okhttp源码特别特别复杂，类涉及较多，导致本文非常长，我相信没有几个人能把本文看完，所以特意录制了跟文章同步的视频。

### 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：

* [okhttp内核分析配套视频一](https://v.qq.com/x/page/j050015e4sm.html)
* [okhttp内核分析配套视频二](https://v.qq.com/x/page/i05006qtood.html)
* [okhttp内核分析配套视频三](https://v.qq.com/x/page/y0500461od9.html)

#### 基本使用

从使用方法出发，首先是怎么使用，其次是我们使用的功能在内部是如何实现的.建议大家下载 OkHttp 源码之后，跟着本文，过一遍源码。

官方博客栗子：<http://square.github.io/okhttp/#examples>

OkHttpClient client = new OkHttpClient();  
  
String run(String url) throws IOException {  
 Request request = new Request.Builder()  
 .url(url)  
 .build();  
  
 Response response = client.newCall(request).execute();  
 return response.body().string();  
}

### Request、Response、Call 基本概念

上面的代码中涉及到几个常用的类：Request、Response和Call。下面分别介绍：

#### Request

每一个HTTP请求包含一个URL、一个方法（GET或POST或其他）、一些HTTP头。请求还可能包含一个特定内容类型的数据类的主体部分。

#### Response

响应是对请求的回复，包含状态码、HTTP头和主体部分。

#### Call

OkHttp使用Call抽象出一个满足请求的模型，尽管中间可能会有多个请求或响应。执行Call有两种方式，同步或异步

### 第一步：创建 OkHttpClient对象,进行源码分析：

OkHttpClient client = new OkHttpClient();

通过okhttp源码分析,直接创建的 OkHttpClient对象并且默认构造builder对象进行初始化

public class OkHttpClient implements Cloneable, Call.Factory, WebSocket.Factory {  
 public OkHttpClient() {  
 this(new Builder());  
 }  
 OkHttpClient(Builder builder) {  
 this.dispatcher = builder.dispatcher;  
 this.proxy = builder.proxy;  
 this.protocols = builder.protocols;  
 this.connectionSpecs = builder.connectionSpecs;  
 this.interceptors = Util.immutableList(builder.interceptors);  
 this.networkInterceptors = Util.immutableList(builder.networkInterceptors);  
 this.eventListenerFactory = builder.eventListenerFactory;  
 this.proxySelector = builder.proxySelector;  
 this.cookieJar = builder.cookieJar;  
 this.cache = builder.cache;  
 this.internalCache = builder.internalCache;  
 this.socketFactory = builder.socketFactory;  
  
 boolean isTLS = false;  
 ......  
  
 this.hostnameVerifier = builder.hostnameVerifier;  
 this.certificatePinner = builder.certificatePinner.withCertificateChainCleaner(  
 certificateChainCleaner);  
 this.proxyAuthenticator = builder.proxyAuthenticator;  
 this.authenticator = builder.authenticator;  
 this.connectionPool = builder.connectionPool;  
 this.dns = builder.dns;  
 this.followSslRedirects = builder.followSslRedirects;  
 this.followRedirects = builder.followRedirects;  
 this.retryOnConnectionFailure = builder.retryOnConnectionFailure;  
 this.connectTimeout = builder.connectTimeout;  
 this.readTimeout = builder.readTimeout;  
 this.writeTimeout = builder.writeTimeout;  
 this.pingInterval = builder.pingInterval;  
 }  
}

### 第二步：接下来发起 HTTP 请求

Request request = new Request.Builder().url("url").build();  
okHttpClient.newCall(request).enqueue(new Callback() {  
 @Override  
 public void onFailure(Call call, IOException e) {  
  
 }  
  
@Override  
public void onResponse(Call call, Response response) throws IOException {  
  
}  
});

### 第二步：代码流程分析：

Request request = new Request.Builder().url("url").build();

初始化构建者模式和请求对象，并且用URL替换Web套接字URL。

public final class Request {  
 public Builder() {  
 this.method = "GET";  
 this.headers = new Headers.Builder();  
 }  
 public Builder url(String url) {  
 ......  
  
