tomcat对JDK线程池的扩展

2015-10-03 17:03 wangmj 0 阅读 85

为了减少线程创建和销毁的消耗，通常会使用线程池，在JDK中，默认集成了高效的几种线程池：在tomcat中的线程池并没有自定义实现，而是利用了JDK的线程池，继承了上述的特性，并基于自己的逻辑进行了扩展；

1.JDK线程配置属性初始化

tomcat的配置文件中，配置<Service>标签中的<Executor>属性：

```
<Service name="Catalina">
    <Executor name="tomcatThreadPool" namePrefix="catalina-exec-"
        maxThreads="150" minSpareThreads="4"/>
    <Connector port="8080" protocol="HTTP/1.1"
               connectionTimeout="20000" executor="tomcatThreadPool"
               redirectPort="8443" />
```

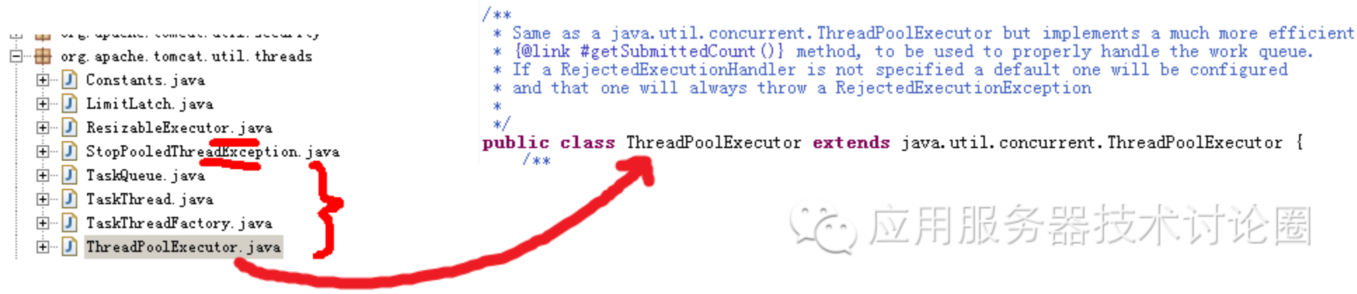

这个Executor，在tomcat各种书籍中叫做执行器，它的实际的作用就是线程池，对于Connector如何与这个Executor进行关联的，它是通过<Connector>元素中的executor属性指向你所配置的<Executor>元素的；对于这个配置，Executor元素对应的是org.apache.catalina.Executor接口，Tomcat中的标准实现是org.apache.catalina.core.StandardThreadExecutor；

tomcat的配置管理会通过digest进行解析这个配置文件，然后将你所填写的属性初始化到这个StandardThreadExecutor类中；但是StandardThreadExecutor类并不是具体对应的线程池的实现，它仅仅起到的是tomcat线程池的初始化和销毁的工作，上述的工作依托于tomcat自身独有的lifecycle接口：

```
public class StandardThreadExecutor extends LifecycleMBeanBase
        implements Executor, ResizableExecutor {

    /**
     * Start the component and implement the requirements
     * of {@link org.apache.catalina.util.LifecycleBase#startInternal()}.
     *
     * @exception LifecycleException if this component detects a fatal error
     *  that prevents this component from being used
     */
    @Override
    protected void startInternal() throws LifecycleException {

        taskqueue = new TaskQueue(maxQueueSize);
        TaskThreadFactory tf = new TaskThreadFactory(namePrefix,daemon,getThreadPriority());
        executor = new ThreadPoolExecutor(getMinSpareThreads(), getMaxThreads(), maxIdleTime, TimeUnit.MILLISECONDS,tas
        executor.setThreadRenewalDelay(threadRenewalDelay);
        if (prestartminSpareThreads) {
            executor.prestartAllCoreThreads();
        }
        task                ecutor);
        setS                e.STARTING);
    }

    /**
     * Stop the component and implement the requirements
     * of {@link org.apache.catalina.util.LifecycleBase#stopInternal()}.
     *
     * @exception LifecycleException if this component detects a fatal error
     *  that needs to be reported
     */
    @Override
    protected void stopInternal() throws LifecycleException {

        setState(LifecycleState.STOPPING);
        if ( executor != null ) executor.shutdownNow();
        executor = null;
        taskqueue = null;
    }
```


