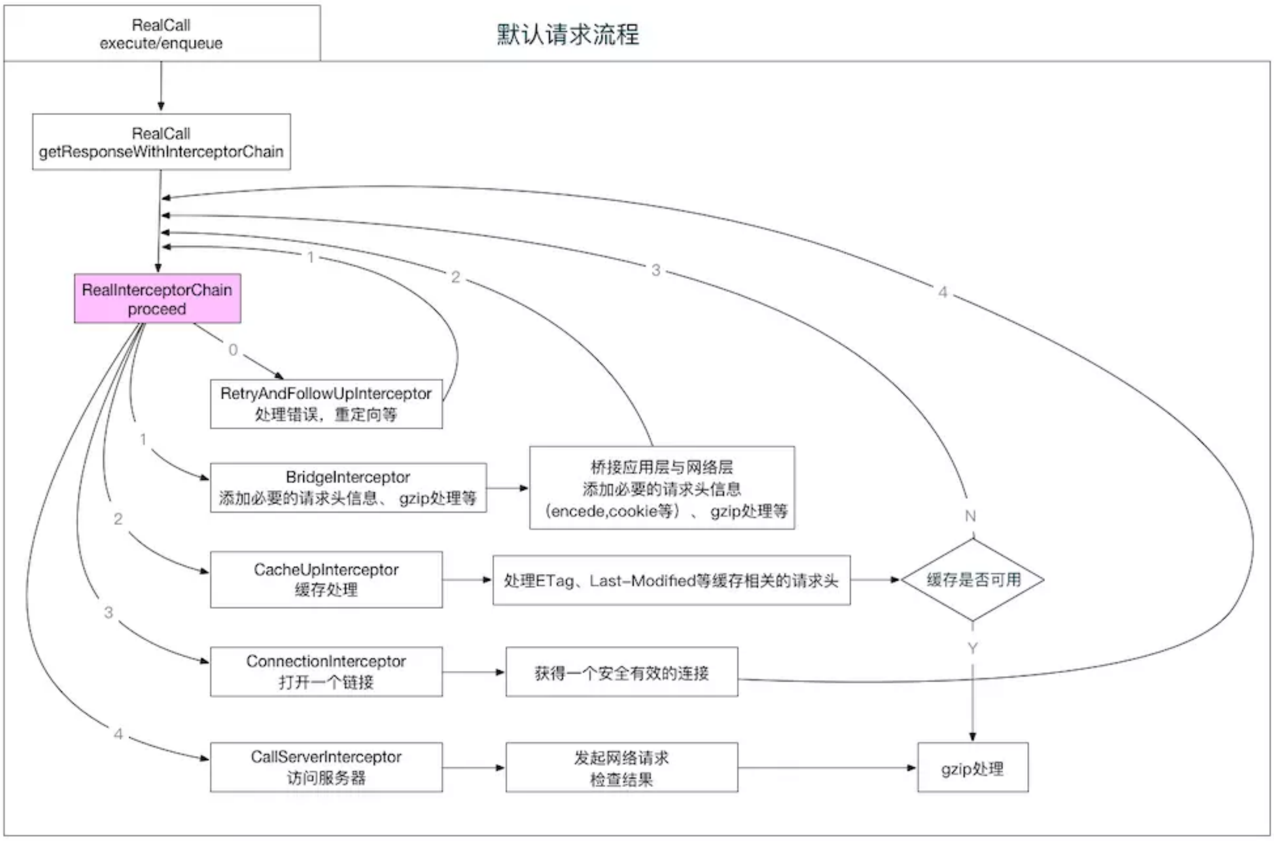
Okhttp3源码分析

(/u/a04b98b0f913)
Jdqm (/u/a04b98b0f913) ＋关注

2018.03.29 10:06* 字数 1435 阅读 4807评论 1喜欢 12

在OkHttp3中，其灵活性很大程度上体现在可以 `intercept` 其任意一个环节，而这个优势便是okhttp3整个请求响应架构体系的精髓所在，先放出一张主框架请求流程图，接着再分析源码。



