调试排错 - Java线程Dump分析

Thread Dump是非常有用的诊断Java应用问题的工具。

Thread Dump介绍

什么是*Thread Dump*

Thread Dump是非常有用的诊断Java应用问题的工具。每一个Java虚拟机都有及时生成所有线程在某一点状态的thread-dump的能力，虽然各个 Java虚拟机打印的thread dump略有不同，但是 大多都提供了当前活动线程的快照，及JVM中所有Java线程的堆栈跟踪信息，堆栈信息一般包含完整的类名及所执行的方法，如果可能的话还有源代码的行数。

*Thread Dump*特点

- 能在各种操作系统下使用；
- 能在各种Java应用服务器下使用；
- 能在生产环境下使用而不影响系统的性能；
- 能将问题直接定位到应用程序的代码行上；

*Thread Dump*抓取

一般当服务器挂起，崩溃或者性能低下时，就需要抓取服务器的线程堆栈（Thread Dump）用于后续的分析。在实际运行中，往往一次 dump的信息，还不足以确认问题。为了反映线程状态的动态变化，需要接连多次做thread dump，每次间隔10-20s，建议至少产生三次 dump信息，如果每次 dump都指向同一个问题，我们才确定问题的典型性。

- 操作系统命令获取ThreadDump

```
ps -ef | grep java
kill -3 <pid>
```

注意：

> 一定要谨慎, 一步不慎就可能让服务器进程被杀死。kill -9 命令会杀死进程。

- JVM 自带的工具获取线程堆栈

```
jps 或 ps -ef | grep java （获取PID）
jstack [-l ] <pid> | tee -a jstack.log（获取ThreadDump）
```

Thread Dump分析

*Thread Dump*信息

- 头部信息：时间，JVM信息

```
2011-11-02 19:05:06
Full thread dump Java HotSpot(TM) Server VM (16.3-b01 mixed mode):
```

- 线程INFO信息块：

```
1. "Timer-0" daemon prio=10 tid=0xac190c00 nid=0xaef in Object.wait() [0xae77d000]
# 线程名称：Timer-0；线程类型：daemon；优先级：10，默认是5；
# JVM线程id：tid=0xac190c00，JVM内部线程的唯一标识（通过java.lang.Thread.getId()获取，通常用自增方式实现）。
# 对应系统线程id（NativeThread ID）：nid=0xaef，和top命令查看的线程pid对应，不过一个是10进制，一个是16进制。（通过命令：top -H -p pid，可以查看该进程的所有线程信息）
# 线程状态：in Object.wait()；
# 起始栈地址：[0xae77d000]，对象的内存地址，通过JVM内存查看工具，能够看出线程是在哪儿个对象上等待；
2.  java.lang.Thread.State: TIMED_WAITING (on object monitor)
3.  at java.lang.Object.wait(Native Method)
4.  -waiting on <0xb3885f60> (a java.util.TaskQueue)     # 继续wait
5.  at java.util.TimerThread.mainLoop(Timer.java:509)
6.  -locked <0xb3885f60> (a java.util.TaskQueue)         # 已经locked
7.  at java.util.TimerThread.run(Timer.java:462)
Java thread statck trace：是上面2-7行的信息。到目前为止这是最重要的数据，Java stack trace提供了大部分信息来精确定位问题根源。
```

- Java thread statck trace详解：

堆栈信息应该逆向解读：程序先执行的是第7行，然后是第6行，依次类推。

```
- locked <0xb3885f60> (a java.util.ArrayList)
- waiting on <0xb3885f60> (a java.util.ArrayList)
```

也就是说对象先上锁，锁住对象0xb3885f60，然后释放该对象锁，进入waiting状态。为啥会出现这样的情况呢？看看下面的java代码示例，就会明白：

```
synchronized(obj) {
   .........
   obj.wait();
   .........
}
```

如上，线程的执行过程，先用 synchronized 获得了这个对象的 Monitor（对应于 locked <0xb3885f60> ）。当执行到 obj.wait()，线程即放弃了 Monitor的所有权，进入 “wait set”队列（对应于 waiting on <0xb3885f60> ）。

在堆栈的第一行信息中，进一步标明了线程在代码级的状态，例如：

```
java.lang.Thread.State: TIMED_WAITING (parking)
```

