

Java多线程详细分析

自我介绍

QQ:494460705

张振华.Jack

2014

zhangzhenhua846@126.com

10年开发

概要

- 概念
- 实现方法
- 生命周期
- 安全和锁
- **Concurrent包(安全集合类、安全Queue)**
- 线程阻塞机制
- 线程池详解(原理，实际使用)
- 线程的监控,分析方法
- 扩展数据库连接池

概念

- 什么是进程
 - 是资源分配的最小单位; (资源, 包括各种表格、内存空间、磁盘空间)
 - 同一进程中的多条线程将共享该进程中的全部系统资源
- 什么是线程
 - 线程只由相关[堆栈](#) ([系统](#)栈或[用户](#)栈) [寄存器](#)和线程控制表TCB组成。 [寄存器](#)可被用来存储线程内的[局部变量](#)
 - 线程是CPU调度的最小单位
- 并行运行:总线程数 \leq CPU数量*核心数:
- 并发运行:总线程数 $>$ CPU数量*核心数: (时间片轮转进程调度算法)

三种实现方法

```
public class A extends Thread {  
    @Override  
    public void run() {  
        // TODO Auto-generated method stub  
        super.run();  
        System.out.println("#####");  
    }  
    public static void main(String args[]) {  
        A a = new A();  
        a.start();  
    }  
}
```

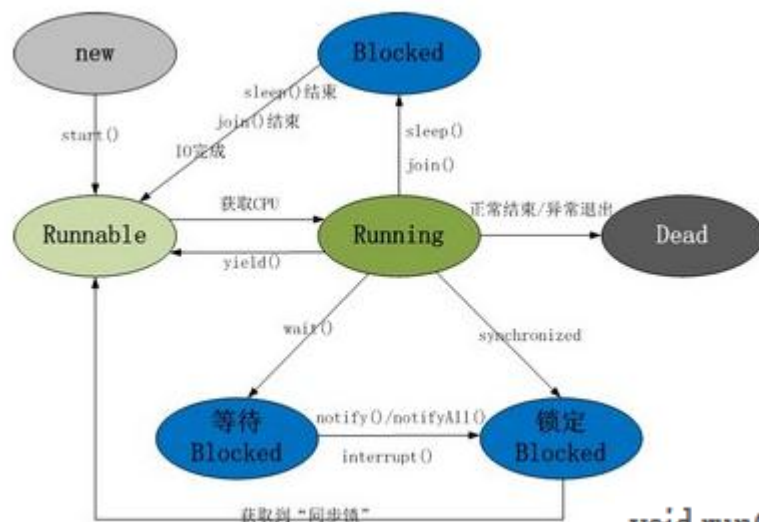
```
public class A implements Runnable {  
    @Override  
    public void run() {  
        System.out.println("zzzz");  
    }  
    public static void main(String args[]) {  
        A a = new A();  
        new Thread(a).start();  
    }  
}
```

多继承问题

```
public class A implements Callable<String> {  
    @Override  
    public String call() {  
        System.out.println("zzzz");  
        return "success";  
    }  
    public static void main(String args[]) {  
        A a = new A();  
        FutureTask<String> future = new FutureTask<String>(a);  
        new Thread(future).start();  
        System.out.println("1111111111");  
        try {  
            System.out.println(future.get());  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
        System.out.println("222222222222");  
    }  
}
```

返回结果

线程的生命周期/常用方法



获得当前线程的副本：

`Thread current = Thread.currentThread();`

例如多线程上传的文件的进度条可以使用

ThreadLocal是如何做到为每一个线程维护变量的副本的呢？查看源码发现，在**ThreadLocal**类中有一个线程安全的**Map**，用于存储每一个线程的变量的副本。

`void run()` 创建该类的子类时必须实现的方法

`void start()` 开启线程的`void run()` 方法

`static void sleep(long t)` 释放CPU的执行权，不释放锁

`static void sleep(long millis,int nanos)`

`final void wait()` 释放CPU的执行权，释放锁

`final void notify()`

`static void yield()` 可以对当前线程进行临时暂停（让线程将资源释放出来）

何为安全

- 什么是安全的？先问个问题
- 是不是不加锁就不是线程安全的，个人感觉只要你代码里面没有变量互串，线程之间互不影响，例如**server**的设计方法；
- 非单例代码也是安全的，注意只有单例模式下才会有锁的问题

锁

- 隐式锁:**synchronized**(同一个对象锁下面的, **synchronized** 区域是互斥的)
- 方法锁(默认是当前对象的锁)
- 代码块锁(性能高于方法锁, 可以指定哪个对象的锁)
- 显示锁: **java.util.concurrent.lock** (需要手动关/开), 注意自己的代码逻辑不要产生死锁了

```
Lock lock = new ReentrantLock();  
lock.lock();  
try {  
    // update object state  
}  
finally {  
    lock.unlock();  
}
```



- 关键字: **volatile**(线程在每次使用变量的时候, 都会读取变量修改后的最新的值)
其实是有风险的, 并行情况下不一定正确, 有可能两个线程同时取到最后修改的值
- 原子操作: **java.util.concurrent.atomic** (**AtomicBoolean**, **AtomicLong**, **AtomicInteger**等一些类库)
(如果查看源码的话, 实现原理是**volatile**, 相对安全的, 个人觉得不绝对)

问一个问题，请看以下下面的锁有用吗

```
--
public class Abc {
    >> private byte[] lock = new byte[1];
    >> public void ab() {
    >>     >> synchronized(lock) {
    >>         >>         System.out.println("eeee");
    >>         >>         try {
    >>             >>             Thread.sleep(50001);
    >>             >>         } catch (InterruptedException e) {
    >>             >>             e.printStackTrace();
    >>             >>         }
    >>         >>         System.out.println("11111");
    >>     }
    >> }
    >>
    >> public synchronized void cd() {
    >>     >>     System.out.println("ffffff");
    >>     >>     try {
    >>         >>         Thread.sleep(50001);
    >>         >>     } catch (InterruptedException e) {
    >>         >>         e.printStackTrace();
    >>         >>     }
    >>     >>     System.out.println("2222");
    >> }
}
}
```

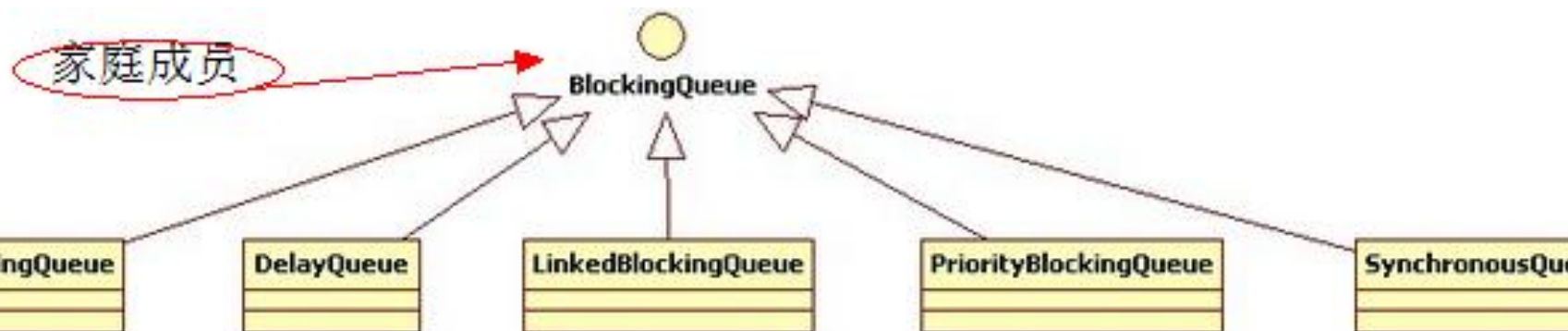

- 这种锁是没有用的啊，因为**synchronized**的对象不一样

常用的Concurrent线程安全类库

- `java.util.concurrent.ConcurrentHashMap`
- `java.util.concurrent.ConcurrentLinkedQueue`
- `java.util.concurrent.ConcurrentMap`
- `java.util.concurrent.ConcurrentNavigableMap`
- `java.util.concurrent.ConcurrentSkipListMap`
- `java.util.concurrent.ConcurrentSkipListSet`
- 实现原理和算法类似lock的

