创建进程方法

所有线程都是Thread类或子类。

方法一

（1）继承Thread的子类

（2）重写public void run(){ }

（3）创建子类实例

（4）start()

public class FirstThread extends Thread {

private int i = 0;//可以自定义变量

public void run() {

for(;i < 100;++i)

System.out.println(i+"--"+getName());

}

/\*getName()获取当前进程的名字，调用任何方法都需要实例或类，

这里是this，省略。\*/

public static void main(String[] args) {

for(int j = 0;j < 100;++j) {

System.out.println(j+"--"+Thread.currentThread().getName());

/\*currentThread()是Thread类的静态方法，返回调用者进程的名字\*/

if(j == 20)

new FirstThread().start();

}

}

}

进程默认名字是Thread-0，Thread-1...可以setName(String name)设置名字

方法二

(1)实现一个继承Runnable接口的类,重写 public void run()

(2)Thread t = new Thread(new出类实例)

(3)t.start();

public class SecondThread implements Runnable {

private int i = 0;

public void run() {

for(;i < 100;++i)

System.out.println(i+"--"+Thread.currentThread().getName());

}

public static void main(String[] args) {

SecondThread st = new SecondThread();

Thread t = new Thread(st,"子进程1");

Thread tt = new Thread(st,"子进程2");

/\*t和tt共用同一个i，最后共打印100项，而不是200项\*/

t.start();

tt.start();

}

}

比较使用Thread类和使用Thread子类创建进程：

使用Thread类还能继承其他类和实现其他接口，使用Thread子类不能再继承其他类。

使用Thread子类，在实现runnable接口时，自定义变量，能让相同进程共享同一变量，达到一个变量控制多个进程或多个进程处理同一资源的目的。

一种使用Lamable表达式快速创建进程的方式：

1，某一方法或对象需要动态处理方法，而方法寄存在对象中，所以参数是实例引用变量

2，实例引用变量可以是继承了【抽象类或接口，把动态处理方法抽象，不必实现】的类

【该类实现了抽象方法】创建的实例

3，特别情况，当是用接口达到目的时，接口可以选择先实现个类，也可以使用Lambda表达式，

如下。

前者：方法可以多次使用。后者：只能使用一次，但是灵活，且没产生新类

4，针对3，创建进程时，类可以达到多个进程共享一个变量的目的，但是lambda表达式不可以

public class ThirdThread {

public static void main(String[] args) {

Thread t = new Thread(new Runnable(){

public void run() {

for(int i = 0; i < 100; ++i)

System.out.println(i+"--"+Thread.currentThread().getName());

}

});

t.start();

}

}

线程池

系统启动一个线程，需要和操作系统交互，开销还是蛮大的，使用线程池可以很好的提高性能，尤其是当程序需要创建大量生存期很短的线程的时候。

线程在系统启动时即创建大量空闲的线程，程序将一个Runnable对象或Callable对象传给线程池，线程池就会启动一个线程执行它们的run或call方法，当run或call方法执行完毕，线程并不会死亡，而是恢复成空闲状态，等待下一个对象传入。

当系统中运行太多线程，系统的性能会急速下降，甚至导致JVM崩溃，使用线程池能够约束系统的线程数。

利用Executors工厂类的几个静态方法来产生线程池

public class ThreadPoolTest **{**

public static void main**(**String**[]** args**)** **throws** Exception **{**

//创建一个容量是6的线程池

ExecutorService pool **=** Executors**.**newFixedThreadPool**(**6**);**

//使用Lanbda表达式创建Runnable对象

Runnable target **=** **()** **->** **{**

**for(**int i **=** 0**;**i **<** 100**;++**i**)** **{**

System**.**out**.**println**(**Thread**.**currentThread**().**getName**()+**i**);**