在组件启动的时候，初始化ThreadExecutor线程池，在组件关闭的时候，销毁ThreadExecutor线程池；之所以这么做，是因为tomcat的StandardXXX 的各种组件，仅仅是Standard引擎的实现，tomcat可以允许你对整个内部实现进行替换；

注意上述的ThreadExecutor线程池，这还不是JDK的线程池，仅仅与其同名而已，它的真正实现是org.apache.tomcat.util.threads.ThreadExecutor；这个类就是tomcat的线程池，它继承于JDK的ThreadExecutor线程池，对其进行了扩展；




```
/**
 * Same as a java.util.concurrent.ThreadPoolExecutor but implements a much more efficient
 * {@link #getSubmittedCount()} method, to be used to properly handle the work queue.
 * If a RejectedExecutionHandler is not specified a default one will be configured
 * and that one will always throw a RejectedExecutionException
 *
 */
public class ThreadPoolExecutor extends java.util.concurrent.ThreadPoolExecutor {
    /**
```

上述的流程是配置了默认的<Executor>元素；

而如果不配置的话，对应的Connector会创建一个默认的线程池，

```
public void createExecutor() {
    internalExecutor = true;
    TaskQueue taskqueue = new TaskQueue();
    TaskThreadFactory tf = new TaskThreadFactory(getName() + "-exec-", daemon, getThreadPriority());
    executor = new ThreadPoolExecutor(getMinSpareThreads(), getMaxThreads(), 60, TimeUnit.SECONDS, taskqueue, tf);
    taskqueue.setParent( (ThreadPoolExecutor) executor);
}
```


2.线程ThreadLocal泄露清理

tomcat的ThreadExecutor其中一个重要的作用，就是对线程的ThreadLocal缓存的变量进行清理；

为什么ThreadLocal要进行清理呢？如果是一个简单的main函数的话，那么在这个主线程中使用ThreadLocal缓存在程序结束之后，自动就随着JVM退出而消亡了；

