# HttpClient 4.3连接池参数配置及源码解读

<http://www.voidcn.com/article/p-wohpkcfq-gt.html>

目前所在公司使用HttpClient 4.3.3版本发送Rest请求，调用接口。最近出现了调用查询接口服务慢的生产问题，在排查整个调用链可能存在的问题时（从客户端发起Http请求->ESB->服务端处理请求，查询数据并返回），发现原本的HttpClient连接池中的一些参数配置可能存在问题，如defaultMaxPerRoute、一些timeout时间的设置等，虽不能确定是由于此连接池导致接口查询慢，但确实存在可优化的地方，故花时间做一些研究。本文主要涉及HttpClient连接池、请求的参数配置，使用及源码解读。

    以下是本文的目录大纲：

    一、HttpClient连接池、请求参数含义

    二、执行原理及源码解读

        1、创建HttpClient，执行request

        2、连接池管理

            2.1、连接池结构

            2.2、分配连接 & 建立连接

            2.3、回收连接 & 保持连接

            2.4、instream.close()、response.close()、httpclient.close()的区别

            2.5、过期和空闲连接清理

    三、如何设置合理的参数

# 一、HttpClient连接池、请求参数含义

import java.io.IOException;

import java.io.InputStream;

import java.io.InterruptedIOException;

import java.net.UnknownHostException;

import java.nio.charset.CodingErrorAction;

import javax.net.ssl.SSLException;

import org.apache.http.Consts;

import org.apache.http.HttpEntity;

import org.apache.http.HttpEntityEnclosingRequest;

import org.apache.http.HttpHost;

import org.apache.http.HttpRequest;

import org.apache.http.client.HttpRequestRetryHandler;

import org.apache.http.client.config.RequestConfig;

import org.apache.http.client.methods.CloseableHttpResponse;

import org.apache.http.client.methods.HttpGet;

import org.apache.http.client.protocol.HttpClientContext;

import org.apache.http.config.ConnectionConfig;

import org.apache.http.config.MessageConstraints;

import org.apache.http.config.SocketConfig;

import org.apache.http.conn.ConnectTimeoutException;

import org.apache.http.conn.routing.HttpRoute;

import org.apache.http.impl.client.CloseableHttpClient;

import org.apache.http.impl.client.DefaultHttpRequestRetryHandler;

import org.apache.http.impl.client.HttpClients;

import org.apache.http.impl.conn.PoolingHttpClientConnectionManager;

import org.apache.http.protocol.HttpContext;

public **class HttpClientParamTest** {

public static void main(String[] args) {

*/\*\**

*\* 创建连接管理器，并设置相关参数*

*\*/*

*//连接管理器，使用无惨构造*

PoolingHttpClientConnectionManager connManager

= new PoolingHttpClientConnectionManager();

*/\*\**

*\* 连接数相关设置*

*\*/*

*//最大连接数*

connManager.setMaxTotal(200);

*//默认的每个路由的最大连接数*

connManager.setDefaultMaxPerRoute(100);

*//设置到某个路由的最大连接数，会覆盖defaultMaxPerRoute*

connManager.setMaxPerRoute(new HttpRoute(new HttpHost("somehost", 80)), 150);

*/\*\**

*\* socket配置（默认配置 和 某个host的配置）*

*\*/*

SocketConfig socketConfig = SocketConfig.custom()

.setTcpNoDelay(true) *//是否立即发送数据，设置为true会关闭Socket缓冲，默认为false*

.setSoReuseAddress(true) *//是否可以在一个进程关闭Socket后，即使它还没有释放端口，其它进程还可以立即重用端口*

.setSoTimeout(500) *//接收数据的等待超时时间，单位ms*

.setSoLinger(60) *//关闭Socket时，要么发送完所有数据，要么等待60s后，就关闭连接，此时socket.close()是阻塞的*

.setSoKeepAlive(true) *//开启监视TCP连接是否有效*

.build();

connManager.setDefaultSocketConfig(socketConfig);

connManager.setSocketConfig(new HttpHost("somehost", 80), socketConfig);

*/\*\**

*\* HTTP connection相关配置（默认配置 和 某个host的配置）*

*\* 一般不修改HTTP connection相关配置，故不设置*

*\*/*

*//消息约束*

MessageConstraints messageConstraints = MessageConstraints.custom()

.setMaxHeaderCount(200)

.setMaxLineLength(2000)

