

## 40 | 谈谈Jetty性能调优的思路

2019-08-13 李号双

深入拆解Tomcat & Jetty

[进入课程 >](#)



讲述：李号双

时长 06:13 大小 5.71M



关于 Tomcat 的性能调优，前面我主要谈了工作经常会遇到的有关 JVM GC、监控、I/O 和线程池以及 CPU 的问题定位和调优，今天我们来看看 Jetty 有哪些调优的思路。


关于 Jetty 的性能调优，官网上给出了一些很好的建议，分为操作系统层面和 Jetty 本身的调优，我们将分别来看一看它们具体是怎么做的，最后再通过一个实战案例来学习一下如何确定 Jetty 的最佳线程数。

### 操作系统层面调优

对于 Linux 操作系统调优来说，我们需要加大一些默认的限制值，这些参数主要可以在 `/etc/security/limits.conf` 中或通过 `sysctl` 命令进行配置，其实这些配置对于 Tomcat 来说也是适用的，下面我来详细介绍一下这些参数。

## TCP 缓冲区大小


TCP 的发送和接收缓冲区最好加大到 16MB，可以通过下面的命令配置：

 复制代码

```
1 sysctl -w net.core.rmem_max = 16777216
2 sysctl -w net.core.wmem_max = 16777216
3 sysctl -w net.ipv4.tcp_rmem = "4096 87380 16777216"
4 sysctl -w net.ipv4.tcp_wmem = "4096 16384 16777216"
```


## TCP 队列大小

`net.core.somaxconn` 控制 TCP 连接队列的大小，默认值为 128，在高并发情况下明显不够用，会出现拒绝连接的错误。但是这个值也不能调得过高，因为过多积压的 TCP 连接会消耗服务端的资源，并且会造成请求处理的延迟，给用户带来不好的体验。因此我建议适当调大，推荐设置为 4096。

 复制代码

```
1 sysctl -w net.core.somaxconn = 4096
```


`net.core.netdev_max_backlog` 用来控制 Java 程序传入数据包队列的大小，可以适当调大。

 复制代码

```
1 sysctl -w net.core.netdev_max_backlog = 16384
2 sysctl -w net.ipv4.tcp_max_syn_backlog = 8192
3 sysctl -w net.ipv4.tcp_syncookies = 1
```

## 端口


如果 Web 应用程序作为客户端向远程服务器建立了很多 TCP 连接，可能会出现 TCP 端口不足的情况。因此最好增加使用的端口范围，并允许在 TIME\_WAIT 中重用套接字：

 复制代码

```
1 sysctl -w net.ipv4.ip_local_port_range = "1024 65535"
2 sysctl -w net.ipv4.tcp_tw_recycle = 1
```

## 文件句柄数


高负载服务器的文件句柄数很容易耗尽，这是因为系统默认值通常比较低，我们可以在 `/etc/security/limits.conf` 中为特定用户增加文件句柄数：

 复制代码

```
1 用户名 hard nofile 40000
2 用户名 soft nofile 40000
```


## 拥塞控制

Linux 内核支持可插拔的拥塞控制算法，如果要获取内核可用的拥塞控制算法列表，可以通过下面的命令：

 复制代码

```
1 sysctl net.ipv4.tcp_available_congestion_control
```

这里我推荐将拥塞控制算法设置为 `cubic`：

 复制代码

```
1 sysctl -w net.ipv4.tcp_congestion_control = cubic
```

## Jetty 本身的调优

Jetty 本身的调优，主要是设置不同类型的线程的数量，包括 `Acceptor` 和 `Thread Pool`。


## Acceptors

Acceptor 的个数 accepts 应该设置为大于等于 1，并且小于等于 CPU 核数。

## Thread Pool

限制 Jetty 的任务队列非常重要。默认情况下，队列是无限的！因此，如果在高负载下超过 Web 应用的处理能力，Jetty 将在队列上积压大量待处理的请求。并且即使负载高峰过去了，Jetty 也不能正常响应新的请求，这是因为仍然有很多请求在队列等着被处理。

因此对于一个高可靠性的系统，我们应该通过使用有界队列立即拒绝过多的请求（也叫快速失败）。那队列的长度设置成多大呢，应该根据 Web 应用的处理速度而定。比如，如果 Web 应用每秒可以处理 100 个请求，当负载高峰到来，我们允许一个请求可以在队列积压 60 秒，那么我们就可以把队列长度设置为  $60 \times 100 = 6000$ 。如果设置得太低，Jetty 将很快拒绝请求，无法处理正常的高峰负载，以下是配置示例：