Okhttp请求流程

```
String url = "http://wwww.baidu.com";
OkHttpClient okHttpClient = new OkHttpClient();
final Request request = new Request.Builder()
        .url(url)
        .build();
Call call = okHttpClient.newCall(request);
call.enqueue(new Callback() {
    @Override
    public void onFailure(Call call, IOException e) {
        Log.d(TAG, "onFailure: ");
    }

    @Override
    public void onResponse(Call call, Response response) throws IOException {
        Log.d(TAG, "onResponse: " + response.body().string());
    }
});
```

这大概是一个最简单的一个例子了，在 `new OkHttpClient()` 内部使用构造器模式初始化了一些配置信息：支持协议、任务分发器（其内部包含一个线程池，执行异步请求）、连接池(其内部包含一个线程池，维护connection)、连接/读/写超时时长等信息。

```
public Builder() {
    dispatcher = new Dispatcher(); //任务调度器
    protocols = DEFAULT_PROTOCOLS; //支持的协议
    connectionSpecs = DEFAULT_CONNECTION_SPECS;
    eventListenerFactory = EventListener.factory(EventListener.NONE);
    proxySelector = ProxySelector.getDefault();
    cookieJar = CookieJar.NO_COOKIES;
    socketFactory = SocketFactory.getDefault();
    hostnameVerifier = OkHostnameVerifier.INSTANCE;
    certificatePinner = CertificatePinner.DEFAULT;
    proxyAuthenticator = Authenticator.NONE;
    authenticator = Authenticator.NONE;
    connectionPool = new ConnectionPool(); //连接池
    dns = Dns.SYSTEM;
    followSslRedirects = true;
    followRedirects = true;
    retryOnConnectionFailure = true;
    connectTimeout = 10_000;//超时时间
    readTimeout = 10_000;
    writeTimeout = 10_000;
    pingInterval = 0;
}
```

第一行创建了一个 `Dispatcher` 任务调度器，它定义了三个双向任务队列，两个异步队列：准备执行的请求队列 `readyAsyncCalls` 、正在运行的请求队列 `runningAsyncCalls` ；一个正在运行的同步请求队列 `runningSyncCalls` ；

```
public final class Dispatcher {
    private int maxRequests = 64; //最大请求数量
    private int maxRequestsPerHost = 5; //每台主机最大的请求数量
    private @Nullable Runnable idleCallback;

    /** Executes calls. Created lazily. */
    private @Nullable ExecutorService executorService; //线程池

    /** Ready async calls in the order they'll be run. */
    private final Deque<AsyncCall> readyAsyncCalls = new ArrayDeque<>();

    /** Running asynchronous calls. Includes canceled calls that haven't finished yet. */
    private final Deque<AsyncCall> runningAsyncCalls = new ArrayDeque<>();

    /** Running synchronous calls. Includes canceled calls that haven't finished yet. */
    private final Deque<RealCall> runningSyncCalls = new ArrayDeque<>();

    /** 这个线程池没有核心线程，线程数量没有限制，空闲60s就会回收*/
    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;
    }
}
```

另外还有一个线程池 `executorService` ，这个线程池跟Android中的CachedThreadPool非常类似，这种类型的线程池，适用于大量的耗时较短的异步任务。下一篇文章(https://www.jianshu.com/p/9deec36f2759) 将对OkHttp框架中的线程池做一个总结。

接下来接着看Request的构造，这个例子Request比较简单，指定了请求方式 `GET` 和请求 `url`

```
  public static class Builder {
    HttpUrl url;
    String method;
    Headers.Builder headers;
    RequestBody body;
    Object tag;

    public Builder() {
      this.method = "GET";
      this.headers = new Headers.Builder();
    }

    public Builder url(HttpUrl url) {
      if (url == null) throw new NullPointerException("url == null");
      this.url = url;
      return this;
    }
    public Request build() {
        if (url == null) throw new IllegalStateException("url == null");
        return new Request(this);
    }
    ...
}
```

紧接着通过 `OkHttpClient` 和 `Request` 构造一个 `Call` 对象，它的实现是 `RealCall`

```
public Call newCall(Request request) {
    return RealCall.newRealCall(this, request, false /* for web socket */);
}

static RealCall newRealCall(OkHttpClient client, Request originalRequest, boolean forWebSocket){
    // Safely publish the Call instance to the EventListener.
    RealCall call = new RealCall(client, originalRequest, forWebSocket);
    call.eventListener = client.eventListenerFactory().create(call);
    return call;
}

private RealCall(OkHttpClient client, Request originalRequest, boolean forWebSocket) {
    this.client = client;
    this.originalRequest = originalRequest;
    this.forWebSocket = forWebSocket;
    this.retryAndFollowUpInterceptor = new RetryAndFollowUpInterceptor(client, forWebSocket);
}
```

