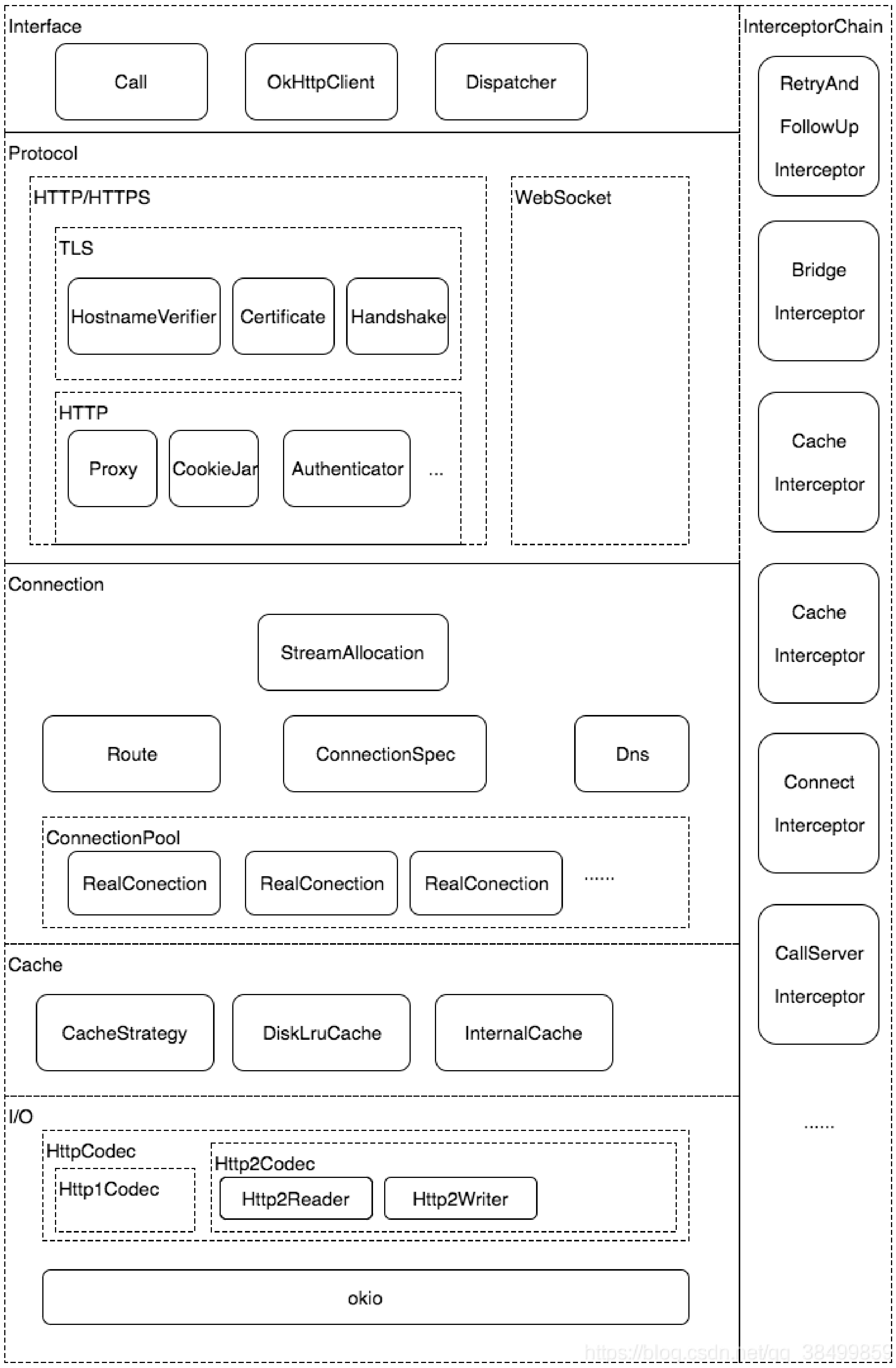
OkHttp

1.OkHttp总体架构

大致可以分为以下几层：

Interface——接口层：接受网络访问请求
Protocol——协议层：处理协议逻辑
Connection——连接层：管理网络连接，发送新的请求，接收服务器访问
Cache——缓存层：管理本地缓存
I/O——I/O层：实际数据读写实现
Inteceptor——拦截器层：拦截网络访问，插入拦截逻辑



1.1每层的含义

1.1.1Interface——接口层:

接口层接收用户的网络访问请求(同步/异步)，发起实际的网络访问。OKHttpClient是OkHttp框架的客户端，更确切的说是一个用户面板。用户使用OkHttp进行各种设置，发起各种网络请求都是通过OkHttpClient完成的。每个OkHttpClient内部都维护了属于自己的任务队列，连接池，Cache，拦截器等，所以在使用OkHttp作为网络框架时应该全局共享一个OkHttpClient实例。

Call描述了一个实际的访问请求，用户的每一个网络请求都是一个Call实例，Call本身是一个接口，定义了Call的接口方法，在实际执行过程中，OkHttp会为每一个请求创建一个RealCall，即Call的实现类。

Dispatcher是OkHttp的任务队列，其内部维护了一个线程池，当有接收到一个Call时，Dispatcher负责在线程池中找到空闲的线程并执行其execute方法。

1.1.2Protocol——协议层:处理协议逻辑

Protocol层负责处理协议逻辑，OkHttp支持Http1/Http2/WebSocket协议，并在3.7版本中放弃了对Spdy协议，鼓励开发者使用Http/2。

1.1.3Connection——连接层：管理网络连接，发送新的请求，接收服务器访问

连接层顾名思义就是负责网络连接，在连接层中有一个连接池，统一管理所有的Scoke连接，当用户发起一个新的网络请求是，OKHttp会在连接池找是否有符合要求的连接，如果有则直接通过该连接发送网络请求；否则新创建一个网络连接。

RealConnection描述一个物理Socket连接，连接池中维护多个RealConnection实例，由于Http/2支持多路复用，一个RealConnection，所以OKHttp又引入了StreamAllocation来描述一个实际的网络请求开销（从逻辑上一个Stream对应一个Call，但在实际网络请求过程中一个Call常常涉及到多次请求。如重定向，Authenticate等场景。所以准确地说，一个Stream对应一次请求，而一个Call对应一组有逻辑关联的Stream），一个RealConnection对应一个或多个StreamAllocation，所以StreamAllocation，是以StreamAllocation可以看做是RealConenction的计数器，当RealConnection的引用计数变为0，且长时间没有被其他请求重新占用就将被释放。

1.1.4Cache——缓存层：管理本地缓存

Cache层负责维护请求缓存，当用户的网络请求在本地已有符合要求的缓存时，OKHttp会直接从缓存中返回结果，从而节省 网络开销。

1.1.5Inteceptor——拦截器层：拦截网络访问，插入拦截逻辑

拦截器层提供了一个类AOP接口，方便用户可以切入到各个层面对网络访问进行拦截并执行相关逻辑。

2.OkHttp发送请求

一个简单的同步请求的OkHttp的示例。

```
new Thread(new Runnable() {
            @Override
            public void run() {
                try {

                    OkHttpClient client = new OkHttpClient();
                    Request request = new
Request.Builder().url("http://www.baidu.com")
                            .build();

                    try {
                        Response response = client.newCall(request).execute();
                        if (response.isSuccessful()) {
                            System.out.println("成功");
                        }
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();
```

