(https://yq.aliyun.com/promotion/543)

OkHttp 3.7源码分析（二）——拦截器&一个实际网络请求的实现

李牙刷儿 (/users/1072397808568247)　2017-05-05 10:26:41　浏览4309　评论2

移动开发与客户端 (/tags/type_blog-tagid_8/)　网络与数据通信 (/tags/type_blog-tagid_9/)
java (/tags/type_blog-tagid_41/)　阿里技术协会 (/tags/type_blog-tagid_523/)
源码 (/tags/type_blog-tagid_603/)　Cache (/tags/type_blog-tagid_814/)
request (/tags/type_blog-tagid_1023/)

摘要： 前一篇博客中我们介绍了OkHttp的总体架构，接下来我们以一个具体的网络请求来讲述OkHttp进行网络访问的具体过程。由于该部分与OkHttp的拦截器概念紧密联系在一起，所以将这两部分放在一起进行讲解。

OkHttp3.7源码分析文章列表如下：

- OkHttp源码分析——整体架构 (https://yq.aliyun.com/articles/78105?spm=5176.8091938.0.0.hlEONd)
- OkHttp源码分析——拦截器 (https://yq.aliyun.com/articles/78104?spm=5176.8091938.0.0.hlEONd)
- OkHttp源码分析——任务队列 (https://yq.aliyun.com/articles/78103?spm=5176.8091938.0.0.hlEONd)
- OkHttp源码分析——缓存策略 (https://yq.aliyun.com/articles/78102?spm=5176.8091938.0.0.hlEONd)
- OkHttp源码分析——多路复用 (https://yq.aliyun.com/articles/78101?spm=5176.8091938.0.0.hlEONd)

前一篇博客中我们介绍了OkHttp的总体架构，接下来我们以一个具体的网络请求来讲述OkHttp进行网络访问的具体过程。由于该部分与OkHttp的拦截器概念紧密联系在一起，所以将这两部分放在一起进行讲解。

1.构造Demo

首先构造一个简单的异步网络访问Demo:

```
OkHttpClient client = new OkHttpClient();
Request request = new Request.Builder()
  .url("http://publicobject.com/helloworld.txt")
  .build();

client.newCall(request).enqueue(new Callback() {
  @Override
  public void onFailure(Call call, IOException e) {
    Log.d("OkHttp", "Call Failed:" + e.getMessage());
  }

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

2. 发起请求

`OkHttpClient.newCall` 实际是创建一个 `RealCall` 实例：

```
  /**
   * Prepares the {@code request} to be executed at some point in the future.
   */
  @Override public Call newCall(Request request) {
    return new RealCall(this, request, false /* for web socket */);
  }
```

`RealCall.enqueue` 实际就是讲一个 `RealCall` 放入到任务队列中，等待合适的机会执行:

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

从代码中可以看到最终 `RealCall` 被转化成一个 `AsyncCall` 并被放入到任务队列中，任务队列中的分发逻辑这里先不说，相关实现会放在[OkHttp源码分析——任务队列]()疑问进行介绍。这里只需要知道AsyncCall的excute方法最终将会被执行:

```
[RealCall.java]
@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) {
        if (signalledCallback) {
          // Do not signal the callback twice!
          Platform.get().log(INFO, "Callback failure for " + toLoggableString(), e);
        } else {
          responseCallback.onFailure(RealCall.this, e);
        }
      } finally {
        client.dispatcher().finished(this);
      }
    }
  }
```

`execute` 方法的逻辑并不复杂，简单的说就是：

- 调用 `getResponseWithInterceptorChain` 获取服务器返回
- 通知任务分发器(`client.dispatcher`)该任务已结束

`getResponseWithInterceptorChain` 构建了一个拦截器链，通过依次执行该拦截器链中的每一个拦截器最终得到服务器返回。

3. 构建拦截器链

首先来看下 `getResponseWithInterceptorChain` 的实现：

```
[RealCall.java]
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);
  }
```

其逻辑大致分为两部分：

- 创建一系列拦截器，并将其放入一个拦截器数组中。这部分拦截器即包括用户自定义的拦截器也包括框架内部拦截器
- 创建一个拦截器链 `RealInterceptorChain` ,并执行拦截器链的 `proceed` 方法

接下来看下 `RealInterceptorChain` 的实现逻辑：

```
[RealInterceptorChain.java]
public final class RealInterceptorChain implements Interceptor.Chain {
  private final List<Interceptor> interceptors;
  private final StreamAllocation streamAllocation;
  private final HttpCodec httpCodec;
  private final RealConnection connection;
  private final int index;
  private final Request request;
  private int calls;

  public RealInterceptorChain(List<Interceptor> interceptors, StreamAllocation streamAlloca
                              HttpCodec httpCodec, RealConnection connection, int index, Re
    this.interceptors = interceptors;
    this.connection = connection;
    this.streamAllocation = streamAllocation;
    this.httpCodec = httpCodec;
    this.index = index;
    this.request = request;
  }