 复制代码

```
1 <Configure id="Server" class="org.eclipse.jetty.server.Server">
2     <Set name="ThreadPool">
3         <New class="org.eclipse.jetty.util.thread.QueuedThreadPool">
4             <!-- specify a bounded queue -->
5             <Arg>
6                 <New class="java.util.concurrent.ArrayBlockingQueue">
7                     <Arg type="int">6000</Arg>
8                 </New>
9             </Arg>
10            <Set name="minThreads">10</Set>
11            <Set name="maxThreads">200</Set>
12            <Set name="detailedDump">false</Set>
13        </New>
14    </Set>
15 </Configure>
```


那如何配置 Jetty 的线程池中的线程数呢？跟 Tomcat 一样，你可以根据实际压测，如果 I/O 越密集，线程阻塞越严重，那么线程数就可以配置多一些。通常情况，增加线程数需要更多的内存，因此内存的最大值也要跟着调整，所以一般来说，Jetty 的最大线程数应该在 50 到 500 之间。

## Jetty 性能测试

接下来我们通过一个实验来测试一下 Jetty 的性能。我们可以在[这里](#)下载 Jetty 的 JAR 包。


```
#ls -l jetty-all-9.4.19.v20190610-uber.jar
-rw-r--r--@ 1 haosli 110503534 3622073 Aug  5 10:08 jetty-all-9.4.19.v20190610-uber.jar
```

第二步我们创建一个 Handler，这个 Handler 用来向客户端返回 “Hello World”，并实现一个 main 方法，根据传入的参数创建相应数量的线程池。

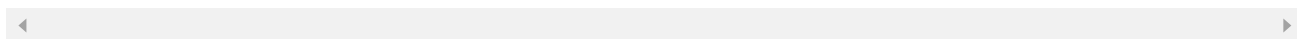
 复制代码

```
1 public class HelloWorld extends AbstractHandler {
2
3     @Override
4     public void handle(String target, Request baseRequest, HttpServletRequest request, I
5         response.setContentType("text/html; charset=utf-8");
6         response.setStatus(HttpServletResponse.SC_OK);
7         response.getWriter().println("<h1>Hello World</h1>");
8         baseRequest.setHandled(true);
9     }
10
11     public static void main(String[] args) throws Exception {
12         // 根据传入的参数控制线程池中最大线程数的大小
13         int maxThreads = Integer.parseInt(args[0]);
14         System.out.println("maxThreads:" + maxThreads);
15
16         // 创建线程池
17         QueuedThreadPool threadPool = new QueuedThreadPool();
18         threadPool.setMaxThreads(maxThreads);
19         Server server = new Server(threadPool);
20
21         ServerConnector http = new ServerConnector(server,
22             new HttpConnectionFactory(new HttpConfiguration()));
23         http.setPort(8000);
24         server.addConnector(http);
25
26         server.start();
27         server.join();
28     }
29 }
```


第三步，我们编译这个 Handler，得到 HelloWorld.class。

 复制代码

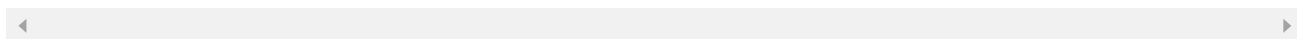
```
1 javac -cp jetty.jar HelloWorld.java
```




第四步，启动 Jetty server，并且指定最大线程数为 4。

 复制代码

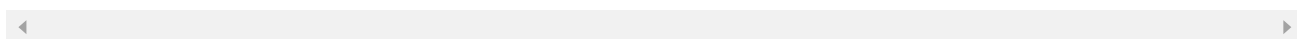
```
1 java -cp .:jetty.jar HelloWorld 4
```



