1、什么是JUC

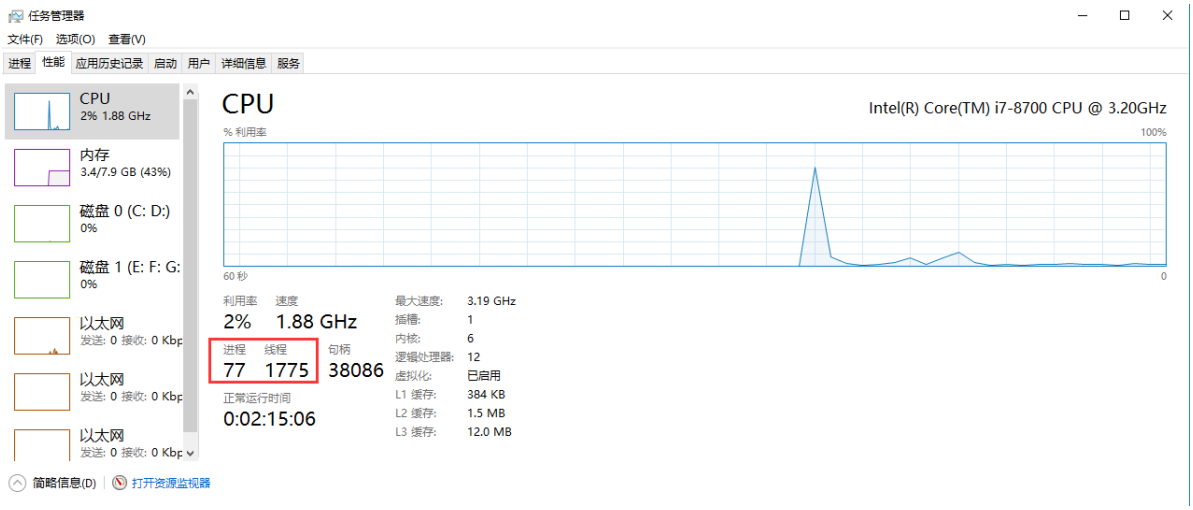
java.util.concurrent 包是在并发编程中使用的工具类，有以下三个包：

2、进程和线程回顾

> 进程 / 线程是什么?

进程：进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。它是操作系统动态执行的基本单元，在传统的操作系统中，进程既是基本的分配单元，也是基本的执行单元。

线程：通常在一个进程中可以包含若干个线程，当然一个进程中至少有一个线程，不然没有存在的意义，线程可以利用进程所有拥有的资源。在引入线程的操作系统中，通常都是把进程作为分配资源的基本单位，而把线程作为独立运行和独立调度的基本单位，由于线程比进程小，基本上不拥有系统资源，故对它的调度所付出的开销就会小得多，能更高效的提高系统多个程序间并发执行的程度。



白话：

进程：就是操作系统中运行的一个程序，QQ.exe，music.exe， word.exe ，这就是多个进程

线程：每个进程中都存在一个或者多个线程，比如用word写文章时，就会有一个线程默默帮你定时自动保存。

学习一些枯燥的概念的时候，一定要与生活中的例子结合起来，这样记忆才是最方便的。

并发 / 并行是什么?

做并发编程之前，必须首先理解什么是并发，什么是并行。

并发和并行是两个非常容易混淆的概念。它们都可以表示两个或多个任务一起执行，但是偏重点有点不同。并发偏重于多个任务交替执行，而多个任务之间有可能还是串行的。并发是逻辑上的同时发生（simultaneous），而并行是物理上的同时发生。然而并行的偏重点在于"同时执行"。

严格意义上来说，并行的多个任务是真实的同时执行，而对于并发来说，这个过程只是交替的，一会运行任务一,一会儿又运行任务二，系统会不停地在两者间切换。但对于外部观察者来说，即使多个任务是串行并发的，也会造成是多个任务并行执行的错觉。

实际上，如果系统内只有一个CPU，而现在而使用多线程或者多线程任务，那么真实环境中这些任务不可能真实并行的，毕竟一个CPU一次只能执行一条指令，这种情况下多线程或者多线程任务就是并发的，而不是并行，操作系统会不停的切换任务。真正的并发也只能够出现在拥有多个CPU的系统中（多核CPU）。

并发的动机：在计算能力恒定的情况下处理更多的任务, 就像我们的大脑, 计算能力相对恒定, 要在一天中处理更多的问题, 我们就必须具备多任务的能力. 现实工作中有很多事情可能会中断你的当前任务, 处理这种多任务的能力就是你的并发能力。

并行的动机：用更多的CPU核心更快的完成任务. 就像一个团队, 一个脑袋不够用了, 一个团队来一起处理一个任务。

例子：

你吃饭吃到一半，电话来了，你一直到吃完了以后才去接，这就说明你不支持并发也不支持并行。

你吃饭吃到一半，电话来了，你停了下来接了电话，接完后继续吃饭，这说明你支持并发。 （不一定是同时的）

你吃饭吃到一半，电话来了，你一边打电话一边吃饭，这说明你支持并行。

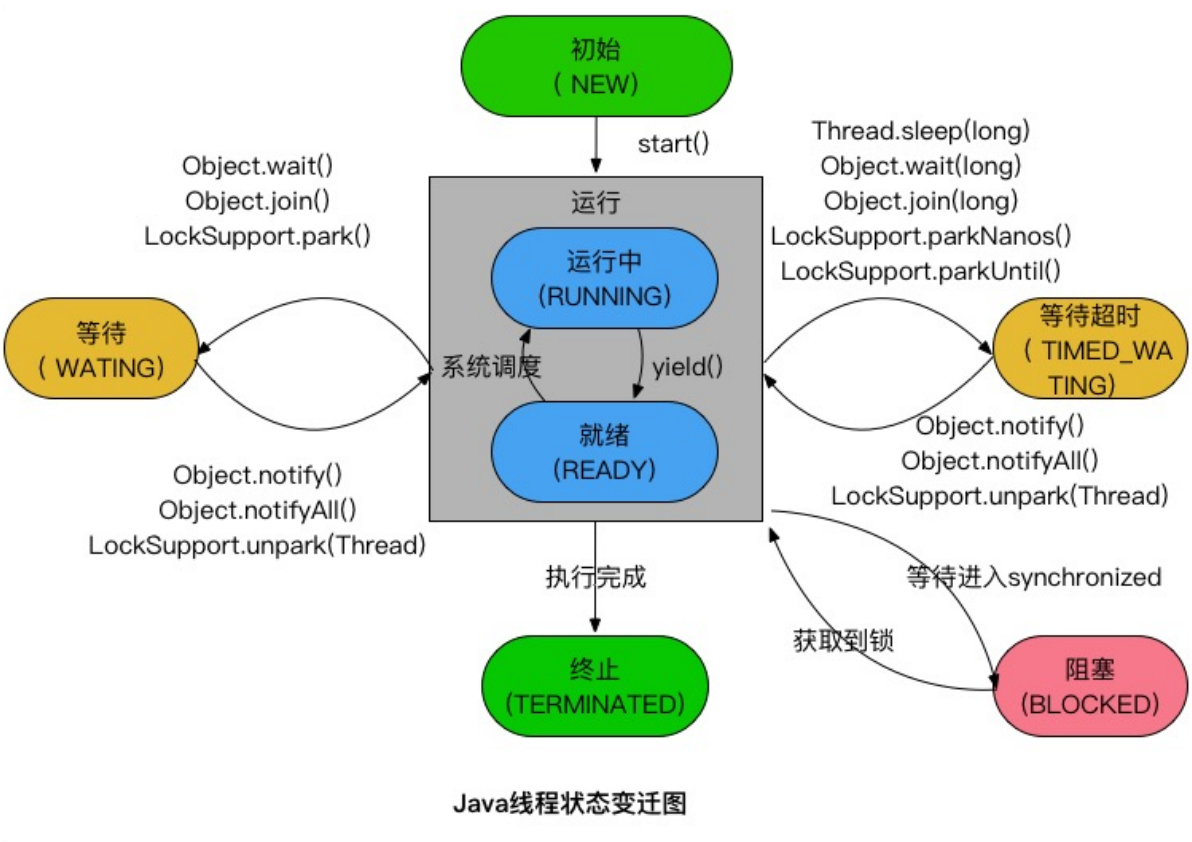
所以并发编程的目标是充分的利用处理器的每一个核，以达到最高的处理性能。

线程的状态

Java的线程有6种状态：可以分析源码：

```
public enum State {
    //线程刚创建
    NEW,

    //在JVM中正在运行的线程
    RUNNABLE,

    //线程处于阻塞状态，等待监视锁，可以重新进行同步代码块中执行
    BLOCKED,

    //等待状态
```

```
    WAITING,

    //调用sleep() join() wait()方法可能导致线程处于等待状态
    TIMED_WAITING,

    //线程执行完毕，已经退出
    TERMINATED;
}
```



Java线程状态变迁图

> wait / sleep 的区别

1、来自不同的类

这两个方法来自不同的类分别是，sleep来自Thread类，和wait来自Object类。

sleep是Thread的静态类方法，谁调用的谁去睡觉，即使在a线程里调用了b的sleep方法，实际上还是a去睡觉，要让b线程睡觉要在b的代码中调用sleep。

2、有没有释放锁(释放资源)

最主要是sleep方法没有释放锁，而wait方法释放了锁，使得其他线程可以使用同步控制块或者方法。

sleep是线程被调用时，占着cpu去睡觉，其他线程不能占用cpu，os认为该线程正在工作，不会让出系统资源，wait是进入等待池等待，让出系统资源，其他线程可以占用cpu。

sleep(100L)是占用cpu，线程休眠100毫秒，其他进程不能再占用cpu资源，wait（100L）是进入等待池中等待，交出cpu等系统资源供其他进程使用，在这100毫秒中，该线程可以被其他线程notify，但不同的是其他在等待池中的线程不被notify不会出来，但这个线程在等待100毫秒后会自动进入就绪队列等待系统分配资源，换句话说，sleep（100）在100毫秒后肯定会运行，但wait在100毫秒后还有等待os调用分配资源，所以wait100的停止运行时间是不确定的，但至少是100毫秒。

就是说sleep有时间限制的就像闹钟一样到时候就叫了，而wait是无限期的除非用户主动notify。

3、使用范围不同

wait，notify和notifyAll只能在同步控制方法或者同步控制块里面使用，而sleep可以在任何地方使用

```
synchronized(x){
    //或者wait()
    x.notify()
}
```

4、是否需要捕获异常

sleep必须捕获异常，而wait，notify和notifyAll不需要捕获异常。

3、Lock锁

> 传统的 synchronized

```
/*
 * 题目：三个售票员 卖出 30张票
 * 多线程编程的企业级套路：
 * 1. 在高内聚低耦合的前提下， 线程  操作(对外暴露的调用方法)  资源类
 */
public class SaleTicketTest1 {
    public static void main(String[] args) throws Exception { // main一切程序的入口

        Ticket ticket = new Ticket();

        new Thread(new Runnable() {
            @Override
            public void run() {
                for (int i = 1; i <=40; i++) {
                    ticket.saleTicket();
                }
            }
        }, "A").start();

        new Thread(new Runnable() {
            @Override
            public void run() {
                for (int i = 1; i <=40; i++) {
                    ticket.saleTicket();
                }
            }
        }, "B").start();

        new Thread(new Runnable() {
            @Override
            public void run() {
                for (int i = 1; i <=40; i++) {
                    ticket.saleTicket();
                }
            }
```

```
        }, "C").start();
    }
}

class Ticket{ // 资源类
    private int number = 30;

    public synchronized void saleTicket(){
        if (number>0){
            System.out.println(Thread.currentThread().getName() + "卖出第 " + (number--) + "票,还剩下:"+number);
        }
    }
}
```

使用 juc.locks 包下的类操作 Lock 锁 + Lambda 表达式


```
public class SaleTicketTest2 {
    public static void main(String[] args) throws Exception { // main一切程序的入口

        Ticket ticket = new Ticket();

        new Thread(()->{for (int i = 1; i <=40; i++) ticket.saleTicket();}, "A").start();
        new Thread(()->{for (int i = 1; i <=40; i++) ticket.saleTicket();}, "B").start();
        new Thread(()->{for (int i = 1; i <=40; i++) ticket.saleTicket();}, "C").start();

    }
```

```
}

class Ticket{ // 资源类

    private Lock lock = new ReentrantLock();

    private int number = 30;

    public void saleTicket(){

        lock.lock();
        try {
            if (number>0){
                System.out.println(Thread.currentThread().getName() + "卖出第" + (number--) + "票,还剩下:"+number);
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

}
```