  @Override public Connection connection() {
    return connection;
  }

  public StreamAllocation streamAllocation() {
    return streamAllocation;
  }

  public HttpCodec httpStream() {
    return httpCodec;
  }

  @Override public Request request() {
    return request;
  }

  @Override public Response proceed(Request request) throws IOException {
    return proceed(request, streamAllocation, httpCodec, connection);
  }

  public Response proceed(Request request, StreamAllocation streamAllocation, HttpCodec htt
      RealConnection connection) throws IOException {

    ......
    // 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;
  }
}
```

在 `proceed` 方法中的核心代码可以看到，proceed实际上也做了两件事：

- 创建下一个拦截链。传入 `index + 1` 使得下一个拦截器链只能从下一个拦截器开始访问
- 执行索引为 `index` 的intercept方法，并将下一个拦截器链传入该方法

接下来再看下第一个拦截器RetryAndFollowUpInterceptor的intercept方法：

```
[RetryAndFollowUpInterceptor.java]

public final class RetryAndFollowUpInterceptor implements Interceptor {
    @Override public Response intercept(Chain chain) throws IOException {
    Request request = chain.request();

    streamAllocation = new StreamAllocation(
        client.connectionPool(), createAddress(request.url()), callStackTrace);

    int followUpCount = 0;
    Response priorResponse = null;
    while (true) {
      if (canceled) {
        streamAllocation.release();
        throw new IOException("Canceled");
      }

      Response response = null;
      boolean releaseConnection = true;
      try {
       //执行下一个拦截器链的proceed方法
        response = ((RealInterceptorChain) chain).proceed(request, streamAllocation, 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(), 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, 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.
      ......

      Request followUp = followUpRequest(response);


      closeQuietly(response.body());
      ...
    }
  }
}
```

这段代码最关键的代码是:

```
response = ((RealInterceptorChain) chain).proceed(request, streamAllocation, null, null);
```

这行代码就是执行下一个拦截器链的proceed方法。而我们知道在下一个拦截器链中又会执行下一个拦截器的intercept方法。所以整个执行链就在拦截器与拦截器链中交替执行，最终完成所有拦截器的操作。这也是OkHttp拦截器的链式执行逻辑。而一个拦截器的intercept方法所执行的逻辑大致分为三部分：

- 在发起请求前对request进行处理
- 调用下一个拦截器，获取response
- 对response进行处理，返回给上一个拦截器

这就是OkHttp拦截器机制的核心逻辑。所以一个网络请求实际上就是一个个拦截器执行其intercept方法的过程。而这其中除了用户自定义的拦截器外还有几个核心拦截器完成了网络访问的核心逻辑，按照先后顺序依次是：

- RetryAndFollowUpInterceptor
- BridgeInterceptor
- CacheInterceptor
- ConnectIntercetot
- CallServerInterceptor

4 RetryAndFollowUpInterceptor

如上文代码所示，RetryAndFollowUpInterceptor负责两部分逻辑：

- 在网络请求失败后进行重试
- 当服务器返回当前请求需要进行重定向时直接发起新的请求，并在条件允许情况下复用当前连接

5 BridgeInterceptor

```
public final class BridgeInterceptor implements Interceptor {
  private final CookieJar cookieJar;

  public BridgeInterceptor(CookieJar cookieJar) {
    this.cookieJar = cookieJar;
  }

  @Override public Response intercept(Chain chain) throws IOException {
    Log.e("haha", "BridgeInterceptor.intercept");
    Request userRequest = chain.request();
    Request.Builder requestBuilder = userRequest.newBuilder();

    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 decompr
    // the transfer stream.
    boolean transparentGzip = false;
    if (userRequest.header("Accept-Encoding") == null && userRequest.header("Range") == nul
      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();
  }
}
```

BridgeInterceptor主要负责以下几部分内容：

- 设置内容长度，内容编码
- 设置gzip压缩，并在接收到内容后进行解压。省去了应用层处理数据解压的麻烦
- 添加cookie
- 设置其他报头，如 `User-Agent` , `Host` , `Keep-alive` 等。其中 `Keep-Alive` 是实现多路复用的必要步骤

6. CacheInterceptor

```
[CacheInterceptor.intercept]

  @Override public Response intercept(Chain chain) throws IOException {
    Log.e("haha", "CacheInterceptor.intercept");
    Response cacheCandidate = cache != null
        ? cache.get(chain.request())
        : null;

    long now = System.currentTimeMillis();

    CacheStrategy strategy = new CacheStrategy.Factory(now, chain.request(), cacheCandidate
    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
    }

    // 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 {
      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, networkReque
        // 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;
  }
```

CacheInterceptor的职责很明确，就是负责Cache的管理

- 当网络请求有符合要求的Cache时直接返回Cache
- 当服务器返回内容有改变时更新当前cache
- 如果当前cache失效，删除

7 ConnectInterceptor

