

04 | 连接池：别让连接池帮了倒忙

2020-03-14 朱晔

Java 业务开发常见错误 100 例

[进入课程 >](#)




讲述：王少泽

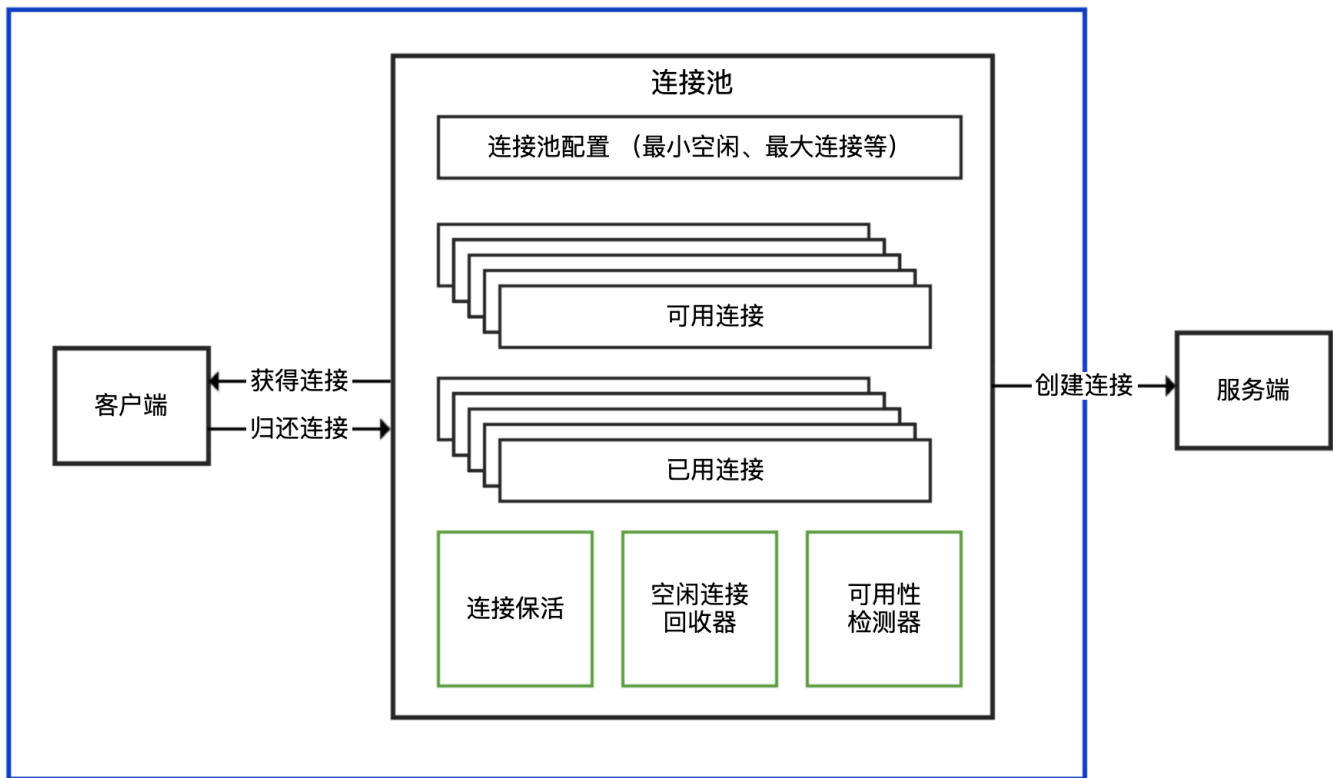
时长 23:41 大小 21.70M



你好，我是朱晔。今天，我们来聊聊使用连接池需要注意的问题。

在上一讲，我们学习了使用线程池需要注意的问题。今天，我再与你说说另一种很重要的池化技术，即连接池。

我先和你说说连接池的结构。连接池一般对外提供获得连接、归还连接的接口给客户端使用，并暴露最小空闲连接数、最大连接数等可配置参数，在内部则实现连接建立、连接心跳保持、连接管理、空闲连接回收、连接可用性检测等功能。连接池的结构示意图，如  示：



业务项目中经常会用到的连接池，主要是数据库连接池、Redis 连接池和 HTTP 连接池。所以，今天我就以这三种连接池为例，和你聊聊使用和配置连接池容易出错的地方。

注意鉴别客户端 SDK 是否基于连接池

在使用三方客户端进行网络通信时，我们首先要确定客户端 SDK 是否是基于连接池技术实现的。我们知道，TCP 是面向连接的基于字节流的协议：

面向连接，意味着连接需要先创建再使用，创建连接的三次握手有一定开销；

基于字节流，意味着字节是发送数据的最小单元，TCP 协议本身无法区分哪几个字节是完整的消息体，也无法感知是否有多个客户端在使用同一个 TCP 连接，TCP 只是一个读写数据的管道。

如果客户端 SDK 没有使用连接池，而直接是 TCP 连接，那么就需要考虑每次建立 TCP 连接的开销，**并且因为 TCP 基于字节流，在多线程的情况下对同一连接进行复用，可能会产生线程安全问题。**

我们先看一下涉及 TCP 连接的客户端 SDK，对外提供 API 的三种方式。在面对各种三方客户端的时候，只有先识别出其属于哪一种，才能理清楚使用方式。

连接池和连接分离的 API：有一个 XXXPool 类负责连接池实现，先从其获得连接 XXXConnection，然后用获得的连接进行服务端请求，完成后使用者需要归还连接。通常，XXXPool 是线程安全的，可以并发获取和归还连接，而 XXXConnection 是非线程安全的。对应到连接池的结构示意图中，XXXPool 就是右边连接池那个框，左边的客户端是我们自己的代码。

内部带有连接池的 API：对外提供一个 XXXClient 类，通过这个类可以直接进行服务端请求；这个类内部维护了连接池，SDK 使用者无需考虑连接的获取和归还问题。一般而言，XXXClient 是线程安全的。对应到连接池的结构示意图中，整个 API 就是蓝色框包裹的部分。

非连接池的 API：一般命名为 XXXConnection，以区分其是基于连接池还是单连接的，而不建议命名为 XXXClient 或直接是 XXX。直接连接方式的 API 基于单一连接，每次使用都需要创建和断开连接，性能一般，且通常不是线程安全的。对应到连接池的结构示意图中，这种形式相当于没有右边连接池那个框，客户端直接连接服务端创建连接。

虽然上面提到了 SDK 一般的命名习惯，但不排除有一些客户端特立独行，因此在使用三方 SDK 时，一定要先查看官方文档了解其最佳实践，或是在类似 Stackoverflow 的网站搜索 XXX threadsafe/singleton 字样看看大家的回复，也可以一层一层往下看源码，直到定位到原始 Socket 来判断 Socket 和客户端 API 的对应关系。

明确了 SDK 连接池的实现方式后，我们就大概知道了使用 SDK 的最佳实践：

如果是分离方式，那么连接池本身一般是线程安全的，可以复用。每次使用需要从连接池获取连接，使用后归还，归还的工作由使用者负责。

如果是内置连接池，SDK 会负责连接的获取和归还，使用的时候直接复用客户端。

如果 SDK 没有实现连接池（大多数中间件、数据库的客户端 SDK 都会支持连接池），那通常不是线程安全的，而且短连接的方式性能不会很高，使用的时候需要考虑是否自己封装一个连接池。

接下来，我就以 Java 中用于操作 Redis 最常见的库 Jedis 为例，从源码角度分析下 Jedis 类到底属于哪种类型的 API，直接在多线程环境下复用连接会产生什么问题，以及如何用最佳实践来修复这个问题。

首先，向 Redis 初始化 2 组数据，Key=a、Value=1，Key=b、Value=2：

[复制代码](#)

```
1 @PostConstruct
2 public void init() {
3     try (Jedis jedis = new Jedis("127.0.0.1", 6379)) {
4         Assert.isTrue("OK".equals(jedis.set("a", "1")), "set a = 1 return OK")
5         Assert.isTrue("OK".equals(jedis.set("b", "2")), "set b = 2 return OK")
6     }
7 }
```

然后，启动两个线程，共享操作同一个 Jedis 实例，每一个线程循环 1000 次，分别读取 Key 为 a 和 b 的 Value，判断是否分别为 1 和 2：

[复制代码](#)

```
1 Jedis jedis = new Jedis("127.0.0.1", 6379);
2 new Thread(() -> {
3     for (int i = 0; i < 1000; i++) {
4         String result = jedis.get("a");
5         if (!result.equals("1")) {
6             log.warn("Expect a to be 1 but found {}", result);
7             return;
8         }
9     }
10 }).start();
11 new Thread(() -> {
12     for (int i = 0; i < 1000; i++) {
13         String result = jedis.get("b");
14         if (!result.equals("2")) {
15             log.warn("Expect b to be 2 but found {}", result);
16             return;
17         }
18     }
19 }).start();
20 TimeUnit.SECONDS.sleep(5);
```