**}**

**};**

//向线程池提交两个线程

pool**.**submit**(**target**);**

pool**.**submit**(**target**);**

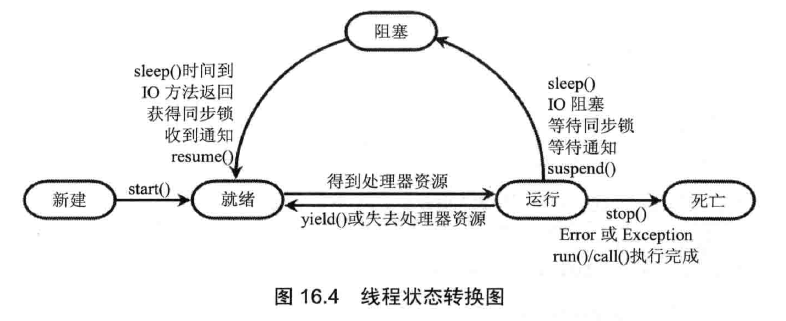
//关闭线程池

pool**.**shutdown**();**//线程不再接收新任务，当前所有任务执行完毕，池中所有线程死亡

**}**

**}**

线程生命周期



Thread t = new Thread(new Runnable(){public void run(){}});申请空白PCB，填写PCB，申请资源。

t.start()就绪，调入内存，可被CPU调度的唯一状态。

t.start()后，进程有资格占用CPU，但不是立即占用CPU，取决于JVM线程调度器的调度。若想子线程在立即被调度t.start()后立即被执行，可用Thread.sleep(1)，让主线程睡眠1毫秒。

线程只能从就绪到执行。阻塞的线程唤醒后只能返回就绪，而不是执行。

isAlive（），线程处于就绪，运行，阻塞返回true，新建和死亡返回false。

对一个线程两次调用start异常：IllegalThreadStateException.

控制进程

线程等待

t.join();主进程在执行流的某处调用t.join()；进程t执行完毕，主进程才继续执行。

通常由使用线程的程序调用，将大问题分解成很多小问题，每个小问题用一个线程解决，所有小问题解决完毕后，才进一步处理。

public class JoinThread extends Thread{

public JoinThread(String name) {

super(name);

}

public void run() {

for(int i = 0; i < 100; ++i) {

System.out.println(i+"--"+getName());

}

}

public static void main(String[] args) {

JoinThread jt = new JoinThread("×Ó½ø³Ì");

jt.start();

for(int j = 0;j < 100;++j) {

System.out.println(j+"--"+Thread.currentThread().getName());

if(j == 50)

try {

jt.join();

}catch(InterruptedException e) {

;

}

}

}

}

join（long millis）等待指定时间，时间到子进程执行不完也不再等待。

线程睡眠

Thread的静态方法，Thread.sleep(long millis)，调用进程睡眠指定时间。

Thread.sleep(long millis)和Thread.sleep(long millis, int nanos)静态方法强制当前正在执行的线程休眠（暂停执行），以“减慢线程”。

1、线程睡眠是帮助所有线程获得运行机会的最好方法。

2、线程睡眠到期自动苏醒，并返回到就绪状态。sleep()中指定的时间是线程不会运行的最短时间。因此，sleep()方法不能保证该线程睡眠到期后就开始执行。

3、sleep()是静态方法，只能控制当前正在运行的线程。

4，睡眠的位置：为了让其他线程有机会执行，可以将Thread.sleep()的调用放线程run()之内。这样才能保证该线程执行过程中会睡眠。

睡眠的实现：调用静态方法。

        try {  
            Thread.sleep(1000);  
        } catch (InterruptedException e) {  
            e.printStackTrace();   
        }

每隔1秒打印一次时间，打印10秒后结束打印

public class ThreadDemo {

public static void main(String[] args) {

MyThread mt = new MyThread();

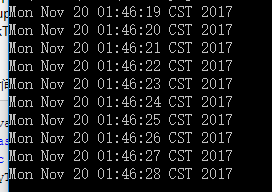
mt.start();

try {

Thread.sleep(10000);

}catch(InterruptedException e) {

;



}

mt.interrupt();

}

}

class MyThread extends Thread {

public void run() {

while(true) {

System.out.println(new Date());

try {

Thread.sleep(1000);

}catch(InterruptedException e) {

return;

}

}

}

}

线程让步

Threda.yield()也是静态方法，调用后，进程从执行进入就绪状态，相当于刷新一下CPU，让优先级大于或等于调用者的线程占用CPU，完全有可能调用者撤下CPU后第一个又占用CPU。

public class ThreadDemo {

public static void main(String[] args) {

MyThread mt = new MyThread();

mt.start();

for(int j = 0;j<10;++j) {

System.out.println("i am main Thread " + j);

try {

Thread.sleep(500);

}catch(InterruptedException e) {

return;