2.1OkHttpClient()类

```
public OkHttpClient() {
    this(new Builder());
  }

  OkHttpClient(Builder builder) {
    this.dispatcher = builder.dispatcher;//调度器
    this.proxy = builder.proxy;//代理
    this.protocols = builder.protocols;//默认支持的Http协议版本
    this.connectionSpecs = builder.connectionSpecs;//OKHttp连接（Connection）配置
    this.interceptors = Util.immutableList(builder.interceptors);
    this.networkInterceptors = Util.immutableList(builder.networkInterceptors);
    this.eventListenerFactory = builder.eventListenerFactory;//一个Call的状态监听器
    this.proxySelector = builder.proxySelector;//使用默认的代理选择器
    this.cookieJar = builder.cookieJar;//默认是没有Cookie的；
    this.cache = builder.cache;//缓存
    this.internalCache = builder.internalCache;
    this.socketFactory = builder.socketFactory;//使用默认的Socket工厂产生Socket；

    boolean isTLS = false;
    for (ConnectionSpec spec : connectionSpecs) {
      isTLS = isTLS || spec.isTls();
    }

    if (builder.sslSocketFactory != null || !isTLS) {
      this.sslSocketFactory = builder.sslSocketFactory;
      this.certificateChainCleaner = builder.certificateChainCleaner;
    } else {
      X509TrustManager trustManager = systemDefaultTrustManager();
      this.sslSocketFactory = systemDefaultSslSocketFactory(trustManager);
      this.certificateChainCleaner = CertificateChainCleaner.get(trustManager);
    }

    this.hostnameVerifier = builder.hostnameVerifier;//安全相关的设置
    this.certificatePinner =
builder.certificatePinner.withCertificateChainCleaner(
        certificateChainCleaner);
    this.proxyAuthenticator = builder.proxyAuthenticator;
    this.authenticator = builder.authenticator;
    this.connectionPool = builder.connectionPool;//连接池
    this.dns = builder.dns;//域名解析系统 domain name -> ip address;
    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;// 这个和WebSocket有关。为了保持长连接，
我们必须间隔一段时间发送一个ping指令进行保活;

    if (interceptors.contains(null)) {
      throw new IllegalStateException("Null interceptor: " + interceptors);
    }
    if (networkInterceptors.contains(null)) {
      throw new IllegalStateException("Null network interceptor: " +
networkInterceptors);
    }
  }
```

在我们定义了请求对象后，我们需要生成一个Call对象，该对象代表一个准备被执行的请求，Call是可以被取消的，Call对象代表了一个request/response 对（Stream）.还有就是一个Call只能被执行一次. 从newCall进入源码

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

继续进入newReakCall中

```
final class RealCall implements Call {
  final OkHttpClient client;
  final RetryAndFollowUpInterceptor retryAndFollowUpInterceptor;

  /**
   * There is a cycle between the {@link Call} and {@link EventListener} that
makes this awkward.
   * This will be set after we create the call instance then create the event
listener instance.
   */
  private EventListener eventListener;

  /** The application's original request unadulterated by redirects or auth
headers. */
  final Request originalRequest;
  final boolean forWebSocket;

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

  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;
  }
    .....
}
```

可以看出在OkHttp中实际生产的是一个Call的实现类RealCall。

2.2Dispatcher类

Dispatcher类负责异步任务的请求策略。

```
public final class Dispatcher {
  private int maxRequests = 64;
    //每个主机的最大请求数,如果超过这个数，那么新的请求就会被放入到readyAsyncCalls队列中
  private int maxRequestsPerHost = 5;
     //是Dispatcher中请求数量为0时的回调，这儿的请求包含同步请求和异步请求，该参数默认为
null。
  private @Nullable Runnable idleCallback;

  /** Executes calls. Created lazily. */
  private @Nullable ExecutorService executorService;
    //任务队列线程池

  /** Ready async calls in the order they'll be run. */
  private final Deque<AsyncCall> readyAsyncCalls = new ArrayDeque<>();
    //待执行异步任务队列

  /** Running asynchronous calls. Includes canceled calls that haven't finished
yet. */
  private final Deque<AsyncCall> runningAsyncCalls = new ArrayDeque<>();
    //运行中的异步任务队列
  /** Running synchronous calls. Includes canceled calls that haven't finished
yet. */
  private final Deque<RealCall> runningSyncCalls = new ArrayDeque<>();
      //运行中同步任务队列
  public Dispatcher(ExecutorService executorService) {
    this.executorService = executorService;
  }

  public Dispatcher() {
  }
  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;
  }
```

查看ThreadPoolExecutor类

```
    public ThreadPoolExecutor(int corePoolSize,
                              int maximumPoolSize,
                              long keepAliveTime,
                              TimeUnit unit,
                              BlockingQueue<Runnable> workQueue,
                              ThreadFactory threadFactory) {
        this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
             threadFactory, defaultHandler);
    }
```

corePoolSize :核心线程数，默认情况下核心线程会一直存活
maximumPoolSize: 线程池所能容纳的最大线程数。超过这个数的线程将被阻塞。
keepAliveTime: 非核心线程的闲置超时时间，超过这个时间就会被回收。
unit: keepAliveTime的单位。
workQueue: 线程池中的任务队列。
threadFactory: 线程工厂，提供创建新线程的功能

corePoolSize设置为0表示一旦有闲置的线程就可以回收。容纳最大线程数设置的非常大，但是由于受到maxRequests的影响，并不会创建特别多的线程。60秒的闲置时间。

2.3同步请求的执行流程

```
 new Thread(new Runnable() {
            @Override
            public void run() {
                try {

                    OkHttpClient client = new OkHttpClient();
                    Request request = new
Request.Builder().url("http://www.baidu.com")
                            .build();

                    try {
                        Response response = client.newCall(request).execute();
                        if (response.isSuccessful()) {
                            System.out.println("成功");
                        }
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();
```

在同步请求里，调用了 client.newCall(request).execute()的方法，在上文说过newCall返回的是一个RealCall对象，所以execute的实现在RealCall中

```
//RealCall类中
@Override public Response execute() throws IOException {
    //设置execute标志为true，即同一个Call只允许执行一次，执行多次就会抛出异常
  synchronized (this) {
    if (executed) throw new IllegalStateException("Already Executed");
    executed = true;
  }
   //重定向拦截器相关
  captureCallStackTrace();
  eventListener.callStart(this);
  try {
      //调用dispatcher()获取Dispatcher对象，调用executed方法
    client.dispatcher().executed(this);
       //getResponseWithInterceptorChain拦截器链
    Response result = getResponseWithInterceptorChain();
    if (result == null) throw new IOException("Canceled");
    return result;
  } catch (IOException e) {
    eventListener.callFailed(this, e);
    throw e;
  } finally {
      //调用Dispatcher的finished方法
    client.dispatcher().finished(this);
  }
}
```

进入 captureCallStackTrace();

```
 private void captureCallStackTrace() {
    Object callStackTrace =
Platform.get().getStackTraceForCloseable("response.body().close()");
    retryAndFollowUpInterceptor.setCallStackTrace(callStackTrace);
  }
```

查看dispatcher的finished方法

```
//Dispatcher类中
  void finished(RealCall call) {
    finished(runningSyncCalls, call, false);
  }

private <T> void finished(Deque<T> calls, T call, boolean promoteCalls) {
    int runningCallsCount;
    Runnable idleCallback;
    synchronized (this) {
       //移出请求，如果不能移除，则抛出异常
      if (!calls.remove(call)) throw new AssertionError("Call wasn't in-
flight!");

      //传入参数为flase，不执行这个语句
      if (promoteCalls) promoteCalls();

      //unningCallsCount统计目前还在运行的请求
```

```
        runningCallsCount = runningCallsCount();

        //请求数为0时的回调
        idleCallback = this.idleCallback;
      }

      //如果请求数为0，且idleCallback不为NULL，回调idleCallback的run方法。
      if (runningCallsCount == 0 && idleCallback != null) {
        idleCallback.run();
      }
    }
```

查看runningCallsCount()方法

```
    public synchronized int runningCallsCount() {
      return runningAsyncCalls.size() + runningSyncCalls.size();
    }
```

