1. 传统的线程方式

|  |
| --- |
| **package** cn.huangwei;  **public** **class** TraditionalThread {  **public** **static** **void** main(String[] args) {  /\*  \* thread一旦start之后就开始执行run方法  \* run里面的代码  \* public void run() {  if (target != null) {  target.run();  }  }  \*/  //第一种：一般创建thread的子类，改写run方法  Thread thread = **new** Thread(){  @Override  **public** **void** run() {};  thread.start();  //第二种：target是runnable类的变量  /\*  \* 通过调用init方法来传入target变量，由于init是私有的，  \* 可以认为是由构造器进行调用，下面的就是，  \* public Thread(Runnable target) {  init(null, target, "Thread-" + nextThreadNum(), 0);  }调用的  \*/  Thread thread2 = **new** Thread(**new** Runnable(){  @Override  **public** **void** run() {});  thread2.start();    //run方法的执行顺序  /\*  \* public void run() {  if (target != null) {  target.run();  }  }  通过代码可知，该run方法是父类的run方法，如果存在有  Thread的子类，那么就会覆盖父类的run方法，即子类的  run方法优先执行  如果没有覆盖子类，那么就按照父类的run方法执行，只要  target不为空，那么就执行target的run方法。  \*/  **new** Thread(**new** Runnable(){  @Override  **public** **void** run() {}  }){  **public** **void** run(){}  }.start();  }  } |

|  |
| --- |
| Java中Runnable和Thread的区别  实现Runnable接口相比继承Thread类有如下好处： 1、避免继承的局限，一个类可以实现多个接口。 2、适合于资源的共享。  Thead子类每个线程都独立，不共享资源  实现接口每个线程共享了对象myRunnable的资源  用runnable更加符合面向对象的思想 |

1. 定时器的使用

|  |
| --- |
| package cn.huangwei;  import java.util.Date;  import java.util.Timer;  import java.util.TimerTask;  public class TraditionalTimerTest {  private static int count = 0;  public static void main(String[] args) {  /\*new Timer().schedule(new TimerTask(){  @Override  public void run() {  // TODO Auto-generated method stub  System.out.println("bombing!");  }  }, 10000, 3000);//第一次10s炸，接下来每隔3s炸一次  while(true){  System.out.println(new Date().getSeconds());  try {  Thread.sleep(1000);  } catch (InterruptedException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }    }\*/    class MyTimerTask extends TimerTask{    @Override  public void run() {  count = (count + 1) % 2;  System.out.println("bombing!");  new Timer().schedule(new MyTimerTask(), 2000 + 2000 \* count);  }  }  new Timer().schedule(new MyTimerTask(), 2000);  while(true){  System.out.println(new Date().getSeconds());  try {  Thread.sleep(1000);  } catch (InterruptedException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }    }  }  } |

1. 传统的synchronized

|  |
| --- |
| **package** cn.huangwei;  **public** **class** TraditionalThreadSynchronized {  **public** **static** **void** main(String[] args) {  **new** TraditionalThreadSynchronized().init();  }    **private** **void** init(){  **final** Outputer outputer = **new** Outputer();  **new** Thread(**new** Runnable(){  @Override  **public** **void** run() {  // **TODO** Auto-generated method stub  **while**(**true**){  **try** {  Thread.*sleep*(10);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  outputer.output("zhangxiaoxiang");  }  }  }).start();    **new** Thread(**new** Runnable(){  @Override  **public** **void** run() {  // **TODO** Auto-generated method stub  **while**(**true**){  **try** {  Thread.*sleep*(10);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  outputer.output("lihuoming");  }  }  }).start();  }  **static** **class** Outputer{  // String xx = "";  **public** **void** output(String name){//或者方法修饰符加synchronized  **int** len = name.length();  **synchronized**(**this**){  **for**(**int** i = 0; i < len; i++){  System.***out***.print(name.charAt(i));  }  System.***out***.println();  }  }    /\*  \* output1与output3能否互斥  \* 不能，静态方法运行的时候有个对象和他关联，那就是  \* 类的字节码，所以output3的synchronized锁的是  \* 类的字节码。  \* 只要把output1的this改成xx.class即可实现同步  \*/    **public** **synchronized** **void** output2(String name){  **int** len = name.length();  **for**(**int** i = 0; i < len; i++){  System.***out***.print(name.charAt(i));  }  System.***out***.println();  }    **public** **static** **synchronized** **void** output3(String name){  **int** len = name.length();  **for**(**int** i = 0; i < len; i++){  System.***out***.print(name.charAt(i));  }  System.***out***.println();  }  }  } |

