多线程面试题汇总

1. 有T1、T2、T3三个线程，如何怎样保证T2在T1执行完后执行，T3在T2执行完后执行?

使用join方法。

join方法的功能是使异步执行的线程变成同步执行。即调用线程实例的start方法后，该方法会立即返回，如果调用start方法后，需要使用一个由这个线程计算得到的值，就必须使用join方法。如果不使用join方法，就不能保证当执行到start方法后面的某条语句时，这个线程一定会执行完。而使用join方法后，直到这个线程退出，程序才会往下执行。

2.Java中的Lock接口，比起synchronized，优势在哪里?

如果需要实现一个高效的缓存，它允许多个用户读，但只允许一个用户写，以此来保持它的完整性，如何实现?

Lock接口最大的优势是为读和写分别提供了锁。

读写锁ReadWriteLock拥有更加强大的功能，它可细分为读锁和解锁。

读锁可以允许多个进行读操作的线程同时进入，但不允许写进程进入；写锁只允许一个写进程进入，在这期间任何进程都不能再进入。（完全符合题目中允许多个用户读和一个用户写的条件）

要注意的是每个读写锁都有挂锁和解锁，最好将每一对挂锁和解锁操作都用try、finally来套入中间的代码，这样就会防止因异常的发生而造成死锁得情况。

下面是一个示例程序：

```
import java.util.Random;
import java.util.concurrent.locks.*;
public class ReadWriteLockTest {
 public static void main(String[] args) {
  final TheData myData=new TheData();  //这是各线程的共享数据
  for(int i=0;i<3;i++){ //开启3个读线程
   new Thread(new Runnable(){
    @Override
    public void run() {
     while(true){
      myData.get();
     }
    }
   }).start();
```

```
  }
  for(int i=0;i<3;i++){ //开启3个写线程
   new Thread(new Runnable(){
    @Override
    public void run() {
     while(true){
      myData.put(new Random().nextInt(10000));
     }
    }
   }).start();
  }
 }
}
class TheData{
 private Object data=null;
 private ReadWriteLock rwl=new ReentrantReadWriteLock();
 public void get(){
  rwl.readLock().lock();  //读锁开启，读线程均可进入
  try { //用try finally来防止因异常而造成的死锁
   System.out.println(Thread.currentThread().getName()+"is ready to read");
   Thread.sleep(new Random().nextInt(100));
   System.out.println(Thread.currentThread().getName()+"have read date"+data);
  } catch (InterruptedException e) {
   e.printStackTrace();
  } finally{
   rwl.readLock().unlock(); //读锁解锁
  }
 }
 public void put(Object data){
  rwl.writeLock().lock();  //写锁开启，这时只有一个写线程进入
  try {
   System.out.println(Thread.currentThread().getName()+"is ready to write");
   Thread.sleep(new Random().nextInt(100));
   this.data=data;
   System.out.println(Thread.currentThread().getName()+"have write date"+data);
  } catch (InterruptedException e) {
   e.printStackTrace();
  } finally{
   rwl.writeLock().unlock(); //写锁解锁
  }
 }
}
```

3. java中wait和sleep方法有何不同?

最大的不同是在等待时wait会释放锁，而sleep一直持有锁。Wait通常被用于线程间交互，sleep通常被用于暂停执行。

其它不同有：

- sleep是Thread类的静态方法，wait是Object方法。
- wait，notify和notifyAll只能在同步控制方法或者同步控制块里面使用，而sleep可以在任何地方使用
- sleep必须捕获异常，而wait，notify和notifyAll不需要捕获异常

4.如何用Java实现阻塞队列?

首先，我们要明确阻塞队列的定义：

阻塞队列（BlockingQueue）是一个支持两个附加操作的队列。这两个附加的操作是：在队列为空时，获取元素的线程会等待队列变为非空。当队列满时，存储元素的线程会等待队列可用。 阻塞队列常用于生产者和消费者的场景，生产者是往队列里添加元素的线程，消费者是从队列里拿元素的线程。阻塞队列就是生产者存放元素的容器，而消费者也只从容器里拿元素。

注：有关生产者——消费者问题，可查阅维基百科网址：

http://zh.wikipedia.org/wiki/%E7%94%9F%E4%BA%A7%E8%80%85%E6%B6%88%E8%B4%B9%E8%80%85%E9%97%AE%E9%A2%98

和百度百科网址：

http://baike.baidu.com/view/10800629.htm

阻塞队列的一个简单实现：

```
public class BlockingQueue {
  private List queue = new LinkedList();
  private int  limit = 10;

  public BlockingQueue(int limit){
    this.limit = limit;
  }

  public synchronized void enqueue(Object item)throws InterruptedException  {
    while(this.queue.size() == this.limit) {
      wait();
    }
    if(this.queue.size() == 0) {
      notifyAll();
    }
    this.queue.add(item);
  }

  public synchronized Object dequeue()  throws InterruptedException{
    while(this.queue.size() == 0){
      wait();
    }
    if(this.queue.size() == this.limit){
      notifyAll();
```

```
    }

    return this.queue.remove(0);
  }
}
```

在enqueue和dequeue方法内部，只有队列的大小等于上限（limit）或者下限（0）时，才调用notifyAll方法。如果队列的大小既不等于上限，也不等于下限，任何线程调用enqueue或者dequeue方法时，都不会阻塞，都能够正常的往队列中添加或者移除元素。

5.编写Java代码，解决生产者——消费者问题。

生产者——消费者问题是研究多线程程序时绕不开的经典问题之一，它描述是有一块缓冲区作为仓库，生产者可以将产品放入仓库，消费者则可以从仓库中取走产品。

使用问题4中阻塞队列实现代码来解决。但此不是唯一解决方案。

解决生产者/消费者问题的方法可分为两类：

- 采用某种机制保护生产者和消费者之间的同步；
- 在生产者和消费者之间建立一个管道。

第一种方式有较高的效率，并且易于实现，代码的可控制性较好，属于常用的模式。第二种管道缓冲区不易控制，被传输数据对象不易于封装等，实用性不强。因此建议使用第一种方式来实现。

同步的核心问题在于：**如何保证同一资源被多个线程并发访问时的完整性?**

常用的同步方法是采用信号或加锁机制，保证资源在任意时刻至多被一个线程访问。Java语言在多线程编程上实现了完全对象化，提供了对同步机制的良好支持。

在Java中一共有四种方法支持同步，其中前三个是同步方法，一个是管道方法。管道方法不建议使用，阻塞队列方法在问题4已有描述，现只提供前两种实现方法。

- wait()/notify()方法
- await()/signal()方法
- BlockingQueue阻塞队列方法
- PipedInputStream/PipedOutputStream

生产者类：

```
public class Producer extends Thread { // 每次生产的产品数量
    private int num;

    // 所在放置的仓库
    private Storage storage;

    // 构造函数，设置仓库
    public Producer(Storage storage) {
        this.storage = storage;
    }
```

```
    // 线程run函数
    public void run() {
        produce(num);
    }

    // 调用仓库Storage的生产函数
    public void produce(int num) {
        storage.produce(num);
    }

    public int getNum() {
        return num;
    }

    public void setNum(int num) {
        this.num = num;
    }

    public Storage getStorage() {
        return storage;
    }

    public void setStorage(Storage storage) {
        this.storage = storage;
    }
}
```

