原 ReentrantLock和内部锁的性能对比

ReentrantLock是jdk5引入的新的锁机制，它与内部锁（synchronize）相同的并发性和内存语义，比如可重入加锁语义。在中等或者更高负荷下，ReentrantLock有更好的性能，并且拥有可轮询和可定时的请求锁等高级功能。这个程序简单对比了ReentrantLock公平锁、ReentrantLock非公平锁以及内部锁的性能，从结果上看，非公平的 ReentrantLock表现最好。内部锁也仅仅是实现统计意义上的公平，结果也比公平的ReentrantLock好上很多。这个程序仅仅是计数，启动N个线程，对同一个Counter进行递增，显然，这个递增操作需要同步以保证原子性，采用不同的锁来实现同步，然后查看结果。

Counter接口：

```
package net.rubyeye.concurrency.chapter13;

public interface Counter {
    public long getValue();

    public void increment();

}
```

然后，首先使用我们熟悉的synchronize来实现同步：

```
package net.rubyeye.concurrency.chapter13;

public class SynchronizeBenchmark implements Counter {
    private long count = 0;

    public long getValue() {
        return count;
    }
```

```
    public synchronized void increment() {
        count++;
    }
}
```

采用ReentrantLock的版本，切记要在finally中释放锁，这是与synchronize使用方式最大的不同，内部锁jvm会自动帮你释放锁，而ReentrantLock需要你自己来处理。

```
package net.rubyeye.concurrency.chapter13;

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class ReentrantLockBeanchmark implements Counter {

    private volatile long count = 0;

    private Lock lock;

    public ReentrantLockBeanchmark() {
        // 使用非公平锁，true就是公平锁
        lock = new ReentrantLock(false);
    }

    public long getValue() {
        // TODO Auto-generated method stub
        return count;
    }

    public void increment() {
        lock.lock();
        try {
            count++;
        } finally {
            lock.unlock();
        }
    }
```

```
}
```

写一个测试程序，使用CyclicBarrier来等待所有任务线程创建完毕以及所有任务线程计算完成，清单如下：

```
package net.rubyeye.concurrency.chapter13;

import java.util.concurrent.CyclicBarrier;

public class BenchmarkTest {
    private Counter counter;

    private CyclicBarrier barrier;

    private int threadNum;

    public BenchmarkTest(Counter counter, int threadNum) {
        this.counter = counter;
        barrier = new CyclicBarrier(threadNum + 1); //关卡计数=线程数+1
        this.threadNum = threadNum;
    }

    public static void main(String args[]) {
        new BenchmarkTest(new SynchronizeBenchmark(), 5000).test();
        //new BenchmarkTest(new ReentrantLockBeanchmark(), 5000).test();
        //new BenchmarkTest(new ReentrantLockBeanchmark(), 5000).test();
    }

    public void test() {
        try {
            for (int i = 0; i < threadNum; i++) {
                new TestThread(counter).start();
            }
            long start = System.currentTimeMillis();
            barrier.await(); // 等待所有任务线程创建
            barrier.await(); // 等待所有任务计算完成
            long end = System.currentTimeMillis();
```

```
            System.out.println("count value:" + counter.getValue());
            System.out.println("花费时间:" + (end - start) + "毫秒");
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    class TestThread extends Thread {
        private Counter counter;

        public TestThread(final Counter counter) {
            this.counter = counter;
        }

        public void run() {
            try {
                barrier.await();
                for (int i = 0; i < 100; i++)
                    counter.increment();
                barrier.await();
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

分别测试一下，

将启动的线程数限定为500，结果为：
公平ReentrantLock：　　210 毫秒
非公平ReentrantLock：　39 毫秒
内部锁：　　39 毫秒

将启动的线程数限定为1000，结果为：
公平ReentrantLock：　　640 毫秒

非公平ReentrantLock： 81 毫秒
内部锁: 60 毫秒

线程数不变，test方法中的循环增加到1000次，结果为:
公平ReentrantLock: 16715 毫秒
非公平ReentrantLock： 168 毫秒
内部锁: 639 毫秒

将启动的线程数增加到2000,结果为:
公平ReentrantLock: 1100 毫秒
非公平ReentrantLock: 125 毫秒
内部锁: 130 毫秒

将启动的线程数增加到3000,结果为:
公平ReentrantLock: 2461 毫秒
非公平ReentrantLock: 254 毫秒
内部锁: 307 毫秒

启动5000个线程，结果如下:
公平ReentrantLock: 6154 毫秒
非公平ReentrantLock: 623 毫秒
内部锁: 720 毫秒

非公平ReentrantLock和内部锁的差距，在jdk6上应该缩小了，据说jdk6的内部锁机制进行了调整。

发表于 @ 2009年01月20日 11:18:00 | 评论(0) | 举报 | 收藏

旧一篇:java.util.concurrent系列文章--(2)JDK1.5 锁机制 | 新一篇:java.util.concurrent.CyclicBarrier

用 户 名: huapuyu3

发表评论