1. 子线程循环10次，接着住线程循环100次，接着又回到子线程循环10次，接着在回到主线程又循环100，如此循环50次，请写出程序

|  |
| --- |
| **package** cn.huangwei;  **public** **class** TraditionalThreadCommunication {  **public** **static** **void** main(String[] args) {  **final** Business business = **new** Business();  **new** Thread(**new** Runnable(){  @Override  **public** **void** run() {  // **TODO** Auto-generated method stub    **for**(**int** i = 1; i < 50; i++){  business.sub(i);  }  }  }).start();      **for**(**int** i = 1; i < 50; i++){  business.main(i);  }  }    **static** **class** Business{  **private** **boolean** shouldSub = **true**;  **public** **synchronized** **void** sub(**int** i){  **if**(!shouldSub){  **try** {  **this**.wait();  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  **for**(**int** j = 1; j <= 10; j++){  System.***out***.println("sub thread sequence of " + j + ", loop of " + i );  }  shouldSub = **false**;  **this**.notify();  }    **public** **synchronized** **void** main(**int** i){  **if**(shouldSub){  **try** {  **this**.wait();  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  **for**(**int** j = 1; j <= 20; j++){  System.***out***.println("main thread sequence of " + j + ", loop of " + i );  }  shouldSub = **true**;  **this**.notify();  }  }  } |

1. ThreadLocal

|  |
| --- |
| **package** cn.huangwei;  **import** java.util.Random;  **public** **class** ThreadLocalTest {  **private** **static** ThreadLocal<Integer> *x* = **new** ThreadLocal<>();  **private** **static** ThreadLocal<MyThreadScopeData> *myThreadScopeData* = **new** ThreadLocal<>();  **public** **static** **void** main(String[] args) {  **for**(**int** i = 0; i < 2; i++){  **new** Thread(**new** Runnable(){  @Override  **public** **void** run() {  **int** data = **new** Random().nextInt();  System.***out***.println(Thread.*currentThread*().getName()  + " has put data: " + data );  *x*.set(data);  /\*MyThreadScopeData myData = new MyThreadScopeData();  myData.setName("name " + data);  myData.setAge(data);  myThreadScopeData.set(myData);\*/  MyThreadScopeData.*getThreadInstance*().setName("name " + data);  MyThreadScopeData.*getThreadInstance*().setAge(data);    **new** A().get();  **new** B().get();  }  }).start();  }    }    **static** **class** A{  **public** **void** get(){  **int** data = *x*.get();  System.***out***.println("a from " + Thread.*currentThread*().getName()  + " get data: " + data);  /\*MyThreadScopeData myData = myThreadScopeData.get();  System.out.println("a from " + Thread.currentThread().getName()  + " getMydata: " + myData.getName() + "," + myData.getAge());\*/  MyThreadScopeData myData = MyThreadScopeData.*getThreadInstance*();  System.***out***.println("a from " + Thread.*currentThread*().getName()  + " getMydata: " + myData.getName() + "," + myData.getAge());  }  }    **static** **class** B{  **public** **void** get(){  **int** data = *x*.get();  System.***out***.println("b from " + Thread.*currentThread*().getName()  + " get data: " + data);  MyThreadScopeData myData = MyThreadScopeData.*getThreadInstance*();  System.***out***.println("b from " + Thread.*currentThread*().getName()  + " getMydata: " + myData.getName() + "," + myData.getAge());  }  }  }  **class** MyThreadScopeData{  **private** **static** ThreadLocal<MyThreadScopeData> *map* = **new** ThreadLocal<>();    **private** MyThreadScopeData(){    }  **public** **static** MyThreadScopeData getThreadInstance(){  MyThreadScopeData instance = *map*.get();  **if**(instance == **null**){  instance = **new** MyThreadScopeData();  *map*.set(instance);  }  **return** instance;  }  **private** String name;  **private** **int** age;  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **int** getAge() {  **return** age;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }    } |