消费者类：

```
public class Consumer extends Thread { // 每次消费的产品数量
    private int num;

    // 所在放置的仓库
    private Storage storage;

    // 构造函数，设置仓库
    public Consumer(Storage storage) {
        this.storage = storage;
    }

    // 线程run函数
    public void run() {
        consume(num);
    }

    // 调用仓库Storage的生产函数
    public void consume(int num) {
        storage.consume(num);
```

```
    }

    // get/set方法
    public int getNum() {
        return num;
    }

    public void setNum(int num) {
        this.num = num;
    }

    public Storage getStorage() {
        return storage;
    }

    public void setStorage(Storage storage) {
        this.storage = storage;
    }
}
```

仓库类：（wait()/notify()方法）

```
public class Storage { // 仓库最大存储量
    private final int MAX_SIZE = 100;

    // 仓库存储的载体
    private LinkedList<Object> list = new LinkedList<Object>();

    // 生产num个产品
    public void produce(int num) {
        // 同步代码段
        synchronized (list) {
            // 如果仓库剩余容量不足
            while (list.size() + num > MAX_SIZE) {
                System.out.print("【要生产的产品数量】:" + num);
                System.out.println(" 【库存量】:" + list.size() + " 暂时不能执行
生产任务!");

                try {
                    list.wait();// 由于条件不满足，生产阻塞
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }

            // 生产条件满足情况下，生产num个产品
            for (int i = 1; i <= num; ++i) {
                list.add(new Object());
```

```
                }

                System.out.print("【已经生产产品数】:" + num);
                System.out.println(" 【现仓储量为】:" + list.size());

                list.notifyAll();
           }
      }

      // 消费num个产品
      public void consume(int num) {
           // 同步代码段
           synchronized (list) {
                // 如果仓库存储量不足
                while (list.size() < num) {
                     System.out.print("【要消费的产品数量】:" + num);
                     System.out.println(" 【库存量】:" + list.size() + " 暂时不能执行
生产任务!");

                     try {
                          // 由于条件不满足，消费阻塞
                          list.wait();
                     } catch (InterruptedException e) {
                          e.printStackTrace();
                     }
                }

                // 消费条件满足情况下，消费num个产品
                for (int i = 1; i <= num; ++i) {
                     list.remove();
                }

                System.out.print("【已经消费产品数】:" + num);
                System.out.println(" 【现仓储)量为】:" + list.size());

                list.notifyAll();
           }
      }

      // get/set方法
      public LinkedList<Object> getList() {
           return list;
      }

      public void setList(LinkedList<Object> list) {
           this.list = list;
      }

      public int getMAX_SIZE() {
```

```
            return MAX_SIZE;
        }
    }
```

仓库类：（await()/signal()方法）

```
public class Storage { // 仓库最大存储量
    // 仓库最大存储量
    private final int MAX_SIZE = 100;

    // 仓库存储的载体
    private LinkedList<Object> list = new LinkedList<Object>();

    // 锁
    private final Lock lock = new ReentrantLock();

    // 仓库满的条件变量
    private final Condition full = lock.newCondition();

    // 仓库空的条件变量
    private final Condition empty = lock.newCondition();

    // 生产num个产品
    public void produce(int num) {
        // 获得锁
        lock.lock();

        // 如果仓库剩余容量不足
        while (list.size() + num > MAX_SIZE) {
            System.out.print("【要生产的产品数量】:" + num);
            System.out.println(" 【库存量】:" + list.size() + " 暂时不能执行生产任
务!");

            try {
                // 由于条件不满足，生产阻塞
                full.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        // 生产条件满足情况下，生产num个产品
        for (int i = 1; i <= num; ++i) {
            list.add(new Object());
        }

        System.out.print("【已经生产产品数】:" + num);
        System.out.println(" 【现仓储量为】:" + list.size());
```

```
            // 唤醒其他所有线程
            full.signalAll();
            empty.signalAll();

            // 释放锁
            lock.unlock();
        }

        // 消费num个产品
        public void consume(int num) {
            // 获得锁
            lock.lock();

            // 如果仓库存储量不足
            while (list.size() < num) {
                System.out.print("【要消费的产品数量】:" + num);
                System.out.println(" 【库存量】:" + list.size() + " 暂时不能执行生产任
务!");

                try {
                    // 由于条件不满足，消费阻塞
                    empty.await();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }

            // 消费条件满足情况下，消费num个产品
            for (int i = 1; i <= num; ++i) {
                list.remove();
            }

            System.out.print("【已经消费产品数】:" + num);
            System.out.println(" 【现仓储)量为】:" + list.size());

            // 唤醒其他所有线程
            full.signalAll();
            empty.signalAll();

            // 释放锁
            lock.unlock();
        }

        // set/get方法
        public int getMAX_SIZE() {
            return MAX_SIZE;
        }
```

```
    public LinkedList<Object> getList() {
        return list;
    }

    public void setList(LinkedList<Object> list) {
        this.list = list;
    }
}
```

6. 如何解决一个用Java编写的会导致死锁的程序?

Java线程死锁问题往往和一个被称之为哲学家就餐的问题相关联。

注：有关哲学家就餐的问题，可查阅维基百科网址：

http://zh.wikipedia.org/wiki/%E5%93%B2%E5%AD%A6%E5%AE%B6%E5%B0%B1%E9%A4%90%E9%97%AE%E9%A2%98

和百度百科网址：

http://baike.baidu.com/link?url=V-QPP4G1a1PDO1krV6GreFQSp7AQl-KhAP8WGzXw4zl7eeevz3vn07MJMf8SmXfz36CtkDQXMh8kZ36_Fwnfxq

导致死锁的根源在于不适当地运用“synchronized”关键词来管理线程对特定对象的访问。

“synchronized”关键词的作用是，确保在某个时刻只有一个线程被允许执行特定的代码块，因此，被允许执行的线程首先必须拥有对变量或对象的排他性的访问权。当线程访问对象 时，线程会给对象加锁，而这个锁导致其它也想访问同一对象的线程被阻塞，直至第一个线程释放它加在对象上的锁。由于这个原因，在使用“synchronized”关键词时，很容易出现两个线程互相等待对方做出某个动作的情形。

死锁程序例子

```
public class Deadlocker implements Runnable {
    public int flag = 1;
    static Object o1 = new Object(), o2 = new Object();

    public void run() {
        System.out.println("flag=" + flag);
        if (flag == 1) {
            synchronized (o1) {
                try {
                    Thread.sleep(500);
                } catch (Exception e) {
                    e.printStackTrace();
                }
                synchronized (o2) {
                    System.out.println("1");
                }
            }
```

```
            }

            if (flag == 0) {
                synchronized (o2) {
                    try {
                        Thread.sleep(500);
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                    synchronized (o1) {
                        System.out.println("0");
                    }
                }
            }
        }

        public static void main(String[] args) {
            Deadlocker td1 = new Deadlocker();
            Deadlocker td2 = new Deadlocker();
            td1.flag = 1;
            td2.flag = 0;
            Thread t1 = new Thread(td1);
            Thread t2 = new Thread(td2);
            t1.start();
            t2.start();
        }
    }
```

