【知识点】OkHttp 原理 8 连问

程序员江同学 郭霖 2022-08-23 08:00 发表于江苏

点击上方蓝字即可关注
关注后可查看所有经典文章

/ 今日科技快讯 /

近日，谷歌母公司Alphabet旗下视频平台YouTube已经删除了两段视频，这些视频显示特斯拉车主在道路或车道上用自己的孩子代替人体模型进行辅助驾驶系统“全自动驾驶”（FSD）进行安全测试。这些测试是为了确定安装了特斯拉最新司机辅助驾驶系统的车辆，在缓慢行驶的情况下，是否会自动避免与正在行走或者在路上站立不动的行人相撞。

/ 作者简介 /

本篇文章转自程序员江同学的博客，文章主要分享了他对OkHttp原理的探索分析，相信会对大家有所帮助！

原文地址：

> https://juejin.cn/post/7020027832977850381

/ 前言 /

OkHttp可以说是Android开发中最常见的网络请求框架，OkHttp使用方便，扩展性强，功能强大，OKHttp源码与原理也是面试中的常客。

但是OKHttp的源码内容比较多，想要学习它的源码往往千头万绪，一时抓不住重点。本文从几个问题出发梳理OKHttp相关知识点，以便快速构建OKHttp知识体系，如果对你有用，欢迎点赞~

本文主要包括以下内容：

1. OKHttp请求的整体流程是怎样的？
2. OKHttp分发器是怎样工作的？
3. OKHttp拦截器是如何工作的？
4. 应用拦截器和网络拦截器有什么区别？
5. OKHttp如何复用TCP连接？
6. OKHttp空闲连接如何清除？
7. OKHttp有哪些优点？
8. OKHttp框架中用到了哪些设计模式？

/ 正文 /

OKHttp请求整体流程介绍

首先来看一个最简单的Http请求是如何发送的。

```
val okHttpClient = OkHttpClient()
val request: Request = Request.Builder()
   .url("https://www.google.com/")
   .build()

okHttpClient.newCall(request).enqueue(object :Callback{
   override fun onFailure(call: Call, e: IOException) {
   }

   override fun onResponse(call: Call, response: Response) {
   }
})
```

这段代码看起来比较简单，OkHttp请求过程中最少只需要接触OkHttpClient、Request、Call、 Response，但是框架内部会进行大量的逻辑处理。

所有网络请求的逻辑大部分集中在拦截器中，但是在进入拦截器之前还需要依靠分发器来调配请求任务。

关于分发器与拦截器，我们在这里先简单介绍下，后续会有更加详细的讲解。

- 分发器：内部维护队列与线程池，完成请求调配；
- 拦截器：五大默认拦截器完成整个请求过程。

整个网络请求过程大致如上所示：

1. 通过建造者模式构建OKHttpClient与 Request
2. OKHttpClient通过newCall发起一个新的请求
3. 通过分发器维护请求队列与线程池，完成请求调配
4. 通过五大默认拦截器完成请求重试，缓存处理，建立连接等一系列操作
5. 得到网络请求结果

OKHttp分发器是怎样工作的?

分发器的主要作用是维护请求队列与线程池,比如我们有100个异步请求，肯定不能把它们同时请求，而是应该把它们排队分个类，分为正在请求中的列表和正在等待的列表，等请求完成后，即可从等待中的列表中取出等待的请求，从而完成所有的请求。

而这里同步请求各异步请求又略有不同。

同步请求

```
synchronized void executed(RealCall call) {
   runningSyncCalls.add(call);
}
```

因为同步请求不需要线程池，也不存在任何限制。所以分发器仅做一下记录。后续按照加入队列的顺序同步请求即可。

异步请求

```
synchronized void enqueue(AsyncCall call) {
   //请求数最大不超过64,同一Host请求不能超过5个
   if (runningAsyncCalls.size() < maxRequests && runningCallsForHost(call) < maxRequestsPe
      runningAsyncCalls.add(call);
      executorService().execute(call);
   } else {
      readyAsyncCalls.add(call);
   }
}
```

当正在执行的任务未超过最大限制64，同时同一Host的请求不超过5个，则会添加到正在执行队列，同时提交给线程池。否则先加入等待队列。每个任务完成后，都会调用分发器的finished方法，这里面会取出等待队列中的任务继续执行。

OKHttp拦截器是怎样工作的?