可以看到在 `RealCall` 的构造方法中创建了一个 `RetryAndFollowUpInterceptor` ，用于处理请求错误和重定向等，这是 `Okhttp` 框架的精髓 `interceptor chain` 中的一环，默认情况下也是第一个拦截器，除非调用 `OkHttpClient.Builder#addInterceptor(Interceptor)` 来添加全局的拦截器。关于拦截器链的顺序参见 `RealCall#getResponseWithInterceptorChain()` 方法。

RealCall#enqueue(Callback)

```
public void enqueue(Callback responseCallback) {
    synchronized (this) {
        //每个请求只能之执行一次
        if (executed) throw new IllegalStateException("Already Executed");
        executed = true;
    }
    captureCallStackTrace();
    eventListener.callStart(this);
    client.dispatcher().enqueue(new AsyncCall(responseCallback));
}
```

可以看到，一个 `Call` 只能执行一次，否则会抛异常，这里创建了一个 `AsyncCall` 并将Callback传入，接着再交给任务分发器 `Dispatcher` 来进一步处理。

```
synchronized void enqueue(AsyncCall call) {
    //正在执行的任务数量小于最大值（64），并且此任务所属主机的正在执行任务小于最大值（5）
    if (runningAsyncCalls.size() < maxRequests && runningCallsForHost(call) < maxRequestsPerHost) {
        runningAsyncCalls.add(call);
        executorService().execute(call);
    } else {
        readyAsyncCalls.add(call);
    }
}
```

从 `Dispatcher#enqueue()` 方法的策略可以看出，对于请求的入队做了一些限制，若正在执行的请求数量小于最大值（默认64），并且此请求所属主机的正在执行任务小于最大值（默认5），就加入正在运行的队列并通过线程池来执行该任务，否则加入准备执行队列中。

流程图

现在回头看看 `AsyncCall` ，它继承自 `NamedRunnable` ，而 `NamedRunnable` 实现了 `Runnable` 接口，它的作用有2个：
①采用模板方法的设计模式，让子类将具体的操作放在 `execute()` 方法中;
②给线程指定一个名字，比如传入模块名称，方便监控线程的活动状态；

```
public abstract class NamedRunnable implements Runnable {
  protected final String name;

  public NamedRunnable(String format, Object... args) {
    this.name = Util.format(format, args);
  }

  @Override public final void run() {
    String oldName = Thread.currentThread().getName();
    Thread.currentThread().setName(name);
    try {
      //采用模板方法让子类将具体的操作放到此execute()方法
      execute();
    } finally {
      Thread.currentThread().setName(oldName);
    }
  }

  protected abstract void execute();
}
```

```
final class AsyncCall extends NamedRunnable {
    //省略...
    @Override protected void execute() {
      boolean signalledCallback = false;
      try {
        //调用 getResponseWithInterceptorChain()获得响应内容
        Response response = getResponseWithInterceptorChain(); //①
        if (retryAndFollowUpInterceptor.isCanceled()) {
          //这个标记为主要是避免异常时2次回调
          signalledCallback = true;
          //回调Callback告知失败
          responseCallback.onFailure(RealCall.this, new IOException("Canceled"));
        } else {
          signalledCallback = true;
          //回调Callback，将响应内容传回去
          responseCallback.onResponse(RealCall.this, response);
        }
      } catch (IOException e) {
        if (signalledCallback) {
          // Do not signal the callback twice!
          Platform.get().log(INFO, "Callback failure for " + toLoggableString(), e);
        } else {
          eventListener.callFailed(RealCall.this, e);
          responseCallback.onFailure(RealCall.this, e);
        }
      } finally {
        //不管请求成功与否，都进行finished()操作
        client.dispatcher().finished(this);//②
      }
    }
}
```