安全线程阻塞队列(1)

java.util.concurrent.BlockingQueue



- **BlockingQueue**很好的解决了多线程中高效安全“传输”数据的问题;
- 基于**java.util.Queue**的基础上做了一些线程安全的封装;
(对以下方法做了一些的扩展和封装)

| | | |
|----------------|-----------------------|--|
| add | 增加一个元素 | 如果队列已满，则抛出一个 IllegalStateException 异常 |
| remove | 移除并返回队列头部的元素 | 如果队列为空，则抛出一个 NoSuchElementException |
| element | 返回队列头部的元素 | 如果队列为空，则抛出一个 NoSuchElementException |
| offer | 添加一个元素并返回 true | 如果队列已满，则返回 false |
| poll | 移除并返回队列头部的元素 | 如果队列为空，则返回 null |
| peek | 返回队列头部的元素 | 如果队列为空，则返回 null |
| put | 添加一个元素 | 如果队列满，则阻塞 |
| take | 移除并返回队列头部的元素 | 如果队列为空，则阻塞 |

安全线程阻塞队列(2)

BlockingQueue常用的儿子

- **1. ArrayBlockingQueue**

基于数组的阻塞队列实现，在ArrayBlockingQueue内部，维护了一个定长数组，以便缓存队列中的数据对象，这是一个常用的阻塞队列，除了一个定长数组外，ArrayBlockingQueue内部还保存着两个整形变量，分别标识着队列的头部和尾部在数组中的位置；

- **2. LinkedBlockingQueue**

基于链表的阻塞队列，同ArrayListBlockingQueue类似，其内部也维持着一个数据缓冲队列（该队列由一个链表构成）

- **3. DelayQueue**

DelayQueue中的元素只有当其指定的延迟时间到了，才能够从队列中获取到该元素。DelayQueue是一个没有大小限制的队列，因此往队列中插入数据的操作（生产者）永远不会被阻塞，而只有获取数据的操作（消费者）才会被阻塞。

使用场景：

DelayQueue使用场景较少，但都相当巧妙，常见的例子比如使用一个DelayQueue来管理一个超时未响应的连接队列。

- **4. PriorityBlockingQueue**

基于优先级的阻塞队列（优先级的判断通过构造函数传入的Compator对象来决定），但需要注意的是PriorityBlockingQueue并不会阻塞数据生产者，而只会在没有可消费的数据时，阻塞数据的消费者。

- **5.SynchronousQueue**

- 1) 种无缓冲的等待队列，同步队列没有任何内部容量，甚至连一个队列的容量都没有；
- 2) 其中每个 put 必须等待一个 take，反之亦然。。
- 3) 无锁的机制实现；（可想而知高并发的时候性能肯定是最高的）

线程伐(1)

java.util.concurrent. CountDownLatch

CountDownLatch类是一个同步计数器,构造时传入int参数,该参数就是计数器的初始值,每调用一次countDown()方法,计数器减1,计数器大于0时,await()方法会阻塞程序继续执行;

【实际场景,如:开5多线程去下载,当5个线程都执行完了才算下载成功!】

```
public class CountDownLatchDemo {  
    >> public static void main(String[] args) throws InterruptedException {  
    >>     CountDownLatch latch = new CountDownLatch(2); // 两个工人的协作  
    >>     Worker worker1 = new Worker("zhang-san", latch);  
    >>     Worker worker2 = new Worker("li-si", latch);  
    >>     worker1.start();  
    >>     worker2.start();  
    >>     latch.await(); // 等待所有工人完成工作  
    >>     System.out.println("all work done at");  
    >> }  
    >> static class Worker extends Thread {  
    >>     String workerName;  
    >>     CountDownLatch latch;  
    >>     public Worker(String workerName, CountDownLatch latch) {  
    >>         this.workerName = workerName;  
    >>         this.latch = latch;  
    >>     }  
    >>     public void run() {  
    >>         System.out.println("Worker-" + workerName + "-do work begin");  
    >>         try {  
    >>             Thread.sleep(5000); // 工作了  
    >>         } catch (InterruptedException e) {}  
    >>         System.out.println("Worker-" + workerName + "-do work complete");  
    >>         latch.countDown(); // 工人完成工作, 计数器减一  
    >>     }  
    >> }  
}
```

线程伐(2)

java.util.concurrent. CyclicBarrier

【应用场景】我们需要统计全国的业务数据。其中各省的数据库是独立的，也就是说按省分库。并且统计的数据量很大，统计过程也比较慢。为了提高性能，快速计算。我们采取并发的方式，多个线程同时计算各省数据，每个省下面又用多线程，最后再汇总统计

```
CyclicBarrier(int):    //设置parties、count及barrierCommand属性。  
CyclicBarrier(int,Runnable):    //当await的数量到达了设定的数量后，首先执行该Runnable对象。  
  
await():    //通知barrier已完成线程  
  
await():    //通知barrier已完成线程  
  
reset():    // 将屏障重置为其初始状态。
```

CyclicBarrier是一个同步辅助类，它允许一组线程互相等待，直到到达某个公共屏障点 (common barrier point)。在涉及一组固定大小的线程的程序中，这些线程必须不时地互相等待，此时 CyclicBarrier 很有用。因为该 barrier 在释放等待线程后可以重用，所以称它为循环的 barrier。

```

public class Total { ...
... public static void main(String[] args) { ...
...     CyclicBarrier barrier = new CyclicBarrier(3, new TotalTask()); ...
...     // 实际系统是查出所有省编码code的列表，然后循环，每个code生成一个线程。 ...
...     new BillTask(barrier, "北京").start(); ...
...     new BillTask(barrier, "上海").start(); ...
...     new BillTask(barrier, "广西").start(); ...
... } ...
} ...

class TotalTask implements Runnable { ... // 主任务：汇总任务 // 当然了这里也可以再拆分很多省份的任务
... public void run() { ...
...     System.out.println("开始全国汇总"); ...
... } ...
} ...

class BillTask extends Thread { ...
... private CyclicBarrier barrier; ...
... private String code; ... // 代码，按省代码分类，各省数据库独立。 ...
... BillTask(CyclicBarrier barrier, String code) { ...
...     this.barrier = barrier; ...
...     this.code = code; ...
... } ...
... public void run() { ...
...     System.out.println("开始计算--" + code + "省--数据!"); ...
...     // Need TODO .....
...     System.out.println(code + "省已经计算完成,并通知汇总Service!"); ...
...     try { ...
...         barrier.await(); ... // 通知barrier已经完成 // 最好放在final里面 ...
...     } catch (Exception e) { ...
...         e.printStackTrace(); ...
...     } ...
} ...