.build();

*//Http connection相关配置*

ConnectionConfig connectionConfig = ConnectionConfig.custom()

.setMalformedInputAction(CodingErrorAction.IGNORE)

.setUnmappableInputAction(CodingErrorAction.IGNORE)

.setCharset(Consts.UTF\_8)

.setMessageConstraints(messageConstraints)

.build();

*//一般不修改HTTP connection相关配置，故不设置*

*//connManager.setDefaultConnectionConfig(connectionConfig);*

*//connManager.setConnectionConfig(new HttpHost("somehost", 80), ConnectionConfig.DEFAULT);*

*/\*\**

*\* request请求相关配置*

*\*/*

RequestConfig defaultRequestConfig = RequestConfig.custom()

.setConnectTimeout(2 \* 1000) *//连接超时时间*

.setSocketTimeout(2 \* 1000) *//读超时时间（等待数据超时时间）*

.setConnectionRequestTimeout(500) *//从池中获取连接超时时间*

.setStaleConnectionCheckEnabled(true)*//检查是否为陈旧的连接，默认为true，类似testOnBorrow*

.build();

*/\*\**

*\* 重试处理*

*\* 默认是重试3次*

*\*/*

*//禁用重试(参数：retryCount、requestSentRetryEnabled)*

HttpRequestRetryHandler requestRetryHandler = new DefaultHttpRequestRetryHandler(0, false);

*//自定义重试策略*

HttpRequestRetryHandler myRetryHandler = new HttpRequestRetryHandler() {

public boolean retryRequest(IOException exception, int executionCount, HttpContext context) {

*//Do not retry if over max retry count*

if (executionCount >= 3) {

return false;

}

*//Timeout*

if (exception instanceof InterruptedIOException) {

return false;

}

*//Unknown host*

if (exception instanceof UnknownHostException) {

return false;

}

*//Connection refused*

if (exception instanceof ConnectTimeoutException) {

return false;

}

*//SSL handshake exception*

if (exception instanceof SSLException) {

return false;

}

HttpClientContext clientContext = HttpClientContext.adapt(context);

HttpRequest request = clientContext.getRequest();

boolean idempotent = !(request instanceof HttpEntityEnclosingRequest);

*//Retry if the request is considered idempotent*

*//如果请求类型不是HttpEntityEnclosingRequest，被认为是幂等的，那么就重试*

*//HttpEntityEnclosingRequest指的是有请求体的request，比HttpRequest多一个Entity属性*

*//而常用的GET请求是没有请求体的，POST、PUT都是有请求体的*

*//Rest一般用GET请求获取数据，故幂等，POST用于新增数据，故不幂等*

if (idempotent) {

return true;

}

return false;

}

};

*/\*\**

*\* 创建httpClient*

*\*/*

CloseableHttpClient httpclient = HttpClients.custom()

.setConnectionManager(connManager) *//连接管理器*

.setProxy(new HttpHost("myproxy", 8080)) *//设置代理*

.setDefaultRequestConfig(defaultRequestConfig) *//默认请求配置*

.setRetryHandler(myRetryHandler) *//重试策略*

.build();

*//创建一个Get请求，并重新设置请求参数，覆盖默认*

HttpGet httpget = new HttpGet("http://www.somehost.com/");

RequestConfig requestConfig = RequestConfig.copy(defaultRequestConfig)

.setSocketTimeout(5000)

.setConnectTimeout(5000)

.setConnectionRequestTimeout(5000)

.setProxy(new HttpHost("myotherproxy", 8080))

.build();

httpget.setConfig(requestConfig);

CloseableHttpResponse response = null;

try {

*//执行请求*

response = httpclient.execute(httpget);

HttpEntity entity = response.getEntity();

*// If the response does not enclose an entity, there is no need*

*// to bother about connection release*

if (entity != null) {

InputStream instream = entity.getContent();

try {

instream.read();

*// do something useful with the response*

}

catch (IOException ex) {

*// In case of an IOException the connection will be released*

*// back to the connection manager automatically*

throw ex;

}

finally {

*// Closing the input stream will trigger connection release*

*// 释放连接回到连接池*

instream.close();

}

}

}

catch (Exception e) {

e.printStackTrace();

}

finally{

if(response != null){

try {

*//关闭连接(如果已经释放连接回连接池，则什么也不做)*

response.close();

} catch (IOException e) {

e.printStackTrace();

