1) 什么是线程？

线程是操作系统能够进行运算调度的最小单位，它被包含在进程之中，是进程中的实际运作单位。

2) 线程和进程有什么区别？

线程包含在进程中，一个进程可以有很多线程，每条线程并行执行不同的任务。不同的进程使用不同的内存空间，而所有的线程共享一片相同的内存空间。每个线程都拥有单独的栈内存用来存储本地数据。

3) 如何在Java中实现线程？

接口所以你可以继承Thread 类或者实现Runnable接口来重写run()方法实现线程

4) 用Runnable还是Thread？

Java不支持类的多重继承，但允许实现多个接口。Runnable接口更常用

5) Thread 类中的start() 和 run() 方法有什么区别？

start()方法被用来启动新创建的线程，而且start()内部调用了run()方法，这和直接调用run()方法的效果不一样。当你调用run()方法的时候，只会是在原来的线程中调用，没有新的线程启动，start()方法才会启动新线程。

6) Java中Runnable和Callable有什么不同？

它们的主要区别是Callable的 call() 方法可以返回值和抛出异常，而Runnable的run()方法没有这些功能。Callable可以返回装载有计算结果的Future对象。

7) Java中CyclicBarrier 和 CountDownLatch有什么不同？

CyclicBarrier 和 CountDownLatch 都可以用来让一组线程等待其它线程。与 CyclicBarrier 不同的是，CountdownLatch 不能重新使用。

8) Java内存模型是什么？

Java内存模型规定和指引Java程序在不同的内存架构、CPU和操作系统间有确定性地行为。Java内存模型对一个线程所做的变动能被其它线程可见提供了保证。如通过happens-before保证

9) Java中的volatile 变量是什么？

只有成员变量才能使用volatile，多处理器中保证共享变量的可见性，对一个volatile变量的读，总是能看到任意线程对这个volatile变量最后的写入（happens-before规则）

10) 什么是线程安全？Vector是一个线程安全类吗？

如果你的代码所在的进程中有多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，就是线程安全的.Vector 是用同步方法来实现线程安全的, 而和它相似的ArrayList不是线程安全的。

11) Java中什么是竞态条件？

多线程对一些资源的竞争的时候就会产生竞态条件，如果首先要执行的程序竞争失败排到后面执行了，那么整个程序就会出现一些不确定的bugs。这种bugs很难发现而且会重复出现，因为线程间的随机竞争

12) Java中如何停止一个线程？

当run() 或者 call() 方法执行完的时候线程会自动结束,如果要手动结束一个线程，你可以用volatile 布尔变量来退出run()方法的循环或者是取消任务来中断线程。stop, suspend, resume有死锁风险

13) 一个线程运行时发生异常会怎样？

如果异常没有被捕获该线程将会停止执行. 由Thread.UncaughtExceptionHandler处理

14) 如何在两个线程间共享数据？

你可以通过共享对象(如类中的static变量，各个线程通过synchronize等来同步访问)来实现这个目的，或者是使用像阻塞队列这样并发的数据结构

15) Java中notify 和 notifyAll有什么区别？

notify()方法不能唤醒某个具体的线程,一般是只有一个等待线程时用； notifyAll()唤醒所有线程并允许他们争夺锁确保了至少有一个线程能继续运行

16) 为什么wait, notify 和 notifyAll这些方法不在thread类里面？

由于wait，notify和notifyAll都是锁级别的操作，所以把他们定义在Object类中因为锁属于对象。

17) 什么是ThreadLocal变量？

一个以ThreadLocal对象为键，任意对象为值得存储结构，附带在线程上，一个线程可以根据一个ThreadLocal对象查找到绑定在该线程上的值. 每个线程都有一个ThreadLocal就是每个线程都拥有了自己独立的一个变量，竞争条件被彻底消除了

private static final ThreadLocal<Long> TIME = new ThreadLocal<Long>();

TIME.set(System.currentTimeMillis());

TIME.get();

18) 什么是FutureTask？

FutureTask表示一个可以取消的异步运算，一般用于线程池返回结果. 它有启动和取消运算、查询运算是否完成和取回运算结果等方法. 由于FutureTask也是实现Runnable接口所以它可以提交给Executor来执行。

