# 并发编程

## 多线程

|  |
| --- |
| 并发：多个任务提交，【1个cpu也可以，只不过cpu轮流执行。】  并行：多个任务执行，【多cpu/多核】 |
|  |

### 线程的创建

|  |
| --- |
| 线程有三种方式创建：   * **继承Thread类**   public class MyThread extends Thread {  @Override  public void run() {  System.out.println(Thread.currentThread().getName());  }  }  MyThread myThread = new MyThread();  myThread.start();   * **实现Runnable接口:常用解决不能多继承问题**   public class MyRunnable implements Runnable {  @Override  public void run() {  System.out.println(Thread.currentThread().getName());  }  }  new Thread(new MyRunnable(),"t2").start();  使用Thread启动   * **现实Callable接口**   类似Runnable但有返回值。返回值使用Future存储。详见[线程池](#_线程池) |

### 线程状态

|  |
| --- |
| 线程创建即为：新生状态  Start()后 就等待cpu的某时刻的调用，此时处于：就绪状态  进入就绪状态的情况：   * Start()方法 * yield() 让出Cpu 重新进入就绪状态   当cpu执行线程时：运行状态  运行结束即为：死亡状态  运行状态进入 阻塞状态：   * Sleep :不释放锁 。Timed Waiting 类的阻塞 * Wait ：释放锁 :WAITING * Join：WAITING * Io的read/write操作   从阻塞到就绪状态：   * Sleep时间到 * Notify/notifyAll :让当前锁对象上的线程 解除阻塞状态进入就绪状态，但不释放锁 |

### 停止线程

|  |
| --- |
| 不要使用 线程中的 stop()/destroy()方法，使用标识字段控制线程停止。如下：  public class TestThreadStop implements Runnable{   private volatile boolean flag = true;  **@Override** public void run() {  while(flag){  System.*out*.println(Thread.*currentThread*().getName() + **""** + System.*currentTimeMillis*());  }  }   public void setStop(){  flag = false;  }   public static void main(String[] args) {  TestThreadStop tts = new TestThreadStop();  new Thread(tts).start();  try {  Thread.*sleep*(1000);  } catch (InterruptedException e) {  e.printStackTrace();  }  tts.setStop();  } } |

### 线程的优先级

|  |
| --- |
| 线程的优先级，优先级高只是表示 被cpu执行的**概率高**，并不能保证一定优先执行。  默认所有线程的优先级均为5  Thread.MAX\_PRIORITY：**10**  Thread.NORM\_PRIORITY:**5**  Thread.MIN\_PRIORITY:**1**  如下：  Thread t2 = new Thread(() -> {  for (int i = 0; i < 100; i++) {  System.out.println(Thread.currentThread().getName() + ":" + i);  }  },"t2");  System.out.println(t2.getPriority());//获取优先级  t2.setPriority(Thread.MAX\_PRIORITY);//设置优先级  t2.start(); |

### 线程分类

|  |
| --- |
| 线程分为：**用户线程** 和 **守护线程（daemon）**  我们创建的一般线程均为 用户线程  守护线程是为 用户线程 服务的 jvm停止 不用等待守护线程执行完毕，只需要用户线程执行完就行。如下：  new Thread(() -> {  for (int i = 1; i < 366; i++) {  System.out.println(Thread.currentThread().getName() + ":" + i);  }  },"t1").start();  Thread daemon = new Thread(() -> {  while (true){  System.out.println(Thread.currentThread().getName() + ",daemon");  }  },"daemon");  **daemon.setDaemon(true);**  daemon.start();  使用**.setDaemon(true);设置为守护线程，jvm不会等守护线程执行完才停止，只要用户线程执行完就停止。** |

### 线程的生命周期

|  |
| --- |
| **Runnable ---->blocked**:如：加了synchronized的代码块或方法，只允许一个线程执行，其他线程执行时就从runnable转化为blocked状态。当获得锁就会从blocked---》runnable.  **Runnable--->waiting:**   * 调用wait() * 调用A.join()的线程会转换为waiting状态，当A执行完后又转换为Runnable状态。 * 调用LockSupport.park()方法。并发包中的方法。   **RUNNABLE与TIMED\_WAITING：**   1. 调用**带超时参数**的Thread.sleep(long millis)方法； 2. 获得synchronized隐式锁的线程，调用**带超时参数**的Object.wait(long timeout)方法； 3. 调用**带超时参数**的Thread.join(long millis)方法； 4. 调用**带超时参数**的LockSupport.parkNanos(Object blocker, long deadline)方法； 5. 调用**带超时参数**的LockSupport.parkUntil(long deadline)方法。   **New--->runnable：调用线程的start()方法。**  **Runnable--->terminated（结束）:**  一般run方法执行完就会转换为terminated状态，但有时需要主动停止线程，则需要：  Stop()(**已经废弃**)，推荐使用interrupt()  Stop方法会强制杀掉线程，有的线程还没有释放锁，stop后其他线程就永远无法获取到锁。  interrupt()方法会通知线程 |

### 常用方法

|  |
| --- |
| Thread.currentThread()//当前线程  t1.setName("t1");//设置线程name  t1.isAlive() //是否还存活 |

## 并发编程



### 多线程同步

|  |
| --- |
| 操作系统的多任务即为：多进程。如：wps,chrome,等等。。。  线程为进程的子任务  Synchronized线程同步，如果修饰对象：则为给这个对象加了一把锁，其他线程访问需先要获取这把锁，锁标记是在该对象存储的**堆**上存储标记的  private int count; private Object o = new Object();  public void m(){  synchronized(o){//或者this  count--;  System.*out*.println(Thread.*currentThread*().getName() + **" count = "** + count);  } }  当synchronized中代码执行结束时，则锁释放，这种只要有一个线程拿到锁，其他线程无法获取 为**互斥锁**  **Synchronized锁定的是对象，不是代码块,加了Synchronized的代码需要去申请对象锁**  public void m(){  synchronized(this){//锁定自身对象  count--;  System.out.println(Thread.currentThread().getName() + " count = " + count);  }  }  也可以一下简写：  public synchronized void m(){  count--;  System.out.println(Thread.currentThread().getName() + " count = " + count);  }  **当Synchronized修饰static方法时**，则表加锁的对象为 该类的class对象，因此，其他任意线程调用该类时都需要先获取 class对象锁。  private static int *count*;  public synchronized static void m(){  *count*--; }  相当于如下：  public static void m2(){  synchronized (Test2.class){  count--;  }  }  **Synchronized 的代码块是原子操作，其他线程不可打断，如下为线程重入**  public class Test3 implements Runnable {  private int count = 10;  @Override  public void run() {  count--;  System.out.println(Thread.currentThread().getName() + " count = " + count);  }  public static void main(String[] args) {  Test3 t3 = new Test3();  for (int i = 0; i < 5; i++) {  new Thread(t3,"Thread" + i).start();  }  }  }  由于几个线程对同一个对象 t3的属性 count进行操作，可能在第一个线程count--后还没有打印，其他线程又来执行，这时再去执行第一个线程打印时count的值可能被--了多次，加synchronized  public synchronized void run() {  count--;  System.out.println(Thread.currentThread().getName() + " count = " + count);  }  在同步方法执行期间，非同步方法是可以调用的，因为不需要去申请锁  public class Test4{   public synchronized void m1(){  System.*out*.println(Thread.*currentThread*().getName() + **" m1 start"**);  try {  Thread.*sleep*(10000);   } catch (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println(Thread.*currentThread*().getName() + **" m1 end"**);  }  public void m2(){  try {  Thread.*sleep*(5000);  System.*out*.println(Thread.*currentThread*().getName() + **"m2"**);  } catch (InterruptedException e) {  e.printStackTrace();  }  }   public static void main(String[] args) {  Test4 t4 = new Test4();  new Thread(t4::m1,**"t1"**).start();  new Thread(t4::m2,**"t2"**).start();  } }  new Thread(t4::m2,"t2").start();为Lambda表达式  相当于 new Runnable并在run方法中调用 t4.m2()方法  new Runnable(){  @Override  public void run() {  t4.m2();  }  } |

### 脏读

|  |
| --- |
| 由于给只给写操作加锁，没给读操作加锁而导致读取到写操作中没有完成的数据或（数据的差异），在写操作执行完后才允许读操作执行。**需看实际业务中是否允许脏读**  如下产生脏读：  public class Test5 {   private String name;  private double balance;   public synchronized void set(String name,double balance){  this.name = name;  try {  Thread.*sleep*(5000);  } catch (InterruptedException e) {  e.printStackTrace();  }  this.balance = balance;  }   public double get(String name){  return this.balance;  }   public static void main(String[] args) {  Test5 t5 = new Test5();  new Thread(()->t5.set(**"x"**,100),**"t1"**).start();  try {  Thread.*sleep*(2000);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println(t5.get(**"x"**));   try {  Thread.*sleep*(3000);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println(t5.get(**"x"**));  } } |

### Synchronized获得的锁是可重入的

|  |
| --- |
| 即为：**同一个线**程内一个同步方法可以调用另一个同步方法，以为已经获得了锁，再去获取只是它的所标记或+1，获得了两次  如下代码：  public class Test6 {   public synchronized void m1(){  System.*out*.println(**"m1.start"**);  try {  Thread.*sleep*(2000);  } catch (InterruptedException e) {  e.printStackTrace();  }  m2();  }   public synchronized void m2(){  try {  Thread.*sleep*(1000);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println(**"m2"**);  }   public static void main(String[] args) {  Test6 t6 = new Test6();  new Thread(t6::m1,**"t1"**).start();   } }  **子类的同步方法可以调用父类的同步方法。**  public class T {  public synchronized void m(){  System.out.println("T.m()");  }  public static void main(String[] args) {  new TT().m();  }  }  class TT extends T{  public synchronized void m(){  System.out.println("TT.m() start");  super.m();  System.out.println("TT.m() end");  }  }  注意：父类中this指的是new出来的子类对象，所以两个同步方法锁定的是一个对象 |