}

}

if(httpclient != null){

try {

*//关闭连接管理器，并会关闭其管理的连接*

httpclient.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

}

    上面的代码参考httpClient 4.3.x的官方样例，其实官方样例中可配置的更多，我只将一些觉得平时常用的摘了出来，其实我们在实际使用中也是使用默认的 socketConfig 和 connectionConfig。具体参数含义请看注释。

    个人感觉在实际应用中连接数相关配置（如maxTotal、maxPerRoute），还有请求相关的超时时间设置（如connectionTimeout、socketTimeout、connectionRequestTimeout）是比较重要的。

    连接数配置有问题就可能产生总的 连接数不够 或者 到某个路由的连接数太小 的问题，我们公司一些项目总连接数800，而defaultMaxPerRoute仅为20，这样导致真正需要比较多连接数，访问量比较大的路由也仅能从连接池中获取最大20个连接，应该在默认的基础上，针对访问量大的路由单独设置。

    连接超时时间，读超时时间，从池中获取连接的超时时间如果不设置或者设置的太大，可能导致当业务高峰时，服务端响应较慢 或 连接池中确实没有空闲连接时，不能够及时将timeout异常抛出来，导致等待读取数据的，或者等待从池中获取连接的越积越多，像滚雪球一样，导致相关业务都开始变得缓慢，而如果配置合理的超时时间就可以及时抛出异常，发现问题。

    后面会尽量去阐述这些重要参数的原理以及如何配置一个合适的值。

# 二、执行原理及源码解读

**1、创建HttpClient，执行request**

*/\*\**

*\* 创建httpClient*

*\*/*

CloseableHttpClient httpclient = HttpClients.custom()

.setConnectionManager(connManager) *//连接管理器*

.setDefaultRequestConfig(defaultRequestConfig) *//默认请求配置*

.setRetryHandler(myRetryHandler) *//重试策略*

.build();

    创建HttpClient的过程就是在设置了“连接管理器”、“请求相关配置”、“重试策略”后，调用 HttpClientBuilder.build()。

    build()方法会根据设置的属性不同，创建不同的Executor执行器，如设置了retryHandler就会 new RetryExec(execChain, retryHandler)，相当于retry Executor。当然有些Executor是必须创建的，如MainClientExec、ProtocolExec。最后new InternalHttpClient(execChain, connManager, routePlanner …)并返回。

CloseableHttpResponse httpResponse = httpClient.execute(httpUriRequest);

    HttpClient使用了责任链模式，所有Executor都实现了ClientExecChain接口的execute()方法，每个Executor都持有下一个要执行的Executor的引用，这样就会形成一个Executor的执行链条，请求在这个链条上传递。按照上面的方式构造的httpClient形成的执行链条为：

HttpRequestExecutor *//发送请求报文，并接收响应信息*

MainClientExec(requestExec, connManager, ...) *//main Executor，负责连接管理相关*

ProtocolExec(execChain, httpprocessor) *//HTTP协议封装*

RetryExec(execChain, retryHandler) *//重试策略*

RedirectExec(execChain, routePlanner, redirectStrategy) *//重定向*

    请求执行是按照从下到上的顺序（即每个下面的Executor都持有上面一个Executor的引用），每一个执行器都会负责请求过程中的一部分工作，最终返回response。

**2、连接池管理**

**2.1、连接池结构**

连接池结构图如下:

6f3717d34737_thumb2

PoolEntry<HttpRoute, ManagedHttpClientConnection>  --  连接池中的实体

包含ManagedHttpClientConnection连接；

连接的route路由信息；

以及连接存活时间相隔信息，如created（创建时间），updated（更新时间，释放连接回连接池时会更新），validUnit（用于初始化expiry过期时间，规则是如果timeToLive>0，则为created+timeToLive，否则为Long.MAX\_VALUE），expiry（过期时间，人为规定的连接池可以保有连接的时间，除了初始化时等于validUnit，每次释放连接时也会更新，但是从newExpiry和validUnit取最小值）。timeToLive是在构造连接池时指定的连接存活时间，默认构造的timeToLive=-1。

