1. 线程的创立的两种方法：

1.1通过继承Thread类

Public Class ThreadExtends extends Thread{

public void run(){

//线程内需要执行的方法。

}

}

通过

ThreadExtends t1=new ThreadExtends();

T1.start(); 开启线程，等到OS把执行权给了这个线程，它就会运行run方法。

1.2通过实现Runnable接口

**private static class TestClient implements** Runnable //  
{  
 **private ThreadUseTest** sn;  
 **public** TestClient(**ThreadUseTest** sn) {  
 **this**.sn = sn;  
 }  
 **public void** run()  
 {  
 //线程内需要执行的方法。  
 }  
}

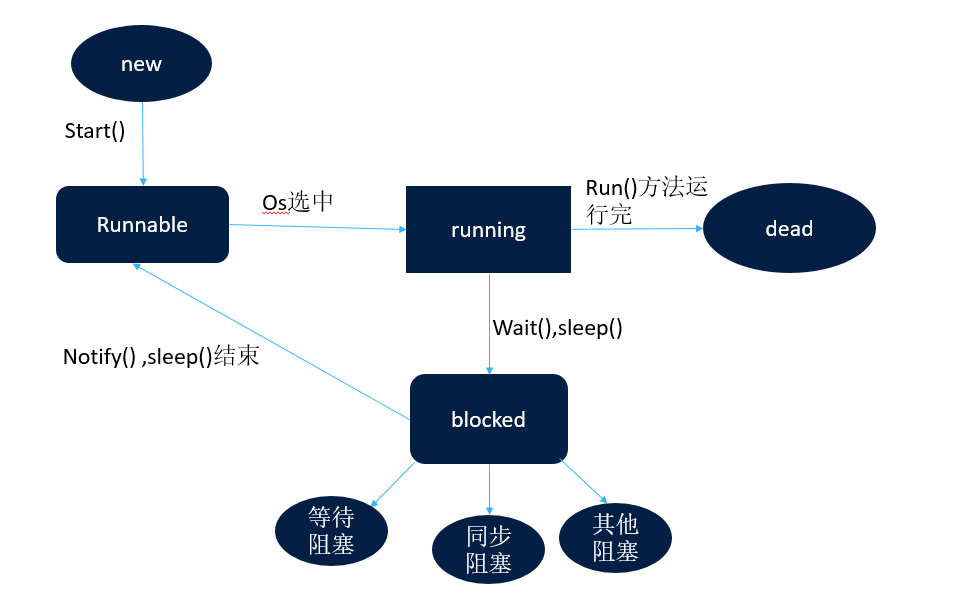
通过

**TestClient** t1 = **new** TestClient(sn);  
**Thread** thread1=**new** Thread(t1);  
**Thread** thread2=**new** Thread(t1);  
**Thread** thread3=**new** Thread(t1);  
thread1.start();  
thread2.start();  
thread3.start();

通过Runable接口

**每个线程调用的是同一个ThreadClient对象中的run()方法，访问的是同一个对象中的变量（sn）的实例，所以可以实现多个线程共享一个资源。**

二．线程的状态以及转换



各种状态的转换很明显，值得一说的是running过程中，可能出现的blocked的状态：

1. 调用join()和sleep()方法，sleep()时间结束或被打断，join()中断，都会回到Runable状态，等待jvm调度。

 当A 线程执行到了 B 线程的join() 方法时，A 线程就会等待，等 B 线程执行完，A 才会执行。即A 线程 等待B 线程终止。

|  |
| --- |
| 创建Runnable 接口实现类的实例  12 JoinText t = new JoinText() ;  13 // 通过 Thread(Runnable target) 创建新线程  14 Thread thread = new Thread(t) ;  15 Thread thread1 = new Thread(t) ;  16 // 启动线程  17 thread.start() ;  18 // 只有等thread 线程执行结束，main 线程才会向下执行；  19 thread.join() ;  20 System.out.println("thread1 线程将要启动");  21 thread1.start() ; |