执行程序多次，可以看到日志中出现了各种奇怪的异常信息，有的是读取 Key 为 b 的 Value 读取到了 1，有的是流非正常结束，还有的是连接关闭异常：

[复制代码](#)

```
1 //错误1
2 [14:56:19.069] [Thread-28] [WARN ] [.t.c.c.redis.JedisMisreuseController:45 ]
3 //错误2
4 redis.clients.jedis.exceptions.JedisConnectionException: Unexpected end of str
5     at redis.clients.jedis.util.RedisInputStream.ensureFill(RedisInputStream.java:
6     at redis.clients.jedis.util.RedisInputStream.readLine(RedisInputStream.java:!
```

```

7   at redis.clients.jedis.Protocol.processError(Protocol.java:114)
8   at redis.clients.jedis.Protocol.process(Protocol.java:166)
9   at redis.clients.jedis.Protocol.read(Protocol.java:220)
10  at redis.clients.jedis.Connection.readProtocolWithCheckingBroken(Connection.:
11  at redis.clients.jedis.Connection.getBinaryBulkReply(Connection.java:255)
12  at redis.clients.jedis.Connection.getBulkReply(Connection.java:245)
13  at redis.clients.jedis.Jedis.get(Jedis.java:181)
14  at org.geekbang.time.commonmistakes.connectionpool.redis.JedisMisreuseContro
15  at java.lang.Thread.run(Thread.java:748)
16  //错误3
17  java.io.IOException: Socket Closed
18  at java.net.AbstractPlainSocketImpl.getOutputStream(AbstractPlainSocketImpl.:
19  at java.net.Socket$3.run(Socket.java:954)
20  at java.net.Socket$3.run(Socket.java:952)
21  at java.security.AccessController.doPrivileged(Native Method)
22  at java.net.Socket.getOutputStream(Socket.java:951)
23  at redis.clients.jedis.Connection.connect(Connection.java:200)
24  ... 7 more

```

让我们分析一下 Jedis 类的源码，搞清楚其中缘由吧。

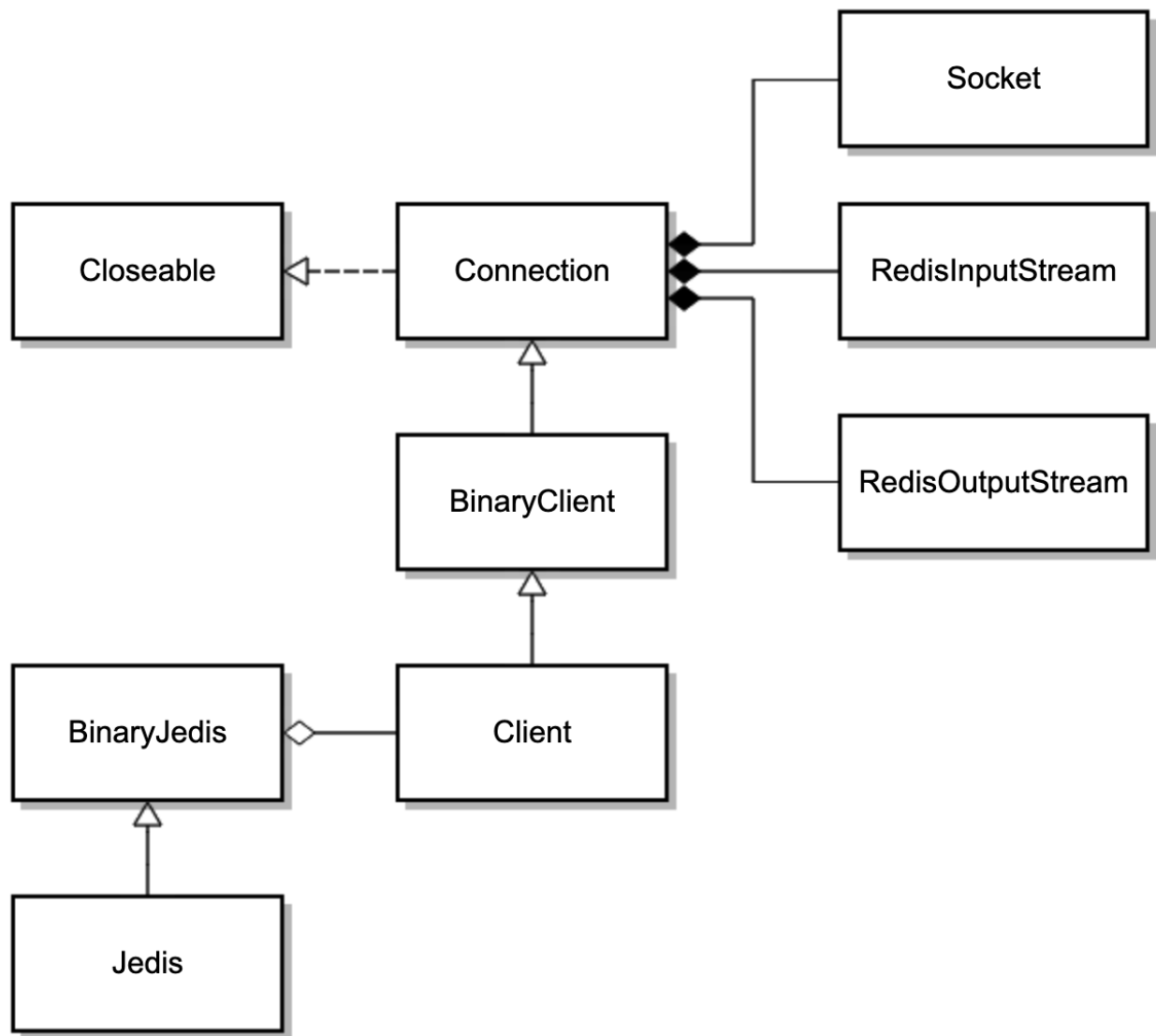
 复制代码

```

1  public class Jedis extends BinaryJedis implements JedisCommands, MultiKeyComm
2      AdvancedJedisCommands, ScriptingCommands, BasicCommands, ClusterCommands, !
3  }
4  public class BinaryJedis implements BasicCommands, BinaryJedisCommands, MultiK
5      AdvancedBinaryJedisCommands, BinaryScriptingCommands, Closeable {
6      protected Client client = null;
7      ...
8  }
9
10 public class Client extends BinaryClient implements Commands {
11 }
12 public class BinaryClient extends Connection {
13 }
14 public class Connection implements Closeable {
15     private Socket socket;
16     private RedisOutputStream outputStream;
17     private RedisInputStream inputStream;
18 }

```

可以看到，Jedis 继承了 BinaryJedis，BinaryJedis 中保存了单个 Client 的实例，Client 最终继承了 Connection，Connection 中保存了单个 Socket 的实例，和 Socket 对应的两个读写流。因此，一个 Jedis 对应一个 Socket 连接。类图如下：



BinaryClient 封装了各种 Redis 命令，其最终会调用基类 Connection 的方法，使用 Protocol 类发送命令。看一下 Protocol 类的 sendCommand 方法的源码，可以发现其发送命令时是直接操作 RedisOutputStream 写入字节。

我们在多线程环境下复用 Jedis 对象，其实就是在复用 RedisOutputStream。**如果多个线程在执行操作，那么既无法确保整条命令以一个原子操作写入 Socket，也无法确保写入后、读取前没有其他数据写到远端：**

[复制代码](#)

```
1 private static void sendCommand(final RedisOutputStream os, final byte[] command,
2     final byte[]... args) {
3     try {
4         os.write(ASTERISK_BYTE);
5         os.writeIntCrLf(args.length + 1);
6         os.write(DOLLAR_BYTE);
7         os.writeIntCrLf(command.length);
```

```

8      os.write(command);
9      os.writeCrLf();
10
11
12      for (final byte[] arg : args) {
13          os.write(DOLLAR_BYTE);
14          os.writeIntCrLf(arg.length);
15          os.write(arg);
16          os.writeCrLf();
17      }
18  } catch (IOException e) {
19      throw new JedisConnectionException(e);
20  }
21 }

```

看到这里我们也可以理解了，为啥多线程情况下使用 Jedis 对象操作 Redis 会出现各种奇怪的问题。

比如，写操作互相干扰，多条命令相互穿插的话，必然不是合法的 Redis 命令，那么 Redis 会关闭客户端连接，导致连接断开；又比如，线程 1 和 2 先后写入了 get a 和 get b 操作的请求，Redis 也返回了值 1 和 2，但是线程 2 先读取了数据 1 就会出现数据错乱的问题。

