**HttpClient**

## 一、HTTP协议

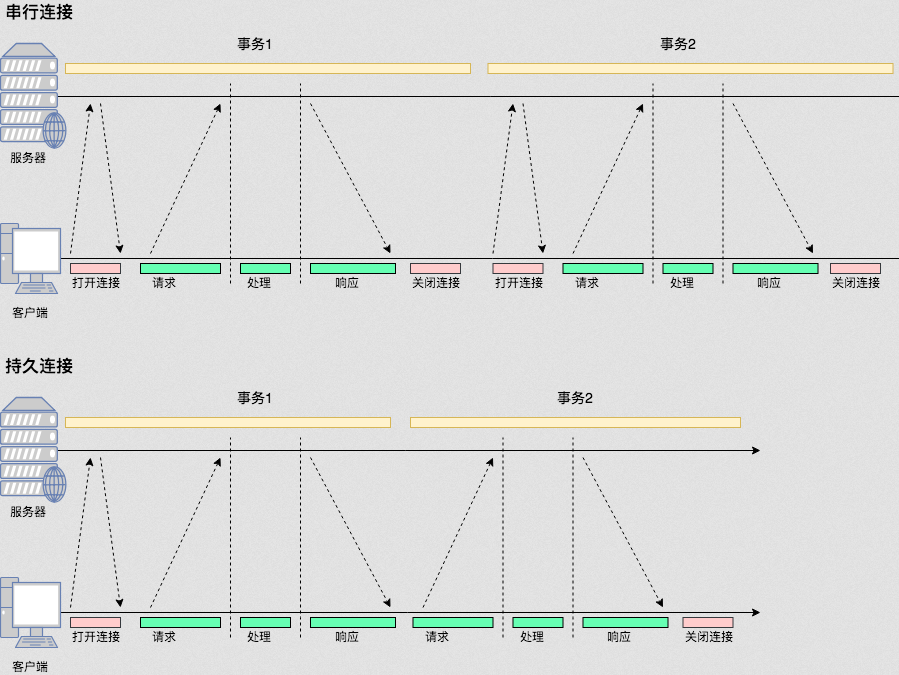
**1 HTTP协议是无状态的协议**

即每一次请求都是互相独立的。因此它的最初实现是，每一个http请求都会打开一个tcp socket连接，当交互完毕后会关闭这个连接。

HTTP协议是全双工的协议，所以建立连接与断开连接是要经过三次握手与四次挥手的。显然在这种设计中，每次发送Http请求都会消耗很多的额外资源，即连接的建立与销毁。

于是，HTTP协议的也进行了发展，通过持久连接的方法来进行socket连接复用。**在串行连接中，每次交互都要打开关闭连接，在持久连接中，第一次交互会打开连接，交互结束后连接并不关闭**，下次交互就省去了建立连接的过程。

持久连接的实现有两种：HTTP/1.0+的keep-alive与HTTP/1.1的持久连接。



**2 HTTP/1.0+的Keep-Alive**

从1996年开始，很多HTTP/1.0浏览器与服务器都对协议进行了扩展，那就是“keep-alive”扩展协议。注意，这个扩展协议是作为1.0的补充的“实验型持久连接”出现的。keep-alive已经不再使用了，最新的HTTP/1.1规范中也没有对它进行说明，只是很多应用延续了下来。

使用HTTP/1.0的客户端在首部中加上"Connection:Keep-Alive"，请求服务端将一条连接保持在打开状态。服务端如果愿意将这条连接保持在打开状态，就会在响应中包含同样的首部。

如果响应中没有包含"Connection:Keep-Alive"首部，则客户端会认为服务端不支持keep-alive，会在发送完响应报文之后关闭掉当前连接。

通过keep-alive补充协议，客户端与服务器之间完成了持久连接，仍然存在着一些问题：

（1）在HTTP/1.0中keep-alive不是标准协议，客户端必须发送Connection:Keep-Alive来激活keep-alive连接。

（2）代理服务器可能无法支持keep-alive，因为一些代理是"盲中继"，无法理解首部的含义，只是将首部逐跳转发。所以可能造成客户端与服务端都保持了连接，但是代理不接受该连接上的数据。

**3 HTTP/1.1的持久连接**

HTTP/1.1采取持久连接的方式替代了Keep-Alive。

HTTP/1.1的连接默认情况下都是持久连接。如果要显式关闭，需要在报文中加上Connection:Close首部。即在HTTP/1.1中，所有的连接都进行了复用。

然而如同Keep-Alive一样，空闲的持久连接也可以随时被客户端与服务端关闭。不发送Connection:Close不意味着服务器承诺连接永远保持打开。

HttpClien中使用了连接池来管理持有连接，同一条TCP链路上，连接是可以复用的。HttpClient通过连接池的方式进行连接持久化。

其实“池”技术是一种通用的设计，其设计思想并不复杂：

（1）当有连接第一次使用的时候建立连接

（2）结束时对应连接不关闭，归还到池中

（3）下次同个目的的连接可从池中获取一个可用连接

（4）定期清理过期连接

## 二、HttpClient项目

1 Http协议非常的重要，HttpClient相比传统JDK自带的URLConnection，增加了易用性和灵活性（具体区别，日后我们再讨论），它不仅是客户端发送Http请求变得容易，而且也方便了开发人员测试接口（基于Http协议的），即提高了开发的效率，也方便提高代码的健壮性。因此熟练掌握HttpClient是很重要的必修内容，掌握HttpClient后，相信对于Http协议的了解会更加深入。

**2 HttpClient是Apache Jakarta Common下的子项目，用来提供高效的、最新的、功能丰富的支持HTTP协议的客户端编程工具包**，并且它支持HTTP协议最新的版本和建议。

HttpClient已经应用在很多的项目中，比如Apache Jakarta上很著名的另外两个开源项目Cactus和HTMLUnit都使用了HttpClient。

3 HTTP是一个属于应用层的面向对象的协议，由于其简捷、快速的方式，适用于分布式超媒体信息系统。它于1990年提出，经过几年的使用与发展，得到不断地完善和扩展。

目前在WWW中使用的是HTTP/1.0的第六版，HTTP/1.1的规范化工作正在进行之中，而且HTTP-NG(Next Generation of HTTP)的建议已经提出。

