RealCall 实现接口Call，并持有Request，OkHttpClient对象，我的理解是RealCall一次实实在在的请求

首先，看下Call接口的定义的方法：

Request request();
返回request对象。

Response execute() throws IOException;
立即执行request，并阻塞直到response被处理或者error。

void enqueue(Callback responseCallback);
request稍后执行

void cancel();
取消request，如果已经完成，不能被设置为canceled

boolean isExecuted();//是否执行

boolean isCanceled();//是否取消

```
 interface Factory {
   Call newCall(Request request);
 }
```

工厂模式， 在OkHttpClient类中实现

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

其中，最重要的2个方法：
①execute ②enqueue 分别对应同步和异步

execute：

首先，我们来分析execute:「同步请求」

```
@Override public Response execute() throws IOException {
  synchronized (this) {
    if (executed) throw new IllegalStateException("Already Executed");//point one
    executed = true;
  }
  captureCallStackTrace();//point two
  eventListener.callStart(this);//point three
  try {
    client.dispatcher().executed(this);//point four
    Response result = getResponseWithInterceptorChain();//point five
    if (result == null) throw new IOException("Canceled");
    return result;
  } catch (IOException e) {
    eventListener.callFailed(this, e);
    throw e;
  } finally {
    client.dispatcher().finished(this);//point six
  }
 }
```

point one :如果已被执行，抛IllegalStateException异常，如未执行，标记为已执行

point two :追踪call的堆栈信息

point three:EventListener点击进去看，发现是一个记录事件的listener。包括：callStart,dnsStart,dnsEnd,connectStart....等。

point four: 我们发现调用了Dispatch类的runningSyncCalls.add(call);将这次call插入了同步call队列。

point five:

```
 Response getResponseWithInterceptorChain() throws IOException {
   // Build a full stack of interceptors.
   List<Interceptor> interceptors = new ArrayList<>();

//添加开发者应用层自定义的Interceptor
   interceptors.addAll(client.interceptors());
//这个Interceptor是处理请求失败的重试，重定向
   interceptors.add(retryAndFollowUpInterceptor);
//这个Interceptor工作是添加一些请求的头部或其他信息
//并对返回的Response做一些友好的处理（有一些信息你可能并不需要）
   interceptors.add(new BridgeInterceptor(client.cookieJar()));
//这个Interceptor的职责是判断缓存是否存在，读取缓存，更新缓存等等
   interceptors.add(new CacheInterceptor(client.internalCache()));
//这个Interceptor的职责是建立客户端和服务器的连接
   interceptors.add(new ConnectInterceptor(client));
   if (!forWebSocket) {
//添加开发者自定义的网络层拦截器
     interceptors.addAll(client.networkInterceptors());
   }
//最后添加CallserverInterceptor
   interceptors.add(new CallServerInterceptor(forWebSocket));
//一个包裹这request的chain
   Interceptor.Chain chain = new RealInterceptorChain(interceptors, null, null,
null, 0,
       originalRequest, this, eventListener, client.connectTimeoutMillis(),
       client.readTimeoutMillis(), client.writeTimeoutMillis());
 //把chain传递到第一个Interceptor手中
   return chain.proceed(originalRequest);
 }
```

interceptor的事，下一篇写。

点开chain.proceed(originalRequest)方法，我们来到RealInterceptorChain这个类并找到proceed方法：

```
 public Response proceed(Request request, StreamAllocation
streamAllocation, HttpCodec httpCodec,
      RealConnection connection) throws IOException {
    if (index >= interceptors.size()) throw new AssertionError();
    calls++;
 //先忽略
  // Call the next interceptor in the chain.
    RealInterceptorChain next = new RealInterceptorChain(interceptors,
streamAllocation, httpCodec,connection, index + 1, request, call,
eventListener, connectTimeout, readTimeout,
        writeTimeout);
    Interceptor interceptor = interceptors.get(index);
    Response response = interceptor.intercept(next);
//先忽略
    return response;
  }
```