说明：

> 当类的对象flag=1时（T1），先锁定O1,睡眠500毫秒，然后锁定O2；
>
> 而T1在睡眠的时候另一个flag=0的对象（T2）线程启动，先锁定O2,睡眠500毫秒，等待T1释放O1；
>
> T1睡眠结束后需要锁定O2才能继续执行，而此时O2已被T2锁定；
>
> T2睡眠结束后需要锁定O1才能继续执行，而此时O1已被T1锁定；
>
> T1、T2相互等待，都需要对方锁定的资源才能继续执行，从而死锁。

避免死锁的一个通用的经验法则是:当几个线程都要访问共享资源A、B、C时，保证使每个线程都按照同样的顺序去访问它们，比如都先访问A，再访问B和C。

如把 Thread t2 = new Thread(td2); 改成 Thread t2 = new Thread(td1);

还有一种方法是对对象进行synchronized，加大锁定的粒度，如上面的例子中使得进程锁定当前对象，而不是逐步锁定当前对象的两个子对象o1和o2。这样就在t1锁定o1之后， 即使发生休眠，当前对象仍然被t1锁定，t2不能打断t1去锁定o2，等t1休眠后再锁定o2，获取资源，执行成功。然后释放当前对象t2，接着t1继续运行。

代码如下：

```
public class Deadlocker implements Runnable {
     public int flag = 1;
     static Object o1 = new Object(), o2 = new Object();

     public synchronized void run() {
          System.out.println("flag=" + flag);
          if (flag == 1) {
               try {
                    Thread.sleep(500);
               } catch (Exception e) {
                    e.printStackTrace();
               }

               System.out.println("1");
          }
          if (flag == 0) {
               try {
                    Thread.sleep(500);
               } catch (Exception e) {
                    e.printStackTrace();
               }
               System.out.println("0");
          }
     }

     public static void main(String[] args) {
          Deadlocker td1 = new Deadlocker();
          Deadlocker td2 = new Deadlocker();
          td1.flag = 1;
          td2.flag = 0;
          Thread t1 = new Thread(td1);
          Thread t2 = new Thread(td2);
          t1.start();
          t2.start();
     }
}
```

> 代码修改成public synchronized void run(){..},去掉子对象锁定。对于一个成员方法加synchronized关键字，实际上是以这个成员方法所在的对象本身作为对象锁。此例中，即对td1，td2这两个Deadlocker 对象进行加锁。

第三种解决死锁的方法是使用实现Lock接口的重入锁类（ReentrantLock），代码如下：

```
public class Deadlocker implements Runnable {
     public int flag = 1;
     static Object o1 = new Object(), o2 = new Object();
     private final Lock lock = new ReentrantLock();

     public boolean checkLock() {
          return lock.tryLock();
     }

     public void run() {
          if (checkLock()) {
               try {
                    System.out.println("flag=" + flag);
                    if (flag == 1) {
                         try {
                              Thread.sleep(500);
                         } catch (Exception e) {
                              e.printStackTrace();
                         }

                         System.out.println("1");
                    }
                    if (flag == 0) {
                         try {
                              Thread.sleep(500);
                         } catch (Exception e) {
                              e.printStackTrace();
                         }
                         System.out.println("0");
                    }
               } finally {
                    lock.unlock();
               }
          }
     }

     public static void main(String[] args) {
          Deadlocker td1 = new Deadlocker();
          Deadlocker td2 = new Deadlocker();
          td1.flag = 1;
          td2.flag = 0;
          Thread t1 = new Thread(td1);
```

```
            Thread t2 = new Thread(td2);
            t1.start();
            t2.start();
        }
    }
```

说明：

> 代码行lock.tryLock()是测试对象操作是否已在执行中，如果已在执行中则不再执行此对象操作，立即返回false，达到忽略对象操作的效果。

7. 什么是原子操作，Java中的原子操作是什么？

所谓原子操作是指不会被线程调度机制打断的操作；这种操作一旦开始，就一直运行到结束，中间切换到另一个线程。

java中的原子操作介绍：

jdk1.5的包为java.util.concurrent.atomic

这个包里面提供了一组原子类。其基本特性就是在多线程环境下，当有多个线程同时执行这些类的实例包含的方法时，具有排他性。

即当某个线程进入方法，执行其中的指令时，不会被其他线程打断，而别的线程就像锁一样，一直等到该方法执行完成，才由JVM从等待队列中选择另一个线程进入，这只是一种逻辑上的理解。实际上是借助硬件的相关指令来实现的，但不会阻塞线程（synchronized 会把别的等待的线程挂，或者说只是在硬件级别上阻塞了)。

其中的类可以分成4组

- AtomicBoolean，AtomicInteger，AtomicLong，AtomicReference
- AtomicIntegerArray，AtomicLongArray
- AtomicLongFieldUpdater，AtomicIntegerFieldUpdater，AtomicReferenceFieldUpdater
- AtomicMarkableReference，AtomicStampedReference，AtomicReferenceArray

Atomic类的作用

- 使得让对单一数据的操作，实现了原子化
- 使用Atomic类构建复杂的，无需阻塞的代码
- 访问对2个或2个以上的atomic变量（或者对单个atomic变量进行2次或2次以上的操作）通常认为是需要同步的，以达到让这些操作能被作为一个原子单元。

AtomicBoolean , AtomicInteger, AtomicLong, AtomicReference 这四种基本类型用来处理布尔，整数，长整数，对象四种数据。

- 构造函数（两个构造函数）
    - 默认的构造函数：初始化的数据分别是false，0，0，null
    - 带参构造函数：参数为初始化的数据

- set( )和get( )方法：可以原子地设定和获取atomic的数据。类似于volatile，保证数据会在主存中设置或读取
- getAndSet( )方法
  - 原子的将变量设定为新数据，同时返回先前的旧数据
  - 其本质是get( )操作，然后做set( )操作。尽管这2个操作都是atomic，但是他们合并在一起的时候，就不是atomic。在Java的源程序的级别上，如果不依赖synchronized的机制来完成这个工作，是不可能的。只有依靠native方法才可以。
- compareAndSet( ) 和weakCompareAndSet( )方法
  - 这两个方法都是conditional modifier方法。这2个方法接受2个参数，一个是期望数据(expected)，一个是新数据(new)；如果atomic里面的数据和期望数据一致，则将新数据设定给atomic的数据，返回true，表明成功；否则就不设定，并返回false。
- 对于AtomicInteger、AtomicLong还提供了一些特别的方法。getAndIncrement( )、incrementAndGet( )、getAndDecrement( )、decrementAndGet ( )、addAndGet( )、getAndAdd( )以实现一些加法，减法原子操作。(注意 --i、++i不是原子操作，其中包含有3个操作步骤：第一步，读取i；第二步，加1或减1；第三步：写回内存)