## 三、HTTP协议的主要特点

**1.支持客户/服务器模式。**

**2.简单快速**：客户向服务器请求服务时，只需传送请求方法和路径。请求方法常用的有GET、HEAD、POST。每种方法规定了客户与服务器联系的类型不同。 由于HTTP协议简单，使得HTTP服务器的程序规模小，因而通信速度很快。

**3.灵活：**HTTP允许传输任意类型的数据对象。正在传输的类型由Content-Type加以标记。

**4.无连接：**无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。

**5.无状态**：HTTP协议是无状态协议。无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。

## 四、HttpClient使用方法

（1）准备工作

使用HttpClient发送请求、接收响应很简单，一般需要如下几步即可。首先需要导入HttpClientjar包，具体可以到官网下载，下载地址: http://hc.apache.org/downloads.cgi，commons-codec-1.7.jar，commons-logging-1.1.1.jar，httpclient-4.2.2.jar，httpcore-4.2.2.jar

1. . 创建HttpClient对象，最新版的httpClient使用实现类的是closeableHTTPClient,以前的default作废。

（3）. 创建请求方法的实例，并指定请求URL。如果需要发送GET请求，创建HttpGet对象；如果需要发送POST请求，创建HttpPost对象。

（4）. 如果需要发送请求参数，可调用HttpGet、HttpPost共同的setParams(HetpParams params)方法来添加请求参数；对于HttpPost对象而言，也可调用setEntity(HttpEntity entity)方法来设置请求参数。

（5）. 调用HttpClient对象的execute(HttpUriRequest request)发送请求，该方法返回一个HttpResponse。

（6）. 调用HttpResponse的getAllHeaders()、getHeaders(String name)等方法可获取服务器的响应头；

调用HttpResponse的getEntity()方法可获取HttpEntity对象，该对象包装了服务器的响应内容。程序可通过该对象获取服务器的响应内容。

（7） 释放连接。无论执行方法是否成功，都必须释放连接

try

{

// 创建一个默认的HttpClient

HttpClient httpclient =new DefaultHttpClient();

// 创建一个GET请求

HttpGet request =new HttpGet("www.google.com");

// 发送GET请求，并将响应内容转换成字符串

String response = httpclient.execute(request, new BasicResponseHandler());

Log.v("response text", response);

} catch (ClientProtocolException e)

{

e.printStackTrace();

} catch (IOException e)

{

e.printStackTrace();

}

## 五、多线程的HttpClient

**1 在实际项目中，我们很可能在多处需要进行HTTP通信**，这时候我们不需要为每个请求都创建一个新的HttpClient。现在我们的应用程序使用同一个HttpClient来管理所有的Http请求，一旦出现并发请求，那么一定会出现多线程的问题。这就好像我们的浏览器只有一个标签页却有多个用户，A要上google，B要上baidu，这时浏览器就会忙不过来了。

幸运的是，HttpClient提供了创建线程安全对象的API

public class CustomerHttpClient

{

private static final String CHARSET = HTTP.UTF\_8;

/\*\*

\* 最大连接数

\*/

public final static int MAX\_TOTAL\_CONNECTIONS = 800;

/\*\*

\* 获取连接的最大等待时间

\*/

public final static int WAIT\_TIMEOUT = 60000;

/\*\*

\* 每个路由最大连接数

\*/

public final static int MAX\_ROUTE\_CONNECTIONS = 400;

/\*\*

\* 连接超时时间

\*/

public final static int CONNECT\_TIMEOUT = 10000;

/\*\*

\* 读取超时时间

\*/

public final static int READ\_TIMEOUT = 10000;

private static HttpClient customerHttpClient;

private CustomerHttpClient() {

}

public static synchronized HttpClient getHttpClient() {

if (null == customerHttpClient) {

HttpParams params = new BasicHttpParams();

// 设置一些基本参数

HttpProtocolParams.setVersion(params, HttpVersion.HTTP\_1\_1);

HttpProtocolParams.setContentCharset(params,

CHARSET);

HttpProtocolParams.setUseExpectContinue(params, true);

HttpProtocolParams

.setUserAgent(

params,

"Mozilla/5.0(Linux;U;Android 2.2.1;en-us;Nexus One Build.FRG83) "

+ "AppleWebKit/553.1(KHTML,like Gecko) Version/4.0 Mobile Safari/533.1");

// 超时设置

/\* 从连接池中取连接的超时时间 \*/

ConnManagerParams.setTimeout(params, WAIT\_TIMEOUT);

/\* 连接超时 \*/

HttpConnectionParams.setConnectionTimeout(params, CONNECT\_TIMEOUT);

/\* 请求超时 \*/

HttpConnectionParams.setSoTimeout(params, READ\_TIMEOUT);

// 设置最大连接数

ConnManagerParams.setMaxTotalConnections(params, MAX\_TOTAL\_CONNECTIONS);

// 设置每个路由最大连接数

ConnPerRouteBean connPerRoute = new ConnPerRouteBean(MAX\_ROUTE\_CONNECTIONS);

ConnManagerParams.setMaxConnectionsPerRoute(params, connPerRoute);

// 设置我们的HttpClient支持HTTP和HTTPS两种模式

SchemeRegistry schReg = new SchemeRegistry();

schReg.register(new Scheme("http", PlainSocketFactory

.getSocketFactory(), 80));

schReg.register(new Scheme("https", SSLSocketFactory

.getSocketFactory(), 443));

// 使用线程安全的连接管理来创建HttpClient

ClientConnectionManager conMgr = new ThreadSafeClientConnManager(

params, schReg);

customerHttpClient = new DefaultHttpClient(conMgr, params);

}

return customerHttpClient;

}

}

**2、超时配置**

上面的代码提到了3种超时设置，比较容易搞混，HttpClient的3种超时说明

/\* 从连接池中取连接的超时时间 \*/

ConnManagerParams.setTimeout(params, 1000);

/\* 连接超时 \*/

HttpConnectionParams.setConnectionTimeout(params, 2000);

/\* 请求超时 \*/

HttpConnectionParams.setSoTimeout(params, 4000);

第一行设置ConnectionPoolTimeout：这定义了从ConnectionManager管理的连接池中取出连接的超时时间，此处设置为1秒。

第二行设置ConnectionTimeout：　　这定义了通过网络与服务器建立连接的超时时间。Httpclient包中通过一个异步线程去创建与服务器的socket连接，这就是该socket连接的超时时间，此处设置为2秒。