2.4异步请求的执行流程

```
   private void getDataAsync() {
          OkHttpClient client = new OkHttpClient();
          Request request = new Request.Builder()
                  .url("http://www.baidu.com")
                  .build();
          client.newCall(request).enqueue(new Callback() {
              @Override
              public void onFailure(Call call, IOException e) {
              }
              @Override
              public void onResponse(Call call, Response response) throws
  IOException {
                  if(response.isSuccessful()){//回调的方法执行在子线程。
                      Log.d("OkHttp","获取数据成功了");
                      Log.d("OkHttp","response.code()=="+response.code());

  Log.d("OkHttp","response.body().string()=="+response.body().string());
                  }
              }
          });
      }
```

和同步请求类似，client.newCall(request).enqueue的方法，所以enqueue的实现在RealCall中

```
    @Override public void enqueue(Callback responseCallback) {
         //设置exexuted参数为true，表示不可以执行两次。
      synchronized (this) {
        if (executed) throw new IllegalStateException("Already Executed");
        executed = true;
      }
      captureCallStackTrace();
      eventListener.callStart(this);
        //调用dispatcher()的enqueuef方法，不过在里面传入一次新的参数，AsyncCall类
      client.dispatcher().enqueue(new AsyncCall(responseCallback));
    }
```

进入 AsyncCall类

```
final class AsyncCall extends NamedRunnable {
    private final Callback responseCallback;

    AsyncCall(Callback responseCallback) {
      super("OkHttp %s", redactedUrl());
      this.responseCallback = responseCallback;
    }

    String host() {
      return originalRequest.url().host();
    }

    Request request() {
      return originalRequest;
    }

    RealCall get() {
      return RealCall.this;
    }

    @Override protected void execute() {
      boolean signalledCallback = false;
      try {
         //执行耗时的IO操作
        //获取拦截器链
        Response response = getResponseWithInterceptorChain();
        if (retryAndFollowUpInterceptor.isCanceled()) {
          signalledCallback = true;
          //回调，注意这里回调是在线程池中，而不是向当前的主线程回调
          responseCallback.onFailure(RealCall.this, new
IOException("Canceled"));
        } else {
          signalledCallback = true;
           //回调，同上
          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);
          //回调
          responseCallback.onFailure(RealCall.this, e);
        }
      } finally {
        client.dispatcher().finished(this);
      }
    }
  }
```

查看AsyncCall的父类NamedRunnable

```
/**
```

```
 * Runnable implementation which always sets its thread name.
 */
//实现了Runnable接口
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 {
      //执行抽象方法，也就是 AsyncCall中的execute
      execute();
    } finally {
      Thread.currentThread().setName(oldName);
    }
  }

  protected abstract void execute();
}
```

回到RealCall的enqueue，进入到Dispatcher().enqueue中

```
//Dispatcher()类
synchronized void enqueue(AsyncCall call) {
     //如果正在运行的异步请求的数量小于maxRequests并且与该请求相同的主机数量小于
maxRequestsPerHost
  if (runningAsyncCalls.size() < maxRequests && runningCallsForHost(call) <
maxRequestsPerHost) {
      //放入runningAsyncCalls队列中
    runningAsyncCalls.add(call);
      //这里调用了executorService()
    executorService().execute(call);
  } else {
      //否则，放入readyAsyncCalls队列
    readyAsyncCalls.add(call);
  }
}
```

当线程池执行AsyncCall任务时，它的execute方法会被调用

查看Dispatcher的finished方法

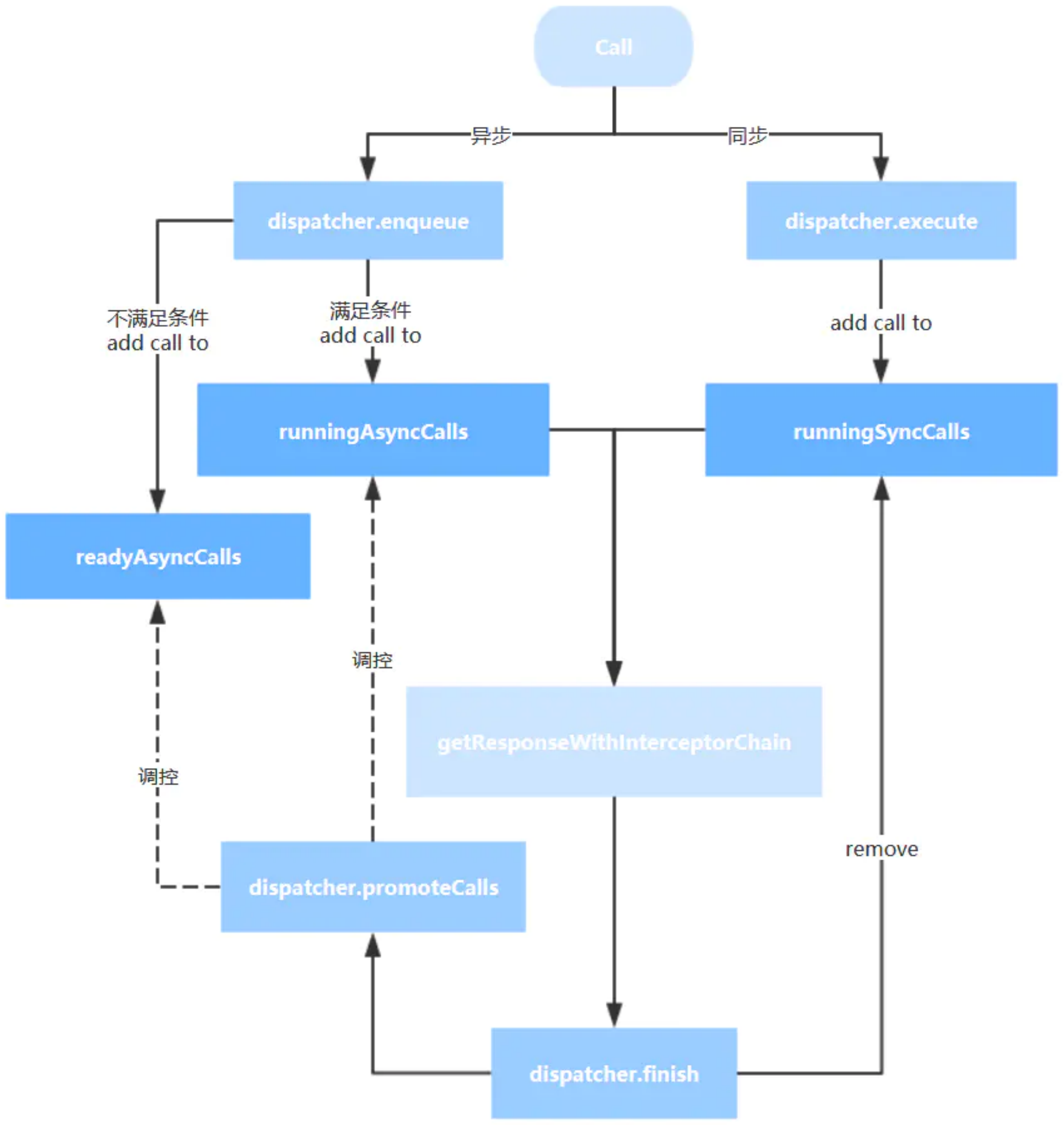
```
//Dispatcher
  /** Used by {@code AsyncCall#run} to signal completion. */
  void finished(AsyncCall call) {
    finished(runningAsyncCalls, call, true);
  }

private <T> void finished(Deque<T> calls, T call, boolean promoteCalls) {
    int runningCallsCount;
    Runnable idleCallback;
    synchronized (this) {
        //从队列中删除
```

```
      if (!calls.remove(call)) throw new AssertionError("Call wasn't in-
flight!");
        //异步请求调用 promoteCalls()方法
      if (promoteCalls) promoteCalls();
      runningCallsCount = runningCallsCount();
      idleCallback = this.idleCallback;
    }

    if (runningCallsCount == 0 && idleCallback != null) {
      idleCallback.run();
    }
  }
```