```

线程伐(3) (信号装置)

java.util.concurrent. Semaphore

概念：

就像一个排队进入上海博物馆一样，放几个人等一下，有几个人走了然后再放几个人进入；像是一种排队机制；

定义：一个计数信号量。

从概念上讲，信号量维护了一个许可集合。如有必要，在许可可用前会阻塞每一个 `acquire()`，然后再获取该许可。每个 `release()` 添加一个许可，从而可能释放一个正在阻塞的获取者。


```
public class HelloSemaphore {  
    public static void main(String[] args) {  
        ExecutorService exec = Executors.newCachedThreadPool(); // 线程池  
        final Semaphore semp = new Semaphore(5); // 只能5个线程同时访问  
        for (int index = 0; index < 50; index++) { // 模拟50个客户端访问  
            final int NO = index;  
            Runnable run = new Runnable() {  
                public void run() {  
                    try {  
                        semp.acquire(); // 获取许可  
                        System.out.println("Accessing: " + NO);  
                        Thread.sleep((long) (Math.random() * 10000));  
                        semp.release(); // 访问完后，释放  
                        // availablePermits() 指的是当前信号灯库中有多少个可以被使用  
                        System.out.println("-----" + semp.availablePermits());  
                    } catch (InterruptedException e) {  
                        e.printStackTrace();  
                    }  
                }  
            };  
            exec.execute(run);  
        }  
        exec.shutdown(); // 退出线程池  
    }  
}
```

线程伐(4) (任务机制)

`java.util.concurrent. Future->FutureTask`

- 1:一般FutureTask多用于耗时的计算，主线程再完成自己的任务后，再去获取结果。
- 2:只有在计算完成时获取，否则会一直阻塞直到任务完成状态。



前面有提到例子，这里不多说了

线程池概念

- 先看三个例子后解释；

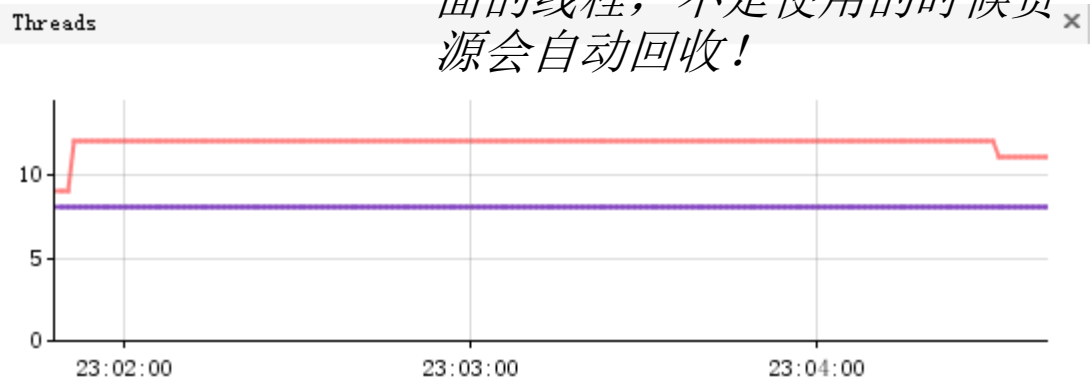
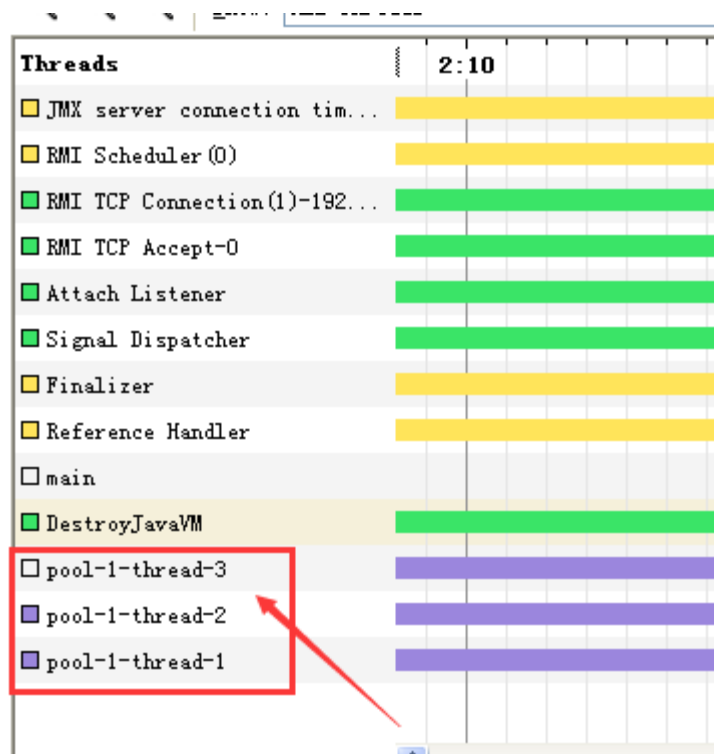
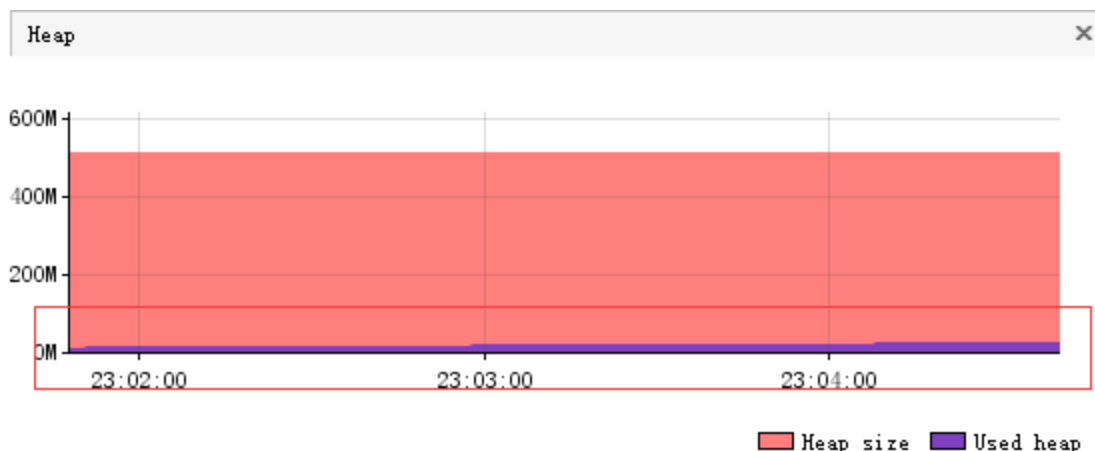
线程池常用的方法

newFixedThreadPool（固定大小线程池）

```
public static void main(String[] args) throws InterruptedException {  
    >> ExecutorService exec = Executors.newFixedThreadPool(3); // 线程池  
    >> final Semaphore semp = new Semaphore(5); // 只能5个线程同时访问  
    >> Thread.sleep(30000);  
    >> for (int index = 0; index < 50; index++) { // 模拟50个客户端访问  
    >>     >> final int NO = index;  
    >>     >> Runnable run = new Runnable() {  
    >>     >>     >> public void run() {  
    >>     >>     >>     >> try {  
    >>     >>     >>     >>     >> semp.acquire(); // 获取许可  
    >>     >>     >>     >>     >> System.out.println("Accessing: " + NO);  
    >>     >>     >>     >>     >> Thread.sleep(10000);  
    >>     >>     >>     >>     >> semp.release(); // 访问完后，释放  
    >>     >>     >>     >>     >> // availablePermits() 指的是当前信号灯库中有多少个可以被使用  
    >>     >>     >>     >>     >> System.out.println("-----" + semp.availablePermits());  
    >>     >>     >>     >>     >> } catch (InterruptedException e) {  
    >>     >>     >>     >>     >> e.printStackTrace();  
    >>     >>     >>     >>     >> }  
    >>     >>     >>     >> }  
    >>     >>     >> }  
    >>     >> }  
    >>     >> exec.execute(run);  
    >> }  
    >> exec.shutdown(); // 退出线程池  
    >> }
```

