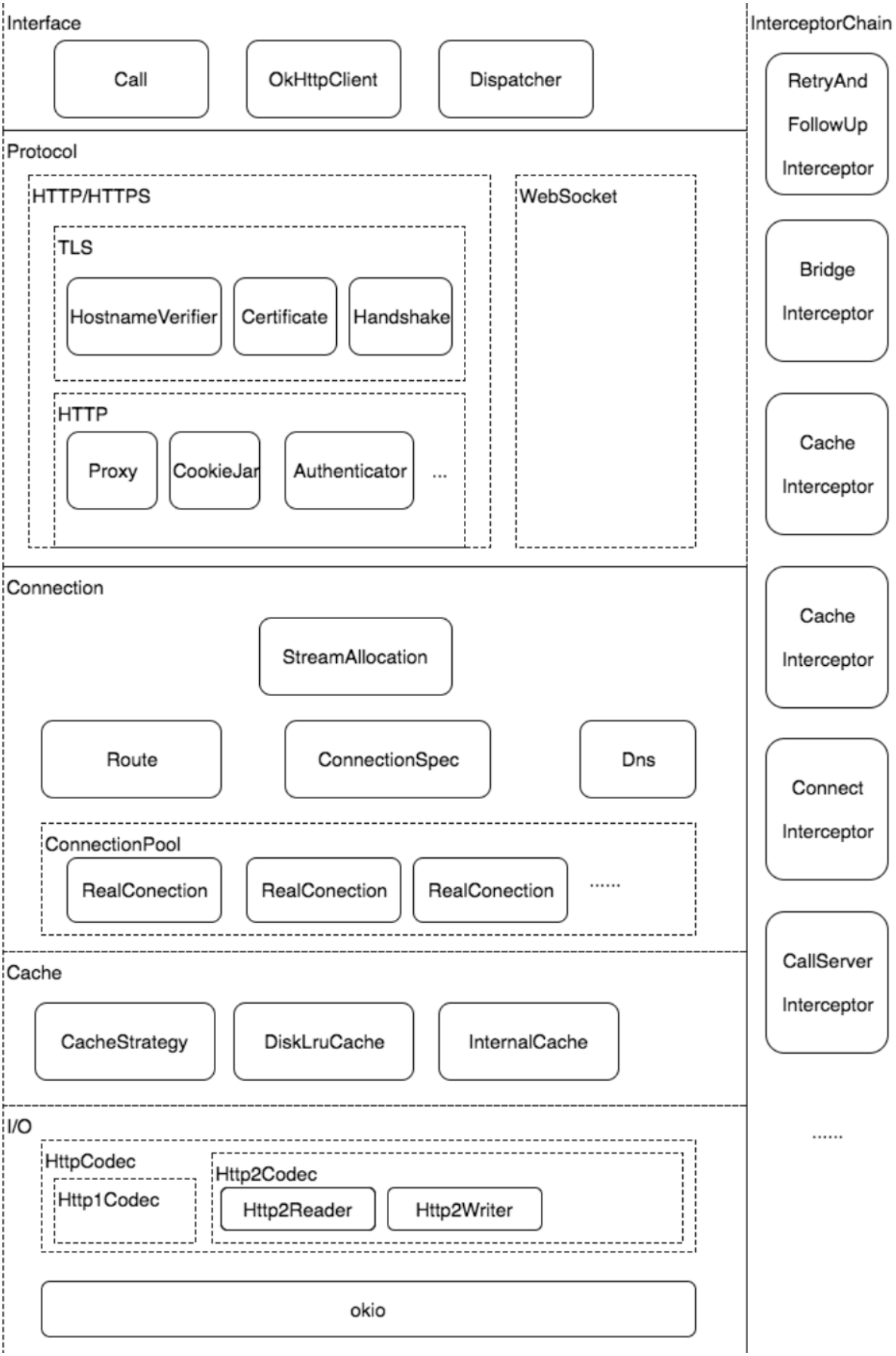
OkHttp

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


Interface
Call
OkHttpClient
Dispatcher
InterceptorChain
RetryAnd
FollowUp
Interceptor
Protocol
HTTP/HTTPS
TLS
HostnameVerifier
Certificate
Handshake
HTTP
Proxy
CookieJar
Authenticator
...
WebSocket
Bridge
Interceptor
Cache
Interceptor
Connection
StreamAllocation
Route
ConnectionSpec
Dns
ConnectionPool
RealConection
RealConection
RealConection
.....
Cache
Interceptor
Connect
Interceptor
Cache
CacheStrategy
DiskLruCache
InternalCache
CallServer
Interceptor
I/O
HttpCodec
Http1Codec
Http2Codec
Http2Reader
Http2Writer
.....
okio

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);
    }
  }
```

定义了请求对象后，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重定向拦截器
    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

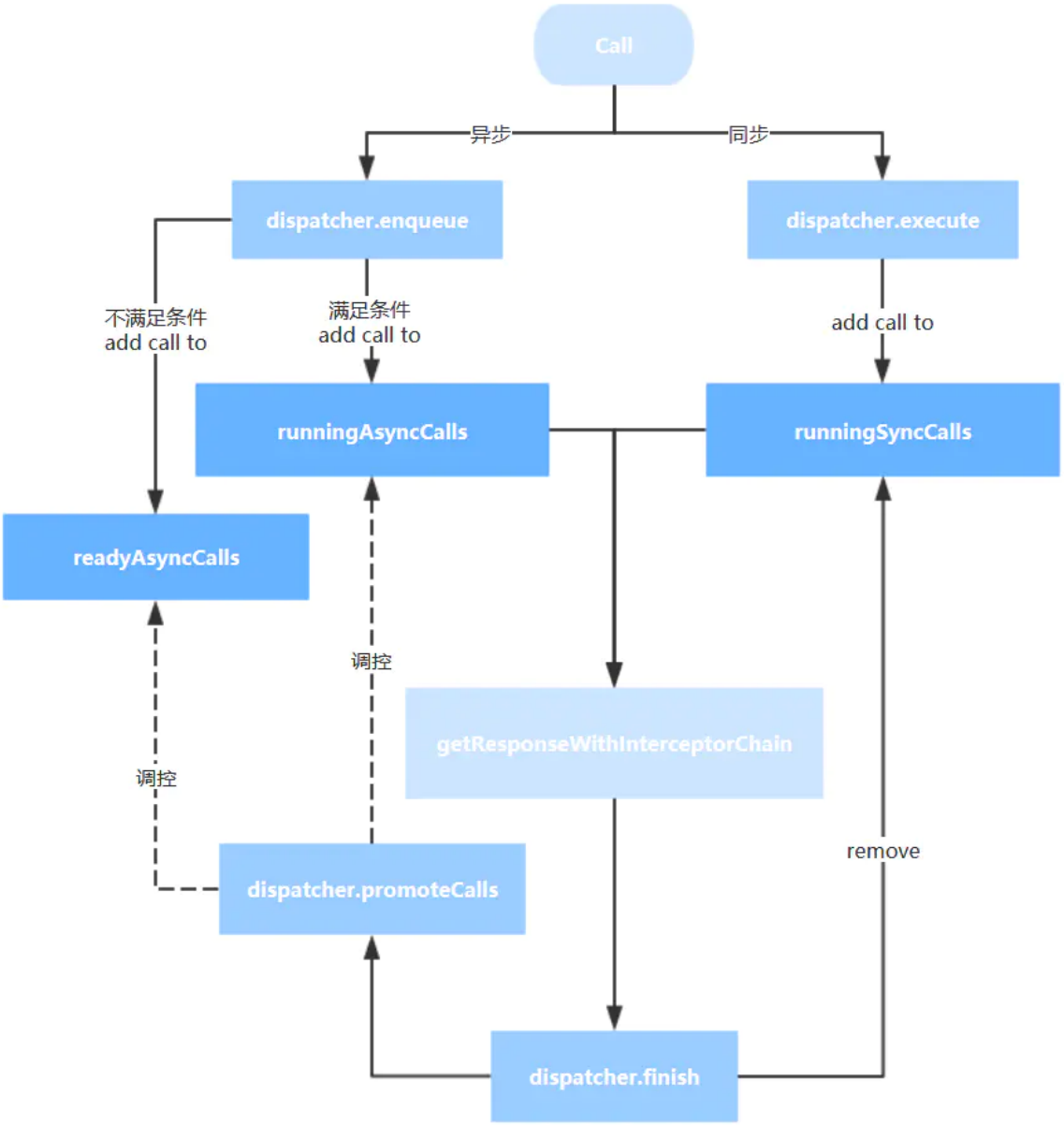
```
//Dispatcher
  /** Used by {@code AsyncCall#run} to signal completion. */
  void finished(AsyncCall call) {
    finished(runningAsyncCalls, call, true);
  }

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

```
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

```
//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client和请求的信息初始化了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

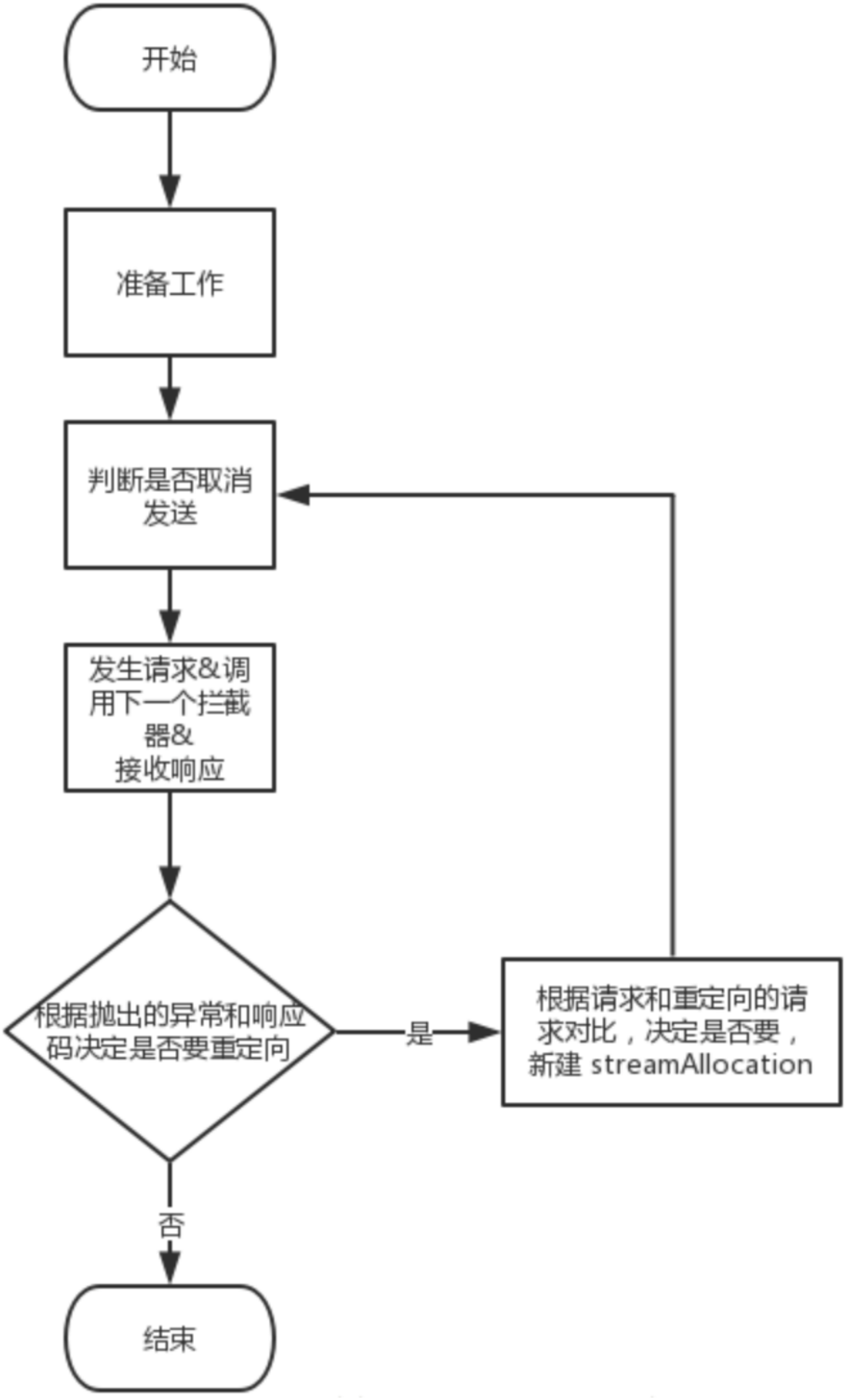
```
  private Request followUpRequest(Response userResponse, Route route) throws
IOException {
    if (userResponse == null) throw new IllegalStateException();
    //返回的响应码
    int responseCode = userResponse.code();

    //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;
    }
  }
```