ManagedHttpClientConnection是httpClient连接，真正建立连接后，其会bind绑定一个socket，用于传输HTTP报文。

LinkedList<PoolEntry>  available  --  存放可用连接

使用完后所有可重用的连接回被放到available链表头部，之后再获取连接时优先从available链表头部迭代可用的连接。

之所以使用LinkedList是利用了其队列的特性，即可以在队首和队尾分别插入、删除。入available链表时都是addFirst()放入头部，获取时都是从头部依次迭代可用的连接，这样可以获取到最新放入链表的连接，其离过期时间更远（这种策略可以尽量保证获取到的连接没有过期，而从队尾获取连接是可以做到在连接过期前尽量使用，但获取到过期连接的风险就大了），删除available链表中连接时是从队尾开始，即先删除最可能快要过期的连接。

HashSet<PoolEntry>  leased  --  存放被租用的连接

所有正在被使用的连接存放的集合，只涉及 add() 和 remove() 操作。

maxTotal限制的是外层httpConnPool中leased集合和available队列的总和的大小，leased和available的大小没有单独限制。

LinkedList<PoolEntryFuture>  pending  --  存放等待获取连接的线程的Future

当从池中获取连接时，如果available链表没有现成可用的连接，且当前路由或连接池已经达到了最大数量的限制，也不能创建连接了，此时不会阻塞整个连接池，而是将当前线程用于获取连接的Future放入pending链表的末尾，之后当前线程调用await()，释放持有的锁，并等待被唤醒。

当有连接被release()释放回连接池时，会从pending链表头获取future，并唤醒其线程继续获取连接，做到了先进先出。

routeToPool  --  每个路由对应的pool

也有针对当前路由的available、leased、pending集合，与整个池的隔离。

maxPerRoute限制的是routeToPool中leased集合和available队列的总和的大小。

**2.2、分配连接 & 建立连接**

**分配连接**

分配连接指的是从连接池获取可用的PoolEntry，大致过程为：

1、获取route对应连接池routeToPool中可用的连接，有则返回该连接，若没有则转入下一步；

2、若routeToPool和外层HttpConnPool连接池均还有可用的空间，则新建连接，并将该连接作为可用连接返回，否则进行下一步；

3、挂起当前线程，将当前线程的Future放入pending队列，等待后续唤醒执行；

整个分配连接的过程采用了异步操作，只在前两步时锁住连接池，一旦发现无法获取连接则释放锁，等待后续继续获取连接。

**建立连接**

当分配到PoolEntry连接实体后，会调用establishRoute()，建立socket连接并与conn绑定。

**2.3、回收连接 & 保持连接**

**回收连接**

连接用完之后连接池需要进行回收（AbstractConnPool#release()），具体流程如下：  
1、若当前连接标记为重用，则将该连接从routeToPool中的leased集合删除，并添加至available队首，同样的将该请求从外层httpConnPool的leased集合删除，并添加至其available队首。同时唤醒该routeToPool的pending队列的第一个PoolEntryFuture，将其从pending队列删除，并将其从外层httpConnPool的pending队列中删除。  
2、若连接没有标记为重用，则分别从routeToPool和外层httpConnPool中删除该连接，并关闭该连接。

**保持连接**

MainClientExec#execute()是负责连接管理的，在执行完后续调用链，并得到response后，会调用保持连接的逻辑，如下：

*// The connection is in or can be brought to a re-usable state.*

*// 根据response头中的信息判断是否保持连接*

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

*// Set the idle duration of this connection*

*// 根据response头中的keep-alive中的timeout属性，得到连接可以保持的时间（ms）*

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

if (this.log.isDebugEnabled()) {

final String s;

if (duration > 0) {

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

} else {

s = "indefinitely";

}

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

}

*//设置连接保持时间，最终是调用 PoolEntry#updateExpiry*

connHolder.setValidFor(duration, TimeUnit.MILLISECONDS);

connHolder.markReusable(); *//设置连接reuse=true*

}

else {

connHolder.markNonReusable();

}

连接是否保持

客户端如果希望保持长连接，应该在发起请求时告诉服务器希望服务器保持长连接（http 1.0设置connection字段为keep-alive，http 1.1字段默认保持）。根据服务器的响应来确定是否保持长连接，判断原则如下：

1、检查返回response报文头的Transfer-Encoding字段，若该字段值存在且不为chunked，则连接不保持，直接关闭。其他情况进入下一步；  
2、检查返回的response报文头的Content-Length字段，若该字段值为空或者格式不正确（多个长度，值不是整数）或者小于0，则连接不保持，直接关闭。其他情况进入下一步  
3、检查返回的response报文头的connection字段（若该字段不存在，则为Proxy-Connection字段）值，如果字段存在，若字段值为close 则连接不保持，直接关闭，若字段值为keep-alive则连接标记为保持。如果这俩字段都不存在，则http 1.1版本默认为保持，将连接标记为保持， 1.0版本默认为连接不保持，直接关闭。