例子-使用AtomicReference创建线程安全的堆栈

```
public class LinkedStack<T> {
     private AtomicReference<Node<T>> stacks = new AtomicReference<Node<T>>();

     public T push(T e) {
          Node<T> oldNode, newNode;
          while (true) { //这里的处理非常的特别，也是必须如此的。
               oldNode = stacks.get();
               newNode = new Node<T>(e, oldNode);
               if (stacks.compareAndSet(oldNode, newNode)) {
                    return e;
               }
          }
     }

     public T pop() {
          Node<T> oldNode, newNode;
          while (true) {
               oldNode = stacks.get();
               newNode = oldNode.next;
               if (stacks.compareAndSet(oldNode, newNode)) {
                    return oldNode.object;
               }
          }
     }

     private static final class Node<T> {
          private T object;
          private Node<T> next;
```

```
            private Node(T object, Node<T> next) {
                this.object = object;
                this.next = next;
            }
        }
    }
```

8. Java中的volatile关键字是什么作用？怎样使用它？在Java中它跟synchronized方法有什么不同？

volatile在多线程中是用来同步变量的。 线程为了提高效率，将某成员变量(如A)拷贝了一份（如B），线程中对A的访问其实访问的是B。只在某些动作时才进行A和B的同步。因此存在A和B不一致的情况。

volatile就是用来避免这种情况的。volatile告诉jvm， 它所修饰的变量不保留拷贝，直接访问主内存中的（也就是上面说的A) 变量。

一个变量声明为volatile，就意味着这个变量是随时会被其他线程修改的，因此不能将它cache在线程memory中。以下例子展现了volatile的作用：

```
public class StoppableTask extends Thread {
     private volatile boolean pleaseStop;

     public void run() {
          while (!pleaseStop) {
               // do some stuff...
          }
     }

     public void tellMeToStop() {
          pleaseStop = true;
     }
}
```

假如pleaseStop没有被声明为volatile，线程执行run的时候检查的是自己的副本，就不能及时得知其他线程已经调用tellMeToStop()修改了pleaseStop的值。

Volatile一般情况下不能代替sychronized，因为volatile不能保证操作的原子性，即使只是i++，实际上也是由多个原子操作组成：

```
read i; inc; write i,
```

假如多个线程同时执行i++，volatile只能保证他们操作的i是同一块内存，但依然可能出现写入脏数据的情况。如果配合Java 5增加的atomic wrapper classes，对它们的increase之类的操作就不需要sychronized。

volatile和synchronized的不同是最容易解释清楚的。volatile是变量修饰符，而synchronized则作用于一段代码或方法；看如下三句get代码：

```
int i1;
   volatile int i2;
   int i3;

   int geti1() {
        return i1;
   }

   int geti2() {
        return i2;
   }

   synchronized int geti3() {
        return i3;
   }
```

得到存储在当前线程中i1的数值。多个线程有多个i1变量拷贝，而且这些i1之间可以互不相同。换句话说，另一个线程可能已经改变了它线程内的 i1值，而这个值可以和当前线程中的i1值不相同。事实上，Java有个思想叫“主”内存区域，这里存放了变量目前的“准确值”。每个线程可以有它自己的 变量拷贝，而这个变量拷贝值可以和“主”内存区域里存放的不同。因此实际上存在一种可能：“主”内存区域里的i1值是1，线程1里的i1值是2，线程2里 的i1值是3——这在线程1和线程2都改变了它们各自的i1值，而且这个改变还没来得及传递给“主”内存区域或其他线程时就会发生。

而 geti2()得到的是“主”内存区域的i2数值。用volatile修饰后的变量不允许有不同于“主”内存区域的变量拷贝。换句话说，一个变量经 volatile修饰后在所有线程中必须是同步的；任何线程中改变了它的值，所有其他线程立即获取到了相同的值。理所当然的，volatile修饰的变量存取时比一般变量消耗的资源要多一点，因为线程有它自己的变量拷贝更为高效。

既然volatile关键字已经实现了线程间数据同步，又要 synchronized干什么呢？它们之间有两点不同。首先，synchronized获得并释放监视器——如果两个线程使用了同一个对象锁，监视器能强制保证代码块同时只被一个线程所执行——这是众所周知的事实。但是，synchronized也同步内存：事实上，synchronized在“ 主”内存区域同步整个线程的内存。因此，执行geti3()方法做了如下几步：

1.线程请求获得监视this对象的对象锁（假设未被锁，否则线程等待直到锁释放）

2.线程内存的数据被消除，从“主”内存区域中读入

3.代码块被执行

4,对于变量的任何改变现在可以安全地写到“主”内存区域中（不过geti3()方法不会改变变量值）

5.线程释放监视this对象的对象锁

因此volatile只是在线程内存和“主”内存间同步某个变量的值，而synchronized通过锁定和解锁某个监视器同步所有变量的值。显然synchronized要比volatile消耗更多资源。

9. 什么是竞争条件？如何发现和解决竞争？

两个线程同步操作同一个对象，使这个对象的最终状态不明——叫做竞争条件。竞争条件可以在任何应该由程序员保证原子操作的，而又忘记使用synchronized的地方。

唯一的解决方案就是加锁。

Java有两种锁可供选择:

- 对象或者类(class)的锁。每一个对象或者类都有一个锁。使用synchronized关键字获取。synchronized加到static方法上面就使用类锁，加到普通方法上面就用对象锁。除此之外synchronized还可以用于锁定关键区域块(Critical Section)。 synchronized之后要制定一个对象(锁的携带者)，并把关键区域用大括号包裹起来。synchronized(this){// critical code}。
- 显示构建的锁(java.util.concurrent.locks.Lock)，调用lock的lock方法锁定关键代码。

10.如何使用thread dump？如何分析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显示所有线程在某一点状态的thread-dump的能力。虽然各个 Java虚拟机打印输出格式上略微有一些不同，但是Thread dumps出来的信息包含线程；线程的运行状态、标识和调用的堆栈；调用的堆栈包含完整的类名，所执行的方法，如果可能的话还有源代码的行数。

SUN JVM 产生ThreadDumpSolaris OS

```
<ctrl>-'\' (Control-Backslash)
kill -QUIT <PID>
```

HP-UX/UNIX/Linux

```
Kill -3 <PID>
```

Windows

```
直接对MSDOS窗口的程序按Ctrl-break
```

有些Java应用服务器是在控制台上运行，如Weblogic，为了方便获取threaddump信息，在weblogic启动的时候，会将其标准输出重定向到一个文件， 用"`nohup ./startWebLogic .sh > log.out &`"命令，执行"`kill -3 <pid>`"，Thread dump就会输出到log.out里。

Tomcat的Thread Dump会输出到命令行控制台或者logs的catalina.out文件里。为了反映线程状态的动态变化，需要接连做三次以上thread dump，每次间隔10-20s。