> synchronized 和 lock 区别

1. 首先synchronized是java内置关键字，在jvm层面，Lock是个java类；
2. synchronized无法判断是否获取锁的状态，Lock可以判断是否获取到锁；
3. synchronized会自动释放锁(a 线程执行完同步代码会释放锁 ； b 线程执行过程中发生异常会释放锁)，Lock需在finally中手工释放锁（unlock()方法释放锁），否则容易造成线程死锁；
4. 用synchronized关键字的两个线程1和线程2，如果当前线程1获得锁，线程2线程等待。如果线程1阻塞，线程2则会一直等待下去，而Lock锁就不一定会等待下去，如果尝试获取不到锁，线程可以不用一直等待就结束了；
5. synchronized的锁可重入、不可中断、非公平，而Lock锁可重入、可判断、可公平（两者皆可）
6. Lock锁适合大量同步的代码的同步问题，synchronized锁适合代码少量的同步问题。

```
题目：  synchronized和Lock有什么区别？用新的Lock有什么好处？你举例说说
1 原始构成
 synchronized是关键字属于JVM层面，
     monitorenter(底层是通过monitor对象来完成，其实wait/notify等方法也依赖于monitor对象只有在同步块或方法中才能调wait/notify等方
     monitorexit
 Lock是具体类(java.util.concurrent.locks.Lock)是api层面的锁

2 使用方法
 synchronized 不需要用户去手动释放锁，当synchronized代码执行完后系统会自动让线程释放对锁的占用
 ReentrantLock则需要用户去手动释放锁若没有主动释放锁，就有可能导致出现死锁现象。
 需要lock()和unlock()方法配合try/finally语句块来完成。

3 等待是否可中断
 synchronized不可中断，除非抛出异常或者正常运行完成
 ReentrantLock 可中断，1.设置超时方法 tryLock(long timeout, TimeUnit unit)
                    2.lockInterruptibly()放代码块中，调用interrupt()方法可中断
4 加锁是否公平
 synchronized非公平锁
 ReentrantLock两者都可以，默认非公平锁，构造方法可以传入boolean值，true为公平锁，false为非公平锁

 5 锁绑定多个条件Condition
 synchronized没有
 ReentrantLock用来实现分组唤醒需要唤醒的线程们，可以精确唤醒，而不是像synchronized要么随机唤醒一个线程要么唤醒全部线程。
```

4、生产者和消费者

线程间的通信，线程之间要协调和调度

> 生产者和消费者 synchroinzed 版

```
package com.kuang;

/**
 * 题目：现在两个线程，可以操作初始值为0的一个变量
 *        实现一个线程对该变量 + 1，一个线程对该变量 -1
 *        实现交替10次
 *
 * 诀窍：
 *      1. 高内聚低耦合的前提下，线程操作资源类
 *      2. 判断 、干活、通知
 */
public class B {
    public static void main(String[] args) throws Exception {
        Data data = new Data();

        new Thread(()->{
            for (int i = 1; i <= 10; i++) {
                try {
                    data.increment();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        },"A").start();

        new Thread(()->{
            for (int i = 1; i <= 10; i++) {
                try {
                    data.decrement();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        },"B").start();
    }
}

class Data{ // 资源类

    private int number = 0;

    public synchronized void increment() throws InterruptedException {
        // 判断该不该这个线程做
        if (number!=0){
            this.wait();
        }
        // 干活
        number++;
        System.out.println(Thread.currentThread().getName()+"\t"+number);
        // 通知
```

```
        this.notifyAll();
    }

    public synchronized void decrement() throws InterruptedException {
        // 判断该不该这个线程做
        if (number==0){
            this.wait();
        }
        // 干活
        number--;
        System.out.println(Thread.currentThread().getName()+"\t"+number);
        // 通知
        this.notifyAll();
    }

}
```

> 问题升级：防止虚假唤醒，4个线程，两个加，两个减

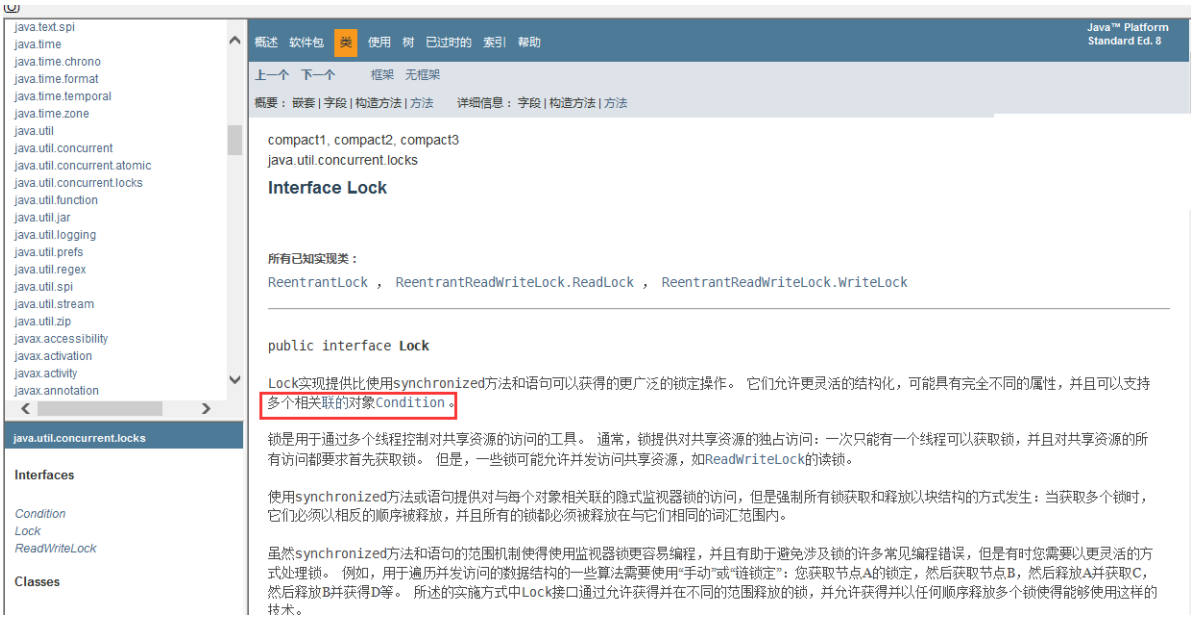
【重点】if 和 while

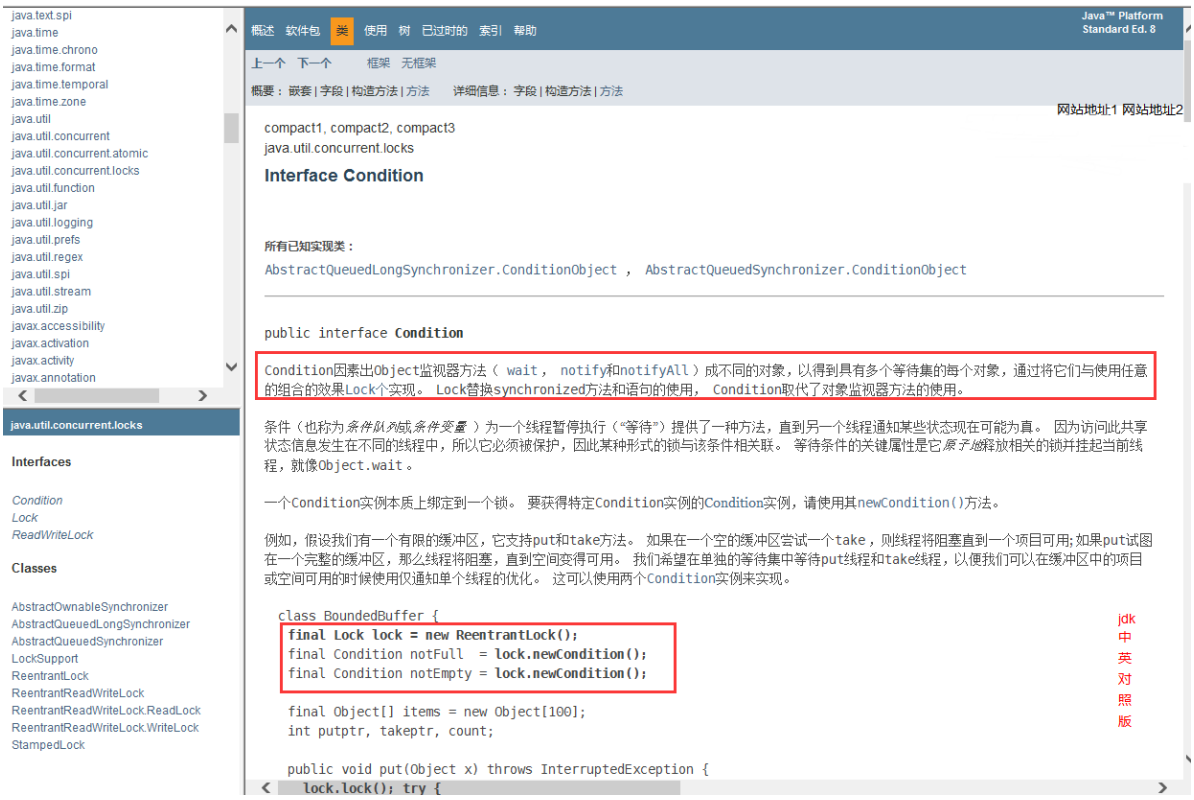

```
package com.kuang;

/**
 * 题目：现在四个线程，可以操作初始值为0的一个变量
 *       实现两个线程对该变量 + 1，两个线程对该变量 -1
 *       实现交替10次
 *
 * 诀窍：
 *     1. 高内聚低耦合的前提下，线程操作资源类
 *     2. 判断 、干活、通知
 *     3. 多线程交互中，必须要防止多线程的虚假唤醒，也即（判断不能用if，只能用while）
 */
public class B {
    public static void main(String[] args) throws Exception {
        Data data = new Data();

        new Thread(()->{
            for (int i = 1; i <= 10; i++) {
```

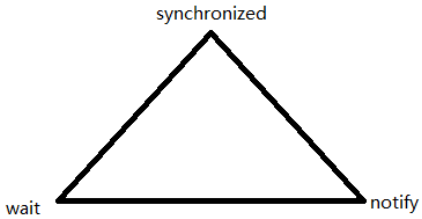
```
                try {
                    data.increment();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        },"A").start();

        new Thread(()->{
            for (int i = 1; i <= 10; i++) {
                try {
                    data.decrement();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        },"B").start();

        new Thread(()->{
            for (int i = 1; i <= 10; i++) {
                try {
                    data.increment();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        },"C").start();

        new Thread(()->{
            for (int i = 1; i <= 10; i++) {
                try {
                    data.decrement();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        },"D").start();
    }
}

class Data{ // 资源类

    private int number = 0;

    public synchronized void increment() throws InterruptedException {
        // 判断该不该这个线程做
        while (number!=0){
            this.wait();
        }
        // 干活
        number++;
        System.out.println(Thread.currentThread().getName()+"\t"+number);
        // 通知
        this.notifyAll();
    }

    public synchronized void decrement() throws InterruptedException {
        // 判断该不该这个线程做
```

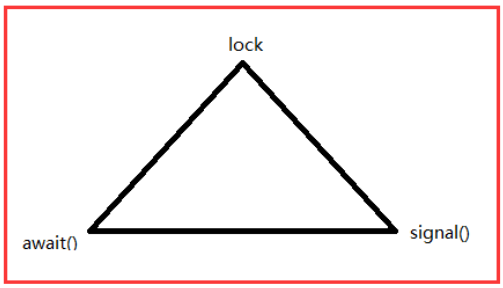
```
77          while (number==0){
78              this.wait();
79          }
80          // 干活
81          number--;
82          System.out.println(Thread.currentThread().getName()+"\t"+number);
83          // 通知
84          this.notifyAll();
85      }
86
87  }
```

新版生产者和消费者写法









闲聊常见笔试题：手写单例模式、手写冒泡排序、手写生产者消费者

代码测试：

```java
package com.kuang;

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * 题目：现在四个线程，可以操作初始值为0的一个变量
 * 实现两个线程对该变量 + 1，两个线程对该变量 -1
 * 实现交替10次
 * <p>
 * 诀窍：
 * 1. 高内聚低耦合的前提下，线程操作资源类
 * 2. 判断 、干活、通知
 * 3. 多线程交互中，必须要防止多线程的虚假唤醒，也即（判断不能用if，只能用while）
 */
public class B {
    public static void main(String[] args) throws Exception {
        Data data = new Data();

        new Thread(() -> {
            for (int i = 1; i <= 10; i++) {
                try {
                    data.increment();
                } catch (InterruptedException e) {
                    e.printStackTrace();
```

```
                }
            }
        }, "A").start();

        new Thread(() -> {
            for (int i = 1; i <= 10; i++) {
                try {
                    data.decrement();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }, "B").start();

        new Thread(() -> {
            for (int i = 1; i <= 10; i++) {
                try {
                    data.increment();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }, "C").start();

        new Thread(() -> {
            for (int i = 1; i <= 10; i++) {
                try {
                    data.decrement();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }, "D").start();
    }
}

class Data { // 资源类

    private int number = 0;
    private Lock lock = new ReentrantLock();
    private Condition condition = lock.newCondition();

    public void increment() throws InterruptedException {

        lock.lock();
        try {
            // 判断该不该这个线程做
            while (number != 0) {
                condition.await();
            }
            // 干活
            number++;
            System.out.println(Thread.currentThread().getName() + "\t" +
number);
            // 通知
            condition.signalAll();
        } catch (Exception e) {
            e.printStackTrace();
```

```
        } finally {
            lock.unlock();
        }

    }

    public void decrement() throws InterruptedException {
        lock.lock();
        try {
            // 判断该不该这个线程做
            while (number == 0) {
                condition.await();
            }
            // 干活
            number--;
            System.out.println(Thread.currentThread().getName() + "\t" +
number);
            // 通知
            condition.signalAll();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

}
```