引用上面的例子注意
这里

注意查看heap占用和执行时间和线程数

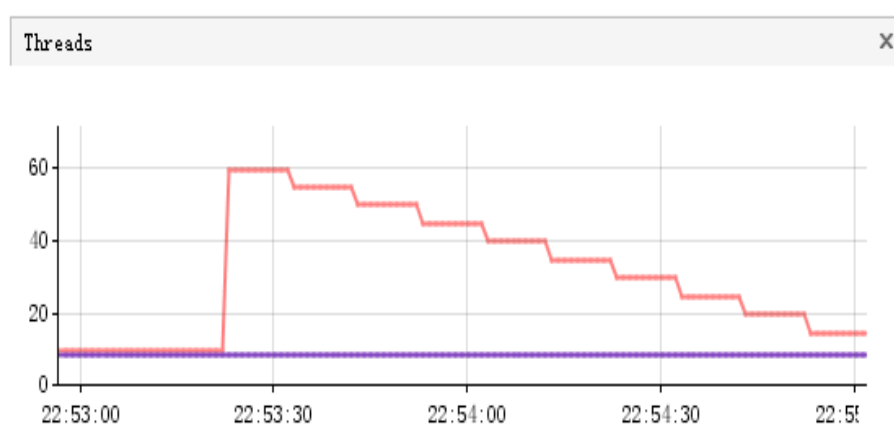
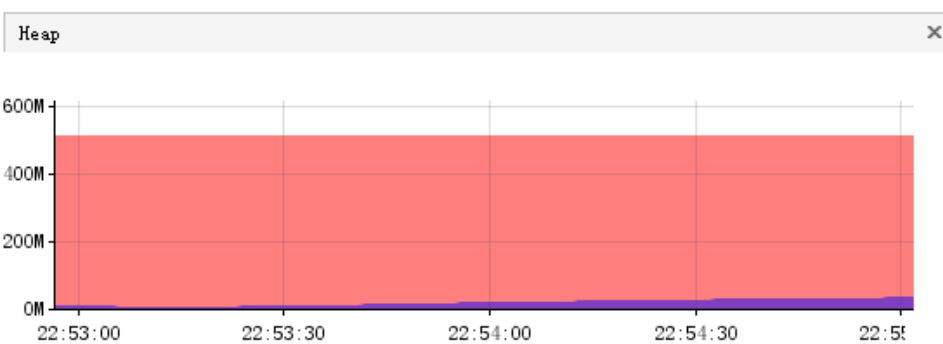


- 1: 内存大小不变;
- 2: 线程数量不变;
- 3: 不会流量上来的时候一下子给服务器弄死机;
- 4: 相对于 `newCachedThreadPool` 来说, 线程池所占的资源永远都不会释放; 而 `newCachedThreadPool` 里面的线程, 不是使用的时候资源会自动回收!