### Synchronized锁升级

|  |
| --- |
| jdk1.6之前为重量级锁，加锁和解锁的过程需要去os（操作系统）去申请，开销比较大，1.6之后得到了优化：（锁升级）   * **偏向锁**：当如果只有一个线程调用（对象头中）记录线程ID,如果该线程后边还需要申请锁，则直接使用不需要获取锁 * **自旋锁**：当第二个线程来获取锁时，升级为自旋锁，cpu占用一直在循环等待（10次）之后升级为重量级锁（等待队列）。 * 重量级锁：   加锁代码执行时间长，线程数多用系统锁  执行时间短，线程数少，用自旋锁。 |

### CAS

|  |
| --- |
| 无锁优化（自旋），必须硬件支持，cpu提供了**CAS指令**(全称是Compare And Swap,即“比较并交换”)。  CAS指令包含三个参数：   * 共享变量内存地址A * 用于比较的值B * 共享变量的新值C   并且只有当内存中地址A处的值等于B时,才能将内存中地址A处的值更新为新值C。  作为一条CPU指令,**CAS指令本身是能够保证原子性的** 。  原理：将共享变量A(可见的)进行操作（如A+1），传入A副本B接着判断此时的A是否等于B,不等于则表明A已经被其他线程修改，循环重新去做A+1操作接着在判断，直到A==B则表明没有其他线程修改A,此时再将A的值更新为（A+1的新值）。    如下伪代码：      Java原子类：Atomic×××其中的实现如下：实际使用见：[使用AtomicXXXX类保证原子性](#_使用AtomicXXXX类保证原子性)    分类： |

### AQS

|  |
| --- |
| AQS(AbstractQueuedSynchronizer):并发包中很多同步工具类的关键。（ReentrantLock,CyclicBarrier,Semaphore,CountDownLatch....）    ReentrantLock的锁是上边实现(定义state:1占用，0没有占用,大于1则为可重入)，如：CountDownLatch另外实现了state(不同表示)  ReentrantLock是否为公平锁，指：新来的线程是否直接竞争锁，如果为公平锁则：查看等待队列是否有线程，有则将该线程加入队尾。非公平则：上来直接竞争锁，没有竞争到则线程加入等待队列尾部（CAS加入）。锁都是由（队列：链表头节点线程去获取的） |

### 死锁

|  |
| --- |
| 线程A在获取获取锁o1期间又去获取锁o2,  而线程B在获取锁o2期间又去获取锁o1  当线程A在没有释放锁o1后锁o2又被B线程占用，但B线程又需要获取o1后才能释放o2所以就产生死锁 如：    public class A {  private Object o1,o2;  public A (Object o1,Object o2){  this.o1 = o1;  this.o2 = o2;  }   public void m1(){  synchronized (o1){  System.*out*.println(Thread.*currentThread*().getName() + **" o1 lock"**);  try {  Thread.*sleep*(3000);  } catch (InterruptedException e) {  e.printStackTrace();  }   synchronized (o2){  System.*out*.println(Thread.*currentThread*().getName() + **" o2 lock"**);  }  }   }   public void m2(){   synchronized (o2){  System.*out*.println(Thread.*currentThread*().getName() + **" o2 lock"**);  try {  Thread.*sleep*(5000);  } catch (InterruptedException e) {  e.printStackTrace();  }  synchronized (o1){  System.*out*.println(Thread.*currentThread*().getName() + **" o1 lock"**);  }  }    }  }  public static void main(String[] args) {  Object o1 = new Object();  Object o2 = new Object();  A a = new A(o1,o2);  new Thread(a::m1,"t1").start();  new Thread(a::m2,"t2").start();  }  只有以下这四个条件都发生时才会出现死锁：   1. 互斥，共享资源X和Y只能被一个线程占用； 2. 占有且等待，线程T1已经取得共享资源X，在等待共享资源Y的时候，不释放共享资源X； 3. 不可抢占，其他线程不能强行抢占线程T1占有的资源； 4. 循环等待，线程T1等待线程T2占有的资源，线程T2等待线程T1占有的资源，就是循环等待。   反过来分析，**也就是说只要我们破坏其中一个，就可以成功避免死锁的发生**。 破坏占用且等待条件：   class Allocator {  private List<Object> als = new ArrayList<>();  // 一次性申请所有资源  synchronized boolean apply(  Object from, Object to){  if(als.contains(from) ||  als.contains(to)){  return false;  } else {  als.add(from);  als.add(to);  }  return true;  }  // 归还资源  synchronized void free(  Object from, Object to){  als.remove(from);  als.remove(to);  }  }  class Account {  // actr应该为单例  private Allocator actr;  private int balance;  // 转账  void transfer(Account target, int amt){  // 一次性申请转出账户和转入账户，直到成功  while(!actr.apply(this, target))  ；  try{  // 锁定转出账户  synchronized(this){  // 锁定转入账户  synchronized(target){  if (this.balance > amt){  this.balance -= amt;  target.balance += amt;  }  }  }  } finally {  actr.free(this, target)  }  }  } 破坏循环等待条件：申请锁的顺序一样 class Account {  private int id;  private int balance;  // 转账  void transfer(Account target, int amt){  Account left = this ①  Account right = target; ②  if (this.id > target.id) { ③  left = target; ④  right = this; ⑤  } ⑥  // 锁定序号小的账户  synchronized(left){  // 锁定序号大的账户  synchronized(right){  if (this.balance > amt){  this.balance -= amt;  target.balance += amt;  }  }  }  }  } |

### 程序运行期间出现异常，锁默认会被释放

|  |
| --- |
| **当在同步代码块中，程序出现异常，则锁被释放。这样可能导致其他线程访问到不完整的数据。如下代码：**  public class Test {  private int count = 0;  public synchronized void add(){  System.out.println(Thread.currentThread().getName() + " start");  while (true){  try {  Thread.sleep(1000);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(Thread.currentThread().getName() + " count=" + count);  count ++;  if(count == 5){  count = 1/0;//异常，释放锁  }  }  }  public static void main(String[] args) {  Test t = new Test();  new Thread(t::add,"t1").start();  try {  Thread.sleep(3000);  } catch (InterruptedException e) {  e.printStackTrace();  }  new Thread(t::add,"t2").start();//只有t1线程释放锁后，t2才会执行  }  }  **将异常处理后就不会释放锁了，如下：**  while (true){  try {  Thread.sleep(1000);  System.out.println(Thread.currentThread().getName() + " count=" + count);  count ++;  if(count == 5){  count = 1/0;  }  } catch (Exception e) {  e.printStackTrace();  }  } |

### Volatile （线程之间可见性）

**可见性**和**禁止指令重排序**。

|  |
| --- |
| Volatile修饰 成员变量 ：表示，当数据被一个线程更改后，会通知其他线程 将数据从主内存区 更新到 cpu缓存中去。从而让 数据 实时保持一致  **成员变量是基本类型则可以保证可见性，成员变量为引用该类型，则只能保证引用可见性，不能保证引用属性可见性。**    如下：主线程修改 flag后 如果 flag 不使用volatile 关键字 则由于 t1线程忙碌（不好判断） 没有从主内存区更新 数据最新值，则线程t1无法停止  Cpu是否忙碌，是否去主内存中刷数据，不好判断，但加volatile会去通知，**无锁同步**  public class Test {  private volatile boolean flag = true;  public void m(){  System.out.println(Thread.currentThread().getName() + " start");  while (flag){  }  System.out.println(Thread.currentThread().getName() + " end");  }  public static void main(String[] args) {  Test t = new Test();  new Thread(t::m,"t1").start();  try {  Thread.sleep(2000);  } catch (InterruptedException e) {  e.printStackTrace();  }  t.flag = false;  }  } |

### Join

|  |
| --- |
| 插队执行，在线程1中执行 xxx.join()方法 线程1就需要等待 xxx线程执行完后才能执行，线程1处于**阻塞状态**  等待子线程运行完后，主线程接着运行  如：  public class Test {   public void m(){  System.*out*.println(Thread.*currentThread*().getName() + **":子线程"**);  try {  Thread.*sleep*(10000);  } catch (InterruptedException e) {  e.printStackTrace();  }  }   public static void main(String[] args) {  System.*out*.println(Thread.*currentThread*().getName() + **"：主线程start"**);  Test t = new Test();  Thread thread = new Thread(t::m,**"t1"**);  thread.start();  try {  thread.join();  } catch (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println(Thread.*currentThread*().getName() + **"：主线程end"**);   }  } |

### yield

|  |
| --- |
| 礼让：当前线程进入就绪状态，重新竞争cpu  new Thread(() ->{  for (int i = 0; i < 100; i++) {  System.out.println(Thread.currentThread().getName() + ":" + i);  }  },"t1").start();  for (int i = 0; i < 100; i++) {  if(i % 10 == 0){  Thread.yield();  }  System.out.println(Thread.currentThread().getName() + ":" + i);  } |

### wait/notify/notifyAll

|  |
| --- |
| 等待-通知机制  **线程首先获取互斥锁，当线程要求的条件不满足时，释放互斥锁，进入等待状态；当要求的条件满足时，通知等待的线程，重新获取互斥锁**。  当调用wait()方法后，当前线程就会被阻塞，并且进入到右边的等待队列中，**这个等待队列也是互斥锁的等待队列**。 线程在进入等待队列的同时，**会释放持有的互斥锁**，线程释放锁后，其他线程就有机会获得锁，并进入临界区了。  wait()、notify()、notifyAll()方法操作的等待队列是互斥锁的等待队列，所以如果synchronized锁定的是this，那么对应的一定是this.wait()、this.notify()、this.notifyAll()；  这三个方法能够被调用的前提是已经获取了相应的互斥锁，所以我们会发现wait()、notify()、notifyAll()都是在**synchronized**{}内部被调用的。  while(条件不满足) {  wait();  } 尽量使用notifyAll() Notify()会随机通知等待队列中的一个线程，虽然只有一个线程进入临界区，但可能导致有的线程永**远不被通知** |