}

}

}

}

class MyThread extends Thread {

public void run() {

for(int i = 0;i < 10;++i) {

System.out.println("i am MyThread " + i);

for(int k = 0;k < 5000000;++k) {

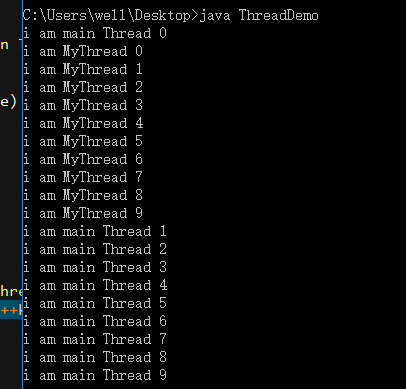
Thread.yield();

}

}

}

}



线程的优先级

线程的优先级越高，获得CPU的时间越长，注意：不同优先级的线程并发，低优先级的线程仅仅是获取CPU的时间较短而已。

优先级1—10

Thread类的三个静态常量：

默认优先级是5  Thread.NORM\_PRIORITY

最低优先级是1 Thread.MIN\_PRIORITY

最高优先级是10 Thread.MAX\_PRIORITY

int getPriority();  获取优先级

void setPriority(int  ); 设置优先级

设置优先级步骤：

（1） 创建线程

（2）设置优先级

（3）启动进程

结束一个线程的最佳方法

重写一个线程的public void run( )时，run的循环结构加设 flag ，flag == true 时，run方法的循环结构才能执行。

再为线程设置一个public void shutDown( ) { flag = false };当想结束进程时，直接调用执行shutDown( )来结束进程。

线程通信

synchronized

多进程互斥访问临界资源。

同步代码块：synchronized（临界资源）{ 临界区 }

同步方法： synchronized void fun（方法参数）{ 临界区 }，同步方法的临界资源默认是包含方法的类，即this，能够互斥访问对象。

同步代码块实现取钱

public void draw(double drawAmount) {

synchronized(this) {

if(balance >= drawAmount) {

balance -= drawAmount;

System.out.println("成功取钱"+drawAmount+"元！");

}

else {

System.out.println("余额不足！");

}

}

}

同步方法实现取钱

public synchronized void draw(double drawAmount) {

if(balance >= drawAmount) {

balance -= drawAmount;

System.out.println("成功取钱"+drawAmount+"元！");

}

else {

System.out.println("余额不足！");

}

}

多线程取钱

public class DrawThread extends Thread {

private Account account;

private double drawAmount;

public DrawThread() {}

public DrawThread(Account account,double drawAmount) {

this.account = account;

this.drawAmount = drawAmount;

}

public void run() {

account.draw(drawAmount);

}

}

对于synchronized修饰的实例方法，无需显示指定同步监视器，同步方法的同步监视器是this，也就是调用该方法的对象。

不可变类总是线程安全的，因为它的对象状态不改变。可变类的成员变量是可以修改的，需要额外的方法保证其线程安全。

通过使用同步方法可以非常方便的实现【线程安全的类】。

什么是线程安全的类？

1. 该类的对象可以被多个线程安全地访问。
2. 每个线程调用该对象的任意方法只后都能得到正确的结果
3. 每个线程调用该对象的任意方法之后，该对象状态依然保持合理的状态。

synchronnized关键字只能修饰方法，代码块。

可变类的线程安全是以降低程序的运行效率实现的，为了减少线程安全带来的负面影响，程序可以采取如下策略：

1. 只对会改变临界资源的方法进行同步。
2. 如果可变类有两种运行环境：单线程和多线程，则应该为该可变类提供两种版本，即线程不安全版本和线程安全版本。在单线程环境中使用线程不安全版本以保证性能，在多线程环境中使用线程安全版本。

对于（2），JDK的StringBuilder,StringBuffer就是为了照顾单线程和多线程环境提供的两个版本。

同步代码块和同步方法何时释放对同步监视器的锁定？

释放对同步监视器的锁定与notify()意思不同，释放对同步监视器的锁定，相当于lock.unlock();释放后，意味着这个线程的剩余代码与锁，同步概念无关，只是一个普通程序。

synchronized和Lock不同之处就是：synchronized释放锁是死的，Lock释放锁却是灵活的，程序员可控的。

synchronized会自动释放锁，而Lock一定要求程序员手工释放，并且必须在finally从句中释放。

1. 方法块或方法执行结束。
2. 方法块或方法出现未处理的Error或Exception，导致代码块代码块或方法结束。
3. 程序执行了同步监视器的wait( )方法，当前线程暂停，并释放同步监视器。
4. Thread.sleep( ),Thread.yield( )不会释放同步监视器。

Object类放入三种方法：wait( ) notify( ) notifyAll( )

这三个方法必须由同步监视器调用。

wait（）：线程调用此方法，会释放对同步监视器的锁定，并导致当前线程等待。

notify（）：只有当前线程释放了同步监视器（wait（）方法），才能唤醒其他等待的线程。

当一个线程进入一个对象的一个synchronized方法后，其它线程是否可进入此对象的其它方法?