先看注释②的行finally块中执行的 `client.dispatcher().finished(this)`

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

private <T> void finished(Deque<T> calls, T call, boolean promoteCalls) {
    int runningCallsCount;
    Runnable idleCallback;
    synchronized (this) {
        //从正在执行的队列中将其移除
        if (!calls.remove(call)) throw new AssertionError("Call wasn't in-flight!");
        if (promoteCalls) promoteCalls(); //推动下一个任务的执行
        runningCallsCount = runningCallsCount();//同步+异步的正在执行任务数量
        idleCallback = this.idleCallback;
    }
    //如果没有正在执行的任务，且idleCallback不为null，则回调通知空闲了
    if (runningCallsCount == 0 && idleCallback != null) {
        idleCallback.run();
    }
}
```

其中 `promoteCalls()` 为推动下一个任务执行，其实它做的也很简单，就是在条件满足的情况下，将 `readyAsyncCalls` 中的任务移动到 `runningAsyncCalls` 中，并交给线程池来执行，以下是它的实现。

```
private void promoteCalls() {
    if (runningAsyncCalls.size() >= maxRequests) return; // Already running max capacity.
    if (readyAsyncCalls.isEmpty()) return; // No ready calls to promote.

    //若条件允许，将readyAsyncCalls中的任务移动到runningAsyncCalls中，并交给线程池执行
    for (Iterator<AsyncCall> i = readyAsyncCalls.iterator(); i.hasNext(); ) {
      AsyncCall call = i.next();

      if (runningCallsForHost(call) < maxRequestsPerHost) {
        i.remove();
        runningAsyncCalls.add(call);
        executorService().execute(call);
      }
      //当runningAsyncCalls满了，直接退出迭代
      if (runningAsyncCalls.size() >= maxRequests) return; // Reached max capacity.
    }
}
```

接下来就回到注释①处的响应内容的获取 `getResponseWithInterceptorChain()`

```
Response getResponseWithInterceptorChain() throws IOException {
    // Build a full stack of interceptors.
    List<Interceptor> interceptors = new ArrayList<>(); //这是一个List，是有序的
    interceptors.addAll(client.interceptors());//首先添加的是用户添加的全局拦截器
    interceptors.add(retryAndFollowUpInterceptor); //错误、重定向拦截器
   //桥接拦截器，桥接应用层与网络层，添加必要的头、
    interceptors.add(new BridgeInterceptor(client.cookieJar()));
    //缓存处理，Last-Modified、ETag、DiskLruCache等
    interceptors.add(new CacheInterceptor(client.internalCache()));
    //连接拦截器
    interceptors.add(new ConnectInterceptor(client));
    //从这就知道，通过okHttpClient.Builder#addNetworkInterceptor()传进来的拦截器只对非网页的请求生效
    if (!forWebSocket) {
      interceptors.addAll(client.networkInterceptors());
    }
    //真正访问服务器的拦截器
    interceptors.add(new CallServerInterceptor(forWebSocket));

    Interceptor.Chain chain = new RealInterceptorChain(interceptors, null, null, null, 0,
        originalRequest, this, eventListener, client.connectTimeoutMillis(),
        client.readTimeoutMillis(), client.writeTimeoutMillis());

    return chain.proceed(originalRequest);
}
```

可以看这块重点就是 `interceptors` 这个集合，首先将前面的 `client.interceptors()` 全部加入其中，还有在创建 `RealCall` 时的 `retryAndFollowUpInterceptor` 加入其中，接着还创建并添加了 `BridgeInterceptor`、`CacheInterceptor`、`ConnectInterceptor`、`CallServerInterceptor` ，最后通过 `RealInterceptorChain#proceed(Request)` 来执行整个 `interceptor chain` ，可见把这个拦截器链搞清楚，整体流程也就明朗了。

RealInterceptorChain#proceed()

```
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 {
    //省略异常处理...