第三行设置SocketTimeout：这定义了Socket读数据的超时时间，即从服务器获取响应数据需要等待的时间，此处设置为4秒。

以上3种超时分别会抛出

ConnectionPoolTimeoutException,ConnectionTimeoutException与SocketTimeoutException。

**3、线程池配，ThreadSafeClientConnManager默认使用了连接池**

//设置最大连接数

ConnManagerParams.setMaxTotalConnections(httpParams, 10);

//设置最大路由连接数

ConnPerRouteBean connPerRoute = new ConnPerRouteBean(10);

ConnManagerParams.setMaxConnectionsPerRoute(httpParams, connPerRoute);

比较特别的是 每个路由(route)最大连接数。

什么是一个route？这里route的概念可以理解为运行环境机器到目标机器的一条线路。举例来说，我们使用HttpClient的实现来分别请求 www.baidu.com 的资源和 www.bing.com 的资源那么他就会产生两个route。这里为什么要特别提到route最大连接数这个参数呢，因为这个参数的默认值为2，如果不设置这个参数值默认情况下对于同一个目标机器的最大并发连接只有2个！这意味着如果你正在执行一个针对某一台目标机器的抓取任务的时候，哪怕你设置连接池的最大连接数为200，但是实际上还是只有2个连接在工作，其他剩余的198个连接都在等待，都是为别的目标机器服务的。

**4、工具类**

有了单例的HttpClient对象，我们就可以把一些常用的发出GET和POST请求的代码也封装起来，写进我们的工具类中了。POST请求示例：

private static final String TAG ="CustomerHttpClient";

public static String post(String url, NameValuePair... params) {

try {

// 编码参数

List<NameValuePair> formparams = new ArrayList<NameValuePair>(); // 请求参数

for (NameValuePair p : params) {

formparams.add(p);

}

UrlEncodedFormEntity entity = new UrlEncodedFormEntity(formparams,

HTTP.UTF\_8);

// 创建POST请求

HttpPost request =new HttpPost(url);

request.setEntity(entity);

// 发送请求

HttpClient client = getHttpClient();

HttpResponse response = client.execute(request);

if(response.getStatusLine().getStatusCode() != HttpStatus.SC\_OK) {

throw new RuntimeException("请求失败");

}

HttpEntity resEntity = response.getEntity();

return (resEntity ==null) ?null : EntityUtils.toString(resEntity, CHARSET);

} catch (UnsupportedEncodingException e) {

Log.w(TAG, e.getMessage());

return null;

} catch (ClientProtocolException e) {

Log.w(TAG, e.getMessage());

return null;

} catch (IOException e) {

throw new RuntimeException("连接失败", e);

}

}

**4、长连接和短连接**

长连接是指客户端与服务器端一旦建立连接以后，可以进行多次数据传输而不需重新建立连接，而短连接则每次数据传输都需要客户端和服务器端建立一次连接。

长连接的优势在于省去了每次数据传输连接建立的时间开销，能够大幅度提高数据传输的速度，对于P2P应用十分适合。

短连接每次数据传输都需要建立连接，我们知道HTTP协议的传输层协议是TCP协议，TCP连接的建立和释放分别需要进行3次握手和4次握手，频繁的建立连接即增加了时间开销，同时频繁的创建和销毁Socket同样是对服务器端资源的浪费。

对于诸如Web网站之类的B2C应用，并发请求量大，每一个用户又不需频繁的操作的场景下，维护大量的长连接对服务器无疑是一个巨大的考验。而此时，短连接可能更加适用。

而对于需要频繁发送HTTP请求的应用，需要在客户端使用HTTP长连接。

**5、连接池**

连接池管理的对象是长连接。连接池技术作为创建和管理连接的缓冲池技术，目前已广泛用于诸如数据库连接等长连接的维护和管理中，能够有效减少系统的响应时间，节省服务器资源开销。其优势主要有两个：其一是减少创建连接的资源开销，其二是资源的访问控制。

HTTP连接是无状态的，这样很容易给我们造成HTTP连接是短连接的错觉，实际上HTTP1.1默认即是持久连接，**HTTP1.0也可以通过在请求头中设置Connection:keep-alive使得连接为长连接。**

**6、HttpConnection**

没有连接池的概念，多少次请求就会建立多少个IO，在访问量巨大的情况下服务器的IO可能会耗尽。

**7、HttpClient3**

也有连接池的东西在里头，使用MultiThreadedHttpConnectionManager，大致过程如下：

MultiThreadedHttpConnectionManager connectionManager = new MultiThreadedHttpConnectionManager();

HttpClient client = new HttpClient(connectionManager);...// 在某个线程中。

GetMethod get = new GetMethod("http://jakarta.apache.org/");

try {

client.executeMethod(get);// print response to stdout

System.out.println(get.getResponseBodyAsStream());

} finally {

// be sure the connection is released back to the connection

managerget.releaseConnection();

}

可以看出来，它的方式与jdbc连接池的使用方式相近，比较不爽的就是需要手动调用releaseConnection去释放连接。对每一个HttpClient.executeMethod须有一个method.releaseConnection()与之匹配。

**8、HttpClient4**

HTTP Client4.0的ThreadSafeClientConnManager实现了HTTP连接的池化管理，其管理连接的基本单位是Route（路由），每个路由上都会维护一定数量的HTTP连接。

这里的Route的概念可以理解为客户端机器到目标机器的一条线路，例如使用HttpClient的实现来分别请求 www.163.com 的资源和 www.sina.com 的资源就会产生两个route。缺省条件下对于每个Route，HttpClient仅维护2个连接，总数不超过20个连接，显然对于大多数应用来讲，都是不够用的，可以通过设置HTTP参数进行调整。

HttpParams params = new BasicHttpParams();

//将每个路由的最大连接数增加到200

ConnManagerParams.setMaxTotalConnections(params,200);

// 将每个路由的默认连接数设置为20

ConnPerRouteBean connPerRoute = new ConnPerRouteBean(20);

// 设置某一个IP的最大连接数

HttpHost localhost = new HttpHost("locahost", 80);

connPerRoute.setMaxForRoute(new HttpRoute(localhost), 50);

ConnManagerParams.setMaxConnectionsPerRoute(params, connPerRoute);