4.2.1.4流程图



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();
    }
```

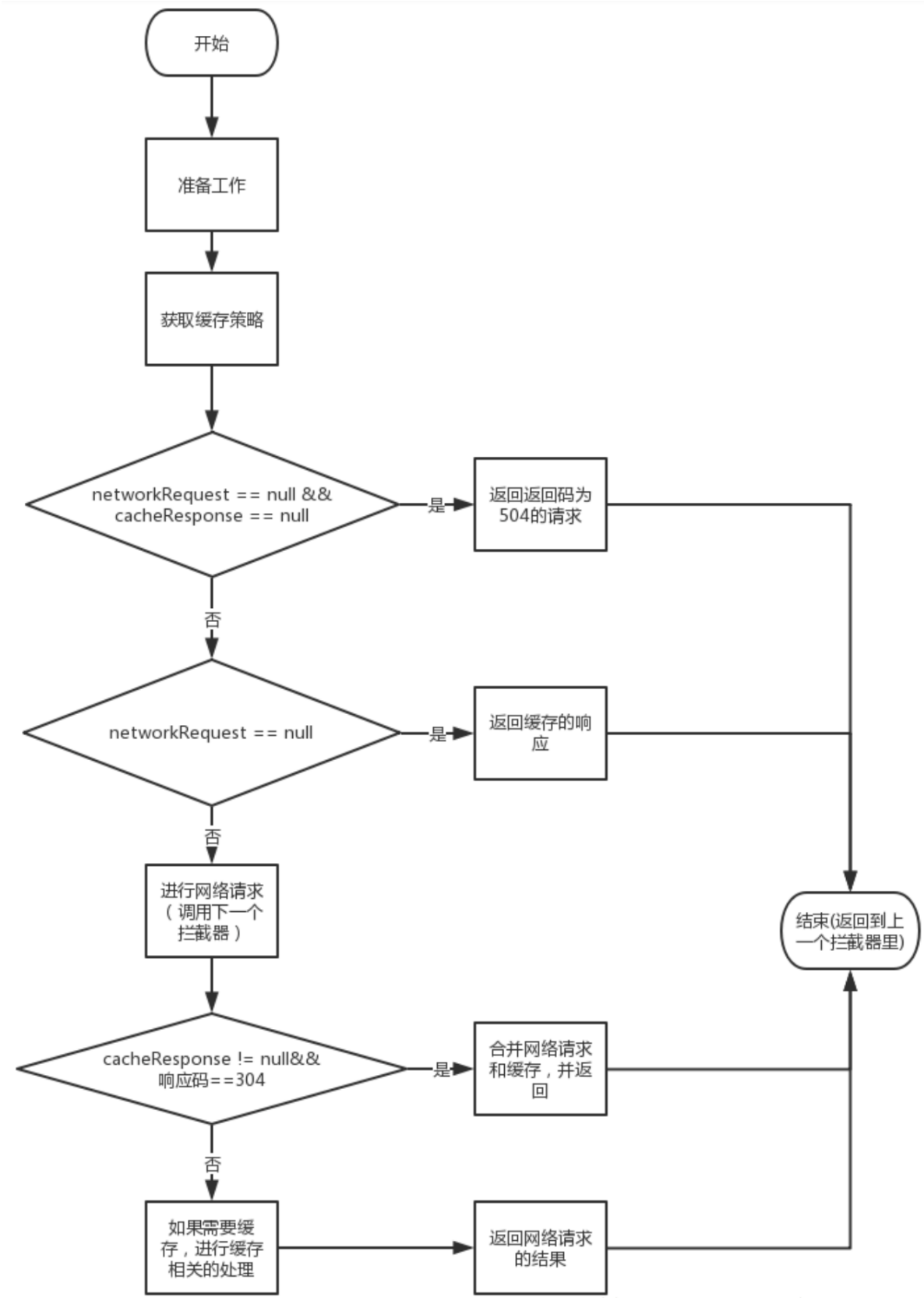
4.4.4进行网络请求

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;
```

4.4.5流程图



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();
    }
```

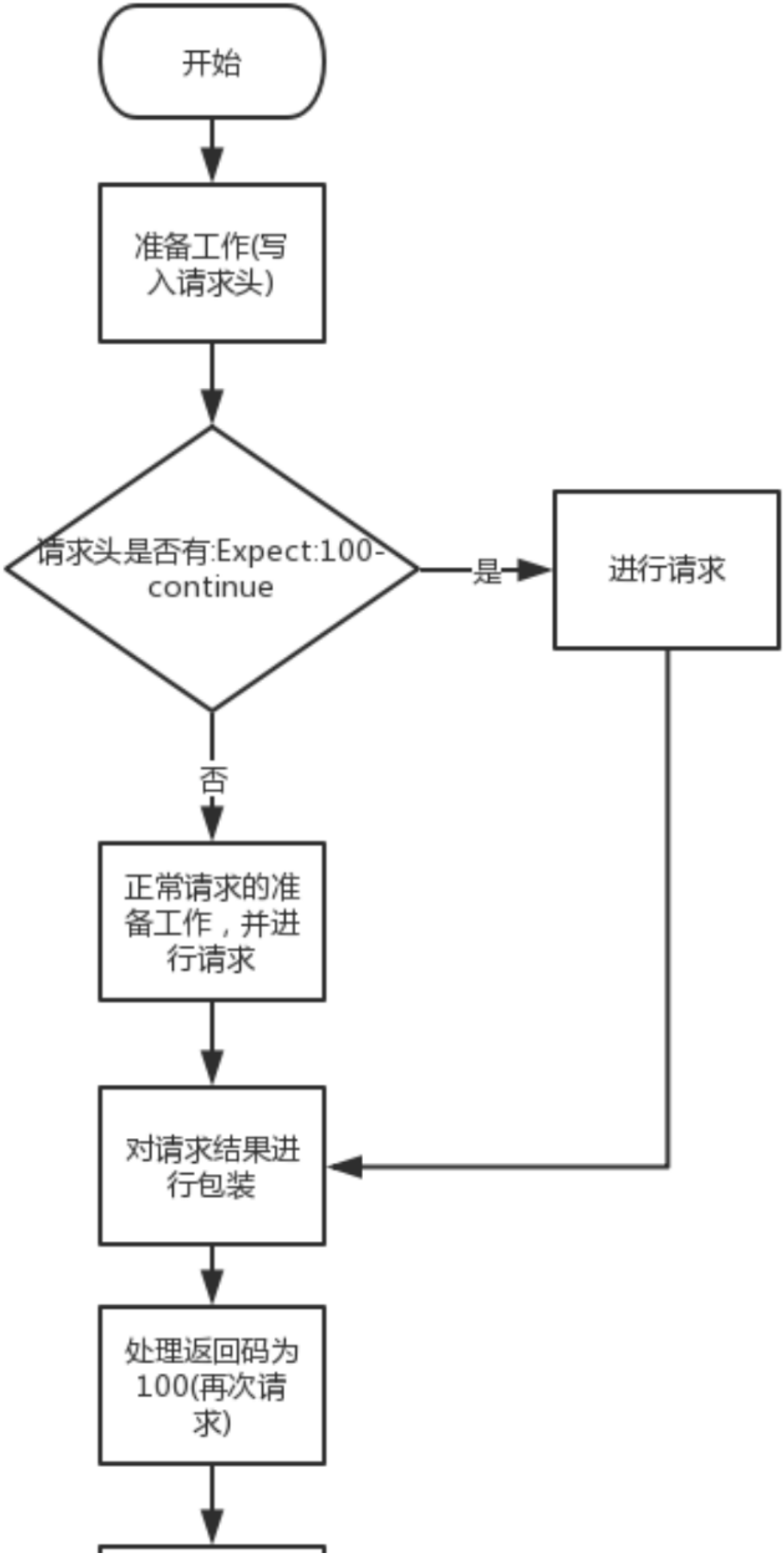
从请求头和响应头判断其中是否有表明需要保持连接打开

```
  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)

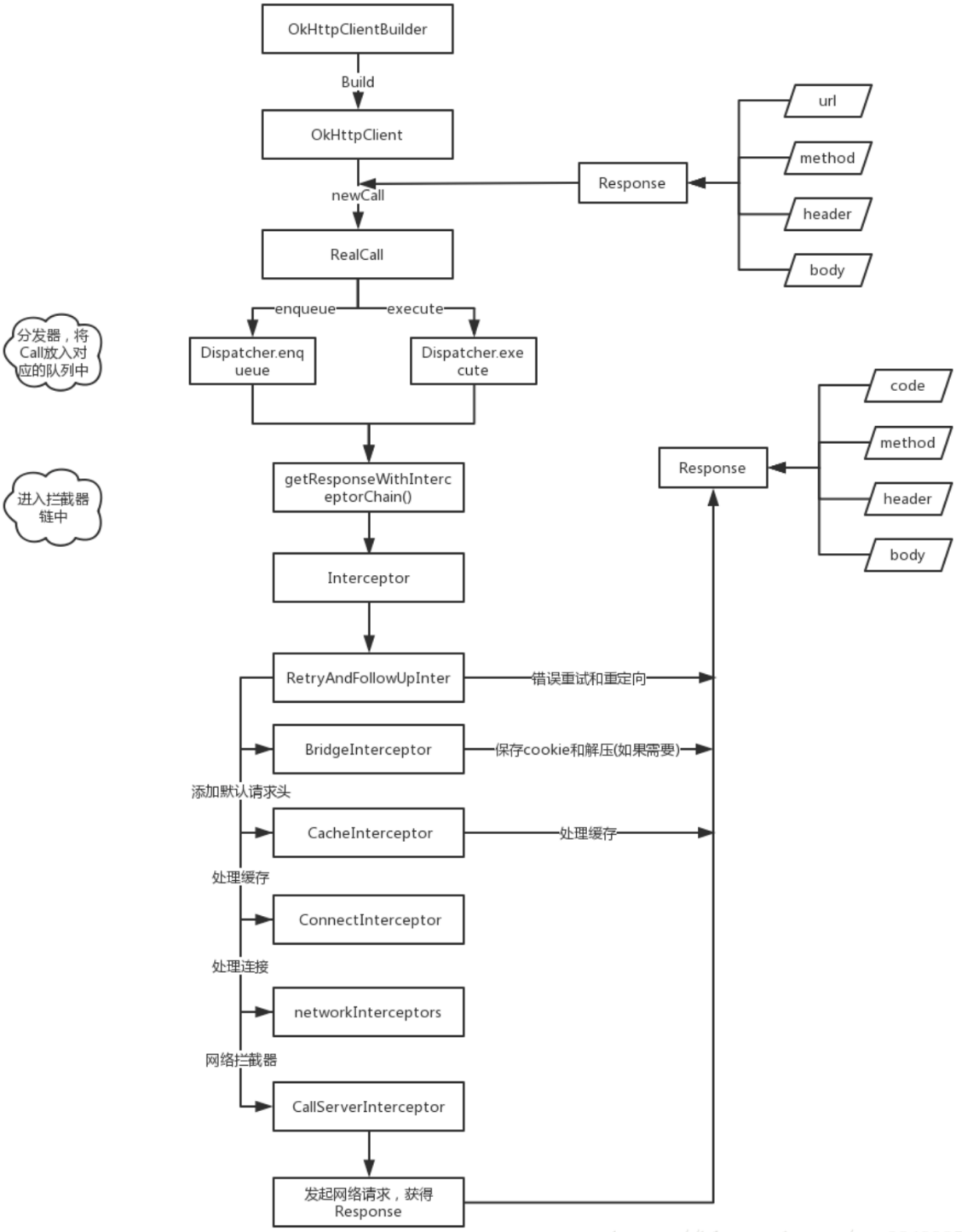
```
 if ((code == 204 || code == 205) && response.body().contentLength() > 0) {
      throw new ProtocolException(
          "HTTP " + code + " had non-zero Content-Length: " +
response.body().contentLength());
    }
```