1. 其他方法前是否加了synchronized关键字，如果没加，则能。  
   2.如果这个方法内部调用了wait，则可以进入其他synchronized方法。  
   3.如果其他个方法都加了synchronized关键字，并且内部没有调用wait，则不能。  
   4.如果其他方法是static，它用的同步锁是当前类的字节码，与非静态的方法不能同步，因为非静态的方法用的是this。

Lock

使用Lock对象保证同步，Condition可以让那些已经得到Lock对象却无法继续执行的线程释放Lock对象，Condition也可以唤醒其他处于等待的线程。

Condition 实例实质上被绑定到一个锁上。要为特定 [Lock](http://www.cnblogs.com/) 实例获得 Condition 实例，使用其[newCondition()](http://www.cnblogs.com/) 方法。

Condition类提供了如下三个方法：

await():造成当前线程在接到信号或被中断之前一直处于等待状态。 该方法流程：

新建Condition Node包装线程，加入Condition队列。

释放当前线程占有的锁

阻塞当前线程

signal():唤醒当前lock对象的一个等待线程。

signalAll():唤醒当前lock对象的所有等待线程。只有当前线程放弃对lock的锁定，被唤醒的线程才可以执行

import java.util.concurrent.locks.\*;

public class TestLock {

private final Lock lock = new ReentrantLock();

private final Condition cond = lock.newCondition();

private boolean flag = false;

public void PrintWord() {

lock.lock();

try {

for(char ch = 'A';ch < 'Z';++ch) {

if(flag == false)

cond.await();

System.out.print(ch);

flag = false;

cond.signal();

}

}catch(InterruptedException e1) {

e1.getMessage();

}finally {

lock.unlock();

}

}

public void PrintDigit() {

lock.lock();

try {

for(int i = 1;i <= 52;++i) {

if(flag == true)

cond.await();

System.out.print(i);

if(i % 2 == 0)

flag = true;

cond.signal();

}

}catch(InterruptedException e2) {

e2.getMessage();

}finally {

lock.unlock();

}

}

public static void main(String[] args) {

TestLock tl = new TestLock();

new Thread(new Runnable() {

public void run() {

tl.PrintDigit();

}

}).start();

new Thread(new Runnable() {

public void run() {

tl.PrintWord();

}

}).start();

}

}

线程同步

按次序执行进程

一个进程打印A，一个进程打印B，要求打印10次AB。

public class ThreadAB {

public static void main(String[] args) {

final Print business = new Print();

new Thread(new Runnable() {

public void run() {

for(int i=0;i<10;i++) {

business.print\_A();

}

}

}).start();

new Thread(new Runnable() {

public void run() {

for(int i=0;i<10;i++) {

business.print\_B();

}

}

}).start();

}

}

class Print {

private boolean flag = true;

public synchronized void print\_A () {

if(flag == false) {

try {

this.wait();

} catch(Exception e) {

e.printStackTrace();

}

}

System.out.print("A");

flag = false;

this.notify();

}

public synchronized void print\_B () {

if(flag) {

try {

this.wait();

} catch(Exception e) {

e.printStackTrace();

}

}

System.out.println("B");

flag = true;

this.notify();

}

}

一个线程打印A，一个打印B，一个打印C，实现循环打印ABC。

public class ThreadABC2 {

public static void main(String[] args) {

Print p = new Print();

new Thread(new Runnable() {

public void run() {

for(int i = 0;i<10;++i) {

p.print\_A();

}

}

}).start();

new Thread(new Runnable() {

public void run() {

while(true)

p.print\_B();

}

}).start();

new Thread(new Runnable() {

public void run() {

while(true)

p.print\_C();

}

}).start();

}

}

class Print {

private boolean should\_a = true;

private boolean should\_b = false;

private boolean should\_c = false;

public synchronized void print\_A () {

while(should\_b || should\_c) {

try {

this.wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

System.out.print("A");

should\_a = false;

should\_b = true;

should\_c = false;

this.notifyAll();