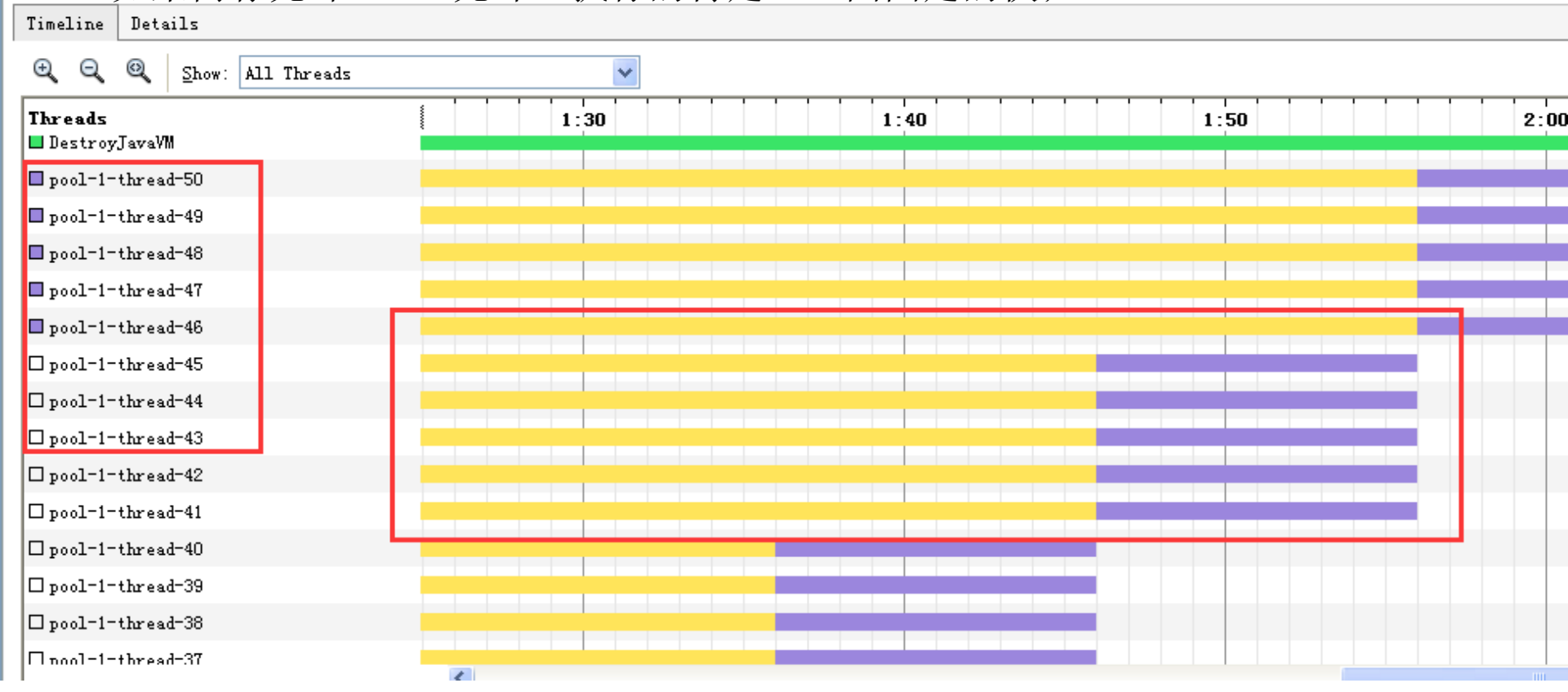
线程池常用的方法

newCachedThreadPool (无界线程池, 可以进行自动线程回收)

```
public static void main(String[] args) throws InterruptedException {  
    >> ExecutorService exec = Executors.newCachedThreadPool(); // 线程池  
    >> final Semaphore semp = new Semaphore(5); // 只能5个线程同时访问  
    >> Thread.sleep(30000);  
    >> for (int index = 0; index < 50; index++) { // 模拟50个客户端访问  
    >>     final int NO = index;  
    >>     Runnable run = new Runnable() {  
    >>     >>         public void run() {  
    >>     >>     >>             try {  
    >>     >>     >>     >>                 semp.acquire(); // 获取许可  
    >>     >>     >>     >>                 System.out.println("Accessing: " + NO);  
    >>     >>     >>     >>                 Thread.sleep(10000);  
    >>     >>     >>     >>                 semp.release(); // 访问完后, 释放  
    >>     >>     >>     >>                 // availablePermits() 指的是当前信号灯库中有多少个可以被使用  
    >>     >>     >>     >>                 System.out.println("-----" + semp.availablePermits());  
    >>     >>     >>     >>             } catch (InterruptedException e) {  
    >>     >>     >>     >>                 e.printStackTrace();  
    >>     >>     >>     >>             }  
    >>     >>     >>     }  
    >>     >>     }  
    >>     }  
    >>     exec.execute(run);  
    >> }  
    >> exec.shutdown(); // 退出线程池  
}
```



- 1: 内存顿时上升;
- 2: 瞬间创建了50个线程, 每5个一并发在等待执行;
- 3: 如果内存允许, CPU允许, 执行的肯定比上面固定的快;



线程池常用的方法

newSingleThreadExecutor （单个后台线程）

```
public static void main(String[] args) throws InterruptedException {
    >> ExecutorService exec = Executors.newSingleThreadExecutor(); // 线程池
    >> final Semaphore semp = new Semaphore(5); // 只能5个线程同时访问
    >> Thread.sleep(30000);
    >> for (int index = 0; index < 50; index++) { // 模拟50个客户端访问
    >>     >> final int NO = index;
    >>     >> Runnable run = new Runnable() {
    >>     >>     >> public void run() {
    >>     >>     >>     >> try {
    >>     >>     >>     >>     >> semp.acquire(); // 获取许可
    >>     >>     >>     >>     >> System.out.println("Accessing: " + NO);
    >>     >>     >>     >>     >> Thread.sleep(10000);
    >>     >>     >>     >>     >> semp.release(); // 访问完后，释放
    >>     >>     >>     >>     >> // availablePermits() 指的是当前信号灯库中有多少个可以被使用
    >>     >>     >>     >>     >> System.out.println("-----" + semp.availablePermits());
    >>     >>     >>     >>     >> } catch (InterruptedException e) {
    >>     >>     >>     >>     >> e.printStackTrace();
    >>     >>     >>     >>     >> }
    >>     >>     >>     >> }
    >>     >>     >> }
    >>     >> }
    >>     >> exec.execute(run);
    >> }
    >> exec.shutdown(); // 退出线程池
}
```

顾名思义：只能有一个线程在执行，就不多说了，内存使用更少，但同时执行的时间也会更长；

线程池的概念

- 单个任务处理的时间比较短(如:互联网)
- 将需处理的任务的数量大
- 接受突发性的的大量请求，不至于当机；
- 使用线程池的好处：
 - 减少在创建和销毁线程上所花的时间以及系统资源的开销
 - 如不使用线程池，有可能造成系统创建大量线程而导致消耗完系统内存以及”过度切换”。

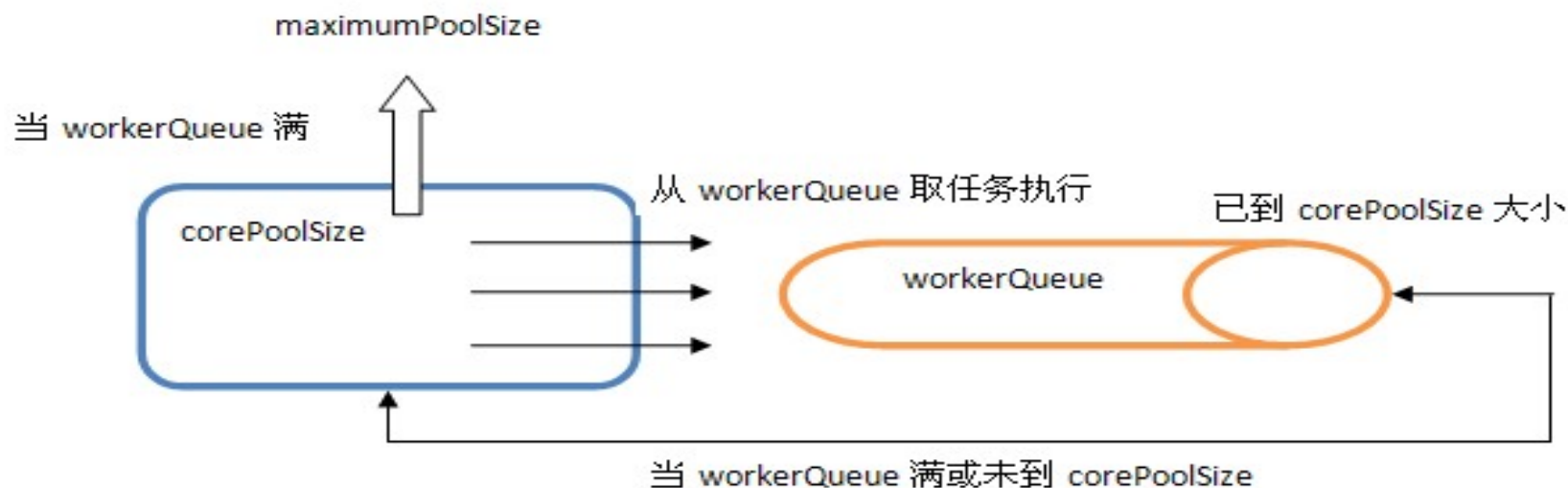
线程池原理

线程池实现原理

最主要的两个类：

- 1) 线程等待池 (`ThreadPool` ,通过`BlockingQueue`进行等待执行) ；
- 2) 任务处理池 (`PoolWorker`) ；

内部结构如下所示：(利用队列原理和任务池，交替切换处理任务，队列里面的每个线程执行完之后就会销毁，释放资源，只是任务池里面的线程，永远都是持有资源的状态，通过java的native方法用c做底层做的)



知道个大概，具体也没仔细研究，没打算自己写个线程池的底层！

认识线程池

- `ExecutorService`;接口类; `ThreadPoolExecutor`的接口; 线程池的可执行服务;

```
ExecutorService exec = Executors.newFixedThreadPool(3); // 线程池
exec.execute(thread);
exec.shutdown(); // 退出线程池
```

- `ThreadPoolExecutor`通过这个类可以自定义线程池

```
public ThreadPoolExecutor(int corePoolSize,  
                           int maximumPoolSize,  
                           long keepAliveTime,  
                           TimeUnit unit,  
                           BlockingQueue<Runnable> workQueue,  
                           ThreadFactory threadFactory,  
                           RejectedExecutionHandler handler)
```