 // Silently replace web socket URLs with HTTP URLs.  
 if (url.regionMatches(true, 0, "ws:", 0, 3)) {  
 url = "http:" + url.substring(3);  
 } else if (url.regionMatches(true, 0, "wss:", 0, 4)) {  
 url = "https:" + url.substring(4);  
 }  
  
 HttpUrl parsed = HttpUrl.parse(url);  
 ......  
 return url(parsed);  
 }  
 public Request build() {  
 ......  
 return new Request(this);  
 }  
}

### 第三步：方法解析：

okHttpClient.newCall(request).enqueue(new Callback() {  
@Override  
public void onFailure(Call call, IOException e) {  
  
}  
  
@Override  
public void onResponse(Call call, Response response) throws IOException {  
  
}  
});

源码分析：

public class OkHttpClient implements Cloneable, Call.Factory, WebSocket.Factory {  
 @Override   
 public Call newCall(Request request) {  
 return new RealCall(this, request, false /\* for web socket \*/);  
 }  
  
  
  
}

RealCall实现了Call.Factory接口创建了一个RealCall的实例，而RealCall是Call接口的实现。

### 异步请求的执行流程

final class RealCall implements Call {  
 @Override   
 public void enqueue(Callback responseCallback) {  
 synchronized (this) {  
 if (executed) throw new IllegalStateException("Already Executed");  
 executed = true;  
 }  
 captureCallStackTrace();  
 client.dispatcher().enqueue(new AsyncCall(responseCallback));  
 }  
}

### 由以上源码得知：

1） 检查这个 call 是否已经被执行了，每个 call 只能被执行一次，如果想要一个完全一样的 call，可以利用 call#clone 方法进行克隆。

2）利用 client.dispatcher().enqueue(this) 来进行实际执行，dispatcher 是刚才看到的 OkHttpClient.Builder 的成员之一

3）AsyncCall是RealCall的一个内部类并且继承NamedRunnable，那么首先看NamedRunnable类是什么样的，如下：

public abstract class NamedRunnable implements Runnable {  
 ......  
  
 @Override   
 public final void run() {  
 ......  
 try {  
 execute();  
 }  
 ......  
 }  
  
 protected abstract void execute();  
}

可以看到NamedRunnable实现了Runnbale接口并且是个抽象类，其抽象方法是execute()，该方法是在run方法中被调用的，这也就意味着NamedRunnable是一个任务，并且其子类应该实现execute方法。下面再看AsyncCall的实现：

final class AsyncCall extends NamedRunnable {  
 private final Callback responseCallback;  
  
 AsyncCall(Callback responseCallback) {  
 super("OkHttp %s", redactedUrl());  
 this.responseCallback = responseCallback;  
 }  
  
 ......  
final class RealCall implements Call {  
 @Override protected void execute() {  
 boolean signalledCallback = false;  
 try {  
 Response response = getResponseWithInterceptorChain();  
 if (retryAndFollowUpInterceptor.isCanceled()) {  
 signalledCallback = true;  
 responseCallback.onFailure(RealCall.this, new IOException("Canceled"));  
 } else {  
 signalledCallback = true;  
 responseCallback.onResponse(RealCall.this, response);  
 }  
 } catch (IOException e) {  
 ......  
 responseCallback.onFailure(RealCall.this, e);  
  
} finally {  
 client.dispatcher().finished(this);  
 }  
}

AsyncCall实现了execute方法，首先是调用getResponseWithInterceptorChain()方法获取响应，然后获取成功后，就调用回调的onReponse方法，如果失败，就调用回调的onFailure方法。最后，调用Dispatcher的finished方法。

关键代码：

responseCallback.onFailure(RealCall.this, new IOException("Canceled"));

和

responseCallback.onResponse(RealCall.this, response);

走完这两句代码会进行回调到刚刚我们初始化Okhttp的地方,如下：

okHttpClient.newCall(request).enqueue(new Callback() {  
 @Override  
 public void onFailure(Call call, IOException e) {  
  
 }  
  
 @Override  
 public void onResponse(Call call, Response response) throws IOException {  
  
 }  
});