精确通知顺序访问

```
package com.kuang;

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * 题目：多线程之间按顺序调用，实现 A->B->C
 *        三个线程启动，要求如下：
 *        AA 打印5次，BB 打印10次。CC打印15次，依次循环
 *
 * 重点：标志位
 */
public class C {
    public static void main(String[] args) {
        Resources resources = new Resources();

        new Thread(()->{
            for (int i = 1; i <=10; i++) {
                resources.print5();
            }
        },"AA").start();

        new Thread(()->{
            for (int i = 1; i <=10; i++) {
```

```
                resources.print10();
            }
        },"BB").start();

        new Thread(()->{
            for (int i = 1; i <=10; i++) {
                resources.print15();
            }
        },"CC").start();
    }
}

class Resources{ // 资源类

    private int number = 1; // 1A  2B  3C
    private Lock lock = new ReentrantLock();
    private Condition condition1 = lock.newCondition();
    private Condition condition2 = lock.newCondition();
    private Condition condition3 = lock.newCondition();

    public void print5(){
        lock.lock();
        try {
            // 判断
            while (number!=1){
                condition1.await();
            }
            // 干活
            for (int i = 1; i <= 5; i++) {

 System.out.println(Thread.currentThread().getName()+'\t'+i);
            }
            // 通知,指定的干活!
            number = 2;
            condition2.signal();

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public void print10(){
        lock.lock();
        try {
            // 判断
            while (number!=2){
                condition2.await();
            }
            // 干活
            for (int i = 1; i <= 10; i++) {

 System.out.println(Thread.currentThread().getName()+'\t'+i);
            }
            // 通知,指定的干活!
            number = 3;
            condition3.signal();
```

```
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public void print15(){
        lock.lock();
        try {
            // 判断
            while (number!=3){
                condition3.await();
            }
            // 干活
            for (int i = 1; i <= 15; i++) {

 System.out.println(Thread.currentThread().getName()+'\t'+i);
            }
            // 通知,指定的干活!
            number = 1;
            condition1.signal();

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

}
```

5、8锁的现象

1、标准访问，请问先打印邮件还是短信?

```
package com.kuang;

/**
 * 多线程的8锁
 * 1、标准访问，请问先打印邮件还是短信？
 */
public class Lock8 {
    public static void main(String[] args) throws InterruptedException {
        Phone phone = new Phone();

        new Thread(()->{
            try {
                phone.sendEmail();
            } catch (Exception e) {
                e.printStackTrace();
            }
        },"A").start();

```

```
        Thread.sleep(200);

        new Thread(()->{
            try {
                phone.sendSMS();
            } catch (Exception e) {
                e.printStackTrace();
            }
        },"B").start();
    }
}


class Phone{
    public synchronized void sendEmail() throws Exception{
        System.out.println("sendEmail");
    }
    public synchronized void sendSMS() throws Exception{
        System.out.println("sendSMS");
    }
}
```

结论：被synchronized修饰的方法，锁的对象是方法的调用者。因为两个方法的调用者是同一个，所以两个方法用的是同一个锁，先调用方法的先执行。

2、邮件方法暂停4秒钟，请问先打印邮件还是短信?

```
package com.kuang;

import java.util.concurrent.TimeUnit;

/**
 * 多线程的8锁
 * 1、标准访问，请问先打印邮件还是短信？
 * 2、邮件方法暂停4秒钟，请问先打印邮件还是短信？
 */
public class Lock8 {
    public static void main(String[] args) throws InterruptedException {
        Phone phone = new Phone();

        new Thread(()->{
            try {
                phone.sendEmail();
            } catch (Exception e) {
                e.printStackTrace();
            }
        },"A").start();

        Thread.sleep(200);

        new Thread(()->{
            try {
                phone.sendSMS();
            } catch (Exception e) {
                e.printStackTrace();
            }
```

```
        },"B").start();
    }
}


class Phone{
    public synchronized void sendEmail() throws Exception{
        try {
            TimeUnit.SECONDS.sleep(4);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("sendEmail");
    }
    public synchronized void sendSMS() throws Exception{
        System.out.println("sendSMS");
    }
}
```

结论：被synchronized修饰的方法，锁的对象是方法的调用者。因为两个方法的调用者是同一个，所以两个方法用的是同一个锁，先调用方法的先执行，第二个方法只有在第一个方法执行完释放锁之后才能执行。

3、新增一个普通方法hello()没有同步,请问先打印邮件还是hello?

```
package com.kuang;

import java.util.concurrent.TimeUnit;

/**
 * 多线程的8锁
 * 1、标准访问，请问先打印邮件还是短信？
 * 2、邮件方法暂停4秒钟，请问先打印邮件还是短信？
 * 3、新增一个普通方法hello()没有同步,请问先打印邮件还是hello?
 */
public class Lock8 {
    public static void main(String[] args) throws InterruptedException {
        Phone phone = new Phone();

        new Thread(()->{
            try {
                phone.sendEmail();
            } catch (Exception e) {
                e.printStackTrace();
            }
        },"A").start();

        Thread.sleep(200);

        new Thread(()->{
            try {
                // phone.sendSMS();
                phone.hello();
            } catch (Exception e) {
                e.printStackTrace();
            }
```

```
        },"B").start();
    }
}

class Phone{
    public synchronized void sendEmail() throws Exception{
        try {
            TimeUnit.SECONDS.sleep(4);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("sendEmail");
    }
    public synchronized void sendSMS() throws Exception{
        System.out.println("sendSMS");
    }

    public void hello(){
        System.out.println("Hello");
    }
}
```

结论：新增的方法没有被synchronized修饰，不是同步方法，不受锁的影响，所以不需要等待。其他线程共用了一把锁，所以还需要等待。

4、两部手机、请问先打印邮件还是短信?

```
package com.kuang;

import java.util.concurrent.TimeUnit;

/**
 * 多线程的8锁
 * 1、标准访问，请问先打印邮件还是短信？
 * 2、邮件方法暂停4秒钟，请问先打印邮件还是短信？
 * 3、新增一个普通方法hello()没有同步,请问先打印邮件还是hello？
 * 4、两部手机、请问先打印邮件还是短信？
 */
public class Lock8 {
    public static void main(String[] args) throws InterruptedException {
        Phone phone = new Phone();
        Phone phone2 = new Phone();

        new Thread(()->{
            try {
                phone.sendEmail();
            } catch (Exception e) {
                e.printStackTrace();
            }
        },"A").start();

        Thread.sleep(200);

        new Thread(()->{
            try {
                // phone.sendSMS();
```

```
                // phone.hello();
                phone2.sendSMS();
            } catch (Exception e) {
                e.printStackTrace();
            }
        },"B").start();
    }
}


class Phone{
    public synchronized void sendEmail() throws Exception{
        try {
            TimeUnit.SECONDS.sleep(4);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("sendEmail");
    }
    public synchronized void sendSMS() throws Exception{
        System.out.println("sendSMS");
    }
}
```

结论：被synchronized修饰的方法，锁的对象是方法的调用者。因为用了两个对象调用各自的方法，所以两个方法的调用者不是同一个，所以两个方法用的不是同一个锁，后调用的方法不需要等待先调用的方法。

5、两个静态同步方法，同一部手机，请问先打印邮件还是短信?

```
package com.kuang;

import java.util.concurrent.TimeUnit;

/**
 * 多线程的8锁
 * 1、标准访问，请问先打印邮件还是短信？
 * 2、邮件方法暂停4秒钟，请问先打印邮件还是短信？
 * 3、新增一个普通方法hello()没有同步,请问先打印邮件还是hello？
 * 4、两部手机、请问先打印邮件还是短信？
 * 5、两个静态同步方法，同一部手机，请问先打印邮件还是短信？
 */
public class Lock8 {
    public static void main(String[] args) throws InterruptedException {
        Phone phone = new Phone();

        new Thread(()->{
            try {
                phone.sendEmail();
            } catch (Exception e) {
                e.printStackTrace();
            }
        },"A").start();

        Thread.sleep(200);

```

```
        new Thread(()->{
            try {
                phone.sendSMS();
            } catch (Exception e) {
                e.printStackTrace();
            }
        },"B").start();
    }
}


class Phone{
    public static synchronized void sendEmail() throws Exception{
        try {
            TimeUnit.SECONDS.sleep(4);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("sendEmail");
    }
    public static synchronized void sendSMS() throws Exception{
        System.out.println("sendSMS");
    }
}
```

结论：被synchronized和static修饰的方法，锁的对象是类的class对象。因为两个同步方法都被static修饰了，所以两个方法用的是同一个锁，后调用的方法需要等待先调用的方法。

6、两个静态同步方法，2部手机，请问先打印邮件还是短信?

```
package com.kuang;

import java.util.concurrent.TimeUnit;

/**
 * 多线程的8锁
 * 1、标准访问，请问先打印邮件还是短信？
 * 2、邮件方法暂停4秒钟，请问先打印邮件还是短信？
 * 3、新增一个普通方法hello()没有同步,请问先打印邮件还是hello？
 * 4、两部手机、请问先打印邮件还是短信？
 * 5、两个静态同步方法，同一部手机，请问先打印邮件还是短信？
 * 6、两个静态同步方法，2部手机，请问先打印邮件还是短信？
 */
public class Lock8 {
    public static void main(String[] args) throws InterruptedException {
        Phone phone = new Phone();
        Phone phone2 = new Phone();

        new Thread(()->{
            try {
                phone.sendEmail();
            } catch (Exception e) {
                e.printStackTrace();
            }
        },"A").start();

```

```
        Thread.sleep(200);

        new Thread(()->{
            try {
                phone2.sendSMS();
            } catch (Exception e) {
                e.printStackTrace();
            }
        },"B").start();
    }
}

class Phone{
    public static synchronized void sendEmail() throws Exception{
        try {
            TimeUnit.SECONDS.sleep(4);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("sendEmail");
    }
    public static synchronized void sendSMS() throws Exception{
        System.out.println("sendSMS");
    }
}
```

结论：被synchronized和static修饰的方法，锁的对象是类的class对象。因为两个同步方法都被static修饰了，即便用了两个不同的对象调用方法，两个方法用的还是同一个锁，后调用的方法需要等待先调用的方法。

7、一个普通同步方法，一个静态同步方法，同一部手机，请问先打印邮件还是短信?

```
package com.kuang;

import java.util.concurrent.TimeUnit;

/**
 * 多线程的8锁
 * 1、标准访问，请问先打印邮件还是短信？
 * 2、邮件方法暂停4秒钟，请问先打印邮件还是短信？
 * 3、新增一个普通方法hello()没有同步,请问先打印邮件还是hello？
 * 4、两部手机、请问先打印邮件还是短信？
 * 5、两个静态同步方法，同一部手机，请问先打印邮件还是短信？
 * 6、两个静态同步方法，2部手机，请问先打印邮件还是短信？
 * 7、一个普通同步方法，一个静态同步方法，同一部手机，请问先打印邮件还是短信？
 */
public class Lock8 {
    public static void main(String[] args) throws InterruptedException {
        Phone phone = new Phone();

        new Thread(()->{
            try {
                phone.sendEmail();
            } catch (Exception e) {
                e.printStackTrace();
            }
```

```
        },"A").start();

        Thread.sleep(200);

        new Thread(()->{
            try {
                phone.sendSMS();
            } catch (Exception e) {
                e.printStackTrace();
            }
        },"B").start();
    }
}


class Phone{
    public static synchronized void sendEmail() throws Exception{
        try {
            TimeUnit.SECONDS.sleep(4);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("sendEmail");
    }
    public synchronized void sendSMS() throws Exception{
        System.out.println("sendSMS");
    }
}
```

结论：被synchronized和static修饰的方法，锁的对象是类的class对象。仅仅被synchronized修饰的方法，锁的对象是方法的调用者。因为两个方法锁的对象不是同一个，所以两个方法用的不是同一个锁，后调用的方法不需要等待先调用的方法。