### 使用AtomicXXXX类保证原子性

更多见上面章节：[CAS](#_CAS)

|  |
| --- |
| Volatile修饰变量只能保证 可见性，但不能保证原子性。如：  Volatile int n = 0;  M(){  n++;  }  使用synchronized效率低，可使用AtomicInteger  多个线程执行m方法并不能保证n的原子性，但可以使用AtomicInteger类，它其中的方法是原子性的并且是可见的，但**方法和方法之间并不具有原子性**。  public class Test {  AtomicInteger count = new AtomicInteger(0);   public void add(){  for (int i = 0; i < 100; i++) {  count.getAndIncrement();  }  }   public static void main(String[] args) {  ArrayList<Thread> threads = new ArrayList<Thread>();  Test t = new Test();  for (int i = 0; i < 10; i++) {  threads.add(new Thread(t::add,**"t"** + i));  }  threads.forEach((o) -> o.start());  threads.forEach((o) -> {  try {  o.join();  } catch (InterruptedException e) {  e.printStackTrace();  }  });  System.*out*.println(t.count);  }   }  public void add(){  for (int i = 0; i < 100; i++) {  If(count.get() < 10){  count.getAndIncrement();  }  }  }  如上 get方法 和 getAndIncrement方法同时调用不具有原子性，有可能调用get()方法之后 count = 9，另一个线程执行了getAndIncrement 操作 count = 10了,这样导致线程1getAndIncrement 完后 count = 11的错误结果。 |
|  |

### Synchronized效率优化/注意

|  |
| --- |
| * Synchronized锁的粒度越细效率越高，Synchronized代码块中的**代码越少**效率越高 * 避免将锁定的对象的引用变成另外一个对象，对象发生改变，锁定的对象变了。 * 不要以字符串常量作为锁定对象如下：   String s1 = “space”;  String s2 = “space”;  S1和s2为同一个字符串，锁定的也就是同一个对象。 |

### Lock和condition

|  |
| --- |
| Lock也是管程的实现，和synchronized区别如下：    使用实例：  public class Test {  private **final Lock rt = new ReentrantLock(true);//为可重入锁，true表示公平锁**  int count;  public void addOne(){  try {  rt.lock();  count = get() + 1;  }finally {  rt.unlock();  }  }  private int get() {  try {  rt.lock();  return count;  }finally {  rt.unlock();  }  }  public static void main(String[] args) {  Test t = new Test();  for (int i = 0; i < 10; i++) {  new Thread(t::addOne,i+"线程").start();  }  try {  Thread.sleep(1000);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(t.count);  }  }  **Lock rt = new ReentrantLock(true);为可重入锁，true为公平锁，当有其他线程释放锁后，会先通知（等待队列前边的）线程进入，false为随机的。ReentrantLock使用CAS获取锁。**  **Lock的使用范例：**  try {  rt.lock();//加锁  }finally {  rt.unlock();//释放锁  }  **推荐的三个用锁的最佳实践，它们分别是：**   * 永远只在更新对象的成员变量时加锁 * 永远只在访问可变的成员变量时加锁 * 永远不在调用其他对象的方法时加锁，（其他对象方法可能也加锁，或有超时的操作）    tryLock()方法返回true则获取锁成功，有返回值的获取锁，**不进入阻塞状态**。  如下：  boolean flag = false;  try {  while(!(flag = **lock.tryLock()**)) {  System.out.println(Thread.currentThread().getName() + ":" + flag);  }  count++;  System.out.println(Thread.currentThread().getName() + ":" + count);  Thread.sleep(7);  } catch (InterruptedException e) {  e.printStackTrace();  }finally {  if(flag) {  lock.unlock();  }  }  **Condition**：实现了管程里的条件变量。如下实现队列：  public class BlockedQueue<T> {  private MyQueue queue = new MyQueue();  private final Lock lock = new ReentrantLock();  //条件：不满  final Condition notFull = lock.newCondition();  //条件：不空  final Condition notEmpty = lock.newCondition();  //入队  public void enq(T x){  lock.lock();  try {  while (queue.isFull()){  notFull.await();  }  queue.Enqueue();//入队操作  notEmpty.signal();//通知不为空条件等待队列的线程进入入口等待队列（即可以出队操作了）  }finally {  lock.unlock();  }  }  //出队  public void deq(){  lock.lock();  try {  while (queue.isEmpty()){  notEmpty.await();  }  queue.dequeue();//出队操作  notFull.signal();//通知不满条件等待队列的线程进入入口等待队列（即可以入队操作了）  }finally {  lock.unlock();  }  }  Dubbo中rpc的调用，异步转同步就是使用lock&condition实现，当调用rpc方法时，  Rt.await(),方法返回了后调用rt.signal()通知 |

### 读写锁-ReadWriteLock实现缓存

|  |
| --- |
| 理论上使用信号量或管程可以解决所有并发问题。但java并发包还是提供了适合不同场景的并发API并做了相应优化。如：读多写少的场景，缓存。  读写锁三个基本原则：   * 允许多个线程同时读共享变量 * 只允许一个线程写共享变量 * 如果一个写线程正在执行写操作,此时禁止读线程读共享变量。   **读写锁**和**互斥锁**的重要区别：读写锁允许多个线程同时读取共享变量,互斥锁不允许，这样在读多写少的场景读写锁性能**优于互斥锁**。但读写锁的写操作是互斥的。  读写锁实现缓存：如下  private final Map<K,V> map = new HashMap<K, V>();  private ReadWriteLock rw = new ReentrantReadWriteLock();  //读锁  final Lock r = rw.readLock();  //写锁  final Lock w = rw.writeLock();  public V get(K k){  r.lock();  try {  return map.get(k);  }finally {  r.unlock();  }  }  public V put(K k,V v){  w.lock();  try {  return map.put(k,v);  }finally {  w.unlock();  }  }  缓存一般如果数据量小，则一次性put入缓存中，如果数据量大则需要类似延迟加载功能，当缓存中没有数据时去数据库中查找并放入缓存。代码如下：  public V get(K k){  V v = null;  r.lock();  try {  v = map.get(k);  }finally {  r.unlock();  }  //缓存中有则返回  if(v != null){  return v;  }  //没有则,查询数据库  w.lock();  try {  //再次判断，有可能其他线程获取到了write锁并已经写入缓存并释放了锁，当前线程获取锁后需再次验证。  v = map.get(k);  if(v == null){  v = db.get(k);  map.put(k,v);  }  }finally {  w.unlock();  }  return v;  }  如上，再次判断，有可能其他线程获取到了write锁并已经写入缓存并释放了锁，当前线程获取锁后需再次验证。    在读锁还没有释放时又去获取写锁，叫做**锁的升级，但ReadWriteLock不支持，如果这样会导致写锁一直等待。**  **但，在获取写锁后，再获取读锁是允许的，称为：锁的降级，如下：**      ReetrantReadWriteLock读写锁的实现中，需要注意的，当有读锁时，写锁就不能获得；而当有写锁时，除了**获得写锁的这个线程可以获得读锁外，其他线程不能获得读锁。**  读锁可以被多个线程获取，但写锁和读锁互斥，即读锁被获取了，就无法获取到写锁，写锁被获取到了，除了当前线程，其他线程无法获取到读锁。 |

### 读写锁的改进-StampedLock

|  |
| --- |
| StampedLock比ReadWriteLock性能更优，StampedLock提供了三种模式：   * 写锁 * 悲观读锁 * 乐观读：乐观读这个操作是无锁的   写锁、悲观读锁的语义和ReadWriteLock的写锁、读锁的语义非常类似,允许多  个线程同时获取悲观读锁,但是只允许一个线程获取写锁,写锁和悲观读锁是互斥的。不同的是:StampedLock里的写锁和悲观读锁加锁成功之后,都会返回一个stamp;然后解锁的时候,需要传入这个stamp。相关的示例代码如下。  final StampedLock sl = new StampedLock();  // 获取/释放悲观读锁示意代码  long stamp = sl.readLock();  try{  //省略业务相关代码  }finally{  sl.unlockRead(stamp);  }  //获取/释放写锁示意代码  long stamp = sl.writeLock();  try{  //省略业务相关代码  }finally{  sl.unlockWrite(stamp);  }  StampedLock的**性能之所以比**ReadWriteLock还要好,**其关键是StampedLock支持乐观读的方式**。ReadWriteLock支持多个线程同时读,但是当多个线程同时读的时候,所有的写操作会被阻塞;而StampedLock提供的乐观读,是允许一个线程获取写锁的,也就是说不是所有的写操作都被阻塞。  使用乐观读如下：**先使用乐观读，读取共享数据，之后判断是否有其他线程执行写操作，如果有则升级为悲观读锁处理，否则直接返回共享数据。**  **long stamp = sl.tryOptimisticRead()：乐观读，**  **sl.validate(stamp)为false则有写操作。否则直接使用数据（数据正确性和唯一性得到保证）**    在上面这个代码示例中,如果执行乐观读操作的期间,存在写操作,会把乐观读升级为悲观读锁。这个做法挺合理的,否则你就需要在一个循环里反复执行乐观读,直到执行乐观读操作的期间没有写操作(只有这样才能保证x和y的正确性和一致性),而循环读会浪费大量的CPU。升级为悲观读锁,代码简练且不易出错,建议你在具体实践时也采用这样的方法。  **stampedLock注意：**   * 锁不可重入 * StampedLock的悲观读锁、写锁都不支持条件变量 * 如果线程阻塞在StampedLock的readLock()或者writeLock()上时,此时调用该阻塞线程的interrupt()方法中断线程,会导致CPU飙升。     所以,使用StampedLock一定**不要调用中断操作**,如果**需要支持中断**功能,一定使用可中断的**悲观读锁readLockInterruptibly()**和**写锁writeLockInterruptibly()** 。这个规则一定要记清楚。 |