连接保持时间

连接交还至连接池时，若连接标记为保持reuse=true，则将由连接管理器保持一段时间；若连接没有标记为保持，则直接从连接池中删除并关闭entry。  
连接保持时，会更新PoolEntry的expiry到期时间，计算逻辑为：  
1、如果response头中的keep-alive字段中timeout属性值存在且为正值：newExpiry = 连接归还至连接池时间System.currentTimeMillis() + timeout；  
2、如timeout属性值不存在或为负值：newExpiry = Long.MAX\_VALUE（无穷）  
3、最后会和PoolEntry原本的expiry到期时间比较，选出一个最小值作为新的到期时间。

**2.4、instream.close()、response.close()、httpclient.close()的区别**

*/\*\**

*\* This example demonstrates the recommended way of using API to make sure*

*\* the underlying connection gets released back to the connection manager.*

*\*/*

public class ClientConnectionRelease {

public final static void main(String[] args) throws Exception {

CloseableHttpClient httpclient = HttpClients.createDefault();

try {

HttpGet httpget = new HttpGet("http://localhost/");

System.out.println("Executing request " + httpget.getRequestLine());

CloseableHttpResponse response = httpclient.execute(httpget);

try {

System.out.println("----------------------------------------");

System.out.println(response.getStatusLine());

*// Get hold of the response entity*

HttpEntity entity = response.getEntity();

*// If the response does not enclose an entity, there is no need*

*// to bother about connection release*

if (entity != null) {

InputStream instream = entity.getContent();

try {

instream.read();

*// do something useful with the response*

} catch (IOException ex) {

*// In case of an IOException the connection will be released*

*// back to the connection manager automatically*

throw ex;

} finally {

*// Closing the input stream will trigger connection release*

instream.close();

}

}

} finally {

response.close();

}

} finally {

httpclient.close();

}

}

}

在HttpClient Manual connection release的例子中可以看到，从内层依次调用的是instream.close()、response.close()、httpClient.close()，那么它们有什么区别呢？

**instream.close()**

在主动操作输入流，或者调用EntityUtils.toString(httpResponse.getEntity())时会调用instream.read()、instream.close()等方法。instream的实现类为org.apache.http.conn.EofSensorInputStream。

在每次通过instream.read()读取数据流后，都会判断流是否读取结束

@Override

public int read(final byte[] b, final int off, final int len) throws IOException {

int l = -1;

if (isReadAllowed()) {

try {

l = wrappedStream.read(b, off, len);

checkEOF(l);

} catch (final IOException ex) {

checkAbort();

throw ex;

}

}

return l;

}

在EofSensorInputStream#checkEOF()方法中如果eof=-1，流已经读完，如果连接可重用，就会尝试释放连接，否则关闭连接。

protected void checkEOF(final int eof) throws IOException {

if ((wrappedStream != null) && (eof < 0)) {

try {

boolean scws = true; *// should close wrapped stream?*

if (eofWatcher != null) {

scws = eofWatcher.eofDetected(wrappedStream);

}

if (scws) {

wrappedStream.close();

}

} finally {

wrappedStream = null;

}

}

}

ResponseEntityWrapper#eofDetected

public boolean eofDetected(final InputStream wrapped) throws IOException {

try {

*// there may be some cleanup required, such as*

*// reading trailers after the response body:*

wrapped.close();

releaseConnection(); *//释放连接 或 关闭连接*

} finally {

cleanup();

}

return false;

}

ConnectionHolder#releaseConnection

public void releaseConnection() {

synchronized (this.managedConn) {

*//如果连接已经释放，直接返回*

if (this.released) {

return;

}

this.released = true;

*//连接可重用，释放回连接池*

if (this.reusable) {

this.manager.releaseConnection(this.managedConn,

this.state, this.validDuration, this.tunit);

}

*//不可重用，关闭连接*

else {

try {

this.managedConn.close();

log.debug("Connection discarded");

} catch (final IOException ex) {

if (this.log.isDebugEnabled()) {

this.log.debug(ex.getMessage(), ex);