修复方式是，使用 Jedis 提供的另一个线程安全的类 JedisPool 来获得 Jedis 的实例。JedisPool 可以声明为 static 在多个线程之间共享，扮演连接池的角色。使用时，按需使用 try-with-resources 模式从 JedisPool 获得和归还 Jedis 实例。

 复制代码

```

1  private static JedisPool jedisPool = new JedisPool("127.0.0.1", 6379);
2
3  new Thread(() -> {
4      try (Jedis jedis = jedisPool.getResource()) {
5          for (int i = 0; i < 1000; i++) {
6              String result = jedis.get("a");
7              if (!result.equals("1")) {
8                  log.warn("Expect a to be 1 but found {}", result);
9                  return;
10             }
11         }
12     }
13 }).start();
14 new Thread(() -> {
15     try (Jedis jedis = jedisPool.getResource()) {


```

```

16         for (int i = 0; i < 1000; i++) {
17             String result = jedis.get("b");
18             if (!result.equals("2")) {
19                 log.warn("Expect b to be 2 but found {}", result);
20                 return;
21             }
22         }
23     }
24 }).start();

```

这样修复后，代码不再有线程安全问题了。此外，我们最好通过 shutdownhook，在程序退出之前关闭 JedisPool：


 复制代码

```

1 @PostConstruct
2 public void init() {
3     Runtime.getRuntime().addShutdownHook(new Thread(() -> {
4         jedisPool.close();
5     }));
6 }

```

看一下 Jedis 类 close 方法的实现可以发现，如果 Jedis 是从连接池获取的话，那么 close 方法会调用连接池的 return 方法归还连接：

 复制代码

```

1 public class Jedis extends BinaryJedis implements JedisCommands, MultiKeyCommam
2     AdvancedJedisCommands, ScriptingCommands, BasicCommands, ClusterCommands, !
3     protected JedisPoolAbstract dataSource = null;
4
5
6 @Override
7 public void close() {
8     if (dataSource != null) {
9         JedisPoolAbstract pool = this.dataSource;
10        this.dataSource = null;
11        if (client.isBroken()) {
12            pool.returnBrokenResource(this);
13        } else {
14            pool.returnResource(this);
15        }
16    } else {
17        super.close();
18    }
19 }

```


如果不是，则直接关闭连接，其最终调用 Connection 类的 disconnect 方法来关闭 TCP 连接：

[复制代码](#)

```
1 public void disconnect() {
2     if (isConnected()) {
3         try {
4             outputStream.flush();
5             socket.close();
6         } catch (IOException ex) {
7             broken = true;
8             throw new JedisConnectionException(ex);
9         } finally {
10             IOUtils.closeQuietly(socket);
11         }
12     }
13 }
```

可以看到，Jedis 可以独立使用，也可以配合连接池使用，这个连接池就是 JedisPool。我们再看看 JedisPool 的实现。

[复制代码](#)

```
1 public class JedisPool extends JedisPoolAbstract {
2     @Override
3     public Jedis getResource() {
4         Jedis jedis = super.getResource();
5         jedis.setDataSource(this);
6         return jedis;
7     }
8
9     @Override
10    protected void returnResource(final Jedis resource) {
11        if (resource != null) {
12            try {
13                resource.resetState();
14                returnResourceObject(resource);
15            } catch (Exception e) {
16                returnBrokenResource(resource);
17                throw new JedisException("Resource is returned to the pool as broken",
18                    e);
19            }
20        }
21    }
```

```
21 }
22
23 public class JedisPoolAbstract extends Pool<Jedis> {
24 }
25
26 public abstract class Pool<T> implements Closeable {
27     protected GenericObjectPool<T> internalPool;
28 }
```

JedisPool 的 getResource 方法在拿到 Jedis 对象后，将自己设置为了连接池。连接池 JedisPool，继承了 JedisPoolAbstract，而后者继承了抽象类 Pool，Pool 内部维护了 Apache Common 的通用池 GenericObjectPool。JedisPool 的连接池就是基于 GenericObjectPool 的。

看到这里我们了解了，Jedis 的 API 实现是我们说的三种类型中的第一种，也就是连接池和连接分离的 API，JedisPool 是线程安全的连接池，Jedis 是非线程安全的单一连接。知道了原理之后，我们再使用 Jedis 就胸有成竹了。

使用连接池务必确保复用

在介绍 [线程池](#) 的时候我们强调过，**池一定是用来复用的，否则其使用代价会比每次创建单一对象更大。对连接池来说更是如此，原因如下：**

创建连接池的时候很可能一次性创建了多个连接，大多数连接池考虑到性能，会在初始化的时候维护一定数量的最小连接（毕竟初始化连接池的过程一般是一次性的），可以直接使用。如果每次使用连接池都按需创建连接池，那么很可能你只用到一个连接，但是创建了 N 个连接。

连接池一般会有一些管理模块，也就是连接池的结构示意图中的绿色部分。举个例子，大多数的连接池都有闲置超时的概念。连接池会检测连接的闲置时间，定期回收闲置的连接，把活跃连接数降到最低（闲置）连接的配置值，减轻服务端的压力。一般情况下，闲置连接由独立线程管理，启动了空闲检测的连接池相当于还会启动一个线程。此外，有些连接池还需要独立线程负责连接保活等功能。因此，启动一个连接池相当于启动了 N 个线程。

除了使用代价，连接池不释放，还可能会引起线程泄露。接下来，我就以 Apache HttpClient 为例，和你说说连接池不复用的问题。

首先，创建一个 `CloseableHttpClient`，设置使用 `PoolingHttpClientConnectionManager` 连接池并启用空闲连接驱逐策略，最大空闲时间为 60 秒，然后使用这个连接来请求一个会返回 OK 字符串的服务端接口：

 复制代码

```
1 @GetMapping("wrong1")
2 public String wrong1() {
3     CloseableHttpClient client = HttpClients.custom()
4         .setConnectionManager(new PoolingHttpClientConnectionManager())
5         .evictIdleConnections(60, TimeUnit.SECONDS).build();
6     try (CloseableHttpResponse response = client.execute(new HttpGet("http://1:
7         return EntityUtils.toString(response.getEntity());
8     } catch (Exception ex) {
9         ex.printStackTrace();
10    }
11    return null;
12 }
```

访问这个接口几次后查看应用线程情况，可以看到有大量叫作 `Connection evictor` 的线程，且这些线程不会销毁：

```
→ ~ jstack 91133 | grep evictor
"Connection evictor" #121 daemon prio=5 os_prio=31 tid=0x00007f87a2d68000 nid=0xdc03 waiting on condition [0x00007000169d4000]
"Connection evictor" #120 daemon prio=5 os_prio=31 tid=0x00007f87a0054800 nid=0xdb03 waiting on condition [0x00007000168d1000]
"Connection evictor" #119 daemon prio=5 os_prio=31 tid=0x00007f879fad6000 nid=0x11d03 waiting on condition [0x00007000167ce000]
"Connection evictor" #118 daemon prio=5 os_prio=31 tid=0x00007f879fad5800 nid=0x11f03 waiting on condition [0x00007000166cb000]
"Connection evictor" #117 daemon prio=5 os_prio=31 tid=0x00007f87a1fd3800 nid=0xda03 waiting on condition [0x00007000165c8000]
"Connection evictor" #116 daemon prio=5 os_prio=31 tid=0x00007f87a0053800 nid=0xd903 waiting on condition [0x00007000164c5000]
"Connection evictor" #115 daemon prio=5 os_prio=31 tid=0x00007f879fad4800 nid=0xd803 waiting on condition [0x00007000163c2000]
"Connection evictor" #114 daemon prio=5 os_prio=31 tid=0x00007f87a0858800 nid=0xd603 waiting on condition [0x00007000162bf000]
"Connection evictor" #113 daemon prio=5 os_prio=31 tid=0x00007f879f949800 nid=0xd403 waiting on condition [0x00007000161bc000]
"Connection evictor" #112 daemon prio=5 os_prio=31 tid=0x00007f87a387d800 nid=0x12303 waiting on condition [0x00007000160b9000]
"Connection evictor" #111 daemon prio=5 os_prio=31 tid=0x00007f87a3b4a000 nid=0xd203 waiting on condition [0x0000700015fb6000]
"Connection evictor" #110 daemon prio=5 os_prio=31 tid=0x00007f87a3b48000 nid=0xd103 waiting on condition [0x0000700015eb3000]
"Connection evictor" #109 daemon prio=5 os_prio=31 tid=0x00007f879f9c9800 nid=0xcf03 waiting on condition [0x0000700015db0000]
"Connection evictor" #108 daemon prio=5 os_prio=31 tid=0x00007f87a2461000 nid=0x12603 waiting on condition [0x0000700015cad000]
"Connection evictor" #107 daemon prio=5 os_prio=31 tid=0x00007f87a322a800 nid=0xcc03 waiting on condition [0x0000700015baa000]
"Connection evictor" #106 daemon prio=5 os_prio=31 tid=0x00007f879fae1000 nid=0x12803 waiting on condition [0x0000700015aa7000]
```