SchemeRegistry schemeRegistry = new SchemeRegistry();

schemeRegistry.register(

new Scheme("http", PlainSocketFactory.getSocketFactory(), 80));

schemeRegistry.register(

new Scheme("https", SSLSocketFactory.getSocketFactory(), 443));

ClientConnectionManager cm = new ThreadSafeClientConnManager(params, schemeRegistry);

HttpClient httpClient = new DefaultHttpClient(cm, params);

可以配置的HTTP参数有：

1） http.conn-manager.timeout 当某一线程向连接池请求分配线程时，如果连接池已经没有可以分配的连接时，该线程将会被阻塞，直至http.conn-manager.timeout超时，抛出ConnectionPoolTimeoutException。

2） http.conn-manager.max-per-route 每个路由的最大连接数；

3） http.conn-manager.max-total 总的连接数；

**9、过期长连接**

连接的有效性检测是所有连接池都面临的一个通用问题，大部分HTTP服务器为了控制资源开销，并不会永久的维护一个长连接，而是一段时间就会关闭该连接。放回连接池的连接，如果在服务器端已经关闭，客户端是无法检测到这个状态变化而及时的关闭Socket的。这就造成了线程从连接池中获取的连接不一定是有效的。这个问题的一个解决方法就是在每次请求之前检查该连接是否已经存在了过长时间，可能已过期。但是这个方法会使得每次请求都增加额外的开销。

**HTTP Client4.0的ThreadSafeClientConnManager 提供了closeExpiredConnections()方法和closeIdleConnections()方法来解决该问题。前一个方法是清除连接池中所有过期的连接，至于连接什么时候过期可以设置，设置方法将在下面提到，而后一个方法则是关闭一定时间空闲的连接，可以使用一个单独的线程完成这个工作。**

public static class IdleConnectionMonitorThread extends Thread {

private final ClientConnectionManager connMgr;

private volatile boolean shutdown;

public IdleConnectionMonitorThread(ClientConnectionManager connMgr) {

super();

this.connMgr = connMgr;

}

@Override

public void run() {

try {

while (!shutdown) {

synchronized (this) {

wait(5000);

// 关闭过期的连接

connMgr.closeExpiredConnections();

// 关闭空闲时间超过30秒的连接

connMgr.closeIdleConnections(30, TimeUnit.SECONDS);

}

}

} catch (InterruptedException ex) {

// terminate

}

}

public void shutdown() {

shutdown = true;

synchronized (this) {

notifyAll();

}

刚才提到，客户端可以设置连接的过期时间，可以通过HttpClient的setKeepAliveStrategy方法设置连接的过期时间，这样就可以配合closeExpiredConnections()方法解决连接池中连接失效的。

DefaultHttpClient httpclient = new DefaultHttpClient();

httpclient.setKeepAliveStrategy(new ConnectionKeepAliveStrategy() {

public long getKeepAliveDuration(HttpResponse response, HttpContext context) {

// Honor 'keep-alive' header

HeaderElementIterator it = new BasicHeaderElementIterator(

response.headerIterator(HTTP.CONN\_KEEP\_ALIVE));

while (it.hasNext()) {

HeaderElement he = it.nextElement();

String param = he.getName();

String value = he.getValue();

if (value != null && param.equalsIgnoreCase("timeout")) {

try {

return Long.parseLong(value) \* 1000;

} catch(NumberFormatException ignore) {

}

}

}

HttpHost target = (HttpHost) context.getAttribute(

ExecutionContext.HTTP\_TARGET\_HOST);

if ("www.163.com".equalsIgnoreCase(target.getHostName())) {

// 对于163这个路由的连接，保持5秒

return 5 \* 1000;

} else {

// 其他路由保持30秒

return 30 \* 1000;

}

}

})

## 六、httpGet和httpPost的区别

**1 HttpClient常用HttpGet和HttpPost这两个类，分别对应Get方式和Post方式。**

HttpPost方法提交HTTP POST请求，需要使用HttpPost类的setEntity方法设置请求参数。参数则必须用NameValuePair[]数组存储。

**2 get方式：**以URL字串本身传递数据参数，在服务器端可以从'QUERY\_STRING'这个变量中直接读取，效率较高，但缺乏安全性，也无法来处理复杂的数据。

**3 post方式：**就传输方式讲参数会被打包在数据包中传输，从CONTENT\_LENGTH这个环境变量中读取，便于传送较大一些的数据，同时因为不暴露数据在浏览器的地址栏中，安全性相对较高，但这样的处理效率会受到影响。

**4 表单提交中的post和get方法的区别：**

1.get是从服务器上查询/获取数据，post是向服务器传输数据。

2. get是把参数数据队列加到提交表单的ACTION属性所指的URL中，值和表单内各个字段一一对应，在URL中可以看到。post是通过HTTP post机制，将表单内各个字段与其内容放置在HTML HEADER内一起传送到ACTION属性所指的URL地址。用户看不到这个过程。

3. 对于get方式，服务器端用Request.QueryString获取变量的值，对于post方式，服务器端用Request.Form获取提交的数据。

4. get传送的数据量较小，不能大于2KB。post传送的数据量较大，一般被默认为不受限制。但理论上，IIS4中最大量为80KB，IIS5中为100KB。

5. get安全性非常低，传输数据可见，post安全性较高，传输数据不可见，但通过抓包工具post传递中的参数也可以看到，所以理论上也不是安全的。

6. get是Form的默认方法。

## 七、Http连接池

随着微服务的流行，服务之间的http调用越来越多。

**1、为什么要用Http连接池**

**（1）降低延迟**：如果不采用连接池，每次连接发起Http请求的时候都会重新建立TCP连接(经历3次握手)，用完就会关闭连接(4次挥手)，如果采用连接池则减少了这部分时间损耗，别小看这几次握手，本人经过测试发现，基本上3倍的时间延迟

**（2）支持更大的并发：**如果不采用连接池，每次连接都会打开一个端口，在大并发的情况下系统的端口资源很快就会被用完，导致无法建立新的连接

**2、HttpConnectionManager.java连接池管理类，支持https协议**

@Component

public class HttpConnectionManager {

PoolingHttpClientConnectionManager cm = null;

@PostConstruct

public void init() {

LayeredConnectionSocketFactory sslsf = null;

try {

sslsf = new SSLConnectionSocketFactory(SSLContext.getDefault());