解释如下：

```
|blocked|
```

```
> This thread tried to enter asynchronized block, but the lock was taken by another thread. This thread isblocked until the lock gets released.

|blocked (on thin lock)|

> This is the same state asblocked, but the lock in question is a thin lock.

|waiting|

> This thread calledObject.wait() on an object. The thread will remain there until some otherthread sends a notification to that object.

|sleeping|

> This thread calledjava.lang.Thread.sleep().

|parked|

> This thread calledjava.util.concurrent.locks.LockSupport.park().

|suspended|

> The thread's execution wassuspended by java.lang.Thread.suspend() or a JVMTI agent call.
```

*Thread*状态分析

线程的状态是一个很重要的东西，因此thread dump中会显示这些状态，通过对这些状态的分析，能够得出线程的运行状况，进而发现可能存在的问题。**线程的状态在Thread.State这个枚举类型中定义**：

```
public enum State
{
        /**
         * Thread state for a thread which has not yet started.
         */
        NEW,

        /**
         * Thread state for a runnable thread.  A thread in the runnable
         * state is executing in the Java virtual machine but it may
         * be waiting for other resources from the operating system
         * such as processor.
         */
        RUNNABLE,

        /**
         * Thread state for a thread blocked waiting for a monitor lock.
         * A thread in the blocked state is waiting for a monitor lock
         * to enter a synchronized block/method or
         * reenter a synchronized block/method after calling
         * {@link Object#wait() Object.wait}.
         */
        BLOCKED,

        /**
         * Thread state for a waiting thread.
         * A thread is in the waiting state due to calling one of the
         * following methods:
```

```
         * <ul>
         *   <li>{@link Object#wait() Object.wait} with no timeout</li>
         *   <li>{@link #join() Thread.join} with no timeout</li>
         *   <li>{@link LockSupport#park() LockSupport.park}</li>
         * </ul>
         *
         * <p>A thread in the waiting state is waiting for another thread to
         * perform a particular action.
         *
         * For example, a thread that has called <tt>Object.wait()</tt>
         * on an object is waiting for another thread to call
         * <tt>Object.notify()</tt> or <tt>Object.notifyAll()</tt> on
         * that object. A thread that has called <tt>Thread.join()</tt>
         * is waiting for a specified thread to terminate.
         */
        WAITING,

        /**
         * Thread state for a waiting thread with a specified waiting time.
         * A thread is in the timed waiting state due to calling one of
         * the following methods with a specified positive waiting time:
         * <ul>
         *   <li>{@link #sleep Thread.sleep}</li>
         *   <li>{@link Object#wait(long) Object.wait} with timeout</li>
         *   <li>{@link #join(long) Thread.join} with timeout</li>
         *   <li>{@link LockSupport#parkNanos LockSupport.parkNanos}</li>
         *   <li>{@link LockSupport#parkUntil LockSupport.parkUntil}</li>
         * </ul>
         */
        TIMED_WAITING,

        /**
         * Thread state for a terminated thread.
         * The thread has completed execution.
         */
        TERMINATED;
    }
```

- NEW：

每一个线程，在堆内存中都有一个对应的Thread对象。Thread t = new Thread();当刚刚在堆内存中创建Thread对象，还没有调用t.start()方法之前，线程就处在NEW状态。在这个状态上，线程与普通的java对象没有什么区别，就仅仅是一个堆内存中的对象。

- RUNNABLE：

该状态表示线程具备所有运行条件，在运行队列中准备操作系统的调度，或者正在运行。 这个状态的线程比较正常，但如果线程长时间停留在在这个状态就不正常了，这说明线程运行的时间很长（存在性能问题），或者是线程一直得不得执行的机会（存在线程饥饿的问题）。

- BLOCKED：

线程正在等待获取java对象的监视器(也叫内置锁)，即线程正在等待进入由synchronized保护的方法或者代码块。synchronized用来保证原子性，任意时刻最多只能由一个线程进入该临界区域，其他线程只能排队等待。

- WAITING：

处在该线程的状态，正在等待某个事件的发生，只有特定的条件满足，才能获得执行机会。而产生这个特定的事件，通常都是另一个线程。也就是说，如果不发生特定的事件，那么处在该状态的线程一直等待，不能获取执行的机会。比如：