8、一个普通同步方法，一个静态同步方法，2部手机，请问先打印邮件还是短信?

```
package com.kuang;

import java.util.concurrent.TimeUnit;

/**
 * 多线程的8锁
 * 1、标准访问，请问先打印邮件还是短信？
 * 2、邮件方法暂停4秒钟，请问先打印邮件还是短信？
 * 3、新增一个普通方法hello()没有同步,请问先打印邮件还是hello？
 * 4、两部手机、请问先打印邮件还是短信？
 * 5、两个静态同步方法，同一部手机，请问先打印邮件还是短信？
 * 6、两个静态同步方法，2部手机，请问先打印邮件还是短信？
 * 7、一个普通同步方法，一个静态同步方法，同一部手机，请问先打印邮件还是短信？
 * 8、一个普通同步方法，一个静态同步方法，2部手机，请问先打印邮件还是短信？
 */
public class Lock8 {
    public static void main(String[] args) throws InterruptedException {
        Phone phone = new Phone();
        Phone phone2 = new Phone();

        new Thread(()->{
```

```
            try {
                phone.sendEmail();
            } catch (Exception e) {
                e.printStackTrace();
            }
        },"A").start();

        Thread.sleep(200);

        new Thread(()->{
            try {
                phone2.sendSMS();
            } catch (Exception e) {
                e.printStackTrace();
            }
        },"B").start();
    }
}


class Phone{
    public static synchronized void sendEmail() throws Exception{
        try {
            TimeUnit.SECONDS.sleep(4);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("sendEmail");
    }
    public synchronized void sendSMS() throws Exception{
        System.out.println("sendSMS");
    }
}
```

结论：被synchronized和static修饰的方法，锁的对象是类的class对象。仅仅被synchronized修饰的方法，锁的对象是方法的调用者。即便是用同一个对象调用两个方法，锁的对象也不是同一个，所以两个方法用的不是同一个锁，后调用的方法不需要等待先调用的方法。

> 小结

new　this　具体的一个手机

static　class　唯一的一个模板

一个对象里面如果有多个synchronized方法，某个时刻内，只要一个线程去调用其中一个synchronized方法了，其他的线程都要等待，换句话说，在某个时刻内，只能有唯一一个线程去访问这些synchronized方法，锁的是当前对象this，被锁定后，其他的线程都不能进入到当前对象的其他的synchronized方法

加个普通方法后发现和同步锁无关，换成两个对象后，不是同一把锁，情况立刻变化

都换成静态同步方法后，情况又变化了。所有的非静态的同步方法用的都是同一把锁----实例对象本身

synchronized实现同步的基础：java中的每一个对象都可以作为锁

具体的表现为以下三种形式：
对于普通同步方法，锁的是当前实例对象
对于静态同步方法，锁的是当前的Class对象。
对于同步方法块，锁是synchronized括号里面的配置对象

当一个线程试图访问同步代码块时，他首先必须得到锁，退出或者是抛出异常时必须释放锁，也就是说如果一个实例对象的非静态同步方法获取锁后，该实例对象的其他非静态同步方法必须等待获取锁的方法释放锁后才能获取锁，可以是别的实例对象非非静态同步方法因为跟该实例对象的非静态同步方法用的是不同的锁，所以必须等待该实例对象已经获取锁的非静态同步方法释放锁就可以获取他们自己的锁。

所有的静态同步方法用的也是同一把锁----类对象本身
这两把锁的是两个不同的对象，所以静态的同步方法与非静态的同步方法之间是不会有竞争条件的，但是一旦一个静态同步方法获取锁后，其他的静态同步方法都必须等待该方法释放锁后才能获取锁，而不是同一个实例对象的静态同步方法之间，还是不同的实例对象的静态同步方法之间，只要他们用一个的是同一个类的实例对象。

6、集合类不安全

> list 不安全

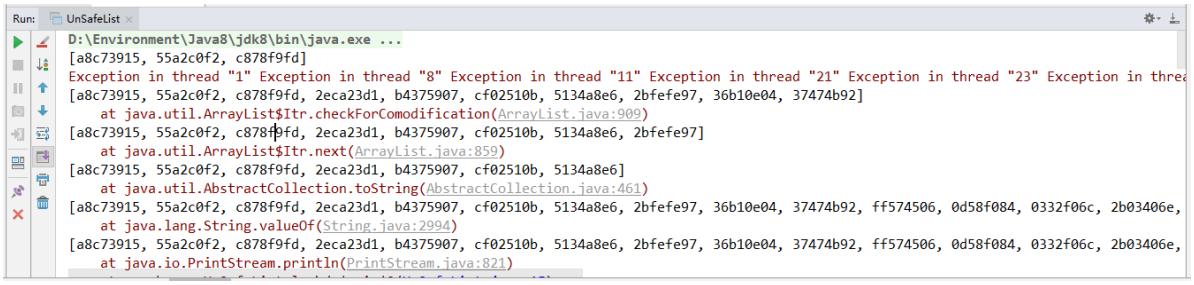
单线程下：

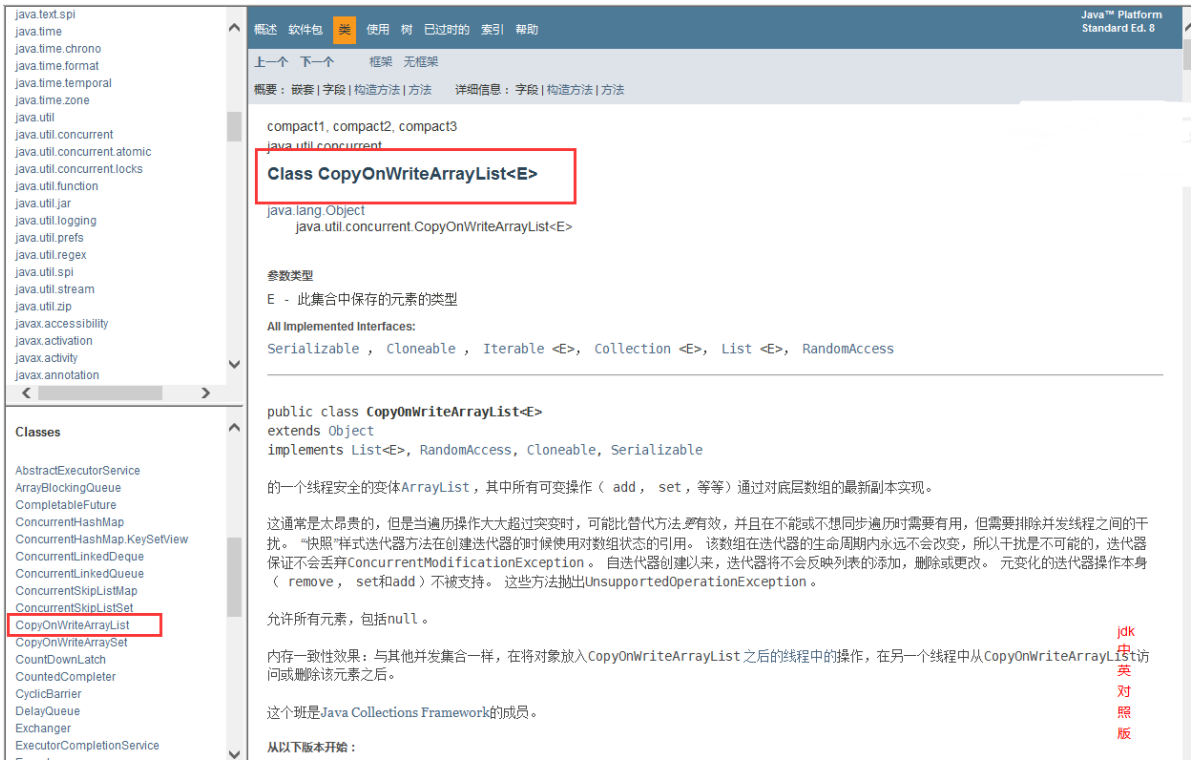
```
package com.kuang;

import java.util.ArrayList;
import java.util.List;
import java.util.UUID;

// 单线程十分安全
public class UnSafeList {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("a","b","c");
        list.forEach(System.out::println);
    }
}
```

多线程下：

```
package com.kuang;

import java.util.ArrayList;
import java.util.List;
import java.util.UUID;

public class UnSafeList {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();

        // 对比3个线程  和  30个线程，看区别
        for (int i = 1; i <= 30; i++) {
            new Thread(()->{
```

```
                list.add(UUID.randomUUID().toString().substring(0,8));
                System.out.println(list);
            },String.valueOf(i)).start();
        }

    }
}
```






```
package com.kuang;

import java.lang.reflect.Array;
import java.util.*;
import java.util.concurrent.CopyOnWriteArrayList;

/**
 * 1 故障现象：ConcurrentModificationException
 * 2 导致原因：add 方法没有加锁
 * 3 解决方案：换一个集合类
 *      1、List<String> list = new Vector<>();   JDK1.0 就存在了！
 *      2、List<String> list = Collections.synchronizedList(new ArrayList<>
());
 *      3、List<String> list = new CopyOnWriteArrayList<>();
 */
public class UnSafeList {
    public static void main(String[] args) {
        List<String> list = new CopyOnWriteArrayList<>();

        for (int i = 1; i <= 30; i++) {
```

```
            new Thread(()->{
                list.add(UUID.randomUUID().toString().substring(0,8));
                System.out.println(list);
            },String.valueOf(i)).start();
        }

    }
}
```

写入时复制（CopyOnWrite）思想

写入时复制（CopyOnWrite，简称COW）思想是计算机程序设计领域中的一种优化策略。其核心思想是，如果有多个调用者（Callers）同时要求相同的资源（如内存或者是磁盘上的数据存储），他们会共同获取相同的指针指向相同的资源，直到某个调用者视图修改资源内容时，系统才会真正复制一份专用副本（private copy）给该调用者，而其他调用者所见到的最初的资源仍然保持不变。这过程对其他的调用者都是透明的（transparently）。此做法主要的优点是如果调用者没有修改资源，就不会有副本（private copy）被创建，因此多个调用者只是读取操作时可以共享同一份资源。

读写分离，写时复制出一个新的数组，完成插入、修改或者移除操作后将新数组赋值给array

CopyOnWriteArrayList为什么并发安全且性能比Vector好

我知道Vector是增删改查方法都加了synchronized，保证同步，但是每个方法执行的时候都要去获得锁，性能就会大大下降，而CopyOnWriteArrayList 只是在增删改上加锁，但是读不加锁，在读方面的性能就好于Vector，CopyOnWriteArrayList支持读多写少的并发情况。

> set 不安全

同理

```
package com.kuang;

import java.lang.reflect.Array;
import java.util.*;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.concurrent.CopyOnWriteArraySet;

/**
 * 1 故障现象：ConcurrentModificationException
 * 2 导致原因：add 方法没有加锁
 * 3 解决方案：换一个集合类
 *      1、Set<String> set = new HashSet<>(); 默认
 *      2、Set<String> set = Collections.synchronizedSet(new HashSet<>());
 *      3、Set<String> set = new CopyOnWriteArraySet();
 * 4 优化建议：（同样的错误，不出现第2次）
 *
 */
public class UnSafeList {
    public static void main(String[] args) {
        Set<String> set = new CopyOnWriteArraySet();

        for (int i = 1; i <= 30; i++) {
            new Thread(()->{
                set.add(UUID.randomUUID().toString().substring(0,8));
                System.out.println(set);
            },String.valueOf(i)).start();
```

```
        }

    }
}
```

hashset底层就是hashMap

```
Set<String> set = new HashSet<>();
// 点进去
public HashSet() {
    map = new HashMap<>();
}

// add方法 就是map的put方法
public boolean add(E e) {
    return map.put(e, PRESENT)==null;
}

//private static final Object PRESENT = new Object(); 不变得值
```

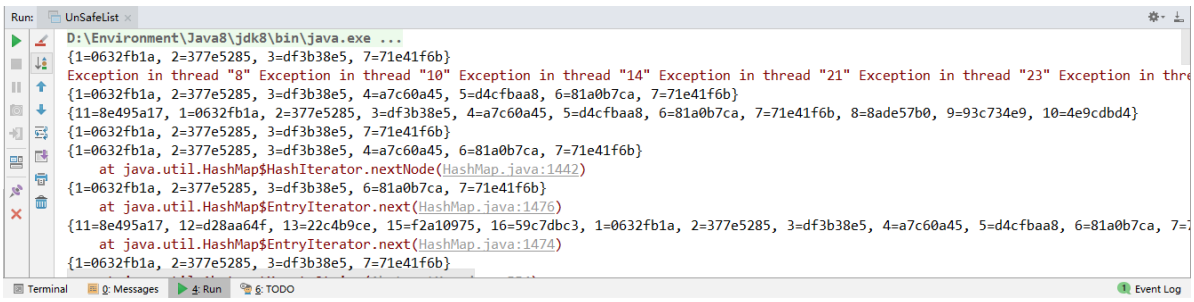
> map 不安全

hashMap底层是数组+链表+红黑树

```
Map<String,String> map = new HashMap<>();
// 等价于
Map<String,String> map = new HashMap<>(16,0.75);
// 工作中，常常会自己根据业务来写参数，提高效率
```

map不安全测试：

```
public static void main(String[] args) {
    Map<String,String> map = new HashMap<>();
    // 人生如程序，不是选择就是循环，时常的自我总结十分的重要
    for (int i = 1; i <= 30; i++) {
        new Thread(()->{

 map.put(Thread.currentThread().getName(),UUID.randomUUID().toString().substring(0,8));
            System.out.println(map);
        },String.valueOf(i)).start();
    }
}
```