} catch (NoSuchAlgorithmException e) {

e.printStackTrace();

}

Registry<ConnectionSocketFactory> socketFactoryRegistry = RegistryBuilder.<ConnectionSocketFactory> create()

.register("https", sslsf)

.register("http", new PlainConnectionSocketFactory())

.build();

cm =new PoolingHttpClientConnectionManager(socketFactoryRegistry);

cm.setMaxTotal(200);

cm.setDefaultMaxPerRoute(20);

}

public CloseableHttpClient getHttpClient() {

CloseableHttpClient httpClient = HttpClients.custom()

.setConnectionManager(cm)

.build();

/\*CloseableHttpClient httpClient = HttpClients.createDefault();//如果不采用连接池就是这种方式获取连接\*/

return httpClient;

}

}

**3、连接池消费类：HaoMaiClient.java**

@Component

public class HaoMaiClient {

@Autowired

HttpConnectionManager connManager;

public <T> T get(String path,Class<T> clazz){

CloseableHttpClient httpClient=connManager.getHttpClient();

HttpGet httpget = new HttpGet(path);

String json=null;

CloseableHttpResponse response=null;

try {

response = httpClient.execute(httpget);

InputStream in=response.getEntity().getContent();

json=IOUtils.toString(in);

in.close();

} catch (UnsupportedOperationException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}finally {

if(response!=null){

try {

response.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

return JSON.parseObject(json, clazz);

}

}

**4、原理及注意事项**

连接池中连接都是在发起请求的时候建立，并且都是长连接。HaoMaiClient.java中的in.close();作用就是将用完的连接释放，下次请求可以复用，这里特别注意的是，如果不使用in.close();而仅仅使用response.close();结果就是连接会被关闭，并且不能被复用，这样就失去了采用连接池的意义。

连接池释放连接的时候，并不会直接对TCP连接的状态有任何改变，只是维护了两个Set，leased和avaliabled，leased代表被占用的连接集合，avaliabled代表可用的连接的集合，释放连接的时候仅仅是将连接从leased中remove掉了，并把连接放到avaliabled集合中

1. **HttpClient连接池的实现**

**（1）MainClientExec**

HttpClient关于持久连接的处理在下面的代码中可以集中体现，下面从MainClientExec摘取了和连接池相关的部分，去掉了其他部分：

public class MainClientExec implements ClientExecChain {

@Override

public CloseableHttpResponse execute(

final HttpRoute route,

final HttpRequestWrapper request,

final HttpClientContext context,

final HttpExecutionAware execAware) throws IOException, HttpException {

　　　　　//从连接管理器HttpClientConnectionManager中获取一个连接请求ConnectionRequest

final ConnectionRequest connRequest = connManager.requestConnection(route, userToken);final HttpClientConnection managedConn;

final int timeout = config.getConnectionRequestTimeout(); //从连接请求ConnectionRequest中获取一个被管理的连接HttpClientConnection

managedConn = connRequest.get(timeout > 0 ? timeout : 0, TimeUnit.MILLISECONDS);

　　　　 //将连接管理器HttpClientConnectionManager与被管理的连接HttpClientConnection交给一个ConnectionHolder持有

final ConnectionHolder connHolder = new ConnectionHolder(this.log, this.connManager, managedConn);

try {

HttpResponse response;

if (!managedConn.isOpen()) {　　　　　　　　　　//如果当前被管理的连接不是出于打开状态，需要重新建立连接

establishRoute(proxyAuthState, managedConn, route, request, context);

}

　　　　　　　//通过连接HttpClientConnection发送请求

response = requestExecutor.execute(request, managedConn, context);

　　　　　　　//通过连接重用策略判断是否连接可重用

if (reuseStrategy.keepAlive(response, context)) {

//获得连接有效期

final long duration = keepAliveStrategy.getKeepAliveDuration(response, context);

//设置连接有效期

connHolder.setValidFor(duration, TimeUnit.MILLISECONDS);　　　　　　　　　 //将当前连接标记为可重用状态

connHolder.markReusable();

} else {

connHolder.markNonReusable();

}

}

final HttpEntity entity = response.getEntity();

if (entity == null || !entity.isStreaming()) {

//将当前连接释放到池中，供下次调用

connHolder.releaseConnection();

return new HttpResponseProxy(response, null);

} else {

return new HttpResponseProxy(response, connHolder);

}

}

这里看到了在Http请求过程中对连接的处理是和协议规范是一致的，这里要展开讲一下具体实现。

**（2）PoolingHttpClientConnectionManager**

PoolingHttpClientConnectionManager是HttpClient默认的连接管理器，首先通过requestConnection()获得一个连接的请求，注意这里不是连接。

public ConnectionRequest requestConnection(

final HttpRoute route,

final Object state) {final Future<CPoolEntry> future = this.pool.lease(route, state, null);

return new ConnectionRequest() {

@Override

public boolean cancel() {

return future.cancel(true);

}

@Override

public HttpClientConnection get(

final long timeout,

final TimeUnit tunit) throws InterruptedException, ExecutionException, ConnectionPoolTimeoutException {

final HttpClientConnection conn = leaseConnection(future, timeout, tunit);

if (conn.isOpen()) {

final HttpHost host;

if (route.getProxyHost() != null) {

host = route.getProxyHost();

} else {

host = route.getTargetHost();

}

final SocketConfig socketConfig = resolveSocketConfig(host);

conn.setSocketTimeout(socketConfig.getSoTimeout());

}

return conn;

}

};

}

可以看到返回的ConnectionRequest对象实际上是一个持有了Future<CPoolEntry>，CPoolEntry是被连接池管理的真正连接实例。

从上面的代码我们应该关注的是：

Future<CPoolEntry> future = this.pool.lease(route, state, null)

　　如何从连接池CPool中获得一个异步的连接，Future<CPoolEntry>

HttpClientConnection conn = leaseConnection(future, timeout, tunit)

　　如何通过异步连接Future<CPoolEntry>获得一个真正的连接HttpClientConnection

**（3） Future<CPoolEntry>**

看一下CPool是如何释放一个Future<CPoolEntry>的，AbstractConnPool核心代码如下：

private E getPoolEntryBlocking(

final T route, final Object state,

final long timeout, final TimeUnit tunit,

final Future<E> future) throws IOException, InterruptedException, TimeoutException {

