列中取出请求并且判断是否被取消了,如果没取消就去请求网络得到响应并回调给主线程。请求网络时调用 this.mNetwork.performRequest(request),这个 mNetwork 是一个接口,实现它的类是 BasicNetwork, 我们来看看 BasicNetwork 的 performRequest()方法:

```
public NetworkResponse performRequest(Request<?> request) throws VolleyError {
        long requestStart = SystemClock.elapsedRealtime();

        while(true) {
            HttpResponse httpResponse = null;
            Object responseContents = null;
            Map responseHeaders = Collections.emptyMap();

            try {
                HashMap e = new HashMap();
                this.addCacheHeaders(e, request.getCacheEntry());
                httpResponse = this.mHttpStack.performRequest(request, e);
                StatusLine statusCode1 = httpResponse.getStatusLine();
                int networkResponse1 = statusCode1.getStatusCode();
                responseHeaders = convertHeaders(httpResponse.getAllHeaders());
                if(networkResponse1 == 304) {
                    Entry requestLifetime2 = request.getCacheEntry();
                    if(requestLifetime2 == null) {
                        return new NetworkResponse(304, (byte[])null,
responseHeaders, true, SystemClock.elapsedRealtime() - requestStart);
                    }

                    requestLifetime2.responseHeaders.putAll(responseHeaders);
                    return new NetworkResponse(304, requestLifetime2.data,
requestLifetime2.responseHeaders, true, SystemClock.elapsedRealtime() - requestStart);
                }

...省略
```

从上面可以看到在12行调用的是HttpStack的performRequest()方法请求网络，接下来根据不同的响应状态码来返回不同的 NetworkResponse。另外HttpStack 也是一个接口，实现它的两个类我们在前面已经提到了就是HurlStack 和 HttpClientStack。让我们再回到 NetworkDispatcher，请求网络后，会将响应结果存在缓存中，如果响应结果成功则调用this.mDelivery.postResponse(request, volleyError1)来回调给主线程。来看看 Delivery 的 postResponse()方法：

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

来看看 ResponseDeliveryRunnable 里面做了什么：

```
private class ResponseDeliveryRunnable implements Runnable {
        private final Request mRequest;
        private final Response mResponse;
        private final Runnable mRunnable;

        public ResponseDeliveryRunnable(Request request, Response response, Runnable
runnable) {
            this.mRequest = request;
            this.mResponse = response;
            this.mRunnable = runnable;
        }

        public void run() {
            if(this.mRequest.isCanceled()) {
```

```
                this.mRequest.finish("canceled-at-delivery");
            } else {
                if(this.mResponse.isSuccess()) {
                    this.mRequest.deliverResponse(this.mResponse.result);
                } else {
                    this.mRequest.deliverError(this.mResponse.error);
                }

                if(this.mResponse.intermediate) {
                    this.mRequest.addMarker("intermediate-response");
                } else {
                    this.mRequest.finish("done");
                }

                if(this.mRunnable != null) {
                    this.mRunnable.run();
                }

            }
        }
    }
```

第17行调用了this.mRequest.deliverResponse(this.mResponse.result)，这个就是实现Request抽象类必须要实现的方法，我们来看看StringRequest的源码：

```
public class StringRequest extends Request<String> {
    private final Listener<String> mListener;
```

```
    public StringRequest(int method, String url, Listener<String> listener, ErrorListener
errorListener) {
        super(method, url, errorListener);
        this.mListener = listener;
    }

    public StringRequest(String url, Listener<String> listener, ErrorListener errorListener) {
        this(0, url, listener, errorListener);
    }

    protected void deliverResponse(String response) {
        this.mListener.onResponse(response);
    }

 …省略
}
```

在 deliverResponse 方法中调用了 this.mListener.onResponse(response)，最终将 response 回调给了 Response.Listener 的 onResponse()方法。我们用 StringRequest 请求网络的写法是这样的：

```
RequestQueue mQueue = Volley.newRequestQueue(getApplicationContext());
        StringRequest mStringRequest = new StringRequest(Request.Method.GET,
"http://www.baidu.com",
                new Response.Listener<String>() {
                    @Override
                    public void onResponse(String response) {
                        Log.i("wangshu", response);
                    }
                }, new Response.ErrorListener() {
```

```
        @Override
        public void onErrorResponse(VolleyError error) {
            Log.e("wangshu", error.getMessage(), error);
        }
    });
//将请求添加在请求队列中
mQueue.add(mStringRequest);
```

看到第5行整个Volley的大致流程都通了吧，好了关于Volley的源码就讲到这里。