查看promoteCalls()

```
  private void promoteCalls() {

    //运行中的异步任务队列大于等于最大的请求数
    if (runningAsyncCalls.size() >= maxRequests) return; // Already running max
capacity.
    //待执行异步任务队列为空
    if (readyAsyncCalls.isEmpty()) return; // No ready calls to promote.

    //遍历等待队列
    for (Iterator<AsyncCall> i = readyAsyncCalls.iterator(); i.hasNext(); ) {
      AsyncCall call = i.next();

      //将符合条件的事件，从等待队列中移出，放入运行队列中。
      if (runningCallsForHost(call) < maxRequestsPerHost) {
        i.remove();
        runningAsyncCalls.add(call);
        executorService().execute(call);
      }

      if (runningAsyncCalls.size() >= maxRequests) return; // Reached max
capacity.
    }
  }
```

总体流程图



3.OkHttp的拦截器和封装

在OKHttp中，中Interceptors拦截器是一种强大的机制，可以监视，重写和重试Call请求。

3.1OkHttp的拦截器的作用：

*拦截器可以一次性对所有请求的返回值进行修改
*拦截器可以一次性对请求的参数和返回的结果进行编码，比如统一设置为UTF-8.
*拦截器可以对所有的请求做统一的日志记录，不需要在每个请求开始或者结束的位置都添加一个日志操作。
*其他需要对请求和返回进行统一处理的需求...

3.2OkHttp拦截器的分类

OkHttp中的拦截器分2个：APP层面的拦截器（Application Interception）网络请求层面的拦截器(Network Interception)。

3.3两种的区别

Application：

*不需要担心是否影响OKHttp的请求策略和请求速度
*即使从缓存中取数据，也会执行Application拦截器
*允许重试，即Chain.proceed()可以执行多次。
*可以监听观察这个请求的最原始的未改变的意图(请求头，请求体等)，无法操作OKHttp为我们自动添加额外的请求头
*无法操作中间的响应结果，比如当URL重定向发生以及请求重试，只能操作客户端主动第一次请求以及最终的响应结果

Network Interceptors

*可以修改OkHttp框架自动添加的一些属性，即允许操作中间响应，比如当请求操作发生重定向或者重试等。
*可以观察最终完整的请求参数（也就是最终服务器接收到的请求数据和熟悉）

3.4实例化appInterceptor拦截器

```
/**
 * 应用拦截器
 */
Interceptor appInterceptor = new Interceptor() {
        @Override
        public Response intercept(Chain chain) throws IOException {
            Request request = chain.request();
            HttpUrl url = request.url();
            String s = url.url().toString();


            Log.d(TAG, "app intercept:begin ");
            Response response = chain.proceed(request);//请求
            Log.d(TAG, "app intercept:end");
            return response;
        }
    };
```

3.5实例化networkInterceptor拦截器

```
    /**
     * 网络拦截器
     */
    Interceptor networkInterceptor = new Interceptor() {
            @Override
            public Response intercept(Chain chain) throws IOException {
                Request request = chain.request();
                Log.d(TAG,"network interceptor:begin");
                Response  response = chain.proceed(request);//请求
                Log.d(TAG,"network interceptor:end");
                return response;
            }
        };
```

3.6拦截器的实际应用

3.6.1统一添加Header

应用场景:后台要求在请求API时，在每一个接口的请求头添加上对于的Token。这时候就可以使用拦截器对他们进行统一配置。

实例化拦截器

```
  Interceptor  TokenHeaderInterceptor = new Interceptor() {
        @Override
        public Response intercept(Chain chain) throws IOException {
            // get token
            String token = AppService.getToken();
            Request originalRequest = chain.request();
            // get new request, add request header
            Request updateRequest = originalRequest.newBuilder()
                    .header("token", token)
                    .build();
            return chain.proceed(updateRequest);
        }
    };
```

3.6.2改变请求体

应用场景:在上面的 login 接口基础上，后台要求我们传过去的请求参数是要按照一定规则经过加密的。

规则:

*请求参数名统一为content;

*content值：JSON 格式的字符串经过 AES 加密后的内容

实例化拦截器

```
    Interceptor  RequestEncryptInterceptor = new Interceptor() {

        private static final String FORM_NAME = "content";
        private static final String CHARSET = "UTF-8";
        @Override
        public Response intercept(Chain chain) throws IOException {
            // get token
            Request request = chain.request();

            RequestBody body = request.body();

            if (body instanceof FormBody){
                FormBody formBody = (FormBody) body;
                Map<String, String> formMap = new HashMap<>();

                // 从 formBody 中拿到请求参数，放入 formMap 中
                for (int i = 0; i < formBody.size(); i++) {
                    formMap.put(formBody.name(i), formBody.value(i));
                }

                // 将 formMap 转化为 json 然后 AES 加密
                Gson gson = new Gson();
                String jsonParams = gson.toJson(formMap);
                String encryptParams =
AESCryptUtils.encrypt(jsonParams.getBytes(CHARSET), AppConstant.getAESKey());

                // 重新修改 body 的内容
                body = new FormBody.Builder().add(FORM_NAME,
encryptParams).build();
            }

            if (body != null) {
                request = request.newBuilder()
                        .post(body)
                        .build();
            }
            return chain.proceed(request);
        }
    };
```

4.拦截器链

4.1getResponseWithInterceptorChain方法

同步和异步响应中都出现了getResponseWithInterceptorChain方法

```
//RealCall
Response getResponseWithInterceptorChain() throws IOException {
    // Build a full stack of interceptors.
    List<Interceptor> interceptors = new ArrayList<>();
    //添加应用拦截器
    interceptors.addAll(client.interceptors());
    //添加重试和重定向拦截器
    interceptors.add(retryAndFollowUpInterceptor);
    //添加转换拦截器
    interceptors.add(new BridgeInterceptor(client.cookieJar()));
    //添加缓存拦截器
    interceptors.add(new CacheInterceptor(client.internalCache()));
    //添加连接拦截器
    interceptors.add(new ConnectInterceptor(client));
     //添加网络拦截器
    if (!forWebSocket) {
      interceptors.addAll(client.networkInterceptors());
    }
    interceptors.add(new CallServerInterceptor(forWebSocket));
    //生成拦截器链
    Interceptor.Chain chain = new RealInterceptorChain(interceptors, null, null,
null, 0,
        originalRequest, this, eventListener, client.connectTimeoutMillis(),
        client.readTimeoutMillis(), client.writeTimeoutMillis());

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

从上面的代码可以看出，向interceptors添加了一系列的拦截器。最后构造了一个RealInterceptorChain对象，该类是拦截器链的具体体现，携带了整个拦截器链，包含了所有的应用拦截器，OKHttp的核心。

OKHttp这种拦截器链采用的是责任链模式，这样的好处就是讲请求的发送和处理分开处理，并且可以动态添加中间处理实现对请求的处理和短路操作。

4.2RealInterceptorChain类

```java
public final class RealInterceptorChain implements Interceptor.Chain {
  private final List<Interceptor> interceptors;//传递的拦截器集合
  private final StreamAllocation streamAllocation;
  private final HttpCodec httpCodec;
  private final RealConnection connection;
  private final int index; //当前拦截器的索引
  private final Request request;//当前的realReques
  private final Call call;
  private final EventListener eventListener;
  private final int connectTimeout;
  private final int readTimeout;
  private final int writeTimeout;
  private int calls;

  public RealInterceptorChain(List<Interceptor> interceptors, StreamAllocation
streamAllocation,
      HttpCodec httpCodec, RealConnection connection, int index, Request
request, Call call,
      EventListener eventListener, int connectTimeout, int readTimeout, int
writeTimeout) {
    this.interceptors = interceptors;
    this.connection = connection;
    this.streamAllocation = streamAllocation;
    this.httpCodec = httpCodec;
    this.index = index;
    this.request = request;
    this.call = call;
    this.eventListener = eventListener;
    this.connectTimeout = connectTimeout;
    this.readTimeout = readTimeout;
    this.writeTimeout = writeTimeout;
  }
   .....
}
```

在getResponseWithInterceptorChain()最后返回代码时调用了拦截器链的prooceed方法