1. 设计4个线程，其中两个线程每次对j增加1，另外两个线程对j每次减少1

|  |
| --- |
| **package** cn.huangwei;  **public** **class** ThreadTestIncDec {  **private** **int** j;  **public** **static** **void** main(String[] args) {  ThreadTestIncDec ttid = **new** ThreadTestIncDec();  Inc inc = ttid.**new** Inc();  Dec dec = ttid.**new** Dec();  **for**(**int** i = 0; i < 2; i++){  **new** Thread(inc).start();  **new** Thread(dec).start();  }  }    **class** Inc **implements** Runnable{  **public** **void** run(){  **for**(**int** i = 0; i < 10; i++){  inc();  }  }  }    **class** Dec **implements** Runnable{  **public** **void** run(){  **for**(**int** i = 0; i < 10; i++){  dec();  }  }  }  **private** **synchronized** **void** inc() {  // **TODO** Auto-generated method stub  j++;  System.***out***.println(Thread.*currentThread*().getName()+ "-inc" + j);  }    **private** **synchronized** **void** dec() {  // **TODO** Auto-generated method stub  j--;  System.***out***.println(Thread.*currentThread*().getName()+ "-dec" + j);  }  } |

1. 线程池

|  |
| --- |
| **package** cn.huangwei;  **import** java.util.concurrent.ExecutorService;  **import** java.util.concurrent.Executors;  **import** java.util.concurrent.TimeUnit;  **public** **class** ThreadPoolTest {  **public** **static** **void** main(String[] args) {  //固定线程池  //ExecutorService threadPool = Executors.newFixedThreadPool(3);  //缓存线程池，线程不够自动创建新的线程  //ExecutorService threadPool = Executors.newCachedThreadPool();  //相当于单线程，池子里一个线程死了，会自动找另一个线程，保证只有一个线程运行  //如何实现线程死掉后，重新启动  ExecutorService threadPool = Executors.*newSingleThreadExecutor*();  **for**(**int** i = 1; i <= 10; i++){  **final** **int** task = i;  threadPool.execute(**new** Runnable(){  @Override  **public** **void** run() {  // **TODO** Auto-generated method stub  **for**(**int** j = 1; j < 10; j ++){  **try** {  Thread.*sleep*(20);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  System.***out***.println(Thread.*currentThread*().getName() + " is looping of " + j + " for task " + task);  }  }  });  }  System.***out***.println("all of 10 task have committed! ");  threadPool.shutdown();//所有任务都执行完了就结束；    //线程池启动定时器  Executors.*newScheduledThreadPool*(3).schedule(**new** Runnable(){  @Override  **public** **void** run() {  // **TODO** Auto-generated method stub  System.***out***.println("bombing!");  }  }, 5, TimeUnit.***SECONDS***);  //5秒之后开始爆炸，接下来每2秒炸一次  Executors.*newScheduledThreadPool*(3).scheduleAtFixedRate(**new** Runnable(){  @Override  **public** **void** run() {  // **TODO** Auto-generated method stub  System.***out***.println("bombing!");  }  }, 5, 2, TimeUnit.***SECONDS***);    }  } |

1. Callable 和Future

|  |
| --- |
| **package** cn.huangwei;  **import** java.util.Random;  **import** java.util.concurrent.Callable;  **import** java.util.concurrent.CompletionService;  **import** java.util.concurrent.ExecutionException;  **import** java.util.concurrent.ExecutorCompletionService;  **import** java.util.concurrent.ExecutorService;  **import** java.util.concurrent.Executors;  **import** java.util.concurrent.Future;  **public** **class** CallableAndFuture {  **public** **static** **void** main(String[] args) {  ExecutorService threadPool = Executors.*newSingleThreadExecutor*();  //callable泛型是什么，future泛型也是什么  Future<String> future =  threadPool.submit(**new** Callable<String>(){  **public** String call() **throws** Exception{  Thread.*sleep*(2000);  **return** "hello";  }  });  System.***out***.println("等待结果：");  **try** {  System.***out***.println("拿到结果：" + future.get());  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  } **catch** (ExecutionException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }    ExecutorService threadPool2 = Executors.*newFixedThreadPool*(10);  CompletionService<Integer> completionService = **new** ExecutorCompletionService<>(threadPool2);  **for**(**int** i = 1; i <= 10; i++){  **final** **int** seq = i;  completionService.submit(**new** Callable<Integer>(){  @Override  **public** Integer call() **throws** Exception {  // **TODO** Auto-generated method stub  Thread.*sleep*(**new** Random().nextInt(5000));  **return** seq;  }  });  }    //take方法返回已完成的一个callable任务对应的future对象  //可以用于同时种几块地的麦子，然后等待收割，收割时，哪块先熟，先割哪块  **for**(**int** i = 0; i < 10; i++){  **try** {  System.***out***.println(completionService.take().get());  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  } **catch** (ExecutionException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  }    } |