19) Java中interrupted 和 isInterruptedd方法的区别？

| **方法** | **含义** | **备注** |
| --- | --- | --- |
| boolean isInterrupted() | 检查this线程是否设置了中断标志 | 成员方法 |
| **static** boolean interrupted() | 检测**当前线程**是否被中断了，并清除中断标注（也就是放回之前的中断状态，然后重置标志） | 静态方法 |
| void interrupt() | 中断this 线程 | 成员方法 |

interrupt（）是用来设置中断状态的。返回true说明中断状态被设置了而不是被清除了。我们调用sleep、wait等此类可中断（throw InterruptedException）方法时，一旦方法抛出InterruptedException，当前调用该方法的线程的中断状态就会被jvm自动清除了，就是说我们调用该线程的isInterrupted 方法时是返回false。如果你想保持中断状态，可以再次调用interrupt方法设置中断状态。这样做的原因是，java的中断并不是真正的中断线程，而只设置标志位（中断位）来通知用户。如果你捕获到中断异常，说明当前线程已经被中断，不需要继续保持中断位。  
interrupted是静态方法，返回的是当前线程的中断状态。例如，如果当前线程被中断（没有抛出中断异常，否则中断状态就会被清除），你调用interrupted方法，第一次会返回true。然后，当前线程的中断状态被方法内部清除了。第二次调用时就会返回false。如果你刚开始一直调用isInterrupted，则会一直返回true，除非中间线程的中断状态被其他操作清除了。

注：清除中断标志其实就是把线程标记为已经中断，即interrupted，ed已。Interrupt是中断操作

线程的最常见写法：

class Worker implements Runnable {

private volatile boolean running = true;

public void run() {

while(running) {

try {

...

} catch (InterruptedException e) {

xxx;

return;

}

}

}

public void shutdown() {

running = false;

}

}

22) 为什么你应该在循环中检查等待条件?.

While(running) {

while(jobs.isEmpty()) {

try {

jobs.wait();

} catch (InterruptedException ex) {

Thread.currentThread().interrupt();

return;

}

}

job = jobs.removeFirst();//取出一个

}

}

因为当一个等待线程醒来时，不能认为它原来的等待状态仍然是有效的，在notify()方法调用之后和等待线程醒来之前这段时间它可能会改变，注意这种写法

class Worker implements Runnable {

private volatile boolean running = true;

public void run() {

while(running) {

Job job = null;

synchronized (jobs) {

while(jobs.isEmpty()) {

try {

jobs.wait();

} catch (InterruptedException ex) {

Thread.currentThread().interrupt();

return;

}

}

job = jobs.removeFirst();//取出一个

}

if (job != null) {

try {

job.run();

} catch (Exception ex) {

...

}

}

}

public void shutdown() {

running = false;

}

}

}

23) Java中的同步集合与并发集合有什么区别?

同步集合与并发集合都为多线程和并发提供了合适的线程安全的集合，不过并发集合的性能更高, 像ConcurrentHashMap，不仅提供线程安全还用锁分离和内部分区等

24) Java中堆和栈有什么不同？

每个线程都有自己的栈内存，用于存储本地变量，方法参数和调用栈，一个线程中存储的变量对其它线程是不可见的。而堆是所有线程共享的一片公用内存区域。对象都在堆里创建，为了提升效率线程会从堆中弄一个缓存到自己的栈，如果多个线程使用该变量就可能引发问题，这时volatile 变量就可以发挥作用了，它要求线程从主存中读取变量的值。

25） 什么是线程池？ 为什么要使用它？

如果任务来了才创建线程那么响应时间会变长，而且一个进程能创建的线程数有限。为了避免这些问题，在程序启动的时候就创建若干线程来响应处理，它们被称为线程池，里面的线程叫工作线程。Executor框架让你可以创建不同的线程池。比如单线程池，每次处理一个任务；数目固定的线程池或者是缓存线程池（一个适合很多生存期短的任务的程序的可扩展线程池）

public static ExecutorService newFixedThreadPool(int nThreads) {

return new ThreadPoolExecutor(nThreads, nThreads,0L, TimeUnit.MILLISECONDS,

new LinkedBlockingQueue<Runnable>());

}

public static ExecutorService newSingleThreadExecutor() {

return new FinalizableDelegatedExecutorService

(new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS,

new LinkedBlockingQueue<Runnable>()));

}

public static ExecutorService newCachedThreadPool() {

return new ThreadPoolExecutor(0, Integer.MAX\_VALUE, 60L, TimeUnit.SECONDS,new SynchronousQueue<Runnable>);