经过上面分发器的任务分发，下面就要利用拦截器开始一系列配置了。

```
# RealCall
 override fun execute(): Response {
  try {
   client.dispatcher.executed(this)
   return getResponseWithInterceptorChain()
  } finally {
   client.dispatcher.finished(this)
  }
 }
```

我们再来看下RealCall的execute方法，可以看出，最后返回了getResponseWithInterceptorChain，责任链的构建与处理其实就是在这个方法里面。

```
internal fun getResponseWithInterceptorChain(): Response {
  // Build a full stack of interceptors.
  val interceptors = mutableListOf<Interceptor>()
  interceptors += client.interceptors
  interceptors += RetryAndFollowUpInterceptor(client)
  interceptors += BridgeInterceptor(client.cookieJar)
  interceptors += CacheInterceptor(client.cache)
  interceptors += ConnectInterceptor
  if (!forWebSocket) {
   interceptors += client.networkInterceptors
  }
  interceptors += CallServerInterceptor(forWebSocket)

  val chain = RealInterceptorChain(
    call = this,interceptors = interceptors,index = 0
  )
  val response = chain.proceed(originalRequest)
 }
```

如上所示，构建了一个OkHttp拦截器的责任链。

责任链，顾名思义，就是用来处理相关事务责任的一条执行链，执行链上有多个节点，每个节点都有机会（条件匹配）处理请求事务，如果某个节点处理完了就可以根据实际业务需求传递给下一个节点继续处理或者返回处理完毕。

如上所示责任链添加的顺序及作用如下表所示：

拦截器	作用
应用拦截器	拿到的是原始请求，可以添加一些自定义header、通用参数、参数加密、网关接入等等。
RetryAndFollowUpInterceptor	处理错误重试和重定向
BridgeInterceptor	应用层和网络层的桥接拦截器，主要工作是为请求添加cookie、添加固定的header，比如Host、Content-Length、Content-Type、User-Agent等等，然后保存响应结果的cookie，如果响应使用gzip压缩过，则还需要进行解压。
CacheInterceptor	缓存拦截器，如果命中缓存则不会发起网络请求。
ConnectInterceptor	连接拦截器，内部会维护一个连接池，负责连接复用、创建连接（三次握手等等）、释放连接以及创建连接上的socket流。
networkInterceptors（网络拦截器）	用户自定义拦截器，通常用于监控网络层的数据传输。
CallServerInterceptor	请求拦截器，在前置准备工作完成后，真正发起了网络请求。

我们的网络请求就是这样经过责任链一级一级的递推下去，最终会执行到CallServerInterceptor的intercept方法，此方法会将网络响应的结果封装成一个Response对象并return。之后沿着责任链一级一级的回溯，最终就回到getResponseWithInterceptorChain方法的返回，如下图所示：

应用拦截器和网络拦截器有什么区别?

从整个责任链路来看，应用拦截器是最先执行的拦截器，也就是用户自己设置request属性后的原始请求，而网络拦截器位于ConnectInterceptor和CallServerInterceptor之间，此时网络链路已经准备好，只等待发送请求数据。它们主要有以下区别。