```java
//RealInterceptorChain
 public Response proceed(Request request, StreamAllocation streamAllocation,
HttpCodec httpCodec,
      RealConnection connection) throws IOException {
    if (index >= interceptors.size()) throw new AssertionError();

    calls++;

    //错误处理相关
    // If we already have a stream, confirm that the incoming request will use
it.
    if (this.httpCodec != null && !this.connection.supportsUrl(request.url())) {
      throw new IllegalStateException("network interceptor " +
interceptors.get(index - 1)
          + " must retain the same host and port");
    }

    // If we already have a stream, confirm that this is the only call to
chain.proceed().
    if (this.httpCodec != null && calls > 1) {
      throw new IllegalStateException("network interceptor " +
interceptors.get(index - 1)
          + " must call proceed() exactly once");
    }

    // Call the next interceptor in the chain.
    //核心代码
    RealInterceptorChain next = new RealInterceptorChain(interceptors,
streamAllocation, httpCodec,
        connection, index + 1, request, call, eventListener, connectTimeout,
readTimeout,
        writeTimeout);
    //获取下一个拦截器
    Interceptor interceptor = interceptors.get(index);
    //调用当前拦截器的intercept方法，并将下一个拦截器传入其中。
    Response response = interceptor.intercept(next);

    // Confirm that the next interceptor made its required call to
chain.proceed().
    if (httpCodec != null && index + 1 < interceptors.size() && next.calls != 1)
{
      throw new IllegalStateException("network interceptor " + interceptor
          + " must call proceed() exactly once");
    }

    // Confirm that the intercepted response isn't null.
    if (response == null) {
      throw new NullPointerException("interceptor " + interceptor + " returned
null");
    }

    if (response.body() == null) {
      throw new IllegalStateException(
          "interceptor " + interceptor + " returned a response with no body");
    }

    return response;
  }
```

4.2.1RetryAndFollowUpInterceptor

按照添加的顺序逐个分析各个拦截器

RetryAndFollowUpInterceptor拦截器可以从错误中恢复和重定向，如果Call被取消了，那么将会抛出IoException。
查看其intercept方法

```
 @Override public Response intercept(Chain chain) throws IOException {
    Request request = chain.request();
    RealInterceptorChain realChain = (RealInterceptorChain) chain;
    Call call = realChain.call();
    EventListener eventListener = realChain.eventListener();

    //①
    StreamAllocation streamAllocation = new
StreamAllocation(client.connectionPool(),
        createAddress(request.url()), call, eventListener, callStackTrace);
    this.streamAllocation = streamAllocation;

    int followUpCount = 0;
    Response priorResponse = null;
    //②
    while (true) {
      if (canceled) {
        streamAllocation.release();
        throw new IOException("Canceled");
      }

      Response response;
      boolean releaseConnection = true;
      try {
        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();
      }

      Request followUp = followUpRequest(response, streamAllocation.route());

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

4.2.1.1StreamAllocation

源码①.创建了一个StreamAllocation，这个是用来做连接分配的，传递的参数有五个，第一个是前面创建的连接池，第二个是调用createAddress创建的Address，第三个是Call。

```
//①
    StreamAllocation streamAllocation = new
StreamAllocation(client.connectionPool(),
         createAddress(request.url()), call, eventListener, callStackTrace);
    this.streamAllocation = streamAllocation;
```

createAddress方法

```
private Address createAddress(HttpUrl url) {
    SSLSocketFactory sslSocketFactory = null;
    HostnameVerifier hostnameVerifier = null;
    CertificatePinner certificatePinner = null;
    //如果是https
    if (url.isHttps()) {
      sslSocketFactory = client.sslSocketFactory();
      hostnameVerifier = client.hostnameVerifier();
      certificatePinner = client.certificatePinner();
    }

    return new Address(url.host(), url.port(), client.dns(),
client.socketFactory(),
        sslSocketFactory, hostnameVerifier, certificatePinner,
client.proxyAuthenticator(),
        client.proxy(), client.protocols(), client.connectionSpecs(),
client.proxySelector());
  }
```

Address类的构造方法

```
 public Address(String uriHost, int uriPort, Dns dns, SocketFactory
socketFactory,
      @Nullable SSLSocketFactory sslSocketFactory, @Nullable HostnameVerifier
hostnameVerifier,
      @Nullable CertificatePinner certificatePinner, Authenticator
proxyAuthenticator,
      @Nullable Proxy proxy, List<Protocol> protocols, List<ConnectionSpec>
connectionSpecs,
      ProxySelector proxySelector) {
    this.url = new HttpUrl.Builder()
        .scheme(sslSocketFactory != null ? "https" : "http")
        .host(uriHost)
        .port(uriPort)
        .build();

    if (dns == null) throw new NullPointerException("dns == null");
    this.dns = dns;

    if (socketFactory == null) throw new NullPointerException("socketFactory ==
null");
    this.socketFactory = socketFactory;

    if (proxyAuthenticator == null) {
      throw new NullPointerException("proxyAuthenticator == null");
    }
    this.proxyAuthenticator = proxyAuthenticator;

    if (protocols == null) throw new NullPointerException("protocols == null");
    this.protocols = Util.immutableList(protocols);

    if (connectionSpecs == null) throw new NullPointerException("connectionSpecs
== null");
    this.connectionSpecs = Util.immutableList(connectionSpecs);

    if (proxySelector == null) throw new NullPointerException("proxySelector ==
null");
    this.proxySelector = proxySelector;
    this.proxy = proxy;
    this.sslSocketFactory = sslSocketFactory;
    this.hostnameVerifier = hostnameVerifier;
    this.certificatePinner = certificatePinner;
  }
```

根据clent和请求的想换信息初始化了Address

查看StreamAllocation

```
  public StreamAllocation(ConnectionPool connectionPool, Address address, Call
call,
      EventListener eventListener, Object callStackTrace) {
    this.connectionPool = connectionPool;
    this.address = address;
    this.call = call;
    this.eventListener = eventListener;
    //路由选择器
    this.routeSelector = new RouteSelector(address, routeDatabase(), call,
eventListener);
    this.callStackTrace = callStackTrace;
  }
```

4.2.1.2发生请求&接收响应

回到intercept，看源码②while处代码,先看上半部分

```
while (true) {
      if (canceled) { //查看请求是否取消
        streamAllocation.release();
        throw new IOException("Canceled");
      }

      Response response;//响应
      boolean releaseConnection = true;//是否需要重连
      try {
        //调用拦截器链的proceed方法，在这个方法中，会调用下一个拦截器
        //这就是之前所说拦截器链的顺序调用
        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.
        //释放资源
          if (releaseConnection) {

            streamAllocation.streamFailed(null);
            streamAllocation.release();
          }
        }

       .......
      }
```

查看recover方法

```
 private boolean recover(IOException e, StreamAllocation streamAllocation,
      boolean requestSendStarted, Request userRequest) {
    streamAllocation.streamFailed(e);

    // The application layer has forbidden retries.
    //应用层禁止重试
    if (!client.retryOnConnectionFailure()) return false;

    // We can't send the request body again.
    //不能再发送请求体了
    if (requestSendStarted && userRequest.body() instanceof
UnrepeatableRequestBody) return false;

    // This exception is fatal.
    //这个异常无法重试
    if (!isRecoverable(e, requestSendStarted)) return false;

    // No more routes to attempt.
    //没有更多的attempt
    if (!streamAllocation.hasMoreRoutes()) return false;

    // For failure recovery, use the same route selector with a new connection.
    //上面的条件都不满足，此时就可以进行重试
    return true;
  }
```