如果是开启的一个线程，这个线程中使用了ThreadLocal缓存，线程退出，这种情况这块内存仍旧会进行回收；

但是，线程池的线程是重复利用的，很有可能会在某处使用了ThreadLocal缓存，但是忘记进行remove掉了，这种在线程池中是很致命的；

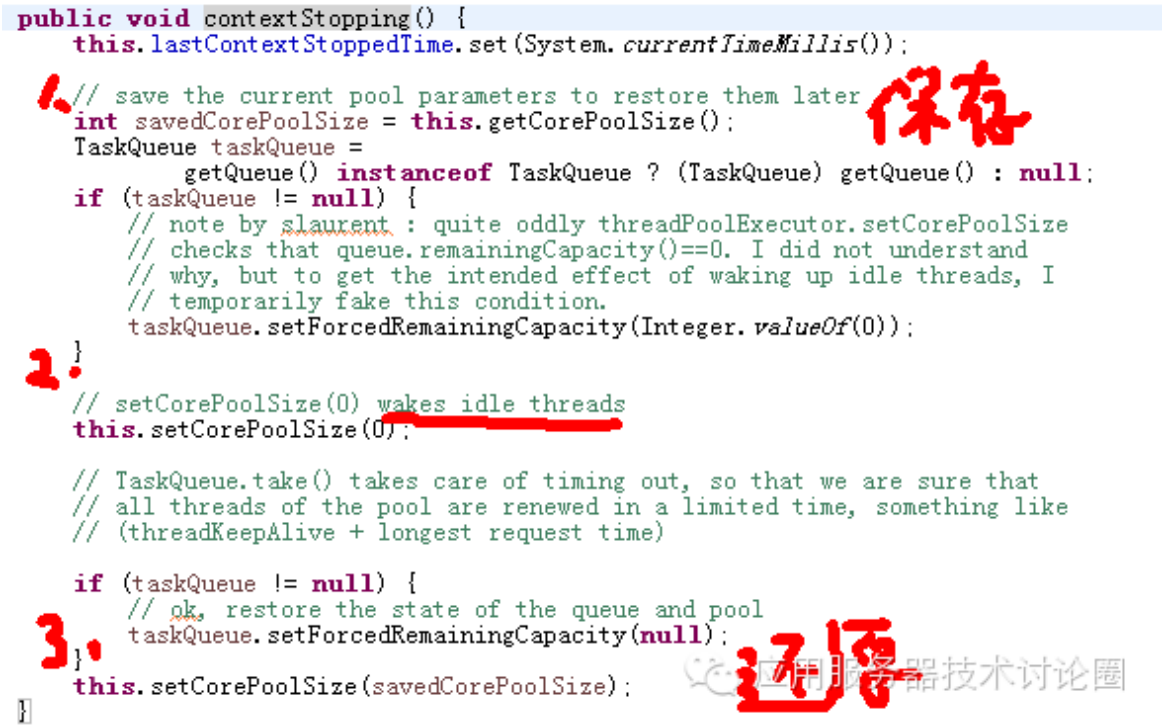
即使，tomcat这种服务器程序非常严谨，但是tomcat也为了以防万一，做了一个预防措施：

```
/**
 * <p>
 * A {@link LifecycleListener} that triggers the renewal of threads in Executor
 * pools when a {@link Context} is being stopped to avoid thread-local related
 * memory leaks.
 * </p>
 * <p>
 * Note : active threads will be renewed one by one when they come back to the
 * pool after executing their task, see
 * {@link org.apache.tomcat.util.threads.ThreadPoolExecutor}.afterExecute().
 * </p>
 *
 * This listener must be declared in server.xml to be active.
 *
 */
public class ThreadLocalLeakPreventionListener implements LifecycleListener,
        ContainerListener {


    @Override
    public void lifecycleEvent(LifecycleEvent event) {
        try {
            Lifecycle lifecycle = event.getLifecycle();
            if (Lifecycle.AFTER_START_EVENT.equals(event.getType()) &&
                    lifecycle instanceof Server) {
                // when the server starts, we register ourself as listener for
                // all context
                // as well as container event listener so that we know when new
                // Context are deployed
                Server server = (Server) lifecycle;
                registerListenersForServer(server);
            }

            if (Lifecycle.BEFORE_STOP_EVENT.equals(event.getType()) &&
                    lifecycle instanceof Server) {
                // Server is shutting down, so thread pools will be shut down so
                // there is no need to clean the threads
                serverStopping = true;
            }

            if (Lifecycle.AFTER_STOP_EVENT.equals(event.getType()) &&
                    lifecycle instanceof Context) {
                stopIdleThreads((Context) lifecycle);
            }
        } catch (Exception e) {
            String msg =
                sm.getString(
                    "threadLocalLeakPreventionListener.lifecycleEvent.error",
                    event);
            log.error(msg, e);
        }
    }

    /**
     * Updates each ThreadPoolExecutor with the current time, which is the time
     * when a context is being stopped.
     *
     * @param context
     *            the context being stopped, used to discover all the Connectors
     *            of its parent Service.
     */
    private void stopIdleThreads(Context context) {
        if (serverStopping) return;

        if (!(context instanceof StandardContext) ||
            !((StandardContext) context).getRenewThreadsWhenStoppingContext()) {
            log.debug("Not renewing threads when the context is stopping. "
                    + "It is not configured to do so.");
            return;
```

重新让线程上下文renew一下

触发的时机是每一个应用stop停止或者卸载的时候，也就是上述的AFTER_STOP_EVENT事件发生调用的stopIdleThread方法，实质是调用tomcat扩展的JDK线程池中的contextStopping方法，这个是将空闲线程重新renew一下；