    // Call the next interceptor in the chain.
    RealInterceptorChain next = new RealInterceptorChain(interceptors, streamAllocation, httpCodec,
        connection, index + 1, request, call, eventListener, connectTimeout, readTimeout,
        writeTimeout);
    Interceptor interceptor = interceptors.get(index);
    Response response = interceptor.intercept(next);

    //省略异常处理...
    return response;
}
```

从这段实现可以看出，是按照添加到 `interceptors` 集合的顺序，逐个往下调用拦截器的intercept()方法，所以在前面的拦截器会先被调用。这个例子中自然就是 `RetryAndFollowUpInterceptor` 了。

```
public Response intercept(Chain chain) throws IOException {
    Request request = chain.request();
    RealInterceptorChain realChain = (RealInterceptorChain) chain;
    Call call = realChain.call();
    EventListener eventListener = realChain.eventListener();
    //创建一个StreamAllocation
    StreamAllocation streamAllocation = new StreamAllocation(client.connectionPool(),
        createAddress(request.url()), call, eventListener, callStackTrace);
    this.streamAllocation = streamAllocation;

    //统计重定向次数，不能大于20
    int followUpCount = 0;
    Response priorResponse = null;
    while (true) {
      if (canceled) {
        streamAllocation.release();
        throw new IOException("Canceled");
      }

      Response response;
      boolean releaseConnection = true;
      try {
        //调用下一个interceptor的来获得响应内容
        response = realChain.proceed(request, streamAllocation, null, null);
        releaseConnection = false;
      } catch (RouteException e) {
        // The attempt to connect via a route failed. The request will not have been sent.
        if (!recover(e.getLastConnectException(), streamAllocation, false, request)) {
          throw e.getLastConnectException();
        }
        releaseConnection = false;
        continue;
      } catch (IOException e) {
        // An attempt to communicate with a server failed. The request may have been sent.
        boolean requestSendStarted = !(e instanceof ConnectionShutdownException);
        if (!recover(e, streamAllocation, requestSendStarted, request)) throw e;
        releaseConnection = false;
        continue;
      } finally {
        // We're throwing an unchecked exception. Release any resources.
        if (releaseConnection) {
          streamAllocation.streamFailed(null);
          streamAllocation.release();
        }
      }

      // Attach the prior response if it exists. Such responses never have a body.
      if (priorResponse != null) {
        response = response.newBuilder()
            .priorResponse(priorResponse.newBuilder()
                    .body(null)
                    .build())
            .build();
      }

     //重定向处理
      Request followUp = followUpRequest(response, streamAllocation.route());

      if (followUp == null) {
        if (!forWebSocket) {
          streamAllocation.release();
```

```java
        }
        return response;
      }

      closeQuietly(response.body());

      if (++followUpCount > MAX_FOLLOW_UPS) {
        streamAllocation.release();
        throw new ProtocolException("Too many follow-up requests: " + followUpCount);
      }

      if (followUp.body() instanceof UnrepeatableRequestBody) {
        streamAllocation.release();
        throw new HttpRetryException("Cannot retry streamed HTTP body", response.code());
      }

      if (!sameConnection(response, followUp.url())) {
        streamAllocation.release();
        streamAllocation = new StreamAllocation(client.connectionPool(),
            createAddress(followUp.url()), call, eventListener, callStackTrace);
        this.streamAllocation = streamAllocation;
      } else if (streamAllocation.codec() != null) {
        throw new IllegalStateException("Closing the body of " + response
            + " didn't close its backing stream. Bad interceptor?");
      }

      request = followUp;
      priorResponse = response;
    }
}
```

这个拦截器就如同它的名字 `retry and followUp` ，主要负责错误处理和重定向等问题，比如路由错误、IO异常等。

接下来就到了 `BridgeInterceptor#intercept()` ，在这个拦截器中，添加了必要请求头信息，gzip处理等。

```
public Response intercept(Chain chain) throws IOException {
    Request userRequest = chain.request();
    Request.Builder requestBuilder = userRequest.newBuilder();

    //从这开始给请求添加了一些请求头信息
    RequestBody body = userRequest.body();
    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);
        String contentType = networkResponse.header("Content-Type");
        responseBuilder.body(new RealResponseBody(contentType, -1L, Okio.buffer(responseBody)));
    }

    return responseBuilder.build();
}
```

这个拦截器处理请求信息、cookie、gzip等，接着往下是 `CacheInterceptor`

```
public Response intercept(Chain chain) throws IOException {
    Response cacheCandidate = cache != null
        ? cache.get(chain.request())
        : null;

    long now = System.currentTimeMillis();

    CacheStrategy strategy = new CacheStrategy.Factory(now, chain.request(), cacheCandidate).get();
    Request networkRequest = strategy.networkRequest;
    Response cacheResponse = strategy.cacheResponse;

    if (cache != null) {
      cache.trackResponse(strategy);
    }

    if (cacheCandidate != null && cacheResponse == null) {
      closeQuietly(cacheCandidate.body()); // The cache candidate wasn't applicable. Close it.
    }