### CountDownLatch和CyclicBarrier

|  |
| --- |
| 如下实例：对账系统：核对订单库 和 派送单 库数据是否有异常。    while(存在未对账订单){  //查询未对账订单  pos = getPOrders();  // 查询派送单  dos = getDOrders();  // 执行对账操作  diff = check(pos, dos);  // 差异写入差异库  save(diff);  }  传统使用单线程效率较低，考虑使用多线程。如下：**让查询订单和派送订单另起线程，使用join让主线程等待两个线程执行完后在进行对账操作**。  while(存在未对账订单){  //查询未对账订单  Thread T1 = new Thread(()->{  pos = getPOrders();  });  T1.start();  //查询派送单  Thread T2 = new Thread(()->{  dos = getDOrders();  });  T2.start();  //等待T1、T2结束  T1.join();  T2.join();  //执行对账操作  diff = check(pos, dos);  //差异写入差异库  save(diff);  }  上边操作存在问题：每循环一次都会new新的线程，太耗费资源。如下优化：   * **用CountDownLatch实现线程等待：类似计数器。** * **使用线程池**       ContDownLatch:**像门栓一样等待初始的值(如：3)为0了继续往下执行，用来 一个线程等待其他线程 的案例。线程结束时进行-1**  如上：两个查订单和 最后的 对账操作还是 串行化的，可以当对账时去并行执行下一次的查询操作，但必须保证两个查询操作一致（一一对应）。这样对账操作类似**消费者**，查询则为：**生产者**。可以将两个消费放入两个队列中    使用三个线程：T1，T2执行查询操作，但两个必须步调一致（先完成的线程需要等待后一个线程，这样保证队列一一对应。），等T1,T2都执行完了去通知T3执行对账操作。    //订单队列  Vector<P> pos;  //派送单队列  Vector<D> dos;//执行回调的线程池  Executor executor =  Executors.newFixedThreadPool(1);  final CyclicBarrier barrier =  new CyclicBarrier(2, () -> {  executor.execute(() -> check());  });  void check() {  P p = pos.remove(0);  D d = dos.remove(0);  //执行对账操作  diff = check(p, d);  //差异写入差异库  save(diff);  }  void checkAll() {  //循环查询订单库  Thread T1 = new Thread(() -> {  while (存在未对账订单) {  //查询订单库  pos.add(getPOrders());  //等待  barrier.await();  }  });  T1.start();  //循环查询运单库  Thread T2 = new Thread(() -> {  while (存在未对账订单) {  //查询运单库  dos.add(getDOrders());  //等待  barrier.await();  }  });  T2.start();  }  如上CyclicBarrier 类会当计数器为0时会回调一个函数（对账操作），为0时会重置计数器为初始值2，上边对账操作线程池大小只有1，因为：如果是多个，可能前一个对账操作还没有完成，订单队列中可能只移除了一个，如果下一个对账操作也去执行，取出的两个查询订单数据不是一一对应的。 |
| CountDownLatch和CyclicBarrier是Java并发包提供的两个非常易用的线程同步工具类,这两个工具类用法的区别在这里还是有必要再强调一下:**CountDownLatch主要用来解决一个线程等待多个线程的场景** ,可以类比旅游团团长要等待所有的游客到齐才能去下一个景点;而**CyclicBarrier是一组线程之间互相等待** ,更像是几个驴友之间不离不弃。除此之外CountDownLatch的计数器是不能循环利用的,也就是说一旦计数器减到0,再有线程调用await(),该线程会直接通过。但CyclicBarrier的计数器是可以循环利用的 ,而且具备自动重置的功能,一旦计数器减到0会自动重置到你设置的初始值。除此之外,CyclicBarrier还可以设置回调函数,可以说是功能丰富。  总结：  CountDownLatch在调用latch.await()时等计数器的值为0后继续执行。  CyclicBarrier会等计数器的值为0时由最后一个执行完的线程调用回调函数（也可以不调），在其他线程中调用await()方法计数器会-1，线程中如果计数器不为0则线程等待其他线程。为0后会重置为初始值。 |

### Phaser

|  |
| --- |
| 阶段性的栅栏。需要继承Phaser实现方法来使用。 |

### Exchanger两个线程交换数据

|  |
| --- |
| 两个线程进行数据交换，T1调用如：如果T1线程先执行了exchanger.exchange(s);  会阻塞等待另一个线程也调用exchange()方法进行数据交换，之后两个线程继续执行。    如下列子：  Exchanger<String> exchanger = new Exchanger<>();  new Thread(new Runnable() {  @Override  public void run() {  String s = "t1";  try {  s = exchanger.exchange(s);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(Thread.currentThread().getName() + ":s = " + s);  }  },"t1").start();  new Thread(new Runnable() {  @Override  public void run() {  String s = "t2";  try {  s = exchanger.exchange(s);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(Thread.currentThread().getName() + ":s = " + s);  }  },"t2").start(); |

### LockSupport

|  |
| --- |
| 控制某个当前线程阻塞，或唤醒指定线程。如下实例：  Thread t1 = new Thread(new Runnable() {  @Override  public void run() {  for (int i = 0; i < 10; i++) {  if(i == 5){  LockSupport.park();  }  System.out.println(Thread.currentThread().getName() + ":" + i);  }  }  });  t1.start();  try {  Thread.sleep(6000);  } catch (InterruptedException e) {  e.printStackTrace();  }  LockSupport.unpark(t1);  Unpark和park就像设置标记（count标记）一样，unpark可以在park方法尚未执行时设置，设置后后边再park时线程不会阻塞，会继续执行。 |

### ThreadLocal

**Spring 声明式事务：保证同一个线程多个方法/数据库操作获取同一个connection连接，这样才能保证这些操作的事务。**

|  |
| --- |
| ThreadLocal可以让通过共享对象ThreadLocal 获取其他线程中存放的数据，原理：  ThreadLocal 中将数据存放在**当前线程** 的Map中，所以不同线程通过ThreadLocal 设置或 获取数据 获取的都是当前线程中存放的数据，这些数据线程间**不共享**  static ThreadLocal<Person> threadLocal = new ThreadLocal<>();  public static void main(String[] args) {  new Thread(()->{  Person p = new Person();  p.setName("ccc");  threadLocal.set(p);  }).start();  new Thread(()->{  try {  Thread.sleep(2000);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(threadLocal.get());  }).start();  threadLocal.remove();//**注意此处，原因在下节**  }  结果打印null,第二个线程没有设置对象，所以在其线程中取不到。  注意：ThreadLocal的坑，如下：[Java引用](#_Java引用) |

### Java引用

引用：变量指向一个对象

|  |
| --- |
| * **强**：普通的引用就是强引用。强引用有指向的对象，则不会被gc回收。 * **软**：当一个软引用指向一个对象，只有当内存不够用时才会被回收。   如下设置jvm： -Xms30m -Xmx30m，如下，当内存不够用时 软引用指向的20m字节数组**被回收**。一般做**缓存**用  public static void main(String[] args) {  SoftReference<byte[]> softReference = new SoftReference<>(new byte[1024\*1024\*20]);  System.out.println(softReference.get());  System.gc();  try {  Thread.sleep(200);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(softReference.get());  byte[] b = new byte[1024\*1024\*15];  System.out.println(softReference.get());  }  打印如下：  [B@1540e19d  [B@1540e19d  null   * **弱**：**只要有gc,发现弱引用就会回收**。作用场景：如果一个强引用也指向弱引用指向的对象，那么当强引用消失后，遇到gc该对象就会被回收。一般在：容器中应用，ThreadLocal中放置的对象是放到当前线程Map中，而map的key为ThreadLocal当前对象（this,value），这里key使用弱引用，（如果使用强引用，由于线程可能长时间执行，即使ThreadLocal tl = ...）中的tl不指向对象了，也不能被回收（**内存泄露**），但用弱引用就可以被回收。     但key指向的对象被回收后，key = null了，但map是当前线程的成员变量，线程没有结束的话，map中不能被访问的对象（k,v）会越来越多也可能导致内存泄露，所以在ThreadLocal里边对象不用后需要remove    WeakReference<Person> weakReference = new WeakReference<>(new Person());//弱引用  System.out.println(weakReference.get());  System.gc();  System.out.println(weakReference.get());  打印结果：  com.space.threadLocal.Person@1540e19d  null   * **虚：有gc就会被回收，而且获取不到里边的对象。写JVM的人用。（管理堆外内存）**   PhantomReference<Person> pf = new PhantomReference<>(new Person(),queue);  虚引用指向Person()对象。  既然获取不到对象，它的作用：当对象被回收后，引用会被放入queue中，作用就是通知 queue 有对象被回收，然后做一些操作。当对象在堆外内存中分配这，这时虚拟机无法回收，使用虚引用，当对象不用时就可以通知queue，然后去清理堆外内存（UnSafe类）。 |

### 并发容器

|  |
| --- |
| Java容器有：list,set,map,要将非线程安全的容器改为线程安全的容器：  **只要把非线程安全的容器封装在对象内部,然后控制好访问路径就可以了。如：**    如上这种包装可以使用Collections来完成如下：  List list = Collections.synchronizedList(new ArrayList());  Set set = Collections.synchronizedSet(new HashSet());  Map map = Collections.synchronizedMap(new HashMap());  在容器领域一个容易被忽视的“坑”是用**迭代器遍历容器。需要加锁**  List list=Collections.  synchronizedList(new ArrayList());  synchronized(list){  Iterator i = list.iterator();  while (i.hasNext())  foo(i.next());  }  jdk1.5之前也就类似上面的实现如：Vector、Stack和Hashtable，这种同步容器性能比较差。Jdk1.5后提供了性能更好的并发容器：     * List:CopyOnWriteArrayList 写时将共享变量拷贝一份，这样做的好处是：读操作无锁。CopyOnWriteArrayList迭代器是**只读**的,不支持**增删改**。因为它遍历的只是一个快照。     如果遍历array时，还有一个写操作（如添加元素），则将array拷贝一份，在新数组上进行添加元素，执行完后将array指向新的数组。    **注意**：   * CopyOnWriteArrayList 只适应与写非常少，并且能容忍读写的短暂不一致（如上面不能遍历到新添加的元素） * CopyOnWriteArrayList迭代器是只读的,不支持增删改。因为它遍历的只是一个快照。 * Map:ConcurrentHashMap和ConcurrentSkipListMap   ConcurrentHashMap的key是**无序的**,而ConcurrentSkipListMap(**跳表实现**)的key是**有序的**     * Set:CopyOnWriteArraySet和ConcurrentSkipListSet,使用如上差不多。 * Queue * 阻塞和非阻塞：当队列已满时,入队操作阻塞；当队列已空时,出队操作阻塞（阻塞队列都用Blocking关键字标识） * 单端与双端 ,单端指的是只能队尾入队,队首出队;而双端指的是队首队尾皆可入队出队   （单端队列使用Queue标识,双端队列使用Deque标识） |