}

26） 如何写代码来解决生产者消费者问题？

用 BlockingQueue来实现生产者消费者模型

LinkedBlockingQueue: add方法在添加元素的时候，若超出了度列的长度会直接抛出异常;  对于put方法，若向队尾添加元素的时候发现队列已经满了会发生阻塞一直等待空间，以加入元素。 offer方法在添加元素时，如果发现队列已满无法添加的话，会直接返回false。

从队列中取出并移除头元素的方法有：poll，remove，take。

poll: 若队列为空，返回null。

remove:若队列为空，抛出NoSuchElementException异常。

take:若队列为空，发生阻塞，等待有元素。

import java.util.concurrent.LinkedBlockingQueue;

public class Storage

{

// 仓库最大存储量

private final int MAX\_SIZE = 100;

// 仓库存储的载体

private LinkedBlockingQueue<Object> list = new LinkedBlockingQueue<Object>(MAX\_SIZE);

// 生产num个产品

public void produce(int num)

{

if (list.size() == MAX\_SIZE)

{

System.out.println("【库存量】:" + MAX\_SIZE + "/t暂时不能执行生产任务!");

}

// 生产条件满足情况下，生产num个产品

for (int i = 1; i <= num; ++i)

{

try

{

// 放入产品，自动阻塞

list.put(new Object());

}

catch (InterruptedException e)

{

e.printStackTrace();

}

System.out.println("【现仓储量为】:" + list.size());

}

}

// 消费num个产品

public void consume(int num)

{

if (list.size() == 0)

{

System.out.println("【库存量】:0/t暂时不能执行生产任务!");

}

// 消费条件满足情况下，消费num个产品

for (int i = 1; i <= num; ++i)

{

try

{

// 消费产品，自动阻塞

list.take();

}

catch (InterruptedException e)

{

e.printStackTrace();

}

}

System.out.println("【现仓储量为】:" + list.size());

}

}

27） 如何避免死锁？

1)尽量使用tryLock(long timeout, TimeUnit unit)的方法(ReentrantLock、ReentrantReadWriteLock)，设置超时时间，超时可以退出防止死锁。   
2)尽量使用java.util.concurrent(jdk 1.5以上)包的并发类代替手写控制并发，比较常用的是ConcurrentHashMap、ConcurrentLinkedQueue、AtomicBoolean等等，实际应用中java.util.concurrent.atomic十分有用，简单方便且效率比使用Lock更高   
3)尽量降低锁的使用粒度，尽量不要几个功能用同一把锁   
4)尽量减少同步的代码块

5) 让所有的线程按照同样的顺序获得一组锁。这种方法消除了 X 和 Y 的拥有者分别等待对方的资源的问题。

29） 怎么检测一个线程是否拥有锁？

java.lang.Thread中有一个方法叫holdsLock()，它返回true如果当且仅当当前线程拥有某个具体对象的锁

30) 你如何在Java中获取线程堆栈？

用jstack <thread id>

jps是jdk提供的一个查看当前[Java](http://lib.csdn.net/base/17)进程的小工具，可以用来查看进程线程

31) JVM中哪个参数是用来控制线程的栈堆栈小的

-Xms 初始堆大小

-Xmx 最大堆大小

32） Java中synchronized 和 ReentrantLock 有什么不同？

synchronized实现原理：JVM基于进入和退出Monitor对象来实现方法同步和代码块同步，其中代码块同步用monitorenter和monitorexit指令实现

ReentrantLock基于队列同步器(AbstractQueuedSynchronizer)

使用一个int变量表示同步状态，通过FIFO队列完成线程资源排队，同步器通过实现AbstractQueuedSynchronizer的抽象方法实现

AbstractQueuedSynchronizer提供了三个方法来管理同步状态：

getState():获取当前同步状态

setState(int newState):设置同步状态

compareAndSetState(int expect, int update):使用CAS设置当前状态，保证设置的原子性

同步器可实现的抽象方法：tryAcquire，tryRelease，tryAcquireShared，tryReleaseShared，isHelExclusively