注意名字：

```java
public class UnSafeList {
    public static void main(String[] args) {
        // Map<String,String> map = new HashMap<>();
        Map<String,String> map = new ConcurrentHashMap<>();
        for (int i = 1; i <= 30; i++) {
            new Thread(()->{

 map.put(Thread.currentThread().getName(),UUID.randomUUID().toString().substring(0,8));
                System.out.println(map);
            },String.valueOf(i)).start();
        }
    }
}
```

7、Callable

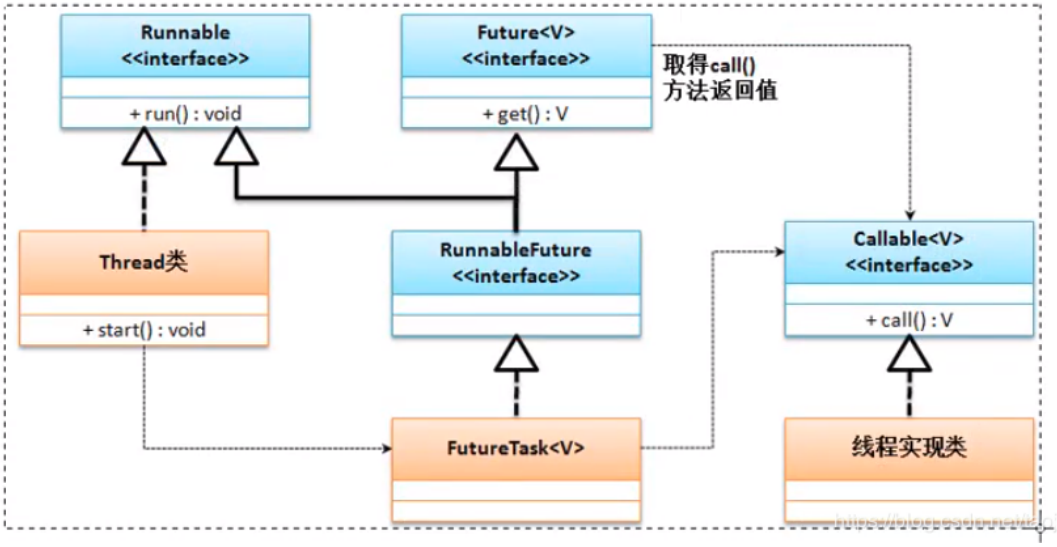
多线程中，第3种获得多线程的方式，Callable。它与Runnable有什么区别呢?

- 是否有返回值
- 是否抛异常
- 方法不一样，一个是call，一个是run

> 基础入门

```java
package com.kuang;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.FutureTask;

public class CallableDemo {
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        MyThread myThread = new MyThread();
        FutureTask futureTask = new FutureTask(myThread); // 适配类

        Thread t1 = new Thread(futureTask,"A"); // 调用执行
        t1.start();

        Integer result = (Integer) futureTask.get(); // 获取返回值
        System.out.println(result);
    }
}

class MyThread implements Callable<Integer>{
    @Override
    public Integer call() throws Exception {
        System.out.println("call 被调用");
        return 1024;
    }

}
```

Callable



> Callable 细节

练武不练功，到头一场空，天下武功没有高低之分，只有习武的人有强弱之别

```
1  package com.kuang;
2
3  import java.util.concurrent.Callable;
4  import java.util.concurrent.ExecutionException;
5  import java.util.concurrent.FutureTask;
6  import java.util.concurrent.TimeUnit;
7
8  /**
9   * 1、get 方法获得返回结果！ 一般放在最后一行！否则可能会阻塞
10  */
11 public class CallableDemo {
12     public static void main(String[] args) throws ExecutionException,
   InterruptedException {
13         MyThread myThread = new MyThread();
14         FutureTask futureTask = new FutureTask(myThread); // 适配类
15
16         new Thread(futureTask,"A").start(); // 调用执行
17         new Thread(futureTask,"B").start(); // 第二次调用执行，会有结果缓存，不用
   再次计算
18
19         System.out.println(Thread.currentThread().getName()+" OK");
20
21         Integer result = (Integer) futureTask.get(); // 获取返回值
22         System.out.println(result);
23
24     }
25 }
26
27 class MyThread implements Callable<Integer>{
28     @Override
29     public Integer call() throws Exception {
30         System.out.println("call 被调用");
31         try {
32             TimeUnit.SECONDS.sleep(3);
33         } catch (InterruptedException e) {
```

```
            e.printStackTrace();
        }
        return 1024;
    }

}
```

8、常用辅助类

8.1、CountDownLatch

```
package com.kuang;

import java.util.concurrent.CountDownLatch;

public class CountDownLatchDemo {

    public static void main(String[] args) throws InterruptedException {
        // 计数器
        CountDownLatch countDownLatch = new CountDownLatch(6);

        for (int i = 1; i <= 6; i++) {
            new Thread(()->{
                System.out.println(Thread.currentThread().getName()+"
Start");
                countDownLatch.countDown(); // 计数器-1
            },String.valueOf(i)).start();
        }
        //阻塞等待计数器归零
        countDownLatch.await();
        System.out.println(Thread.currentThread().getName()+" End");
    }

    /**
     * 顺序不一定，结果诡异，达不到预期的最后End
     */
    public void test1(){
        for (int i = 1; i <= 6; i++) {
            new Thread(()->{
                System.out.println(Thread.currentThread().getName()+"
Start");
            },String.valueOf(i)).start();
        }
        System.out.println(Thread.currentThread().getName()+" End");
    }
}
```

原理：

- CountDownLatch 主要有两个方法，当一个或多个线程调用 await 方法时，这些线程会阻塞
- 其他线程调用CountDown方法会将计数器减1（调用CountDown方法的线程不会阻塞）
- 当计数器变为0时，await 方法阻塞的线程会被唤醒，继续执行

8.2、CyclicBarrier

翻译：CyclicBarrier 篱栅

作用：和上面的减法相反，这里是加法，好比集齐7个龙珠召唤神龙，或者人到齐了再开会！

```
package com.kuang;

import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class CyclicBarrierDemo {
    public static void main(String[] args) {
        // CyclicBarrier(int parties, Runnable barrierAction)
        CyclicBarrier cyclicBarrier = new CyclicBarrier(7,()->{
            System.out.println("召唤神龙成功");
        });

        for (int i = 1; i <= 7; i++) {
            final int tempInt = i;
            new Thread(()->{
                System.out.println(Thread.currentThread().getName()+"收集了第"+ tempInt +"颗龙珠");
                try {
                    cyclicBarrier.await(); // 等待
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } catch (BrokenBarrierException e) {
                    e.printStackTrace();
                }
            }).start();
        }

    }
}
```

8.3、Semaphore

翻译：Semaphore 信号量;信号灯;信号

作用：抢车位

```
package com.kuang;

import java.util.concurrent.Semaphore;
import java.util.concurrent.TimeUnit;

/**
 * 信号灯
 */
public class SemaphoreDemo {
    public static void main(String[] args) {
        // 模拟资源类，有3个空车位
        Semaphore semaphore = new Semaphore(3);
```

```
13
14          for (int i = 1; i <= 6; i++) { // 模拟6个车
15              new Thread(()->{
16                  try {
17                      semaphore.acquire(); // acquire 得到
18                      System.out.println(Thread.currentThread().getName()+" 抢到了车位");
19                      TimeUnit.SECONDS.sleep(3); // 停3秒钟
20                      System.out.println(Thread.currentThread().getName()+" 离开了车位");
21                  } catch (InterruptedException e) {
22                      e.printStackTrace();
23                  } finally {
24                      semaphore.release(); // 释放这个位置
25                  }
26              },String.valueOf(i)).start();
27          }
28
29      }
30  }
```

原理：

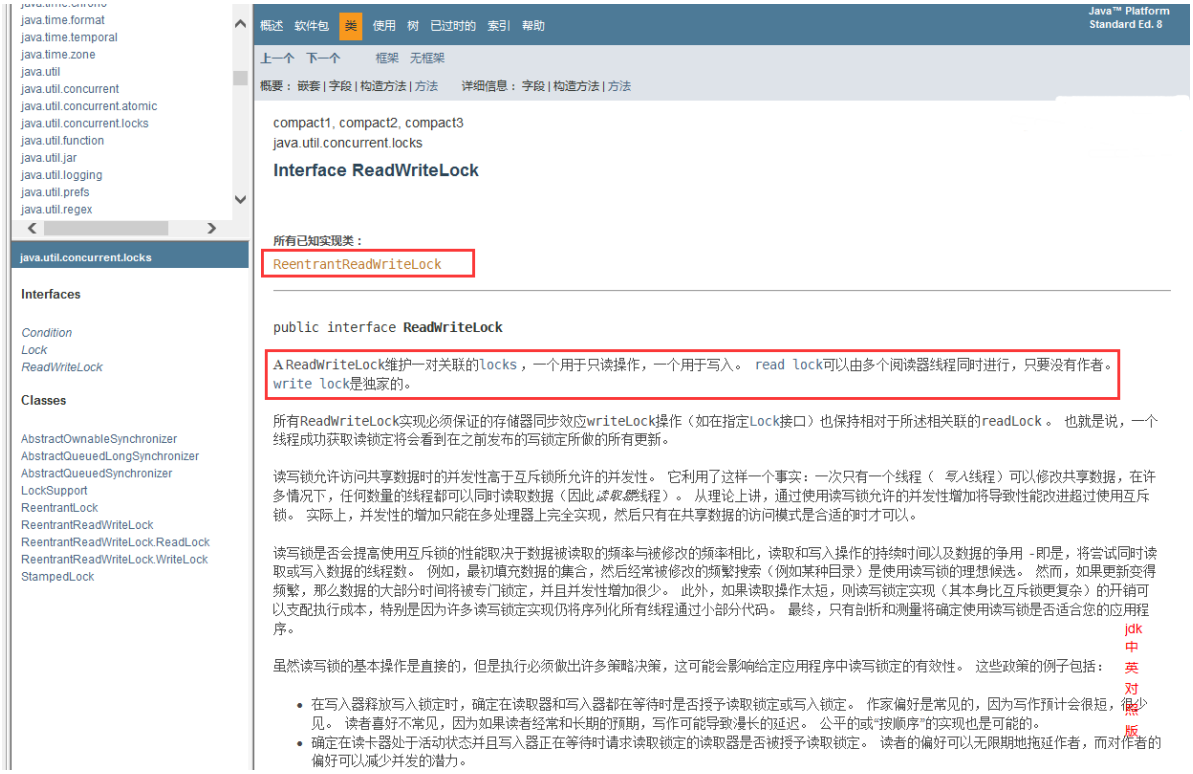
在信号量上我们定义两种操作：

- acquire（获取）
  - 当一个线程调用 acquire 操作时，他要么通过成功获取信号量（信号量-1）
  - 要么一直等下去，直到有线程释放信号量，或超时
- release （释放）
  - 实际上会将信号量的值 + 1，然后唤醒等待的线程。
- （release）