### 容器效率和注意

|  |
| --- |
| 最早容器只有：vector和hashtable（**该两种基本不用**）,这两个每个方法都加synchronized,但由于多数情况都是单线程使用所以有了：hashMap 和ArrayList等非线程安全的容器，后来有了Collections.synchronizedList（） 理论上比hashTable效率高，但也是重量级的。所以就有了ConCurrentHashMap,它的put不一定比上边的两中效率高（因为链表可能要转红黑树），但get（读）的效率比上边的高很多。所以读多写少的并发情况还是使用：ConCurrentHashMap  Queue:有界（基于数组，**常用**）和无界，（基于链表的队列也算是有界的最大int的最大值。）  **阻塞和非阻塞**,比如:非阻塞队列在如果队列满了，无法加入则返回【false/异常】失败，但阻塞队列会线程会阻塞住，直到有线程通知它有位置去添加了。  如：LinkedBlockingQueue无界阻塞队列，中常用方法：   * **offer:**[添加元素，非阻塞的] * **poll:**[取出元素并移除，非阻塞] * **peek:**[取出元素不移除，非阻塞] * **put**:[添加元素，阻塞] * **take**:[取出元素并移除，阻塞]   但非阻塞队列如：ConcurrentLinkedQueue它offer/poll/peek方法会循环去插入或获取值，直到获取到才退出循环（**比较耗费CPU**）。其中给队尾部加/删 元素使用**CAS**方式。  特殊的阻塞队列：   * DelayQueue：延时队列，按照指定时间进行优先级排列取出。（按照排序的顺序取出） 应用于**调度任务执行**。   public class Test {  static BlockingQueue q = new DelayQueue<>();  public static void main(String[] args) throws InterruptedException {  long currentTime = System.currentTimeMillis();  MyTask t1 = new MyTask("t1",currentTime + 1000);  MyTask t2 = new MyTask("t2",currentTime + 2000);  MyTask t3 = new MyTask("t3",currentTime + 1500);  MyTask t4 = new MyTask("t4",currentTime + 2500);  MyTask t5 = new MyTask("t5",currentTime + 500);  q.put(t1);  q.put(t2);  q.put(t3);  q.put(t4);  q.put(t5);  int len = q.size();  for (int i = 0; i < len; i++) {  System.out.println(q.take());  }  }  static class MyTask implements Delayed{  String name;  long timeout;  public MyTask(String name, long timeout) {  this.name = name;  this.timeout = timeout;  }  @Override  public long getDelay(TimeUnit unit) {  return unit.convert(this.timeout - System.currentTimeMillis(),TimeUnit.MILLISECONDS);  }  @Override  public int compareTo(Delayed o) {  if (this.getDelay(TimeUnit.MILLISECONDS) < o.getDelay(TimeUnit.MILLISECONDS)) {  return -1;  }else if (this.getDelay(TimeUnit.MILLISECONDS) > o.getDelay(TimeUnit.MILLISECONDS)){  return 1;  } else {  return 0;  }  }  @Override  public String toString() {  return name + " " + timeout;  }  }  }   * SynchronousQueue：容量为0，put进数据由于长度为0，满了所以阻塞住，直到消费者来消费（take），一个线程。   如下：有些类似Exchanger两个线程交换数据。  BlockingQueue<String> q = new SynchronousQueue<>();  new Thread(()->{  try {  Thread.sleep(1000);  System.out.println(q.take());  } catch (InterruptedException e) {  e.printStackTrace();  }  }).start();  q.put("111");//阻塞等消费者来消费。  System.out.println(q.size());   * TransferQueue：放入之后必须等消费了才能继续去执行后续和SynchronousQueue很像但可以放入多个和多个线程去执行，如：订单队列，需要被处理才能去做后续操作。如下：   LinkedTransferQueue<String> q = new LinkedTransferQueue<>();  new Thread(()->{  try {  System.out.println(q.take());  } catch (InterruptedException e) {  e.printStackTrace();  }  }).start();  q.transfer("1");  System.out.println("1111111");  q.transfer("2"); |

### 线程池

|  |
| --- |
| Callable类似Runnable,但有返回结果的。结果使用Future(存储将来的结果)存储。如下：  Callable<Integer> callable = new Callable<Integer>() {  @Override  public Integer call() throws Exception {  return 1;  }  };  ExecutorService executorService= Executors.newCachedThreadPool();  Future<Integer> future = executorService.submit(callable);//异步执行  try {  System.out.println(future.get());//阻塞方法，知道返回结果  } catch (InterruptedException e) {  e.printStackTrace();  } catch (ExecutionException e) {  e.printStackTrace();  }  executorService.shutdown();  也可以使用FutureTask:(合并Runnable和Future)  FutureTask<Integer> futureTask = new FutureTask<Integer>(()->{  return 11;  });  new Thread(futureTask).start();  System.out.println(futureTask.get());  CompletableFuture（多任务结果管理）：可以执行多个任务，并且类似 多线程门栓一样，等多个任务执行完后再去继续执行或任意一个执行就可以继续执行。并且可以对结果进行多层处理。  **线程池**：一组线程和一个任务队列   * ThreadPoolExecutor：父类为[ExecutorService--->Executor] * ForkJoinPool：分解汇总任务。（很像大数据MR一样）   **自定义线程池：**  ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(2//**核心线程数一直存活的**  ,4//**最大线程数**  ,60//**空闲多少时间后回收线程给操作系统，核心线程不会被回收。**  , TimeUnit.SECONDS//**空闲时间单位**  ,new ArrayBlockingQueue<Runnable>(4)**//任务队列**  , Executors.defaultThreadFactory()//**线程工厂，怎样产生线程的类，可以自定义一个**  ,new ThreadPoolExecutor.DiscardOldestPolicy());//**线程池忙且任务队列满则执行拒绝策略**  拒绝策略默认提供了4种，可以自定义（一般都为自定义）。   * Abort:抛异常 * Discard:扔掉，不抛异常 * DiscardOldest:扔掉排队时间最久的。 * CallerRuns:哪个线程提交的任务由那个线程去执行处理这些任务。     注意：  通过ThreadPoolExecutor对象的execute()方法提交任务时,如果任务在执行的过程中出现运行时异常,会导**致执行任务的线程终止**;不过,最致命的是任务虽然异常了,但是你却获取不到任何通知,这会让你误以为任务都执行得很正常。所以最好捕获：  try {  //业务逻辑  } catch (RuntimeException x) {  //按需处理  } catch (Throwable x) {  //按需处理  }  public class Test {  static class Task implements Runnable{  private int i;   public Task(int i) {  this.i = i;  }   **@Override** public void run() {  System.*out*.println(Thread.*currentThread*().getName() + **"Task "** + this.i);  try {  System.*in*.read();//阻塞住  } catch (IOException e) {  e.printStackTrace();  }  }   **@Override** public String toString() {  return **"Task{"** +  **"i="** + i +  **'}'**;  }  }  public static void main(String[] args) {   ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(2  ,4  ,60  , TimeUnit.*SECONDS* ,new ArrayBlockingQueue<Runnable>(4)  , Executors.*defaultThreadFactory*()  ,new ThreadPoolExecutor.CallerRunsPolicy());   for (int i = 0; i < 8; i++) {  threadPoolExecutor.execute(new Task(i));  }  System.*out*.println(threadPoolExecutor.getQueue());  threadPoolExecutor.execute(new Task(100));//最大线程4，队列能容纳4，所以能处理8个任务，大于8就会按照拒绝策略执行  threadPoolExecutor.shutdown();   } }  JDK常见线程池（不建议使用），Executers(可以看作为：线程池的工厂)   * **SingleThreadExecutor**：单个线程的线程池，任务队列大小为Integer.MAX，可以顺序的执行任务。   ExecutorService executorService = Executors.newSingleThreadExecutor();   * **CachedThreadPool**:0个核心数，Integer.MAX个最大线程数，由于任务队列为SynchronousQueue【大小为一个】所以，如果线程被占用则会创建新线程。   ExecutorService executorService = Executors.newCachedThreadPool();   * **FixedThreadPool：固**定线程数，核心线程数和最大线程数一样。   Executors.newFixedThreadPool(5);   * **ScheduledThreadPool**：定时任务线程池，它的任务队列为DelayedWorkQueue   Executors.newScheduledThreadPool(5);  **以上，底层均有ThreadPoolExecutor实现**   * 任务来了先创建核心线程个数内的 线程去处理任务， * 核心线程都占用的话，新来的任务会放入任务队列中， * 当任务队列满了，再去创建非核心线程去处理任务。 * 非核心线程也达到最大值后，则安装拒绝策略去处理。   **ForkJoinPool型线程池：【jdk提供的如下】**   * **WorkStealingPool：**ForkJoinPool和ThreadPoolExecutor区别，它的每个线程都有一个任务队列。当某个线程执行完了自己的队列后,会去拿（偷）其他线程未执行完的队列中的**队尾**任务加入自己队列尾去执行。     **其中poll要加锁**  **ForkJoinPool:有返回值的形式，和无返回值的，如下：**  public class ForkJoinPoolTest {  static int[] *nums* = new int[100000];  static final int *MAX\_MUM* = 5000;  static {  for (int i = 0; i < *nums*.length; i++) {  *nums*[i] = new Random().nextInt(100);  }  System.*out*.println(Arrays.*stream*(*nums*).sum());  }    */\*\*  \* RecursiveAction:无返回值  \*/* static class AddTask extends RecursiveAction{  private int start;  private int end;  public AddTask(int start, int end) {  this.start = start;  this.end = end;  }  **@Override** protected void compute() {  if(this.end - this.start <= *MAX\_MUM*){  long sum = 0L;  for (int i = start; i < end ; i++) {  sum += *nums*[i];  }  System.*out*.println(**"from: "** + start + **" to: "** + end + **" = "** + sum);  }else {  int middle = start + (end - start)/2;  AddTask task1 = new AddTask(start,middle);  AddTask task2 = new AddTask(middle,end);  task1.fork();  task2.fork();  }  }  }  public static void main(String[] args) throws IOException {  ForkJoinPool forkJoinPool = new ForkJoinPool();  AddTask addTask = new AddTask(0,*nums*.length);  forkJoinPool.execute(addTask);  System.*in*.read();//阻塞住  } }  **有返回值：**  public class ForkJoinPoolTest {  static int[] *nums* = new int[100000];  static final int *MAX\_MUM* = 5000;  static {  for (int i = 0; i < *nums*.length; i++) {  *nums*[i] = new Random().nextInt(100);  }  System.*out*.println(Arrays.*stream*(*nums*).sum());  }  */\*\*  \* RecursiveTask:有返回值  \*/* static class AddTask2 extends RecursiveTask<Long>{  private int start;  private int end;  public AddTask2(int start, int end) {  this.start = start;  this.end = end;  }  **@Override** protected Long compute() {  if(this.end - this.start <= *MAX\_MUM*){  long sum = 0L;  for (int i = start; i < end ; i++) {  sum += *nums*[i];  }  return sum;   }else {  int middle = start + (end - start)/2;  AddTask2 task1 = new AddTask2(start,middle);  AddTask2 task2 = new AddTask2(middle,end);  task1.fork();  task2.fork();  return task1.join() + task2.join();  }  }  }  public static void main(String[] args) throws IOException {  ForkJoinPool forkJoinPool = new ForkJoinPool();  AddTask2 addTask = new AddTask2(0,*nums*.length);  forkJoinPool.execute(addTask);  System.*out*.println(addTask.join());  }  }   * **ParallelStream：并行流式处理，底层也为ForkJoinPool**   public static void main(String[] args) {  List<Integer> nums = new ArrayList<>();  for (int i = 0; i < nums.size(); i++) {  nums.add(new Random().nextInt(100));  }  long start = System.currentTimeMillis();  nums.parallelStream().forEach(ParallelStreamTest::isPrime);//\*\*\*\*\*\*\*\*\*\*\*\*  long end = System.currentTimeMillis();  System.out.println(end - start);  }  static boolean isPrime(int num) {  for(int i=2; i<=num/2; i++) {  if(num % i == 0) return false;  }  return true;  } |