1. ReadWriteLock

|  |
| --- |
| **package** cn.huangwei;  **import** java.util.Random;  **import** java.util.concurrent.locks.ReentrantReadWriteLock;  **public** **class** ReadWriteLockTest {  **public** **static** **void** main(String[] args) {  **final** Queue3 q3 = **new** Queue3();  **for**(**int** i = 0; i < 3; i++){  **new** Thread(){  **public** **void** run() {  **while**(**true**){  q3.get();  }  };  }.start();  **new** Thread(){  **public** **void** run() {  **while**(**true**){  q3.put(**new** Random().nextInt(10000));  }  };  }.start();  }  }    }  **class** Queue3{  **private** Object data = **null**;//共享数据，只能有一个线程使用  **private** ReentrantReadWriteLock rw1 = **new** ReentrantReadWriteLock();  **public** **void** get(){  rw1.readLock().lock();  **try**{  System.***out***.println(Thread.*currentThread*().getName() + "be ready to read data");  Thread.*sleep*((**long**)(Math.*random*()\*1000));  System.***out***.println(Thread.*currentThread*().getName() + "hava read data :" + data);  }**catch**(InterruptedException e){  e.printStackTrace();  }**finally**{  rw1.readLock().unlock();  }  }    **public** **void** put(Object data){  rw1.writeLock().lock();  **try**{  System.***out***.println(Thread.*currentThread*().getName() + "be ready to write data");  Thread.*sleep*((**long**)(Math.*random*()\*1000));  **this**.data = data;  System.***out***.println(Thread.*currentThread*().getName() + "have write data:" + data);  }**catch**(InterruptedException e){  e.printStackTrace();  }**finally**{  rw1.writeLock().unlock();  }  }  } |

1. Condition的功能类似在传统线程技术中的wait和notify的功能，在等待condition时，允许发生“虚假唤醒”，这通常作为对基础平台语义的让步。

|  |
| --- |
| package cn.huangwei;  import java.util.concurrent.locks.Condition;  import java.util.concurrent.locks.Lock;  import java.util.concurrent.locks.ReentrantLock;  public class ThreeConditionCommunication {  public static void main(String[] args) {  final Business business = new Business();  new Thread(new Runnable(){  @Override  public void run() {  // TODO Auto-generated method stub  try {  Thread.sleep(200);  } catch (InterruptedException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  for(int i = 1; i < 5; i++){  business.sub(i);  }  }  }).start();    new Thread(new Runnable(){  @Override  public void run() {  // TODO Auto-generated method stub  try {  Thread.sleep(200);  } catch (InterruptedException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  for(int i = 1; i < 5; i++){  business.sub2(i);  }  }  }).start();    try {  Thread.sleep(200);  } catch (InterruptedException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  for(int i = 1; i < 5; i++){  business.main(i);  }  }    static class Business{  Lock lock = new ReentrantLock();  Condition main = lock.newCondition();  Condition condition1 = lock.newCondition();  Condition condition2 = lock.newCondition();  private int shouldSub = 0;  public void sub(int i){  lock.lock();  try{  while(shouldSub != 1){  try {  // this.wait();  condition1.await();  } catch (Exception e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  }  for(int j = 1; j <= 10; j++){  System.out.println("sub1 thread sequence of " + j + ", loop of " + i );  }  shouldSub = 2;  // this.notify();  condition2.signal();  }finally{  lock.unlock();  }  }    public void sub2(int i){  lock.lock();  try{  while(shouldSub != 2){  try {  // this.wait();  condition2.await();  } catch (Exception e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  }  for(int j = 1; j <= 11; j++){  System.out.println("sub2 thread sequence of " + j + ", loop of " + i );  }  shouldSub = 0;  // this.notify();  main.signal();  }finally{  lock.unlock();  }  }    public void main(int i){  lock.lock();  try{  while(shouldSub != 0){  try {  // this.wait();  main.await();  } catch (Exception e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  }  for(int j = 1; j <= 12; j++){  System.out.println("main thread sequence of " + j + ", loop of " + i );  }  shouldSub = 1;  // this.notify();  condition1.signal();  }finally{  lock.unlock();  }  }  }  } |