那么，tomcat的ThreadExecutor是如何做到将现有的空闲线程进行清理的呢？我们来看看ThreadExecutor类中的contextStopping方法具体实现：




```
public void contextStopping() {
    this.lastContextStoppedTime.set(System.currentTimeMillis());

    // save the current pool parameters to restore them later
    int savedCorePoolSize = this.getCorePoolSize();
    TaskQueue taskQueue =
            getQueue() instanceof TaskQueue ? (TaskQueue) getQueue() : null;
    if (taskQueue != null) {
        // note by slaurent : quite oddly threadPoolExecutor.setCorePoolSize
        // checks that queue.remainingCapacity()==0. I did not understand
        // why, but to get the intended effect of waking up idle threads, I
        // temporarily fake this condition.
        taskQueue.setForcedRemainingCapacity(Integer.valueOf(0));
    }

    // setCorePoolSize(0) wakes idle threads
    this.setCorePoolSize(0);

    // TaskQueue.take() takes care of timing out, so that we are sure that
    // all threads of the pool are renewed in a limited time, something like
    // (threadKeepAlive + longest request time)

    if (taskQueue != null) {
        // ok, restore the state of the queue and pool
        taskQueue.setForcedRemainingCapacity(null);
    }
    this.setCorePoolSize(savedCorePoolSize);
}
```

a. 首先先拿到JDK线程池中标准线程的数目（标准线程池可以理解为一个线程池的阈值，超出它的话，会通过keepalive时间对idle线程进行时间控制）

这一步是为了缓存；

b. 设置标准线程数为0；

c.还原原来的标准线程的数目；

这其中第二步，标准线程的数目设置为0，最为关键，这部分的代码在JDK的线程池中，

```
/**
 * Sets the core number of threads.  This overrides any value set
 * in the constructor.  If the new value is smaller than the
 * current value, excess existing threads will be terminated when
 * they next become idle.  If larger, new threads will, if needed,
 * be started to execute any queued tasks.
 *
 * @param corePoolSize the new core size
 * @throws IllegalArgumentException if {@code corePoolSize < 0}
 * @see #getCorePoolSize
 */
public void setCorePoolSize(int corePoolSize) {
    if (corePoolSize < 0)
        throw new IllegalArgumentException();
    int delta = corePoolSize - this.corePoolSize;
    this.corePoolSize = corePoolSize;
    if (workerCountOf(ctl.get()) > corePoolSize)
        interruptIdleWorkers();
    else if (delta > 0) {
        // We don't really know how many new threads are "needed".
        // As a heuristic, prestart enough new workers (up to new
        // core size) to handle the current number of tasks in
        // queue, but stop if queue becomes empty while doing so.
        int k = Math.min(delta, workQueue.size());
        while (k-- > 0 && addWorker(null, true)) {
            if (workQueue.isEmpty())
                break;
        }
    }
}

private void interruptIdleWorkers(boolean onlyOne) {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        for (Worker w : workers) {
            Thread t = w.thread;
            if (!t.isInterrupted() && w.tryLock()) {
                try {
                    t.interrupt();
                } catch (SecurityException ignore) {
                } finally {
                    w.unlock();
                }
            }
            if (onlyOne)
                break;
        }
    } finally {
        mainLock.unlock();
    }
}
```


它所做的是将线程池中idle的空闲线程进行interrupt打断，这样空闲线程就会消失；

同时，随着业务客户端的使用，线程数不够，线程池会重新开启thread；

通过上述的方式，来完成线程的renew的工作，顺便将ThreadLocal进行了一下清理；

值得注意的一点是，上述的做法清理不了正在给其他应用服务的线程的ThreadLocal，但是会将idle的线程重新renew一下；

如果此刻，tomcat负载不是很多的话，基本上大多数的线程都会被renew；

对于上述的线程池renew的过程，tomcat的ThreadExecutor线程池通过threadRenewDelay参数进行调配和控制的

threadRenewalDelay

这个属性是扩展来的，它的原版解释是：

(long) If a ThreadLocalLeakPreventionListener is configured, it will notify this executor about stopped contexts. After a context is stopped, threads in the pool are renewed. To avoid renewing all threads at the same time, this option sets a delay between renewal of any 2 threads. The value is in ms, default value is 1000 ms. If value is negative, threads are not renewed.