}

} finally {

this.manager.releaseConnection(

this.managedConn, null, 0, TimeUnit.MILLISECONDS);

}

}

}

}

如果没有instream.read()读取数据，在instream.close()时会调用EofSensorInputStream#checkClose()，也会有类似上面的逻辑。

所以就如官方例子注释的一样，在正常操作输入流后，会释放连接。

**response.close()**

最终是调用ConnectionHolder#abortConnection()

public void abortConnection() {

synchronized (this.managedConn) {

*//如果连接已经释放，直接返回*

if (this.released) {

return;

}

this.released = true;

try {

*//关闭连接*

this.managedConn.shutdown();

log.debug("Connection discarded");

} catch (final IOException ex) {

if (this.log.isDebugEnabled()) {

this.log.debug(ex.getMessage(), ex);

}

} finally {

this.manager.releaseConnection(

this.managedConn, null, 0, TimeUnit.MILLISECONDS);

}

}

}

所以，如果在调用response.close()之前，没有读取过输入流，也没有关闭输入流，那么连接没有被释放，released=false，就会关闭连接。

**httpClient.close()**

最终调用的是InternalHttpClient#close()，会关闭整个连接管理器，并关闭连接池中所有连接。

public void close() {

this.connManager.shutdown();

if (this.closeables != null) {

for (final Closeable closeable: this.closeables) {

try {

closeable.close();

} catch (final IOException ex) {

this.log.error(ex.getMessage(), ex);

}

}

}

}

**总结：**

1、使用连接池时，要正确释放连接需要通过读取输入流 或者 instream.close()方式；

2、如果已经释放连接，response.close()直接返回，否则会关闭连接；

3、httpClient.close()会关闭连接管理器，并关闭其中所有连接，谨慎使用。

**2.5、过期和空闲连接清理**

**在连接池保持连接的这段时间，可能出现两种导致连接过期或失效的情况：**

1、连接保持时间到期

每个连接对象PoolEntry都有expiry到期时间，在创建和释放归还连接是都会为expiry到期时间赋值，在连接池保持连接的这段时间，连接已经到了过期时间（注意，这个过期时间是为了管理连接所设定的，并不是指的TCP连接真的不能使用了）。

对于这种情况，在每次从连接池获取连接时，都会从routeToPool的available队列获取Entry并检测此时Entry是否已关闭或者已过期，若是则关闭并分别从routeToPool、httpConnPool的available队列移除该Entry，之后再次尝试获取连接。代码如下

*/\*\*AbstractConnPool#getPoolEntryBlocking()\*/*

for (;;) {

*//从availabe链表头迭代查找符合state的entry*

entry = pool.getFree(state);

*//找不到entry，跳出*

if (entry == null) {

break;

}

*//如果entry已关闭或已过期，关闭entry，并从routeToPool、httpConnPool的available队列移除*

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

entry.close();

this.available.remove(entry);

pool.free(entry, false);

}

else { *//找到可用连接*

break;

}

}

2、底层连接已被关闭

在连接池保持连接的时候，可能会出现连接已经被服务端关闭的情况，而此时连接的客户端并没有阻塞着去接收服务端的数据，所以客户端不知道连接已关闭，无法关闭自身的socket。

对于这种情况，在从连接池获取可用连接时无法知晓，在获取到可用连接后，如果连接是打开的，会有判断连接是否陈旧的逻辑，如下

*/\*\*MainClientExec#execute()\*/*

if (config.isStaleConnectionCheckEnabled()) {

*// validate connection*

if (managedConn.isOpen()) {

this.log.debug("Stale connection check");

if (managedConn.isStale()) {

this.log.debug("Stale connection detected");

managedConn.close();

}

}

}

isOpen()会通过连接的状态判断连接是否是open状态；

isStale()会通过socket输入流尝试读取数据，在读取前暂时将soTimeout设置为1ms，如果读取到的字节数小于0，即已经读到了输入流的末尾，或者发生了IOException，可能连接已经关闭，那么isStale()返回true，需要关闭连接；如果读到的字节数大于0，或者发生了SocketTimeoutException，可能是读超时，isStale()返回false，连接还可用。

*/\*\*BHttpConnectionBase#isStale()\*/*

public boolean isStale() {

if (!isOpen()) {

return true;

}

try {

final int bytesRead = fillInputBuffer(1);

return bytesRead < 0;

} catch (final SocketTimeoutException ex) {

return false;