信号量主要用于两个目的：一个是用于多个共享资源的互斥使用，另一个用于并发线程数的控制。

9、读写锁

练武不练功，到老一场空！



独占锁（写锁）：指该锁一次只能被一个线程锁持有。对于ReentranrLock和 Synchronized 而言都是独占锁。

共享锁（读锁）：该锁可被多个线程所持有。

对于ReentrantReadWriteLock其读锁时共享锁，写锁是独占锁，读锁的共享锁可保证并发读是非常高效的。

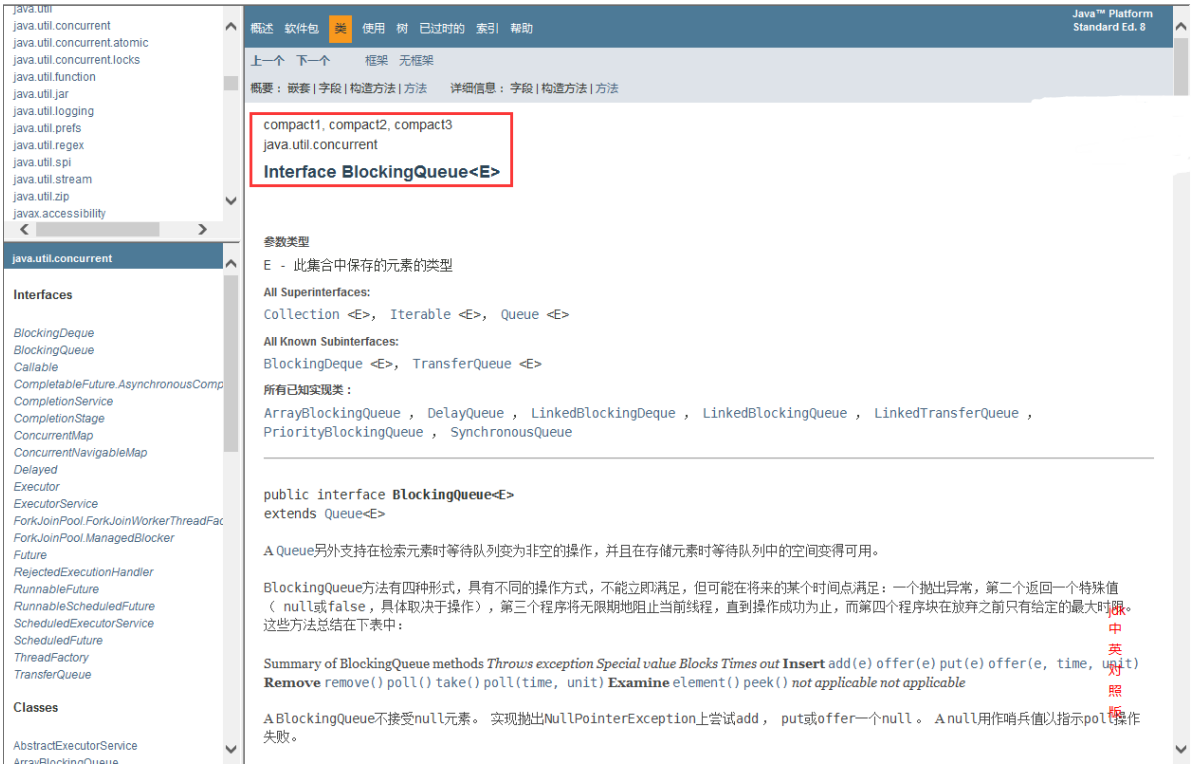
测试：

```
1  package com.kuang;
2
3
4  import java.util.HashMap;
5  import java.util.Map;
6  import java.util.concurrent.locks.ReadWriteLock;
7  import java.util.concurrent.locks.ReentrantReadWriteLock;
8
9  /**
10  * 多个线程同时读一个资源类没有任何问题，所以为了满足并发量，读取共享资源应该可以同时进行。
11  * 但是，如果有一个线程想去写共享资源，就不应该再有其他线程可以对该资源进行读或写。
12  * 1. 读-读 可以共存
13  * 2. 读-写 不能共存
14  * 3. 写-写 不能共存
15  */
16 public class ReadWriteLockDemo {
17     public static void main(String[] args) {
18         MyCacheLock myCache = new MyCacheLock();
19         // 写
20         for (int i = 1; i <= 5; i++) {
21             final int tempInt = i;
22             new Thread(()->{
23                 myCache.put(tempInt+"",tempInt+"");
24             },String.valueOf(i)).start();
25         }
```

```
26
27          // 读
28          for (int i = 1; i <= 5; i++) {
29              final int tempInt = i;
30              new Thread(()->{
31                  myCache.get(tempInt+"");
32              },String.valueOf(i)).start();
33          }
34      }
35  }
36
37  // 测试发现问题：写入的时候，还没写入完成，会存在其他的写入！造成问题
38  class MyCache{
39      private volatile Map<String,Object> map = new HashMap<>();
40
41      public void put(String key,Object value){
42          System.out.println(Thread.currentThread().getName()+" 写入"+key);
43          map.put(key,value);
44          System.out.println(Thread.currentThread().getName()+" 写入成功!");
45      }
46
47      public void get(String key){
48          System.out.println(Thread.currentThread().getName()+" 读取"+key);
49          Object result = map.get(key);
50          System.out.println(Thread.currentThread().getName()+" 读取结
    果："+result);
51      }
52  }
53
54  // 加锁
55  class MyCacheLock{
56      private volatile Map<String,Object> map = new HashMap<>();
57      private ReadWriteLock readWriteLock = new ReentrantReadWriteLock(); //
    读写锁
58
59      public void put(String key,Object value){
60          // 写锁
61          readWriteLock.writeLock().lock();
62          try {
63              System.out.println(Thread.currentThread().getName()+" 写入"+key);
64              map.put(key,value);
65              System.out.println(Thread.currentThread().getName()+" 写入成
    功!");
66          } catch (Exception e) {
67              e.printStackTrace();
68          } finally {
69              readWriteLock.writeLock().unlock();
70          }
71      }
72
73      public void get(String key){
74          // 读锁
75          readWriteLock.readLock().lock();
76          try {
77              System.out.println(Thread.currentThread().getName()+" 读取"+key);
78              Object result = map.get(key);
79              System.out.println(Thread.currentThread().getName()+" 读取结
    果："+result);
```

```
80          } catch (Exception e) {
81              e.printStackTrace();
82          } finally {
83              readWriteLock.readLock().unlock();
84          }
85      }
86  }
```

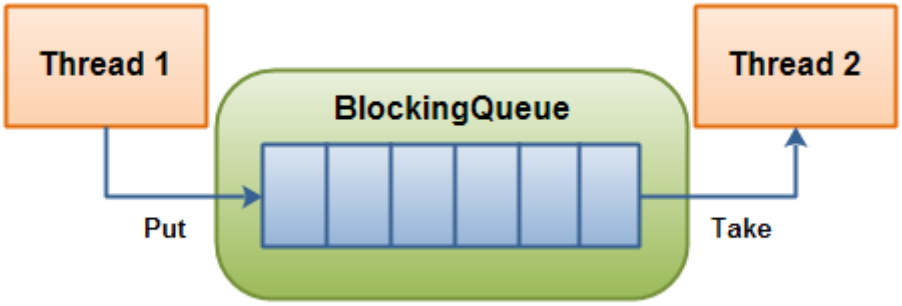
10、阻塞队列



> 阻塞队列

阻塞：必须要阻塞、不得不阻塞

阻塞队列是一个队列，在数据结构中起的作用如下图：



当队列是空的，从队列中获取元素的操作将会被阻塞。

当队列是满的，从队列中添加元素的操作将会被阻塞。

试图从空的队列中获取元素的线程将会被阻塞，直到其他线程往空的队列插入新的元素。

试图向已满的队列中添加新元素的线程将会被阻塞，直到其他线程从队列中移除一个或多个元素或者完全清空，使队列变得空闲起来并后续新增。

阻塞队列的用处：

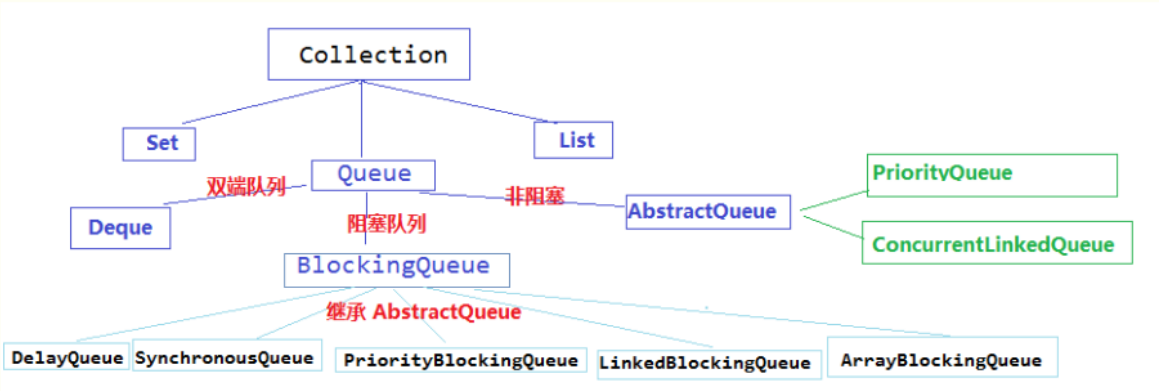
在多线程领域：所谓阻塞，在某些情况下会挂起线程（即阻塞），一旦条件满足，被挂起的线程又会自动被唤起。

为什么需要 BlockingQueue?

好处是我们不需要关心什么时候需要阻塞线程，什么时候需要唤醒线程，因为这一切BlockingQueue 都给你一手包办了。

在 concurrent 包发布以前，在多线程环境下，我们每个程序员都必须自己去控制这些细节，尤其还要兼顾效率和线程安全，而这会给我们的程序带来不小的复杂度。

> 接口架构图



- **ArrayBlockingQueue：由数组结构组成的有界阻塞队列。**
- **LinkedBlockingQueue：由链表结构组成的有界（默认值为：integer.MAX_VALUE）阻塞队列。**
- PriorityBlockingQueue：支持优先级排序的无界阻塞队列
- DelayQueue：使用优先级队列实现的延迟无界阻塞队列。
- **SynchronousQueue：不存储元素的阻塞队列，也即单个元素的队列。**
- LinkedTransferQueue：由链表组成的无界阻塞队列。
- LinkedBlockingDeque：由链表组成的双向阻塞队列。

> API 的使用

常用API：

方法\处理方式	抛出异常	返回特殊值	一直阻塞	超时退出
插入方法	add(e)	offer(e)	put(e)	offer(e,time,unit)
移除方法	remove()	poll()	take()	poll(time,unit)
检查方法	element()	peek()	不可用	不可用

尽量按组匹配使用

解释：

抛出异常	当阻塞队列满时，再往队列里add插入元素会抛IllegalStateException:Queue full 当阻塞队列空时，再往队列里remove移除元素会抛NoSuchElementException
特殊值	插入方法，成功ture失败false 移除方法，成功返回出队列的元素，队列里没有就返回null
一直阻塞	当阻塞队列满时，生产者线程继续往队列里put元素，队列会一直阻塞生产者线程直到put数据or响应中断退出 当阻塞队列空时，消费者线程试图从队列里take元素，队列会一直阻塞消费者线程直到队列可用
超时退出	当阻塞队列满时，队列会阻塞生产者线程一定时间，超过限时后生产者线程会退出

代码测试（抛出异常）：

```
package com.kuang;

import java.util.concurrent.ArrayBlockingQueue;

public class BlockingQueueDemo {
    public static void main(String[] args) {
        // 队列大小
        ArrayBlockingQueue blockingQueue = new ArrayBlockingQueue<>(3);

        System.out.println(blockingQueue.add("a"));
        System.out.println(blockingQueue.add("b"));
        System.out.println(blockingQueue.add("c"));
        System.out.println(blockingQueue.add("d")); //
java.lang.IllegalStateException: Queue full

    }
}
```

```
package com.kuang;

import java.util.concurrent.ArrayBlockingQueue;

public class BlockingQueueDemo {
    public static void main(String[] args) {
        // 队列大小
        ArrayBlockingQueue blockingQueue = new ArrayBlockingQueue<>(3);

        System.out.println(blockingQueue.add("a"));
        System.out.println(blockingQueue.add("b"));
        System.out.println(blockingQueue.add("c"));

        System.out.println(blockingQueue.element()); // 检测队列队首元素！

        // public E remove()  返回值E，就是移除的值
        System.out.println(blockingQueue.remove()); //a
        System.out.println(blockingQueue.remove()); //b
        System.out.println(blockingQueue.remove()); //c

        System.out.println(blockingQueue.remove()); //
java.util.NoSuchElementException

    }
}
```

代码测试（返回特殊值）：

```
public class BlockingQueueDemo {
    public static void main(String[] args) {
        // 队列大小
```

```
        ArrayBlockingQueue blockingQueue = new ArrayBlockingQueue<>(3);

        System.out.println(blockingQueue.offer("a")); // true
        System.out.println(blockingQueue.offer("b")); // true
        System.out.println(blockingQueue.offer("c")); // true
        //System.out.println(blockingQueue.offer("d")); // false

        System.out.println(blockingQueue.peek()); // 检测队列队首元素!

        // public E poll()
        System.out.println(blockingQueue.poll()); // a
        System.out.println(blockingQueue.poll()); // b
        System.out.println(blockingQueue.poll()); // c
        System.out.println(blockingQueue.poll()); // null

    }
}
```

代码测试（一直阻塞）：

```
public class BlockingQueueDemo {
    public static void main(String[] args) throws InterruptedException {
        // 队列大小
        ArrayBlockingQueue blockingQueue = new ArrayBlockingQueue<>(3);

        // 一直阻塞
        blockingQueue.put("a");
        blockingQueue.put("b");
        blockingQueue.put("c");
        // blockingQueue.put("d");

        System.out.println(blockingQueue.take()); // a
        System.out.println(blockingQueue.take()); // b
        System.out.println(blockingQueue.take()); // c
        System.out.println(blockingQueue.take()); // 阻塞不停止等待
    }
}
```

代码测试（超时退出）：

```
public class BlockingQueueDemo {
    public static void main(String[] args) throws InterruptedException {
        // 队列大小
        ArrayBlockingQueue blockingQueue = new ArrayBlockingQueue<>(3);

        // 一直阻塞
        blockingQueue.offer("a");
        blockingQueue.offer("b");
        blockingQueue.offer("c");
        blockingQueue.offer("d",3L,TimeUnit.SECONDS); // 等待3秒超时退出

        System.out.println(blockingQueue.poll()); // a
        System.out.println(blockingQueue.poll()); // b
        System.out.println(blockingQueue.poll()); // c
        System.out.println(blockingQueue.poll(3L,TimeUnit.SECONDS)); // 阻塞
不停止等待
    }
}
```