认识主要参数

- **corePoolSize** - 池中所保存的线程数，包括空闲线程。
- **maximumPoolSize** - 池中允许的最大线程数。
- **keepAliveTime** - 当线程数大于核心时，此为终止前多余的空闲线程等待新任务的最长时间。
- **unit** - keepAliveTime 参数的时间单位。
- **workQueue** - 执行前用于保持任务的队列。此队列仅保持由 `execute` 方法提交的 `Runnable` 任务。
- **threadFactory** - 执行程序创建新线程时使用的工厂。
- **handler** - (异常处理程序)由于超出线程范围和队列容量而使执行被阻塞时所使用的处理程序。

- **queue上的三种类型**
- 直接提交。SynchronousQueue。
- 无界队列。使用无界队列（例如，不具有预定义容量的 LinkedBlockingQueue）
- 有界队列。当使用有限 maximumPoolSizes 时，有界队列（如 ArrayBlockingQueue）有助于防止资源耗尽。

RejectedExecutionHandler来处理线程池处理失败的任务；

AbortPolicy(默认)：这种策略直接抛出异常，丢弃任务。

DiscardPolicy：不能执行的任务将被删除；这种策略和AbortPolicy几乎一样，也是丢弃任务，只不过他不抛出异常。

DiscardOldestPolicy：如果执行程序尚未关闭，则位于工作队列头部的任务被删除，然后重试执行程序（如果再次失败，则重复此过程）当然也可以自定义：

ThreadFactory：默认情况下为Executors.defaultThreadFactory()：我们可以采用自定义的ThreadFactory工厂，增加对线程创建与销毁等更多的控制，

问个问题？

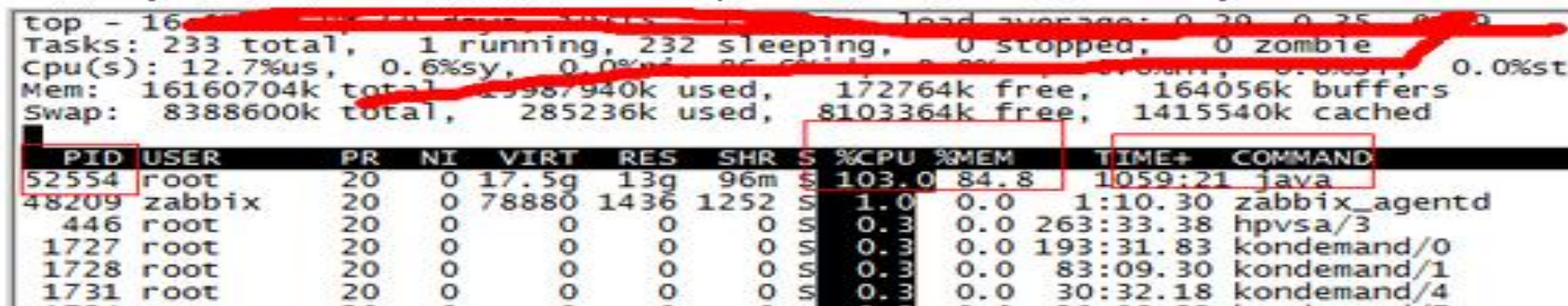
- Web项目中咱们常用的线程池是不是单利的？
- 我见过有的同事，在**services**方法里面去创建线程池，这是不可以去的，因为每当这个方法被调用的时候不是创建多少个线程的问题了，而是创建出来了一大堆线程池！

线程的监控方法(1)全手动

- **top -p 12377 -H**

查看java进程的有哪些线程的运行情况;

先用top命令找出占用资源厉害的java进程id, 如图: # top



```
top - 16:00:00 up 10 days, 10:00, 1 user, load average: 0.20, 0.25, 0.30
Tasks: 233 total, 1 running, 232 sleeping, 0 stopped, 0 zombie
Cpu(s): 12.7%us, 0.6%sy, 0.0%ni, 86.7%id, 0.0%wa, 0.0%st, 0.0%gn, 0.0%cs, 0.0%st
Mem: 16160704k total, 15987940k used, 172764k free, 164056k buffers
Swap: 8388600k total, 285236k used, 8103364k free, 1415540k cached
```

| PID | USER | PR | NI | VIRT | RES | SHR | S | %CPU | %MEM | TIME+ | COMMAND |
|-------|--------|----|----|-------|------|------|---|-------|------|-----------|---------------|
| 52554 | root | 20 | 0 | 17.5g | 13g | 96m | S | 103.0 | 84.8 | 1059:21 | java |
| 48209 | zabbix | 20 | 0 | 78880 | 1436 | 1252 | S | 1.0 | 0.0 | 1:10.30 | zabbix_agentd |
| 446 | root | 20 | 0 | 0 | 0 | 0 | S | 0.3 | 0.0 | 263:33.38 | hpvsa/3 |
| 1727 | root | 20 | 0 | 0 | 0 | 0 | S | 0.3 | 0.0 | 193:31.83 | kondemand/0 |
| 1728 | root | 20 | 0 | 0 | 0 | 0 | S | 0.3 | 0.0 | 83:09.30 | kondemand/1 |
| 1731 | root | 20 | 0 | 0 | 0 | 0 | S | 0.3 | 0.0 | 30:32.18 | kondemand/4 |

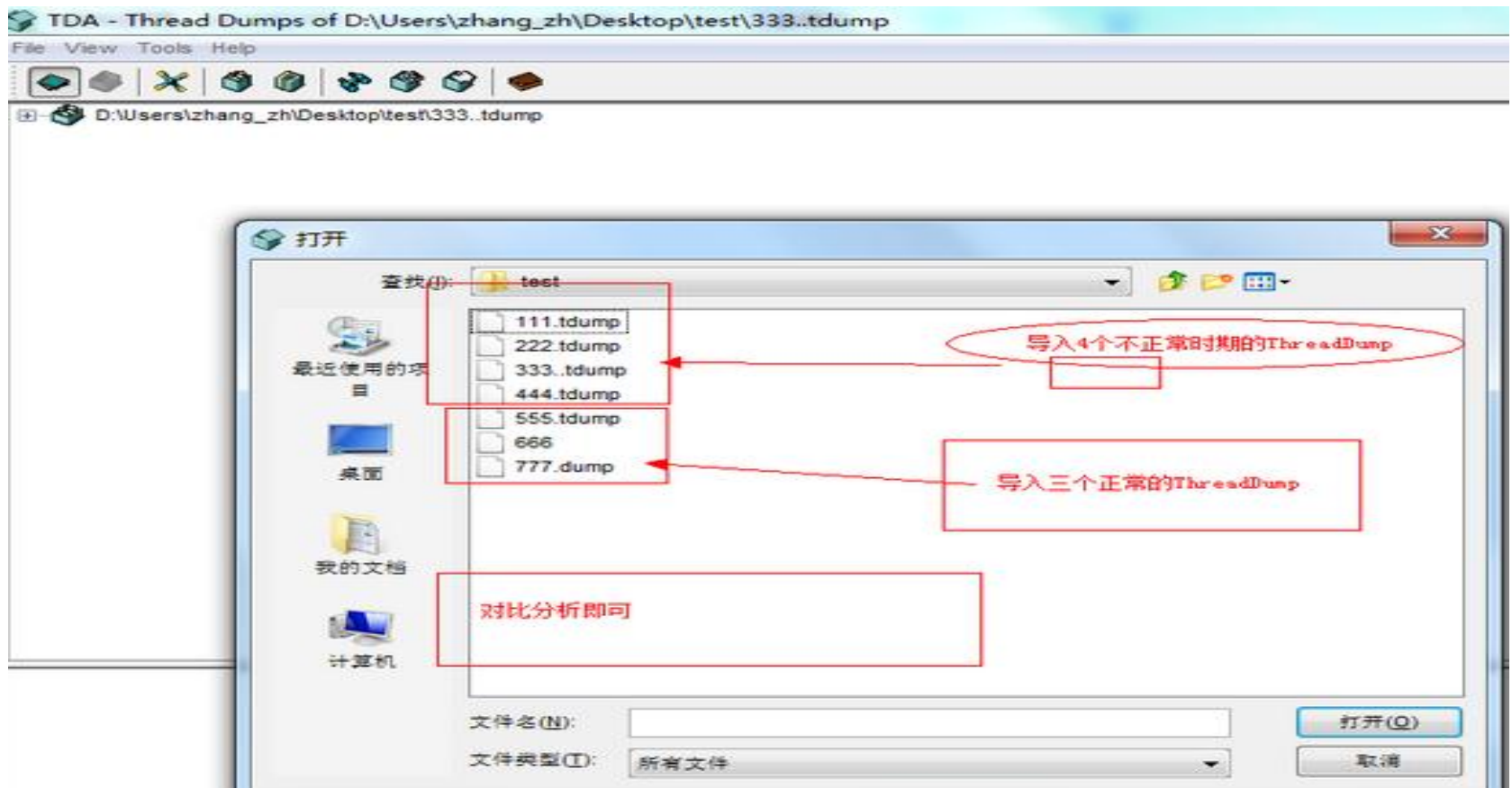
- **Jstack** 生成线程栈dump的信息
- 然后将上面的pid转换成16进制到dump文件里面查看占cpu和mem高的线程信息即可;