同步器维护一个FIFO的链表，添加节点，删除节点都需要CAS保证原子性，链表的首尾地址存在同步器中

1、ReentrantLock 拥有Synchronized相同的并发性和内存语义，此外还多了 锁投票，定时锁等候和中断锁等候

     线程A和B都要获取对象O的锁定，假设A获取了对象O锁，B将等待A释放对O的锁定，

     如果使用 synchronized ，如果A不释放，B将一直等下去，不能被中断

     如果 使用ReentrantLock，如果A不释放，可以使B在等待了足够长的时间以后，中断等待，而干别的事情

    ReentrantLock获取锁定与三种方式：  
    a)  lock(), 如果获取了锁立即返回，如果别的线程持有锁，当前线程则一直处于休眠状态，直到获取锁

    b) tryLock(), 如果获取了锁立即返回true，如果别的线程正持有锁，立即返回false；

    c)**tryLock**(long timeout,[TimeUnit](http://houlinyan.iteye.com/java/util/concurrent/TimeUnit.html) unit)，   如果获取了锁定立即返回true，如果别的线程正持有锁，会等待参数给定的时间，在等待的过程中，如果获取了锁定，就返回true，如果等待超时，返回false；

    d) lockInterruptibly:如果获取了锁定立即返回，如果没有获取锁定，当前线程处于休眠状态，直到或者锁定，或者当前线程被别的线程中断

2、synchronized是在JVM层面上实现的，不但可以通过一些监控工具监控synchronized的锁定，而且在代码执行时出现异常，JVM会自动释放锁定，但是使用Lock则不行，lock是通过代码实现的，要保证锁定一定会被释放，就必须将unLock()放到finally{}中

3、在资源竞争不是很激烈的情况下，Synchronized的性能要优于ReetrantLock，但是在资源竞争很激烈的情况下，Synchronized的性能会下降几十倍，但是ReetrantLock的性能能维持常态

33） 有三个线程T1，T2，T3，怎么确保它们按顺序执行？

用线程类的join()方法在一个线程中启动另一个线程，另外一个线程完成该线程继续执行。为了确保三个线程的顺序你应该先启动最后一个(T3调用T2，T2调用T1)，这样T1就会先完成而T3最后完成。

1.join方法只有在继承了Thread类的线程中才有。

2.线程必须要start() 后再join才能起作用。

将另外一个线程join到当前线程，则需要等到join进来的线程执行完才会继续执行当前线程。

package Thread.join;

class Sleeper extends Thread {

private int duration;

public Sleeper(String name, int sleepTime) {

super(name);

duration = sleepTime;

start();

}

public void run() {

try {

sleep(duration);

} catch (InterruptedException e) {

System.out.println(getName() + " was interrupted."

+ "isInterrupted():" + isInterrupted());

return;

}

System.out.println(getName() + " has awakened");

}

}

class Joiner extends Thread {

private Sleeper sleeper;

public Joiner(String name, Sleeper sleeper) {

super(name);

this.sleeper = sleeper;

start();

}

public void run() {

try {

sleeper.join();

} catch (InterruptedException e) {

System.out.println("Interrupted");

}

System.out.println(getName() + " join completed");

}

}

public class Joining {

public static void main(String[] args) {

Sleeper sleepy01 = new Sleeper("Sleepy01", 1500), sleepy02 = new Sleeper(

"Sleepy02", 1500);

Joiner joiner01 = new Joiner("Joiner01", sleepy01), joiner02 = new Joiner("Joiner02",

sleepy02);

joiner01.interrupt();