A线程调用了obj对象的obj.wait()方法，如果没有线程调用obj.notify或obj.notifyAll，那么A线程就没有办法恢复运行； 如果A线程调用了LockSupport.park()，没有别的线程调用LockSupport.unpark(A)，那么A没有办法恢复运行。
TIMED_WAITING：

J.U.C中很多与线程相关类，都提供了限时版本和不限时版本的API。TIMED_WAITING意味着线程调用了限时版本的API，正在等待时间流逝。当等待时间过去后，线程一样可以恢复运行。如果线程进入了WAITING状态，一定要特定的事件发生才能恢复运行；而处在TIMED_WAITING的线程，如果特定的事件发生或者是时间流逝完毕，都会恢复运行。

- TERMINATED：

线程执行完毕，执行完run方法正常返回，或者抛出了运行时异常而结束，线程都会停留在这个状态。这个时候线程只剩下Thread对象了，没有什么用了。

关键状态分析

- **Wait on condition**：The thread is either sleeping or waiting to be notified by another thread.

该状态说明它在等待另一个条件的发生，来把自己唤醒，或者干脆它是调用了 sleep(n)。

此时线程状态大致为以下几种：

```
java.lang.Thread.State: WAITING (parking)：一直等那个条件发生；
java.lang.Thread.State: TIMED_WAITING (parking或sleeping)：定时的，那个条件不到来，也将定时唤醒自己。
```

- **Waiting for Monitor Entry 和 in Object.wait()**：The thread is waiting to get the lock for an object (some other thread may be holding the lock). This happens if two or more threads try to execute synchronized code. Note that the lock is always for an object and not for individual methods.

在多线程的JAVA程序中，实现线程之间的同步，就要说说 Monitor。**Monitor是Java中用以实现线程之间的互斥与协作的主要手段，它可以看成是对象或者Class的锁。每一个对象都有，也仅有一个 Monitor 。**下面这个图，描述了线程和 Monitor之间关系，以及线程的状态转换图：

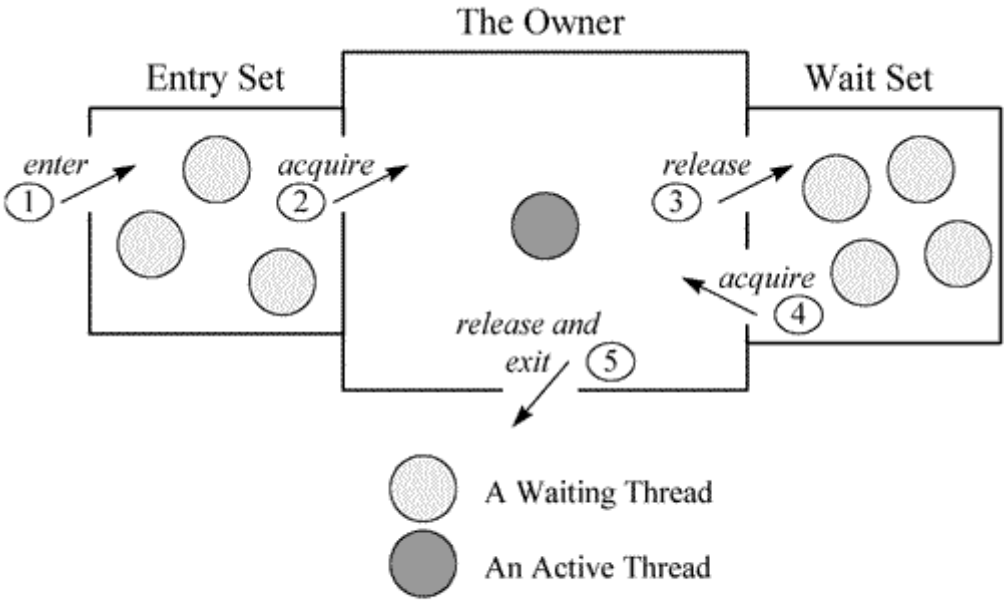

Figure 20-1. A Java monitor.