### JMH:代码微基准测试

|  |
| --- |
| **主要是用来JVM上代码性能测试的框架。每秒并发量/每秒执行次数的测试。**  官网http://openjdk.java.net/projects/code-tools/jmh/  **创建JMH测试**   * 创建Maven项目，添加依赖   <dependencies>  <!-- https://mvnrepository.com/artifact/org.openjdk.jmh/jmh-core -->  <dependency>  <groupId>org.openjdk.jmh</groupId>  <artifactId>jmh-core</artifactId>  <version>1.21</version>  </dependency>  <!-- https://mvnrepository.com/artifact/org.openjdk.jmh/jmh-generator-annprocess -->  <dependency>  <groupId>org.openjdk.jmh</groupId>  <artifactId>jmh-generator-annprocess</artifactId>  <version>1.21</version>  <scope>test</scope>  </dependency>   * idea安装JMH插件 JMH plugin v1.0.3   由于用到了注解，打开运行程序注解配置：  compiler -> Annotation Processors -> Enable Annotation Processing   * 定义需要测试类PS (ParallelStream)   package com.mashibing.jmh;  import java.util.ArrayList;  import java.util.List;  import java.util.Random;  public class PS {  static List<Integer> nums = new ArrayList<>();  static {  Random r = new Random();  for (int i = 0; i < 10000; i++) nums.add(1000000 + r.nextInt(1000000));  }  static void foreach() {  nums.forEach(v->isPrime(v));  }  static void parallel() {  nums.parallelStream().forEach(PS::isPrime);  }    static boolean isPrime(int num) {  for(int i=2; i<=num/2; i++) {  if(num % i == 0) return false;  }  return true;  }  }   * 写单元测试，这个测试类一定要在test package下面   package com.mashibing.jmh;  import org.openjdk.jmh.annotations.Benchmark;  import static org.junit.jupiter.api.Assertions.\*;  public class PSTest {  @Benchmark  public void testForEach() {  PS.foreach();  }  }   * 运行测试类，如果遇到下面的错误：   ERROR: org.openjdk.jmh.runner.RunnerException: ERROR: Exception while trying to acquire the JMH lock (C:\WINDOWS\/jmh.lock): C:\WINDOWS\jmh.lock (拒绝访问。), exiting. Use -Djmh.ignoreLock=true to forcefully continue.  at org.openjdk.jmh.runner.Runner.run(Runner.java:216)  at org.openjdk.jmh.Main.main(Main.java:71)  这个错误是因为JMH运行需要访问系统的TMP目录，解决办法是：  打开RunConfiguration -> Environment Variables -> include system environment viables   * 阅读测试报告   **JMH中的基本概念**   * Warmup 预热，由于JVM中对于特定代码会存在优化（本地化），预热对于测试结果很重要 * Mesurement 总共执行多少次测试 * Timeout * Threads 线程数，由fork指定 * Benchmark mode 基准测试的模式 * Benchmark 测试哪一段代码   管网实例：  http://hg.openjdk.java.net/code-tools/jmh/file/tip/jmh-samples/src/main/java/org/openjdk/jmh/samples/ |

### Disruptor

|  |
| --- |
| Disruptor:瓦解，分裂。单线程每秒/600万订单。单机最快的MQ(消息队列)，单机支持高并发。只能存储在内存中的一个队列。  **无锁/CAS方式，高并发，环形缓存，用于生产者-消费者模式。**    **由于计算如：存12个数 环形队列长度为8，则第12个数存在：12%8=4=num & (size -1)，第4个（index:3）,所以队列的长度最后为2的幂次方，方便计算。**  只需维护一个指针，并且直接覆盖原有数据，遍历查询快于链表形式的队列。效率更高。  过程如下：  生产者往队列中加入，当加满了队列，如果0位置还没被消费【是否被消费有标记】，则有多种策略【如：生产者会阻塞等待】，直到消费者消费了后 唤醒生产者。  **Disruptor为什么那么高效**：  1. 内存分配更加合理,使用RingBuffer数据结构,数组元素在初始化时一次性全部创建,提升缓存命中率;对象循环利用,避免频繁GC。  2. 能够避免伪共享,提升缓存利用率。  3. 采用无锁算法,避免频繁加锁、解锁的性能消耗。  4. 支持批量消费,消费者可以无锁方式消费多个消息。  消费方式：  handleEventsWithWorkerPool就可以完成**不重复消费**  使用handleEventsWith就是**重复消费**  **Disruptor开发步骤：**   * **定义Event - 队列中需要处理的元素** * **定义Event工厂，用于填充队列**   这里牵扯到效率问题：disruptor初始化的时候，会调用Event工厂，对环形队列（ringBuffer）进行内存的提前分配【提前new好，将属性值赋值就行，new的对象就少】，GC产频率会降低   * **定义EventHandler（消费者），处理容器中的元素**   如下：  public class LongEvent {  private long value;  public void setValue(long value){  this.value = value;  }  @Override  public String toString() {  return "LongEvent{" +  "value=" + value +  '}';  }  }  public class LongEventFactory implements EventFactory<LongEvent> {  public LongEvent newInstance() {  return new LongEvent();  }  }  public class LongEventHandler implements EventHandler<LongEvent> {  private static long count = 0;  /\*\*  \*  \* @param longEvent  \* @param sequence  \* @param b:是否为最后一个元素  \* @throws Exception  \*/  public void onEvent(LongEvent longEvent, long sequence, boolean b) throws Exception {  count ++;  System.out.println("[" + Thread.currentThread().getName() + "]" + longEvent + " 序号：" + sequence);  }  }  一般的使用：  LongEventFactory eventFactory = new LongEventFactory();//event工厂  int size = 1024;//大小  Disruptor<LongEvent> disruptor =  new Disruptor<LongEvent>(eventFactory,size, Executors.defaultThreadFactory());  disruptor.handleEventsWith(new LongEventHandler());  disruptor.start();  RingBuffer<LongEvent> ringBuffer = disruptor.getRingBuffer();  long sequence = ringBuffer.next();//获取放入的序号  try {  LongEvent longEvent = ringBuffer.get(sequence);  longEvent.setValue(8888L);//设置值  }finally {  ringBuffer.publish(sequence);//提交  }  为java8写法做准备：  LongEventFactory eventFactory = new LongEventFactory();  int size = 1024;  Disruptor<LongEvent> disruptor =  new Disruptor<LongEvent>(eventFactory,size, DaemonThreadFactory.INSTANCE);  disruptor.handleEventsWith(new LongEventHandler());  disruptor.start();  RingBuffer<LongEvent> ringBuffer = disruptor.getRingBuffer();  //无参数,EventTranslator为java8 lamda表达式做准备  EventTranslator<LongEvent> eventTranslator = new EventTranslator<LongEvent>() {  public void translateTo(LongEvent longEvent, long l) {  longEvent.setValue(888L);  }  };  ringBuffer.publishEvent(eventTranslator);  //一个参数  EventTranslatorOneArg<LongEvent,Long> eventTranslatorOneArg =  new EventTranslatorOneArg<LongEvent, Long>() {  public void translateTo(LongEvent longEvent, long l, Long aLong) {  longEvent.setValue(aLong);  }  };  ringBuffer.publishEvent(eventTranslatorOneArg,888L);  //两个参数  EventTranslatorTwoArg<LongEvent,Long,Long> eventTranslatorTwoArg =  new EventTranslatorTwoArg<LongEvent, Long, Long>() {  public void translateTo(LongEvent longEvent, long l, Long aLong, Long aLong2) {  longEvent.setValue(aLong + aLong2);  }  };  ringBuffer.publishEvent(eventTranslatorTwoArg,222L,444L);  //多个参数  EventTranslatorVararg<LongEvent> eventTranslatorVararg =  new EventTranslatorVararg<LongEvent>() {  public void translateTo(LongEvent longEvent, long l, Object... objects) {  long result = 0L;  for (int i = 0; i < objects.length; i++) {  long temp = (Long)objects[i];  result += temp;  }  longEvent.setValue(result);  }  };  ringBuffer.publishEvent(eventTranslatorVararg,1L,2L,3L,4L);  Java8写法：  int size = 1024;  Disruptor<LongEvent> disruptor =  new Disruptor<LongEvent>(LongEvent::new,size, DaemonThreadFactory.INSTANCE);  disruptor.handleEventsWith((event,sequence,endOfBatch) -> System.out.println("Event:" + event));  disruptor.start();  RingBuffer<LongEvent> ringBuffer = disruptor.getRingBuffer();  ringBuffer.publishEvent((event,sequence,l) -> event.setValue(l),88L);  当指定生产者模式为单线程模式，则注意：此时sequence不加锁，那么当有多个线程消费时就可能将前一个线程刚写入的值覆盖掉。当确定生产者和消费者都为单线程时，ProducerType.SINGLE效率会更高（不加锁）。  Disruptor<LongEvent> disruptor =  new Disruptor<LongEvent>(LongEvent::new,size, DaemonThreadFactory.INSTANCE,  ProducerType.SINGLE,new BlockingWaitStrategy());  **当环形buffer满了，生产者使用哪些策略**：   * (常用）BlockingWaitStrategy：通过线程阻塞的方式，等待生产者唤醒，被唤醒后，再循环检查依赖的sequence是否已经消费。 * BusySpinWaitStrategy：线程一直自旋等待，可能比较耗cpu * LiteBlockingWaitStrategy：线程阻塞等待生产者唤醒，与BlockingWaitStrategy相比，区别在signalNeeded.getAndSet,如果两个线程同时访问一个访问waitfor,一个访问signalAll时，可以减少lock加锁次数 * LiteTimeoutBlockingWaitStrategy：与LiteBlockingWaitStrategy相比，设置了阻塞时间，超过时间后抛异常。 * PhasedBackoffWaitStrategy：根据时间参数和传入的等待策略来决定使用哪种等待策略 * TimeoutBlockingWaitStrategy：相对于BlockingWaitStrategy来说，设置了等待时间，超过后抛异常 * （常用）YieldingWaitStrategy：尝试100次，然后Thread.yield()让出cpu * （常用）SleepingWaitStrategy : sleep |

