

21 | 总结：Tomcat和Jetty的高性能、高并发之道

2019-06-27 李号双

深入拆解Tomcat & Jetty

[进入课程 >](#)



讲述：李号双

时长 08:34 大小 7.86M



高性能程序就是高效的利用 CPU、内存、网络 and 磁盘等资源，在短时间内处理大量的请求。那如何衡量“短时间和大量”呢？其实就是两个关键指标：响应时间和每秒事务处理量（TPS）。

那什么是资源的高效利用呢？我觉得有两个原则：

1. **减少资源浪费。** 比如尽量避免线程阻塞，因为一阻塞就会发生线程上下文切换，就需要耗费 CPU 资源；再比如网络通信时数据从内核空间拷贝到 Java 堆内存，需要通过本地内存中转。
2. **当某种资源成为瓶颈时，用另一种资源来换取。** 比如缓存和对象池技术就是用内存换 CPU；数据压缩后再传输就是用 CPU 换网络。

Tomcat 和 Jetty 中用到了大量的高性能、高并发的设计，我总结了几点：I/O 和线程模型、减少系统调用、池化、零拷贝、高效的并发编程。下面我会详细介绍这些设计，你也可以将这些技术用到实际的工作中去。

I/O 和线程模型

I/O 模型的本质就是为了缓解 CPU 和外设之间的速度差。当线程发起 I/O 请求时，比如读写网络数据，网卡数据还没准备好，这个线程就会被阻塞，让出 CPU，也就是说发生了线程切换。而线程切换是无用功，并且线程被阻塞后，它持有内存资源并没有释放，阻塞的线程越多，消耗的内存就越大，因此 I/O 模型的目标就是尽量减少线程阻塞。Tomcat 和 Jetty 都已经抛弃了传统的同步阻塞 I/O，采用了非阻塞 I/O 或者异步 I/O，目的是业务线程不需要阻塞在 I/O 等待上。

除了 I/O 模型，线程模型也是影响性能和并发的关键点。Tomcat 和 Jetty 的总体处理原则是：

连接请求由专门的 Acceptor 线程组处理。

I/O 事件侦测也由专门的 Selector 线程组来处理。

具体的协议解析和业务处理可能交给线程池（Tomcat），或者交给 Selector 线程来处理（Jetty）。

将这些事情分开的好处是解耦，并且可以根据实际情况合理设置各部分的线程数。这里请你注意，线程数并不是越多越好，因为 CPU 核的个数有限，线程太多也处理不过来，会导致大量的线程上下文切换。

减少系统调用

其实系统调用是非常耗资源的一个过程，涉及 CPU 从用户态切换到内核态的过程，因此我们在编写程序的时候要有意识尽量避免系统调用。比如在 Tomcat 和 Jetty 中，系统调用最多的就是网络通信操作了，一个 Channel 上的 write 就是系统调用，为了降低系统调用的次数，最直接的方法就是使用缓冲，当输出数据达到一定的大小才 flush 缓冲区。Tomcat 和 Jetty 的 Channel 都带有输入输出缓冲区。

还有值得一提的是，Tomcat 和 Jetty 在解析 HTTP 协议数据时，都采取了**延迟解析**的策略，HTTP 的请求体（HTTP Body）直到用的时候才解析。也就是说，当 Tomcat 调用

Servlet 的 service 方法时，只是读取了解析了 HTTP 请求头，并没有读取 HTTP 请求体。

直到你的 Web 应用程序调用了 ServletRequest 对象的 getInputStream 方法或者 getParameter 方法时，Tomcat 才会去读取和解析 HTTP 请求体中的数据；这意味着如果你的应用程序没有调用上面那两个方法，HTTP 请求体的数据就不会被读取和解析，这样就省掉了一次 I/O 系统调用。

池化、零拷贝


关于池化和零拷贝，我在专栏前面已经详细讲了它们的原理，你可以回过头看看[专栏第 20 期](#)和[第 16 期](#)。其实池化的本质就是用内存换 CPU；而零拷贝就是不做无用功，减少资源浪费。

高效的并发编程

我们知道并发的过程中为了同步多个线程对共享变量的访问，需要加锁来实现。而锁的开销是比较大的，拿锁的过程本身就是个系统调用，如果锁没拿到线程会阻塞，又会发生线程上下文切换，尤其是大量线程同时竞争一把锁时，会浪费大量的系统资源。因此作为程序员，要有意识的尽量避免锁的使用，比如可以使用原子类 CAS 或者并发集合来代替。如果万不得已需要用到锁，也要尽量缩小锁的范围和锁的强度。接下来我们来看看 Tomcat 和 Jetty 如何做到高效的并发编程的。

缩小锁的范围

缩小锁的范围，其实就是不直接在方法上加 synchronized，而是使用细粒度的对象锁。

 复制代码

```
1 protected void startInternal() throws LifecycleException {
2
3     setState(LifecycleState.STARTING);
4
5     // 锁 engine 成员变量
6     if (engine != null) {
7         synchronized (engine) {
8             engine.start();
9         }
10    }
11
12    // 锁 executors 成员变量
```

```


13     synchronized (executors) {
14         for (Executor executor: executors) {
15             executor.start();
16         }
17     }
18
19     mapperListener.start();
20
21     // 锁 connectors 成员变量
22     synchronized (connectorsLock) {
23         for (Connector connector: connectors) {
24             // If it has already failed, don't try and start it
25             if (connector.getState() != LifecycleState.FAILED) {
26                 connector.start();
27             }
28         }
29     }
30 }