对这个接口进行几秒的压测（压测使用 `wrk`，1 个并发 1 个连接）可以看到，已经建立了三千多个 TCP 连接到 45678 端口（其中有 1 个是压测客户端到 Tomcat 的连接，大部分都是 `HttpClient` 到 Tomcat 的连接）：

```
→ ~ lsof -nP -i4TCP:45678 | wc -l
3686
```

好在有了空闲连接回收的策略，60 秒之后连接处于 CLOSE_WAIT 状态，最终彻底关闭。


```
→ ~ lsof -nP -iTCP:45678
```

COMMAND	PID	USER	FD	TYPE	DEVICE	SIZE/OFF	NODE	NAME
java	91133	zhuye	403u	IPv6	0x8a653ae8f9ffb77b	0t0	TCP	*:45678 (LISTEN)
java	91133	zhuye	416u	IPv6	0x8a653ae92777eefb	0t0	TCP	127.0.0.1:62655->127.0.0.1:45678 (CLOSE_WAIT)

这 2 点证明，CloseableHttpClient 属于第二种模式，即内部带有连接池的 API，其背后是连接池，最佳实践一定是复用。

复用方式很简单，你可以把 CloseableHttpClient 声明为 static，只创建一次，并且在 JVM 关闭之前通过 addShutdownHook 钩子关闭连接池，在使用的时候直接使用 CloseableHttpClient 即可，无需每次都创建。

首先，定义一个 right 接口来实现服务端接口调用：

 复制代码

```
1 private static CloseableHttpClient httpClient = null;
2 static {
3     //当然，也可以把CloseableHttpClient定义为Bean，然后在@PreDestroy标记的方法内close
4     httpClient = HttpClients.custom().setMaxConnPerRoute(1).setMaxConnTotal(1)
5     Runtime.getRuntime().addShutdownHook(new Thread(() -> {
6         try {
7             httpClient.close();
8         } catch (IOException ignored) {
9         }
10    }));
11 }
12
13 @GetMapping("right")
14 public String right() {
15     try (CloseableHttpResponse response = httpClient.execute(new HttpGet("http
16         return EntityUtils.toString(response.getEntity());
17     } catch (Exception ex) {
18         ex.printStackTrace();
19     }
20     return null;
21 }
```

然后，重新定义一个 wrong2 接口，修复之前按需创建 CloseableHttpClient 的代码，每次用完之后确保连接池可以关闭：

```
1 @GetMapping("wrong2")
2 public String wrong2() {
3     try (CloseableHttpClient client = HttpClients.custom()
4         .setConnectionManager(new PoolingHttpClientConnectionManager())
5         .evictIdleConnections(60, TimeUnit.SECONDS).build());
6         CloseableHttpResponse response = client.execute(new HttpGet("http://1:
7         return EntityUtils.toString(response.getEntity());
8     } catch (Exception ex) {
9         ex.printStackTrace();
10    }
11    return null;
12 }
```

使用 wrk 对 wrong2 和 right 两个接口分别压测 60 秒，可以看到两种使用方式性能上的差异，每次创建连接池的 QPS 是 337，而复用连接池的 QPS 是 2022：

```
➔ ~ wrk -c1 -t1 -d 10s http://localhost:45678/httpclientnotreuse/wrong2
Running 10s test @ http://localhost:45678/httpclientnotreuse/wrong2
 1 threads and 1 connections
Thread Stats   Avg      Stdev     Max   +/-  Stdev
  Latency    6.74ms  26.12ms 283.57ms  97.67%
  Req/Sec   345.46   122.63  565.00   62.24%
3376 requests in 10.02s, 379.75KB read
Requests/sec:   337.01
Transfer/sec:    37.91KB

➔ ~ wrk -c1 -t1 -d 10s http://localhost:45678/httpclientnotreuse/right
Running 10s test @ http://localhost:45678/httpclientnotreuse/right
 1 threads and 1 connections
Thread Stats   Avg      Stdev     Max   +/-  Stdev
  Latency   562.45us   0.88ms  20.67ms   98.20%
  Req/Sec    2.03k   520.14   3.34k    69.31%
20438 requests in 10.10s, 2.25MB read
Requests/sec:  2022.79
Transfer/sec:   227.54KB
```

如此大的性能差异显然是因为 TCP 连接的复用。你可能注意到了，刚才定义连接池时，我将最大连接数设置为 1。所以，复用连接池方式复用的始终应该是同一个连接，而新建连接池方式应该是每次都会创建新的 TCP 连接。

接下来，我们通过网络抓包工具 Wireshark 来证实这一点。

如果调用 wrong2 接口每次创建新的连接池来发起 HTTP 请求，从 Wireshark 可以看到，每次请求服务端 45678 的客户端端口都是新的。这里我发起了三次请求，程序通过

HttpClient 访问服务端 45678 的客户端端口号，分别是 51677、51679 和 51681：

tcp.port == 45678 && http							
No.	Time	Source	Source port	Dest port	Destination	Protocol	Length
11	2.113048	:::1	51676	45678	:::1	HTTP	
17	2.270371	127.0.0.1	51677	45678	127.0.0.1	HTTP	
19	2.292614	127.0.0.1	45678	51677	127.0.0.1	HTTP	
23	2.306919	:::1	45678	51676	:::1	HTTP	
37	2.738059	:::1	51678	45678	:::1	HTTP	
43	2.743239	127.0.0.1	51679	45678	127.0.0.1	HTTP	
45	2.745434	127.0.0.1	45678	51679	127.0.0.1	HTTP	
51	2.747162	:::1	45678	51678	:::1	HTTP	
63	3.386317	:::1	51680	45678	:::1	HTTP	
69	3.389662	127.0.0.1	51681	45678	127.0.0.1	HTTP	
71	3.391503	127.0.0.1	45678	51681	127.0.0.1	HTTP	
77	3.392697	:::1	45678	51680	:::1	HTTP	

```
► Frame 43: 229 bytes on wire (1832 bits), 229 bytes captured (1832 bits) on interface 0
► Null/Loopback
► Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
► Transmission Control Protocol, Src Port: 51679, Dst Port: 45678, Seq: 43730075, Ack: 2097713985, Len: 173
▼ Hypertext Transfer Protocol
► GET /httpClientnotreuse/test HTTP/1.1\r\n
  Host: 127.0.0.1:45678\r\n
  Connection: Keep-Alive\r\n
  User-Agent: Apache-HttpClient/4.5.9 (Java/1.8.0_211)\r\n
  Accept-Encoding: gzip,deflate\r\n
  \r\n
  [Full request URI: http://127.0.0.1:45678/httpclientnotreuse/test]
  [HTTP request 1/1]
  [Response in frame: 45]
```

也就是说，每次都是新的 TCP 连接，放开 HTTP 这个过滤条件也可以看到完整的 TCP 握手、挥手的过程：

9	0.681922	127.0.0.1	51775	45678	127.0.0.1	TCP	握手
10	0.682008	127.0.0.1	45678	51775	127.0.0.1	TCP	
11	0.682022	127.0.0.1	51775	45678	127.0.0.1	TCP	
12	0.682032	127.0.0.1	45678	51775	127.0.0.1	TCP	设置窗口大小
13	0.682202	127.0.0.1	51775	45678	127.0.0.1	HTTP	
14	0.682213	127.0.0.1	45678	51775	127.0.0.1	TCP	
15	0.690042	127.0.0.1	45678	51775	127.0.0.1	HTTP	
16	0.690064	127.0.0.1	51775	45678	127.0.0.1	TCP	
17	0.691058	127.0.0.1	51775	45678	127.0.0.1	TCP	挥手
18	0.691072	127.0.0.1	45678	51775	127.0.0.1	TCP	
19	0.691549	127.0.0.1	45678	51775	127.0.0.1	TCP	
20	0.691603	127.0.0.1	51775	45678	127.0.0.1	TCP	
21	0.691618	:::1	45678	51774	:::1	HTTP	
22	0.691631	:::1	51774	45678	:::1	TCP	
23	0.691759	:::1	51774	45678	:::1	TCP	
24	0.691771	:::1	45678	51774	:::1	TCP	
25	0.691988	:::1	45678	51774	:::1	TCP	
26	0.692024	:::1	51774	45678	:::1	TCP	