IBM JVM 产生Thread Dump

在AIX上用IBM的JVM，内存溢出时默认地会产生javacore文件（关于cpu的）和heapdump文件(关于内存的)。 如果没有,则参照下列方法：

1. 在server启动前设置下面环境变量（可以加在启动脚本中）

```
export IBM_HEAPDUMP=true
```

export IBM_HEAP_DUMP=true

export IBM_HEAPDUMP_OUTOFMEMORY=true

export IBM_HEAPDUMPDIR=

2. 用set命令检查参数设置，确保没有设置DISABLE_JAVADUMP，然后启动server

3. 执行kill -3 命令可以生成javacore文件和heapdump文件

拿到java thread dump后，你要做的就是查找"waiting for monitor entry"的thread，如果大量thread都在等待给同一个地址上锁（因为对于Java，一个对象只有一把锁），这说明很可能死锁发生了。比如：

"service-j2ee" prio=5 tid=0x024f1c28 nid=0x125 waiting for monitor entry [62a3e000..62a3f690] [27/Jun/2006:10:03:08] WARNING (26140): CORE3283: stderr: at com.sun.enterprise.resource.IASNonSharedResourcePool.internalGetResource(IASNonSharedResourcePool.java:625) [27/Jun/2006:10:03:08] WARNING (26140): CORE3283: stderr: - waiting to lock <0x965d8110> (a com.sun.enterprise.resource.IASNonSharedResourcePool) [27/Jun/2006:10:03:08] WARNING (26140): CORE3283: stderr: at com.sun.enterprise.resource.IASNonSharedResourcePool.getResource(IASNonSharedResourcePool.java:520) .....

为了确定问题，常常需要在隔两分钟后再次收集一次thread dump，如果得到的输出相同，仍然是大量thread都在等待给同一个地址上锁，那么肯定是死锁了。

如何找到当前持有锁的线程是解决问题的关键。方法是搜索thread dump，查找"`locked <0x965d8110>`"， 找到持有锁的线程。

[27/Jun/2006:10:03:08] WARNING (26140): CORE3283: stderr: "Thread-20" daemon prio=5 tid=0x01394f18 nid=0x109 runnable [6716f000..6716fc28]

[27/Jun/2006:10:03:08] WARNING (26140): CORE3283: stderr: at java.net.SocketInputStream.socketRead0(Native Method)

[27/Jun/2006:10:03:08] WARNING (26140): CORE3283: stderr: at java.net.SocketInputStream.read(SocketInputStream.java:129)

[27/Jun/2006:10:03:08] WARNING (26140): CORE3283: stderr: at oracle.net.ns.Packet.receive(Unknown Source)

[27/Jun/2006:10:03:08] WARNING (26140): CORE3283: stderr: at oracle.net.ns.DataPacket.receive(Unknown Source)

[27/Jun/2006:10:03:08] WARNING (26140): CORE3283: stderr: at oracle.net.ns.NetInputStream.getNextPacket(Unknown Source)

[27/Jun/2006:10:03:08] WARNING (26140): CORE3283: stderr: at oracle.net.ns.NetInputStream.read(Unknown Source)

[27/Jun/2006:10:03:08] WARNING (26140): CORE3283: stderr: at oracle.net.ns.NetInputStream.read(Unknown Source)

[27/Jun/2006:10:03:08] WARNING (26140): CORE3283: stderr: at oracle.net.ns.NetInputStream.read(Unknown Source)

[27/Jun/2006:10:03:08] WARNING (26140): CORE3283: stderr: at oracle.jdbc.ttc7.MAREngine.unmarshalUB1(MAREngine.java:929)

[27/Jun/2006:10:03:08] WARNING (26140): CORE3283: stderr: at oracle.jdbc.ttc7.MAREngine.unmarshalSB1(MAREngine.java:893)

[27/Jun/2006:10:03:08] WARNING (26140): CORE3283: stderr: at oracle.jdbc.ttc7.Ocommoncall.receive(Ocommoncall.java:106)

[27/Jun/2006:10:03:08] WARNING (26140): CORE3283: stderr: at oracle.jdbc.ttc7.TTC7Protocol.logoff(TTC7Protocol.java:396)

[27/Jun/2006:10:03:08] WARNING (26140): CORE3283: stderr: - locked <0x954f47a0> (a oracle.jdbc.ttc7.TTC7Protocol)

[27/Jun/2006:10:03:08] WARNING (26140): CORE3283: stderr: at oracle.jdbc.driver.OracleConnection.close(OracleConnection.java:1518)

[27/Jun/2006:10:03:08] WARNING (26140): CORE3283: stderr: - locked <0x954f4520> (a oracle.jdbc.driver.OracleConnection)

[27/Jun/2006:10:03:08] WARNING (26140): CORE3283: stderr: at com.sun.enterprise.resource.JdbcUrlAllocator.destroyResource(JdbcUrlAllocator.java:122)

[27/Jun/2006:10:03:08] WARNING (26140): CORE3283: stderr: at com.sun.enterprise.resource.IASNonSharedResourcePool.destroyResource(IASNonSharedResourcePool.java:872)

[27/Jun/2006:10:03:08] WARNING (26140): CORE3283: stderr: at com.sun.enterprise.resource.IASNonSharedResourcePool.resizePool(IASNonSharedResourcePool.java:1086)

[27/Jun/2006:10:03:08] WARNING (26140): CORE3283: stderr: - locked <0x965d8110> (a com.sun.enterprise.resource.IASNonSharedResourcePool)

[27/Jun/2006:10:03:08] WARNING (26140): CORE3283: stderr: at com.sun.enterprise.resource.IASNonSharedResourcePool$Resizer.run(IASNonSharedResourcePool.java:1178)

[27/Jun/2006:10:03:08] WARNING (26140): CORE3283: stderr: at java.util.TimerThread.mainLoop(Timer.java:432)

[27/Jun/2006:10:03:08] WARNING (26140): CORE3283: stderr: at java.util.TimerThread.run(Timer.java:382)

在这个例子里，持有锁的线程在等待Oracle返回结果，却始终等不到响应，因此发生了死锁。

如果持有锁的线程还在等待给另一个对象上锁，那么还是按上面的办法顺藤摸瓜，直到找到死锁的根源为止。 另外，在thread dump里还会经常看到这样的线程，它们是等待一个条件而主动放弃锁的线程。例如：

"Thread-1" daemon prio=5 tid=0x014e97a8 nid=0x80 in Object.wait() [68c6f000..68c6fc28] at java.lang.Object.wait(Native Method) - waiting on <0x95b07178> (a java.util.LinkedList) at com.iplanet.ias.util.collection.BlockingQueue.remove(BlockingQueue.java:258)

- locked <0x95b07178> (a java.util.LinkedList) at com.iplanet.ias.util.threadpool.FastThreadPool$ThreadPoolThread.run(FastThreadPool.java:241)
  at java.lang.Thread.run(Thread.java:534)

有时也会需要分析这类线程，尤其是线程等待的条件。

其实，Java thread dump并不只用于分析死锁，其它Java应用运行时古怪的行为都可以用thread dump来分析。

在Java SE 5里，增加了jstack的工具，也可以获取thread dump。在Java SE 6里， 通过jconsole的图形化工具也可以方便地查找涉及object monitors 和java.util.concurrent.locks死锁。