    // If we're forbidden from using the network and the cache is insufficient, fail.
    if (networkRequest == null && cacheResponse == null) {
      return new Response.Builder()
          .request(chain.request())
          .protocol(Protocol.HTTP_1_1)
          .code(504)
          .message("Unsatisfiable Request (only-if-cached)")
          .body(Util.EMPTY_RESPONSE)
          .sentRequestAtMillis(-1L)
          .receivedResponseAtMillis(System.currentTimeMillis())
          .build();
    }

    // If we don't need the network, we're done.
    if (networkRequest == null) {
      return cacheResponse.newBuilder()
          .cacheResponse(stripBody(cacheResponse))
          .build();
    }

    Response networkResponse = null;
    try {
      //调用下一个拦截器进行网络请求
      networkResponse = chain.proceed(networkRequest);
    } finally {
      // If we're crashing on I/O or otherwise, don't leak the cache body.
      if (networkResponse == null && cacheCandidate != null) {
        closeQuietly(cacheCandidate.body());
      }
    }

    // If we have a cache response too, then we're doing a conditional get.
    if (cacheResponse != null) {
      if (networkResponse.code() == HTTP_NOT_MODIFIED) {
        Response response = cacheResponse.newBuilder()
            .headers(combine(cacheResponse.headers(), networkResponse.headers()))
            .sentRequestAtMillis(networkResponse.sentRequestAtMillis())
            .receivedResponseAtMillis(networkResponse.receivedResponseAtMillis())
            .cacheResponse(stripBody(cacheResponse))
            .networkResponse(stripBody(networkResponse))
            .build();
        networkResponse.body().close();
```

```
        // Update the cache after combining headers but before stripping the
        // Content-Encoding header (as performed by initContentStream()).
        cache.trackConditionalCacheHit();
        cache.update(cacheResponse, response);
        return response;
      } else {
        closeQuietly(cacheResponse.body());
      }
    }

    Response response = networkResponse.newBuilder()
        .cacheResponse(stripBody(cacheResponse))
        .networkResponse(stripBody(networkResponse))
        .build();

    if (cache != null) {
      if (HttpHeaders.hasBody(response) && CacheStrategy.isCacheable(response, networkRequest)) {
        // Offer this request to the cache.
        CacheRequest cacheRequest = cache.put(response);
        return cacheWritingResponse(cacheRequest, response);
      }

      if (HttpMethod.invalidatesCache(networkRequest.method())) {
        try {
          cache.remove(networkRequest);
        } catch (IOException ignored) {
          // The cache cannot be written.
        }
      }
    }

    return response;
}
```

这个拦截器主要工作是做做缓存处理，如果有有缓存并且缓存可用，那就使用缓存，否则进行调用下一个拦截器 `ConnectionInterceptor` 进行网络请求，并将响应内容缓存。

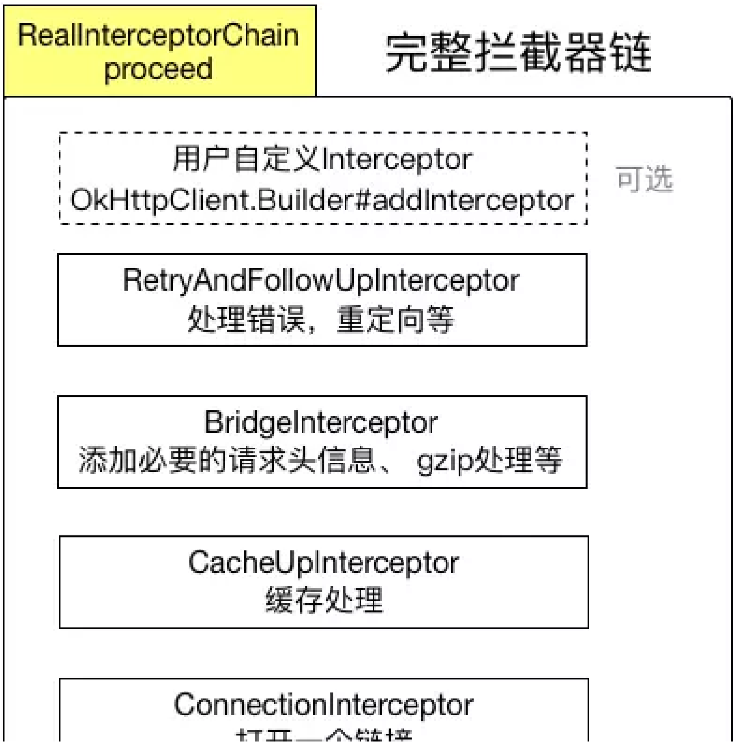
```
public Response intercept(Chain chain) throws IOException {
    RealInterceptorChain realChain = (RealInterceptorChain) chain;
    Request request = realChain.request();
    StreamAllocation streamAllocation = realChain.streamAllocation();