如上图，每个Monitor在某个时刻，只能被一个线程拥有，**该线程就是 “ActiveThread”，而其它线程都是 “Waiting Thread”，分别在两个队列“Entry Set”和“Wait Set”里等候。**在“Entry Set”中等待的线程状态是“Waiting for monitor entry”，而在“Wait Set”中等待的线程状态是“in Object.wait()”。

先看“Entry Set”里面的线程。我们称被 synchronized保护起来的代码段为临界区。**当一个线程申请进入临界区时，它就进入了“Entry Set”队列。**对应的 code就像：

```
synchronized(obj) {
   .........
}
```

这时有两种可能性：

- 该 monitor不被其它线程拥有， Entry Set里面也没有其它等待线程。本线程即成为相应类或者对象的 Monitor的 Owner，执行临界区的代码。
- 该 monitor被其它线程拥有，本线程在 Entry Set队列中等待。

在第一种情况下，线程将处于 “Runnable”的状态，而第二种情况下，线程 DUMP会显示处于 “waiting for monitor entry”。如下：

```
"Thread-0" prio=10 tid=0x08222eb0 nid=0x9 waiting for monitor entry [0xf927b000..0xf927bdb8]
at testthread.WaitThread.run(WaitThread.java:39)
- waiting to lock <0xef63bf08> (a java.lang.Object)
- locked <0xef63beb8> (a java.util.ArrayList)
at java.lang.Thread.run(Thread.java:595)
```

临界区的设置，是为了保证其内部的代码执行的原子性和完整性。但是因为临界区在任何时间只允许线程串行通过，这和我们多线程的程序的初衷是相反的。**如果在多线程的程序中，大量使用 synchronized，或者不适当的使用了它，会造成大量线程在临界区的入口等待，造成系统的性能大幅下降。**如果在线程 DUMP中发现了这个情况，应该审查源码，改进程序。

再看“Wait Set”里面的线程。**当线程获得了 Monitor，进入了临界区之后，如果发现线程继续运行的条件没有满足，它则调用对象（一般就是被 synchronized 的对象）的 wait() 方法，放弃 Monitor，进入 “Wait Set”队列。只有当别的线程在该对象上调用了 notify() 或者 notifyAll()，“Wait Set”队列中线程才得到机会去竞争**，但是只有一个线程获得对象的Monitor，恢复到运行态。在 “Wait Set”中的线程， DUMP中表现为： in Object.wait()。如下：

```
"Thread-1" prio=10 tid=0x08223250 nid=0xa in Object.wait() [0xef47a000..0xef47aa38]
 at java.lang.Object.wait(Native Method)
 - waiting on <0xef63beb8> (a java.util.ArrayList)
 at java.lang.Object.wait(Object.java:474)
 at testthread.MyWaitThread.run(MyWaitThread.java:40)
 - locked <0xef63beb8> (a java.util.ArrayList)
 at java.lang.Thread.run(Thread.java:595)
综上，一般CPU很忙时，则关注runnable的线程，CPU很闲时，则关注waiting for monitor entry的线程。
```

- **JDK 5.0 的 Lock**

上面提到如果 synchronized和 monitor机制运用不当，可能会造成多线程程序的性能问题。在 JDK 5.0中，引入了 Lock机制，从而使开发者能更灵活的开发高性能的并发多线程程序，可以替代以往 JDK中的 synchronized和 Monitor的 机制。但是，**要注意的是，因为 Lock类只是一个普通类，JVM无从得知 Lock对象的占用情况，所以在线程 DUMP中，也不会包含关于 Lock的信息**， 关于死锁等问题，就不如用 synchronized的编程方式容易识别。

关键状态示例

- **显示BLOCKED状态**

```
package jstack;

public class BlockedState
{
    private static Object object = new Object();

    public static void main(String[] args)
    {
        Runnable task = new Runnable() {

            @Override
            public void run()
            {
                synchronized (object)
                {
                    long begin = System.currentTimeMillis();

                    long end = System.currentTimeMillis();

                    // 让线程运行5分钟,会一直持有object的监视器
                    while ((end - begin) <= 5 * 60 * 1000)
                    {

                    }
                }
            }
        };

        new Thread(task, "t1").start();
        new Thread(task, "t2").start();
    }
}
```