▶ Frame 20: 56 bytes on wire (448 bits), 56 bytes captured (448 bits) on interface 0
 ▶ Null/Loopback
 ▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
 ▼ Transmission Control Protocol, Src Port: 51775, Dst Port: 45678, Seq: 353173988, Ack: 2740860102, Len: 0
 Source Port: 51775
 Destination Port: 45678
 [Stream index: 2]
 [TCP Segment Len: 0]
 Sequence number: 353173988
 [Next sequence number: 353173988]
 Acknowledgment number: 2740860102
 1000 = Header Length: 32 bytes (8)
 ▶ Flags: 0x010 (ACK)
 Window size value: 6377
 [Calculated window size: 408128]
 [Window size scaling factor: 64]
 Checksum: 0xfe28 [unverified]
 [Checksum Status: Unverified]
 Urgent pointer: 0
 ▶ Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
 ▼ [SEQ/ACK analysis]
 [\[This is an ACK to the segment in frame: 19\]](#)
 [The RTT to ACK the segment was: 0.000054000 seconds]

而复用连接池方式的接口 right 的表现就完全不同了。可以看到，第二次 HTTP 请求 #41 的客户端端口 61468 和第一次连接 #23 的端口是一样的，Wireshark 也提示了整个 TCP 会话中，当前 #41 请求是第二次请求，前一次是 #23，后面一次是 #75：

19	3.563647	127.0.0.1	61468	45678	127.0.0.1	TCP
20	3.563757	127.0.0.1	45678	61468	127.0.0.1	TCP
21	3.563774	127.0.0.1	61468	45678	127.0.0.1	TCP
22	3.563784	127.0.0.1	45678	61468	127.0.0.1	TCP
23	3.563967	127.0.0.1	61468	45678	127.0.0.1	HTTP
24	3.563984	127.0.0.1	45678	61468	127.0.0.1	TCP
25	3.566317	127.0.0.1	45678	61468	127.0.0.1	HTTP
26	3.566341	127.0.0.1	61468	45678	127.0.0.1	TCP
27	3.567827	:::1	45678	61467	:::1	HTTP
28	3.567862	:::1	61467	45678	:::1	TCP
29	3.568278	:::1	61467	45678	:::1	TCP
30	3.568299	:::1	45678	61467	:::1	TCP
31	3.568656	:::1	45678	61467	:::1	TCP
32	3.568690	:::1	61467	45678	:::1	TCP
35	4.509686	:::1	61470	45678	:::1	TCP
36	4.509785	:::1	45678	61470	:::1	TCP
37	4.509798	:::1	61470	45678	:::1	TCP
38	4.509805	:::1	45678	61470	:::1	TCP
39	4.509856	:::1	61470	45678	:::1	HTTP
40	4.509876	:::1	45678	61470	:::1	TCP
41	4.512295	127.0.0.1	61468	45678	127.0.0.1	HTTP
42	4.512315	127.0.0.1	45678	61468	127.0.0.1	TCP
43	4.517425	127.0.0.1	45678	61468	127.0.0.1	HTTP
44	4.517460	127.0.0.1	61468	45678	127.0.0.1	TCP
45	4.518995	:::1	45678	61470	:::1	HTTP
46	4.519019	:::1	61470	45678	:::1	TCP
47	4.519133	:::1	61470	45678	:::1	TCP

▶ Frame 41: 229 bytes on wire (1832 bits), 229 bytes captured (1832 bits) on interface 0

▶ Null/Loopback

▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1

▶ Transmission Control Protocol, Src Port: 61468, Dst Port: 45678, Seq: 532212967, Ack: 160269583, Len: 173

▼ Hypertext Transfer Protocol

▶ GET /httpClientnotreuse/test HTTP/1.1\r\n

Host: 127.0.0.1:45678\r\n

Connection: Keep-Alive\r\n

User-Agent: Apache-HttpClient/4.5.9 (Java/1.8.0_211)\r\n

Accept-Encoding: gzip,deflate\r\n

\r\n

[Full request URI: <http://127.0.0.1:45678/httpclientnotreuse/test>]

[HTTP request 2/3]

[Prev request in frame: 23]

[Response in frame: 43]

[Next request in frame: 75]

只有 TCP 连接闲置超过 60 秒后才会断开，连接池会新建连接。你可以尝试通过 Wireshark 观察这一过程。

接下来，我们就继续聊聊连接池的配置问题。

连接池的配置不是一成不变的

为方便根据容量规划设置连接处的属性，连接池提供了许多参数，包括最小（闲置）连接、最大连接、闲置连接生存时间、连接生存时间等。其中，最重要的参数是最大连接数，它决定了连接池能使用的连接数量上限，达到上限后，新来的请求需要等待其他请求释放连接。

但，**最大连接数不是设置得越大越好**。如果设置得太大，不仅仅是客户端需要耗费过多的资源维护连接，更重要的是由于服务端对应的是多个客户端，每一个客户端都保持大量的连

接，会给服务端带来更大的压力。这个压力又不仅仅是内存压力，可以想一下如果服务端的网络模型是一个 TCP 连接一个线程，那么几千个连接意味着几千个线程，如此多的线程会造成大量的线程切换开销。

当然，连接池最大连接数设置得太小，很可能会导致获取连接的等待时间太长，导致吞吐量低下，甚至超时无法获取连接。

接下来，我们就模拟下压力增大导致数据库连接池打满的情况，来实践下如何确认连接池的使用情况，以及有针对性地进行参数优化。

首先，定义一个用户注册方法，通过 @Transactional 注解为方法开启事务。其中包含了 500 毫秒的休眠，一个数据库事务对应一个 TCP 连接，所以 500 多毫秒的时间都会占用数据库连接：

 复制代码

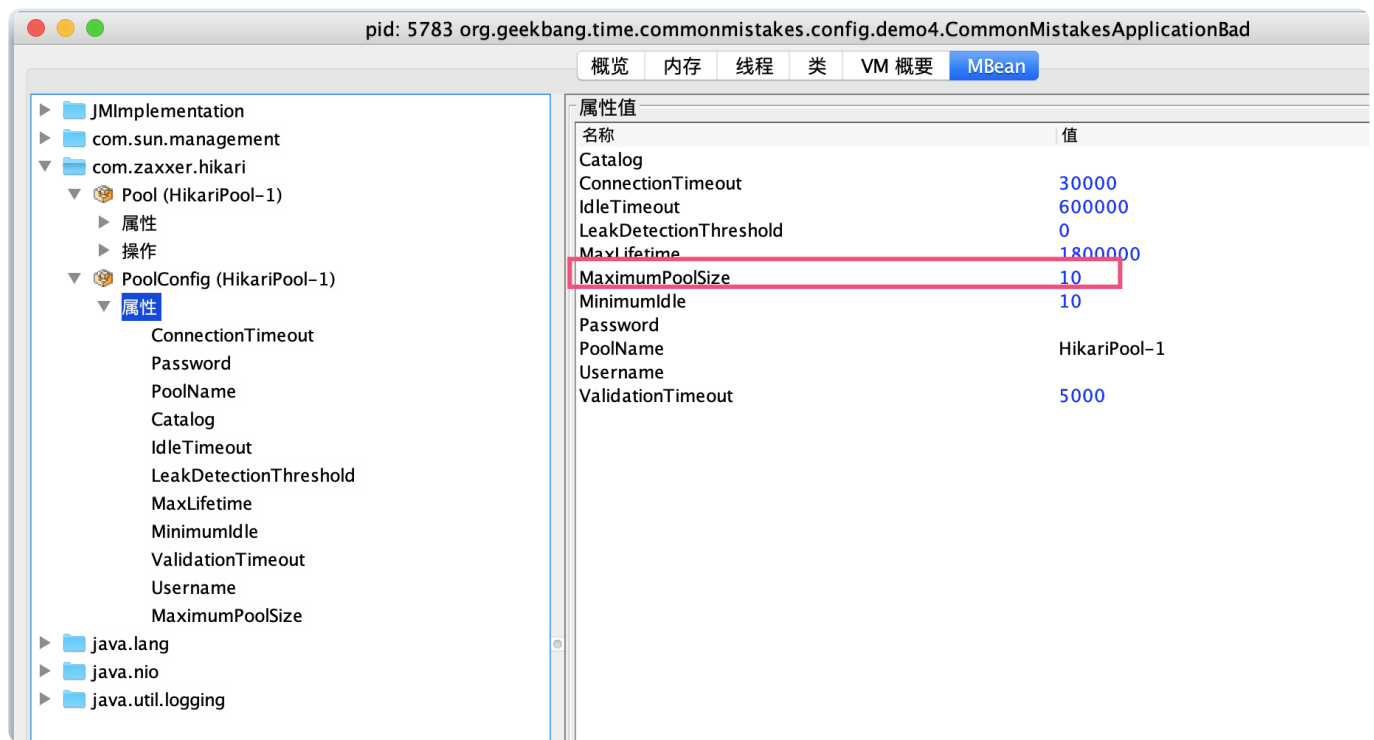
```
1 @Transactional
2 public User register(){
3     User user=new User();
4     user.setName("new-user-"+System.currentTimeMillis());
5     userRepository.save(user);
6     try {
7         TimeUnit.MILLISECONDS.sleep(500);
8     } catch (InterruptedException e) {
9         e.printStackTrace();
10    }
11    return user;
12 }
```