4.2.1.3错误重试和重定向

来看while循环的下半部分

```
while(true){
......
  // Attach the prior response if it exists. Such responses never have a body.
  //priorResponse不为空，说明之前已经获得响应
      if (priorResponse != null) {
      //结合当前的response和之前的response获得新的response。
        response = response.newBuilder()
            .priorResponse(priorResponse.newBuilder()
                    .body(null)
                    .build())
            .build();
      }

     //调用followUpRequest查看响应是否需要重定向，如果不需要重定向则返回当前请求,如果需要返
回新的请求
     // followUpRequest源码见下
      Request followUp = followUpRequest(response, streamAllocation.route());

      //不需要重定向或者无法重定向
      if (followUp == null) {
        if (!forWebSocket) {
          streamAllocation.release();
        }
        return response;
      }

      closeQuietly(response.body());

     //重试次数+1
     //重试次数超过MAX_FOLLOW_UPS（默认20），抛出异常
      if (++followUpCount > MAX_FOLLOW_UPS) {
        streamAllocation.release();
        throw new ProtocolException("Too many follow-up requests: " +
followUpCount);
      }

      //followUp与当前的响应对比，是否为同一个连接
      if (followUp.body() instanceof UnrepeatableRequestBody) {
        streamAllocation.release();
        throw new HttpRetryException("Cannot retry streamed HTTP body",
response.code());
      }

     //followUp与当前请求的不是同一个连接时，则重写申请重新设置streamAllocation
      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?");
      }

      //重新设置reques，并把当前的Response保存到priorResponse，继续while循环
      request = followUp;
      priorResponse = response;
    }
```

followUpRequest的源码

```
  private Request followUpRequest(Response userResponse, Route route) throws
IOException {
    if (userResponse == null) throw new IllegalStateException();
    //返回的响应码
    int responseCode = userResponse.code();

    //请求方法
    final String method = userResponse.request().method();
    switch (responseCode) {
     //407请求要求代理的身份认证
      case HTTP_PROXY_AUTH:
        Proxy selectedProxy = route != null
            ? route.proxy()
            : client.proxy();
        if (selectedProxy.type() != Proxy.Type.HTTP) {
          throw new ProtocolException("Received HTTP_PROXY_AUTH (407) code while
not using proxy");
        }
        return client.proxyAuthenticator().authenticate(route, userResponse);

      //401请求要求用户的身份认证
      case HTTP_UNAUTHORIZED:
        return client.authenticator().authenticate(route, userResponse);

      //307&308 临时重定向。使用GET请求重定向
      case HTTP_PERM_REDIRECT:
      case HTTP_TEMP_REDIRECT:
        // "If the 307 or 308 status code is received in response to a request
other than GET
        // or HEAD, the user agent MUST NOT automatically redirect the request"
        if (!method.equals("GET") && !method.equals("HEAD")) {
          return null;
        }
        // fall-through

      case HTTP_MULT_CHOICE: //300多种选择。请求的资源可包括多个位置，相应可返回一个资源特
征与地址的列表用于用户终端（例如：浏览器）选择
      case HTTP_MOVED_PERM://301永久移动。请求的资源已被永久的移动到新URI，返回信息会包括
新的URI，浏览器会自动定向到新URI。
      case HTTP_MOVED_TEMP://302临时移动。与301类似。但资源只是临时被移动。
      case HTTP_SEE_OTHER://303查看其它地址。与301类似。使用GET和POST请求查看
        // Does the client allow redirects?
        if (!client.followRedirects()) return null;

        String location = userResponse.header("Location");
        if (location == null) return null;
        HttpUrl url = userResponse.request().url().resolve(location);

        // Don't follow redirects to unsupported protocols.
        if (url == null) return null;

        // If configured, don't follow redirects between SSL and non-SSL.
        boolean sameScheme =
url.scheme().equals(userResponse.request().url().scheme());
        if (!sameScheme && !client.followSslRedirects()) return null;

        // Most redirects don't include a request body.
        Request.Builder requestBuilder = userResponse.request().newBuilder();
        if (HttpMethod.permitsRequestBody(method)) {
          final boolean maintainBody = HttpMethod.redirectsWithBody(method);
          if (HttpMethod.redirectsToGet(method)) {
            requestBuilder.method("GET", null);
          } else {
            RequestBody requestBody = maintainBody ?
userResponse.request().body() : null;
```

```
            requestBuilder.method(method, requestBody);
          }
          if (!maintainBody) {

            requestBuilder.removeHeader("Transfer-Encoding");
            requestBuilder.removeHeader("Content-Length");
            requestBuilder.removeHeader("Content-Type");
          }
        }

        // When redirecting across hosts, drop all authentication headers. This
        // is potentially annoying to the application layer since they have no
        // way to retain them.
        if (!sameConnection(userResponse, url)) {
        //移出请求头
          requestBuilder.removeHeader("Authorization");
        }

        return requestBuilder.url(url).build();

      case HTTP_CLIENT_TIMEOUT: //408 服务器无法根据客户端请求的内容特性完成请求
        // 408's are rare in practice, but some servers like HAProxy use this
response code. The
        // spec says that we may repeat the request without modifications.
Modern browsers also
        // repeat the request (even non-idempotent ones.)
        if (!client.retryOnConnectionFailure()) {
          // The application layer has directed us not to retry the request.
          return null;
        }

        if (userResponse.request().body() instanceof UnrepeatableRequestBody) {
          return null;
        }

        if (userResponse.priorResponse() != null
            && userResponse.priorResponse().code() == HTTP_CLIENT_TIMEOUT) {
          // We attempted to retry and got another timeout. Give up.
          return null;
        }

        if (retryAfter(userResponse, 0) > 0) {
          return null;
        }

        return userResponse.request();

      case HTTP_UNAVAILABLE://503   由于超载或系统维护，服务器暂时的无法处理客户端的请求。
延时的长度可包含在服务器的Retry-After头信息中
        if (userResponse.priorResponse() != null
            && userResponse.priorResponse().code() == HTTP_UNAVAILABLE) {
          // We attempted to retry and got another timeout. Give up.
          return null;
        }

        if (retryAfter(userResponse, Integer.MAX_VALUE) == 0) {
          // specifically received an instruction to retry without delay
          return userResponse.request();
        }

        return null;

      default:
        return null;
    }
  }
```

4.3BridgeInterceptor类

在RetryAndFollowUpInterceptor 执行response = realChain.proceed(request, streamAllocation, null, null)代码时，此时会调用下一个拦截器，即BridgeInterceptor拦截器

BridgeInterceptor转换拦截器主要工作就是为请求添加请求头，为响应添加响应头

4.3.1intercept

BridgeInterceptor的intercept代码

下面代码主要为request添加Content-Type(文档类型)、Content-Length(内容长度)或Transfer-Encoding，从这里我们也可以发现其实这些头信息是不需要我们手动添加的.即使我们手动添加也会被覆盖掉。

```
    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");
      }
    }
```

下面的代码时为Host、Connection和User-Agent字段添加默认值，不过不同于上面的，这几个属性只有用户没有设置时，OkHttp会自动添加，如果你收到添加时，不会被覆盖掉。

```
  if (userRequest.header("Host") == null) {
      requestBuilder.header("Host", hostHeader(userRequest.url(), false));
    }

    if (userRequest.header("Connection") == null) {
      requestBuilder.header("Connection", "Keep-Alive");
    }

    if (userRequest.header("User-Agent") == null) {
      requestBuilder.header("User-Agent", Version.userAgent());
    }
```

