

17 | Executor组件：Tomcat如何扩展Java线程池？

2019-06-18 李号双

深入拆解Tomcat & Jetty

[进入课程 >](#)



讲述：李号双

时长 09:33 大小 8.75M



在开发中我们经常会碰到“池”的概念，比如数据库连接池、内存池、线程池、常量池等。为什么需要“池”呢？程序运行的本质，就是通过使用系统资源（CPU、内存、网络、磁盘等）来完成信息的处理，比如在 JVM 中创建一个

对象实例需要消耗 CPU 和内存资源，如果你的程序需要频繁创建大量的对象，并且这些对象的存活时间短，就意味着需要进行频繁销毁，那么很有可能这部分代码会成为性能的瓶颈。

而“池”就是用来解决这个问题的，简单来说，对象池就是把用过的对象保存起来，等下一次需要这种对象的时候，直接从对象池中拿出来重复使用，避免频繁地创建和销毁。在 Java 中万物皆对象，线程也是一个对象，Java 线程是对操作系统线程的封装，创建 Java 线程也需要消耗系统资源，因此就有了线程池。JDK 中提供了线程池的默认实现，我们也可以通过扩展 Java 原生线程池来实现自己的线程池。


同样，为了提高处理能力和并发度，Web 容器一般会把处理请求的工作放到线程池里来执行，Tomcat 扩展了原生的 Java 线程池，来满足 Web 容器高并发的需求，下面我们就来学习一下 Java 线程池的原理，以及 Tomcat 是如何扩展 Java 线程池的。

Java 线程池

简单的说，Java 线程池里内部维护一个线程数组和一个任务队列，当任务处理不过来的时，就把任务放到队列里慢慢处理。

ThreadPoolExecutor

我们先来看看 Java 线程池核心类 `ThreadPoolExecutor` 的构造函数，你需要知道 `ThreadPoolExecutor` 是如何使用这些参数的，这是理解 Java 线程工作原理的关键。

 复制代码

```
1 public ThreadPoolExecutor(int corePoolSize,  
2                           int maximumPoolSize,  
3                           long keepAliveTime,  
4                           TimeUnit unit,  
5                           BlockingQueue<Runnable> workQ,  
6                           ThreadFactory threadFactory,  
7                           RejectedExecutionHandler hand
```

每次提交任务时，如果线程数还没达到核心线程数 **corePoolSize**，线程池就创建新线程来执行。当线程数达到 **corePoolSize** 后，新增的任务就放到工作队列 **workQueue** 里，而线程池中的线程则努力地从 **workQueue** 里拉活来干，也就是调用 `poll` 方法来获取任务。

如果任务很多，并且 **workQueue** 是个有界队列，队列可能会满，此时线程池就会紧急创建新的临时线程来救场，如果


```
5 }  
6  
7 public static ExecutorService newCachedThreadPool() {  
8     return new ThreadPoolExecutor(0, Integer.MAX_VALUE,  
9                                     60L, TimeUnit.SECONDS,  
10                                    new SynchronousQueue<  
11 }
```



从上面的代码你可以看到：

FixedThreadPool 有固定长度（`nThreads`）的线程数组，忙不过来时会把任务放到无限长的队列里，这是因为 **LinkedBlockingQueue** 默认是一个无界队列。

CachedThreadPool 的 `maximumPoolSize` 参数值是 `Integer.MAX_VALUE`，因此它对线程个数不做限制，忙不过来时无限创建临时线程，闲下来时再回收。它的任务队列是 **SynchronousQueue**，表明队列长度为 0。

Tomcat 线程池

跟 **FixedThreadPool**/**CachedThreadPool** 一样，Tomcat 的线程池也是一个定制版的 **ThreadPoolExecutor**。


定制版的 **ThreadPoolExecutor**

通过比较 `FixedThreadPool` 和 `CachedThreadPool`，我们发现它们传给 `ThreadPoolExecutor` 的参数有两个关键点：