先获取object的线程会执行5分钟，**这5分钟内会一直持有object的监视器，另一个线程无法执行处在BLOCKED状态**:

```
Full thread dump Java HotSpot(TM) Server VM (20.12-b01 mixed mode):

"DestroyJavaVM" prio=6 tid=0x00856c00 nid=0x1314 waiting on condition [0x00000000]
java.lang.Thread.State: RUNNABLE

"t2" prio=6 tid=0x27d7a800 nid=0x1350 waiting for monitor entry [0x2833f000]
java.lang.Thread.State: BLOCKED (on object monitor)
     at jstack.BlockedState$1.run(BlockedState.java:17)
     - waiting to lock <0x1cfcdc00> (a java.lang.Object)
     at java.lang.Thread.run(Thread.java:662)

"t1" prio=6 tid=0x27d79400 nid=0x1338 runnable [0x282ef000]
 java.lang.Thread.State: RUNNABLE
     at jstack.BlockedState$1.run(BlockedState.java:22)
     - locked <0x1cfcdc00> (a java.lang.Object)
     at java.lang.Thread.run(Thread.java:662)
```

通过thread dump可以看到：**t2线程确实处在BLOCKED (on object monitor)。waiting for monitor entry 等待进入synchronized保护的区域。**

- **显示WAITING状态**

```
package jstack;

public class WaitingState
{
    private static Object object = new Object();

    public static void main(String[] args)
    {
        Runnable task = new Runnable() {

            @Override
            public void run()
            {
                synchronized (object)
                {
                    long begin = System.currentTimeMillis();
                    long end = System.currentTimeMillis();

                    // 让线程运行5分钟,会一直持有object的监视器
                    while ((end - begin) <= 5 * 60 * 1000)
                    {
                        try
                        {
                            // 进入等待的同时,会进入释放监视器
                            object.wait();
                        } catch (InterruptedException e)
                        {
                            e.printStackTrace();
                        }
                    }
                }
            }
        };

        new Thread(task, "t1").start();
        new Thread(task, "t2").start();
    }
}
```

```
Full thread dump Java HotSpot(TM) Server VM (20.12-b01 mixed mode):

"DestroyJavaVM" prio=6 tid=0x00856c00 nid=0x1734 waiting on condition [0x00000000]
java.lang.Thread.State: RUNNABLE

"t2" prio=6 tid=0x27d7e000 nid=0x17f4 in Object.wait() [0x2833f000]
java.lang.Thread.State: WAITING (on object monitor)
     at java.lang.Object.wait(Native Method)
     - waiting on <0x1cfcdc00> (a java.lang.Object)
     at java.lang.Object.wait(Object.java:485)
     at jstack.WaitingState$1.run(WaitingState.java:26)
     - locked <0x1cfcdc00> (a java.lang.Object)
     at java.lang.Thread.run(Thread.java:662)

"t1" prio=6 tid=0x27d7d400 nid=0x17f0 in Object.wait() [0x282ef000]
java.lang.Thread.State: WAITING (on object monitor)
```

```
    at java.lang.Object.wait(Native Method)
    - waiting on <0x1cfcdc00> (a java.lang.Object)
    at java.lang.Object.wait(Object.java:485)
    at jstack.WaitingState$1.run(WaitingState.java:26)
    - locked <0x1cfcdc00> (a java.lang.Object)
    at java.lang.Thread.run(Thread.java:662)
```

可以发现t1和t2都处在WAITING (on object monitor)，进入等待状态的原因是调用了in Object.wait()。通过J.U.C包下的锁和条件队列，也是这个效果，大家可以自己实践下。