默认支持gzip压缩

```
    // 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");
    }
```

cookie部分

```
   List<Cookie> cookies = cookieJar.loadForRequest(userRequest.url());
    if (!cookies.isEmpty()) {
      requestBuilder.header("Cookie", cookieHeader(cookies));
    }
```

进入cookHeader方法

```
/** 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();
  }
```

之后就是进入下一个拦截器中，并将最后的响应返回

```
  Response networkResponse = chain.proceed(requestBuilder.build());
```

在获得响应后，如果有cookie，则保存

```
HttpHeaders.receiveHeaders(cookieJar, userRequest.url(),
networkResponse.headers());
```

```
 public static void receiveHeaders(CookieJar cookieJar, HttpUrl url, Headers
headers) {
    if (cookieJar == CookieJar.NO_COOKIES) return;

    List<Cookie> cookies = Cookie.parseAll(url, headers);
    if (cookies.isEmpty()) return;

    cookieJar.saveFromResponse(url, cookies);
  }
```

下面就是对response的解压工作，将流转换为直接能使用的response，然后对header进行了一些处理构建了一个response返回给上一个拦截器。

```
 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();
```

4.3.2总结

从上面的代码可以看出了，先获取原请求头，然后在请求中添加请求头，然后在根据需求，决定是否要填充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

4.4CacheInterceptor类

4.4.1传入参数

CacheInterceptor创建时传入的参数

```
interceptors.add(new CacheInterceptor(client.internalCache()));
```

查看client的internalCache方法，可以看出。CacheInterceptor使用OkHttpClient的internalCache方法的返回值作为参数

```
  InternalCache internalCache() {
    return cache != null ? cache.internalCache : internalCache;
  }
```

而Cache和InternalCache都是OkHttpClient.Builder中可以设置的，而其设置会互相抵消，代码如下：

```
    /** Sets the response cache to be used to read and write cached responses. */
    void setInternalCache(@Nullable InternalCache internalCache) {
      this.internalCache = internalCache;
      this.cache = null;
    }

    /** Sets the response cache to be used to read and write cached responses.
*/
    public Builder cache(@Nullable Cache cache) {
      this.cache = cache;
      this.internalCache = null;
      return this;
    }
```

默认的，如果没有对Builder进行缓存设置，那么cache和internalCache都为null，那么传入到CacheInterceptor中的也是null

4.4.2缓存策略

接下来进入CacheInterceptor的intercept方法中
下面这段代码是获得缓存响应 和获得响应策略

```
//CacheInterceptor.intercept（）中
//得到候选响应
Response cacheCandidate = cache != null
        ? cache.get(chain.request())
        : null;

    long now = System.currentTimeMillis();

     //根据请求以及缓存响应得出缓存策略
    CacheStrategy strategy = new CacheStrategy.Factory(now, chain.request(),
cacheCandidate).get();
    Request networkRequest = strategy.networkRequest; //网络请求，如果为null就代表不用进行网络请求
    Response cacheResponse = strategy.cacheResponse;//缓存响应，如果为null，则代表不使用缓存
```

进入查看CacheStrategy中的Factory类

```
//CacheStrategy.Factory类
//构造方法
 public Factory(long nowMillis, Request request, Response cacheResponse) {
      this.nowMillis = nowMillis;
      this.request = request;
      this.cacheResponse = cacheResponse;

      if (cacheResponse != null) {
        this.sentRequestMillis = cacheResponse.sentRequestAtMillis();
        this.receivedResponseMillis = cacheResponse.receivedResponseAtMillis();
        Headers headers = cacheResponse.headers();

        //获取响应头的各种信息
        for (int i = 0, size = headers.size(); i < size; i++) {
          String fieldName = headers.name(i);
          String value = headers.value(i);
          if ("Date".equalsIgnoreCase(fieldName)) {
            servedDate = HttpDate.parse(value);
            servedDateString = value;
          } else if ("Expires".equalsIgnoreCase(fieldName)) {
            expires = HttpDate.parse(value);
          } else if ("Last-Modified".equalsIgnoreCase(fieldName)) {
            lastModified = HttpDate.parse(value);
            lastModifiedString = value;
          } else if ("ETag".equalsIgnoreCase(fieldName)) {
            etag = value;
          } else if ("Age".equalsIgnoreCase(fieldName)) {
            ageSeconds = HttpHeaders.parseSeconds(value, -1);
          }
        }
      }
    }
```

继续查看Factory的get方法

```
//CacheStrategy.Factory类
    public CacheStrategy get() {
      CacheStrategy candidate = getCandidate();

      //如果设置取消缓存
      if (candidate.networkRequest != null &&
request.cacheControl().onlyIfCached()) {
        // We're forbidden from using the network and the cache is insufficient.
        return new CacheStrategy(null, null);
      }

      return candidate;
    }
```

继续查看getCandidate()方法,可以看出，在这个方法里，就是最终决定缓存策略的方法

```
//CacheStrategy.Factory类
private CacheStrategy getCandidate() {
      // No cached response.
      //如果没有response的缓存，那就使用请求。
      if (cacheResponse == null) {
        return new CacheStrategy(request, null);
      }

      // Drop the cached response if it's missing a required handshake.
      //如果请求是https的并且没有握手，那么重新请求。
      if (request.isHttps() && cacheResponse.handshake() == null) {
        return new CacheStrategy(request, null);
      }

      // If this response shouldn't have been stored, it should never be used
      // as a response source. This check should be redundant as long as the
      // persistence store is well-behaved and the rules are constant.
      //如果response是不该被缓存的，就请求，isCacheable()内部是根据状态码判断的。
      if (!isCacheable(cacheResponse, request)) {
        return new CacheStrategy(request, null);
      }
      //如果请求指定不使用缓存响应，或者是可选择的，就重新请求。
      CacheControl requestCaching = request.cacheControl();
      if (requestCaching.noCache() || hasConditions(request)) {
        return new CacheStrategy(request, null);
      }

      //强制使用缓存
      CacheControl responseCaching = cacheResponse.cacheControl();
      if (responseCaching.immutable()) {
        return new CacheStrategy(null, cacheResponse);
      }

      long ageMillis = cacheResponseAge();
      long freshMillis = computeFreshnessLifetime();

      if (requestCaching.maxAgeSeconds() != -1) {
        freshMillis = Math.min(freshMillis,
SECONDS.toMillis(requestCaching.maxAgeSeconds()));
      }

      long minFreshMillis = 0;
      if (requestCaching.minFreshSeconds() != -1) {
        minFreshMillis = SECONDS.toMillis(requestCaching.minFreshSeconds());
      }

      long maxStaleMillis = 0;
      if (!responseCaching.mustRevalidate() && requestCaching.maxStaleSeconds()
!= -1) {
        maxStaleMillis = SECONDS.toMillis(requestCaching.maxStaleSeconds());
      }

       //如果response有缓存，并且时间比较近，添加一些头部信息后，返回request = null的策略
       /（意味着虽过期，但可用，只是会在响应头添加warning）
      if (!responseCaching.noCache() && ageMillis + minFreshMillis < freshMillis
+ maxStaleMillis) {
        Response.Builder builder = cacheResponse.newBuilder();
        if (ageMillis + minFreshMillis >= freshMillis) {
          builder.addHeader("Warning", "110 HttpURLConnection \"Response is
stale\"");
        }
        long oneDayMillis = 24 * 60 * 60 * 1000L;
        if (ageMillis > oneDayMillis && isFreshnessLifetimeHeuristic()) {
          builder.addHeader("Warning", "113 HttpURLConnection \"Heuristic
expiration\"");
        }
        return new CacheStrategy(null, builder.build());
      }

      // Find a condition to add to the request. If the condition is satisfied,
the response body
      // will not be transmitted.
      String conditionName;
      //流程走到这，说明缓存已经过期了
      //添加请求头：If-Modified-Since或者If-None-Match
      //etag与If-None-Match配合使用
      //lastModified与If-Modified-Since配合使用
      //前者和后者的值是相同的
      //区别在于前者是响应头，后者是请求头。
      //后者用于服务器进行资源比对，看看是资源是否改变了。
      // 如果没有，则本地的资源虽过期还是可以用的      String conditionValue;

      if (etag != null) {
        conditionName = "If-None-Match";
        conditionValue = etag;
      } else if (lastModified != null) {
        conditionName = "If-Modified-Since";
        conditionValue = lastModifiedString;
      } else if (servedDate != null) {
        conditionName = "If-Modified-Since";
        conditionValue = servedDateString;
      } else {
        return new CacheStrategy(request, null); // No condition! Make a regular
request.
      }

      Headers.Builder conditionalRequestHeaders =
request.headers().newBuilder();
      Internal.instance.addLenient(conditionalRequestHeaders, conditionName,
conditionValue);

      Request conditionalRequest = request.newBuilder()
          .headers(conditionalRequestHeaders.build())
          .build();
      return new CacheStrategy(conditionalRequest, cacheResponse);
    }
```