### 核心重点类Dispatcher线程池介绍

public final class Dispatcher {  
 /\*\* 最大并发请求数为64 \*/  
 private int maxRequests = 64;  
 /\*\* 每个主机最大请求数为5 \*/  
 private int maxRequestsPerHost = 5;  
  
 /\*\* 线程池 \*/  
 private ExecutorService executorService;  
  
 /\*\* 准备执行的请求 \*/  
 private final Deque<AsyncCall> readyAsyncCalls = new ArrayDeque<>();  
  
 /\*\* 正在执行的异步请求，包含已经取消但未执行完的请求 \*/  
 private final Deque<AsyncCall> runningAsyncCalls = new ArrayDeque<>();  
  
 /\*\* 正在执行的同步请求，包含已经取消单未执行完的请求 \*/  
 private final Deque<RealCall> runningSyncCalls = new ArrayDeque<>();

在OkHttp，使用如下构造了单例线程池

public synchronized ExecutorService executorService() {  
 if (executorService == null) {  
 executorService = new ThreadPoolExecutor(0, Integer.MAX\_VALUE, 60, TimeUnit.SECONDS,  
 new SynchronousQueue<Runnable>(), Util.threadFactory("OkHttp Dispatcher", false));  
 }  
 return executorService;  
 }

构造一个线程池ExecutorService：

executorService = new ThreadPoolExecutor(  
//corePoolSize 最小并发线程数,如果是0的话，空闲一段时间后所有线程将全部被销毁  
 0,   
//maximumPoolSize: 最大线程数，当任务进来时可以扩充的线程最大值，当大于了这个值就会根据丢弃处理机制来处理  
 Integer.MAX\_VALUE,   
//keepAliveTime: 当线程数大于corePoolSize时，多余的空闲线程的最大存活时间  
 60,   
//单位秒  
 TimeUnit.SECONDS,  
//工作队列,先进先出  
 new SynchronousQueue<Runnable>(),   
//单个线程的工厂   
 Util.threadFactory("OkHttp Dispatcher", false));

可以看出，在Okhttp中，构建了一个核心为[0, Integer.MAX\_VALUE]的线程池，它不保留任何最小线程数，随时创建更多的线程数，当线程空闲时只能活60秒，它使用了一个不存储元素的阻塞工作队列，一个叫做"OkHttp Dispatcher"的线程工厂。

也就是说，在实际运行中，当收到10个并发请求时，线程池会创建十个线程，当工作完成后，线程池会在60s后相继关闭所有线程。

synchronized void enqueue(AsyncCall call) {  
 if (runningAsyncCalls.size() < maxRequests && runningCallsForHost(call) < maxRequestsPerHost) {  
 runningAsyncCalls.add(call);  
 executorService().execute(call);  
 } else {  
 readyAsyncCalls.add(call);  
 }  
 }

从上述源码分析，如果当前还能执行一个并发请求，则加入 runningAsyncCalls ，立即执行，否则加入 readyAsyncCalls 队列。

#### Dispatcher线程池总结

1）调度线程池Disptcher实现了高并发，低阻塞的实现 2）采用Deque作为缓存，先进先出的顺序执行 3）任务在try/finally中调用了finished函数，控制任务队列的执行顺序，而不是采用锁，减少了编码复杂性提高性能

这里是分析OkHttp源码，并不详细讲线程池原理，如对线程池不了解请参考如下链接

[点我，线程池原理，在文章性能优化最后有视频对线程池原理讲解](http://www.jianshu.com/p/c22398f8587f)

try {  
 Response response = getResponseWithInterceptorChain();  
 if (retryAndFollowUpInterceptor.isCanceled()) {  
 signalledCallback = true;  
 responseCallback.onFailure(RealCall.this, new IOException("Canceled"));  
 } else {  
 signalledCallback = true;  
 responseCallback.onResponse(RealCall.this, response);  
 }  
 } finally {  
 client.dispatcher().finished(this);  
 }