- **显示TIMED_WAITING状态**

```
package jstack;

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

public class TimedWaitingState
{
    // java的显示锁,类似java对象内置的监视器
    private static Lock lock = new ReentrantLock();

    // 锁关联的条件队列(类似于object.wait)
    private static Condition condition = lock.newCondition();

    public static void main(String[] args)
    {
        Runnable task = new Runnable() {

            @Override
            public void run()
            {
                // 加锁,进入临界区
                lock.lock();

                try
                {
                    condition.await(5, TimeUnit.MINUTES);
                } catch (InterruptedException e)
                {
                    e.printStackTrace();
                }

                // 解锁,退出临界区
                lock.unlock();
            }
        };

        new Thread(task, "t1").start();
        new Thread(task, "t2").start();
    }
}
```

```
Full thread dump Java HotSpot(TM) Server VM (20.12-b01 mixed mode):

"DestroyJavaVM" prio=6 tid=0x00856c00 nid=0x169c waiting on condition [0x00000000]
```

```
java.lang.Thread.State: RUNNABLE

"t2" prio=6 tid=0x27d7d800 nid=0xc30 waiting on condition [0x2833f000]
java.lang.Thread.State: TIMED_WAITING (parking)
     at sun.misc.Unsafe.park(Native Method)
     - parking to wait for  <0x1cfce5b8> (a
java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject)
     at java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:196)
     at
java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchr
onizer.java:2116)
     at jstack.TimedWaitingState$1.run(TimedWaitingState.java:28)
     at java.lang.Thread.run(Thread.java:662)

"t1" prio=6 tid=0x280d0c00 nid=0x16e0 waiting on condition [0x282ef000]
java.lang.Thread.State: TIMED_WAITING (parking)
     at sun.misc.Unsafe.park(Native Method)
     - parking to wait for  <0x1cfce5b8> (a
java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject)
     at java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:196)
     at
java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchr
onizer.java:2116)
     at jstack.TimedWaitingState$1.run(TimedWaitingState.java:28)
     at java.lang.Thread.run(Thread.java:662)
```

可以看到t1和t2线程都处在java.lang.Thread.State: TIMED_WAITING (parking)，这个parking代表是调用的JUC下的工具类，而不是java默认的监视器。

案例分析

问题场景

- **CPU飙高，load高，响应很慢**
  1. 一个请求过程中多次dump；
  2. 对比多次dump文件的runnable线程，如果执行的方法有比较大变化，说明比较正常。如果在执行同一个方法，就有一些问题了；
- **查找占用CPU最多的线程**
  1. 使用命令：top -H -p pid（pid为被测系统的进程号），找到导致CPU高的线程ID，对应thread dump信息中线程的nid，只不过一个是十进制，一个是十六进制；
  2. 在thread dump中，根据top命令查找的线程id，查找对应的线程堆栈信息；
- **CPU使用率不高但是响应很慢**

进行dump，查看是否有很多thread struck在了i/o、数据库等地方，定位瓶颈原因；

- **请求无法响应**

多次dump，对比是否所有的runnable线程都一直在执行相同的方法，如果是的，恭喜你，锁住了！

死锁

死锁经常表现为程序的停顿，或者不再响应用户的请求。从操作系统上观察，对应进程的CPU占用率为零，很快会从top或prstat的输出中消失。

比如在下面这个示例中，是个较为典型的死锁情况：

```
"Thread-1" prio=5 tid=0x00acc490 nid=0xe50 waiting for monitor entry [0x02d3f000
..0x02d3fd68]
at deadlockthreads.TestThread.run(TestThread.java:31)
- waiting to lock <0x22c19f18> (a java.lang.Object)
- locked <0x22c19f20> (a java.lang.Object)

"Thread-0" prio=5 tid=0x00accdb0 nid=0xdec waiting for monitor entry [0x02cff000
..0x02cff9e8]
at deadlockthreads.TestThread.run(TestThread.java:31)
- waiting to lock <0x22c19f20> (a java.lang.Object)
- locked <0x22c19f18> (a java.lang.Object)
```

在 JAVA 5中加强了对死锁的检测。**线程 Dump中可以直接报告出 Java级别的死锁**，如下所示：

```
Found one Java-level deadlock:
=============================
"Thread-1":
waiting to lock monitor 0x0003f334 (object 0x22c19f18, a java.lang.Object),
which is held by "Thread-0"

"Thread-0":
waiting to lock monitor 0x0003f314 (object 0x22c19f20, a java.lang.Object),
which is held by "Thread-1"
```