}

}

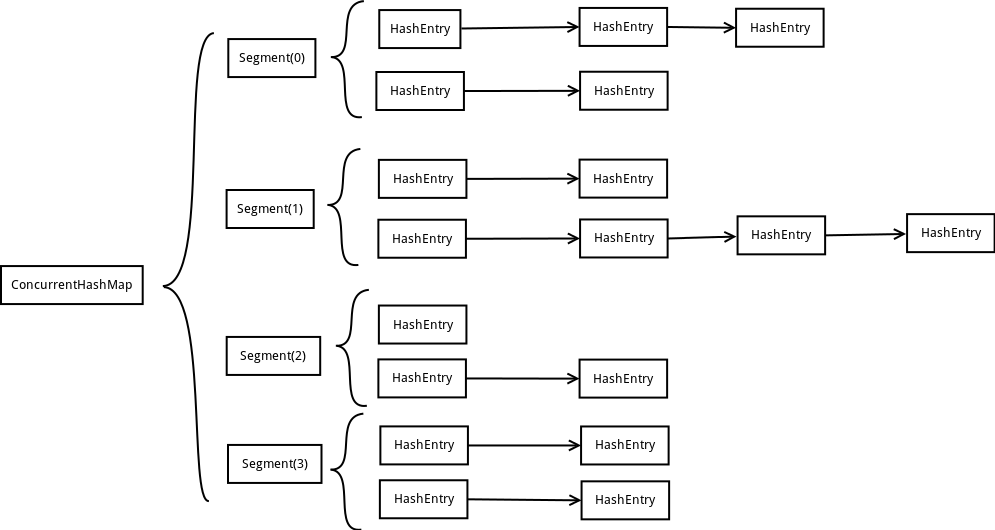
注：持有上一个线程的引用，就可以直接start或join了

34) Thread类中的yield方法有什么作用？

Yield是一个静态方法，当前线程放弃CPU，执行yield()的线程有可能在进入到暂停状态后马上又被执行。

35） Java中ConcurrentHashMap的并发度是什么？

ConcurrentHashMap把实际map划分成若干部分来实现它的性能和线程安全。这种划分是使用并发度获得的，它是ConcurrentHashMap类构造函数的一个可选参数，默认值为16

　ConcurrentHashMap为了提高本身的并发能力，在内部采用了一个叫做Segment的结构，一个Segment其实就是一个类Hash Table的结构，Segment内部维护了一个链表数组，我们用下面这一幅图来看下ConcurrentHashMap的内部结构：  
[](http://www.yupoo.com/photos/goldendoc/81556254/)

36） Java中Semaphore是什么？

Semaphore是一种同步类，它是一个计数信号。Semaphore可以控制某个资源可被同时访问的个数，通过 acquire() 获取一个许可，如果没有就等待，而 release() 释放一个许可。信号量常常用于多线程的代码中，比如数据库连接池。

37）如果你提交任务时，线程池队列已满。会时发会生什么？

ThreadPoolExecutor’s submit()方法将会抛出一个RejectedExecutionException异常

38) Java线程池中submit() 和 execute()方法有什么区别？

两个方法都可以向线程池提交任务，execute()方法的返回类型是void，它定义在Executor接口中, 而submit()方法可以返回持有计算结果的Future对象，它定义在ExecutorService接口中，它扩展了Executor接口，其它线程池类像ThreadPoolExecutor和ScheduledThreadPoolExecutor都有这些方法。

39) 什么是阻塞式方法？

阻塞式方法是指程序会一直等待该方法完成期间不做其他事情，ServerSocket的accept()方法就是一直等待客户端连接。这里的阻塞是指调用结果返回之前，当前线程会被挂起，直到得到结果之后才会返回。

“阻塞”与"非阻塞"与"同步"与“异步"不能简单的从字面理解，提供一个从分布式系统角度的回答。  
**1.同步与异步**  
同步和异步关注的是**消息通信机制** (synchronous communication/ asynchronous communication)  
所谓同步，就是在发出一个\*调用\*时，在没有得到结果之前，该\*调用\*就不返回。但是一旦调用返回，就得到返回值了。  
换句话说，就是由\*调用者\*主动等待这个\*调用\*的结果。  
  
而异步则是相反，**\*调用\*在发出之后，这个调用就直接返回了，所以没有返回结果**。换句话说，当一个异步过程调用发出后，调用者不会立刻得到结果。而是在\*调用\*发出后，\*被调用者\*通过状态、通知来通知调用者，或通过回调函数处理这个调用。  
  
典型的异步编程模型比如Node.js  
  
举个通俗的例子：  
你打电话问书店老板有没有《分布式系统》这本书，如果是同步通信机制，书店老板会说，你稍等，”我查一下"，然后开始查啊查，等查好了（可能是5秒，也可能是一天）告诉你结果（返回结果）。  
而异步通信机制，书店老板直接告诉你我查一下啊，查好了打电话给你，然后直接挂电话了（不返回结果）。然后查好了，他会主动打电话给你。在这里老板通过“回电”这种方式来回调。  
  
2. 阻塞与非阻塞  
阻塞和非阻塞关注的是**程序在等待调用结果（消息，返回值）时的状态.**  
  
阻塞调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回。  
非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程。  
  
还是上面的例子，  
你打电话问书店老板有没有《分布式系统》这本书，你如果是阻塞式调用，你会一直把自己“挂起”，直到得到这本书有没有的结果，如果是非阻塞式调用，你不管老板有没有告诉你，你自己先一边去玩了， 当然你也要偶尔过几分钟check一下老板有没有返回结果。  
在这里阻塞与非阻塞与是否同步异步无关。跟老板通过什么方式回答你结果无关。