## 并发编程-bug源头

### 可见性-缓存带来的问题

|  |
| --- |
| 最初的单核CPU,由于只有一个核心，cpu缓存也就只有一个，所以数据的对各个线程都是可见的，不存在问题，一个线程修改了，其他线程读取的是同一缓存的数据。如下：    一个线程对共享变量的修改，另外一个线程能够立刻看到，我们称为**可见性**。  多核时代：每个核心都有自己的缓存 |

### 原子性-线程切换

|  |
| --- |
| **线程切换：**    **我们把一个或者多个操作在CPU执行的过程中不被中断的特性称为原子性**  如：count+=1 能需要三条cpu指令。如：   * 指令1：首先，需要把变量count从内存加载到CPU的寄存器； * 指令2：之后，在寄存器中执行+1操作； * 指令3：最后，将结果写入内存（缓存机制导致可能写入的是CPU缓存而不是内存）。   如果是在指令1后切换到 其他线程 可能导致数据不正确    CPU能保证的原子操作是CPU指令级别的，而不是高级语言的操作符 |

### 编译器优化-有序性问题

|  |
| --- |
| 编译器优化可能带来有序性问题。  如：  a = 7;  b = 8;  优化后可能先执行 b = 8 后执行 a = 7;  这样就可能出现问题，如下单例写法问题：  public class Singleton {  static Singleton instance;  static Singleton getInstance(){  if (instance == null) {  synchronized(Singleton.class) {  if (instance == null)  instance = new Singleton();  }  }  return instance;  }  }  由于编译器优化，我们以为的new操作应该是：   1. 分配一块内存M； 2. 在内存M上初始化Singleton对象； 3. 然后M的地址赋值给instance变量。   但是实际上优化后的执行路径却是这样的：   1. 分配一块内存M； 2. 将M的地址赋值给instance变量； 3. 最后在内存M上初始化Singleton对象。   当线程A在执行到 2，给instance 赋了值后发生线程切换，B线程调用 getInstance()方法，判断instance ！ = null,就去返回了对象，但其实对象 并没有初始化完成，就会出现问题（如：空指针） |

### 如何解决并发bug

**方法包括：volatile**、**synchronized** 和 **final** 三个关键字，以及六项 **Happens-Before 规则**

### 可见性和有序性的解决

|  |
| --- |
| 可见性可以使用 volatile修饰来使数据被各个线程所见。如下：  // 以下代码来源于【参考1】  class VolatileExample {  int x = 0;  volatile boolean v = false;  public void writer() {  x = 42;  v = true;  }  public void reader() {  if (v == true) {  // 这里x会是多少呢？  }  }  }  分别让两个线程去执行 writer和 reader,reader方法 中 当 v==true时，x会是多少呢？  Jdk1.5之前 x可能是0 也可能是 42  0是因为 线程2使用的本地存储/缓存 中的副本，也可能副本已被跟新所以为42  但在jdk1.5之后 x = 42,1.5之后增加了Happens-Before 规则  Happens-Before 规则：**前面一个操作的结果对后续操作是可见的** 程序的顺序性规则 如下：  X = 42;  V = true;  X = 42；的操作对v = true；操作是可见的 volatile变量规则 一个volatile变量的写操作， Happens-Before 于后续对这个volatile变量的读操作，单一这条规则这和禁用缓存很像 传递性 如果 **A** Happens-Before **B ,B** Happens-Before **C 那么：A** Happens-Before **C**  如上边的代码：  因为 writer中 x=42 Happens-Before v = true;又因为 规则2  V = true的写 Happens-Before reader方法中 读 v == true,所以  **X = 42 Happens-Before v == true;即：此处 x = 42是可见的**   管程中锁的规则 这条规则是指对一个锁的解锁 Happens-Before 于后续对这个锁的加锁。  **管程**是一种通用的同步原语，在Java中指的就是synchronized,synchronized是Java里对管程的实现。  如：  synchronized (this) { //此处自动加锁  // x是共享变量,初始值=10  if (this.x < 12) {  this.x = 12;  }  } //此处自动解锁  意思在线程1 对共享变量x设置为12后，线程2在进入代码块时，能够看到线程1对x的修改，即可以看到x = 12 线程 start() 规则 这条是关于线程启动的。它是指主线程A启动子线程B后，子线程B能够看到主线程在启动子**线程B前的操作**。  换句话说就是，如果线程A调用线程B的 start() 方法（即在线程A中启动线程B），那么该start()操作 Happens-Before 于线程B中的任意操作。具体可参考下面示例代码。  Thread B = new Thread(()->{  // 主线程调用B.start()之前  // 所有对共享变量的修改，此处皆可见  // 此例中，var==77  });  // 此处对共享变量var修改  var = 77;  // 主线程启动子线程  B.start();  主线程中调用了B.start()，所以 在线程B中可以获得 var = 77这个值 线程 join() 规则 这条是关于线程等待的。它是指主线程A等待子线程B完成（主线程A通过调用子线程B的join()方法实现），当子线程B完成后（主线程A中join()方法返回），主线程能够看到子线程的操作。当然所谓的“看到”，指的是对**共享变量**的操作。  Thread B = new Thread(()->{  // 此处对共享变量var修改  var = 66;  });  // 例如此处对共享变量修改，  // 则这个修改结果对线程B可见  // 主线程启动子线程  B.start();  B.join()  // 子线程所有对共享变量的修改  // 在主线程调用B.join()之后皆可见  // 此例中，var==66  主线程A 调用 join等待子线程B,如果join操作正常返回，那么线程B中的操作 在join返回后的主线程中可见，即：共享变量数据可见  在Java语言里面，Happens-Before的语义本质上是一种可见性，A Happens-Before B 意味着A事件对B事件来说是可见的，无论A事件和B事件是否发生在同一个线程里。例如A事件发生在线程1上，B事件发生在线程2上，Happens-Before规则保证线程2上也能看到A事件的发生。 |

### 原子性的解决

|  |
| --- |
| 原子性问题是由于 **线程切换导致 解决方法：同一时间只有一个线程在执行**  “**同一时刻只有一个线程执行**”这个条件非常重要，我们称之为**互斥**。  如果我们能够保证对共享变量的修改是互斥的，那么，无论是单核CPU还是多核CPU，就都能保证原子性了。    我们把一段需要互斥执行的代码称为**临界区** Java语言提供的锁技术：synchronized 当修饰静态方法的时候，锁定的是当前类的Class对象，在上面的例子中就是Class X； 当修饰非静态方法的时候，锁定的是当前实例对象this。  **如下:**  class SafeCalc {  long value = 0L;  long get() {  return value;  }  synchronized void addOne() {  value += 1;  }  }  两个线程分别执行 addOne和 get方法，由于 happens-before原则：管程中的锁规则  在其他线程调用 addOne方法 value是可见的，但get不可见，所以也需要加锁。 锁和受保护资源的关系 **受保护资源和锁之间的关联关系是N:1的关系**。  一个资源不能被多把锁保护  如下会出现并发问题：  class SafeCalc {  static long value = 0L;  synchronized long get() {  return value;  }  synchronized static void addOne() {  value += 1;  }  }  没有关联的资源：  **用不同的锁对受保护资源进行精细化管理，能够提升性能**。这种锁还有个名字，叫**细粒度锁**。  class Account {  // 锁：保护账户余额  private final Object balLock = new Object();  // 账户余额  private Integer balance;  // 锁：保护账户密码  private final Object pwLock = new Object();  // 账户密码  private String password;  // 取款  void withdraw(Integer amt) {  synchronized(balLock) {  if (this.balance > amt){  this.balance -= amt;  }  }  }  // 查看余额  Integer getBalance() {  synchronized(balLock) {  return balance;  }  }  // 更改密码  void updatePassword(String pw){  synchronized(pwLock) {  this.password = pw;  }  }  // 查看密码  String getPassword() {  synchronized(pwLock) {  return password;  }  }  } 保护有关联关系的多个资源 如下会出现编发问题：转账问题  class Account {  private int balance;  // 转账  synchronized void transfer(  Account target, int amt){  if (this.balance > amt) {  this.balance -= amt;  target.balance += amt;  }  }  }  如果有如下转账：线程1执行：A转B 100，线程2：B转C 100,初始化都为200  则：线程1锁定的对象为A.this,线程2锁定的对象为B.this,这样不能保证并发问题    **用Account.class作为共享的锁**。  class Account {  private int balance;  // 转账  void transfer(Account target, int amt){  synchronized(Account.class) {  if (this.balance > amt) {  this.balance -= amt;  target.balance += amt;  }  }  }  }    **解决原子性问题，是要保证中间状态对外不见** |