当任务执行完成后，无论是否有异常，finally代码段总会被执行，也就是会调用Dispatcher的finished函数

void finished(AsyncCall call) {  
 finished(runningAsyncCalls, call, true);  
 }

从上面的代码可以看出，第一个参数传入的是正在运行的异步队列，第三个参数为true，下面再看有是三个参数的finished方法：

private <T> void finished(Deque<T> calls, T call, boolean promoteCalls) {  
 int runningCallsCount;  
 Runnable idleCallback;  
 synchronized (this) {  
 if (!calls.remove(call)) throw new AssertionError("Call wasn't in-flight!");  
 if (promoteCalls) promoteCalls();  
 runningCallsCount = runningCallsCount();  
 idleCallback = this.idleCallback;  
 }  
  
 if (runningCallsCount == 0 && idleCallback != null) {  
 idleCallback.run();  
 }  
 }

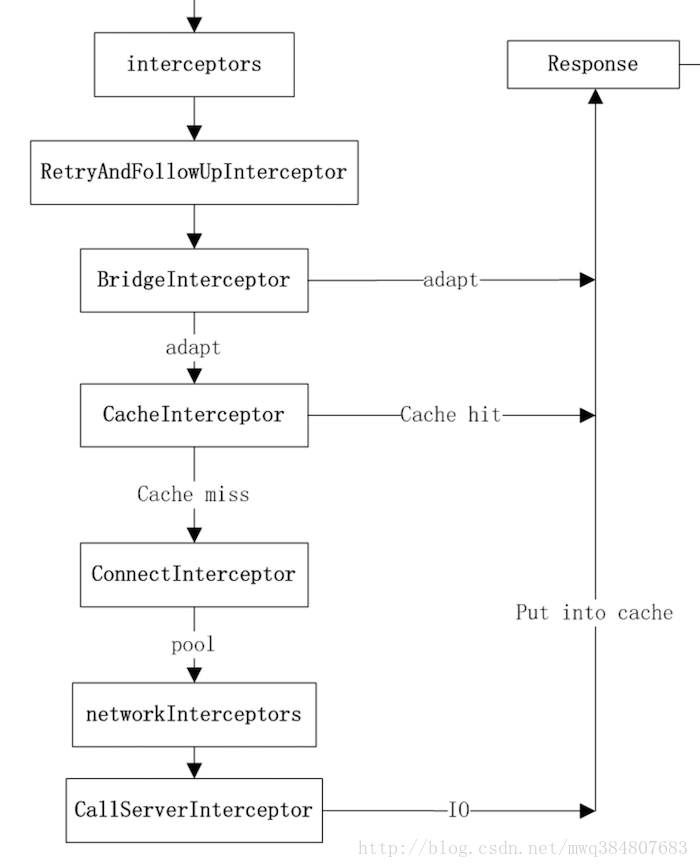
打开源码，发现它将正在运行的任务Call从队列runningAsyncCalls中移除后，获取运行数量判断是否进入了Idle状态,接着执行promoteCalls()函数,下面是promoteCalls()方法：

private void promoteCalls() {  
 if (runningAsyncCalls.size() >= maxRequests) return; // Already running max capacity.  
 if (readyAsyncCalls.isEmpty()) return; // No ready calls to promote.  
  
 for (Iterator<AsyncCall> i = readyAsyncCalls.iterator(); i.hasNext(); ) {  
 AsyncCall call = i.next();  
  
 if (runningCallsForHost(call) < maxRequestsPerHost) {  
 i.remove();  
 runningAsyncCalls.add(call);  
 executorService().execute(call);  
 }  
  
 if (runningAsyncCalls.size() >= maxRequests) return; // Reached max capacity.  
 }  
 }

主要就是遍历等待队列，并且需要满足同一主机的请求小于maxRequestsPerHost时，就移到运行队列中并交给线程池运行。就主动的把缓存队列向前走了一步，而没有使用互斥锁等复杂编码

### 核心重点getResponseWithInterceptorChain方法

Response getResponseWithInterceptorChain() throws IOException {  
 // Build a full stack of interceptors.  
 List<Interceptor> interceptors = new ArrayList<>();  
 interceptors.addAll(client.interceptors());  
 interceptors.add(retryAndFollowUpInterceptor);  
 interceptors.add(new BridgeInterceptor(client.cookieJar()));  
 interceptors.add(new CacheInterceptor(client.internalCache()));  
 interceptors.add(new ConnectInterceptor(client));  
 if (!forWebSocket) {  
 interceptors.addAll(client.networkInterceptors());  
 }  
 interceptors.add(new CallServerInterceptor(forWebSocket));  
  