(线程的挂起操作实质上就是使线程进入“非可执行”状态下，在这个状态下CPU不会分给线程时间片，进入这个状态可以用来暂停一个线程的运行。在线程挂起后，可以通过重新唤醒线程来使之恢复运行

     A，通过调用sleep()方法使线程进入休眠状态，线程在指定时间内不会运行。  
     B，通过调用join()方法使线程挂起，如果某个线程在另一个线程t上调用t.join()，这个线程将被挂起，直到线程t执行完毕为止。  
     C，通过调用wait()方法使线程挂起，直到线程得到了notify()和notifyAll()消息，线程才会进入“可执行”状态。)

43) 如何在Java中创建Immutable(不可变)对象？

创建不可变类要实现下面几个步骤：通过构造方法初始化所有成员、对变量不要提供setter方法、将所有的成员声明为私有的，这样就不允许直接访问这些成员、在getter方法中，不要直接返回对象本身，而是克隆对象，并返回对象的拷贝

44） Java中的ReadWriteLock是什么？

ReentrantReadWriteLock，支持多线程读，单线程写，一般情况下比排他性锁性能好，因为大多数场景是读

原理是自定义同步器中同步状态整型变量的高16位负责读，低16位负责写，通过位运算获取读写状态

读：int c = c + (1<<16);

1向右移16位于当前获取锁线程的数值相加，获取读锁线程数

比如，一共有3个线程有读锁

00000000000000110000000000000000

+

00000000000000010000000000000000

00000000000001000000000000000000

即高16位是4

写的话就必须是0，没有线程的状态，然后直接加1就行，这就是为什么低位负责写

00000000000000000000000000000001

注，之所以高读低写是因为如果反过来都需要做位运算，而现在只是读需要位运算

45) 多线程中的忙循环是什么?

忙循环就是程序员用循环让一个线程等待，不像传统方法wait(), sleep() 或 yield() 它们都放弃了CPU控制，而忙循环不会放弃CPU，它就是在运行一个空循环。这么做的目的是为了保留CPU缓存，在多核系统中，一个等待线程醒来的时候可能会在另一个内核运行，这样会重建缓存。为了避免重建缓存和减少等待重建的时间就可以使用它了。（CAS可能和这个有关）

46）volatile 变量和 atomic 变量有什么不同？

Volatile写操作会发生在后续的读操作之前, 但它并不能保证原子性。例如用volatile修饰count变量那么 count++ 操作就不是原子性的。而AtomicInteger类提供的atomic方法可以让这种操作具有原子性如getAndIncrement()方法会原子性的进行增量操作把当前值加一，其它数据类型和引用变量也可以进行相似操作。

47) 如果同步块内的线程抛出异常会发生什么？

无论你的同步块是正常还是异常退出的，里面的线程都会释放锁，所以对比锁接口我更喜欢同步块，因为它不用我花费精力去释放锁，锁在finally里释放，因为不会自动释放

48） 单例模式的双检锁是什么？

是一个用来创建线程安全的单例的老方法，当单例实例第一次被创建时它试图用单个锁进行性能优化

public class Singleton {

private static Singleton instance;

private Singleton (){

}

public static Singleton getInstance(){ //对获取实例的方法进行同步

if (instance == null){

synchronized(Singleton.class){

if (instance == null)

instance = new Singleton();

}

}

return instance;

}

}

49） 如何在Java中创建线程安全的Singleton？

利用JVM的类加载和静态变量初始化特征来创建Singleton实例，或者是利用枚举类型来创建Singleton

50) 写出3条你遵循的多线程最佳实践

给你的线程起个有意义的名字，这样可以方便找bug或追踪

避免锁定和缩小同步的范围，因此相对于同步方法我更喜欢同步块，它给我拥有对锁的绝对控制权

多用同步类少用wait 和 notify  
首先，CountDownLatch, Semaphore, CyclicBarrier 和 Exchanger 这些同步类简化了编码操作，而用wait和notify很难实现对复杂控制流的控制。其次，这些类是由最好的企业编写和维护在后续的JDK中它们还会不断优化和完善，使用这些更高等级的同步工具你的程序可以不费吹灰之力获得优化。

多用并发集合少用同步集合，如ConcurrentHashMap而不是HashTable

52) Java中的fork join框架是什么？

fork join框架是JDK7中出现的一款高效的工具，它是专门为了那些可以递归划分成许多子模块设计的，目的是将所有可用的处理能力用来提升程序的性能。fork join框架一个巨大的优势是它使用了工作窃取算法，可以完成更多任务的工作线程可以从其它线程中窃取任务来执行。主要是把大任务分成小任务（Fork），Join把结果汇总