### 编发编程需要注意的问题

|  |
| --- |
| 需注意一下问题：   * **安全性问题：正确性，程序按照我们预期执行。**   当有多个线程会同时读写同一数据，存在数据共享并且数据状态不发生改变。  使用锁来解决。   * **活跃性问题：指操作无法执行下去，如：死锁，活锁，饥饿**   活锁：如：两个人一直互相谦让。解决办法：谦让时，尝试等待一个随机的时间就可以了  饥饿：线程因无法访问所需资源,如：线程的优先级不均，可能导致优先级低的线程无法别执行，持有锁的线程，如果执行的时间过长，也可能导致“饥饿”问题。  解决“饥饿”问题的方案很简单，有三种方案：一是保证资源充足，二是**公平地分配资源**，三就是避免持有锁的线程长时间执行。这三个方案中，方案一和方案三的适用场景比较有限，因为很多场景下，资源的稀缺性是没办法解决的，持有锁的线程执行的时间也很难缩短。倒是方案二的适用场景相对来说更多一些。  那如何公平地分配资源呢？在并发编程里，主要是使用**公平锁**。所谓公平锁，是一种先来后到的方案，线程的等待是有顺序的，排在等待队列前面的线程会优先获得资源。   * **性能问题：**   第一，既然使用锁会带来性能问题，那最好的方案自然就是**使用无锁的算法和数据结构**了。在这方面有很多相关的技术，例如线程本地存储(Thread Local Storage, TLS)、写入时复制(Copy-on-write)、乐观锁等；Java并发包里面的原子类也是一种无锁的数据结构；Disruptor则是一个无锁的内存队列，性能都非常好……  第二，**减少锁持有的时间**。互斥锁本质上是将并行的程序串行化，所以要增加并行度，一定要减少持有锁的时间。这个方案具体的实现技术也有很多，例如使用细粒度的锁，一个典型的例子就是Java并发包里的ConcurrentHashMap，它使用了所谓分段锁的技术（这个技术后面我们会详细介绍）；还可以使用读写锁，也就是读是无锁的，只有写的时候才会互斥。  性能方面的度量指标有很多，我觉得有三个指标非常重要，就是：吞吐量、延迟和并发量。  吞吐量：指的是单位时间内能处理的请求数量。吞吐量越高，说明性能越好。  延迟：指的是从发出请求到收到响应的时间。延迟越小，说明性能越好。  并发量：指的是能同时处理的请求数量，一般来说随着并发量的增加、延迟也会增加。所以延迟这个指标，一般都会是基于并发量来说的。例如并发量是1000的时候，延迟是50毫秒。 |
| 总结 并发编程是一个复杂的技术领域，微观上涉及到原子性问题、可见性问题和有序性问题，宏观则表现为安全性、活跃性以及性能问题。 |

### 解决并发的万能钥匙-管程

|  |
| --- |
| 管程：**指的是管理共享变量以及对共享变量的操作过程，让他们支持并发**  翻译为Java领域的语言，就是管理类的成员变量和成员方法，让这个类是线程安全的。  Java管程模型图：    Synchronized实现的管程只支持一个条件变量，如Synchronized(this),this就是这个条件变量。  但lock和condition实现的管程可以有多个条件变量。  条件变量满足才能去执行对共享变量的操作（方法），否则需要在该条件变量对象的等待队列中等待。可以定义一个对象如：A来表示一个条件变量，那么T1线程A.wait()则会去等待队列中等待，当其T2线程A.notify()通知条件变量中的T1条件满足了，T1会重新进入入口等待队列中等待进入管程，但可能这是A条件已经不满足了,所以需要while循环验证条件变量。  如上实现非阻塞队列：  public class BlockedQueue<T> {  private MyQueue queue = new MyQueue();  private final Lock lock = new ReentrantLock();  //条件：不满  final Condition notFull = lock.newCondition();  //条件：不空  final Condition notEmpty = lock.newCondition();  //入队  public void enq(T x){  lock.lock();  try {  while (queue.isFull()){  notFull.await();  }  queue.Enqueue();//入队操作  notEmpty.signal();//通知不为空条件等待队列的线程进入入口等待队列（即可以出队操作了）  }finally {  lock.unlock();  }  }  //出队  public void deq(){  lock.lock();  try {  while (queue.isEmpty()){  notEmpty.await();  }  queue.dequeue();//出队操作  notFull.signal();//通知不满条件等待队列的线程进入入口等待队列（即可以入队操作了）  }finally {  lock.unlock();  }  } |

### 信号量

|  |
| --- |
| 知道管程出现之前，**信号量**一直是并发编程的**终结者**。  信号量模型：一个计数器，一个等待队列，三个方法。计数器和等待队列是透明的（看不到），只能通过三个方法访问。Init(),down(),up()     * Init()：设置计数器的初始值。 * down():计数器减1；如果计数器的值小于0，则当前线程被阻塞，否则当前线程继续执行。 * up()：计数器的值加1;如果此时计数器的值小于或者等于0,则唤醒等待队列中的一个线程,并将其从等待队列中移除   **这三个方法都是原子性的**。Java中信号量模型由java.util.concurrent.Semaphore实现。  上边模型如下边代码：    在Java SDK并发包里,down()和up()对应的则是acquire()和release()。  如下使用信号量模型实例：  public class SemaphoreTest {  static int *count*;  static final Semaphore *sh* = new Semaphore(1);  public static void addOne(){  try {  *sh*.acquire();  } catch (InterruptedException e) {  e.printStackTrace();  }  try {  *count* += 1;  }finally {  *sh*.release();  }  }  public static void main(String[] args) {  for (int i = 0; i < 100; i++) {  new Thread(SemaphoreTest::*addOne*,i + **"线程"**).start();  }  try {  Thread.*sleep*(1000);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println(*count*);   }   }  如上执行过程：线程T1和线程T2,同时执行addOne()方法，由于acquire()是原子操作，如T1先进入，计数器-1=0，由于不小于0所以当前线程继续执行，T1执行acquire()完后，T2执行，计数器-1=-1，小于0，T2线程阻塞。T1执行完count+=1操作后，执行  Release()方法，计数器+1=0，等于0唤醒等待队列中的T2,并将其从等待队列中移除，T2也进行count+=1操作。**于是T2在T1执行完临界区代码之后才获得了进入临界区执行的机会,从而保证了互斥性**。  上面的例子,我们用信号量实现了一个最简单的互斥锁功能。估计你会觉得奇怪,既然有Java SDK里面提供了Lock,为啥还要提供一个Semaphore ?其实实现一个互斥锁,仅仅是Semaphore的部分功能,Semaphore还有一个功能是Lock不容易实现的,那就是:**Semaphore可以允许多个线程访问一个临界区** 。（如：连接池）  信号量的计数器,在上面的例子中,我们设置成了1,**这个1表示只允许一个线程进入临界区**,但如果我们把计数器的值设置成对象池里对象的个数N,就能完美解决对象池的限流问题了。  如下：  public class ObjPool<T,R> {  final List<T> pool;  // 用信号量实现限流器  final Semaphore sem;  // 构造函数  public ObjPool(int size,T t){  pool = new Vector<T>(){};  for(int i=0;i<size;i++){  pool.add(t);  }  sem = new Semaphore(size);  }  // 利用对象池的对象,调用func  public R exec(Function<T,R> func){  T t = null;  try {  sem.acquire();  } catch (InterruptedException e){  e.printStackTrace();  }  try{  t = pool.remove(0);  return func.apply(t);  }finally{  pool.add(t);  sem.release();  }  }  public static void main(String[] args) {  ObjPool<Long,String> pool = new ObjPool<Long,String>(10,2l);  //通过对象池获取t,之后执行  pool.exec(t -> {  System.out.println(t);  return t.toString();  });    }  } |

### 衡量多线程性能（多少线程合适）

|  |
| --- |
| 多线程的性能衡量：延迟和吞吐量  多核时代，当cup和IO的利用率不高时，就可以通过多线程来提高性能。   * **I/O密集型计算**：I/O操作执行的时间相对于CPU计算来说都非常长   大体可以如公式：最佳线程数=CPU核数 \* [ 1 +（I/O耗时 / CPU耗时）]   * **Cpu密集型计算**：CPU执行的时间相对于I/O操作来说都非常长   对于CPU密集型的计算场景，理论上“线程的数量=CPU核数”就是最合适的。不过在工程上，线程的数量一般会设置为“CPU核数+1” |

## Java内存模型

|  |
| --- |
| Jmm是一个抽象的逻辑模型，对应jvm的实现为：主要为堆（主内存），栈（工作内存）    局部变量（基本类型（int,boolean...）,引用类型（引用））存储在栈内存中。而成员变量会随对象本身存储在堆中。  如上对应硬件：主内存对应内存，工作内存（**缓存、写缓冲区、寄存器以及其他的硬件和编译器优化**） |