4.6.2流程图


开始
准备工作(写入请求头)
请求头是否有:Expect:100-continue
是
进行请求
否
正常请求的准备工作，并进行请求
对请求结果进行包装
处理返回码为100(再次请求)


处理返回码
101,204,205
结束，将响应传给上一个拦截器

4.7总结


OkHttpClientBuilder
Build
OkHttpClient
newCall
RealCall
enqueue
execute
Dispatcher.enqueue
Dispatcher.execute
分发器，将Call放入对应的队列中
getResponseWithInterceptorChain()
进入拦截器链中
Interceptor
RetryAndFollowUpInter
错误重试和重定向
BridgeInterceptor
保存cookie和解压(如果需要)
添加默认请求头
CacheInterceptor
处理缓存
处理缓存
ConnectInterceptor
处理连接
networkInterceptors
网络拦截器
CallServerInterceptor
发起网络请求，获得Response
Response
url
method
header
body
Response
code
method
header
body

5网络操作

5.1名次解释

5.1.1URL

URL是HTTP和Interbe的基础。除了作为Web上所有内容的通用，分散的命名方案之外，它们还指定了如何访问Web资源。还可以指定访问时使用Http还是Https。

5.1.2Addresses

Addresses指定网络服务器，和所有的静态必要的配置，已经连接到该服务器：端口号，Https设置，和首选的的网络协议(如Http.2或者SPDY)。

共享相同地址的URL也可以共享相同的基础TCP套接字，共享一个连接有一个很好的性能优点:更低的延迟，更高的吞吐量（由于TCP的慢启动）和省电，OkHttp使用连接池自动再利用HTTP/1.x的连接，复用HTTP/2和SPDY的连接。

在OkHttp中，address的一些字段来自URL(模式、主机名、端口)，剩下的部分来自OkHttpClient。

5.1.3Routes

Routes提供一个真的连接到网络服务器的所需的动态信息。这指定了尝试的Ip地址(或者讲过DNS查询得到的地址)，使用的代理服务器(如果使用了ProxySelector)和使用哪个版本的TLS进行谈判。(对于HTTPS连接)

对于一个地址，可能有多个路由。举个例子，一个网路服务器托管在多个数据中心，那么在DNS中可能会产生多个IP地址。

5.1.4Connection

Connection:物理连接的封装
ConnectionPool:连接池，实现连接的复用；

5.1.5StreamAllocation

直译就是流分配。流是什么呢？我们知道Connection是一个连接远程服务器的物理Socket连接，而Stream则是基于Connection的逻辑Http请求/响应对。StreamAllocation会通过ConnectPool获取或者新生成一个RealConnection来得到一个连接到Server的Connection连接，同时会生成一个HttpCodec用于下一个CallServerInterceptor，以完成最终的请求；

5.1.6URL请求的过程

当使用OkHttp进行一个URL请求时，下面是他的操作流程：

使用URL和配置好的OkHttpClient创建一个address。这个地址指明我们如何连接网络服务器。
尝试从连接池中得到该地址的一条连接
如果在连接池中没有找到一条连接，那么选择一个route进行尝试。通常这意味着做一个DNS请求得到服务器IP的地址，必要是会选择一个TSL版本和一个代理服务器。
如果是一条新的路由，那么建立一条直接的socket连接或者TLS通道或者一个直接的TLS连接。
发生Http请求，读取响应
如果连接出现问题，OkHttp会选择另外一条路由进行再次尝试。这使得OkHttp在一个服务器的一些地址不可用时仍然可用。
一旦读取到相应后，连接将会退换到连接池中，方便复用。连接在池中闲置一段时间后将会被释放。

5.2Address的创建

Address的创建在RetryAndFollowupInterceptor中的createAddress方法中，代码如下：

```
 private Address createAddress(HttpUrl url) {
    SSLSocketFactory sslSocketFactory = null;
    HostnameVerifier hostnameVerifier = null;
    CertificatePinner certificatePinner = null;

    //是否是Hpptps
    if (url.isHttps()) {
      sslSocketFactory = client.sslSocketFactory();
      hostnameVerifier = client.hostnameVerifier();
      certificatePinner = client.certificatePinner();
    }

    //可以看出了Address的信息一部分由URL提供，主要包括主机名和端口；另一部分由OkHttpClient
提供，如dns、socketFactory等等。
    return new Address(url.host(), url.port(), client.dns(),
client.socketFactory(),
        sslSocketFactory, hostnameVerifier, certificatePinner,
client.proxyAuthenticator(),
        client.proxy(), client.protocols(), client.connectionSpecs(),
client.proxySelector());
  }
```

查看Address的构造方法,发现在Address的构造方法里，将相关的参数保存起来。

```
  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;
  }
```

5.3StreamAllocation的创建

StreamAllocation类负责管理连接，流和请求三者之间的关系，其创建在RetryAndFollowupInterceptor的intercept方法中，使用OkHttpClient的连接池以及上面创建的Address进行初始化，代码如下：

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

5.3.1connetionpool

client的connectionPool参数的连接池是在OkHttpClient.Builder中设置的，而其设置在Builder的构造方法中，调用的是ConnectionPool的默认构造方法，代码如下：

```
 public Builder() {
      ...
      //默认连接池
      connectionPool = new ConnectionPool();
      dns = Dns.SYSTEM;
      followSslRedirects = true;
      followRedirects = true;
      retryOnConnectionFailure = true;
      connectTimeout = 10_000;
      readTimeout = 10_000;
      writeTimeout = 10_000;
    }
```

查看 ConnectionPool的

```
  public ConnectionPool() {
    this(5, 5, TimeUnit.MINUTES);
  }

  public ConnectionPool(int maxIdleConnections, long keepAliveDuration, TimeUnit
timeUnit) {
    this.maxIdleConnections = maxIdleConnections;
    this.keepAliveDurationNs = timeUnit.toNanos(keepAliveDuration);

    // Put a floor on the keep alive duration, otherwise cleanup will spin loop.
    if (keepAliveDuration <= 0) {
      throw new IllegalArgumentException("keepAliveDuration <= 0: " +
keepAliveDuration);
    }
  }
```

从上面可以看到，默认的连接池的最大空闲连接数为5，最长存活时间为5min。

5.3.2StreamAllocation

StreamAllocation的构造方法,就是保存了相应的参数

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

5.4Connection

httpCodec：针对不同的版本，OkHttp为我们提供了HttpCodec1（Http1.x）和HttpCodec2(Http2).

Connection:物理连接。

5.4.1ConnectInterceptor

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

    // We need the network to satisfy this request. Possibly for validating a
conditional GET.
    boolean doExtensiveHealthChecks = !request.method().equals("GET");
    HttpCodec httpCodec = streamAllocation.newStream(client, chain,
doExtensiveHealthChecks);
    RealConnection connection = streamAllocation.connection();

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

streamAllocation.newStream方法我们之后再分析，先看connection()方法

```
public synchronized RealConnection connection() {
    return connection;
  }
```

进去后发现，它就是很简单的返回connection。那么connection是在什么时候赋值的

```
  public void acquire(RealConnection connection, boolean reportedAcquired) {
    assert (Thread.holdsLock(connectionPool));
    if (this.connection != null) throw new IllegalStateException();

    this.connection = connection;
    this.reportedAcquired = reportedAcquired;
    connection.allocations.add(new StreamAllocationReference(this,
callStackTrace));
  }
```

```
 public Socket releaseAndAcquire(RealConnection newConnection) {
    assert (Thread.holdsLock(connectionPool));
    if (codec != null || connection.allocations.size() != 1) throw new
IllegalStateException();

    // Release the old connection.
    Reference<StreamAllocation> onlyAllocation = connection.allocations.get(0);
    Socket socket = deallocate(true, false, false);

    // Acquire the new connection.
    this.connection = newConnection;
    newConnection.allocations.add(onlyAllocation);

    return socket;
  }
```

acquire是在创建新连接和从连接池里获得是，调用的。

5.4.2newStream

首先来看HttpCodec的对象的创建是调用streamAllocation.newStream方法，查看streamAllocation.newStream方法是在连接池的deduplicate返回中调用

```
 public HttpCodec newStream(
      OkHttpClient client, Interceptor.Chain chain, boolean
doExtensiveHealthChecks) {
    //得到连接时长，读取超时已经写超时参数
    int connectTimeout = chain.connectTimeoutMillis();
    int readTimeout = chain.readTimeoutMillis();
    int writeTimeout = chain.writeTimeoutMillis();
    int pingIntervalMillis = client.pingIntervalMillis();
    boolean connectionRetryEnabled = client.retryOnConnectionFailure();

    try {
      //得到一个健康的连接
      RealConnection resultConnection = findHealthyConnection(connectTimeout,
readTimeout,
          writeTimeout, pingIntervalMillis, connectionRetryEnabled,
doExtensiveHealthChecks);
      //获得HttpCodec
      HttpCodec resultCodec = resultConnection.newCodec(client, chain, this);

      synchronized (connectionPool) {
        codec = resultCodec;
        return resultCodec;
      }
    } catch (IOException e) {
      throw new RouteException(e);
    }
  }
```

继续查看findHealthyConnection方法

```
private RealConnection findHealthyConnection(int connectTimeout, int
readTimeout,
      int writeTimeout, int pingIntervalMillis, boolean connectionRetryEnabled,
      boolean doExtensiveHealthChecks) throws IOException {

    //死循环
    while (true) {
      //获得一个健康的连接
      RealConnection candidate = findConnection(connectTimeout, readTimeout,
writeTimeout,
          pingIntervalMillis, connectionRetryEnabled);

      // If this is a brand new connection, we can skip the extensive health
checks
      // 如果是一个全新的连接，跳过额外的健康检查
      synchronized (connectionPool) {
        if (candidate.successCount == 0) {
          return candidate;
        }
      }
```

```
      // Do a (potentially slow) check to confirm that the pooled connection is
still good. If it
      // isn't, take it out of the pool and start again.
      //如果候选连接通不过额外的健康检查，那么继续寻找一个新的候选连接
      if (!candidate.isHealthy(doExtensiveHealthChecks)) {
        noNewStreams();
        continue;
      }

      return candidate;
    }
  }
```