是否限制线程个数。

是否限制队列长度。

对于 Tomcat 来说，这两个资源都需要限制，也就是说要对高并发进行控制，否则 CPU 和内存有资源耗尽的风险。因此 Tomcat 传入的参数是这样的：

 复制代码

```
1 // 定制版的任务队列
2 taskqueue = new TaskQueue(maxQueueSize);
3
4 // 定制版的线程工厂
5 TaskThreadFactory tf = new TaskThreadFactory(namePrefix
6
7 // 定制版的线程池
8 executor = new ThreadPoolExecutor(getMinSpareThreads(),
```

你可以看到其中的两个关键点：

Tomcat 有自己的定制版任务队列和线程工厂，并且可以限制任务队列的长度，它的最大长度是

maxQueueSize。

Tomcat 对线程数也有限制，设置了核心线程数 (minSpareThreads) 和最大线程池数 (maxThreads) 。

除了资源限制以外，Tomcat 线程池还定制自己的任务处理流程。我们知道 Java 原生线程池的任务处理逻辑比较简单：


1. 前 corePoolSize 个任务时，来一个任务就创建一个新线程。
2. 后面再来任务，就把任务添加到任务队列里让所有的线程去抢，如果队列满了就创建临时线程。
3. 如果总线程数达到 maximumPoolSize，**执行拒绝策略。**

Tomcat 线程池扩展了原生的 ThreadPoolExecutor，通过重写 execute 方法实现了自己的任务处理逻辑：

1. 前 corePoolSize 个任务时，来一个任务就创建一个新线程。
2. 再来任务的话，就把任务添加到任务队列里让所有的线程去抢，如果队列满了就创建临时线程。
3. 如果总线程数达到 maximumPoolSize，**则继续尝试把任务添加到任务队列中去。**

4. 如果缓冲队列也满了，插入失败，执行拒绝策略。

观察 Tomcat 线程池和 Java 原生线程池的区别，其实就是在第 3 步，Tomcat 在线程总数达到最大数时，不是立即执行拒绝策略，而是再尝试向任务队列添加任务，添加失败后再执行拒绝策略。那具体如何实现呢，其实很简单，我们来看一下 Tomcat 线程池的 execute 方法的核心代码。

 复制代码

```
1 public class ThreadPoolExecutor extends java.util.concu
2
3     ...
4
5     public void execute(Runnable command, long timeout, T
6         submittedCount.incrementAndGet();
7         try {
8             // 调用 Java 原生线程池的 execute 去执行任务
9             super.execute(command);
10        } catch (RejectedExecutionException rx) {
11            // 如果总线程数达到 maximumPoolSize, Java 原生线
12            if (super.getQueue() instanceof TaskQueue) {
13                final TaskQueue queue = (TaskQueue)super.
14                try {
15                    // 继续尝试把任务放到任务队列中去
16                    if (!queue.force(command, timeout, ur
17                        submittedCount.decrementAndGet();
18                    // 如果缓冲队列也满了，插入失败，执行
19                    throw new RejectedExecutionExcept
20                }
21            }
22        }
```




```
23     }  
24 }
```

从这个方法你可以看到，Tomcat 线程池的 `execute` 方法会调用 Java 原生线程池的 `execute` 去执行任务，如果总线程数达到 `maximumPoolSize`，Java 原生线程池的 `execute` 方法会抛出 `RejectedExecutionException` 异常，但是这个异常会被 Tomcat 线程池的 `execute` 方法捕获到，并继续尝试把这个任务放到任务队列中去；如果任务队列也满了，再执行拒绝策略。

定制版的任务队列

细心的你有没有发现，在 Tomcat 线程池的 `execute` 方法最开始有这么一行：

 复制代码

```
1 submittedCount.incrementAndGet();
```

这行代码的意思把 `submittedCount` 这个原子变量加一，并且在任务执行失败，抛出拒绝异常时，将这个原子变量减一：


```
1 submittedCount.decrementAndGet();
```