}

public synchronized void print\_B () {

while(should\_a || should\_c) {

try {

this.wait();

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

System.out.print("B");

should\_a = false;

should\_b = false;

should\_c = true;

this.notifyAll();

}

public synchronized void print\_C () {

while(should\_a || should\_b) {

try {

this.wait();

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

System.out.print("C");

should\_a = true;

should\_b = false;

should\_c = false;

this.notifyAll();

}

}

生产者消费者

import java.util.\*;

public class Test {

public static void main(String[] args) {

Storage storage = new Storage();

new Thread(new Runnable(){

public void run() {

while(true)

storage.produce(2);

}

}).start();

new Thread(new Runnable(){

public void run() {

while(true)

storage.consume(3);

}

}).start();

}

}

class Storage {

private final int MAX\_SIZE = 10;

private LinkedList<Object> list = new LinkedList<>();

void produce(int num) {

synchronized(list) {

if(list.size()+num<=MAX\_SIZE) {

for(int i = 0;i < num;++i)

list.add(new Object());

System.out.println("生产了"+num+"个产品，此时仓库存储量是"+list.size());

try{list.notifyAll();}catch(Exception e){e.printStackTrace();}

}

else {

System.out.println("仓库已满，无法生产...");

try{list.wait();}catch(Exception e){e.printStackTrace();}

}

}

}

void consume(int num) {

synchronized(list) {

if(list.size()-num>=0) {

for(int i = 0;i < num;++i)

list.remove();

System.out.println("消费了"+num+"个产品，此时仓库存储量是"+list.size());

try{list.notifyAll();}catch(Exception e){e.printStackTrace();}

}

else {

System.out.println("存货不足，请稍等...");

try{list.wait();}catch(Exception e){e.printStackTrace();}

}

}

}

}

程序运行结果：

12A34B56C78D910E1112F1314G1516H1718I1920J2122K2324L2526M2728N2930O3132P3334Q3536R3738S3940T4142U4344V4546W4748X4950Y5152

ThreadLocal类

定义成员变量：private ThreadLocal<> var = new ThreadLocal<>( );

变量的类型取决于传入的泛型类型。

get()读取副本的值。set()设置当前副本的值

假设有3个线程访问此变量，会拷贝3个完全相同副本，3个线程各自拥有，去操作。解决了多线程的并发访问，因此不需要对变量同步了。但是ThreadLocal和同步意义不同！

JAVA内存模型

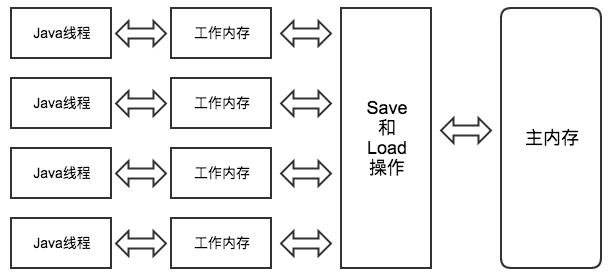
Java虚拟机规范中试图定义一种Java内存模型（Java Memory Model，JMM）来屏蔽掉各种硬件和操作系统的内存访问差异，以实现让Java程序在各种平台下都能达到一致的内存访问效果。

Java内存模型的主要目标是定义程序中各个变量的访问规则，即在虚拟机中将变量存储到内存和从内存中取出变量这样的底层细节。  
这里说的变量包括实例字段、静态字段和构成数组对象的元素，不包括局部变量与方法参数，因为后者是线程私有的，不会共享，也就不存在竞争的问题。

Java内存模型规定了所有的变量都存储在主内存（Main Memory）中，此外每条线程还有自己的工作内存（Working Memory）。

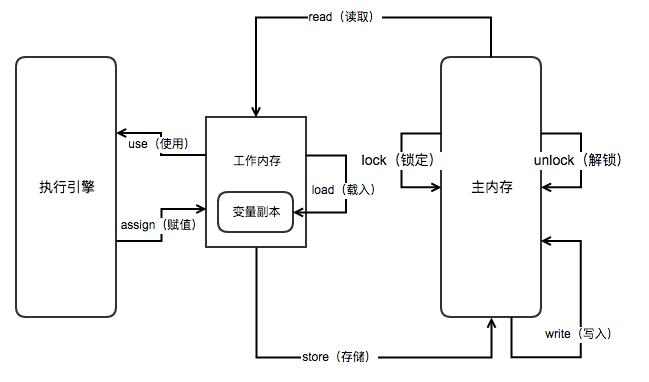
线程的工作内存中保存了被该线程使用到的变量的主内存副本拷贝，线程对变量的所有操作（读取、赋值等）都必须在工作内存中进行，不能直接读写主内存中的变量。