　　　　　//首先对当前连接池加锁，当前锁是可重入锁ReentrantLockthis.lock.lock();

try {　　　　　　　 //获得一个当前HttpRoute对应的连接池，对于HttpClient的连接池而言，总池有个大小，每个route对应的连接也是个池，所以是“池中池”

final RouteSpecificPool<T, C, E> pool = getPool(route);

E entry;

for (;;) {

Asserts.check(!this.isShutDown, "Connection pool shut down");　　　　　　　　　　//死循环获得连接

for (;;) {　　　　　　　　　　　　//从route对应的池中拿连接，可能是null，也可能是有效连接

entry = pool.getFree(state);　　　　　　　　　　　　//如果拿到null，就退出循环

if (entry == null) {

break;

}　　　　　　　　　　　　//如果拿到过期连接或者已关闭连接，就释放资源，继续循环获取

if (entry.isExpired(System.currentTimeMillis())) {

entry.close();

}

if (entry.isClosed()) {

this.available.remove(entry);

pool.free(entry, false);

} else {　　　　　　　　　　　　　　//如果拿到有效连接就退出循环

break;

}

}　　　　　　　　　　//拿到有效连接就退出

if (entry != null) {

this.available.remove(entry);

this.leased.add(entry);

onReuse(entry);

return entry;

}

　　　　　　　　　 //到这里证明没有拿到有效连接，需要自己生成一个

final int maxPerRoute = getMax(route);

//每个route对应的连接最大数量是可配置的，如果超过了，就需要通过LRU清理掉一些连接

final int excess = Math.max(0, pool.getAllocatedCount() + 1 - maxPerRoute);

if (excess > 0) {

for (int i = 0; i < excess; i++) {

final E lastUsed = pool.getLastUsed();

if (lastUsed == null) {

break;

}

lastUsed.close();

this.available.remove(lastUsed);

pool.remove(lastUsed);

}

}

　　　　　　　　　 //当前route池中的连接数，没有达到上线

if (pool.getAllocatedCount() < maxPerRoute) {

final int totalUsed = this.leased.size();

final int freeCapacity = Math.max(this.maxTotal - totalUsed, 0);　　　　　　　　　　　　//判断连接池是否超过上线，如果超过了，需要通过LRU清理掉一些连接

if (freeCapacity > 0) {

final int totalAvailable = this.available.size();　　　　　　　　　　　　　　 //如果空闲连接数已经大于剩余可用空间，则需要清理下空闲连接

if (totalAvailable > freeCapacity - 1) {

if (!this.available.isEmpty()) {

final E lastUsed = this.available.removeLast();

lastUsed.close();

final RouteSpecificPool<T, C, E> otherpool = getPool(lastUsed.getRoute());

otherpool.remove(lastUsed);

}

}　　　　　　　　　　　　　　//根据route建立一个连接

final C conn = this.connFactory.create(route);　　　　　　　　　　　　　　//将这个连接放入route对应的“小池”中

entry = pool.add(conn);　　　　　　　　　　　　　　//将这个连接放入“大池”中

this.leased.add(entry);

return entry;

}

}

　　　　　　　　　//到这里证明没有从获得route池中获得有效连接，并且想要自己建立连接时当前route连接池已经到达最大值，即已经有连接在使用，但是对当前线程不可用

boolean success = false;

try {

if (future.isCancelled()) {

throw new InterruptedException("Operation interrupted");

}　　　　　　　　　　　　//将future放入route池中等待

pool.queue(future);　　　　　　　　　　　　//将future放入大连接池中等待

this.pending.add(future);　　　　　　　　　　　　//如果等待到了信号量的通知,success为true

if (deadline != null) {

success = this.condition.awaitUntil(deadline);

} else {

this.condition.await();

success = true;

}

if (future.isCancelled()) {

throw new InterruptedException("Operation interrupted");

}

} finally {

//从等待队列中移除

pool.unqueue(future);

this.pending.remove(future);

}

//如果没有等到信号量通知并且当前时间已经超时，则退出循环

if (!success && (deadline != null && deadline.getTime() <= System.currentTimeMillis())) {

break;

}

}　　　　　　　//最终也没有等到信号量通知，没有拿到可用连接，则抛异常

throw new TimeoutException("Timeout waiting for connection");

} finally {　　　　　　　//释放对大连接池的锁

this.lock.unlock();

}

}

**上面的代码逻辑有几个重要点：**

（1）连接池有个最大连接数，每个route对应一个小连接池，也有个最大连接数

（2）不论是大连接池还是小连接池，当超过数量的时候，都要通过LRU释放一些连接

（3）如果拿到了可用连接，则返回给上层使用

（4）如果没有拿到可用连接，HttpClient会判断当前route连接池是否已经超过了最大数量，没有到上限就会新建一个连接，并放入池中

（5）如果到达了上限，就排队等待，等到了信号量，就重新获得一次，等待不到就抛超时异常

（6）通过线程池获取连接要通过ReetrantLock加锁，保证线程安全

**（4） HttpClientConnection**

protected HttpClientConnection leaseConnection(

final Future<CPoolEntry> future,

final long timeout,

final TimeUnit tunit) throws InterruptedException, ExecutionException, ConnectionPoolTimeoutException {

final CPoolEntry entry;

try {　　　　　　　//从异步操作Future<CPoolEntry>中获得CPoolEntry

entry = future.get(timeout, tunit);

if (entry == null || future.isCancelled()) {

throw new InterruptedException();

}

Asserts.check(entry.getConnection() != null, "Pool entry with no connection");

if (this.log.isDebugEnabled()) {

this.log.debug("Connection leased: " + format(entry) + formatStats(entry.getRoute()));

}　　　　　　　//获得一个CPoolEntry的代理对象，对其操作都是使用同一个底层的HttpClientConnection

return CPoolProxy.newProxy(entry);

} catch (final TimeoutException ex) {

throw new ConnectionPoolTimeoutException("Timeout waiting for connection from pool");

}

}

## 八、HttpClient如何复用持久连接

**1 在上一章中，我们看到了HttpClient通过连接池来获得连接**，当需要使用连接的时候从池中获得。

当有连接第一次使用的时候建立连接

结束时对应连接不关闭，归还到池中

下次同个目的的连接可从池中获取一个可用连接

定期清理过期连接

**2 我们在第四章中看到了HttpClient是如何处理1、3的问题的**，那么第2个问题是怎么处理的呢？

即HttpClient如何判断一个连接在使用完毕后是要关闭，还是要放入池中供他人复用？再看一下MainClientExec的代码

//发送Http连接 response = requestExecutor.execute(request, managedConn, context);