随后，修改配置文件启用 register-mbeans，使 Hikari 连接池能通过 JMX MBean 注册连接池相关统计信息，方便观察连接池：

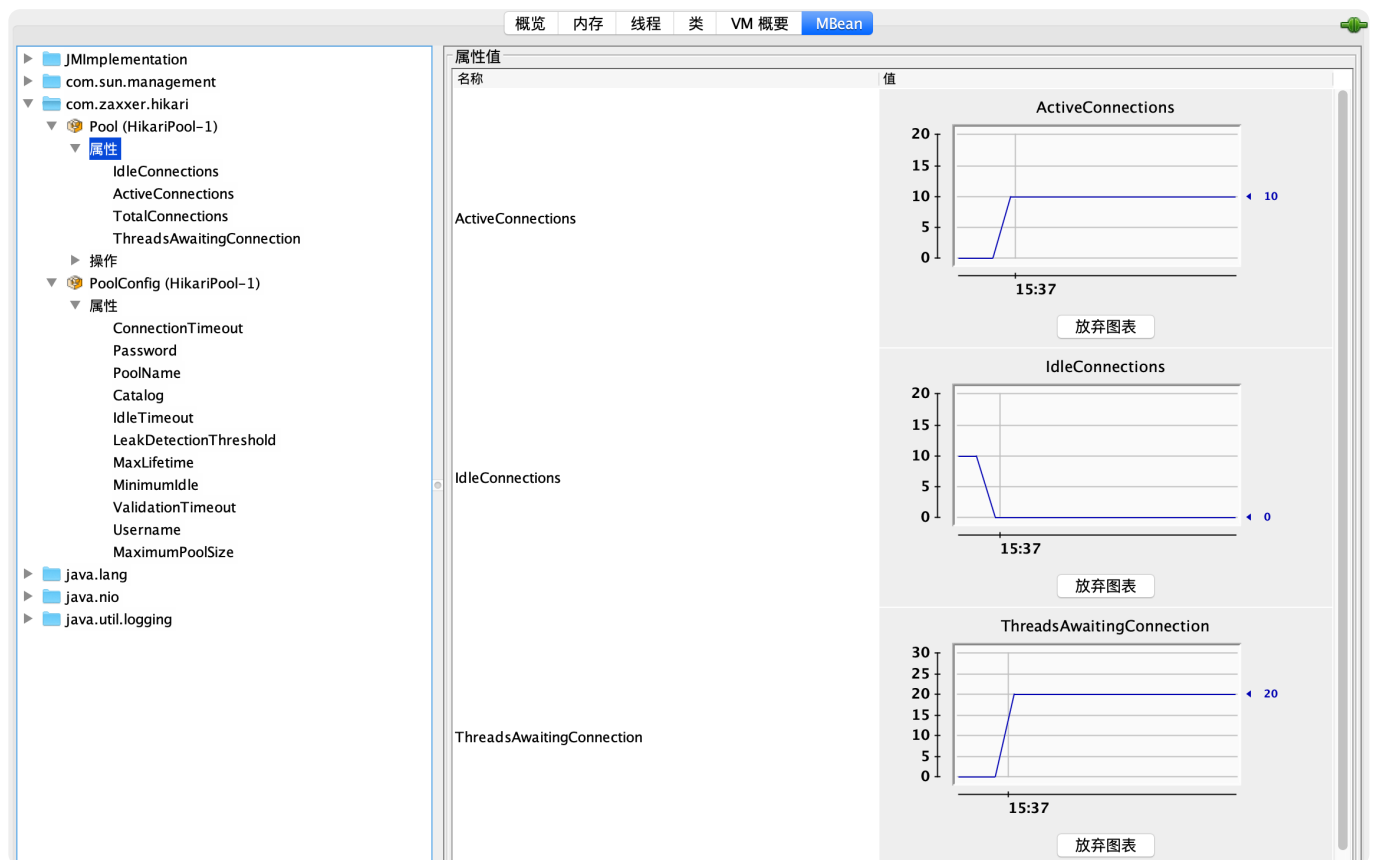
 复制代码

```
1 spring.datasource.hikari.register-mbeans=true
```

启动程序并通过 JConsole 连接进程后，可以看到默认情况下最大连接数为 10：



使用 wrk 对应用进行压测，可以看到连接数一下子从 0 到了 10，有 20 个线程在等待获取连接：



不久就出现了无法获取数据库连接的异常，如下所示：

复制代码

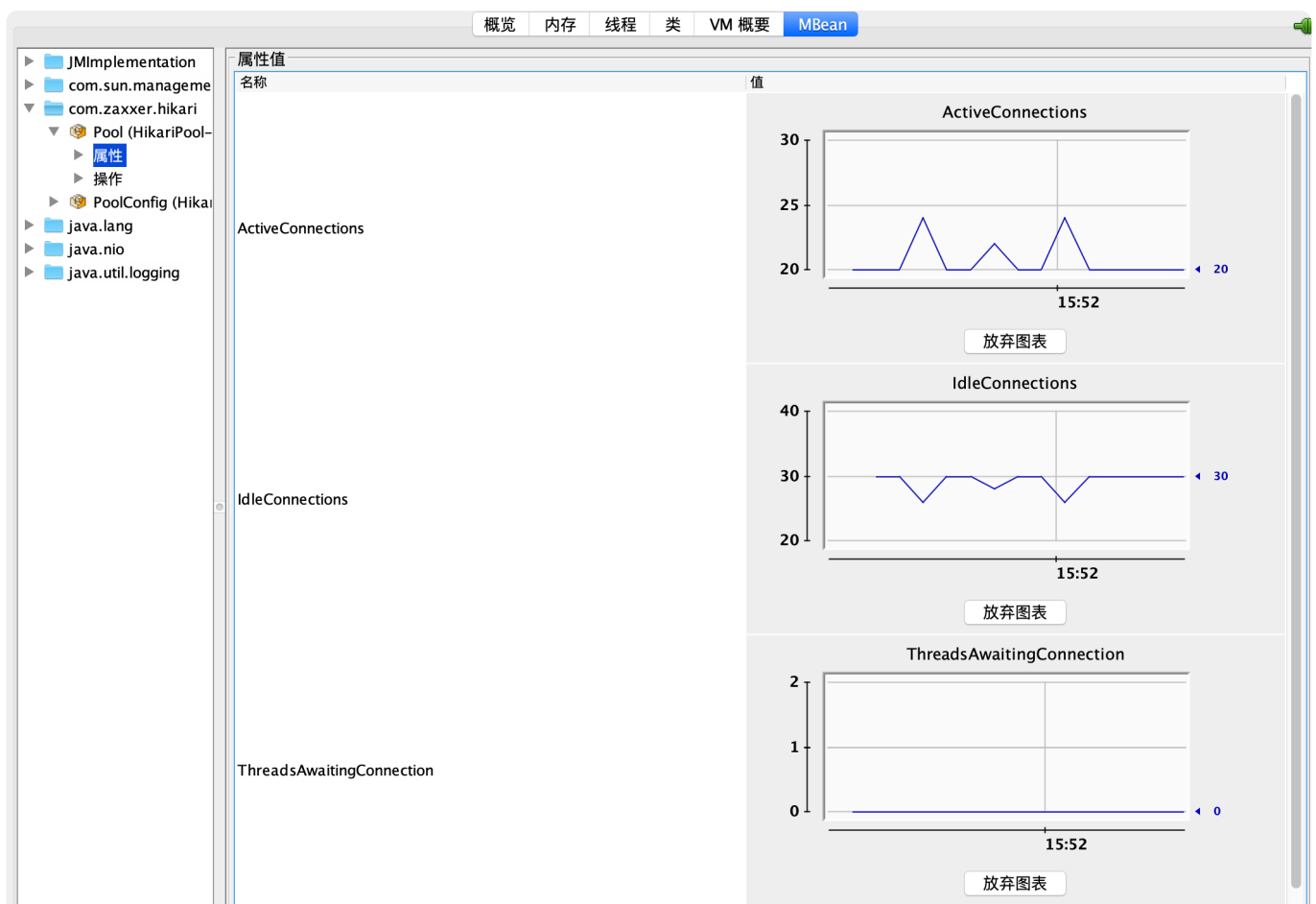
```
1 [15:37:56.156] [http-nio-45678-exec-15] [ERROR] [.a.c.c.C.[./].[dispatcherS  
2 java.sql.SQLException: HikariPool-1 - Connection is not ava
```