 Interceptor.Chain chain = new RealInterceptorChain(  
 interceptors, null, null, null, 0, originalRequest);  
 return chain.proceed(originalRequest);  
 }



img

1）在配置 OkHttpClient 时设置的 interceptors； 2）负责失败重试以及重定向的 RetryAndFollowUpInterceptor； 3）负责把用户构造的请求转换为发送到服务器的请求、把服务器返回的响应转换为用户友好的响应的 BridgeInterceptor； 4）负责读取缓存直接返回、更新缓存的 CacheInterceptor； 5）负责和服务器建立连接的 ConnectInterceptor； 6）配置 OkHttpClient 时设置的 networkInterceptors； 7）负责向服务器发送请求数据、从服务器读取响应数据的 CallServerInterceptor。

OkHttp的这种拦截器链采用的是责任链模式，这样的好处是将请求的发送和处理分开，并且可以动态添加中间的处理方实现对请求的处理、短路等操作。

从上述源码得知，不管okhttp有多少拦截器最后都会走，如下方法：

Interceptor.Chain chain = new RealInterceptorChain(  
 interceptors, null, null, null, 0, originalRequest);  
return chain.proceed(originalRequest);

从方法名字基本可以猜到是干嘛的，调用 chain.proceed(originalRequest); 将request传递进来，从拦截器链里拿到返回结果。那么拦截器Interceptor是干嘛的，Chain是干嘛的呢？继续往下看RealInterceptorChain

RealInterceptorChain类

下面是RealInterceptorChain的定义，该类实现了Chain接口，在getResponseWithInterceptorChain调用时好几个参数都传的null。

public final class RealInterceptorChain implements Interceptor.Chain {  
  
 public RealInterceptorChain(List<Interceptor> interceptors, StreamAllocation streamAllocation,  
 HttpCodec httpCodec, RealConnection connection, int index, Request request) {  
 this.interceptors = interceptors;  
 this.connection = connection;  
 this.streamAllocation = streamAllocation;  
 this.httpCodec = httpCodec;  
 this.index = index;  
 this.request = request;  
 }  
 ......  
  
 @Override   
 public Response proceed(Request request) throws IOException {  
 return proceed(request, streamAllocation, httpCodec, connection);  
 }  
  
 public Response proceed(Request request, StreamAllocation streamAllocation, HttpCodec httpCodec,  
 RealConnection connection) throws IOException {  
 if (index >= interceptors.size()) throw new AssertionError();  
  
 calls++;  
  
 ......  
  
 // Call the next interceptor in the chain.  
 RealInterceptorChain next = new RealInterceptorChain(  
 interceptors, streamAllocation, httpCodec, connection, index + 1, request);  
 Interceptor interceptor = interceptors.get(index);  
 Response response = interceptor.intercept(next);  
  
 ......  
  
 return response;  
 }  
  
 protected abstract void execute();  
}

主要看proceed方法，proceed方法中判断index（此时为0）是否大于或者等于client.interceptors(List )的大小。由于httpStream为null，所以首先创建next拦截器链，主需要把索引置为index+1即可；然后获取第一个拦截器，调用其intercept方法。

Interceptor 代码如下：

public interface Interceptor {  
 Response intercept(Chain chain) throws IOException;  
  
 interface Chain {  
 Request request();  
  
 Response proceed(Request request) throws IOException;  
  
 Connection connection();  
 }  
}

BridgeInterceptor

BridgeInterceptor从用户的请求构建网络请求，然后提交给网络，最后从网络响应中提取出用户响应。从最上面的图可以看出，BridgeInterceptor实现了适配的功能。下面是其intercept方法：

public final class BridgeInterceptor implements Interceptor {  
 ......  
  