在其他(非CallServerInterceptor和自定义interceptor)Interceptor中的intercept(Chain chain)方法中，retrun realChain.proceed(request, streamAllocation, httpCodec, connection)；index+1，利用递归，走完每一个interceptor。【责任链模式】

递归到哪里停止呢，我们往上看getResponseWithInterceptorChain方法的interceptors.add(new CallServerInterceptor(forWebSocket));这一句。

为什么最后添加这个CallServerInterceptor是有原因的，因为它最后return response。不再调用realChain.proceed

point six：

同样是调用了dispatch的 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) {
      if (!calls.remove(call))
throw new AssertionError("Call wasn't in-flight!");//⑴
      if (promoteCalls) promoteCalls();//⑵
      runningCallsCount = runningCallsCount();//⑶
      idleCallback = this.idleCallback;//⑷
    }

    if (runningCallsCount == 0 && idleCallback != null) {
      idleCallback.run();// ⑸
    }
  }
```

⑴先将call移除队列

⑵ finished(runningSyncCalls, call, false);

false， 不执行，异步会讲到

⑶ runningAsyncCalls.size() + runningSyncCalls.size();

⑷ 闲置线程赋值

⑸ 线程池为空时，执行回调

enqueue：

enqueue与execute相类似：「异步请求」

```
 @Override public void enqueue(Callback responseCallback) {
    synchronized (this) {
      if (executed) throw new IllegalStateException("Already Executed");//1
      executed = true;
    }
    captureCallStackTrace();//2
    eventListener.callStart(this);//3
    client.dispatcher().enqueue(new AsyncCall(responseCallback));//4
  }
```

1，2，3与同步请求相同。我们具体分析4,调用Dispatch类的enqueue方法

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

翻译过来就是当正在执行的异步队列个数小于maxRequest(64)并且请求同一个主机的个数小于maxRequestsPerHost(5)时，则将这个请求加入异步执行队列runningAsyncCall，并用线程池执行这个call，否则加入异步等待队列。

我们这里发现一个RealCall的内部类AsyncCall，AsyncCall继承于NamedRunnable