```

比如上面的代码是 Tomcat 的 StandardService 组件的启动方法，这个启动方法要启动三种子组件：engine、executors 和 connectors。它没有直接在方法上加锁，而是用了三把细粒度的锁，来分别用来锁三个成员变量。如果直接在方法上加 synchronized，多个线程执行到这个方法时需要排队；而在对象级别上加 synchronized，多个线程可以并行执行这个方法，只是在访问某个成员变量时才需要排队。

用原子变量和 CAS 取代锁

下面的代码是 Jetty 线程池的启动方法，它的主要功能就是根据传入的参数启动相应个数的线程。

 复制代码

```

1 private boolean startThreads(int threadsToStart)
2 {
3     while (threadsToStart > 0 && isRunning())
4     {
5         // 获取当前已经启动的线程数，如果已经够了就不需要启动了
6         int threads = _threadsStarted.get();
7         if (threads >= _maxThreads)
8             return false;
9
10        // 用 CAS 方法将线程数加一，请注意执行失败走 continue，继续尝试
11        if (!_threadsStarted.compareAndSet(threads, threads + 1))
12            continue;

```

```

13
14     boolean started = false;
15     try
16     {
17         Thread thread = new Thread(_runnable);
18         thread.setDaemon(isDaemon());
19         thread.setPriority(getThreadsPriority());
20         thread.setName(_name + "-" + thread.getId());
21         _threads.add(thread); // _threads 并发集合
22         _lastShrink.set(System.nanoTime()); // _lastShrink 是原子变量
23         thread.start();
24         started = true;
25         --threadsToStart;
26     }
27     finally
28     {
29         // 如果最终线程启动失败，还需要把线程数减一
30         if (!started)
31             _threadsStarted.decrementAndGet();
32     }
33 }
34 return true;
35 }

```

你可以看到整个函数的实现是一个**while 循环**，并且是**无锁的**。`_threadsStarted`表示当前线程池已经启动了多少个线程，它是一个原子变量 `AtomicInteger`，首先通过它的 `get` 方法拿到值，如果线程数已经达到最大值，直接返回。否则尝试用 CAS 操作将 `_threadsStarted` 的值加一，如果成功了意味着没有其他线程在改这个值，当前线程可以继续往下执行；否则走 `continue` 分支，也就是继续重试，直到成功为止。在这里当然你也可以使用锁来实现，但是我们的目的是无锁化。

并发容器的使用

`CopyOnWriteArrayList` 适用于读多写少的场景，比如 Tomcat 用它来“存放”事件监听器，这是因为监听器一般在初始化过程中确定后就基本不会改变，当事件触发时需要遍历这个监听器列表，所以这个场景符合读多写少的特征。

 复制代码

```


1 public abstract class LifecycleBase implements Lifecycle {
2
3     // 事件监听器集合
4     private final List<LifecycleListener> lifecycleListeners = new CopyOnWriteArrayList<

```

```
5
6     ...
7 }
```

volatile 关键字的使用

再拿 Tomcat 中的 LifecycleBase 作为例子，它里面的生命状态就是用 volatile 关键字修饰的。volatile 的目的是为了保证一个线程修改了变量，另一个线程能够读到这种变化。对于生命状态来说，需要在各个线程中保持是最新的值，因此采用了 volatile 修饰。

 复制代码

```
1 public abstract class LifecycleBase implements Lifecycle {
2
3     // 当前组件的生命状态，用 volatile 修饰
4     private volatile LifecycleState state = LifecycleState.NEW;
5
6 }
```

本期精华

高性能程序能够高效的利用系统资源，首先就是减少资源浪费，比如要减少线程的阻塞，因为阻塞会导致资源闲置和线程上下文切换，Tomcat 和 Jetty 通过合理的 I/O 模型和线程模型减少了线程的阻塞。

另外系统调用会导致用户态和内核态切换的过程，Tomcat 和 Jetty 通过缓存和延迟解析尽量减少系统调用，另外还通过零拷贝技术避免多余的数据拷贝。

高效的利用资源还包括另一层含义，那就是我们在系统设计的过程中，经常会用一种资源换取另一种资源，比如 Tomcat 和 Jetty 中使用的对象池技术，就是用内存换取 CPU，将数据压缩后再传输就是用 CPU 换网络。

除此之外，高效的并发编程也很重要，多线程虽然可以提高并发度，也带来了锁的开销，因此我们在实际编程过程中要尽量避免使用锁，比如可以用原子变量和 CAS 操作来代替锁。如果实在避免不了用锁，也要尽量减少锁的范围和强度，比如可以用细粒度的对象锁或者低强度的读写锁。Tomcat 和 Jetty 的代码也很好的实践了这一理念。