热锁

热锁，也往往是导致系统性能瓶颈的主要因素。其表现特征为：**由于多个线程对临界区，或者锁的竞争**，可能出现：

- **频繁的线程的上下文切换**：从操作系统对线程的调度来看，当线程在等待资源而阻塞的时候，操作系统会将之切换出来，放到等待的队列，当线程获得资源之后，调度算法会将这个线程切换进去，放到执行队列中。
- **大量的系统调用**：因为线程的上下文切换，以及热锁的竞争，或者临界区的频繁的进出，都可能导致大量的系统调用。
- **大部分CPU开销用在“系统态”**：线程上下文切换，和系统调用，都会导致 CPU在 “系统态 ”运行，换而言之，虽然系统很忙碌，但是CPU用在 “用户态 ”的比例较小，应用程序得不到充分的 CPU资源。
- **随着CPU数目的增多，系统的性能反而下降。**因为CPU数目多，同时运行的线程就越多，可能就会造成更频繁的线程上下文切换和系统态的CPU开销，从而导致更糟糕的性能。

上面的描述，都是一个 scalability（可扩展性）很差的系统的表现。从整体的性能指标看，由于线程热锁的存在，程序的响应时间会变长，吞吐量会降低。

那么，怎么去了解 “热锁 ”出现在什么地方呢?

一个重要的方法是 结合操作系统的各种工具观察系统资源使用状况，以及收集Java线程的DUMP信息，看线程都阻塞在什么方法上，了解原因，才能找到对应的解决方法。

JVM重要线程

JVM运行过程中产生的一些比较重要的线程罗列如下：

线程名称	解释说明
Attach Listener	Attach Listener 线程是负责接收到外部的命令，而对该命令进行执行的并把结果返回给发送者。通常我们会用一些命令去要求JVM给我们一些反馈信息，如：java -version、jmap、jstack等等。 如果该线程在JVM启动的时候没有初始化，那么，则会在用户第一次执行JVM命令时，得到启动。
Signal Dispatcher	前面提到Attach Listener线程的职责是接收外部JVM命令，当命令接收成功后，会交给signal dispather线程去进行分发到各个不同的模块处理命令，并且返回处理结果。signal dispather线程也是在第一次接收外部JVM命令时，进行初始化工作。
CompilerThread0	用来调用JITing，实时编译装卸class 。 通常，JVM会启动多个线程来处理这部分工作，线程名称后面的数字也会累加，例如：CompilerThread1。
Concurrent Mark-Sweep GC Thread	并发标记清除垃圾回收器（就是通常所说的CMS GC）线程， 该线程主要针对于老年代垃圾回收。ps：启用该垃圾回收器，需要在JVM启动参数中加上：-XX:+UseConcMarkSweepGC。
DestroyJavaVM	执行main()的线程，在main执行完后调用JNI中的 jni_DestroyJavaVM() 方法唤起 DestroyJavaVM 线程，处于等待状态，等待其它线程（Java线程和Native线程）退出时通知它卸载JVM。每个线程退出时，都会判断自己当前是否是整个JVM中最后一个非deamon线程，如果是，则通知DestroyJavaVM 线程卸载JVM。
Finalizer Thread	这个线程也是在main线程之后创建的，其优先级为10，主要用于在垃圾收集前，调用对象的finalize()方法；关于Finalizer线程的几点：1) 只有当开始一轮垃圾收集时，才会开始调用finalize()方法；因此并不是所有对象的finalize()方法都会被执行；2) 该线程也是daemon线程，因此如果虚拟机中没有其他非daemon线程，不管该线程有没有执行完finalize()方法，JVM也会退出；3) JVM在垃圾收集时会将失去引用的对象包装成Finalizer对象（Reference的实现），并放入ReferenceQueue，由Finalizer线程来处理；最后将该Finalizer对象的引用置为null，由垃圾收集器来回收；4) JVM为什么要单独用一个线程来执行finalize()方法呢？如果JVM的垃圾收集线程自己来做，很有可能由于在finalize()方法中误操作导致GC线程停止或不可控，这对GC线程来说是一种灾难；
Low Memory Detector	这个线程是负责对可使用内存进行检测，如果发现可用内存低，分配新的内存空间。
Reference Handler	JVM在创建main线程后就创建Reference Handler线程，其优先级最高，为10，它主要用于处理引用对象本身（软引用、弱引用、虚引用）的垃圾回收问题 。
VM Thread	这个线程就比较牛b了，是JVM里面的线程母体，根据hotspot源码（vmThread.hpp）里面的注释，它是一个单个的对象（最原始的线程）会产生或触发所有其他的线程，这个单个的VM线程是会被其他线程所使用来做一些VM操作（如：清扫垃圾等）。