1. Semaphore

|  |
| --- |
| package cn.huangwei;  import java.util.concurrent.ExecutorService;  import java.util.concurrent.Executors;  import java.util.concurrent.Semaphore;  /\*\*  \* semaphore实现的功能就类似厕所有5个坑，假如有十个人要上厕所  \* ，那么同事能有多少个人去上厕所呢，同时只有5个人能够占用，当5个人  \* 中的任何一个人让开后，其中在等待的另外5个人中又有一个人可以占用了  \*  \* 另外等待的5个人中可以随机获得优先机会，可以是按照先来后到的顺序获得机会  \* ，这取决于构造semaphore对象时传入的参数选项  \* final Semaphore sp = new Semaphore(3, true);true表示按照先来后来顺序  \*  \*单个信号量的semaphore对象可以实现互斥锁的功能，并且可以由一个线程  \*获得锁，另一个线程释放锁，应用于死锁回复的场合  \*  \* @author 49692  \*  \*/  public class SemaphoreTest {  public static void main(String[] args) {  ExecutorService service = Executors.newCachedThreadPool();  final Semaphore sp = new Semaphore(3);  for(int i = 0; i < 10; i ++){  Runnable runnable = new Runnable(){  public void run(){  try{  sp.acquire();  }catch(InterruptedException e1){  e1.printStackTrace();  }  System.out.println("线程" + Thread.currentThread().getName() +  "进入，当前已有" + (3 - sp.availablePermits()) + "并发");  try{  Thread.sleep((long)(Math.random() \* 1000));  }catch(InterruptedException e){  e.printStackTrace();  }  System.out.println("线程" + Thread.currentThread().getName() + "即将离开");  sp.release();  //下面代码有时候执行不准确，因为其没有和上面代码合成原子步骤  System.out.println("线程" + Thread.currentThread().getName() + "已离开，当前已有"  + (3-sp.availablePermits()) + "并发");  }  };  service.execute(runnable);  }  }  } |

1. CyclicBarrier

|  |
| --- |
| **package** cn.huangwei;  **import** java.util.concurrent.BrokenBarrierException;  **import** java.util.concurrent.CyclicBarrier;  **import** java.util.concurrent.ExecutorService;  **import** java.util.concurrent.Executors;  /\*\*  \*  \* 表示大家彼此等待，大家集合好后才出发，分散活动后又在指定地点  \* 结合碰面，这就好比整个公司的人员利用周末时间集体郊游一样，先各自  \* 从家出发到公司集合后，在同事出发到公园游玩，在指定地点集合后再  \* 同事开始就餐  \*  \* **@author** 49692  \*  \*/  **public** **class** CyclicBarrierTest {  **public** **static** **void** main(String[] args) {  ExecutorService service = Executors.*newCachedThreadPool*();  **final** CyclicBarrier cb = **new** CyclicBarrier(3);  **for**(**int** i = 0; i < 3; i++){  Runnable runnable = **new** Runnable(){  @Override  **public** **void** run() {  // **TODO** Auto-generated method stub  **try** {  Thread.*sleep*((**long**)(Math.*random*() \* 1000));  System.***out***.println("线程" + Thread.*currentThread*().getName() +  "即将到达集合地点1，当前已有" + (cb.getNumberWaiting() + 1) +"队伍到达" +  (cb.getNumberWaiting() == 2 ? "都到齐了，继续走" : "正在等候"));  cb.await();    Thread.*sleep*((**long**)(Math.*random*() \* 1000));  System.***out***.println("线程" + Thread.*currentThread*().getName() +  "即将到达集合地点2，当前已有" + (cb.getNumberWaiting() + 1) +"队伍到达" +  (cb.getNumberWaiting() == 2 ? "都到齐了，继续走" : "正在等候"));  cb.await();    Thread.*sleep*((**long**)(Math.*random*() \* 1000));  System.***out***.println("线程" + Thread.*currentThread*().getName() +  "即将到达集合地点3，当前已有" + (cb.getNumberWaiting() + 1) +"队伍到达" +  (cb.getNumberWaiting() == 2 ? "都到齐了，继续走" : "正在等候"));  cb.await();  } **catch** (InterruptedException | BrokenBarrierException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  };  service.execute(runnable);  }  service.shutdown();  }  } |