```
[ConnectInterceptor.java]
public final class ConnectInterceptor implements Interceptor {
  public final OkHttpClient client;

  public ConnectInterceptor(OkHttpClient client) {
    this.client = client;
  }

  @Override public Response intercept(Chain chain) throws IOException {
    Log.e("haha", "ConnectInterceptor.intercept");
    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 G
    boolean doExtensiveHealthChecks = !request.method().equals("GET");
    HttpCodec httpCodec = streamAllocation.newStream(client, doExtensiveHealthChecks);
    RealConnection connection = streamAllocation.connection();

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

ConnectInterceptor的intercept方法只有一行关键代码:

```
RealConnection connection = streamAllocation.connection();
```

即为当前请求找到合适的连接，可能复用已有连接也可能是重新创建的连接，返回的连接由连接池负责决定。

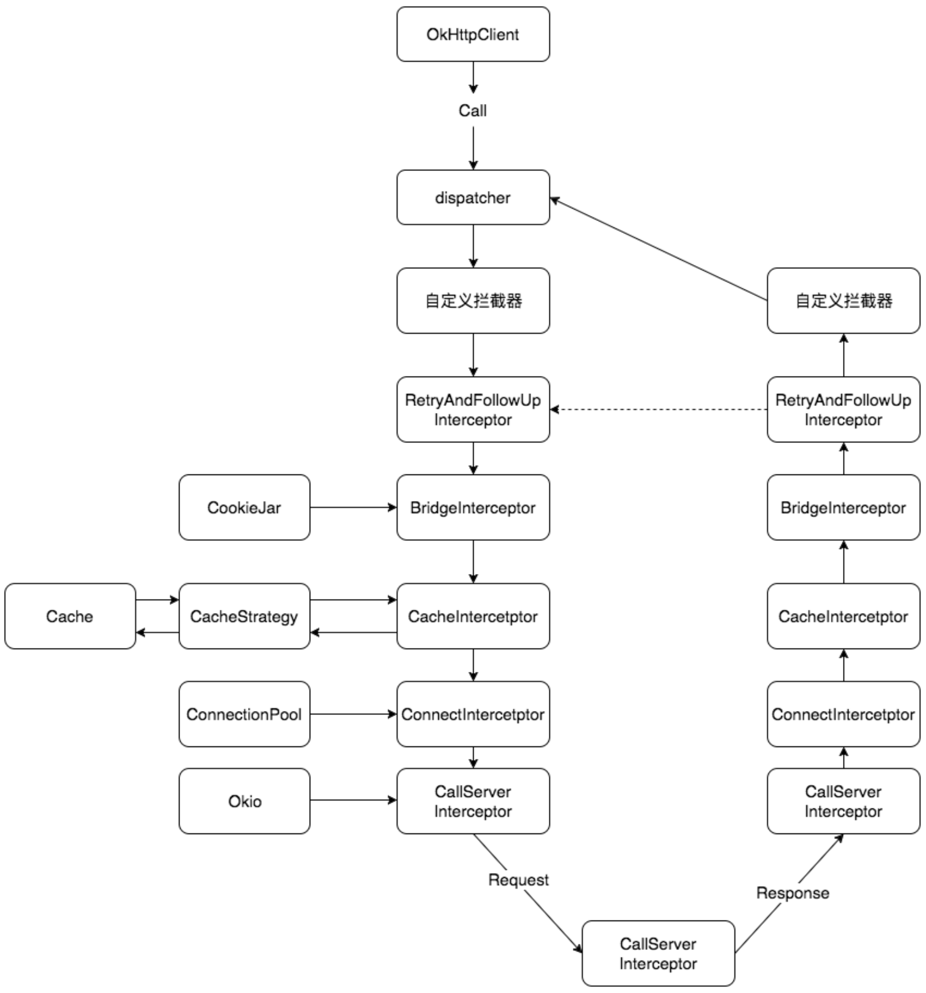
8. CallServerInterceptor

```
[CallServerInterceptor.java]
@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;
      ......

    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 respons
      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;
  }
```

CallServerInterceptor负责向服务器发起真正的访问请求，并在接收到服务器返回后读取响应返回。

8.整体流程

以上就是整个网络访问的核心步骤，总结起来如下图所示：


OkHttpClient
Call
dispatcher
自定义拦截器
RetryAndFollowUp Interceptor
CookieJar
BridgeInterceptor
Cache
CacheStrategy
CacheIntercetptor
ConnectionPool
ConnectIntercetptor
Okio
CallServer Interceptor
Request
CallServer Interceptor
Response
CallServer Interceptor
ConnectIntercetptor
CacheIntercetptor
BridgeInterceptor
RetryAndFollowUp Interceptor
自定义拦截器