CacheStrategy的构造方法

```
 CacheStrategy(Request networkRequest, Response cacheResponse) {
    this.networkRequest = networkRequest;
    this.cacheResponse = cacheResponse;
  }
```

4.4.3执行策略

intercept中执行策略的部分

```
//intercept中
     //根据缓存策略，更新统计指标：请求次数、使用网络请求次数、使用缓存次数
    if (cache != null) {
      cache.trackResponse(strategy);
    }

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

    //如果既无网络请求可用，又无缓存，返回504错误
    // 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.
    //缓存可用，直接返回缓存
    if (networkRequest == null) {
      return cacheResponse.newBuilder()
          .cacheResponse(stripBody(cacheResponse))
          .build();
    }
```

4.4.4进行网络请求

intercept中进行网络请求的部分

```
//intercept中
 Response networkResponse = null;
    try {
      //进行网络请求-->调用下一个拦截器
      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) {

      //响应码为304，缓存有效，合并网络请求和缓存
      //304 请求资源未修改
      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()).
        //在合并头部之后更新缓存，但是在剥离内容编码头之前（由initContentStream（）执行）。
        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) {
      //如果有响应体并且可缓存，那么将响应写入缓存。
      if (HttpHeaders.hasBody(response) && CacheStrategy.isCacheable(response,
networkRequest)) {
        // Offer this request to the cache.
        CacheRequest cacheRequest = cache.put(response);
        return cacheWritingResponse(cacheRequest, response);
      }

      //如果request无效
      if (HttpMethod.invalidatesCache(networkRequest.method())) {
        try {
        //从缓存删除
          cache.remove(networkRequest);
        } catch (IOException ignored) {
          // The cache cannot be written.
        }
      }
    }

    return response;
```

4.5ConnectInterceptor类

ConnectInterceptor,是一个连接相关的拦截器,作用就是打开与服务器之间的连接，正式开启OkHttp的网络请求

首先还是先看ConnectInterceptor类的intercept方法

4.5.1 intercept

```
 @Override public Response intercept(Chain chain) throws IOException {
    RealInterceptorChain realChain = (RealInterceptorChain) chain;
    Request request = realChain.request();
    //首先从realChain拿到了streamAllocation对象，这个对象在RetryAndFollowInterceptor中
就已经初始化过了
    //只不过一直没有使用，到了ConnectTnterceptor才使用。
    StreamAllocation streamAllocation = realChain.streamAllocation();

    // We need the network to satisfy this request. Possibly for validating a
conditional GET.
    //判断是否为GET请求
    boolean doExtensiveHealthChecks = !request.method().equals("GET");
    //生成一个HttpCodec对象。这个对象是用于编码request和解码response的一个封装好的对象。
    HttpCodec httpCodec = streamAllocation.newStream(client, chain,
doExtensiveHealthChecks);
```

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

    //将创建好的HttpCode和connection对象传递给下一个拦截器
    return realChain.proceed(request, streamAllocation, httpCodec, connection);
  }
```

4.6CallServerInterceptor类

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。

4.6.1intercept

准备工作，首先是获得各种对象，然后将请求写入 httpCodec中

```
@Override public Response intercept(Chain chain) throws IOException {
    RealInterceptorChain realChain = (RealInterceptorChain) chain;
    HttpCodec httpCodec = realChain.httpStream();


    StreamAllocation streamAllocation = realChain.streamAllocation();
    //上一步已经完成连接工作的连接
    RealConnection connection = (RealConnection) realChain.connection();
    Request request = realChain.request();

    long sentRequestMillis = System.currentTimeMillis();

    realChain.eventListener().requestHeadersStart(realChain.call());
    //将请求头写入
    httpCodec.writeRequestHeaders(request);
    realChain.eventListener().requestHeadersEnd(realChain.call(), request);
```

再将请求头写入后，会有一个关于Expect:100-continue的请求头处理。

```
/**
    http 100-continue用于客户端在发送POST数据给服务器前，征询服务器情况，看服务器是否处理
POST的数据，如果不处理，客户端则不上传POST数据，如果处理，则POST上传数据。在现实应用中，通过在
POST大数据时，才会使用100-continue协议。如果服务器端可以处理，则会返回100，负责会返回错误码
*/
if ("100-continue".equalsIgnoreCase(request.header("Expect"))) { //如果有
Expect:100-continue的请求头
        httpCodec.flushRequest();
        realChain.eventListener().responseHeadersStart(realChain.call());
        responseBuilder = httpCodec.readResponseHeaders(true); //读取响应头
      }
```

当返回的结果为null，或者不存在Expect:100-continue的请求头，则执行下面的代码，

```
@Override public Response intercept(Chain chain) throws IOException {
    RealInterceptorChain realChain = (RealInterceptorChain) chain;
    HttpCodec httpCodec = realChain.httpStream();


    StreamAllocation streamAllocation = realChain.streamAllocation();
    //上一步已经完成连接工作的连接
    RealConnection connection = (RealConnection) realChain.connection();
    Request request = realChain.request();
    long sentRequestMillis = System.currentTimeMillis();

    realChain.eventListener().requestHeadersStart(realChain.call());
    //将请求头写入
    httpCodec.writeRequestHeaders(request);
    realChain.eventListener().requestHeadersEnd(realChain.call(), request);
```

如果没有经历上面的Expect:100-continue的请求头，则重新请求一次。

```
httpCodec.finishRequest();

    if (responseBuilder == null) {
      realChain.eventListener().responseHeadersStart(realChain.call());
      responseBuilder = httpCodec.readResponseHeaders(false);
    }
```

将请求的结果(可能是Expect:100-continue请求的结果，也可能是正常的情况下)包装成response。

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

如果请求的返回码为100(继续。客户端应继续其请求)

```
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); //重新请求一次

      response = responseBuilder //覆盖之前的响应
              .request(request)
              .handshake(streamAllocation.connection().handshake())
              .sentRequestAtMillis(sentRequestMillis)
              .receivedResponseAtMillis(System.currentTimeMillis())
              .build();

      code = response.code();
    }
```

判断是否是是websocket并且响应码为101(切换协议)

```
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)//赋空值
          .build();
    } else {
      response = response.newBuilder()
          .body(httpCodec.openResponseBody(response)) //填充response的body
          .build();
    }
```

从请求头和响应头判断其中是否有表明需要保持连接打开

```
  if ("close".equalsIgnoreCase(response.request().header("Connection"))
        || "close".equalsIgnoreCase(response.header("Connection"))) {
      streamAllocation.noNewStreams();
    }
```

处理204(无内容)和205(重置内容)

```
 if ((code == 204 || code == 205) && response.body().contentLength() > 0) {
      throw new ProtocolException(
          "HTTP " + code + " had non-zero Content-Length: " +
response.body().contentLength());
    }
```