} catch (final IOException ex) {

return true;

}

}

如果在整个判断过程中发现连接是陈旧的，就会关闭连接，那么这个从连接池获取的连接就是不可用的，后面的代码逻辑里会重建当前PoolEntry的socket连接，继续后续请求逻辑。

**后台监控线程检查连接**

上述过程是在从连接池获取连接后，检查连接是否可用，如不可用需重新建立socket连接，建立连接的过程是比较耗时的，可能导致性能问题，也失去了连接池的意义，针对这种情况，HttpClient采取一个策略，通过一个后台的监控线程定时的去检查连接池中连接是否还“新鲜”，如果过期了，或者空闲了一定时间则就将其从连接池里删除掉。

ClientConnectionManager提供了 closeExpiredConnections()和closeIdleConnections()两个方法，关闭过期或空闲了一段时间的连接，并从连接池删除。

closeExpiredConnections()  
该方法关闭超过连接保持时间的连接，并从池中移除。

closeIdleConnections(timeout,tunit)

该方法关闭空闲时间超过timeout的连接，空闲时间从交还给连接池时开始，不管是否已过期，超过空闲时间则关闭。

下面是httpClient官方给出的清理过期、空闲连接的例子

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);

*// Close expired connections*

connMgr.closeExpiredConnections();

*// Optionally, close connections*

*// that have been idle longer than 30 sec*

connMgr.closeIdleConnections(30, TimeUnit.SECONDS);

}

}

} catch (InterruptedException ex) {

*// terminate*

}

}

public void shutdown() {

shutdown = true;

synchronized (this) {

notifyAll();

}

}

}

# 三、如何设置合理的参数

关于设置合理的参数，这个说起来真的不是一个简单的话题，需要考虑的方面也听到，是需要一定经验的，这里先简单的说一下自己的理解，欢迎各位批评指教。

这里主要涉及两部分参数：连接数相关参数、超时时间相关参数

**1、连接数相关参数**

根据“利尔特法则”可以得到简单的公式：

bb1dddfc6ee63

简单地说，利特尔法则解释了这三种变量的关系：L—系统里的请求数量、λ—请求到达的速率、W—每个请求的处理时间。例如，如果每秒10个请求到达，处理一个请求需要1秒，那么系统在每个时刻都有10个请求在处理。如果处理每个请求的时间翻倍，那么系统每时刻需要处理的请求数也翻倍为20，因此需要20个线程。连接池的大小可以参考 L。

qps指标可以作为“λ—请求到达的速率”，由于httpClient是作为http客户端，故需要通过一些监控手段得到服务端集群访问量较高时的qps，如客户端集群为4台，服务端集群为2台，监控到每台服务端机器的qps为100，如果每个请求处理时间为1秒，那么2台服务端每个时刻总共有 100 \* 2 \* 1s = 200 个请求访问，平均到4台客户端机器，每台要负责50，即每台客户端的连接池大小可以设置为50。

当然实际的情况是更复杂的，上面的请求平均处理时间1秒只是一种业务的，实际情况的业务情况更多，评估请求平均处理时间更复杂。所以在设置连接数后，最好通过比较充分性能测试验证是否可以满足要求。

还有一些Linux系统级的配置需要考虑，如单个进程能够打开的最大文件描述符数量open files默认为1024，每个与服务端建立的连接都需要占用一个文件描述符，如果open files值太小会影响建立连接。

还要注意，连接数主要包含maxTotal-连接总数、maxPerRoute-路由最大连接数，尤其是maxPerRoute默认值为2，很小，设置不好的话即使maxTotal再大也无法充分利用连接池。

**2、超时时间相关参数**

connectTimeout  --  连接超时时间

根据网络情况，内网、外网等，可设置连接超时时间为2秒，具体根据业务调整

socketTimeout  --  读超时时间（等待数据超时时间）

需要根据具体请求的业务而定，如请求的API接口从接到请求到返回数据的平均处理时间为1秒，那么读超时时间可以设置为2秒，考虑并发量较大的情况，也可以通过性能测试得到一个相对靠谱的值。

socketTimeout有默认值，也可以针对每个请求单独设置。

connectionRequestTimeout  --  从池中获取连接超时时间

建议设置500ms即可，不要设置太大，这样可以使连接池连接不够时不用等待太久去获取连接，不要让大量请求堆积在获取连接处，尽快抛出异常，发现问题。