1. 首先，应用拦截器在RetryAndFollowUpInterceptor和CacheInterceptor之前，所以一旦发生错误重试或者网络重定向，网络拦截器可能执行多次，因为相当于进行了二次请求，但是应用拦截器永远只会触发一次。另外如果在CacheInterceptor中命中了缓存就不需要走网络请求了，因此会存在短路网络拦截器的情况。
2. 其次，除了CallServerInterceptor之外，每个拦截器都应该至少调用一次realChain.proceed方法。实际上在应用拦截器这层可以多次调用proceed方法（本地异常重试）或者不调用proceed方法（中断），但是网络拦截器这层连接已经准备好，可且仅可调用一次proceed方法。
3. 最后，从使用场景看，应用拦截器因为只会调用一次，通常用于统计客户端的网络请求发起情况；而网络拦截器一次调用代表了一定会发起一次网络通信，因此通常可用于统计网络链

路上传输的数据。

OKHttp如何复用TCP连接?

ConnectInterceptor的主要工作就是负责建立TCP连接，建立TCP连接需要经历三次握手四次挥手等操作，如果每个HTTP请求都要新建一个TCP消耗资源比较多。

而Http1.1已经支持keep-alive，即多个Http请求复用一个TCP连接，OKHttp也做了相应的优化，下面我们来看下OKHttp是怎么复用TCP连接的。

ConnectInterceptor中查找连接的代码会最终会调用到ExchangeFinder.findConnection方法，具体如下：

```
# ExchangeFinder
//为承载新的数据流 寻找 连接。寻找顺序是 已分配的连接、连接池、新建连接
private RealConnection findConnection(int connectTimeout, int readTimeout, int writeTimeo
   int pingIntervalMillis, boolean connectionRetryEnabled) throws IOException {
  synchronized (connectionPool) {
   // 1.尝试使用 已给数据流分配的连接.（例如重定向请求时，可以复用上次请求的连接）
   releasedConnection = transmitter.connection;
   result = transmitter.connection;

   if (result == null) {
    // 2. 没有已分配的可用连接，就尝试从连接池获取。（连接池稍后详细讲解）
    if (connectionPool.transmitterAcquirePooledConnection(address, transmitter, null, fa
     result = transmitter.connection;
    }
   }
  }

  synchronized (connectionPool) {
   if (newRouteSelection) {
    //3. 现在有了IP地址，再次尝试从连接池获取。可能会因为连接合并而匹配。（这里传入了routes，
null)
    routes = routeSelection.getAll();
    if (connectionPool.transmitterAcquirePooledConnection(address, transmitter, routes, f
     foundPooledConnection = true;
     result = transmitter.connection;
    }
   }

  // 4.第二次没成功，就把新建的连接，进行TCP + TLS 握手，与服务端建立连接. 是阻塞操作
```

```
    result.connect(connectTimeout, readTimeout, writeTimeout, pingIntervalMillis,
        connectionRetryEnabled, call, eventListener);

    synchronized (connectionPool) {
      // 5. 最后一次尝试从连接池获取，注意最后一个参数为true，即要求 多路复用（http2.0）
      //意思是，如果本次是http2.0，那么为了保证 多路复用性，（因为上面的握手操作不是线程安全）会再次确认
此时是否已有同样连接
      if (connectionPool.transmitterAcquirePooledConnection(address, transmitter, routes, tr
        // 如果获取到，就关闭我们创建里的连接，返回获取的连接
        result = transmitter.connection;
      } else {
        //最后一次尝试也没有的话，就把刚刚新建的连接存入连接池
        connectionPool.put(result);
      }
    }

    return result;
  }
```

上面精简了部分代码，可以看出，连接拦截器使用了5种方法查找连接。

1. 首先会尝试使用 已给请求分配的连接。（已分配连接的情况例如重定向时的再次请求，说明上次已经有了连接）
2. 若没有 已分配的可用连接，就尝试从连接池中匹配获取。因为此时没有路由信息，所以匹配条件：address一致——host、port、代理等一致，且匹配的连接可以接受新的请求。
3. 若从连接池没有获取到，则传入routes再次尝试获取，这主要是针对Http2.0的一个操作，Http2.0可以复用square.com与square.ca的连接
4. 若第二次也没有获取到，就创建RealConnection实例，进行TCP + TLS握手，与服务端建立连接。
5. 此时为了确保Http2.0连接的多路复用性，会第三次从连接池匹配。因为新建立的连接的握手过程是非线程安全的，所以此时可能连接池新存入了相同的连接。
6. 第三次若匹配到，就使用已有连接，释放刚刚新建的连接；若未匹配到，则把新连接存入连接池并返回。

以上就是连接拦截器尝试复用连接的操作，流程图如下：

OKHttp空闲连接如何清除?