并且，不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量值得传递均需要通过主内存来完成，线程、主内存、工作内存关系如下图



也可以把这里的主内存与工作内存概念与JVM运行时数据区进行对应，主内存主要对应Java堆中的对象实例数据部分，工作内存对应于虚拟机栈中的部分区域。

|  |  |
| --- | --- |
| 动作 | 作用 |
| lock  （锁定） | 作用于主内存变量，把一个变量标示为一条线程独占的状态 |
| unlock  （解锁） | 作用于主内存的变量，把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定 |
| read  （读取） | 作用于主内存的变量，把一个变量的值从主内存传输到线程的工作内存中，以便随后的load动作使用 |
| load  （载入） | 作用于工作内存的变量，把read操作从主存中得到的变量值放入工作内存的变量副本中 |
| use  （使用） | 作用于工作内存的变量，把工作内存中一个变量的值传递给执行引擎，每当虚拟机遇到一个需要使用到变量的值的字节码指令时将会执行这个操作 |
| assign  （赋值） | 作用于工作内存的变量，把一个从执行引擎接收到的值赋给工作内存中的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作 |
| store  （存储） | 作用于工作内存的变量，把工作内存中一个变量的值传送到主内存中，以便随后的write操作使用 |
| write  （写入） | 作用于主内存的变量，把store操作从工作内存中得到的变量的值放入主内存的变量中 |



Volatile

非 volatile 变量进行读写的时候，每个线程先从内存拷贝变量到CPU缓存中。而声明变量是 volatile 的，JVM 保证了每次读变量都从内存中读，跳过 CPU cache 这一步。volatile 保证了新值能立即同步到主内存，保证了可见性，一个线程修改的结果，另一个线程马上就能看到。

指令重排序：是指CPU采用了允许将多条指令不按程序规定的顺序分开发送给各相应电路单元处理。在没有同步的情况下，编译器、处理器以及运行时等都可能对操作的执行顺序进行一些意想不到的调整。在缺乏足够同步的多线程程序中，要想对内存操作的执行顺序进行判断，无法得到正确的结论。当把变量声明为volatile类型后，禁止指令重排序优化，编译器与运行时都会注意到这个变量是共享的，因此不会将该变量上的操作与其他内存操作一起重排序。

在 Java 中 volatile、synchronized 和 final 实现可见性。

在 Java 中 synchronized 和在 lock、unlock 中操作保证原子性。

Java 语言提供了 volatile 和 synchronized 两个关键字来保证线程之间操作的有序性。

wait() 和 sleep()方法有什么不同？

wait和sleep都是阻塞状态，sleep不会释放锁，wait会释放锁。

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

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。

多用并发集合少用同步集合，如果下一次你需要用到map，你应该首先想到用ConcurrentHashMap。

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，

#### 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。

#### 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。此外，还有异步和非阻塞式方法在任务完成前就返回

#### 如果你提交任务时，线程池队列已满。会时发会生什么？

submit()方法将会抛出一个RejectedExecutionException异常。

#### 有三个线程T1，T2，T3，怎么确保它们按顺序执行？

在多线程中有多种方法让线程按特定顺序执行，你可以用线程类的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。

#### 如何写代码来解决生产者消费者问题？

比较赞的办法是用Semaphore 或者 BlockingQueue来实现生产者消费者模型

#### Java中Runnable和Callable有什么不同？

Runnable和Callable都代表那些要在不同的线程中执行的任务。Runnable从JDK1.0开始就有了，Callable是在JDK1.5增加的。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。

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。这是一个严重的问题，因为死锁会让你的程序挂起无法完成任务，死锁的发生必须满足以下四个条件：

* 互斥条件：一个资源每次只能被一个进程使用。
* 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
* 不剥夺条件：进程已获得的资源，在末使用完之前，不能强行剥夺。
* 循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系。

避免死锁最简单的方法就是阻止循环等待条件，将系统中所有的资源设置标志位、排序，规定所有的进程申请资源必须以一定的顺序（升序或降序）做操作来避免死锁。