> SynchronousQueue 同步队列

SynchronousQueue 没有容量。

与其他的 BlockingQueue 不同，SynchronousQueue是一个不存储元素的 BlockingQueue 。

每一个put操作必须要等待一个take操作，否则不能继续添加元素，反之亦然。

```
package com.kuang;

import jdk.nashorn.internal.ir.Block;

import java.util.concurrent.BlockingQueue;
import java.util.concurrent.SynchronousQueue;
import java.util.concurrent.TimeUnit;

public class SynchronousQueueDemo {
    public static void main(String[] args) {
        BlockingQueue<String> blockingQueue = new SynchronousQueue<>();

        new Thread(()->{
            try {
                System.out.println(Thread.currentThread().getName()+" put 1");
                blockingQueue.put("1");
                System.out.println(Thread.currentThread().getName()+" put 2");
                blockingQueue.put("2");
                System.out.println(Thread.currentThread().getName()+" put 3");
                blockingQueue.put("3");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        },"T1").start();

        new Thread(()->{
            try {
                TimeUnit.SECONDS.sleep(3);
 System.out.println(Thread.currentThread().getName()+blockingQueue.take());
                TimeUnit.SECONDS.sleep(3);
 System.out.println(Thread.currentThread().getName()+blockingQueue.take());
                TimeUnit.SECONDS.sleep(3);
 System.out.println(Thread.currentThread().getName()+blockingQueue.take());
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        },"T1").start();
    }
}
```

11、线程池

> 池化技术

程序的运行，其本质上，是对系统资源(CPU、内存、磁盘、网络等等)的使用。如何高效的使用这些资源是我们编程优化演进的一个方向。今天说的线程池就是一种对CPU利用的优化手段。

通过学习线程池原理，明白所有池化技术的基本设计思路。遇到其他相似问题可以解决。

池化技术

前面提到一个名词——池化技术，那么到底什么是池化技术呢？

池化技术简单点来说，就是提前保存大量的资源，以备不时之需。在机器资源有限的情况下，使用池化技术可以大大的提高资源的利用率，提升性能等。

在编程领域，比较典型的池化技术有：

线程池、连接池、内存池、对象池等。

主要来介绍一下其中比较简单的线程池的实现原理，希望读者们可以举一反三，通过对线程池的理解，学习并掌握所有编程中池化技术的底层原理。

我们通过创建一个线程对象，并且实现Runnable接口就可以实现一个简单的线程。可以利用上多核CPU。当一个任务结束，当前线程就接收。

但很多时候，我们不止会执行一个任务。如果每次都是如此的创建线程->执行任务->销毁线程，会造成很大的性能开销。

那能否一个线程创建后，执行完一个任务后，又去执行另一个任务，而不是销毁。这就是线程池。

这也就是池化技术的思想，通过预先创建好多个线程，放在池中，这样可以在需要使用线程的时候直接获取，避免多次重复创建、销毁带来的开销。

> 为什么使用线程池

10 年前单核CPU电脑，假的多线程，像马戏团小丑玩多个球 ，CPU 需要来回切换。

现在是多核电脑，多个线程各自跑在独立的CPU上，不用切换效率高。

线程池的优势：

线程池做的工作主要是：控制运行的线程数量，处理过程中将任务放入队列，然后在线程创建后启动这些任务，如果线程数量超过了最大数量，超出数量的线程排队等候，等其他线程执行完毕，再从队列中取出任务来执行。

它的主要特点为：线程复用，控制最大并发数，管理线程。

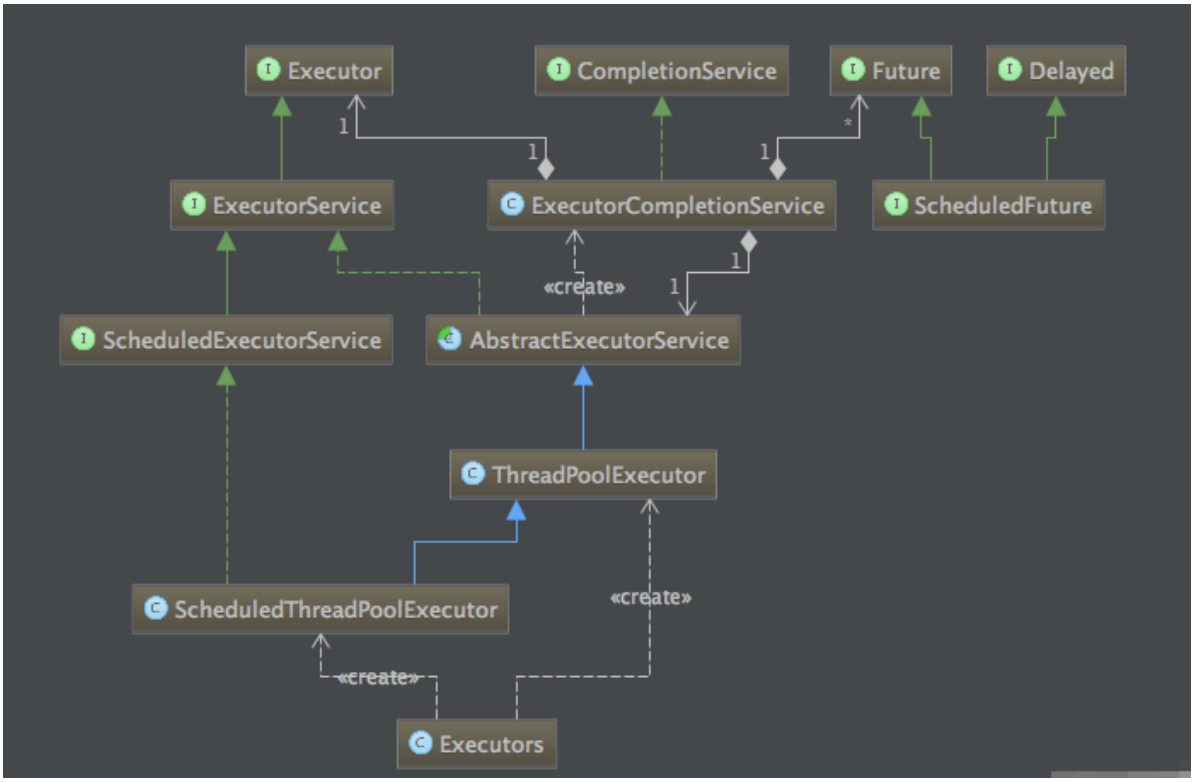
第一：降低资源消耗，通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

第二：提高响应速度。当任务到达时，任务可以不需要等待线程创建就能立即执行。

第三：提高线程的可管理性，线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一分配，调优和监控。

> 线程池的三大方法

Java中的线程池是通过 Executor 框架实现的，该框架中用到了 Executor ， Executors，ExecutorService，ThreadPoolExecutor 这几个类。



三大方法说明：

- `Executors.newFixedThreadPool(int)`
  - 执行长期任务性能好，创建一个线程池，一池有N个固定的线程，有固定线程数的线程。

```
public class MyThreadPoolDemo {
    public static void main(String[] args) {
        // 池子大小 5
        ExecutorService threadPool =
Executors.newFixedThreadPool(5);

        try {
            // 模拟有10个顾客过来银行办理业务，池子中只有5个工作人员受理业务
            for (int i = 1; i <= 10; i++) {
                threadPool.execute(()->{

 System.out.println(Thread.currentThread().getName()+" 办理业务");
                });
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            threadPool.shutdown(); // 用完记得关闭
        }

    }
}
```

- `Executors.newSingleThreadExecutor()`

- 只有一个线程

```
public class MyThreadPoolDemo {
    public static void main(String[] args) {
        // 有且只有一个固定的线程
        ExecutorService threadPool =
Executors.newSingleThreadExecutor();

        try {
            // 模拟有10个顾客过来银行办理业务，池子中只有1个工作人员受理业务
            for (int i = 1; i <= 10; i++) {
                threadPool.execute(()->{

 System.out.println(Thread.currentThread().getName()+" 办理业务");
                });
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            threadPool.shutdown(); // 用完记得关闭
        }

    }
}
```

- `Executors.newCachedThreadPool();`
  - 执行很多短期异步任务，线程池根据需要创建新线程，但在先构建的线程可用时将重用他们。可扩容，遇强则强

```
public class MyThreadPoolDemo {
    public static void main(String[] args) {
        // 一池N线程，可扩容伸缩
        ExecutorService threadPool =
Executors.newCachedThreadPool();

        try {
            // 模拟有10个顾客过来银行办理业务，池子中只有N个工作人员受理业务
            for (int i = 1; i <= 10; i++) {
                // 模拟延时看效果
                // try {
                //      TimeUnit.SECONDS.sleep(1);
                // } catch (InterruptedException e) {
                //      e.printStackTrace();
                //  }
                threadPool.execute(()->{

 System.out.println(Thread.currentThread().getName()+" 办理业务");
                });
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            threadPool.shutdown(); // 用完记得关闭
        }

    }
}
```

> ThreadPoolExecutor 七大参数

操作：查看三大方法的底层源码，发现本质都是调用了 `new ThreadPoolExecutor ( 7 大参数 )`

```
// 源码
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) {
    if (corePoolSize < 0 ||
        maximumPoolSize <= 0 ||
        maximumPoolSize < corePoolSize ||
        keepAliveTime < 0)
        throw new IllegalArgumentException();
    if (workQueue == null || threadFactory == null || handler == null)
        throw new NullPointerException();
    this.acc = System.getSecurityManager() == null ?
        null :
    AccessController.getContext();
    this.corePoolSize = corePoolSize;
    this.maximumPoolSize = maximumPoolSize;
    this.workQueue = workQueue;
    this.keepAliveTime = unit.toNanos(keepAliveTime);
    this.threadFactory = threadFactory;
    this.handler = handler;
}
```

参数理解：

`corePollSize`：核心线程数。在创建了线程池后，线程中没有任何线程，等到有任务到来时才创建线程去执行任务。默认情况下，在创建了线程池后，线程池中的线程数为0，当有任务来之后，就会创建一个线程去执行任务，当线程池中的线程数目达到corePoolSize后，就会把到达的任务放到缓存队列当中。

`maximumPoolSize`：最大线程数。表明线程中最多能够创建的线程数量，此值必须大于等于1。

`keepAliveTime`：空闲的线程保留的时间。

`TimeUnit`：空闲线程的保留时间单位。

```
TimeUnit.DAYS;               //天
TimeUnit.HOURS;              //小时
TimeUnit.MINUTES;            //分钟
TimeUnit.SECONDS;            //秒
TimeUnit.MILLISECONDS;       //毫秒
TimeUnit.MICROSECONDS;       //微妙
TimeUnit.NANOSECONDS;        //纳秒
```

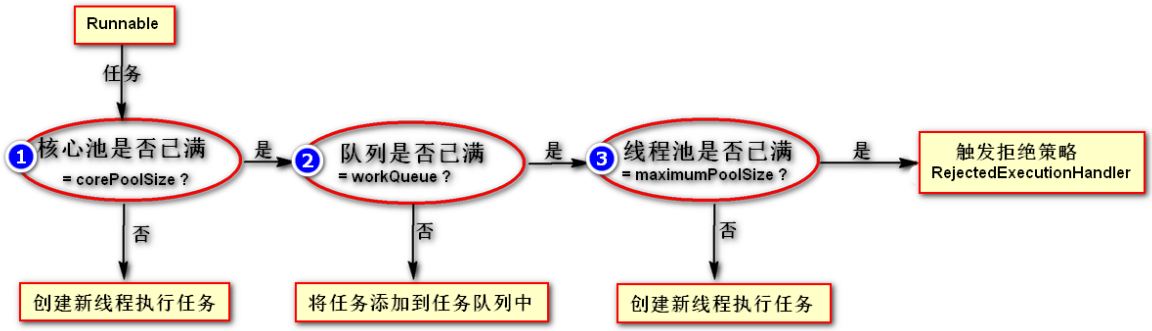
`BlockingQueue< Runnable>`：阻塞队列，存储等待执行的任务。参数有ArrayBlockingQueue、LinkedBlockingQueue、SynchronousQueue可选。

`ThreadFactory`：线程工厂，用来创建线程，一般默认即可

`RejectedExecutionHandler`：队列已满，而且任务量大于最大线程的异常处理策略。有以下取值

```
ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。
ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。
ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）
ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务
```