上面说到我们会建立一个TCP连接池，但如果没有任务了，空闲的连接也应该及时清除，OKHttp是如何做到的呢?

```
# RealConnectionPool
private val cleanupQueue: TaskQueue = taskRunner.newQueue()
private val cleanupTask = object : Task("$okHttpName ConnectionPool") {
 override fun runOnce(): Long = cleanup(System.nanoTime())
}

long cleanup(long now) {
 int inUseConnectionCount = 0;//正在使用的连接数
 int idleConnectionCount = 0;//空闲连接数
 RealConnection longestIdleConnection = null;//空闲时间最长的连接
 long longestIdleDurationNs = Long.MIN_VALUE;//最长的空闲时间

 //遍历连接：找到待清理的连接，找到下一次要清理的时间（还未到最大空闲时间）
 synchronized (this) {
  for (Iterator<RealConnection> i = connections.iterator(); i.hasNext(); ) {
   RealConnection connection = i.next();

   //若连接正在使用，continue，正在使用连接数+1
   if (pruneAndGetAllocationCount(connection, now) > 0) {
    inUseConnectionCount++;
    continue;
   }
   //空闲连接数+1
   idleConnectionCount++;

   // 赋值最长的空闲时间和对应连接
   long idleDurationNs = now - connection.idleAtNanos;
   if (idleDurationNs > longestIdleDurationNs) {
    longestIdleDurationNs = idleDurationNs;
    longestIdleConnection = connection;
   }
  }
  //若最长的空闲时间大于5分钟 或 空闲数 大于5，就移除并关闭这个连接
  if (longestIdleDurationNs >= this.keepAliveDurationNs
    || idleConnectionCount > this.maxIdleConnections) {
```

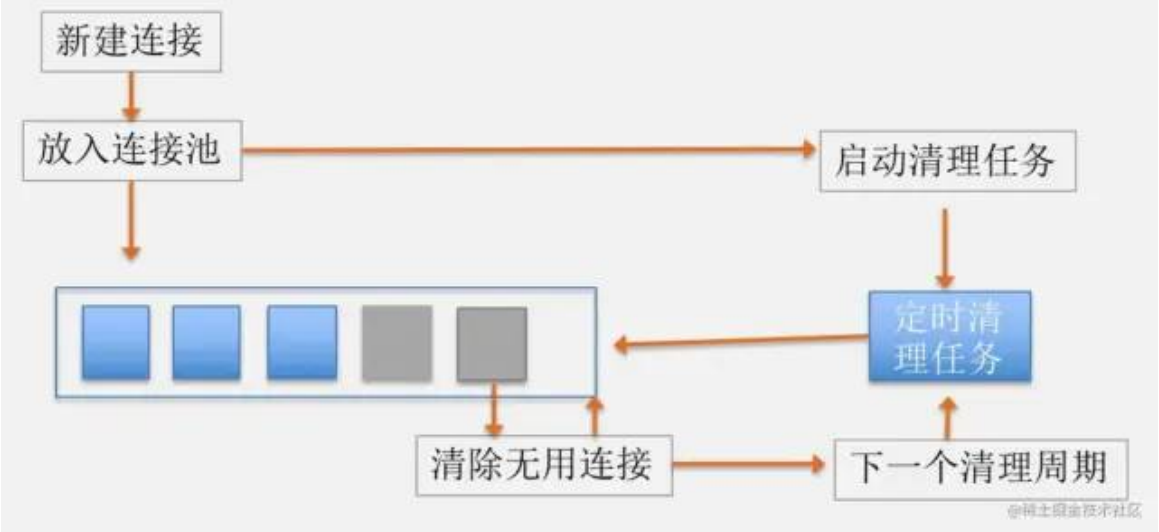
```
      connections.remove(longestIdleConnection);
    } else if (idleConnectionCount > 0) {
      // else，就返回 还剩多久到达5分钟，然后wait这个时间再来清理
      return keepAliveDurationNs - longestIdleDurationNs;
    } else if (inUseConnectionCount > 0) {
      //连接没有空闲的，就5分钟后再尝试清理.
      return keepAliveDurationNs;
    } else {
      // 没有连接，不清理
      cleanupRunning = false;
      return -1;
    }
  }
  //关闭移除的连接
  closeQuietly(longestIdleConnection.socket());

  //关闭移除后 立刻 进行下一次的 尝试清理
  return 0;
}
```