CountDownLatch

犹如倒计时计数器，调用countDownLatch对象的countDown方法就将计数器减1，当计数器到达0时，则所有等待者或者单个等待着开始执行，

可以实现一个人（也可以是多个人）等待其他人都来此通知他，可以实现一个人通知多个人的效果，类似裁判一声口令，运动员开始奔跑，或者所有运动员都跑到终点，裁判才可以公布结果

|  |
| --- |
| **package** cn.huangwei;  **import** java.util.concurrent.CountDownLatch;  **import** java.util.concurrent.ExecutorService;  **import** java.util.concurrent.Executors;  **public** **class** CountDownLatchTest {  **public** **static** **void** main(String[] args) {  ExecutorService service = Executors.*newCachedThreadPool*();  **final** CountDownLatch cdOrder = **new** CountDownLatch(1);  **final** CountDownLatch cdAnswer = **new** CountDownLatch(3);  **for**(**int** i = 0; i < 3; i++){  Runnable runnable = **new** Runnable(){  @Override  **public** **void** run() {  // **TODO** Auto-generated method stub  **try** {  System.***out***.println("线程" + Thread.*currentThread*().getName() +  "正在准备接受命令");  cdOrder.await();  System.***out***.println("线程" + Thread.*currentThread*().getName() +  "已接收命令");  Thread.*sleep*((**long**)Math.*random*()\*1000);  System.***out***.println("线程" + Thread.*currentThread*().getName() +  "回应命令处理结果");  cdAnswer.countDown();  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  };  service.execute(runnable);  }  **try**{  Thread.*sleep*((**long**)Math.*random*()\*1000);  System.***out***.println("线程" + Thread.*currentThread*().getName() +  "即将发布命令");  cdOrder.countDown();  System.***out***.println("线程" + Thread.*currentThread*().getName() +  "已发送命令，正在等待结果");  cdAnswer.await();  System.***out***.println("线程" + Thread.*currentThread*().getName() +  "已收到所有响应结果");  }**catch**(Exception e){  e.printStackTrace();  }  service.shutdown();    }  } |

Exchanger

用于实现两个人之间的数据交换，每个人在完成一定的事务后想与对方交换数据，第一个先拿出数据的人将一直等待第二个人拿着数据到来时，才能彼此交换数据

|  |
| --- |
|  |

1. 组赛队列

只有put和take有阻塞功能

|  |
| --- |
| **package** cn.huangwei;  **import** java.util.concurrent.ArrayBlockingQueue;  **import** java.util.concurrent.BlockingQueue;  **public** **class** BlockingQueueTest {  **public** **static** **void** main(String[] args) {  **final** BlockingQueue queue = **new** ArrayBlockingQueue(3);  **for**(**int** i = 0; i < 2; i++){  **new** Thread(){  **public** **void** run(){  **while**(**true**){  **try**{  Thread.*sleep*((**long**)Math.*random*() \* 1000);  System.***out***.println(Thread.*currentThread*().getName() + "准备放数据");  queue.put(1);  System.***out***.println(Thread.*currentThread*().getName() + "已经放了数据," +  "队列目前有" + queue.size() + "个数据");    }**catch**(Exception e){  e.printStackTrace();  }  }  }  }.start();    **new** Thread(){  **public** **void** run(){  **while**(**true**){  **try**{  //将此处分别改为100和1000观察结果  Thread.*sleep*(100);  System.***out***.println(Thread.*currentThread*().getName() + "准备取数据");  queue.take();  System.***out***.println(Thread.*currentThread*().getName() + "已经取了数据," +  "队列目前有" + queue.size() + "个数据");  }**catch**(Exception e){  e.printStackTrace();  }  }  }  }  .start();  }  }  } |