这个方法用于查找一条健康的连接并返回，如果连接不健康，那么就会重复查找。
上面的代码总结一下就是下面几种情况。

- 候选连接是一个新连接，直接返回
- 候选连接不是一个全新连接，但是是健康的，也直接返回
- 候选连接不上全新连接，并且不健康，那么继续下一轮的循环。

获取连接的方法 findConnection

```
private RealConnection findConnection(int connectTimeout, int readTimeout, int
writeTimeout,
      int pingIntervalMillis, boolean connectionRetryEnabled) throws IOException
{
    boolean foundPooledConnection = false;
    RealConnection result = null;
    Route selectedRoute = null;
    Connection releasedConnection;
    Socket toClose;
   //加锁
    synchronized (connectionPool) {
      //处理异常
      if (released) throw new IllegalStateException("released");
      if (codec != null) throw new IllegalStateException("codec != null");
      if (canceled) throw new IOException("Canceled");

      // Attempt to use an already-allocated connection. We need to be careful
here because our
      // already-allocated connection may have been restricted from creating new
streams.
      releasedConnection = this.connection;
      toClose = releaseIfNoNewStreams();

        //存在可使用的已分配连接
      if (this.connection != null) {
        // We had an already-allocated connection and it's good.
        //赋值
        result = this.connection;
        releasedConnection = null;
      }
      if (!reportedAcquired) {
        // If the connection was never reported acquired, don't report it as
released!
        releasedConnection = null;
      }
```

```
      //没有可使用的连接，去连接池(连接池在后面会讲解)中找
      if (result == null) {
        // Attempt to get a connection from the pool.
        //去连接池中查找
        Internal.instance.get(connectionPool, address, this, null);

        if (connection != null) {
          foundPooledConnection = true;
          result = connection;
        } else {
          selectedRoute = route;
        }
      }
    }
    closeQuietly(toClose);

    if (releasedConnection != null) {
      eventListener.connectionReleased(call, releasedConnection);
    }
    if (foundPooledConnection) {
      eventListener.connectionAcquired(call, result);
    }
    if (result != null) {
      // If we found an already-allocated or pooled connection, we're done.
     //找到了一个已分配或者连接池中的连接，此过程结束，返回
      return result;
    }

    }

     //否则，我们需要一个路由信息，这是一个阻塞的操作
    // If we need a route selection, make one. This is a blocking operation.
    boolean newRouteSelection = false;
    if (selectedRoute == null && (routeSelection == null ||
!routeSelection.hasNext())) {
      newRouteSelection = true;
      routeSelection = routeSelector.next();
    }

    synchronized (connectionPool) {
      if (canceled) throw new IOException("Canceled");

      if (newRouteSelection) {
        // Now that we have a set of IP addresses, make another attempt at
getting a connection from
        // the pool. This could match due to connection coalescing.
        //提供更加全面的路由信息，再次从连接池中获取连接
        List<Route> routes = routeSelection.getAll();
        for (int i = 0, size = routes.size(); i < size; i++) {
          Route route = routes.get(i);
          Internal.instance.get(connectionPool, address, this, route);
          if (connection != null) {
            foundPooledConnection = true;
            result = connection;
            this.route = route;
            break;
          }
```

```
        }
      }

     //依然没有找到，生成新的连接
      if (!foundPooledConnection) {
        if (selectedRoute == null) {
          selectedRoute = routeSelection.next();
        }

        // Create a connection and assign it to this allocation immediately.
This makes it possible
        // for an asynchronous cancel() to interrupt the handshake we're about
to do.
        route = selectedRoute;
        refusedStreamCount = 0;
        result = new RealConnection(connectionPool, selectedRoute);
        //将新连接保存到this.connection中
        acquire(result, false);
      }
    }

    // If we found a pooled connection on the 2nd time around, we're done.
    //如果连接是从连接池中找到的，说明是可复用的。不是新生成的，因为新生成的连接，
    // 需要去连接服务器之后才能可用呀
    if (foundPooledConnection) {
      eventListener.connectionAcquired(call, result);
      return result;
    }

    // Do TCP + TLS handshakes. This is a blocking operation.
    //新连接 练接server
    result.connect(connectTimeout, readTimeout, writeTimeout,
pingIntervalMillis,
        connectionRetryEnabled, call, eventListener);
    routeDatabase().connected(result.route());

    Socket socket = null;
    synchronized (connectionPool) {
      reportedAcquired = true;

      // Pool the connection.
      //将新连接放入请求池中
      Internal.instance.put(connectionPool, result);

      // If another multiplexed connection to the same address was created
concurrently, then
      // release this connection and acquire that one.
      //如果是一个http2连接
      //确保其多路复用的特性。
      if (result.isMultiplexed()) {
        socket = Internal.instance.deduplicate(connectionPool, address, this);
        result = connection;
      }
    }
    closeQuietly(socket);

    eventListener.connectionAcquired(call, result);
    return result;
```

```
  }
```

这里关于连接的查找：

如果有连接，直接用
没有可用的连接，第一次去连接池中查找，找到后直接用
没有找到，补充路由信息，在连接池中二次查找。
依然没有找到，创建新连接，然后连接server，将其放入到连接池中

下面看连接是如何建立连接的，在findConnection方法中，当创建了一个新的Connection后，调用了其connect方法，connect负责将客户端Socket连接到服务端Socket，代码如下：

```
// Do TCP + TLS handshakes. This is a blocking operation.
    //新连接 练接server
    result.connect(connectTimeout, readTimeout, writeTimeout,
pingIntervalMillis,
        connectionRetryEnabled, call, eventListener);
```

```
//RealConnection类
 public void connect(int connectTimeout, int readTimeout, int writeTimeout,
      int pingIntervalMillis, boolean connectionRetryEnabled, Call call,
      EventListener eventListener) {
    if (protocol != null) throw new IllegalStateException("already connected");

    RouteException routeException = null;
    List<ConnectionSpec> connectionSpecs = route.address().connectionSpecs();
    ConnectionSpecSelector connectionSpecSelector = new
ConnectionSpecSelector(connectionSpecs);

   //不是HTTPS协议
    if (route.address().sslSocketFactory() == null) {
      if (!connectionSpecs.contains(ConnectionSpec.CLEARTEXT)) {
        throw new RouteException(new UnknownServiceException(
            "CLEARTEXT communication not enabled for client"));
      }
      String host = route.address().url().host();
      if (!Platform.get().isCleartextTrafficPermitted(host)) {
        throw new RouteException(new UnknownServiceException(
            "CLEARTEXT communication to " + host + " not permitted by network
security policy"));
      }
    }

    while (true) {
      try {
        if (route.requiresTunnel()) {
          connectTunnel(connectTimeout, readTimeout, writeTimeout, call,
eventListener);
          if (rawSocket == null) {
            // We were unable to connect the tunnel but properly closed down our
resources.
            break;
          }
        } else {
```

```
          connectSocket(connectTimeout, readTimeout, call, eventListener);//创建
Socket以及连接Socke
        }
        establishProtocol(connectionSpecSelector, pingIntervalMillis, call,
eventListener);
        eventListener.connectEnd(call, route.socketAddress(), route.proxy(),
protocol);
        break;
      } catch (IOException e) { //处理异常
        closeQuietly(socket);//清理各种数据，进入下一次循环
        closeQuietly(rawSocket);
        socket = null;
        rawSocket = null;
        source = null;
        sink = null;
        handshake = null;
        protocol = null;
        http2Connection = null;

        eventListener.connectFailed(call, route.socketAddress(), route.proxy(),
null, e);

        if (routeException == null) {
          routeException = new RouteException(e);
        } else {
          routeException.addConnectException(e);
        }

        if (!connectionRetryEnabled ||
!connectionSpecSelector.connectionFailed(e)) {
          throw routeException;
        }
      }
    }

    if (route.requiresTunnel() && rawSocket == null) {
      ProtocolException exception = new ProtocolException("Too many tunnel
connections attempted: "
          + MAX_TUNNEL_ATTEMPTS);
      throw new RouteException(exception);
    }

    if (http2Connection != null) {
      synchronized (connectionPool) {
        allocationLimit = http2Connection.maxConcurrentStreams();
      }
    }
  }
```

在这里出现两个很重要的方法，connectSocket和 establishProtocol

connectSocket为创建Socket以及连接Socket

```
//RealConnection类
 /** Does all the work necessary to build a full HTTP or HTTPS connection on a
raw socket. */
  private void connectSocket(int connectTimeout, int readTimeout, Call call,
```

```
      EventListener eventListener) throws IOException {
    //首先获取代理和地址
    Proxy proxy = route.proxy();
    Address address = route.address();

    //建立socket
    //代理的类型是使用SocketFactory工厂创建无参的rawSocket
    //还是使用带代理参数的Socket构造方法，得到了rawSocket对象
    rawSocket = proxy.type() == Proxy.Type.DIRECT || proxy.type() ==
Proxy.Type.HTTP
        ? address.socketFactory().createSocket()
        : new Socket(proxy);

    eventListener.connectStart(call, route.socketAddress(), proxy);
    rawSocket.setSoTimeout(readTimeout);
    try {
       //连接socket
       //调用connectSocket进行Socket的连接
       //Plateform.get()方法返回不同平台的信息
      Platform.get().connectSocket(rawSocket, route.socketAddress(),
connectTimeout);
    } catch (ConnectException e) {
      ConnectException ce = new ConnectException("Failed to connect to " +
route.socketAddress());
      ce.initCause(e);
      throw ce;
    }

    // The following try/catch block is a pseudo hacky way to get around a crash
on Android 7.0
    // More details:
    // https://github.com/square/okhttp/issues/3245
    // https://android-review.googlesource.com/#/c/271775/
    try {
      //使用Okio封装Socket的输入输出流
      source = Okio.buffer(Okio.source(rawSocket));
      sink = Okio.buffer(Okio.sink(rawSocket));
    } catch (NullPointerException npe) {
      if (NPE_THROW_WITH_NULL.equals(npe.getMessage())) {
        throw new IOException(npe);
      }
    }
  }
```

Plateform.get()方法返回不同平台的信息，因为OkHttp是可以用于Android和Java平台的，而Java又有多个版本，所以进行了平台判断。get()是一个单例，其初始化在findPlatform方法中，如下：

```
//Platform
 private static Platform findPlatform() {
    Platform android = AndroidPlatform.buildIfSupported();

    if (android != null) {
      return android;
    }

    if (isConscryptPreferred()) {
      Platform conscrypt = ConscryptPlatform.buildIfSupported();
```

```
      if (conscrypt != null) {
        return conscrypt;
      }
    }

    Platform jdk9 = Jdk9Platform.buildIfSupported();

    if (jdk9 != null) {
      return jdk9;
    }

    Platform jdkWithJettyBoot = JdkWithJettyBootPlatform.buildIfSupported();

    if (jdkWithJettyBoot != null) {
      return jdkWithJettyBoot;
    }

    // Probably an Oracle JDK like OpenJDK.
    return new Platform();
  }
```

可以看到findPlatform分为了android平台、jdk9、有JettyBoot的jdk还有默认的平台几类。这边看默认的Platform就可以了

继续看connectSocket

```
// Platform类中
 public void connectSocket(Socket socket, InetSocketAddress address,
      int connectTimeout) throws IOException {
    socket.connect(address, connectTimeout);
  }
```

可以看到就是调用socket的connect方法，至此，本地Socket与后台Socket建立了连接，并得到了输入输出流。

重新回到connect方法中，看 connectSocket方法下的establishProtocol方法

```
//RealConnection类
private void establishProtocol(ConnectionSpecSelector connectionSpecSelector,
      int pingIntervalMillis, Call call, EventListener eventListener) throws
IOException {
      //如果不是Hppts
    if (route.address().sslSocketFactory() == null) {
      protocol = Protocol.HTTP_1_1;
      socket = rawSocket;
      return;
    }

    eventListener.secureConnectStart(call);
    connectTls(connectionSpecSelector);
    eventListener.secureConnectEnd(call, handshake);

   //如过是Http2协议
    if (protocol == Protocol.HTTP_2) {
      socket.setSoTimeout(0); // HTTP/2 connection timeouts are set per-stream.
      http2Connection = new Http2Connection.Builder(true)
```

```
            .socket(socket, route.address().url().host(), source, sink)
            .listener(this)
            .pingIntervalMillis(pingIntervalMillis)
            .build();
        http2Connection.start();
      }
    }
```

可以看出OkHttp如果获得一个连接，新连接如果将本地socket和后台socke连接起来。在上面获得新连接时，提到一个连接池，接下来继续分析连接池。以及如何在连接池中将连接放入和取出。