思路还是很清晰的：

1. 在将连接加入连接池时就会启动定时任务。
2. 有空闲连接的话，如果最长的空闲时间大于5分钟或空闲数大于5，就移除关闭这个最长空闲连接；如果空闲数不大于5且最长的空闲时间不大于5分钟，就返回到5分钟的剩余时间，然后等待这个时间再来清理。
3. 没有空闲连接就等5分钟后再尝试清理。
4. 没有连接不清理。

流程如下图所示：



OKHttp有哪些优点?

1. 使用简单，在设计时使用了外观模式，将整个系统的复杂性给隐藏起来，将子系统接口通过一个客户端OkHttpClient统一暴露出来。
2. 扩展性强，可以通过自定义应用拦截器与网络拦截器，完成用户各种自定义的需求
3. 功能强大，支持Spdy、Http1.X、Http2、以及WebSocket等多种协议
4. 通过连接池复用底层TCP(Socket)，减少请求延时
5. 无缝的支持GZIP减少数据流量
6. 支持数据缓存，减少重复的网络请求
7. 支持请求失败自动重试主机的其他ip，自动重定向

OKHttp框架中用到了哪些设计模式?

1. 构建者模式：OkHttpClient与Request的构建都用到了构建者模式。
2. 外观模式：OkHttp使用了外观模式,将整个系统的复杂性给隐藏起来，将子系统接口通过一个客户端OkHttpClient统一暴露出来。
3. 责任链模式：OKHttp的核心就是责任链模式，通过5个默认拦截器构成的责任链完成请求的配置。
4. 享元模式：享元模式的核心即池中复用，OKHttp复用TCP连接时用到了连接池，同时在异步请求中也用到了线程池。

/ 总结 /

本文主要梳理了OKHttp原理相关知识点，并回答了以下问题：

1. OKHttp请求的整体流程是怎样的？
2. OKHttp分发器是怎样工作的？
3. OKHttp拦截器是如何工作的？
4. 应用拦截器和网络拦截器有什么区别？
5. OKHttp如何复用TCP连接？
6. OKHttp空闲连接如何清除？
7. OKHttp有哪些优点？
8. OKHttp框架中用到了哪些设计模式？

如果对您有所帮助，欢迎点赞，谢谢~

推荐阅读：

我的新书，《第一行代码 第3版》已出版！

一个Android沉浸式状态栏上的黑科技

Android 13运行时权限变更一览

欢迎关注我的公众号

学习技术或投稿

▼

长按上图，识别图中二维码即可关注

阅读原文

喜欢此内容的人还喜欢

MongoDB 系列 - ObjectId() 是如何实现的 “千万级” 分布式唯一 ID?

编程界

Mybatis中SQL注入攻击的3种方式，真是防不胜防！

江南一点雨

科技与狠活？JDK19中的虚拟线程到底什么鬼？

Hollis