@Override   
public Response intercept(Chain chain) throws IOException {  
 Request userRequest = chain.request();  
 Request.Builder requestBuilder = userRequest.newBuilder();  
  
 RequestBody body = userRequest.body();  
 //如果存在请求主体部分，那么需要添加Content-Type、Content-Length首部  
 if (body != null) {  
 MediaType contentType = body.contentType();  
 if (contentType != null) {  
 requestBuilder.header("Content-Type", contentType.toString());  
 }  
  
 long contentLength = body.contentLength();  
 if (contentLength != -1) {  
 requestBuilder.header("Content-Length", Long.toString(contentLength));  
 requestBuilder.removeHeader("Transfer-Encoding");  
 } else {  
 requestBuilder.header("Transfer-Encoding", "chunked");  
 requestBuilder.removeHeader("Content-Length");  
 }  
 }  
  
 if (userRequest.header("Host") == null) {  
 requestBuilder.header("Host", hostHeader(userRequest.url(), false));  
 }  
  
 if (userRequest.header("Connection") == null) {  
 requestBuilder.header("Connection", "Keep-Alive");  
 }  
  
 // If we add an "Accept-Encoding: gzip" header field we're responsible for also decompressing  
 // the transfer stream.  
 boolean transparentGzip = false;  
 if (userRequest.header("Accept-Encoding") == null && userRequest.header("Range") == null) {  
 transparentGzip = true;  
 requestBuilder.header("Accept-Encoding", "gzip");  
 }  
  
 List<Cookie> cookies = cookieJar.loadForRequest(userRequest.url());  
 if (!cookies.isEmpty()) {  
 requestBuilder.header("Cookie", cookieHeader(cookies));  
 }  
  
 if (userRequest.header("User-Agent") == null) {  
 requestBuilder.header("User-Agent", Version.userAgent());  
 }  
  
Response networkResponse = chain.proceed(requestBuilder.build());  
  
HttpHeaders.receiveHeaders(cookieJar, userRequest.url(), networkResponse.headers());  
  
Response.Builder responseBuilder = networkResponse.newBuilder()  
 .request(userRequest);  
  
 if (transparentGzip  
 && "gzip".equalsIgnoreCase(networkResponse.header("Content-Encoding"))  
 && HttpHeaders.hasBody(networkResponse)) {  
 GzipSource responseBody = new GzipSource(networkResponse.body().source());  
 Headers strippedHeaders = networkResponse.headers().newBuilder()  
 .removeAll("Content-Encoding")  
 .removeAll("Content-Length")  
 .build();  
 responseBuilder.headers(strippedHeaders);  
 responseBuilder.body(new RealResponseBody(strippedHeaders, Okio.buffer(responseBody)));  
 }  
  
 return responseBuilder.build();  
 }  
  
 /\*\* Returns a 'Cookie' HTTP request header with all cookies, like {@code a=b; c=d}. \*/  
 private String cookieHeader(List<Cookie> cookies) {  
 StringBuilder cookieHeader = new StringBuilder();  
 for (int i = 0, size = cookies.size(); i < size; i++) {  
 if (i > 0) {  
 cookieHeader.append("; ");  
 }  
 Cookie cookie = cookies.get(i);  
 cookieHeader.append(cookie.name()).append('=').append(cookie.value());  
 }  
 return cookieHeader.toString();  
 }  
}

从上面的代码可以看出，首先获取原请求，然后在请求中添加头，比如Host、Connection、Accept-Encoding参数等，然后根据看是否需要填充Cookie，在对原始请求做出处理后，使用chain的procced方法得到响应，接下来对响应做处理得到用户响应，最后返回响应。接下来再看下一个拦截器ConnectInterceptor的处理。

public final class ConnectInterceptor implements Interceptor {  
 ......  
  
 @Override   
 public Response intercept(Chain chain) throws IOException {  
 RealInterceptorChain realChain = (RealInterceptorChain) chain;  
Request request = realChain.request();  
StreamAllocation streamAllocation = realChain.streamAllocation();  
  
 // We need the network to satisfy this request. Possibly for validating a conditional GET.  
 boolean doExtensiveHealthChecks = !request.method().equals("GET");  
 HttpCodec httpCodec = streamAllocation.newStream(client, doExtensiveHealthChecks);  
 RealConnection connection = streamAllocation.connection();  
  
 return realChain.proceed(request, streamAllocation, httpCodec, connection);  
 }  
}