参考文章：
<http://www.cnblogs.com/zhengyun_ustc/archive/2013/01/06/dumpanalysis.html>

##**11. 为什么调用start()方法时会执行run()方法，而不能直接调用run()方法？**

调用start()方法时，将会创建新的线程，并且执行在run()方法里的代码。但如果直接调用 run()方法，它不会创建新的线程也不会执行调用线程的代码。

##**12. Java中怎样唤醒一个阻塞的线程？**

如果是IO阻塞，创建线程时，加一个数量的阈值，超过该值后则不再创建。或者为每个线程设置标志变量标志该线程是否已经束，三就是直接加入线程组去管理。

如果线程因为调用 wait()、sleep()、或者join()方法而导致的阻塞，你可以中断线程，并且通过抛出InterruptedException来唤醒它。

##**13. Java中CycliBarriar和CountdownLatch有什么区别？**

CountdownLatch: 一个线程(或者多个)，等待另外N个线程完成某个事情之后才能执行。

CycliBarriar：N个线程相互等待，任何一个线程完成之前，所有的线程都必须等待。

这样应该就清楚一点了，对于CountDownLatch来说，重点是那个“一个线程”，是它在等待，而另外那N的线程在把“某个事情”做完之后可以继续等待，也可以终止。

而对于CyclicBarrier来说，重点是那N个线程，他们之间任何一个没有完成，所有的线程都必须等待。

1．CyclicBarrier可以多次使用，CountDownLatch只能用一次（为0后不可变）
2．Barrier是等待指定数量线程到达再继续处理；Latch是等待指定事件变为指定状态后发生再继续处理，对于CountDown就是计数减为0的事件，但你也可以实现或使用其他Latch，就不是这个事件了...
3．Barrier是等待指定数量任务完成，Latch是等待其他任务完成指定状态的改变再继续

##**14．什么是不可变对象，它对写并发应用有什么帮助？**

不可变对象（英语：Immutable object）是一种对象,在被创造之后，它的状态就不可以被改变。

由于它不可更改，并发时不需要其他额外的同步保证，故相比其他的锁同步等方式的并发性能要好。

衍生问题：**为什么String是不可变的？**

- 字符串常量池的需要

字符串常量池(String pool，String intern pool，String保留池）是Java堆内存中一个特殊的存储区域，当创建一个String对象时,假如此字符串值已经存在于常量池中,则不会创建一个新的对象,而是引用已经存在的对象。

如下面的代码所示,将会在堆内存中只创建一个实际String对象.

String s1 = "abcd";
String s2 = "abcd";

示意图如下所示：![enter image description here]
(http://www.ituring.com.cn/download/01frm63XPsMd)

假若字符串对象允许改变,那么将会导致各种逻辑错误,比如改变一个对象会影响到另一个独立对象．严格来说，这种常量池的思想,是一种优化手段.

请思考：**假若代码如下所示，s1和s2还会指向同一个实际的String对象吗?**

String s1= "ab" + "cd";
String s2= "abc" + "d";

也许这个问题违反新手的直觉，但是考虑到现代编译器会进行常规的优化，所以他们都会指向常量池中的同一个对象．或者,你可以用 jd-gui 之类的工具查看一下编译后的class文件．

- 允许String对象缓存HashCode

Java中String对象的哈希码被频繁地使用，比如在hashMap 等容器中。

字符串不变性保证了hash码的唯一性,因此可以放心地进行缓存.这也是一种性能优化手段,意味着不必每次都去计算新的哈希码．在String类的定义中有如下代码：

private int hash;//用来缓存HashCode

- 安全性

String被许多的Java类(库)用来当做参数,例如 网络连接地址URL,文件路径path,还有反射机制所需要的String参数等，假若String不是固定不变的,将会引起各种安全隐患。

假如有如下的代码：

boolean connect(String s) {

```
  if (!isSecure(s)) {
       throw new SecurityException();
  }
  // 如果在其他地方可以修改String,那么此处就会引起各种预料不到的问题/错误
  causeProblem(s);
```

}

##**15．多线程环境中遇到的常见问题是什么？如何解决？**

多线程和并发程序中常遇到的有Memory-interface、竞争条件、死锁、活锁和饥饿。

Memory-interface（暂无资料）[X]

竞争条件见第9题

死锁见第6题

活锁和饥饿：

活锁(英文 livelock)

概念：指事物1可以使用资源，但它让其他事物先使用资源；事物2可以使用资源，但它也让其他事物先使用资源，于是两者一直谦让，都无法使用资源。活锁有一定几率解开。而死锁（deadlock）是无法解开的。

解决：避免活锁的简单方法是采用先来先服务的策略。当多个事务请求封锁同一数据对象时，封锁子系统按请求封锁的先后次序对事务排队，数据对象上的锁一旦释放就批准申请队列中第一个事务获得锁。

饥饿

概念：是指如果事务T1封锁了数据R,事务T2又请求封锁R，于是T2等待。T3也请求封锁R，当T1释放了R上的封锁后，系统首先批准了T3的请 求，T2仍然等待。然后T4又请求封锁R，当T3释放了R上的封锁之后，系统又批准了T4的请求.....T2可能永远等待，这就是饥饿。

解决： 公平锁： 每一个调用lock()的线程都会进入一个队列，当解锁后，只有队列里的第一个线程被允许锁住Farlock实例，所有其它的线程都将处于等待状态，直到他们处于队列头部。

代码示例 公平锁类：

public class FairLock {

```
 private boolean isLocked = false;
 private Thread lockingThread = null;
 private List<QueueObject> waitingThreads = new ArrayList<QueueObject>();

 public void lock() throws InterruptedException {
      QueueObject queueObject = new QueueObject();
      boolean isLockedForThisThread = true;
      synchronized (this) {
           waitingThreads.add(queueObject);
      }
      while (isLockedForThisThread) {
           synchronized (this) {
                isLockedForThisThread = isLocked || waitingThreads.get(0) !=
queueObject;
                if (!isLockedForThisThread) {
                     isLocked = true;
                     waitingThreads.remove(queueObject);
                     lockingThread = Thread.currentThread();
                     return;
                }
           }
           try {
                queueObject.doWait();
           } catch (InterruptedException e) {
                synchronized (this) {
                     waitingThreads.remove(queueObject);
                }
                throw e;
```

```
            }
        }
    }

    public synchronized void unlock() {
        if (this.lockingThread != Thread.currentThread()) {
            throw new IllegalMonitorStateException("Calling thread has not locked
    this lock");
        }
        isLocked = false;
        lockingThread = null;
        if (waitingThreads.size() > 0) {
            waitingThreads.get(0).doNotify();
        }
    }
```

}

队列对象类：

public class QueueObject {

```
    private boolean isNotified = false;
```

```
    public synchronized void doWait() throws InterruptedException {
        while (!isNotified) {
            this.wait();
        }
        this.isNotified = false;
    }

    public synchronized void doNotify() {
        this.isNotified = true;
        this.notify();
    }

    public boolean equals(Object o) {
        return this == o;
    }
```