|  |
| --- |
| **package** cn.huangwei;  **import** java.util.concurrent.ArrayBlockingQueue;  **import** java.util.concurrent.BlockingQueue;  /\*  \* 用两个具有一个空间的队列来实现同步通知的功能  \*/  **public** **class** BlockingQueueThreadCommunication {  **public** **static** **void** main(String[] args) {  **final** Business business = **new** Business();  **new** Thread(**new** Runnable(){  @Override  **public** **void** run() {  // **TODO** Auto-generated method stub    **for**(**int** i = 1; i < 50; i++){  business.sub(i);  }  }  }).start();      **for**(**int** i = 1; i < 50; i++){  business.main(i);  }  }    **static** **class** Business{  BlockingQueue<Integer> queue1 = **new** ArrayBlockingQueue<Integer>(1);  BlockingQueue<Integer> queue2 = **new** ArrayBlockingQueue<Integer>(1);    {  **try** {  queue2.put(1);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }    **public** **void** sub(**int** i){  **try** {  queue1.put(1);  **for**(**int** j = 1; j <= 10; j++){  System.***out***.println("sub thread sequence of " + j + ", loop of " + i );  }  queue2.take();  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }    **public** **void** main(**int** i){  **try** {  queue2.put(1);  **for**(**int** j = 1; j <= 20; j++){  System.***out***.println("main thread sequence of " + j + ", loop of " + i );  }  queue1.take();  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  }  } |

1. ConcurrentModificationException

|  |
| --- |
| **package** cn.huangwei;  **import** java.util.ArrayList;  **import** java.util.Collection;  **import** java.util.Iterator;  **public** **class** CheckModifyUserTest {  **public** **static** **void** main(String[] args) {  Collection users = **new** ArrayList();  users.add(**new** User("张三",28));  users.add(**new** User("李四",25));  users.add(**new** User("王五",31));  Iterator itrUsers = users.iterator();  **while**(itrUsers.hasNext()){  User user = (User) itrUsers.next();  **if**("张三".equals(user.getName())){  users.remove(user);  }**else**{  System.***out***.println(user);    }  }  }  }          迭代的时候next方法要检查modcount和expectedModcount是否相等  增加三个user后modcount=3，此时得到iterator，此时的expectedModcount=3，后来进行删除后modcount=4，引起不相等，所以抛异常 |