5.4.3connectionPool

首先ConnectionPool的实例化过程，一个OkHttpClient只包含一个ConnectionPool，其实例化过程也在OkHttpClient的实例化过程中实现，值得一提的是ConnectionPool各个方法的调用并没有直接对外暴露，而是通过OkHttpClient的Internal接口统一对外暴露：

```
static {
    Internal.instance = new Internal() {
      @Override public void addLenient(Headers.Builder builder, String line) {
        builder.addLenient(line);
      }

      @Override public void addLenient(Headers.Builder builder, String name,
String value) {
        builder.addLenient(name, value);
      }

      @Override public void setCache(OkHttpClient.Builder builder, InternalCache
internalCache) {
        builder.setInternalCache(internalCache);
      }

      @Override public boolean connectionBecameIdle(
          ConnectionPool pool, RealConnection connection) {
        return pool.connectionBecameIdle(connection);
      }

      @Override public RealConnection get(ConnectionPool pool, Address address,
          StreamAllocation streamAllocation, Route route) {
        return pool.get(address, streamAllocation, route);
      }

      @Override public boolean equalsNonHost(Address a, Address b) {
        return a.equalsNonHost(b);
      }

      @Override public Socket deduplicate(
          ConnectionPool pool, Address address, StreamAllocation
streamAllocation) {
        return pool.deduplicate(address, streamAllocation);
      }

      @Override public void put(ConnectionPool pool, RealConnection connection)
{
        pool.put(connection);
      }
```

```
      @Override public RouteDatabase routeDatabase(ConnectionPool
connectionPool) {
        return connectionPool.routeDatabase;
      }

      @Override public int code(Response.Builder responseBuilder) {
        return responseBuilder.code;
      }

      @Override
      public void apply(ConnectionSpec tlsConfiguration, SSLSocket sslSocket,
boolean isFallback) {
        tlsConfiguration.apply(sslSocket, isFallback);
      }

      @Override public HttpUrl getHttpUrlChecked(String url)
          throws MalformedURLException, UnknownHostException {
        return HttpUrl.getChecked(url);
      }

      @Override public StreamAllocation streamAllocation(Call call) {
        return ((RealCall) call).streamAllocation();
      }

      @Override public Call newWebSocketCall(OkHttpClient client, Request
originalRequest) {
        return RealCall.newRealCall(client, originalRequest, true);
      }
    };
  }
```

连接池的维护：

ConnectionPool内部通过一个双端队列(dequeue)来维护当前所有连接，主要涉及到的操作包括：

put：放入新连接
get：从连接池中获取连接
evictAll：关闭所有连接
connectionBecameIdle：连接变空闲后调用清理线程
deduplicate：清除重复的多路复用线程

首先来看Internal.instance的put方法调用了连接池的put方法，下面是ConnectionPool的put方法：

```
  void put(RealConnection connection) {
    assert (Thread.holdsLock(this));
       //如果清理线程没有开启，则开启
    if (!cleanupRunning) {
      cleanupRunning = true;
      //cleanupRunnable见下
      executor.execute(cleanupRunnable);
    }
    connections.add(connection);
  }
```

当第一个连接被添加就线程池时，开启清除线程，主要清除那些连接池中过期的连接，然后将连接添加就connections对象中。下面看一下cleanupRunnable和connections的定义，其中connections是一个阻塞队列cleanupRunnable

connections

```
private final Deque<RealConnection> connections = new ArrayDeque<>();
```

```
 private final Runnable cleanupRunnable = new Runnable() {
    @Override public void run() {
      while (true) {
        //得到下一次清除的等待时长
        long waitNanos = cleanup(System.nanoTime());
        //没有连接，清除任务终结
        if (waitNanos == -1) return;
        if (waitNanos > 0) {
        //等待时间
          long waitMillis = waitNanos / 1000000L;
          waitNanos -= (waitMillis * 1000000L);
          synchronized (ConnectionPool.this) {
            try {
              ConnectionPool.this.wait(waitMillis, (int) waitNanos);
            } catch (InterruptedException ignored) {
            }
          }
        }
      }
    }
  };
```

可以看到cleadupRunnbale是一个死循环，调用cleanup方法进行清理工作并返回一个等待时长，如果有等待时长，那么让连接池进行休眠。其中清理工作在cleanup方法中，代码如下：

```
long cleanup(long now) {
    int inUseConnectionCount = 0;
    int idleConnectionCount = 0;
    RealConnection longestIdleConnection = null;
    long longestIdleDurationNs = Long.MIN_VALUE;

    // Find either a connection to evict, or the time that the next eviction is
due.
    synchronized (this) {
     //检查每个连接
      for (Iterator<RealConnection> i = connections.iterator(); i.hasNext(); ) {
        RealConnection connection = i.next();

        // If the connection is in use, keep searching.
        //如果连接正在使用，则跳过
        if (pruneAndGetAllocationCount(connection, now) > 0) {
          inUseConnectionCount++;
          continue;
        }

        idleConnectionCount++;

        // If the connection is ready to be evicted, we're done.
```

```
        //找出空闲时间最长的连接
        long idleDurationNs = now - connection.idleAtNanos;
        if (idleDurationNs > longestIdleDurationNs) {
          longestIdleDurationNs = idleDurationNs;
          longestIdleConnection = connection;
        }
      }

 //如果时间超出规定的空闲时间或者数量达到最大空闲树，那么移除。关闭操作在后面
      if (longestIdleDurationNs >= this.keepAliveDurationNs
          || idleConnectionCount > this.maxIdleConnections) {
        // We've found a connection to evict. Remove it from the list, then
close it below (outside
        // of the synchronized block).
        connections.remove(longestIdleConnection);
      } else if (idleConnectionCount > 0) {
       //如果时间和数量都没有到达上限，那么得到存活时间
        // A connection will be ready to evict soon.
        return keepAliveDurationNs - longestIdleDurationNs;
      } else if (inUseConnectionCount > 0) {
      //如果所有连接都在使用中，返回最大存活时间
        // All connections are in use. It'll be at least the keep alive duration
'til we run again.
        return keepAliveDurationNs;
      } else {
         //没有连接，关闭清除线程
        // No connections, idle or in use.
        cleanupRunning = false;
        return -1;
      }
    }
```

从代码中可以看出，对当前连接池中保存的所有连接进行遍历，然后调用pruneAndGetAllocationCount()方法获取连接上可用的StreamAllocation的数量以及删除不可用的StreamAllocation，如果数量大于0，则表示该连接还在使用，那么继续下一次遍历；否则空闲连接数+1,需要查找出所有不可用的连接中最大的空闲时间。遍历做完后，根据不同情况不同的值返回不同的结果，一旦找到了最大的空闲连接，那么在同步块外部调用closeQuietly关闭连接。

pruneAndGetAllocationCount()方法用于删除连接上不可用的StreamAllocation以及可用的StreamAllocation的数量，下面是其具体实现：

```
 private int pruneAndGetAllocationCount(RealConnection connection, long now) {
      //得到关联在连接上StramAllocation对象列表
    List<Reference<StreamAllocation>> references = connection.allocations;
    for (int i = 0; i < references.size(); ) {
      Reference<StreamAllocation> reference = references.get(i);

       //可用
      if (reference.get() != null) {
        i++;
        continue;
      }

      // We've discovered a leaked allocation. This is an application bug.
      StreamAllocation.StreamAllocationReference streamAllocRef =
          (StreamAllocation.StreamAllocationReference) reference;
      String message = "A connection to " + connection.route().address().url()
```

```
          + " was leaked. Did you forget to close a response body?";
      Platform.get().logCloseableLeak(message, streamAllocRef.callStackTrace);

      references.remove(i);
      connection.noNewStreams = true;

      // If this was the last allocation, the connection is eligible for
immediate eviction.
      if (references.isEmpty()) {
        connection.idleAtNanos = now - keepAliveDurationNs;
        return 0;
      }
    }

    return references.size();
  }
```

到目前为止，一个连接是如何被添加到线程池中的以及线程池的自动清除线程是如何工作的。

下面讲一个连接如何从连接池中获得的

get方法

```
// ConnectionPool
  @Nullable RealConnection get(Address address, StreamAllocation
streamAllocation, Route route) {
    assert (Thread.holdsLock(this));
    for (RealConnection connection : connections) {  //遍历队列

      if (connection.isEligible(address, route)) { //如果满足条件
        streamAllocation.acquire(connection, true); //streamAllocation赋给连接
        return connection;
      }
    }
    return null;
  }
```

需要满足的条件

```
public boolean isEligible(Address address, @Nullable Route route) {
    // If this connection is not accepting new streams, we're done.
    //如果这个连接不接受新流
    if (allocations.size() >= allocationLimit || noNewStreams) return false;

    // If the non-host fields of the address don't overlap, we're done.
    //如果主机的地址不同
    if (!Internal.instance.equalsNonHost(this.route.address(), address)) return
false;

    // If the host exactly matches, we're done: this connection can carry the
address.
    //如果.url().host()相同
    if (address.url().host().equals(this.route().address().url().host())) {
      return true; // This connection is a perfect match.
    }
```

```
    // At this point we don't have a hostname match. But we still be able to
carry the request if
    // our connection coalescing requirements are met. See also:
    // https://hpbn.co/optimizing-application-delivery/#eliminate-domain-
sharding
    // https://daniel.haxx.se/blog/2016/08/18/http2-connection-coalescing/

    // 1. This connection must be HTTP/2.
    //如果是HTTP/2.协议
    if (http2Connection == null) return false;

    // 2. The routes must share an IP address. This requires us to have a DNS
address for both
    // hosts, which only happens after route planning. We can't coalesce
connections that use a
    // proxy, since proxies don't tell us the origin server's IP address.
  // 这些路由必须共享一个IP地址。这要求我们为两个主机都有一个DNS地址，
  //这只在路由计划之后才会发生。由于代理没有告诉我们源服务器的IP地址，所以我们不能合并使用代理的
连接。
    if (route == null) return false;
    if (route.proxy().type() != Proxy.Type.DIRECT) return false;
    if (this.route.proxy().type() != Proxy.Type.DIRECT) return false;
    if (!this.route.socketAddress().equals(route.socketAddress())) return false;

    // 3. This connection's server certificate's must cover the new host.
    //这个连接的服务器证书必须覆盖新主机
    if (route.address().hostnameVerifier() != OkHostnameVerifier.INSTANCE)
return false;
    if (!supportsUrl(address.url())) return false;

    // 4. Certificate pinning must match the host.
    //证书必须与主机匹配。
    try {
      address.certificatePinner().check(address.url().host(),
handshake().peerCertificates());
    } catch (SSLPeerUnverifiedException e) {
      return false;
    }

    return true; // The caller's address can be carried by this connection.
  }
```

继续来看 streamAllocation.acquire方法

```
 public void acquire(RealConnection connection, boolean reportedAcquired) {
    assert (Thread.holdsLock(connectionPool));
    if (this.connection != null) throw new IllegalStateException();

    this.connection = connection; //将connection赋值给StreamAllocation的connection
    this.reportedAcquired = reportedAcquired;
    connection.allocations.add(new StreamAllocationReference(this,
callStackTrace));
  }
```

到目前为止，我们分析了一个连接如何放入连接池，如何从连接池中取出。同时也分析了连接池如何清除过期的连接。

5.5发送请求和获取响应

在连接拦截器中，配置好这次发送请求的连接后，在网络拦截器中，进行具体的连接操作.