}

说明：

首先lock()方法不再声明为synchronized，取而代之的是对必需同步的代码，在synchronized中进行嵌套。 FairLock新创建一个QueueObject的实例，并对每个调用lock()的线程进行入队列。调用

unlock()的线程将从队列头部获取QueueObject，并对其调用doNotify()，用以唤醒在该对象上等待的线程。通过这种方式，在同一时间仅有一个等待线程获得唤醒，而不是所有的等待线程。这也是实现了FairLock公平性。

注意，在同一个同步块中，锁状态依然被检查和设置，以避免出现滑漏条件。还有，QueueObject实际是一个semaphore。doWait()和doNotify()方法在QueueObject中保存着信号。这样做以避免一个线程在调用queueObject.doWait()之前被另一个调用unlock()并随之调用 queueObject.doNotify()的线程重入，从而导致信号丢失。queueObject.doWait()调用放置在 synchronized(this)块之外，以避免被monitor嵌套锁死，所以只要没有线程在lock方法的 synchronized(this)块中执行，另外的线程都可以被解锁。

最后，注意到queueObject.doWait()在try – catch块中是怎样调用的。在InterruptedException抛出的情况下，线程得以离开lock()，并需让它从队列中移除。

##**16. 在java中绿色线程和本地线程区别? **

绿色线程执行用户级别的线程，且一次只使用一个OS线程。 本地线程用的是OS线程系统，在每个JAVA线程中使用一个OS线程。 在执行java时，可通过使用-green或 -native标志来选择所用线程是绿色还是本地。

##**17. 线程与进程的区别? **

线程是指进程内的一个执行单元,也是进程内的可调度实体.

与进程的区别:

- 地址空间:进程内的一个执行单元;进程至少有一个线程;它们共享进程的地址空间;而进程有自己独立的地址空间;
- 资源拥有:进程是资源分配和拥有的单位,同一个进程内的线程共享进程的资源
- 线程是处理器调度的基本单位,但进程不是.
- 二者均可并发执行.

进程和线程都是由操作系统所体会的程序运行的基本单元，系统利用该基本单元实现系统对应用的并发性。进程和线程的区别在于:

简而言之,一个程序至少有一个进程,一个进程至少有一个线程。线程的划分尺度小于进程，使得多线程程序的并发性高。

另外，进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。 线程在执行过程中与进程还是有区别的。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。这就是进程和线程的重要区别。

进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行资源分配和调度的一个独立单位.

线程是进程的一个实体,是CPU调度和分派的基本单位,它是比进程更小的能独立运行的基本单位.线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源.

一个线程可以创建和撤销另一个线程;同一个进程中的多个线程之间可以并发执行.

##**18. 什么是多线程中的上下文切换? **

操作系统管理很多进程的执行。有些进程是来自各种程序、系统和应用程序的单独进程，而某些进程来自被分解为很多进程的应用或程序。当一个进程从内核中移出， 另一个进程成为活动的，这些进程之间便发生了上下文切换。操作系统必须记录重启进程和启动新进程使之活动所需要的所有信息。这些信息被称作上下文，它描述 了进程的现有状态。当进程成为活动的，它可以继续从被抢占的位置开始执行。

当线程被抢占时，就会发生线程之间的上下文切换。如果线程属于相同的进程，它们共享相同的地址空间，因为线程包含在它们所属于的进程的地址空间内。这样，进程需要恢复的多数信息对于线程而言是不需要的。尽管进程和它的线程共享了很多内容，但最为重要的是其地址空间和资源，有些信息对于线程而言是本地且唯一 的，而线程的其他方面包含在进程的各个段的内部。

##**19. 死锁与活锁的区别，死锁与饥饿的区别? **

死锁： 是指两个或两个以上的进程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生 了死锁，这些永远在互相等待的进程称为死锁进程。 由于资源占用是互斥的，当某个进程提出申请资源后，使得有关进程在无外力协助下，永远分配不到必需的资源而无法继续运行，这就产生了一种特殊现象：死锁。

虽然进程在运行过程中，可能发生死锁，但死锁的发生也必须具备一定的条件，死锁的发生必须具备以下四个必要条件。

- **互斥条件**：指进程对所分配到的资源进行排它性使用，即在一段时间内某资源只由一个进程占用。如果此时还有其它进程请求资源，则请求者只能等待，直至占有资源的进程用毕释放。
- **请求和保持条件**：指进程已经保持至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占有，此时请求进程阻塞，但又对自己已获得的其它资源保持不放。
- **不剥夺条件**：指进程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放。
- **环路等待条件**：指在发生死锁时，必然存在一个进程—资源的环形链，即进程集合{P0，P1，P2，···，Pn}中的P0正在等待一个P1占用的资源；P1正在等待P2占用的资源，……，Pn正在等待已被P0占用的资源。

活锁：指事物1可以使用资源，但它让其他事物先使用资源；事物2可以使用资源，但它也让其他事物先使用资源，于是两者一直谦让，都无法使用资源。

活锁有一定几率解开。而死锁（deadlock）是无法解开的。

避免活锁的简单方法是采用先来先服务的策略。当多个事务请求封锁同一数据对象时，封锁子系统按请求封锁的先后次序对事务排队，数据对象上的锁一旦释放就批准申请队列中第一个事务获得锁。

**死锁与饥饿的区别? **见第15题

##**20. Java中用到的线程调度算法是什么? **

计算机通常只有一个CPU,在任意时刻只能执行一条机器指令,每个线程只有获得CPU的使用权才能执行指令．所谓多线程的并发运行,其实是指从宏观上看,各个线程轮流获得CPU的使用权,分别执行各自的任务．在运行池中,会有多个处于就绪状态的线程在等待CPU,JAVA虚拟机的一项任务就是负责线程的调度,线程调度是指按照特定机制为多个线程分配CPU的使用权

java虚拟机采用抢占式调度模型，是指优先让可运行池中优先级高的线程占用CPU，如果可运行池中的线程优先级相同，那么就随机选择一个线程，使其占用CPU。处于运行状态的线程会一直运行，直至它不得不放弃CPU。

一个线程会因为以下原因而放弃CPU。

- java虚拟机让当前线程暂时放弃CPU，转到就绪状态，使其它线程获得运行机会。
- 当前线程因为某些原因而进入阻塞状态
- 线程结束运行

需要注意的是，线程的调度不是跨平台的，它不仅仅取决于java虚拟机，还依赖于操作系统。在某些操作系统中，只要运行中的线程没有遇到阻塞，就不会放弃CPU；

在某些操作系统中，即使线程没有遇到阻塞，也会运行一段时间后放弃CPU，给其它线程运行的机会。java的线程调度是不分时的，同时启动多个线程后，不能保证各个线程轮流获得均等的CPU时间片。 如果希望明确地让一个线程给另外一个线程运行的机会，可以采取以下办法之一。