其实 Tomcat 线程池是用这个变量 `submittedCount` 来维护已经提交到了线程池，但是还没有执行完的任务个数。Tomcat 为什么要维护这个变量呢？这跟 Tomcat 的定制版的任务队列有关。Tomcat 的任务队列 `TaskQueue` 扩展了 Java 中的 `LinkedBlockingQueue`，我们知道 `LinkedBlockingQueue` 默认情况下长度是没有限制的，除非给它一个 `capacity`。因此 Tomcat 给了它一个 `capacity`，`TaskQueue` 的构造函数中有个整型的参数 `capacity`，`TaskQueue` 将 `capacity` 传给父类 `LinkedBlockingQueue` 的构造函数。

```
1 public class TaskQueue extends LinkedBlockingQueue<Runn
2
3     public TaskQueue(int capacity) {
4         super(capacity);
5     }
6     ...
7 }
```

这个 capacity 参数是通过 Tomcat 的 maxQueueSize 参数来设置的，但问题是默认情况下 maxQueueSize 的值是 Integer.MAX_VALUE，等于没有限制，这样就带来一个问题：当前线程数达到核心线程数之后，再来任务的话线程池会把任务添加到任务队列，并且总是会成功，这样永远不会有机会创建新线程了。

为了解决这个问题，TaskQueue 重写了 LinkedBlockingQueue 的 offer 方法，在合适的时机返回 false，返回 false 表示任务添加失败，这时线程池会创建新的线程。那什么是合适的时机呢？请看下面 offer 方法的核心源码：

 复制代码

```
1 public class TaskQueue extends LinkedBlockingQueue<Runn
2
3     ...
4     @Override
5     // 线程池调用任务队列的方法时，当前线程数肯定已经大于核心线程数
6     public boolean offer(Runnable o) {
7
8         // 如果线程数已经到了最大值，不能创建新线程了，只能把
9         if (parent.getPoolSize() == parent.getMaximumPool
10             return super.offer(o);
11
12         // 执行到这里，表明当前线程数大于核心线程数，并且小于
13         // 表明是可以创建新线程的，那到底要不要创建呢？分两种
14
15         //1. 如果已提交的任务数小于当前线程数，表示还有空闲线
```

```
16         if (parent.getSubmittedCount() <= (parent.getPoolSi
17             return super.offer(o);
18
19         //2. 如果已提交的任务数大于当前线程数，线程不够用了，
20         if (parent.getPoolSize() < parent.getMaximumPoolSiz
21             return false;
22
23         // 默认情况下总是把任务添加到任务队列
24         return super.offer(o);
25     }
26
27 }
```



从上面的代码我们看到，只有当前线程数大于核心线程数、小于最大线程数，并且已提交的任务个数大于当前线程数时，也就是说线程不够用了，但是线程数又没达到极限，才会去创建新的线程。这就是为什么 Tomcat 需要维护已提交任务数这个变量，它的目的就是**在任务队列的长度无限制的情况下，让线程池有机会创建新的线程。**

当然默认情况下 Tomcat 的任务队列是没有限制的，你可以通过设置 `maxQueueSize` 参数来限制任务队列的长度。

本期精华

池化的目的是为了**避免频繁地创建和销毁对象，减少对系统资源的消耗**。Java 提供了默认的线程池实现，我们也可以

扩展 Java 原生的线程池来实现定制自己的线程池，Tomcat 就是这么做的。Tomcat 扩展了 Java 线程池的核心类 `ThreadPoolExecutor`，并重写了它的 `execute` 方法，定制了自己的任务处理流程。同时 Tomcat 还实现了定制版的任务队列，重写了 `offer` 方法，使得在任务队列长度无限制的情况下，线程池仍然有机会创建新的线程。

课后思考

请你再仔细看看 Tomcat 的定制版任务队列 `TaskQueue` 的 `offer` 方法，它多次调用了 `getPoolSize` 方法，但是这个方法是有锁的，锁会引起线程上下文切换而损耗性能，请问这段代码可以如何优化呢？

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。

© 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 16 | AprEndpoint组件：Tomcat APR提高I/O性能的...

下一篇 18 | 新特性：Tomcat如何支持WebSocket?

精选留言 (16)

写留言



W.T

2019-06-19

给李老师点赞👍解析得非常到位!