实际上建立连接就是创建了一个 HttpCodec 对象，它利用 Okio 对 Socket 的读写操作进行封装，Okio 以后有机会再进行分析，现在让我们对它们保持一个简单地认识：它对 java.io 和 java.nio 进行了封装，让我们更便捷高效的进行 IO 操作。

CallServerInterceptor

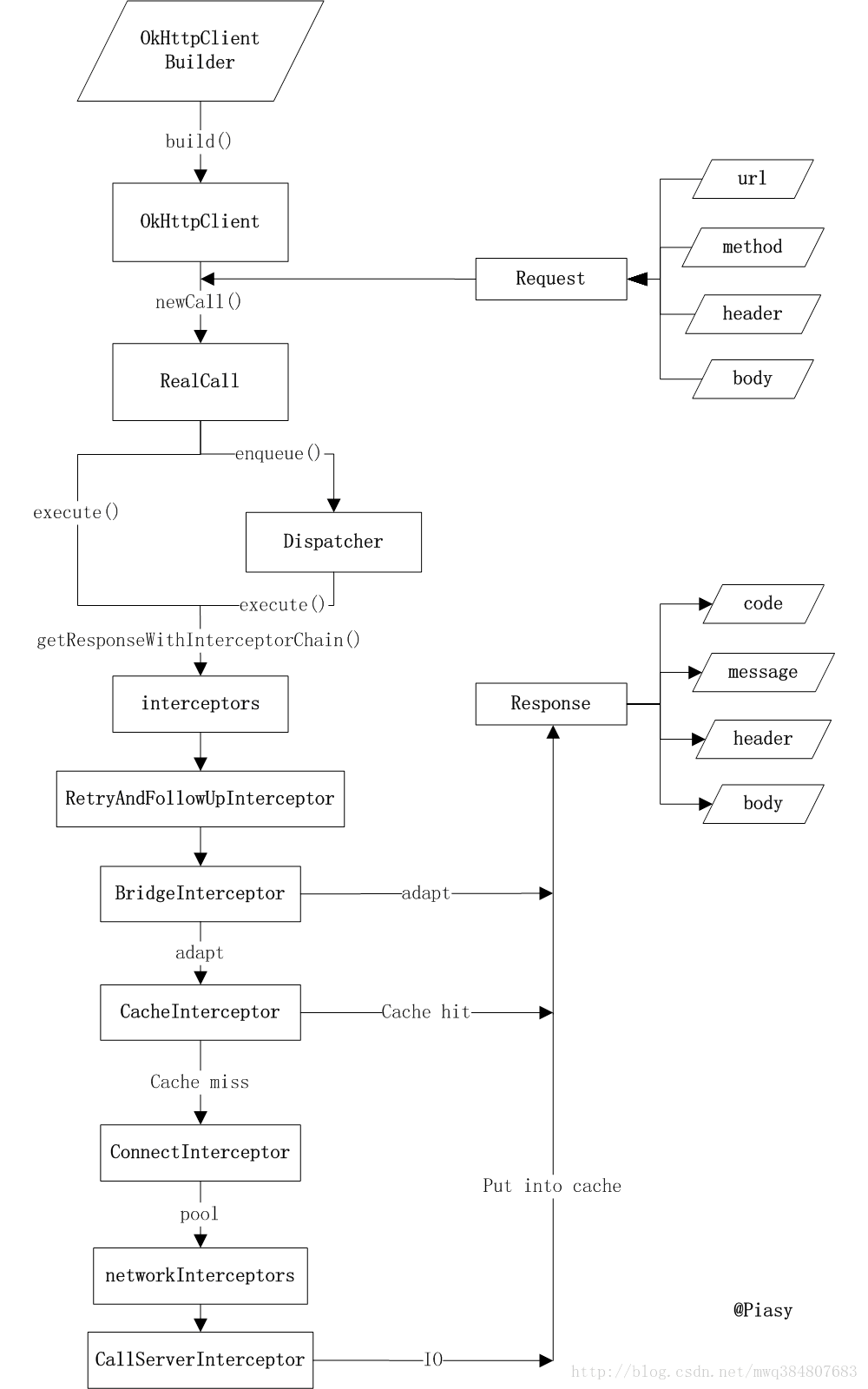
CallServerInterceptor是拦截器链中最后一个拦截器，负责将网络请求提交给服务器。它的intercept方法实现如下：