调整各个线程的优先级

- 让处于运行状态的线程调用Thread.sleep()方法
- 让处于运行状态的线程调用Thread.yield()方法
- 让处于运行状态的线程调用另一个线程的join()方法

##**21.在Java中什么是线程调度？**

见上题

##**22．在线程中，怎么处理不可捕捉异常？**

捕捉异常有两种方法。

- 把线程的错误捕捉到，往上抛
- 通过线程池工厂，把异常捕捉到，uncaughtException往log4j写错误日志

示例代码：

public class TestThread implements Runnable {

```
 public void run() {
      throw new RuntimeException("throwing runtimeException.....");
 }
```

}

当线程代码抛出运行级别异常之后，线程会中断。主线程不受这个影响，不会处理这个，而且根本不能捕捉到这个异常，仍然继续执行自己的代码。

- 方法1）代码示例：

```
public class TestMain {
    public static void main(String[] args) {
        try {
             TestThread t = new TestThread();
             ExecutorService exec = Executors.newCachedThreadPool();
             Future future = exec.submit(t);
             exec.shutdown();
             future.get();//主要是这句话起了作用，调用get()方法，异常重抛出，包装在
ExecutorException
        } catch (Exception e) {//这里可以把线程的异常继续抛出去
             System.out.println("Exception Throw:" + e.getMessage());
        }
   }
```

}

- 方法2）代码示例：

public class HandlerThreadFactory implements ThreadFactory {

```
   public Thread newThread(Runnable runnable) {
        Thread t = new Thread(runnable);
        MyUncaughtExceptionHandler myUncaughtExceptionHandler = new
MyUncaughtExceptionHandler();
        t.setUncaughtExceptionHandler(myUncaughtExceptionHandler);
        return t;
   }
```

}

public class MyUncaughtExceptionHandler implements Thread.UncaughtExceptionHandler {

```
     public void uncaughtException(Thread t, Throwable e) {
        System.out.println("write logger here:" + e);
   }
```

}

public class TestMain {

```
  public static void main(String[] args) {
        try {
             TestThread t = new TestThread();
             ExecutorService exec = Executors.newCachedThreadPool(new
HandlerThreadFactory());
             exec.execute(t);
        } catch (Exception e) {
             System.out.println("Exception Throw:" + e.getMessage());
        }
   }
```

}

##**23. 什么是线程组，为什么在Java中不推荐使用? **

ThreadGroup线程组表示一个线程的集合。此外，线程组也可以包含其他线程组。线程组构成一棵树，在树中，除了初始线程组外，每个线程组都有一个父线程组。

允许线程访问有关自己的线程组的信息，但是不允许它访问有关其线程组的父线程组或其他任何线程组的信息。线程组的目的就是对线程进行管理。

线程组为什么不推荐使用

节省频繁创建和销毁线程的开销，提升线程使用效率。

衍生问题：**线程组和线程池的区别在哪里？**

一个线程的周期分为：创建、运行、销毁三个阶段。处理一个任务时，首先创建一个任务线程，然后执行任务，完了，销毁线程。而线程处于运行状态的时候，才是真的在处理我们交给它的任务，这个阶段才是有效运行时间。所以，我们希望花在创建和销毁线程的资源越少越好。如果不销毁线程，而这个线程又不能被其他的任务调用，那么就会出现资源的浪费。为了提高效率，减少创建和销毁线程带来时间和空间上的浪费，出现了线程池技术。这种技术是在开始就创建一定量的线程，批量处理一类任务，等待任务的到来。任务执行完毕后，线程又可以执行其他的任务。等不再需要线程的时候，就销毁。这样就省去了频繁创建和销毁线程的麻烦。

##**24. 为什么使用Executor框架比使用应用创建和管理线程好? **

大多数并发应用程序是以执行任务（task）为基本单位进行管理的。通常情况下，我们会为每个任务单独创建一个线程来执行。

这样会带来两个问题：

一，大量的线程（>100）会消耗系统资源，使线程调度的开销变大，引起性能下降；

二，对于生命周期短暂的任务，频繁地创建和消亡线程并不是明智的选择。因为创建和消亡线程的开销可能会大于使用多线程带来的性能好处。

一种更加合理的使用多线程的方法是使用线程池（Thread Pool）。 java.util.concurrent 提供了一个灵活的线程池实现：Executor 框架。这个框架可以用于异步任务执行，而且支持很多不同类型的任务执行策略。它还为任务提交和任务执行之间的解耦提供了标准的方法，为使用 Runnable 描述任务提供了通用的方式。 Executor的实现还提供了对生命周期的支持和hook 函数，可以添加如统计收集、应用程序管理机制和监视器等扩展。

在线程池中执行任务线程，可以重用已存在的线程，免除创建新的线程。这样可以在处理多个任务时减少线程创建、消亡的开销。同时，在任务到达时，工作线程通常已经存在，用于创建线程的等待时间不会延迟任务的执行，因此提高了响应性。通过适当的调整线程池的大小，在得到足够多的线程以保持处理器忙碌的同时，还可以防止过多的线程相互竞争资源，导致应用程序在线程管理上耗费过多的资源。

###**25. 在Java中Executor和Executors的区别? **

Executor是接口，是用来执行 Runnable 任务的；它只定义一个方法- execute(Runnable command)；执行 Ruannable 类型的任务。

Executors是类，提供了一系列工厂方法用于创建线程池，返回的线程池都实现了ExecutorService接口。

Executors几个重要方法：

callable(Runnable task)： 将 Runnable 的任务转化成 Callable 的任务

newSingleThreadExecutor()： 产生一个ExecutorService对象，这个对象只有一个线程可用来执行任务，若任务多于一个，任务将按先后顺序执行。

newCachedThreadPool()： 产生一个ExecutorService对象，这个对象带有一个线程池，线程池的大小会根据需要调整，线程执行完任务后返回线程池，供执行下一次任务使用。

newFixedThreadPool(int poolSize)： 产生一个ExecutorService对象，这个对象带有一个大小为 poolSize 的线程池，若任务数量大于 poolSize ，任务会被放在一个 queue 里顺序执行。

newSingleThreadScheduledExecutor()： 产生一个ScheduledExecutorService对象，这个对象的线程池大小为 1 ，若任务多于一个，任务将按先后顺序执行。

newScheduledThreadPool(int poolSize)： 产生一个ScheduledExecutorService对象，这个对象的线程池大小为 poolSize ，若任务数量大于 poolSize ，任务会在一个 queue 里等待执行。

**26. 如何在Windows和Linux上查找哪个线程使用的CPU时间最长? **

其实就是找CPU占有率最高的那个线程

Windows

任务管理器里面看，如下图：

![enter image description here](http://www.ituring.com.cn/download/01frm6XC438a)

Linux

可以用下面的命令将 cpu 占用率高的线程找出来：

$ ps H -eo user,pid,ppid,tid,time,%cpu,cmd –sort=%cpu

这个命令首先指定参数’H'，显示线程相关的信息，格式输出中包含：

user,pid,ppid,tid,time,%cpu,cmd

然后再用%cpu字段进行排序。这样就可以找到占用处理器的线程了。