//根据重用策略判断当前连接是否要复用

if (reuseStrategy.keepAlive(response, context)) {

//需要复用的连接，获取连接超时时间，以response中的timeout为准

final long duration = keepAliveStrategy.getKeepAliveDuration(response, context);

if (this.log.isDebugEnabled()) {

final String s;　　　　　　　　　　　　　　 //timeout的是毫秒数，如果没有设置则为-1，即没有超时时间

if (duration > 0) {

s = "for " + duration + " " + TimeUnit.MILLISECONDS;

} else {

s = "indefinitely";

}

this.log.debug("Connection can be kept alive " + s);

}　　　　　　　　　　　　//设置超时时间，当请求结束时连接管理器会根据超时时间决定是关闭还是放回到池中

connHolder.setValidFor(duration, TimeUnit.MILLISECONDS);

//将连接标记为可重用　　　　　　　　　　　　connHolder.markReusable();

} else {　　　　　　　　　　　　//将连接标记为不可重用

connHolder.markNonReusable();

}

**3 可以看到，当使用连接发生过请求之后**，有连接重试策略来决定该连接是否要重用，如果要重用就会在结束后交给HttpClientConnectionManager放入池中。

那么连接复用策略的逻辑是怎么样的呢？

public class DefaultClientConnectionReuseStrategy extends DefaultConnectionReuseStrategy {

public static final DefaultClientConnectionReuseStrategy INSTANCE = new DefaultClientConnectionReuseStrategy();

@Override

public boolean keepAlive(final HttpResponse response, final HttpContext context) {

　　　　　//从上下文中拿到request

final HttpRequest request = (HttpRequest) context.getAttribute(HttpCoreContext.HTTP\_REQUEST);

if (request != null) {　　　　　　　//获得Connection的Header

final Header[] connHeaders = request.getHeaders(HttpHeaders.CONNECTION);

if (connHeaders.length != 0) {

final TokenIterator ti = new BasicTokenIterator(new BasicHeaderIterator(connHeaders, null));

while (ti.hasNext()) {

final String token = ti.nextToken();　　　　　　　　　　　　//如果包含Connection:Close首部，则代表请求不打算保持连接，会忽略response的意愿，该头部这是HTTP/1.1的规范

if (HTTP.CONN\_CLOSE.equalsIgnoreCase(token)) {

return false;

}

}

}

}　　　　 //使用父类的的复用策略

return super.keepAlive(response, context);

}

}

**4 看一下父类的复用策略**

if (canResponseHaveBody(request, response)) {

final Header[] clhs = response.getHeaders(HTTP.CONTENT\_LEN);

//如果reponse的Content-Length没有正确设置，则不复用连接　　　　　　　　　 //因为对于持久化连接，两次传输之间不需要重新建立连接，则需要根据Content-Length确认内容属于哪次请求，以正确处理“粘包”现象 //所以，没有正确设置Content-Length的response连接不能复用

if (clhs.length == 1) {

final Header clh = clhs[0];

try {

final int contentLen = Integer.parseInt(clh.getValue());

if (contentLen < 0) {

return false;

}

} catch (final NumberFormatException ex) {

return false;

}

} else {

return false;

}

}

if (headerIterator.hasNext()) {

try {

final TokenIterator ti = new BasicTokenIterator(headerIterator);

boolean keepalive = false;

while (ti.hasNext()) {

final String token = ti.nextToken();　　　　　　　　　　　　//如果response有Connection:Close首部，则明确表示要关闭，则不复用

if (HTTP.CONN\_CLOSE.equalsIgnoreCase(token)) {

return false;　　　　　　　　　　　　//如果response有Connection:Keep-Alive首部，则明确表示要持久化，则复用

} else if (HTTP.CONN\_KEEP\_ALIVE.equalsIgnoreCase(token)) {

keepalive = true;

}

}

if (keepalive) {

return true;

}

} catch (final ParseException px) {

return false;

}

}

　//如果response中没有相关的Connection首部说明，则高于HTTP/1.0版本的都复用连接

return !ver.lessEquals(HttpVersion.HTTP\_1\_0);

**5 总结**

（1）如果request首部中包含Connection:Close，不复用

（2）如果response中Content-Length长度设置不正确，不复用

（3）如果response首部包含Connection:Close，不复用

（4）如果reponse首部包含Connection:Keep-Alive，复用

（5）都没命中的情况下，如果HTTP版本高于1.0则复用

## 九、HttpClient如何清理过期连接

**1 在HttpClient4.4版本之前，在从连接池中获取重用连接的时候会检查下是否过期，过期则清理。**之后的版本则不同，会有一个单独的线程来扫描连接池中的连接，发现有离最近一次使用超过设置的时间后，就会清理。默认的超时时间是2秒钟。

public CloseableHttpClient build() { //如果指定了要清理过期连接与空闲连接，才会启动清理线程，默认是不启动的

if (evictExpiredConnections || evictIdleConnections) {　　　　　　　　　　//创造一个连接池的清理线程

final IdleConnectionEvictor connectionEvictor = new IdleConnectionEvictor(cm,

maxIdleTime > 0 ? maxIdleTime : 10, maxIdleTimeUnit != null ? maxIdleTimeUnit : TimeUnit.SECONDS,

maxIdleTime, maxIdleTimeUnit);

closeablesCopy.add(new Closeable() {

@Override

public void close() throws IOException {

connectionEvictor.shutdown();

try {

connectionEvictor.awaitTermination(1L, TimeUnit.SECONDS);

} catch (final InterruptedException interrupted) {

Thread.currentThread().interrupt();

}

}

});　　　　　　　　　　//执行该清理线程

connectionEvictor.start();