从异常信息中可以看到，数据库连接池是 HikariPool，解决方式很简单，修改一下配置文件，调整数据库连接池最大连接参数到 50 即可。

复制代码

```
1 spring.datasource.hikari.maximum-pool-size=50
```

然后，再观察一下这个参数是否适合当前压力，满足需求的同时也不占用过多资源。从监控来看这个调整是合理的，有一半的富余资源，再也没有线程需要等待连接了：



在这个 Demo 里，我知道压测大概能对应使用 25 左右的并发连接，所以直接把连接池最大连接设置为了 50。在真实情况下，只要数据库可以承受，你可以选择在遇到连接超限的时候先设置一个足够大的连接数，然后观察最终应用的并发，再按照实际并发数留出一半的余量来设置最终的最大连接。

其实，看到错误日志后再调整已经有点儿晚了。更合适的做法是，**对类似数据库连接池的重要资源进行持续检测，并设置一半的使用量作为报警阈值，出现预警后及时扩容。**

在这里我是为了演示，才通过 JConsole 查看参数配置后的效果，生产上需要把相关数据对接到指标监控体系中持续监测。

这里要强调的是，修改配置参数务必验证是否生效，并且在监控系统中确认参数是否生效、是否合理。之所以要“强调”，是因为这里有坑。

我之前就遇到过这样一个事故。应用准备针对大促活动进行扩容，把数据库配置文件中 Druid 连接池最大连接数 maxActive 从 50 提高到了 100，修改后并没有通过监控验证，结果大促当天应用因为连接池连接数不够爆了。

经排查发现，当时修改的连接数并没有生效。原因是，应用虽然一开始使用的是 Druid 连接池，但后来框架升级了，把连接池替换为了 Hikari 实现，原来的那些配置其实都是无效的，修改后的参数配置当然也不会生效。

所以说，对连接池进行调参，一定要眼见为实。

重点回顾

今天，我以三种业务代码最常用的 Redis 连接池、HTTP 连接池、数据库连接池为例，和你探讨了有关连接池实现方式、使用姿势和参数配置的三大问题。

客户端 SDK 实现连接池的方式，包括池和连接分离、内部带有连接池和非连接池三种。要正确使用连接池，就必须首先鉴别连接池的实现方式。比如，Jedis 的 API 实现的是池和连接分离的方式，而 Apache HttpClient 是内置连接池的 API。

对于使用姿势其实就是两点，一是确保连接池是复用的，二是尽可能在程序退出之前显式关闭连接池释放资源。连接池设计的初衷就是为了保持一定量的连接，这样连接可以随取随用。从连接池获取连接虽然很快，但连接池的初始化会比较慢，需要做一些管理模块的初始化以及初始最小闲置连接。一旦连接池不是复用的，那么其性能会比随时创建单一连接更差。

最后，连接池参数配置中，最重要的是最大连接数，许多高并发应用往往因为最大连接数不够导致性能问题。但，最大连接数不是设置得越大越好，够用就好。需要注意的是，针对数据库连接池、HTTP 连接池、Redis 连接池等重要连接池，务必建立完善的监控和报警机制，根据容量规划及时调整参数配置。

今天用到的代码，我都放在了 GitHub 上，你可以点击 [🔗 这个链接](#) 查看。

思考与讨论

1. 有了连接池之后，获取连接是从连接池获取，没有足够连接时连接池会创建连接。这时，获取连接操作往往有两个超时时间：一个是从连接池获取连接的最长等待时间，通常叫作请求超时 `connectRequestTimeout` 或等待超时 `connectWaitTimeout`；一个是连接池新建 TCP 连接三次握手的连接超时，通常叫作连接超时 `connectTimeout`。针对 JedisPool、Apache HttpClient 和 Hikari 数据库连接池，你知道如何设置这 2 个参数吗？
2. 对于带有连接池的 SDK 的使用姿势，最主要的是鉴别其内部是否实现了连接池，如果实现了连接池要尽量复用 Client。对于 NoSQL 中的 MongoDB 来说，使用 MongoDB Java 驱动时，MongoClient 类应该是每次都创建还是复用呢？你能否在 [🔗 官方文档](#) 中找到答案呢？

关于连接池，你还遇到过什么坑吗？我是朱晔，欢迎在评论区与我留言分享，也欢迎你把这篇文章分享给你的朋友或同事，一起交流。

点击参与 

进入朱晔老师「读者群」带你攻克 Java 业务开发常见错误



添加Java班长，报名入群



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 03 | 线程池：业务代码最常用也最容易犯错的组件

下一篇 加餐1 | 带你吃透课程中Java 8的那些重要知识点（上）

精选留言 (20)

 写留言



Wiggle Wiggle 置顶

2020-03-14

请问对于连接池的监控，是把监控系统直连JMX，监控、修改操作都走连接池已经实现好的JMX比较好？还是自己做一层封装，对外暴露接口，以编程方式获取、设置参数比较好？

作者回复：自己封装必要不大，然后如果需要走http的话用jolokia，或者使用prometheus的jmx_exporter以agent方式暴露mbeans，https://github.com/prometheus/jmx_exporter



5



Darren 置顶

2020-03-16

实操性比较强，收获满满！！

自从spring boot 2.x版本后，有较大的改动：

默认的redis的连接池从JedisPool变成了LettucePool，Lettuce主要利用netty实现与redis的同步和异步通信。所以更安全和性能更好；

默认的数据库连接池也变更为HikariCP，HiKariCP 号称是业界跑得最快的数据库连接...
展开 ▾

作者回复：

假设我们希望设置连接超时5s，获取连接超时10s：

hikari两个参数设置方式：

```
spring.datasource.hikari.connection-timeout=10000
```

```
spring.datasource.url=jdbc:mysql://localhost:6657/common_mistakes?connectTimeout=5000&characterEncoding=UTF-8&useSSL=false&rewriteBatchedStatements=true
```

jedis两个参数设置：

```
JedisPoolConfig config = new JedisPoolConfig();
    config.setMaxWaitMillis(10000);
    try (JedisPool jedisPool = new JedisPool(config, "127.0.0.1", 6379, 5000);
        Jedis jedis = jedisPool.getResource()) {
        return jedis.set("test", "test");
    }
```

httpClient两个参数设置：

```
RequestConfig requestConfig = RequestConfig.custom()
    .setConnectTimeout(5000)
    .setConnectionRequestTimeout(10000)
    .build();

HttpGet httpGet = new HttpGet("http://127.0.0.1:45678/twotimeoutconfig/test");
httpGet.setConfig(requestConfig);
try (CloseableHttpResponse response = httpClient.execute(httpGet)) {...
```

1

4



置顶

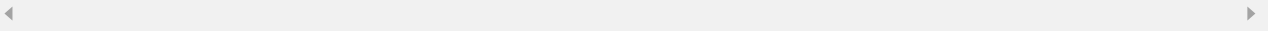
2020-03-16

个人总结：

1. 池化技术的核心在于，在鱼塘养好一群鱼，需要的时候就从里面拿一条，用完再放回去。而不是自己生产一条鱼，然后用完就销毁。从而减少了开销。
2. 大多已经实现的连接池，都是有线程安全处理的。通常比个人创建管理连接更加安全。
3. 使用了连接池技术，就要保证连接池能够被有效复用。频繁创建连接池比频繁创建链...

展开 ▾

作者回复: 总结的不错



1



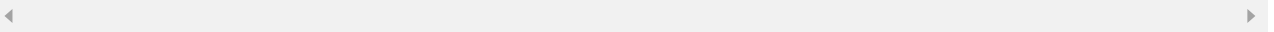
每天晒白牙

2020-03-14

干货满满，还需要慢慢消化一下

展开 ▾

作者回复: 如有收货，欢迎转发



3

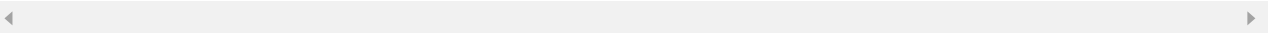


小美

2020-03-14

退出程序前为什么要关闭连接池啊，程序都结束了连接不就释放了么

作者回复: 优雅关闭总是更好的



2



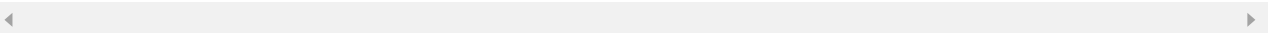
Husiun

2020-03-14

每次更新都是第一时间打开，每一课都干货满满，必须给老师赞一个，http那个平时研究不多还需要好好消化一下。

展开 ▾

作者回复: 有收货就好



2



2020-03-16

课后题2:

受限于本人英文水平，无奈与使用谷歌翻译阅读文档。从文档中得知，MongoClient 对象的正确使用姿势应该是：使用 MongoClient.create()（或者其他有参）方法创建，并再

整个应用程序中使用它。文档内容如下：

...

