**jstack Dump 日志文件中的线程状态**

dump 文件里，值得关注的线程状态有：

1. 死锁，**Deadlock（重点关注）**
2. 执行中，Runnable
3. 等待资源，**Waiting on condition（重点关注）**
4. 等待获取监视器，**Waiting on monitor entry（重点关注）**
5. 暂停，Suspended
6. 对象等待中，Object.wait() 或 TIMED\_WAITING
7. 阻塞，**Blocked（重点关注）**
8. 停止，Parked

下面我们先从第一个例子开始分析，然后再列出不同线程状态的含义以及注意事项，最后再补充两个实例。

**综合示范一：Waiting to lock 和 Blocked**

实例如下：

"RMI TCP Connection(267865)-172.16.5.25" daemon prio=10 tid=0x00007fd508371000 nid=0x55ae **waiting for monitor entry** [0x00007fd4f8684000]

   java.lang.Thread.State: **BLOCKED (on object monitor)**

at org.apache.log4j.Category.callAppenders(Category.java:201)

- **waiting to lock <0x00000000acf4d0c0>** (a org.apache.log4j.Logger)

at org.apache.log4j.Category.forcedLog(Category.java:388)

at org.apache.log4j.Category.log(Category.java:853)

at org.apache.commons.logging.impl.Log4JLogger.warn(Log4JLogger.java:234)

at com.tuan.core.common.lang.cache.remote.SpyMemcachedClient.get(SpyMemcachedClient.java:110)

……

1）线程状态是 **Blocked**，阻塞状态。说明线程等待资源超时！

2）“ waiting to lock <0x00000000acf4d0c0>”指，线程在等待给这个 0x00000000acf4d0c0 地址上锁（英文可描述为：trying to obtain  0x00000000acf4d0c0 lock）。

3）在 dump 日志里查找字符串 0x00000000acf4d0c0，发现有大量线程都在等待给这个地址上锁。如果能在日志里找到谁获得了这个锁（如locked < 0x00000000acf4d0c0 >），就可以顺藤摸瓜了。

4）“**waiting for monitor entry**”说明此线程通过 synchronized(obj) {……} 申请进入了临界区，从而进入了下图1中的“Entry Set”队列，但该 obj 对应的 monitor 被其他线程拥有，所以本线程在 Entry Set 队列中等待。

5）第一行里，"RMI TCP Connection(267865)-172.16.5.25"是 Thread Name 。tid指Java Thread id。nid指native线程的id。prio是线程优先级。[0x00007fd4f8684000]是线程栈起始地址。

**Dump文件中的线程状态含义及注意事项**

含义如下所示：

* Deadlock：死锁线程，一般指多个线程调用间，进入相互资源占用，导致一直等待无法释放的情况。
* Runnable：一般指该线程正在执行状态中，该线程占用了资源，正在处理某个请求，有可能正在传递SQL到数据库执行，有可能在对某个文件操作，有可能进行数据类型等转换。
* Waiting on condition：等待资源，或等待某个条件的发生。具体原因需结合 stacktrace来分析。
  + 如果堆栈信息明确是应用代码，则证明该线程正在等待资源。一般是大量读取某资源，且该资源采用了资源锁的情况下，线程进入等待状态，等待资源的读取。
  + 又或者，正在等待其他线程的执行等。
  + 如果发现有大量的线程都在处在 Wait on condition，从线程 stack看，正等待网络读写，这可能是一个网络瓶颈的征兆。因为网络阻塞导致线程无法执行。
    - 一种情况是网络非常忙，几乎消耗了所有的带宽，仍然有大量数据等待网络读写；
    - 另一种情况也可能是网络空闲，但由于路由等问题，导致包无法正常的到达。
  + 另外一种出现 Wait on condition的常见情况是该线程在 sleep，等待 sleep的时间到了时候，将被唤醒。
* Blocked：线程阻塞，是指当前线程执行过程中，所需要的资源长时间等待却一直未能获取到，被容器的线程管理器标识为阻塞状态，可以理解为等待资源超时的线程。
* Waiting for monitor entry 和 in Object.wait()：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。从下图1中可以看出，每个 Monitor在某个时刻，只能被一个线程拥有，该线程就是 “Active Thread”，而其它线程都是 “Waiting Thread”，分别在两个队列 “ Entry Set”和 “Wait Set”里面等候。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()”。