第五步，启动压测工具 Apache Bench。关于 Apache Bench 的使用，你可以参考[这里](#)。

 复制代码

```
1 ab -n 200000 -c 100 http://localhost:8000/
```



上面命令的意思是向 Jetty server 发出 20 万个请求，开启 100 个线程同时发送。

经过多次压测，测试结果稳定以后，在 Linux 4 核机器上得到的结果是这样的：

```
Server Software:      Jetty(9.4.19.v20190610)
Server Hostname:      localhost
Server Port:          8000

Document Path:        /
Document Length:      317 bytes

Concurrency Level:    100
Time taken for tests:  9.990 seconds
Complete requests:    200000
Failed requests:      0
Non-2xx responses:    200000
Total transferred:    105600000 bytes
HTML transferred:     63400000 bytes
Requests per second:  20020.62 [#/sec] (mean)
Time per request:     4.995 [ms] (mean)
Time per request:     0.050 [ms] (mean, across all concurrent requests)
Transfer rate:        10323.13 [Kbytes/sec] received
```

```
Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1  24.8      0   1002
Processing:      0    4   2.7      4    205
Waiting:         0    4   2.7      4    205
Total:          0    5  25.0      4   1203
```

从上面的测试结果我们可以看到，20 万个请求在 9.99 秒内处理完成，RPS 达到了 20020。不知道你是否好奇，为什么我把最大线程数设置为 4 呢？是不是有点小？

别着急，接下来我们就试着逐步加大最大线程数，直到找到最佳值。下面这个表格显示了在其他条件不变的情况下，只调整线程数对 RPS 的影响。

maxThread	4	6	8	16	128	256
RPS	20020	23431	22571	21255	17938	15296

我们发现一个有意思的现象，线程数从 4 增加到 6，RPS 确实增加了。但是线程数从 6 开始继续增加，RPS 不但没有跟着上升，反而下降了，而且线程数越多，RPS 越低。

发生这个现象的原因是，测试机器的 CPU 只有 4 核，而我们测试的程序做得事情比较简单，没有 I/O 阻塞，属于 CPU 密集型程序。对于这种程序，最大线程数可以设置为比 CPU 核心稍微大一点点。那具体设置成多少是最佳值呢，我们需要根据实验里的步骤反复



测试。你可以看到在我们这个实验中，当最大线程数为 6，也就 CPU 核数的 1.5 倍时，性能达到最佳。

## 本期精华

今天我们首先学习了 Jetty 调优的基本思路，主要分为操作系统级别的调优和 Jetty 本身的调优，其中操作系统级别也适用于 Tomcat。接着我们通过一个实例来寻找 Jetty 的最佳线程数，在测试我们发现，对于 CPU 密集型应用，将最大线程数设置 CPU 核数的 1.5 倍是最佳的。因此，在我们的实际工作中，切勿将线程池直接设置得很大，因为程序所需要的线程数可能会比我们想象的要小。

## 课后思考

我在今天文章前面提到，Jetty 的最大线程数应该在 50 到 500 之间。但是我们的实验中测试发现，最大线程数为 6 时最佳，这是不是矛盾了？

不知道今天的内容你消化得如何？如果还有疑问，请大胆的在留言区提问，也欢迎你把你的课后思考和心得记录下来，与我和其他同学一起讨论。如果你觉得今天有所收获，欢迎你把它分享给你的朋友。



# 深入拆解 Tomcat & Jetty

从源码角度深度探索 Java 中间件

李号双

eBay 技术主管



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。



上一篇 39 | Tomcat进程占用CPU过高怎么办？

下一篇 41 | 热点问题答疑（4）：Tomcat和Jetty有哪些不同？

## 精选留言 (4)

写留言



-W.LI-

2019-08-13

1.课后习题不矛盾。老师课上说了实力是纯CPU密集型没有IO阻塞，这种情况下线程数比核数多一点就好。正式环境，要连接各种缓存，数据库，第三方调用都会IO阻塞。IO阻塞越多可开的线程数越多。

老师好!服务器分配的端口号只是服务监听的端口号，然后服务器作为客户端调用别的服...

展开

1

2



许童童

2019-08-13

但是我们的实验中测试发现，最大线程数为 6 时最佳，这是不是矛盾了？

不矛盾，老师已经说了，这个案例里面没有IO操作。

有IO操作的时候，用这个公式： $(\text{线程IO阻塞时间} + \text{线程CPU时间}) / \text{线程CPU时间}$

展开

1

1



逆流的鱼

2019-08-13

系统相关调节和servlet容器强相关？

展开

1

1



门窗小二

2019-08-13

测试中的最大线程数指的是接入线程类似netty的boss eventloop，50到500处理线程类似work eventloop！猜测是这样？

展开

1

1