53） Java多线程中调用wait() 和 sleep()方法有什么不同？

wait()方法用于线程间通信，如果等待条件为真且其它线程被唤醒时它会释放锁，而sleep()方法仅仅释放CPU资源或者让当前线程停止执行一段时间，但不会释放锁。

54)多线程单例模式

主要有两种比较好，一个是static内部类、块，另一个是枚举

1. **package** org.mlinge.s06;
3. **public** **class** MySingleton {
5. //内部类
6. **private** **static** **class** MySingletonHandler{
7. **private** **static** MySingleton instance = **new** MySingleton();
8. }
10. **private** MySingleton(){}
12. **public** **static** MySingleton getInstance() {
13. **return** MySingletonHandler.instance;
14. }
15. }

或者用代码块，都需要有个getInstance方法

静态代码块中的代码在使用类的时候就已经执行了，所以可以应用静态代码块的这个特性的实现单例设计模式。

1. **package** org.mlinge.s08;
3. **public** **class** MySingleton{
5. **private** **static** MySingleton instance = **null**;
7. **private** MySingleton(){}
9. **static**{
10. instance = **new** MySingleton();
11. }
13. **public** **static** MySingleton getInstance() {
14. **return** instance;
15. }
16. }

枚举

1. **package** org.mlinge.s10;
3. **public** **class** ClassFactory{
5. **private** **enum** MyEnumSingleton{
6. singletonFactory;
8. **private** MySingleton instance;
10. **private** MyEnumSingleton(){//枚举类的构造方法在类加载是被实例化
11. instance = **new** MySingleton();
12. }
14. **public** MySingleton getInstance(){
15. **return** instance;
16. }
17. }
19. **public** **static** MySingleton getInstance(){
20. **return** MyEnumSingleton.singletonFactory.getInstance();
21. }
22. }
24. **class** MySingleton{//需要获实现单例的类，比如数据库连接Connection
25. **public** MySingleton(){}
26. }

都是在类一加载就被实例化，不存在多线程创建多个实例的可能性

实际枚举最好，以前的所有的单例模式都有一个比较大的问题，就是一旦实现了Serializable接口之后，就不再是单例得了，因为，每次调用 readObject()方法返回的都是一个新创建出来的对象，有一种解决办法就是使用readResolve()方法来避免此事发生。但是，为了保证枚举类型像Java规范中所说的那样，每一个枚举类型极其定义的枚举变量在JVM中都是唯一的，在枚举类型的序列化和反序列化上，Java做了特殊的规定在序列化的时候Java仅仅是将枚举对象的name属性输出到结果中，反序列化的时候则是通过java.lang.Enum的valueOf方法来根据名字查找枚举对象。同时，编译器是不允许任何对这种序列化机制的定制的，因此禁用了writeObject、readObject、readObjectNoData、writeReplace和readResolve等方法

枚举实例创建是thread-safe(线程安全的)

优势：**1. 枚举写法简单2. 枚举自己处理序列化3.枚举线程安全**

1. **package** org.mlinge.s10;
3. **public** **class** ClassFactory{
5. **private** **enum** MyEnumSingleton{
6. singletonFactory;
8. **private** MySingleton instance;
10. **private** MyEnumSingleton(){//枚举类的构造方法在类加载是被实例化
11. instance = **new** MySingleton();
12. }
14. **public** MySingleton getInstance(){
15. **return** instance;
16. }
17. }
19. **public** **static** MySingleton getInstance(){
20. **return** MyEnumSingleton.singletonFactory.getInstance();
21. }
22. }
24. **class** MySingleton{//需要获实现单例的类，比如数据库连接Connection
25. **public** MySingleton(){}
26. }

测试类

1. **public** **class** MyThread **extends** Thread{
3. @Override
4. **public** **void** run() {
5. System.out.println(ClassFactory.getInstance().hashCode());
6. }
8. **public** **static** **void** main(String[] args) {
10. MyThread[] mts = **new** MyThread[10];
11. **for**(**int** i = 0 ; i < mts.length ; i++){
12. mts[i] = **new** MyThread();
13. }
15. **for** (**int** j = 0; j < mts.length; j++) {
16. mts[j].start();
17. }
18. }
19. }

Base64  