课后思考

今天的文章提到我们要有意识尽量避免系统调用，那你知道有哪些 Java API 会导致系统调用吗？

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。

 极客时间

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。

© 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 20 | 总结：Tomcat和Jetty中的对象池技术

下一篇 22 | 热点问题答疑（2）：内核如何阻塞与唤醒进程？

精选留言 (9)

写留言



QQ怪

2019-06-29

@正是那朵玫瑰同学，我认为老师所说的耗费cpu资源指的是线程阻塞和等待进行额外的上

下文切换，要理解其，先要知道上下文是什么，具体来说，一个线程被剥夺处理器的使用权而被暂停运行，就是“切出”；一个线程被选中占用处理器开始或者继续运行，就是“切入”。在这种切出切入的过程中，操作系统需要保存和恢复相应的进度信息，这个进度信息就是“上下文”了。至于系统开销具体发生在切换过程中的哪些具体环节，总...

作者回复: 🙏



👍 1



nightmare

2019-06-28

文件读写，socket网络编程，HeapByteBuffer，JNI都会涉及到系统调用



👍 1



正是那朵玫瑰

2019-06-27

老师有点疑问：

- 1、线程阻塞或者等待会发生上下文切换，耗费cpu资源，线程阻塞或者等待不是会让出cpu吗，应该是浪费cpu资源才对啊？我在压测的时候发现当有大量的线程在await状态时，cpu的利用率立马下降，应该是浪费cpu资源的！老师说会耗费cpu资源 怎么理解呢？
- 2、无锁固然是好的、但是无限重试是不是也会耗掉cpu资源，因为很多时候重试都是无...

💬 1

👍 1

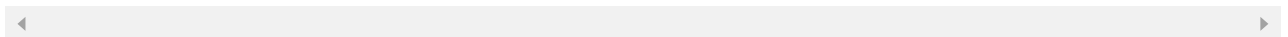


802.11

2019-06-29

刚才看copyonwrite的相关代码，很多地方用了数组拷贝，想问老师，是不是java api中所有带native关键字的方法都走了系统调用

作者回复: 不一定，native方法只是表明java调用了c函数，c函数一定都调用了系统API



802.11

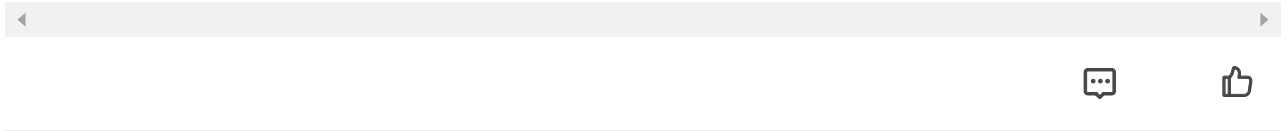
2019-06-29

standservice这个例子中，虽然没有加在方法上，在方法里面加是解决了粒度的问题，但

是加了3次和只在方法上加1次，系统调用层面上哪个更少哪个更多呢

作者回复: JDK1.6 以后 对锁的实现引入了大量的优化，如偏向锁、轻量级锁、自旋锁、适应性自旋锁、锁消除、锁粗化等技术来减少锁操作的开销。

其实就是避免每次synchronized操作都引起系统调用和上下文切换



Grubby

2019-06-29

tomcat的startInternal方法，理论上不是应该只有一个线程去调用吗？为什么要加synchronized？

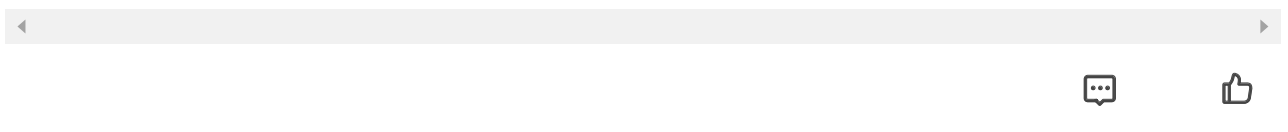


WL

2019-06-27

老师能不能指点一下在Http11Processor的service()中调用的Http11InputBuffer的parseHeaders()方法是怎么解析header的, 我今天看晕了, 没看懂是怎么将buffer中数据取出解析成header的

作者回复: 建议你先了解一下“状态机”，这是一种设计模式。

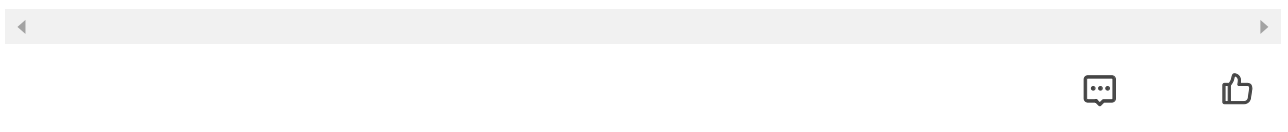


-W.LI-

2019-06-27

老师好!那就是get请求没有body。使用get请求把参数放url上少一次系统调用?

作者回复: 给你的思考点赞，url长度有限制，另外敏感数据不合适放url，否则是可以的。



刘章周

2019-06-27

网络编程socket,输入输出流, 创建线程thread,这些属于吧。