5.5.1发送请求

CallServerInterceptor的intercept方法

```
 //发送HTTP首部信息
    httpStream.writeRequestHeaders(request);

//发送请求体
 BufferedSink bufferedRequestBody = Okio.buffer(requestBodyOut);
equest.body().writeTo(bufferedRequestBody);
```

写请求和读响应都是通过HttpStream对象，在前面的分析中知道了HttpStream的具体实现是Http1xStream或Http2xStream。我们主要看Http1xStream的各个实现，首先看写头部信息的writeRequestHeaders方法，下面是Http1xStream的具体实现：

```
//Http1Codec
@Override public void writeRequestHeaders(Request request) throws IOException {
    String requestLine = RequestLine.get(
        request, streamAllocation.connection().route().proxy().type());//首先获取
HTTP请求行
    writeRequest(request.headers(), requestLine);//具体的写操作
  }
```

查看writeRequest方法

```
  /** Returns bytes of a request header for sending on an HTTP transport. */
  public void writeRequest(Headers headers, String requestLine) throws
IOException {

    if (state != STATE_IDLE) throw new IllegalStateException("state: " + state);
//判断状态
    sink.writeUtf8(requestLine).writeUtf8("\r\n"); //写入请求行和空行
    for (int i = 0, size = headers.size(); i < size; i++) {//对头部信息做遍历,逐个写
入
      sink.writeUtf8(headers.name(i))
          .writeUtf8(": ")
          .writeUtf8(headers.value(i))
          .writeUtf8("\r\n");
    }
    sink.writeUtf8("\r\n");
    state = STATE_OPEN_REQUEST_BODY; //状态置为STATE_OPEN_REQUEST_BODY
  }
```

发送之后会刷新sink

```
//拦截器中
httpCodec.flushRequest();
```

```
  @Override public void flushRequest() throws IOException {
    sink.flush();
  }
```

上面是请求体的发送，下面来看请求头的发送。

```
public abstract void writeTo(BufferedSink sink) throws IOException;
```

可以看到 writeTo是一个抽象方法，看他的实现

首先是FormBody。

```
 @Override public void writeTo(BufferedSink sink) throws IOException {
    writeOrCountBytes(sink, false);
  }
```

继续来看writeOrCountBytes

```
private long writeOrCountBytes(@Nullable BufferedSink sink, boolean countBytes)
{
    long byteCount = 0L;

    Buffer buffer;
    if (countBytes) {
      buffer = new Buffer();
    } else {
      buffer = sink.buffer();
    }

    for (int i = 0, size = encodedNames.size(); i < size; i++) {  //get方法拼凑后面
的参数
      if (i > 0) buffer.writeByte('&');
      buffer.writeUtf8(encodedNames.get(i));
      buffer.writeByte('=');
      buffer.writeUtf8(encodedValues.get(i));
    }

    if (countBytes) {
      byteCount = buffer.size();
      buffer.clear();
    }

    return byteCount;
  }
```

继续看其他的实现:MultipartBody

```
@Override public void writeTo(BufferedSink sink) throws IOException {
    writeOrCountBytes(sink, false);
  }
```

继续看writeOrCountBytes方法

```
 private long writeOrCountBytes(
```

```
    @Nullable BufferedSink sink, boolean countBytes) throws IOException {
  long byteCount = 0L;

  Buffer byteCountBuffer = null;
  if (countBytes) {
    sink = byteCountBuffer = new Buffer();
  }

  for (int p = 0, partCount = parts.size(); p < partCount; p++) {
    Part part = parts.get(p);
    Headers headers = part.headers;
    RequestBody body = part.body;

    sink.write(DASHDASH);
    sink.write(boundary);
    sink.write(CRLF);

    if (headers != null) {
      for (int h = 0, headerCount = headers.size(); h < headerCount; h++) {
        sink.writeUtf8(headers.name(h))
            .write(COLONSPACE)
            .writeUtf8(headers.value(h))
            .write(CRLF);
      }
    }

    //请求体类型
    MediaType contentType = body.contentType();
    if (contentType != null) {
      sink.writeUtf8("Content-Type: ")
          .writeUtf8(contentType.toString())
          .write(CRLF);
    }

    //请求体长度
    long contentLength = body.contentLength();
    if (contentLength != -1) {
      sink.writeUtf8("Content-Length: ")
          .writeDecimalLong(contentLength)
          .write(CRLF);
    } else if (countBytes) {
      // We can't measure the body's size without the sizes of its components.
      byteCountBuffer.clear();
      return -1L;
    }

    sink.write(CRLF);

    if (countBytes) {
      byteCount += contentLength;
    } else {
      body.writeTo(sink); //发送请求体，body类型： RequestBody
    }

    sink.write(CRLF);
  }

  sink.write(DASHDASH);
```

```
    sink.write(boundary);
    sink.write(DASHDASH);
    sink.write(CRLF);

    if (countBytes) {
      byteCount += byteCountBuffer.size();
      byteCountBuffer.clear();
    }

    return byteCount;
  }
```

来看最后的实现： RequestBody

```
  @Override
    public void writeTo(BufferedSink sink) throws IOException {
        if (progressListener == null) {
            mRequestBody.writeTo(sink);
            return;
        }
        ProgressOutputStream progressOutputStream = new
ProgressOutputStream(sink.outputStream(), progressListener, contentLength());
        BufferedSink progressSink =
Okio.buffer(Okio.sink(progressOutputStream));//发送
        mRequestBody.writeTo(progressSink); //递归调用
        progressSink.flush();
    }
```

上面分析了请求头和几种不同的请求体的发送，下面我们来看响应的接收

5.6接收响应

首先来看接收响应头

```
responseBuilder = httpCodec.readResponseHeaders(true);
```

```
// http1Codec
@Override public Response.Builder readResponseHeaders(boolean expectContinue)
throws IOException {
    if (state != STATE_OPEN_REQUEST_BODY && state !=
STATE_READ_RESPONSE_HEADERS) { //判断状态
      throw new IllegalStateException("state: " + state);
    }

    try {
      StatusLine statusLine = StatusLine.parse(readHeaderLine());

      Response.Builder responseBuilder = new Response.Builder()
          .protocol(statusLine.protocol)
          .code(statusLine.code)
          .message(statusLine.message)
          .headers(readHeaders());//调用readHeaders()进行请求

      if (expectContinue && statusLine.code == HTTP_CONTINUE) { //返回码为100并且存
在00-continue请求头
```

```
        return null;
      } else if (statusLine.code == HTTP_CONTINUE) { //返回码为100
        state = STATE_READ_RESPONSE_HEADERS;
        return responseBuilder;
      }

      state = STATE_OPEN_RESPONSE_BODY;
      return responseBuilder;
    } catch (EOFException e) {
      // Provide more context if the server ends the stream before sending a
response.
      IOException exception = new IOException("unexpected end of stream on " +
streamAllocation);
      exception.initCause(e);
      throw exception;
    }
  }
```

继续来看readHeaders()方法

```
 public Headers readHeaders() throws IOException {
    Headers.Builder headers = new Headers.Builder();
    // parse the result headers until the first blank line
    //可以看到每行遍历直到第一个空行，然后调用Internal.instance的
    //addLenient方法将这一行的信息解析并添加到头部中
    for (String line; (line = readHeaderLine()).length() != 0; ) {
      Internal.instance.addLenient(headers, line);
    }
    return headers.build();
  }
```

继续来看Internal.instance.addLenient

```
//OkHttpClient.instance.addLenient
  Internal.instance = new Internal() {
      @Override public void addLenient(Headers.Builder builder, String line) {
        builder.addLenient(line);
      }
```

```
  Builder addLenient(String line) {
      int index = line.indexOf(":", 1);//获取:
      if (index != -1) {
        return addLenient(line.substring(0, index), line.substring(index + 1));
//添加到列表
      } else if (line.startsWith(":")) {
        // Work around empty header names and header names that start with a
        // colon (created by old broken SPDY versions of the response cache).
        return addLenient("", line.substring(1)); // Empty header name.
      } else {
        return addLenient("", line); // No header name.
      }
```

从上面的代码可以看到，首先获取“：”的位置，如果存在“：”，那么调用addLenient将名和值添加进列表中，如果以“:”开宇，则头信息的名称为空，有值；如果都没有，那么没有头部信息名。三种情况都是调用addLenient方法，如下：

```
  Builder addLenient(String name, String value) {
      namesAndValues.add(name); //字符传列表
      namesAndValues.add(value.trim());
      return this;
    }
```

到上面为此，读取响应的头部信息已经完成，接下来在CallServerInterceptor中做的是调用openResponseBody方法读取响应的主体部分

```
httpCodec.openResponseBody(response)
```

来看其具体实现

```
@Override public ResponseBody openResponseBody(Response response) throws
IOException {
    streamAllocation.eventListener.responseBodyStart(streamAllocation.call);
    String contentType = response.header("Content-Type");

    if (!HttpHeaders.hasBody(response)) {
      Source source = newFixedLengthSource(0);
      return new RealResponseBody(contentType, 0, Okio.buffer(source)); // 如果响
应主体部分不应有内容，那么返回newFixedLengthSource(0)
    }

    if ("chunked".equalsIgnoreCase(response.header("Transfer-Encoding"))) {
      Source source = newChunkedSource(response.request().url());
      return new RealResponseBody(contentType, -1L, Okio.buffer(source)); // 如果
响应头部中Transfer-Encoding为chunked，即分块了，那么返回newChunkedSource
    }

    long contentLength = HttpHeaders.contentLength(response);
    if (contentLength != -1) {
      Source source = newFixedLengthSource(contentLength);
      return new RealResponseBody(contentType, contentLength,
Okio.buffer(source));// 如果响应中有个具体长度，那么返回newFixedLengthSource，并且指定长
度
    }

    return new RealResponseBody(contentType, -1L,
Okio.buffer(newUnknownLengthSource()));//以上情况均不满足，返回
newUnknownLengthSource
  }
```

5.7总结

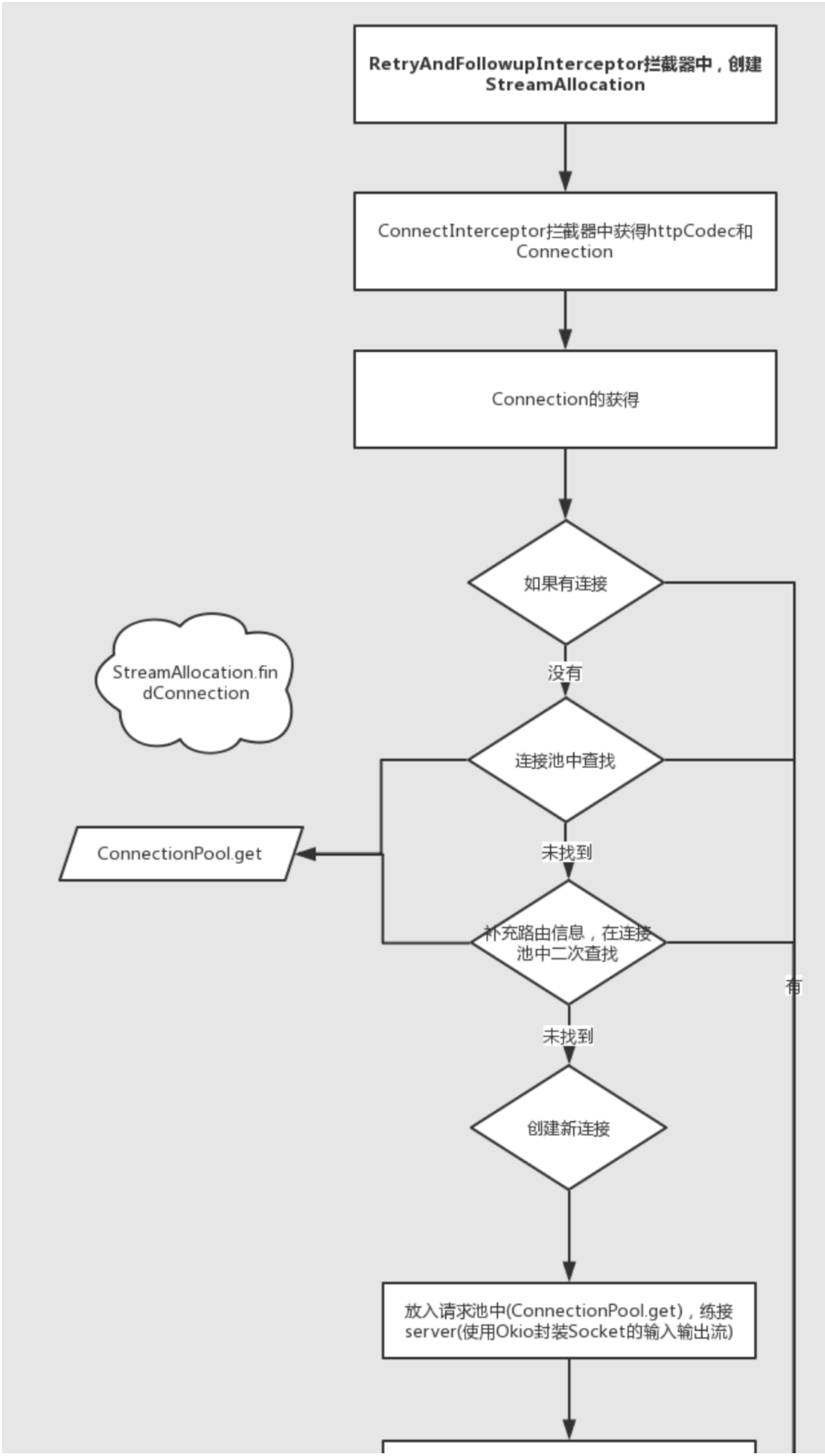
至此，分析完了OKHttp的网络请求部分，总结一下，在重试拦截器中获得Address和StreamAllocation(负责根据请求创建连接)，在连接拦截器中获得连接，最后在网络进行发送请求头，请求体，获得响应头，获得响应体。