IMG_256

图1 A Java Monitor

**综合示范二：Waiting on condition 和 TIMED\_WAITING**

实例如下：

"RMI TCP Connection(idle)" daemon prio=10 tid=0x00007fd50834e800 nid=0x56b2 **waiting on condition** [0x00007fd4f1a59000]

   java.lang.Thread.State: **TIMED\_WAITING (parking)**

at sun.misc.Unsafe.park(Native Method)

- **parking to wait for  <0x00000000acd84de8>** (a java.util.concurrent.SynchronousQueue$TransferStack)

at java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:198)

at java.util.concurrent.SynchronousQueue$TransferStack.awaitFulfill(SynchronousQueue.java:424)

at java.util.concurrent.SynchronousQueue$TransferStack.transfer(SynchronousQueue.java:323)

at java.util.concurrent.SynchronousQueue.poll(SynchronousQueue.java:874)

at java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:945)

at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:907)

at java.lang.Thread.run(Thread.java:662)

1）“**TIMED\_WAITING (parking)**”中的 timed\_waiting 指等待状态，但这里指定了时间，到达指定的时间后自动退出等待状态；parking指线程处于挂起中。

2）“**waiting on condition**”需要与堆栈中的“**parking to wait for  <0x00000000acd84de8>** (a java.util.concurrent.SynchronousQueue$TransferStack)”结合来看。首先，本线程肯定是在等待某个条件的发生，来把自己唤醒。其次，SynchronousQueue 并不是一个队列，只是线程之间移交信息的机制，当我们把一个元素放入到 SynchronousQueue 中时必须有另一个线程正在等待接受移交的任务，因此这就是本线程在等待的条件。

3）别的就看不出来了。

**综合示范三：in Obejct.wait() 和 TIMED\_WAITING**

实例如下：

"RMI RenewClean-[172.16.5.19:28475]" daemon prio=10 tid=0x0000000041428800 nid=0xb09 **in Object.wait()** [0x00007f34f4bd0000]

   java.lang.Thread.State: **TIMED\_WAITING (on object monitor)**

at java.lang.Object.wait(Native Method)

- **waiting on <0x00000000aa672478> (a java.lang.ref.ReferenceQueue$Lock)**

at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:118)

- **locked <0x00000000aa672478> (a java.lang.ref.ReferenceQueue$Lock)**

at sun.rmi.transport.DGCClient$EndpointEntry$RenewCleanThread.run(DGCClient.java:516)

at java.lang.Thread.run(Thread.java:662)

1）“**TIMED\_WAITING (on object monitor)**”，对于本例而言，是因为本线程调用了 java.lang.Object.wait(long timeout) 而进入等待状态。

2）“Wait Set”中等待的线程状态就是“ **in Object.wait()** ”。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，放弃了 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。

3）RMI RenewClean 是 DGCClient 的一部分。[DGC](http://doc.java.sun.com/DocWeb/api/java.rmi.dgc.DGC?lang=zh_cn&mode=Read) 指的是 Distributed GC，即分布式垃圾回收。

4）请注意，是先 **locked <0x00000000aa672478>**，后 **waiting on <0x00000000aa672478>**，之所以先锁再等同一个对象，请看下面它的代码实现：

static private class  Lock { };

private Lock lock = new Lock();

public Reference<? extends T> [remove](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/lang/ref/ReferenceQueue.java)(long timeout)

{

    synchronized (lock) {

        Reference<? extends T> r = [reallyPoll](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/lang/ref/ReferenceQueue.java" \l "ReferenceQueue.reallyPoll()" \o "java.lang.ref.ReferenceQueue.reallyPoll() : Reference)();

        if (r != null) return r;

        for (;;) {

[lock](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/lang/ref/ReferenceQueue.java" \l "ReferenceQueue.0lock" \o "Lock lock).[wait](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/lang/Object.java" \l "Object.wait(long)" \o "java.lang.Object.wait(long) : void)(timeout);

            r = [reallyPoll](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/lang/ref/ReferenceQueue.java" \l "ReferenceQueue.reallyPoll()" \o "java.lang.ref.ReferenceQueue.reallyPoll() : Reference)();

            ……

       }

}

即，线程的执行中，先用 synchronized 获得了这个对象的 Monitor（对应于  **locked <0x00000000aa672478>** ）；当执行到 lock.wait(timeout);，线程就放弃了 Monitor 的所有权，进入“Wait Set”队列（对应于  **waiting on <0x00000000aa672478>** ）。

5）从堆栈信息看，是正在清理 remote references to remote objects ，引用的租约到了，分布式垃圾回收在逐一清理呢。