👍 2



世纪猛男

2019-06-18

关于今日的思考题 getPoolSize. 用Volatile去修饰一个变量不可行，因为变更过程，会基于之前的pool size，无法做到原子操作。用atomic 也不合适 并发量高的时候会导致 大量的更新失败，持续消耗CPU。所以还不如加锁来的痛快。请教老师的想法

作者回复: 可以加锁，但是没有必要多次调用，调一次把结果存起来就行。



👍 2



迎风劲草

2019-06-18

老师，核心线程如果超过keeplive时间，是否也会回收？还有如果我的队列中还有等待执行的runable,这时候kill进程，时候需要等到所有runable被执行要，进程才结束吗？

作者回复: 1.可以调用ThreadPoolExecutor的这个方法来指定是否回收核心线程：

```
public void allowCoreThreadTimeOut(boolean value)
```

2.kill进程会立即退出，内核会负责清理这个进程的所有资源。



永光

2019-06-18

观察 Tomcat 线程池和 Java 原生线程池的区别，其实就是在第 3 步，Tomcat 在线程总数达到最大数时，不是立即执行拒绝策略，而是再尝试向任务队列添加任务，添加失败后再执行拒绝策略。

问题： ...

作者回复: 有可能第一次尝试放队列是满的，失败，再尝试创建临时线程，也满了，但是这个过程中，队列中的任务可能被临时线程消费了一部分，再往队列中送可能会成功。



Geek_7c24a2

2019-06-24

李老师恕我能力有限，execute里的offer方法不是父类里的么？



东方奇骥



2019-06-23

我们实际项目里一般禁止使用无界队列，因为业务量大的时候，无界队列可能会引起OOM



slowChef

2019-06-21

李老师说下这几种进程池都分别适用于什么样的使用场景吗？



Geek_7c24a2

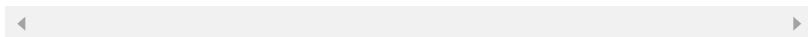
2019-06-21

李老师说下TaskQueue里的offer方法在什么地方被调用了嘛？多谢了

作者回复: 就是Tomcat定制版的线程池

`org.apache.tomcat.util.threads.ThreadPoolExecutor`

的execute方法里。



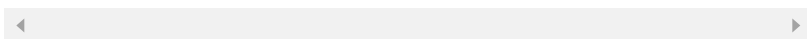


-W.LI-

2019-06-20

李老师好。我有个问题，原生队列是在队列满时新建线程处理。然后当线程达到最大线程数的时候，不就是队列已满，线程也开满了么。Tomcat捕获异常后再往队列里放一次，只是为了做后的努力争取不丢任务么？

作者回复: 对的



QQ怪

2019-06-19

看了下源码getpoolsize方法使用的是可重入锁 ReentrantLock锁，可以使用atomic+volatile方法来优化，但要用好，不然会事半功倍，不过最后我还是觉得加锁好吧，atomic最基本的++操作还好，复杂的还是考虑加锁来解决，降低点性能来换取稳定和可靠应该是更好的选择😁



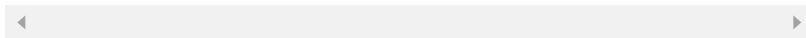
z.l

2019-06-18

感觉直接读workers.size()就可以了么，因为创建线程和销毁线程的方法都加锁了，而且是同一把锁，不懂为啥

getPoolSize()方法还要额外加锁？

作者回复: 是的，这个地方Tomcat的实现可以简化。



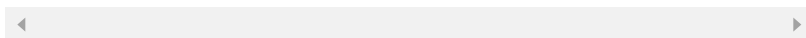
陆离

2019-06-18

corePoolSize的有什么设置的策略吗？

需要和CPU个数联系起来吗？

作者回复: 如何是纯粹的CPU密集型应用，corepoolsize可以设置为CPU核数



Stalary

2019-06-18

volatile+atomic把



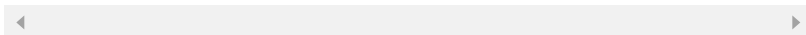
Geek_8eedf1



2019-06-18

看明白了😓，老师，学习 Java 线程池有哪些需要注意的点呢？

作者回复: 主要还是学会用，根据场景选用合适的线程池类型。



nightmare

2019-06-18

用一个volatile的变量在第一次获取的时候接收一个core pool size就行了，一般设置之后也不会变化



Liam

2019-06-18

可以用atomic原子变量替换锁吧