值得注意的是连接的获得 即:

如果有连接，直接用
没有可用的连接，第一次去连接池中查找，找到后直接用
没有找到，补充路由信息，在连接池中二次查找。
依然没有找到，创建新连接，然后连接server，将其放入到连接池中

其次是连接放入连接池，和从连接池中取出的过程。还有新建一个连接后，将其从跟后台socket进行连接(封装成Okio)。

最后是通过OKio发送请求和获得响应。


RetryAndFollowupInterceptor拦截器中，创建StreamAllocation
ConnectInterceptor拦截器中获得httpCodec和Connection
Connection的获得
如果有连接
没有
StreamAllocation.findConnection
连接池中查找
ConnectionPool.get
未找到
补充路由信息，在连接池中二次查找
有
未找到
创建新连接
放入请求池中(ConnectionPool.get)，练接server(使用Okio封装Socket的输入输出流)

6缓存相关

6.1Cache-Control

OkHttp根据HTTP头部中的CacheControl进行缓存控制。

6.1.1HTTP中的Cache-Control首部

HTTP头部中的Cache-Control首部可以指示对应请求该如何获取响应，比如应该直接使用缓存的响应还是应该从网络获取响应；可以指示响应该如何缓存，比如是否应该缓存下来还是设置一个过期时间等。Cache-Control首部的一些值既可以用于请求首部又可以用于响应首部。具体的值有no-cache、nostore、max-age、s-maxage、only-if-cached等。

6.1.2OkHttp中的CacheControl类

CacheControl类是对HTTP的Cache-Control首部的描述。CacheControl没有公共的构造方法，内部通过一个Builder进行设置值，获取值可以通过CacheControl对象进行获取。Builder中具体有如下设置方法：

```
CacheControl(Builder builder) {
    this.noCache = builder.noCache;
    this.noStore = builder.noStore;
    this.maxAgeSeconds = builder.maxAgeSeconds;
    this.sMaxAgeSeconds = -1;
    this.isPrivate = false;
    this.isPublic = false;
    this.mustRevalidate = false;
    this.maxStaleSeconds = builder.maxStaleSeconds;
    this.minFreshSeconds = builder.minFreshSeconds;
    this.onlyIfCached = builder.onlyIfCached;
    this.noTransform = builder.noTransform;
    this.immutable = builder.immutable;
  }
```

noCache()

对应于“no-cache”，如果出现在响应首部，不是表示不允许对响应进行缓存，而是表示客户端需要与服务器进行再验证，进行一个额外的GET请求得到最新的响应；如果出现在请求首部，表示不适用缓存响应，即进行网络请求得到响应

noStore()

对应于“no-store”，只能出现在响应首部，表明该响应不应该被缓存

maxAge(int maxAge, TimeUnit timeUnit)

对应于“max-age”，设置缓存响应的最大存活时间。如果缓存响应达到了最大存活时间，那么将不会再使用而会进行网络请求

maxStale(int maxStale,TimeUnit timeUnit)

对应于“max-stale”，缓存响应可以接受的最大过期时间，如果没有指定该参数，那么过期缓存响应将不会使用。

minFresh(int minFresh,TimeUnit timeUnit)

对应于“min-fresh”，设置一个响应将会持续刷新的最小秒数。如果一个响应当minFresh过去后过期了，那么缓存响应不会再使用了，会进行网络请求。

onlyIfCached()

对应于“onlyIfCached”，用于请求首部，表明该请求只接受缓存中的响应。如果缓存中没有响应，那么返回一个状态码为504的响应。

6.2Cache类

Cache中很多方法都是通过DiskLruCache实现的

OKHttp在DiskLruCache的基础上进行了修改，将IO操作改成了OKio

在OkHttp中Cache负责将响应缓存到文件中，以便可以重用和减少带宽。
在Cache类内部又一个InternalCache的实现了类

```
 //根据请求得到响应
 final InternalCache internalCache = new InternalCache() {
    @Override public Response get(Request request) throws IOException {
      return Cache.this.get(request);
    }
 //缓存响应
    @Override public CacheRequest put(Response response) throws IOException {
      return Cache.this.put(response);
    }

//移出响应
    @Override public void remove(Request request) throws IOException {
      Cache.this.remove(request);
    }

//更新响应
    @Override public void update(Response cached, Response network) {
      Cache.this.update(cached, network);
    }

    @Override public void trackConditionalCacheHit() {
      Cache.this.trackConditionalCacheHit();
    }

    @Override public void trackResponse(CacheStrategy cacheStrategy) {
      Cache.this.trackResponse(cacheStrategy);
    }
  };
```

在代码中可以看出来，ternalCache接口中的每个方法的实现都交给了外部类Cache，所以主要看Cache类中的各个方法，而Cache类的这些方法又主要交给了DiskLruCache来实现。

6.2.1缓存响应

首先来看缓存响应的Put方法

```
@Nullable CacheRequest put(Response response) {
    //得到请求的方法
    String requestMethod = response.request().method();

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

//不缓存非GET方法
    if (!requestMethod.equals("GET")) {
      // Don't cache non-GET responses. We're technically allowed to cache
      // HEAD requests and some POST requests, but the complexity of doing
      // so is high and the benefit is low.
      return null;
    }

    //如果请求头中如果含有星号，也不进行缓存
    if (HttpHeaders.hasVaryAll(response)) {
      return null;
    }

   //使用DiskLruCache进行缓冲
    Entry entry = new Entry(response);
    DiskLruCache.Editor editor = null;
    try {
      editor = cache.edit(key(response.request().url()));
      if (editor == null) {
        return null;
      }
      entry.writeTo(editor);
      return new CacheRequestImpl(editor);
    } catch (IOException e) {
      abortQuietly(editor);
      return null;
    }
  }
```

上面的代码对请求进行判断，如果满足条件，则使用响应创建一个Entry，然后使用DiskLruCache写入缓存，最终返回一个CacheRequestImpl对象。cache是DiskLruCache的实例，调用edit方法传入响应的key值。下面是Key方法的实现：

```
public static String key(HttpUrl url) {
    return ByteString.encodeUtf8(url.toString()).md5().hex(); //对其请求的Url做MD5，然后获得其值。
  }
```

然后查看edit方法

```
 /**
   * Returns an editor for the entry named {@code key}, or null if another edit is in progress.
   */
  public @Nullable Editor edit(String key) throws IOException {
    return edit(key, ANY_SEQUENCE_NUMBER);
  }

  synchronized Editor edit(String key, long expectedSequenceNumber) throws IOException {
    initialize(); //初始化

    checkNotClosed();
    validateKey(key);
    Entry entry = lruEntries.get(key);//通过key获得entry
    if (expectedSequenceNumber != ANY_SEQUENCE_NUMBER && (entry == null
        || entry.sequenceNumber != expectedSequenceNumber)) {
      return null; // Snapshot is stale.
    }
    if (entry != null && entry.currentEditor != null) {
      return null; // Another edit is in progress. // 当前cache entry正在被其他对象操作
    }
    if (mostRecentTrimFailed || mostRecentRebuildFailed) {
      // The OS has become our enemy! If the trim job failed, it means we are storing more data than
      // requested by the user. Do not allow edits so we do not go over that limit any further. If
      // the journal rebuild failed, the journal writer will not be active, meaning we will not be
      // able to record the edit, causing file leaks. In both cases, we want to retry the clean up
      // so we can get out of this state!
      executor.execute(cleanupRunnable);
      return null;
    }

  // 日志接入DIRTY记录
    // Flush the journal before creating files to prevent file leaks.
    journalWriter.writeUtf8(DIRTY).writeByte(' ').writeUtf8(key).writeByte('\n');
    journalWriter.flush();

    if (hasJournalErrors) {
      return null; // Don't edit; the journal can't be written.
    }

    if (entry == null) {
      entry = new Entry(key);
```

```
      lruEntries.put(key, entry);
    }
    Editor editor = new Editor(entry);
    entry.currentEditor = editor;
    return editor;
  }
```

在这里获得Editor后，然后调用editor.writeTo(editor),将editor写入

```
public void writeTo(DiskLruCache.Editor editor) throws IOException {
      BufferedSink sink = Okio.buffer(editor.newSink(ENTRY_METADATA));

      //缓存请求有关信息
      sink.writeUtf8(url)
          .writeByte('\n');
      sink.writeUtf8(requestMethod)
          .writeByte('\n');
      sink.writeDecimalLong(varyHeaders.size())
          .writeByte('\n');
      for (int i = 0, size = varyHeaders.size(); i < size; i++) {
        sink.writeUtf8(varyHeaders.name(i))
            .writeUtf8(": ")
            .writeUtf8(varyHeaders.value(i))
            .writeByte('\n');
      }

       //缓存Http响应行
      sink.writeUtf8(new StatusLine(protocol, code, message).toString())
          .writeByte('\n');
       //缓存响应首部
      sink.writeDecimalLong(responseHeaders.size() + 2)
          .writeByte('\n');
      for (int i = 0, size = responseHeaders.size(); i < size; i++) {
        sink.writeUtf8(responseHeaders.name(i))
            .writeUtf8(": ")
            .writeUtf8(responseHeaders.value(i))
            .writeByte('\n');
      }
      sink.writeUtf8(SENT_MILLIS)
          .writeUtf8(": ")
          .writeDecimalLong(sentRequestMillis)
          .writeByte('\n');
      sink.writeUtf8(RECEIVED_MILLIS)
          .writeUtf8(": ")
          .writeDecimalLong(receivedResponseMillis)
          .writeByte('\n');

   //是Https请求，缓存握手，证书信息
      if (isHttps()) {
        sink.writeByte('\n');
        sink.writeUtf8(handshake.cipherSuite().javaName())
            .writeByte('\n');
        writeCertList(sink, handshake.peerCertificates());
        writeCertList(sink, handshake.localCertificates());
        sink.writeUtf8(handshake.tlsVersion().javaName()).writeByte('\n');
      }
      sink.close();
```

```
    }
```