1. 同步函数或者同步代码快中，通过wait()也会使得线程进入等待池中，知道notify()，线程被唤醒放到锁定池中，释放同步锁，使线程回到可运行状态。

三．多线程的安全问题

3.1问题出现的原因

当多条语句在操作同一个线程共享数据，一个线程对多条语句只执行了 一部 分，还没执行完，另一个线程参与进来执行，会导致共享数据错误。

这就是所谓的并发问题！！

3.2问题解决的办法：

3.2.1 采用synchronized 同步：

在函数上加synchronized同步关键字，那么同步函数使用的锁是this 锁。在同步中，才可以使用wait()、notify()、notifyAll()等函数。

这是因为java中的每个对象都有一个监视器Monitor，在非并发的程序 中，监视器不发挥作用，而在synchronized中监视器发挥作用。

注意：在synchronized块中可以用任意对象作为锁和监视器。

典型应用：

多个生产者，多个消费者。

|  |
| --- |
| /\*\*  \* Created by yuandl on 2016-10-11./\*\*  \* 资源  \*/  public class Resource {  /\*资源序号\*/  private int number = 0;  /\*资源标记\*/  private boolean flag = false;  /\*\*  \* 生产资源  \*/  public synchronized void create() {  while (flag) {//先判断标记是否已经生产了，如果已经生产，等待消费；  try {  wait();//让生产线程等待  } catch (InterruptedException e) {  e.printStackTrace();  }  }  number++;//生产一个  System.out.println(Thread.currentThread().getName() + "生产者------------" + number);  flag = true;//将资源标记为已经生产  notifyAll();//唤醒在等待操作资源的线程（队列）  }  /\*\*  \* 消费资源  \*/  public synchronized void destroy() {  while (!flag) {  try {  wait();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  System.out.println(Thread.currentThread().getName() + "消费者\*\*\*\*" + number);  flag = false;  notifyAll();  }  } |

这里详解一下：

为什么判断标志需要用while()，

因为wait()锁定的线程，被唤醒的时候会从当前唤醒，如果不用while()它就会继续执行下去，会出现差错。应该让它循环回来再判断一次。

为什么要用notifyAll()?

因为notify()唤醒线程是随机的，这是因为一个锁就只有一个监视器，每一个线程都是通过这个监视器wait()的，自然唤醒的时候是随机的，但是如果唤醒的仍然是相同类型的线程，就会产生死锁问题。

这个代码执行结果？

生产1，消费1，生产2，消费2，，这样很有规律的运作，但实际上这样是不好的，可以把while(flag)改成了while(num<maxSize)；这样判断条件就不会强制性生产一个。

3.2.1 采用Lock机制：

上面提到了，采用synchronized机制锁和监视器是绑定在一起的，因此每一个线程的wait(),notify()都是通过一个监视器来弄的，唤醒的时候就不能准确唤醒想要唤醒的线程。因此有了锁机制这个东西了。

Lock接口，将锁单独封装成了一个接口。并且将锁和监视器分开了Condition 监视器类。

简单用法：

public synchronized void method(){

}

转换成Lock机制的

Lock lock=new ReentrantLock(); //创立一个锁

Condition condition1=lock.newConditon(); //绑定第一个监视器

Condition condition2=lock.newConditon(); //绑定第二个监视器

public void method(){

lock.lock(); //得到锁

//需要同步的代码。 多个线程共用一个锁，但是各有各的监听器

Condition1.await(); //冻结指定线程

Condition1.signal(); //唤醒指定线程。

finally{

lock.unlock(); //释放锁

}

}

3.3 死锁程序

一个简单的死锁就是a方法持有了b方法需要的锁，b方法持有了a方法需要的锁。

1. **public** **void** run(){
2. **if**(flag){
3. **while**(**true**){                           //线程t1进入
4. **synchronized** (MyLock.lockA) {
5. System.out.println("if LockA");
6. **synchronized** (MyLock.lockB) {
7. System.out.println("if LockB");
8. }
9. }
10. }
11. }**else**{
12. **while**(**true**){                          //线程t2进入
13. **synchronized** (MyLock.lockB) {
14. System.out.println("else LockB");
15. **synchronized** (MyLock.lockA) {
16. System.out.println("else LockA");
17. }
18. }
19. }
20. }
21. }