```
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 {
      execute();
    } finally {
      Thread.currentThread().setName(oldName);
    }
  }
  protected abstract void execute();
}
```

很简单，回到AsyncCall的execute方法：

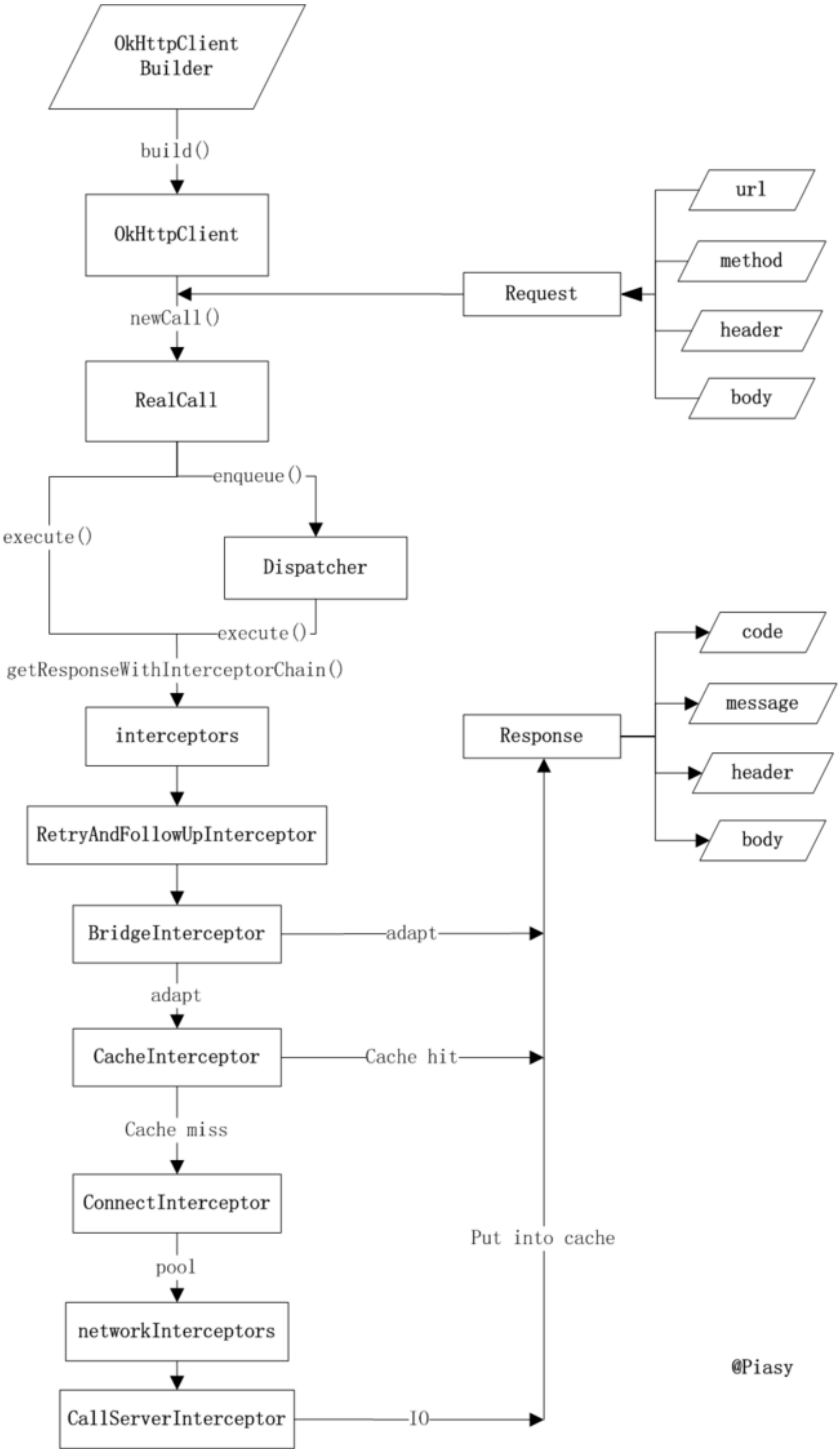
```
    @Override protected void execute() {
      boolean signalledCallback = false;
      try {
        Response response = getResponseWithInterceptorChain();
        if (retryAndFollowUpInterceptor.isCanceled()) {
          signalledCallback = true;
          responseCallback.onFailure(RealCall.this, new
IOException("Canceled"));
        } else {
          signalledCallback = true;
          responseCallback.onResponse(RealCall.this, response);
        }
      } catch (IOException e) {
        if (signalledCallback) {
          // Do not signal the callback twice!
          Platform.get().log(INFO, "Callback failure for " + toLoggableString(), e);
        } else {
          eventListener.callFailed(RealCall.this, e);
          responseCallback.onFailure(RealCall.this, e);
        }
      } finally {
        client.dispatcher().finished(this);
      }
    }
```

同步请求与异步请求大体相似，除了回调，retryAndFollowUpInterceptor的处理

和finish的处理。

finished中，传递的是AsyncCall对象。 finished(runningAsyncCalls, call, true);
会比同步多执行promoteCalls方法
promoteCalls()方法源码很简单，不贴了。
总结出来就是：
当前线程大于最大线程(64)return；
预备线程等于0的话 return；
开始遍历readyAsyncCalls:取出一个call，并把这个call放入runningAsyncCalls，然后执行execute。在遍历过程中如果runningAsyncCalls超过maxRequest则不再添加，否则一直添加。

最后附带一张网图：


OkHttpClient Builder
build()
OkHttpClient
Request
url
method
header
body
newCall()
RealCall
enqueue()
execute()
Dispatcher
execute()
getResponseWithInterceptorChain()
interceptors
Response
code
message
header
body
RetryAndFollowUpInterceptor
BridgeInterceptor
adapt
adapt
CacheInterceptor
Cache hit
Cache miss
ConnectInterceptor
Put into cache
pool
networkInterceptors
CallServerInterceptor
IO

@Piasy