Tomcat定义了监听器，它的目的是阻止ThreadLocal泄露。

当有个一个Context组件（web应用）停止stopped后，这个监听器会去通知Executor，执行器就会将池里的线程全部清空，再重新创建。

为了阻止所有的线程在同一时间重建，设置了这样一个延时在两个线程之间。

如果此值是负数，线程不renew。

3.submittedCount计数

```
/**
 * The number of tasks submitted but not yet finished. This includes tasks
 * in the queue and tasks that have been handed to a worker thread but the
 * latter did not start executing the task yet.
 * This number is always greater or equal to {@link #getActiveCount()}.
 */
private final AtomicInteger submittedCount = new AtomicInteger(0);
```


在JDK的线程池中，ActiveCount就是指正在工作的线程，而在queue中排队的任务其实也是没有完成的，这个数字在tomcat的线程监控中也有实际的意义；

因此，tomcat的线程池专门做了这样的一个属性进行监控，其实现也很简单；

实现JDK线程池的回调函数

```java
@Override
protected void afterExecute(Runnable r, Throwable t) {
    submittedCount.decrementAndGet();

    if (t == null) {
        stopCurrentThreadIfNeeded();
    }
}

/**
 * Executes the given command at some time in the future.  The command
 * may execute in a new thread, in a pooled thread, or in the calling
 * thread, at the discretion of the <tt>Executor</tt> implementation.
 * If no threads are available, it will be added to the work queue.
 * If the work queue is full, the system will wait for the specified
 * time and it throw a RejectedExecutionException if the queue is still
 * full after that.
 *
 * @param command the runnable task
 * @throws RejectedExecutionException if this task cannot be
 * accepted for execution - the queue is full
 * @throws NullPointerException if command or unit is null
 */
public void execute(Runnable command, long timeout, TimeUnit unit) {
    submittedCount.incrementAndGet();
    try {
        super.execute(command);
    } catch (RejectedExecutionException rx) {
        if (super.getQueue() instanceof TaskQueue) {
            final TaskQueue queue = (TaskQueue)super.getQueue();
            try {
                if (!queue.force(command, timeout, unit)) {
                    submittedCount.decrementAndGet();
                    throw new RejectedExecutionException("Queue capacity is full.");
                }
            } catch (InterruptedException x) {
                submittedCount.decrementAndGet();
                throw new RejectedExecutionException(x);
            }
        } else {
            submittedCount.decrementAndGet();
            throw rx;
        }

    }
}
```


覆盖了execute方法，传递过来一个任务，那么就相当于没有完成的任务；而afterExecute是JDK线程池中的每一个task完成的回调方法，进行这个属性的减少；而这个int，是在大量并发请求下的原子操作，因此需要Integer转化为原子类，以防出现并发问题；**总结：**tomcat的线程池实际上就是利用了JDK的线程池，只不过在其基础上对属性通过xml可配，并对未完成的task和线程泄露等功能进行扩展；

东方通加码大数据业务 拟募资8亿收购微智信业 (/html/308/201504/206211355/1.html)

玩转Netty – 从Netty3升级到Netty4 (/html/308/201504/206233287/1.html)

金蝶中间件2015招聘来吧！Come on！ (/html/308/201505/206307460/1.html)

GlassFish 4.1 发布，J2EE 应用服务器 (/html/308/201505/206323120/1.html)

Tomcat对keep-alive的实现逻辑 (/html/308/201505/206357679/1.html)

猜您喜欢

SQL优化（五） PostgreSQL （递归）CTE 通用表表达式 (/html/166/201604/2648885469/1.html)

十个经典Android开源APP项目 (/html/428/201407/201155637/1.html)

【浙江】海正药业诚聘软件质量工程师 (/html/326/201410/200814192/1.html)

如何不用那么担心成为一个坏程序员 (/html/376/201606/2650116896/1.html)

创业初期，如何搭建起一个优秀的团队 (/html/414/201609/2247483892/1.html)