**线程（Thread），锁（wait()、notify()）,任务（Runnable）,对资源执行不同的操作，这个可以通过不同任务来区分，但对资源相同操作的细节，就应该用锁来限制（wait()当前线程，notify()唤醒锁池中线程，但这个唤醒是随机的），同时，对资源的操作为了保持并发性，应该使用同一把锁。**

**一把锁，多个监视器，一个监视器，监视一个任务。**

四．线程间通信和不通信对于并发问题的解决

4.1线程通信

线程间通信指，多个线程会对同一资源进行操作，每一个线程对资源有不同的任务操作，所以这个时候为了避免并发问题，就采用同步的方式来解决问题。

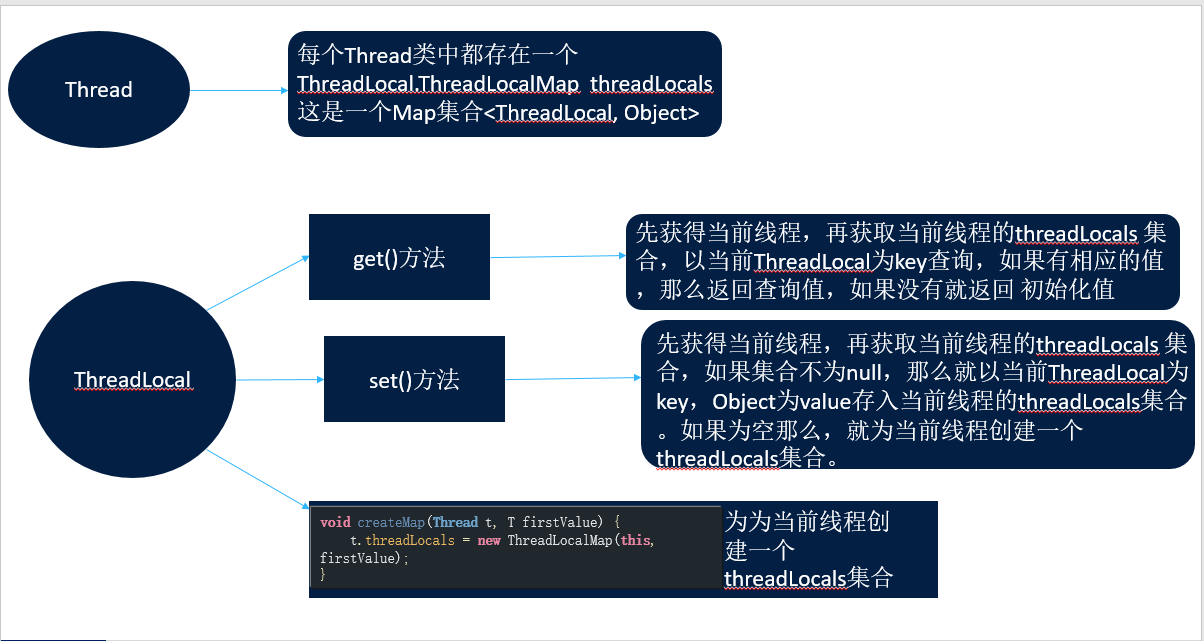
关于同步如何解决 前面已经详细讲了。

4.2 线程间不通信

线程间不通信，指的是，对多个线程都要使用那个变量，但不需要互相干扰。

对于这时候如果并发，可以采用ThreadLocal 来解决。

当很多线程需要多次使用同一个对象，并且需要该对象具有相同初始化值的时候最适合使用ThreadLocal。



如上图可以看出，ThreadLocal实际上，是一个中介，通过这个中介，可以为每一个线程都创建自己独有的ThreadLocalMap 集合，把变量复制并且存到每个线程对应的ThreadLocalMap 集合（key为当前ThreadLocal）中。这样就实现了，线程隔离使用变量。