线程的监控方法(2)插件分析

- **thread dump analyzer**

下载: <https://java.net/projects/tda/downloads/directory/visualvm>

- 将正常的，非正常的线程dump拿来对比分析



可以详细看出线程里面的一些类的相关信息

TDA - Thread Dumps of C:\Users\zhang_zh\Desktop\tech\333\tdump...

File View Tools Help

Threads (128 Threads overall)

Threads waiting for Monitors (3 Threads waiting)

Threads sleeping on Monitors (123 Threads sleeping)

Threads locking Monitors (94 Threads locking)

Monitors (142 Monitors)

Dump No. 1 at line 2 around 2014-06-06 14:01:14

Threads (128 Threads overall)

Threads waiting for Monitors (3 Threads waiting)

Threads sleeping on Monitors (123 Threads sleeping)

Threads locking Monitors (94 Threads locking)

Monitors (142 Monitors)

Dump No. 1 at line 2 around 2014-06-06 13:59:30

Threads (128 Threads overall)

Threads waiting for Monitors (3 Threads waiting)

Threads sleeping on Monitors (123 Threads sleeping)

Threads locking Monitors (94 Threads locking)

Monitors (142 Monitors)

Dump No. 1 at line 2 around 2014-06-06 14:00:30

Threads (128 Threads overall)

Threads waiting for Monitors (3 Threads waiting)

Threads sleeping on Monitors (123 Threads sleeping)

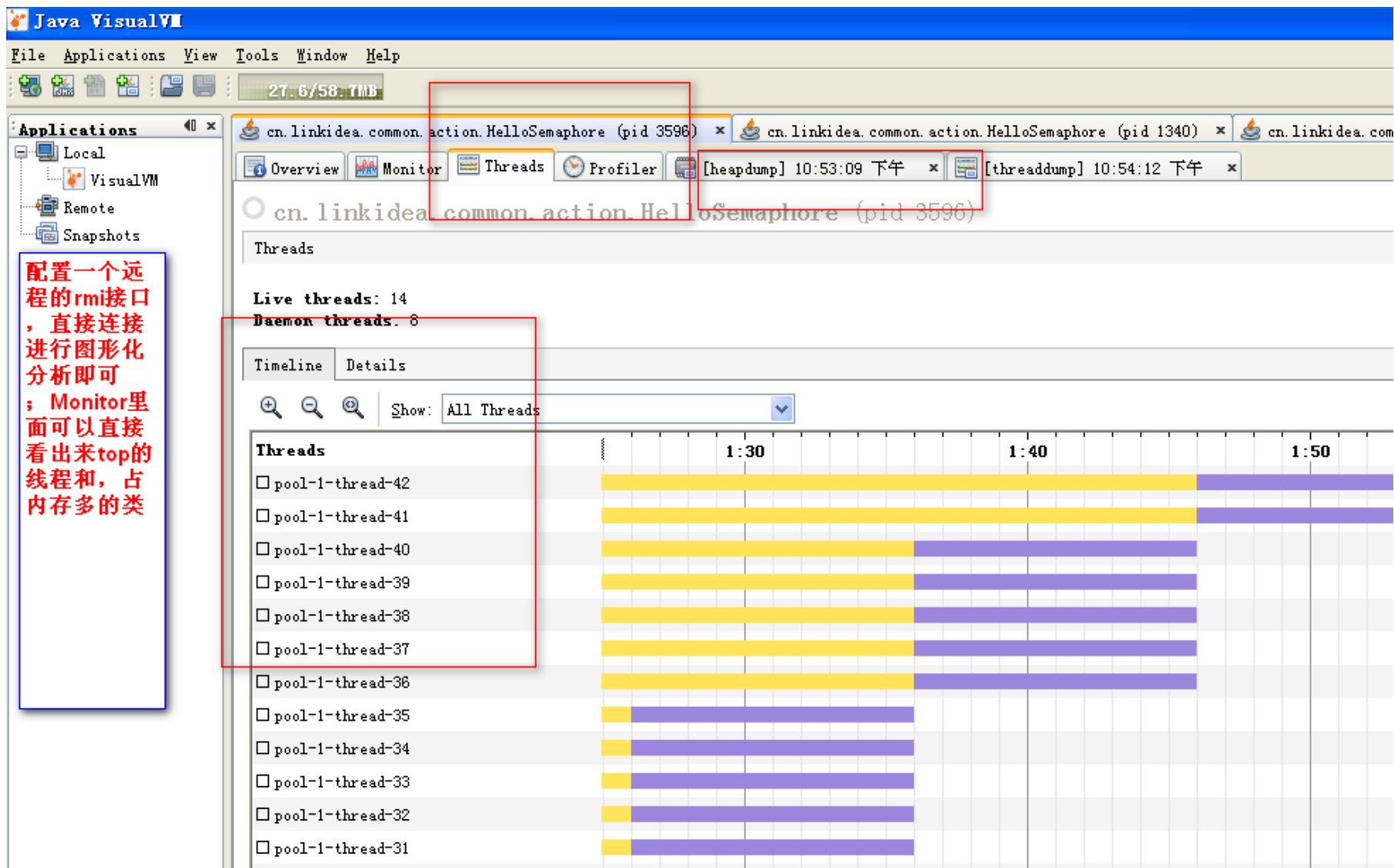
Threads locking Monitors (94 Threads locking)