展开 ▾

作者回复: 是的



1



2020-03-16

课后题1:

Hikari 可以再Spring的配置文件中配置各项参数。

作者回复: 具体配置参数是什么呢?



1



pedro

2020-03-14

干货很多，收获很大。问老师一个问题，使用hook来关闭连接池的时候，都会创建一个线程，那如果有多个连接池，每个连接池都有一个线程来调用hook，这样做是否有点奢侈，有没有更优的办法？

作者回复: 可以都放到一个Thread，放到多个的话会并行执行，只要不是太多问题不大。或者用Spring的话还可以使用@PreDestroy来实现资源释放。



1

1



eazonshaw

2020-03-17

老师，感觉文章对网络抓包工具 Wireshark 的截图信息不是很完整，right接口的#75也没有看到，这一块看着有点懵。可以详细说明下调用wrong2和right的区别吗？谢谢。

作者回复: wrong2的截图已经替换了，晚上会更新

right的#75图上看不到，图太长了，不过截图已经可以连接复用了





hellojd

2020-03-17

进程都kill了，相关的资源应该自动释放吧，比如启动的线程和连接。jvm环境都没了，依赖的基础都没了

作者回复: 优雅关闭总是好的，特别是涉及到服务端的tcp连接



hellojd

2020-03-17

几个疑问:第一个:demo中的/right接口，是不是也算有问题的，仅仅方便对比效果而已，这让请求串行了吧。第二个:http client还有一套连接剔除机制，demo里没讲到。第三个:spring boot客户端工具，默认设置的一些配置，有需要注意的地方吗？第四个:连接数监控，什么情况代表着有问题,有时候一个数据库资源，被多个微服务公用，怎么划分配额。

展开 ∨

作者回复: 3、可以找一下spring boot auto configuration类，看看其自动配置是否符合需求

4、这可能还有一个设计问题，理论上微服务应该只能访问自己domain的数据库，给其他微服务提供数据只能以接口提供；当然资源有限的情况下混用数据库实例的话，一般连接数量不会成为明显瓶颈，有瓶颈的话可能要考虑实例拆分了。



god

2020-03-17

辛苦！老师

展开 ∨



LittleFatz

2020-03-16

请教老师一个问题，在wrong2的wireshark截图里，除了#39使用端口60686外，#37、#35和#42都使用了这个端口，单从截图而言，貌似无法证明每次request都使用了新的端口发送请求。

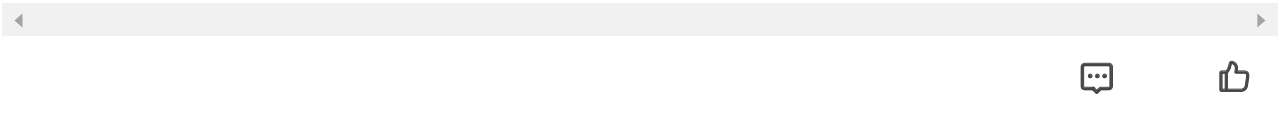
作者回复: 你这个问题非常好，看的很仔细。

主要还是这个图太长了，我这里不好贴。

你提到的35、37和42可能是握手和挥手，最前面有三次握手+更新窗口大小四个TCP，最后面有

挥手四个TCP，看到的这些其实还是一个请求的。

你可以使用wireshark来调试一下wrong2接口，看看是不是每次都换了端口。



2020-03-16

hikari具体配置项为application.yml 中 spring.datasource.hikari.connection-timeout 点进去可以发现是 HikariDataSource 类，继承了HikariConfig。
点进HikariConfig可看出 connectionTimeout不允许小于250毫秒，小于250ms会被强制重置为30秒。

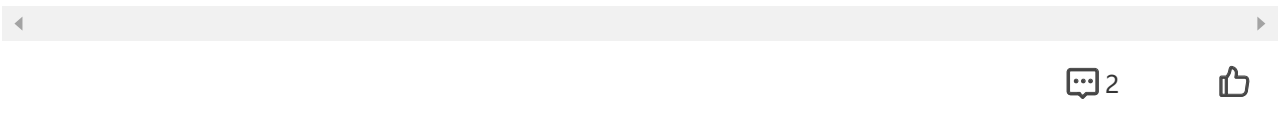
参数connectionTimeout定义是并未赋初始值的原始类型long，初始值应该是0L； ...

展开 ∨

作者回复: 回答一下Hikari的配置，其ConnectionTimeout是从连接池获取数据库连接的超时，不是和MySQL建立连接的超时，后者需要设置JDBC连接字符串中的connectTimeout属性。对于Hikari的JavaConfig配置这2个参数的方式是：

@Bean

```
public DataSource dataSource(){
    HikariConfig hikariConfig = new HikariConfig();
    hikariConfig.setConnectionTimeout(2400);
    hikariConfig.setJdbcUrl("jdbc:mysql://localhost:6658/common_mistakes");
    hikariConfig.addDataSourceProperty("connectTimeout", "1200");
    HikariDataSource dataSource = new HikariDataSource(hikariConfig);
    return dataSource;
}
```



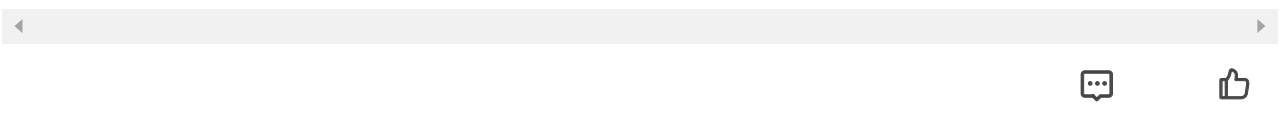
mgs2002

2020-03-16

看了一下mongDB官方文档，应该是第二种类型的，连接池的维护管理都是mongDB自己进行，用户不用手动去关闭，要手动的话也可以调用close()方法

展开 ∨

作者回复: 是的，要关闭的话也应该在程序shutdown的时候去关闭





袁素芬

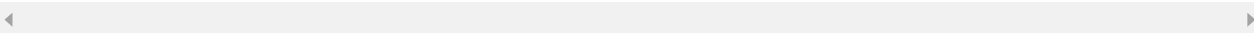
2020-03-15

辛苦老师

展开 ▾

作者回复:

:)



陈天柱

2020-03-14

看了今天老师的连接池文章后，才注意到jedisPool与redisTemplate的区别正是老师提到的连接池client是否内置池的区别，赞一个！针对课后的第二个问题，去查阅了MongoClient的源码，发现MongoClient的父类Mongo内置了ServerSessionPool池，所以MongoClient应该属于老师说的内置池那种实现，因此每次创建MongoClient都必须复用，不知道老师，我的理解对不对？

展开 ▾

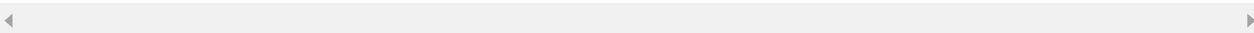
作者回复: 没错，官方文档里面是这么说的：

Typically you only create one MongoClient instance for a given MongoDB deployment (e.g. standalone, replica set, or a sharded cluster) and use it across your application. However, if you do create multiple instances:

All resource usage limits (e.g. max connections, etc.) apply per MongoClient instance.

To dispose of an instance, call MongoClient.close() to clean up resources.

其实复用不等于在任何情况下就只用一个，HttpClient也是同样的道理



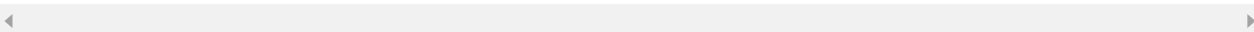
吴国帅

2020-03-14

哈哈 实名推荐 老哥牛逼

展开 ▾

作者回复: 如有收货，欢迎转发





Wiggle Wiggle

2020-03-14

对于问题1，第一个参数是客户端与连接池交互的关键参数，这个参数的取值与客户端的需求紧密相关。第二个参数是连接池与服务器交互的关键参数，这个参数的取值应该参考服务器的一般情况。

对于问题2，应该复用，为什么？因为我踩过坑.....短时间内新建大量client撑爆了mong...

展开 ∨

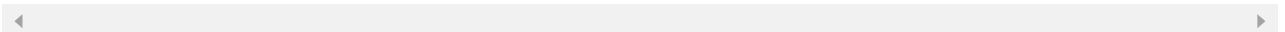
作者回复: 1. 是的，这两个参数需要分清楚功能

2.

Typically you only create one MongoClient instance for a given MongoDB deployment (e.g. standalone, replica set, or a sharded cluster) and use it across your application. However, if you do create multiple instances:

All resource usage limits (e.g. max connections, etc.) apply per MongoClient instance.

To dispose of an instance, call `MongoClient.close()` to clean up resources.



💬 2