    // 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, chain, doExtensiveHealthChecks);
    RealConnection connection = streamAllocation.connection();

    return realChain.proceed(request, streamAllocation, httpCodec, connection);
}
```

这个拦截器主要是打开一个到目标服务器的 `connection` 并调用下一个拦截器 `CallServerInterceptor` ，这是拦截器链最后一个拦截器，它向服务器发起真正的网络请求。

```java
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();

    realChain.eventListener().requestHeadersStart(realChain.call());
    httpCodec.writeRequestHeaders(request);
    realChain.eventListener().requestHeadersEnd(realChain.call(), 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();
        realChain.eventListener().responseHeadersStart(realChain.call());
        responseBuilder = httpCodec.readResponseHeaders(true);
      }

      if (responseBuilder == null) {
        // Write the request body if the "Expect: 100-continue" expectation was met.
        realChain.eventListener().requestBodyStart(realChain.call());
        long contentLength = request.body().contentLength();
        CountingSink requestBodyOut =
            new CountingSink(httpCodec.createRequestBody(request, contentLength));
        BufferedSink bufferedRequestBody = Okio.buffer(requestBodyOut);

        request.body().writeTo(bufferedRequestBody);
        bufferedRequestBody.close();
        realChain.eventListener()
            .requestBodyEnd(realChain.call(), requestBodyOut.successfulCount);
      } 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) {
      realChain.eventListener().responseHeadersStart(realChain.call());
      responseBuilder = httpCodec.readResponseHeaders(false);
    }

    Response response = responseBuilder
        .request(request)
        .handshake(streamAllocation.connection().handshake())
        .sentRequestAtMillis(sentRequestMillis)
        .receivedResponseAtMillis(System.currentTimeMillis())
        .build();

    int code = response.code();
    if (code == 100) {
      // server sent a 100-continue even though we did not request one.
```

```
      // try again to read the actual response
      responseBuilder = httpCodec.readResponseHeaders(false);

      response = responseBuilder
              .request(request)
              .handshake(streamAllocation.connection().handshake())
              .sentRequestAtMillis(sentRequestMillis)
              .receivedResponseAtMillis(System.currentTimeMillis())
              .build();

      code = response.code();
    }

    realChain.eventListener()
            .responseHeadersEnd(realChain.call(), response);

    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;
  }
```

从上面的请求流程图可以看出，OkHttp的拦截器链可谓是其整个框架的精髓，用户可传入的 interceptor 分为两类：
①一类是全局的 interceptor，该类 interceptor 在整个拦截器链中最早被调用，通过 OkHttpClient.Builder#addInterceptor(Interceptor) 传入；
②另外一类是非网页请求的 interceptor ，这类拦截器只会在非网页请求中被调用，并且是在组装完请求之后，真正发起网络请求前被调用，所有的 interceptor 被保存在 List<Interceptor> interceptors 集合中，按照添加顺序来逐个调用，具体可参考 RealCall#getResponseWithInterceptorChain() 方法。通过 OkHttpClient.Builder#addNetworkInterceptor(Interceptor) 传入；



完整interceptor-chain

相关阅读

1.Okhttp的基本使用 (https://www.jianshu.com/p/da4a806e599b)
2.Okhttp主流程源码分析 (https://www.jianshu.com/p/b0353ed71151)
3.Okhttp3架构分析，主要通过一些流程图类展现 (https://www.jianshu.com/p/9deec36f2759)

关注微信公众号，第一时间接收推送!