上面的代码里面有，将响应头的头部信息还有请求头的部分信息(URL 请求方法 请求头部)进行缓存。同时对于一个请求和响应而言，缓存中的key值是请求的URL的MD5值，而value包括请求和响应部分。Entry的writeTo()方法只把请求的头部和响应的头部保存了，最关键的响应主体部分在哪里保存呢？它就在put方法的返回体CacheRequestImpl，下面是这个类的实现：

```
private final class CacheRequestImpl implements CacheRequest {
    private final DiskLruCache.Editor editor;
    private Sink cacheOut;
    private Sink body;
    boolean done;

    CacheRequestImpl(final DiskLruCache.Editor editor) {
      this.editor = editor;
      this.cacheOut = editor.newSink(ENTRY_BODY);
      this.body = new ForwardingSink(cacheOut) {
        @Override public void close() throws IOException {
          synchronized (Cache.this) {
            if (done) {
              return;
            }
            done = true;
            writeSuccessCount++;
          }
          super.close();
          editor.commit();
        }
      };
    }

    @Override public void abort() {
      synchronized (Cache.this) {
        if (done) {
          return;
        }
        done = true;
        writeAbortCount++;
      }
      Util.closeQuietly(cacheOut);
      try {
        editor.abort();
      } catch (IOException ignored) {
      }
    }

    @Override public Sink body() {
      return body;
    }
  }
```

close,abort方法会调用editor.abort和editor.commit来更新日志，editor.commit还会将dirtyFile重置为cleanFile作为稳定可用的缓存，先看adort方法

```
public void abort() throws IOException {
      synchronized (DiskLruCache.this) {
        if (done) {
          throw new IllegalStateException();
        }
        if (entry.currentEditor == this) {
          completeEdit(this, false);
        }
        done = true;
      }
    }
```

继续来看 completeEdit()方法

```
  synchronized void completeEdit(Editor editor, boolean success) throws
IOException {
    Entry entry = editor.entry;
    if (entry.currentEditor != editor) {
      throw new IllegalStateException();
    }

    // If this edit is creating the entry for the first time, every index must
have a value.
    //如果这辑第一次创建条目，那么每个索引都必须有一个值。
    if (success && !entry.readable) {
      for (int i = 0; i < valueCount; i++) {
        if (!editor.written[i]) {
          editor.abort();
          throw new IllegalStateException("Newly created entry didn't create
value for index " + i);
        }
        if (!fileSystem.exists(entry.dirtyFiles[i])) {
          editor.abort();
          return;
        }
      }
    }

    for (int i = 0; i < valueCount; i++) {
      File dirty = entry.dirtyFiles[i];
      if (success) {
        if (fileSystem.exists(dirty)) {
          File clean = entry.cleanFiles[i];
          fileSystem.rename(dirty, clean);
          long oldLength = entry.lengths[i];
          long newLength = fileSystem.size(clean);
          entry.lengths[i] = newLength;
          size = size - oldLength + newLength;
        }
      } else {
        fileSystem.delete(dirty);//若失败则删除dirtyfile
      }
    }

    redundantOpCount++;
    entry.currentEditor = null;
```

```
    //更新日志
    if (entry.readable | success) {
      entry.readable = true;
      journalWriter.writeUtf8(CLEAN).writeByte(' ');
      journalWriter.writeUtf8(entry.key);
      entry.writeLengths(journalWriter);
      journalWriter.writeByte('\n');
      if (success) {
        entry.sequenceNumber = nextSequenceNumber++;
      }
    } else {
      lruEntries.remove(entry.key);
      journalWriter.writeUtf8(REMOVE).writeByte(' ');
      journalWriter.writeUtf8(entry.key);
      journalWriter.writeByte('\n');
    }
    journalWriter.flush();

    if (size > maxSize || journalRebuildRequired()) {
      executor.execute(cleanupRunnable);
    }
  }
```

6.2.2获取缓存

获取缓存在get方法里

```
  @Nullable Response get(Request request) {
    //获得key值
    String key = key(request.url());
     //从DiskLruCache中得到缓存
    DiskLruCache.Snapshot snapshot;
    Entry entry;
    try {
      snapshot = cache.get(key);
      if (snapshot == null) {  //如果没有找到
        return null;
      }
    } catch (IOException e) {
      // Give up because the cache cannot be read.
      return null;
    }

    try {
      entry = new Entry(snapshot.getSource(ENTRY_METADATA)); //创建entry对象
    } catch (IOException e) {
      Util.closeQuietly(snapshot);
      return null;
    }

    Response response = entry.response(snapshot); //获得响应对象

    if (!entry.matches(request, response)) { //如果请求和响应不匹配
      Util.closeQuietly(response.body());
      return null;
```

```
    }

    return response;
  }
```

6.2.3Entry

首先来看其构造方法

```
 Entry(Source in) throws IOException {
      try {
        BufferedSource source = Okio.buffer(in);
        //读取请求相关的信息
        url = source.readUtf8LineStrict();
        requestMethod = source.readUtf8LineStrict();
        Headers.Builder varyHeadersBuilder = new Headers.Builder();
        int varyRequestHeaderLineCount = readInt(source);
        for (int i = 0; i < varyRequestHeaderLineCount; i++) {
          varyHeadersBuilder.addLenient(source.readUtf8LineStrict());
        }
        varyHeaders = varyHeadersBuilder.build();

          //读响应状态行
        StatusLine statusLine = StatusLine.parse(source.readUtf8LineStrict());
        protocol = statusLine.protocol;
        code = statusLine.code;
        message = statusLine.message;

       //读响应行状态
        Headers.Builder responseHeadersBuilder = new Headers.Builder();
        int responseHeaderLineCount = readInt(source);
        for (int i = 0; i < responseHeaderLineCount; i++) {
          responseHeadersBuilder.addLenient(source.readUtf8LineStrict());
        }
        String sendRequestMillisString =
responseHeadersBuilder.get(SENT_MILLIS);
        String receivedResponseMillisString =
responseHeadersBuilder.get(RECEIVED_MILLIS);
        responseHeadersBuilder.removeAll(SENT_MILLIS);
        responseHeadersBuilder.removeAll(RECEIVED_MILLIS);
        sentRequestMillis = sendRequestMillisString != null
            ? Long.parseLong(sendRequestMillisString)
            : 0L;
        receivedResponseMillis = receivedResponseMillisString != null
            ? Long.parseLong(receivedResponseMillisString)
            : 0L;
        responseHeaders = responseHeadersBuilder.build();

        //是Https协议，读握手，证书信息
        if (isHttps()) {
          String blank = source.readUtf8LineStrict();
          if (blank.length() > 0) {
            throw new IOException("expected \"\" but was \"" + blank + "\"");
          }
          String cipherSuiteString = source.readUtf8LineStrict();
          CipherSuite cipherSuite = CipherSuite.forJavaName(cipherSuiteString);
          List<Certificate> peerCertificates = readCertificateList(source);
```

```
          List<Certificate> localCertificates = readCertificateList(source);
          TlsVersion tlsVersion = !source.exhausted()
              ? TlsVersion.forJavaName(source.readUtf8LineStrict())
              : TlsVersion.SSL_3_0;
          handshake = Handshake.get(tlsVersion, cipherSuite, peerCertificates,
localCertificates);
        } else {
          handshake = null;
        }
      } finally {
        in.close();
      }
    }
```

在put方法中我们知道了缓存中保存了请求的信息和响应的信息，这个构造方法主要用于从缓存中解析出各个字段。当获得这些信息后，就可以用过response() (get方法最后的调用)获得对应的响应

```
public Response response(DiskLruCache.Snapshot snapshot) {
      String contentType = responseHeaders.get("Content-Type");
      String contentLength = responseHeaders.get("Content-Length");
      Request cacheRequest = new Request.Builder()  //缓存的请求
          .url(url)
          .method(requestMethod, null)
          .headers(varyHeaders)
          .build();
      return new Response.Builder()//缓存的响应
          .request(cacheRequest)
          .protocol(protocol)
          .code(code)
          .message(message)
          .headers(responseHeaders)
          .body(new CacheResponseBody(snapshot, contentType, contentLength)) //获
得请求体
          .handshake(handshake)
          .sentRequestAtMillis(sentRequestMillis)
          .receivedResponseAtMillis(receivedResponseMillis)
          .build();
    }
```

查看 CacheResponseBody类的构造方法

```
  CacheResponseBody(final DiskLruCache.Snapshot snapshot,
        String contentType, String contentLength) {
      this.snapshot = snapshot;
      this.contentType = contentType;
      this.contentLength = contentLength;

      Source source = snapshot.getSource(ENTRY_BODY);
      bodySource = Okio.buffer(new ForwardingSource(source) {
        @Override public void close() throws IOException {
          snapshot.close();
          super.close();
        }
      });
    }
```

上面那个是get方法是的构造方法，Entry还有一种构造方法,即将响应里的内容保存起来。

```
    Entry(Response response) {
      this.url = response.request().url().toString();
      this.varyHeaders = HttpHeaders.varyHeaders(response);
      this.requestMethod = response.request().method();
      this.protocol = response.protocol();
      this.code = response.code();
      this.message = response.message();
      this.responseHeaders = response.headers();
      this.handshake = response.handshake();
      this.sentRequestMillis = response.sentRequestAtMillis();
      this.receivedResponseMillis = response.receivedResponseAtMillis();
    }
```

6.3缓存的使用

Cache的设置均在OkHttpClient的Builder中设置，有两个方法可以设置，分别是setInternalCache()和cache()方法，如下：

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

    public Builder cache(Cache cache) {
      this.cache = cache;
      this.internalCache = null;
      return this;
    }
```

从代码中可以看出，这两个方法会互相消除彼此。在之前讲到的InternalCache类，该类是一个接口，文档中说应用不应该实现该类，所以这儿，我也明白为什么OkHttpClient为什么还提供这样一个接口。

当设置好Cache后，我们再来看下Cache的构造方法：

```
public Cache(File directory, long maxSize) {
    this(directory, maxSize, FileSystem.SYSTEM);
  }

  Cache(File directory, long maxSize, FileSystem fileSystem) {
    this.cache = DiskLruCache.create(fileSystem, directory, VERSION,
ENTRY_COUNT, maxSize);
  }
```

可以看到暴露对外的构造方法只有两个参数，一个目录，一个最大尺寸，而其内部使用的DiskLruCache的create静态工厂方法。这里面FileSystem.SYSTEM是FileSystem接口的一个实现类，该类的各个方法使用Okio对文件I/O进行封装。

DiskLruCache的create()方法中传入的目录将会是缓存的父目录，其中ENTRY_COUNT表示每一个缓存实体中的值的个数，这儿是2。（第一个是请求头部和响应头部，第二个是响应主体部分）至此，Cache和其底层的DiskLruCache创建成功了。

6.4CacheInterceptor

在Okhttp缓存的具体执行时机是在缓存拦截器中

6.4.1intercept

```
@Override public Response intercept(Chain chain) throws IOException {
    //得到候选缓存响应，可能为空
    Response cacheCandidate = cache != null
        ? cache.get(chain.request())
        : null;

    long now = System.currentTimeMillis();

    //得到缓存策略
    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.
    }

    // 只要缓存响应，但是缓存响应不存在，返回504错误
    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(EMPTY_BODY)
          .sentRequestAtMillis(-1L)
          .receivedResponseAtMillis(System.currentTimeMillis())
          .build();
    }

    // 不使用网络，直接返回缓存响应
    if (networkRequest == null) {
      return cacheResponse.newBuilder()
          .cacheResponse(stripBody(cacheResponse))
          .build();
    }

    //进行网络操作获取响应
    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());
      }
    }

    // 如果也有缓存响应，则需要检查缓存响应是否需要进行更新
    if (cacheResponse != null) {
      //需要更新
      if (validate(cacheResponse, networkResponse)) {
        Response response = cacheResponse.newBuilder()
            .headers(combine(cacheResponse.headers(),
networkResponse.headers()))
            .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();

    //保存缓存
    if (HttpHeaders.hasBody(response)) {
      CacheRequest cacheRequest = maybeCache(response,
networkResponse.request(), cache);
      response = cacheWritingResponse(cacheRequest, response);
    }

    return response;
  }
```

6.4.2缓存策略

进入查看CacheStrategy中的Factory类(工厂类)

```
//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;
  }
```

6.5总结

OKHttp的缓存部分，一个是设置缓存这一方面由用户(程序员自己调用)，还有进行缓存的时机，在缓存拦截器中发生。在获取缓存时，主要是缓存的存和取，这两个是由DiskLruCache+Okio一同实现的，同时在缓存拦截器跟据请求来进行不同的缓存策略。