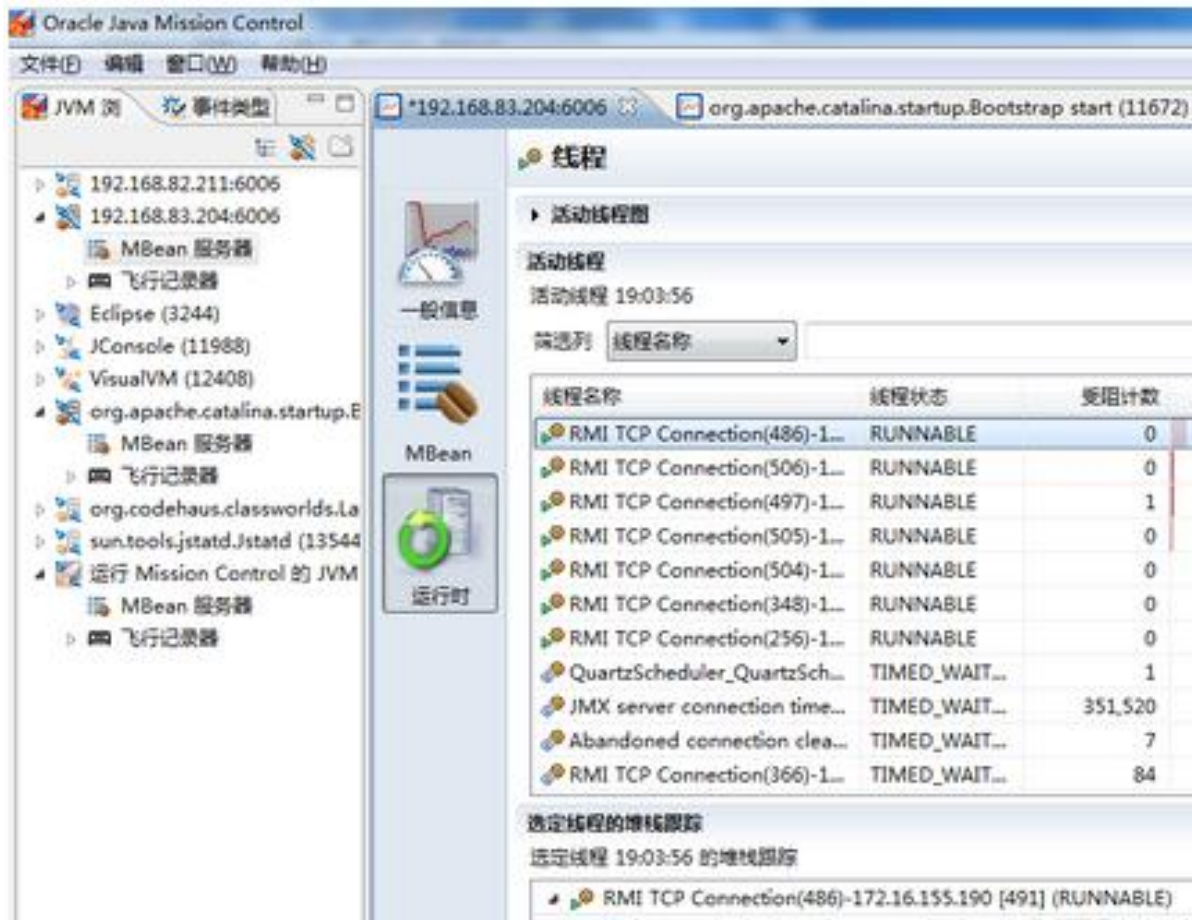
Monitors (142 Monitors)

| Name | Type | Prio | Thread ID | Native ID | State | Address Range |
|------------------------------------|--------|------|----------------|-----------|----------------------|--------------------|
| StreamCompletionService thread-400 | Task | 10 | 14077426145280 | 30302 | waiting on condition | 0x000077432c0000 |
| StreamCompletionService thread-400 | Task | 10 | 14077426145280 | 30303 | waiting on condition | 0x000077432c0000 |
| Attach Listener | Daemon | 10 | 14077426145280 | 30304 | waiting on condition | 0x0000000000000000 |
| http-nio-80-exec-240 | Daemon | 10 | 14077426145280 | 26898 | runnable | 0x000077432c0000 |
| http-nio-80-exec-240 | Daemon | 10 | 14077426145280 | 26897 | waiting on condition | 0x000077432c0000 |
| http-nio-80-exec-238 | Daemon | 10 | 14077426145280 | 26896 | waiting on condition | 0x000077432c0000 |
| http-nio-80-exec-236 | Daemon | 10 | 14077426145280 | 26895 | runnable | 0x000077432c0000 |
| http-nio-80-exec-237 | Daemon | 10 | 14077426145280 | 26894 | waiting on condition | 0x000077432c0000 |
| http-nio-80-exec-236 | Daemon | 10 | 14077426145280 | 26893 | waiting on condition | 0x000077432c0000 |
| http-nio-80-exec-235 | Daemon | 10 | 14077426145280 | 26892 | waiting on condition | 0x000077432c0000 |
| http-nio-80-exec-234 | Daemon | 10 | 14077426145280 | 26891 | runnable | 0x000077432c0000 |
| http-nio-80-exec-233 | Daemon | 10 | 14077426145280 | 26890 | waiting on condition | 0x000077432c0000 |
| http-nio-80-exec-232 | Daemon | 10 | 14077426145280 | 26889 | waiting on condition | 0x000077432c0000 |
| http-nio-80-exec-231 | Daemon | 10 | 14077426145280 | 26888 | waiting on condition | 0x000077432c0000 |
| http-nio-80-exec-230 | Daemon | 10 | 14077426145280 | 26887 | waiting on condition | 0x000077432c0000 |
| http-nio-80-exec-229 | Daemon | 10 | 14077426145280 | 26886 | waiting on condition | 0x000077432c0000 |
| Thread-24103 | Daemon | 10 | 14077426145280 | 26716 | runnable | 0x000077432c0000 |
| Thread-24101 | Daemon | 10 | 14077426145280 | 26712 | runnable | 0x000077432c0000 |
| http-nio-80-exec-228 | Daemon | 10 | 14077426145280 | 26730 | waiting on condition | 0x000077432c0000 |
| http-nio-80-exec-227 | Daemon | 10 | 14077426145280 | 26729 | runnable | 0x000077432c0000 |
| http-nio-80-exec-226 | Daemon | 10 | 14077426145280 | 26728 | runnable | 0x000077432c0000 |
| http-nio-80-exec-225 | Daemon | 10 | 14077426145280 | 26725 | runnable | 0x000077432c0000 |
| http-nio-80-exec-224 | Daemon | 10 | 14077426145280 | 26722 | waiting on condition | 0x000077432c0000 |
| Thread-24064 | Daemon | 10 | 14077426145280 | 26295 | runnable | 0x000077432c0000 |

速来千禧吧，极力推荐使用此工具！

线程的监控方法(3) jvisualvm.exe





还有很多等等

注意事项

- 1: 百变不离其宗，需要日积月累，不断弄清楚每个线程里面的类的意思，才好分析问题；
- 2: 没有必要为了线程而线程；
- 3: 其实APP应用的IOS和Android也好，只要弄清楚里面的线程运行机制，就可以掌握其精髓；时间有限就不总结APP里面的线程原理了，欢迎下次交流！

Thank You!

- 欢迎交流！
- 共同学习与努力！
- Zhangzhenhua_846@126.com
- 张振华.Jack
- 流传的时候请注意版权
- QQ:494460705