1. 空中网面试题

|  |
| --- |
| package cn.huangwei.test;  import java.util.concurrent.ArrayBlockingQueue;  import java.util.concurrent.BlockingQueue;  public class TestPrint {  public static void main(String[] args) {    BlockingQueue<String> queue = new ArrayBlockingQueue<>(16);    for(int i = 0; i < 4; i++){  new Thread(new Runnable(){  @Override  public void run() {  while(true){  try {  String log = queue.take();  TestPrint.parseLog(log);  } catch (InterruptedException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  }  }  }).start();  }  System.out.println("begin:" + (System.currentTimeMillis() / 1000));  /\*  \* 模拟处理16行日志，下面的代码产生了16个日志对象，当前  \* 代码需要运行16秒才能打印完这些日志  \*  \* 修改程序代码，开四个线程让这16个对象在4秒钟打完  \*/  for(int i = 0; i < 16; i++){  final String log = "" + (i + 1);  {  //TestPrint.parseLog(log);//只能修改此处代码  try {  queue.put(log);  } catch (InterruptedException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  }  }    }  public static void parseLog(String log) {  // TODO Auto-generated method stub  System.out.println(log + ":" + System.currentTimeMillis()/1000);  try {  Thread.sleep(1000);  } catch (InterruptedException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  }  }  package cn.huangwei.test;  import java.util.concurrent.Semaphore;  import java.util.concurrent.SynchronousQueue;  /\*\*  \* testCustomer类中不断产生数据，然后交给testdo.dosome方法去处理  \* ，就好像生产者在不断的产生数据，消费者在不断的消费数据  \* 将程序改成10个线程来消费生产者产生数据，这些消费者都调用  \* dosome方法进行处理，每个消费者都需要一秒才能处理完，程序应保证  \* 这些消费者依次有序的消费数据，只有上一个消费者消费完了之后，  \* 下一个消费者才能消费数据，下一个消费者是谁都行，保证拿到的数据  \* 是有序的  \* @author 49692  \*  \*/  public class TestCustomer {  public static void main(String[] args) {  final Semaphore semaphore = new Semaphore(1);  //只有有人读的时候，才能放数据，否则堵塞  final SynchronousQueue<String> queue = new SynchronousQueue<>();  for(int i = 0; i < 10; i++){  new Thread(new Runnable(){  @Override  public void run() {  String input;  try {  semaphore.acquire();  input = queue.take();  String output = TestDo.doSome(input);  System.out.println(Thread.currentThread().getName() + ":" + output);  semaphore.release();  } catch (InterruptedException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  }  }).start();  }    System.out.println("begin:" + (System.currentTimeMillis()/1000));  for(int i = 0; i < 10; i++){  String input = i + "";  try {  queue.put(input);  } catch (InterruptedException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  }  }  }  class TestDo{  public static String doSome(String input){  try{  Thread.sleep(1000);  }catch(Exception e){  e.printStackTrace();  }  String output = input + ":" + (System.currentTimeMillis() / 1000);  return output;  }  }  **package** cn.huangwei.test;  **import** java.util.ArrayList;  **import** java.util.Iterator;  **import** java.util.concurrent.CopyOnWriteArrayList;  /\*\*  \* 现有程序同时启动4个线程去调用 TestDo。dosome方法，  \* 方法内代码先暂停一秒，然后输出以秒为单位的当前时间值，所以  \* 会打印4个相同值  \*  \* 修改代码，如果有几个线程调用dosome方法时，传递进去的key相等  \* equals比较为true，则这几个线程应互斥排队输出结果，即当有两个线程  \* 的key都是1时，他们中的一个要比另外其他线程晚一秒输出结果  \* **@author** 49692  \*  \*/  **public** **class** TestCustomer2 **extends** Thread{  **private** TestDoSome testDo;  **private** String value;  **private** String key;    **public** TestCustomer2(String key, String key2, String value) {  **this**.testDo = TestDoSome.*getInstance*();  /\*  \* 常量1和1是同一个对象，下面这行代码就是用1+的方式产生新对象，  \* 以实现内容没有改变，仍然相等，但对象却不在是同一个对象  \*/  **this**.key = key + key2;  **this**.value = value;  }  **public** **static** **void** main(String[] args) {  TestCustomer2 a = **new** TestCustomer2("1", "", "1");  TestCustomer2 b = **new** TestCustomer2("1", "", "2");  TestCustomer2 c = **new** TestCustomer2("3", "", "3");  TestCustomer2 d = **new** TestCustomer2("4", "", "4");  System.***out***.println("begin:" + (System.*currentTimeMillis*() / 1000));  a.start();  a.sleep();  b.start();  b.sleep();  c.start();  c.sleep();  d.start();  d.sleep();  }    **public** **void** run(){  testDo.doSome(key, value);  }  **private** **void** sleep() {  // **TODO** Auto-generated method stub  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  }  **class** TestDoSome{  **private** TestDoSome(){}  **private** **static** TestDoSome *\_instance* = **new** TestDoSome();  **public** **static** TestDoSome getInstance(){  **return** *\_instance*;  }  //容易出现问题，因为迭代过程中不能对集合进行操作，  //否则容易出现并发修改异常  //private ArrayList keys = new ArrayList();  **private** CopyOnWriteArrayList keys = **new** CopyOnWriteArrayList<>();  **public** **void** doSome(Object key, String value){  Object o = key;  /\*\*  \* 如果没有含有，就加入，如果有，就是用集合中原来  \* 对象作为o  \*/  **if**(!keys.contains(o)){//含有是指equals  keys.add(o);  }**else**{  //容易出现问题，因为迭代过程中不能对集合进行操作，  //否则容易出现并发修改异常  **for**(Iterator iter = keys.iterator(); iter.hasNext();){  **try** {  Thread.*sleep*(200);  Object oo = iter.next();  **if**(oo.equals(o)){  o = oo;  }  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  }  **synchronized**(o)  {  **try** {  Thread.*sleep*(1000);  System.***out***.println(key + ":" + value + ":"  +(System.*currentTimeMillis*()/1000));  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  }      } |