}

**2 可以看到在HttpClientBuilder进行build**的时候,如果指定了开启清理功能，会创建一个连接池清理线程并运行它。

public IdleConnectionEvictor(

final HttpClientConnectionManager connectionManager,

final ThreadFactory threadFactory,

final long sleepTime, final TimeUnit sleepTimeUnit,

final long maxIdleTime, final TimeUnit maxIdleTimeUnit) {

this.connectionManager = Args.notNull(connectionManager, "Connection manager");

this.threadFactory = threadFactory != null ? threadFactory : new DefaultThreadFactory();

this.sleepTimeMs = sleepTimeUnit != null ? sleepTimeUnit.toMillis(sleepTime) : sleepTime;

this.maxIdleTimeMs = maxIdleTimeUnit != null ? maxIdleTimeUnit.toMillis(maxIdleTime) : maxIdleTime;

this.thread = this.threadFactory.newThread(new Runnable() {

@Override

public void run() {

try {　　　　　　　　　　　　//死循环，线程一直执行

while (!Thread.currentThread().isInterrupted()) {　　　　　　　　　　　　　　//休息若干秒后执行，默认10秒

Thread.sleep(sleepTimeMs);　　　　　　　　　　　　　　 //清理过期连接

connectionManager.closeExpiredConnections();　　　　　　　　　　　　　　 //如果指定了最大空闲时间，则清理空闲连接

if (maxIdleTimeMs > 0) {

connectionManager.closeIdleConnections(maxIdleTimeMs, TimeUnit.MILLISECONDS);

}

}

} catch (final Exception ex) {

exception = ex;

}

}

});

}

**3 总结**

@只有在HttpClientBuilder手动设置后，才会开启清理过期与空闲连接。

@手动设置后，会启动一个线程死循环执行，每次执行sleep一定时间，调用HttpClientConnectionManager的清理方法清理过期与空闲连接。

@HTTP协议通过持久连接的方式，减轻了早期设计中的过多连接问题。

@持久连接有两种方式：HTTP/1.0+的Keep-Avlive与HTTP/1.1的默认持久连接

@HttpClient通过连接池来管理持久连接，连接池分为两个，一个是总连接池，一个是每个route对应的连接池。

@HttpClient通过异步的Future<CPoolEntry>来获取一个池化的连接。

@默认连接重用策略与HTTP协议约束一致，根据response先判断Connection:Close则关闭，在判断Connection:Keep-Alive则开启，最后版本大于1.0则开启。

@只有在HttpClientBuilder中手动开启了清理过期与空闲连接的开关后，才会清理连接池中的连接。

@HttpClient4.4之后的版本通过一个死循环线程清理过期与空闲连接，该线程每次执行都sleep一会，以达到定期执行的效果。

## 十、HttpClient问题诊断

**1 最近发现数据同步迟滞了,** 于是立马去check下服务进程. 发现进程活得好好的(白白胖胖那种), 稍松了口气, 看来没有OOM, 以及被误杀. 但日志没有滚动, 也就是说日志好久没更新了. 这是啥回事呢?

　　难道说worker线程退出江湖了, 死锁大爷又来踢馆了, 还是说路遇张麻子(此山是我开, 此树是我栽)?使用jstack -l <pid>, 连续多次采样thread dump. 发现worker线程一直在read socket上等待.

　　大概率上来讲, 进程是被堵塞了于此了.

　　当然于此同时, 还发现了被动关闭, 哎呀妈呀, 问题还真不少.

**2 问题分析:**

代码很简洁, 也能很好地work, 那究竟哪个环节有疏忽呢？

由于问题都出现在Apache Httpclient 4.x上, 因此我们先看看网上是否也有类似的案例场景.

简单解读下某电商平台的SDK代码片段:

public HttpResponse get(String method, HashMap<String,String> parames) throws Exception{

　　　　String url = apiEntry + getParamStr(method, parames);

　　　　HttpClient client = new DefaultHttpClient();

　　　　HttpGet request = new HttpGet(url);

　　　　request.addHeader("User-Agent", DefaultUserAgent);

　　　　HttpResponse response = client.execute(request);

　　　　return response;

}

**3 socket阻塞**

　　参考了相关文章, 基本认可博文<<Apache HttpClient 没有设置time out导致应用长时间阻塞的问题>>中的说法: socket没有设置超时选项导致.

　　具体解读, 可以理解为服务端对端不回应, 或者因网络异常, 导致socket一直阻塞于读. Apache HttpClient默认设置的socket为SO\_TIMEOUT为0, 既无限等待.

　　结合之上的代码分析, 确实忽视了超时设定, 陷入了Apache HttpClient隐藏很深的坑中. ^\_^!

　　解决方案是：**为对应的httpclient设置超时即可.**

httpClient = new DefaultHttpClient();

httpClient.getParams().setIntParameter(CoreConnectionPNames.SO\_TIMEOUT, 2000);

httpClient.getParams().setIntParameter(CoreConnectionPNames.CONNECTION\_TIMEOUT, 2000);

这样阻塞的问题就可以暂时告一段落了.

**4 句柄泄露(CLOSE\_WAIT半连接关闭)**

　　同样参考了博文: <<HttpClient容易忽视的细节——连接关闭>>与<<HttpClient 与 Close\_Wait>>.

　　按apache httpclient的设计理念, 当http client 处于高并发时, 默认机制导致的CLOSE\_WAIT会影响服务的可用性.

　　对比之上的代码, 确实没有找到主动关闭连接的代码.

　　Apache HttpClient本身变动多, 接口又杂, 以至每个版本的解决方案又有所不同, 有些让人眼花缭乱.

　　比较一致的解决方式是，在处理完单个请求后, 调用如下清理代码:

httpmethod.releaseConnection();

client.getConnectionManager().shutdown();

　　但不管怎么说, Apache HttpClient 4.x的关闭机制, 还是值得大书特书的, 这边显得简略, 希望以后有机会能深入研究.

**5总结:**

　　这次算是事出有因, 问题出在了第三方的SDK上. 由此可见, 对第三方的SDK的使用, 还是得留个心眼. 实际上, 很多平台的SDK, 往往是实习生所编写的, 且没有经过严苛的测试, 所以质量难免有些参差不齐.

　　无论是SDK的使用者, 还是SDK的开发者, 都需要精益求精, 在质量(性能和稳定), 用户体验(易用/易理解)上做足功夫了, 这也是程序员的软实力之一.

对Apache HttpClient 4.x的阻塞陷阱做了介绍, 展现了其案例表现, 以及解决方案. 但缺少对Apache HttpClient深入分析, 以及使用方式推荐. 希望自己能有机会, 好好地整理一番.