且，ThreadLocal<T> 是一个泛型类，因此，最后存入ThreadLocalMap中key类型为ThreadLocal ，value类型为 T

同时，对需要在一个线程不同层使用的对象，也可以存到ThreadLocal中去，ThreadLocal贯穿整个线程，因此对象也就贯穿整个线程。

4.2.1 ThreadLocal使用的一般步骤

1、在多线程的类（如ThreadDemo类）中，创建一个ThreadLocal对象threadXxx，用来保存线程间需要隔离处理的对象xxx。

2、在ThreadDemo类中，创建一个获取要隔离访问的数据的方法getXxx()，在方法中判断，若ThreadLocal对象为null时候，应该new()一个隔离访问类型的对象，并强制转换为要应用的类型。

3、在ThreadDemo类的run()方法中，通过getXxx()方法获取要操作的数据，这样可以保证每个线程对应一个数据对象，在任何时刻都操作的是这个对象。

下面就是上面说的getXxx()方法（参考）：

**public int getNextNum**(){  
 *threadCurr*.set(*threadCurr*.get()+1); //设置当前线程的变量值。即：当前线程为key，threadCurr.get()+1 为 value  
 **return** *threadCurr*.get(); //得到value的值。  
}

五．线程池相关

5.1 为什么需要线程池？

线程池和连接池的作用其实是一样的，由于打开线程，关闭线程同打开连接和关闭连接一样也是很耗资源的。因此在高并发的条件下，如果可以在一个线程执行完任务后不关闭，而是继续执行下一个任务。那么就不会浪费资源了。

5.2 java线程池的原理

Java线程池主要是通过线程池的状态以及当前存在的线程数量来实现一个线程执行多个任务以及线程池中放入多个线程。

5.2.1 线程池的状态：

Running 线程池刚创建时候的状态。

ShutDown 在执行完shutDown()方法后的状态，此时线程池不允许加入新

的任务，但会等待任务缓存队列中任务执行完。

Stop 执行完shutDownNow()方法后，和上面不同，任务队列中都会不执行

Terminated shutdown 和 Stop 的最终状态。

5.2.2 线程池的运行过程。

1.当添加任务的时候，如果此时线程池是运行状态的时候，且线程数大于线程池核心数的时候，就把任务放入到任务队列中，等待执行。如果此时核心队列都满了的时候，就会强行创建线程去执行（只要这个线程数还小于maxminSize）

2.当线程数小于核心数的时候，就会创建一个新的线程去执行。

创建一个新线程，并把它加入到工作线程中的第一个线程。

public void run() {

try {

Runnable task = firstTask; //把第一个线程指向一个新任务。以此为中介执行完刚创立的任务，然后以此为中介，执行从任务队列中得到的任务。

firstTask = null;

while (task != null || (task = getTask()) != null) {

runTask(task);

task = null;

}

} finally {

workerDone(this);

}

}

3.线程池不能再加入任务了，那么就会执行 拒绝 策略。

为什么线程池中要有很多个if判断，因为前面的判断都是未加锁的，有可能会变化。

5.2.3 线程池的使用

实际上，常用线程池并不是，你通过

**ThreadPoolExecutor** executor = **new ThreadPoolExecutor**(5, 10, 200, TimeUnit.MILLISECONDS,  
 **new** ArrayBlockingQueue<Runnable>(5));

这样来创造的，因为线程池的配置非常的复杂。通常都是通过Executors类中的静态方法来实现的。

一共有4种：

newFixedThreadPool创建的线程池corePoolSize和maximumPoolSize值是相等的，它使用的LinkedBlockingQueue；

newSingleThreadExecutor将corePoolSize和maximumPoolSize都设置为1，也使用的LinkedBlockingQueue；

newCachedThreadPool将corePoolSize设置为0，将maximumPoolSize设置为Integer.MAX\_VALUE，使用的SynchronousQueue，也就是说来了任务就创建线程运行，当线程空闲超过60秒，就销毁线程。 （最常用）

创立一个线程池，通过execute(Runnable task)来执行任务。