@Override   
public Response intercept(Chain chain) throws IOException {  
 RealInterceptorChain realChain = (RealInterceptorChain) chain;  
 HttpCodec httpCodec = realChain.httpStream();  
 StreamAllocation streamAllocation = realChain.streamAllocation();  
 RealConnection connection = (RealConnection) realChain.connection();  
 Request request = realChain.request();  
  
 long sentRequestMillis = System.currentTimeMillis();  
 httpCodec.writeRequestHeaders(request);  
  
 Response.Builder responseBuilder = null;  
 if (HttpMethod.permitsRequestBody(request.method()) && request.body() != null) {  
 // If there's a "Expect: 100-continue" header on the request, wait for a "HTTP/1.1 100  
 // Continue" response before transmitting the request body. If we don't get that, return what  
 // we did get (such as a 4xx response) without ever transmitting the request body.  
 if ("100-continue".equalsIgnoreCase(request.header("Expect"))) {  
 httpCodec.flushRequest();  
 responseBuilder = httpCodec.readResponseHeaders(true);  
 }  
  
 if (responseBuilder == null) {  
 // Write the request body if the "Expect: 100-continue" expectation was met.  
 Sink requestBodyOut = httpCodec.createRequestBody(request, request.body().contentLength());  
 BufferedSink bufferedRequestBody = Okio.buffer(requestBodyOut);  
 request.body().writeTo(bufferedRequestBody);  
 bufferedRequestBody.close();  
 } else if (!connection.isMultiplexed()) {  
 // If the "Expect: 100-continue" expectation wasn't met, prevent the HTTP/1 connection from  
 // being reused. Otherwise we're still obligated to transmit the request body to leave the  
 // connection in a consistent state.  
 streamAllocation.noNewStreams();  
 }  
 }  
  
 httpCodec.finishRequest();  
  
 if (responseBuilder == null) {  
 responseBuilder = httpCodec.readResponseHeaders(false);  
 }  
  
 Response response = responseBuilder  
 .request(request)  
 .handshake(streamAllocation.connection().handshake())  
 .sentRequestAtMillis(sentRequestMillis)  
 .receivedResponseAtMillis(System.currentTimeMillis())  
 .build();  
  
 int code = response.code();  
 if (forWebSocket && code == 101) {  
 // Connection is upgrading, but we need to ensure interceptors see a non-null response body.  
 response = response.newBuilder()  
 .body(Util.EMPTY\_RESPONSE)  
 .build();  
 } else {  
 response = response.newBuilder()  
 .body(httpCodec.openResponseBody(response))  
 .build();  
 }  
  
 if ("close".equalsIgnoreCase(response.request().header("Connection"))  
 || "close".equalsIgnoreCase(response.header("Connection"))) {  
 streamAllocation.noNewStreams();  
 }  
  
 if ((code == 204 || code == 205) && response.body().contentLength() > 0) {  
 throw new ProtocolException(  
 "HTTP " + code + " had non-zero Content-Length: " + response.body().contentLength());  
 }  
  
 return response;  
 }

从上面的代码中可以看出，首先获取HttpStream对象，然后调用writeRequestHeaders方法写入请求的头部，然后判断是否需要写入请求的body部分，最后调用finishRequest()方法将所有数据刷新给底层的Socket，接下来尝试调用readResponseHeaders()方法读取响应的头部，然后再调用openResponseBody()方法得到响应的body部分，最后返回响应。

### 最后总结

OkHttp的底层是通过Java的Socket发送HTTP请求与接受响应的(这也好理解，HTTP就是基于TCP协议的)，但是OkHttp实现了连接池的概念，即对于同一主机的多个请求，其实可以公用一个Socket连接，而不是每次发送完HTTP请求就关闭底层的Socket，这样就实现了连接池的概念。而OkHttp对Socket的读写操作使用的OkIo库进行了一层封装。



img

* 欢迎关注微信公众号,长期推荐技术文章和技术视频

微信公众号名称：Android干货程序员

img

img