> ThreadPoolExecutor 底层工作原理



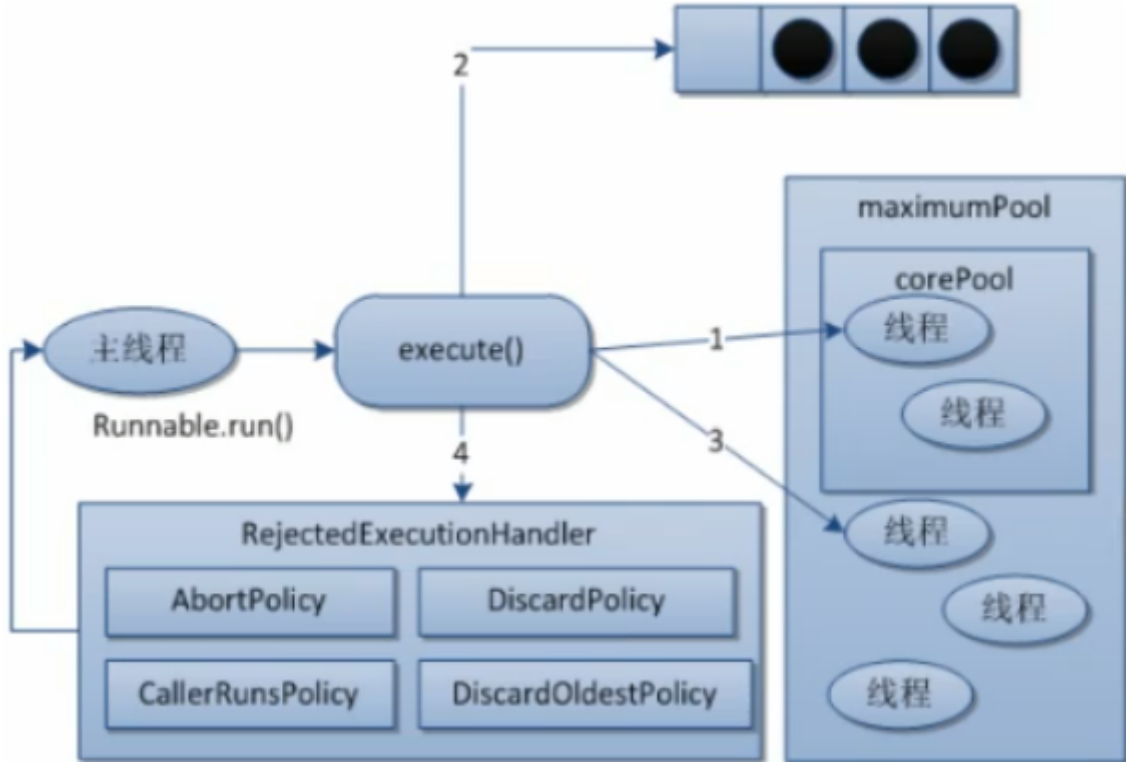
举例：8个人进银行办理业务

1、1~2人被受理（核心大小core）

2、3~5人进入队列（Queue）

3、6~8人到最大线程池（扩容大小max）

4、再有人进来就要被拒绝策略接受了



1. 在创建了线程池后，开始等待请求。
2. 当调用execute()方法添加一个请求任务时，线程池会做出如下判断：
    1. 如果正在运行的线程数量小于corePoolSize，那么马上创建线程运行这个任务：

2. 如果正在运行的线程数量大于或等于corePoolSize，那么将这个任务放入队列：
3. 如果这个时候队列满了且正在运行的线程数量还小于maximumPoolSize，那么还是要创建非核心线程立刻运行这个任务；
4. 如果队列满了且正在运行的线程数量大于或等于1Size，那么线程池会启动饱和拒绝策略来执行。

3. 当一个线程完成任务时，它会从队列中取下一个任务来执行。
4. 当一个线程无事可做超过一定的时间(keepA1iveTime)时，线程会判断：
   - 如果当前运行的线程数大于coreP佣1Size，那么这个线程就被停掉。
   - 所以线程池的所有任务完成后，它最终会收缩到 corePoolSize 的大小。

> 线程池用哪个？生产中如何设置合理参数

在工作中单一的/固定数的/可变的三种创建线程池的方法哪个用的多？坑

答案是一个都不用，我们工作中只能使用自定义的；

Executors 中 JDK 已经给你提供了，为什么不用？

> 代码测试

线程池的拒绝策略：

```
RejectedExecutionHandler rejected = null;
rejected = new ThreadPoolExecutor.AbortPolicy();//默认，队列满了丢任务，抛出异常
rejected = new ThreadPoolExecutor.DiscardPolicy();//队列满了丢任务，不抛出异常【如果允许任务丢失这是最好的】
rejected = new ThreadPoolExecutor.DiscardOldestPolicy();//将最早进入队列的任务删，之后再尝试加入队列
rejected = new ThreadPoolExecutor.CallerRunsPolicy();//如果添加到线程池失败，那么主线程会自己去执行该任务，回退
```

测试代码

```
public class MyThreadPoolDemo {
    public static void main(String[] args) {

        // 获得CPU的内核数
        System.out.println(Runtime.getRuntime().availableProcessors());

        // 自定义 ThreadPoolExecutor
```

```
        ExecutorService threadPool = new ThreadPoolExecutor(
                2,
                5,
                2L,
                TimeUnit.SECONDS,
                new LinkedBlockingDeque<>(3),
                Executors.defaultThreadFactory(),
                new ThreadPoolExecutor.DiscardPolicy());

        try {
            // 模拟有6,7,8,9,10个顾客过来银行办理业务，观察结果情况
            // 最大容量为：maximumPoolSize + workQueue = 最大容量数
            for (int i = 1; i <= 19; i++) {
                threadPool.execute(()->{
                    System.out.println(Thread.currentThread().getName()+" 办理业务");
                });
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            threadPool.shutdown(); // 用完记得关闭
        }

    }
}
```

思考题：线程是否越多越好?

一个计算为主的程序（专业一点称为**CPU密集型程序**）。多线程跑的时候，可以充分利用起所有的cpu核心，比如说4个核心的cpu,开4个线程的时候，可以同时跑4个线程的运算任务，此时是最大效率。

但是如果线程远远超出cpu核心数量 反而会使得任务效率下降，因为频繁的切换线程也是要消耗时间的。

因此对于cpu密集型的任务来说，线程数等于cpu数是最好的了。

如果是一个磁盘或网络为主的程序（**IO密集型**）。一个线程处在IO等待的时候，另一个线程还可以在CPU里面跑，有时候CPU闲着没事干，所有的线程都在等着IO，这时候他们就是同时的了，而单线程的话此时还是在一个一个等待的。我们都知道IO的速度比起CPU来是慢到令人发指的。所以开多线程，比方说多线程网络传输，多线程往不同的目录写文件，等等。

此时 线程数等于IO任务数是最佳的。

12、四大函数式接口

java.util.function , Java 内置核心四大函数式接口，可以使用lambda表达式

函数式接口	参数类型	返回类型	用途
Consumer<T> 消费型接口	T	void	对类型为T的对象应用操作，包含方法： void accept(T t)
Supplier<T> 供给型接口	无	T	返回类型为T的对象，包含方法：T get();
Function<T, R> 函数型接口	T	R	对类型为T的对象应用操作，并返回结果。结果是R类型的对象。包含方法：R apply(T t);
Predicate<T> 断定型接口	T	boolean	确定类型为T的对象是否满足某约束，并返回 boolean 值。包含方法 boolean test(T t);

> 函数型接口，有一个输入，有一个输出

```
public static void main(String[] args) {
    // 函数式接口，可以改为 lambda 表达式
    //Function<String,Integer> function = new Function<String, Integer>() {
    //    @Override
    //    public Integer apply(String s) {
    //        return 1024;
    //    }
    //};

    // 简写
    Function<String,Integer> function = s->{return s.length();};
    System.out.println(function.apply("abc"));

}
```

> 断定型接口，有一个输入参数，返回只有布尔值。

```
public static void main(String[] args) {

    //Predicate<String> predicate = new Predicate<String>() {
    //    @Override
    //    public boolean test(String s) {
    //        return false;
    //    }
    //};

    // 简写
    Predicate<String> predicate = s -> {return s.isEmpty();};
    System.out.println(predicate.test("abc"));

}
```

> 消费型接口，有一个输入参数，没有返回值

```
public static void main(String[] args) {

    //        Consumer<String> consumer = new Consumer<String>() {
    //            @Override
    //            public void accept(String s) {
    //
    //            }
    //        };

    // 简写
    Consumer<String> consumer = s -> { System.out.println(s);};
    consumer.accept("abc");

}
```

> 供给型接口，没有输入参数，只有返回参数

```
public static void main(String[] args) {

    //    Supplier<String> supplier = new Supplier<String>() {
    //        @Override
    //        public String get() {
    //            return null;
    //        }
    //    };
    Supplier<String> supplier = ()->{return "abc";};
    System.out.println(supplier.get());

}
```

13、Stream流式计算

链式编程、流式计算、lambda表达式，现在的 Java程序员必会！

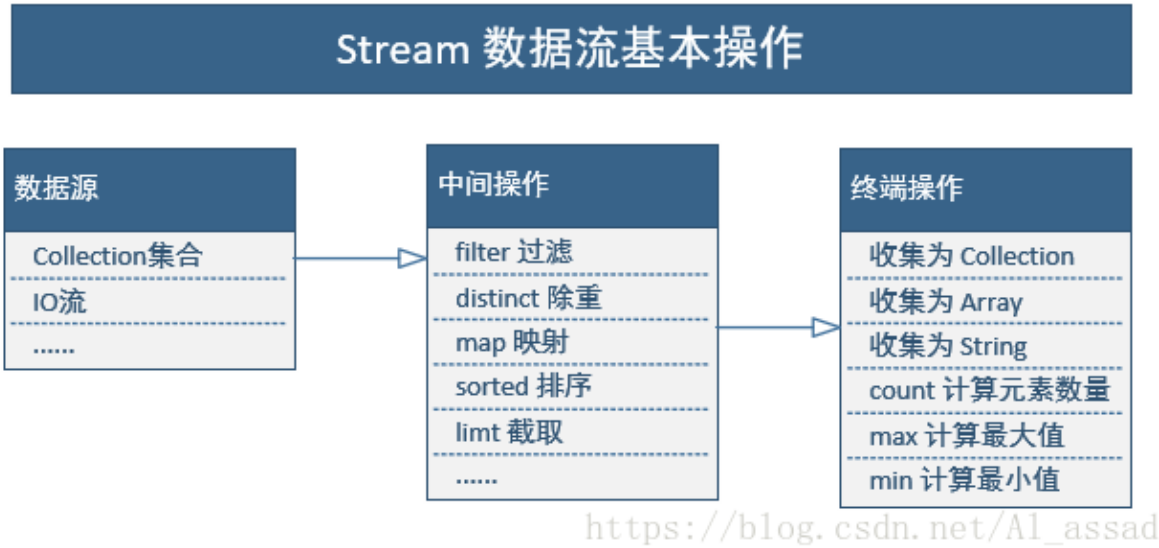
> 流（Stream）到底是什么呢?

是数据渠道，用于操作数据源（集合、数组等）所生成的元素序列。

“集合讲的是数据，流讲的是计算！”

特点:

- Stream 自己不会存储元素。
- Stream 不会改变源对象，相反，他们会返回一个持有结果的新Stream。
- Stream 操作是延迟执行的。这意味着他们会等到需要结果的时候才执行。



> 代码验证

User实体类

```java
public class User {
    private int id;
    private String userName;
    private int age;
    //get、set、有参/无参构造器、toString
}
```

Stream算法题

```java
import java.util.Arrays;
import java.util.List;

/*
 * 题目：请按照给出数据，找出同时满足以下条件的用户
 *        也即以下条件：
 *           1、全部满足偶数ID
 *           2、年龄大于24
 *           3、用户名转为大写
 *           4、用户名字母倒排序
 *           5、只输出一个用户名字 limit
 **/
public class StreamDemo {
    public static void main(String[] args) {

        User u1 = new User(11, "a", 23);
        User u2 = new User(12, "b", 24);
        User u3 = new User(13, "c", 22);
        User u4 = new User(14, "d", 28);
        User u5 = new User(16, "e", 26);

        List<User> list = Arrays.asList(u1, u2, u3, u4, u5);

        /*
         * 1. 首先我们需要将 list 转化为stream流
         * 2. 然后将用户过滤出来，这里用到一个函数式接口Predicate<? super T>，我们可以使用lambda表达式简化
         * 3. 这里面传递的参数，就是Stream流的泛型类型，也就是User，所以，这里可以直接返回用户id为偶数的用户信息;
         * 4. 通过forEach进行遍历，直接简化输出 System.out::println ，等价于System.out.println(u);
         **/

        //list.stream().filter(u -> {return u.getId()%2==0;}).forEach(System.out::println);
        //list.stream().filter(u -> {return u.getId()%2==0;})
            //.filter(u -> {return u.getAge()>24;}).forEach(System.out::println);

        //sorted()  自然排序，正排序 D->E

        list.stream()
                .filter(u -> {return u.getId()%2==0;})
                .filter(u -> {return u.getAge()>24;})
                .map(u -> {return u.getUserName().toUpperCase();})
               //.sorted() //默认正排序  自己用 compareTo 比较
                .sorted((o1,o2)->{return o2.compareTo(o1);})
                .limit(1)
```

```
                .forEach(System.out::println);

        /*
        map解释
        List<Integer> list2 = Arrays.asList(1,2,3);
        list2 = list2.stream().map(x -> {return
x*2;}).collect(Collectors.toList());

        for (Integer element : list2) {
            System.out.println(element);
        }
        */

    }
}
```

14、分支合并

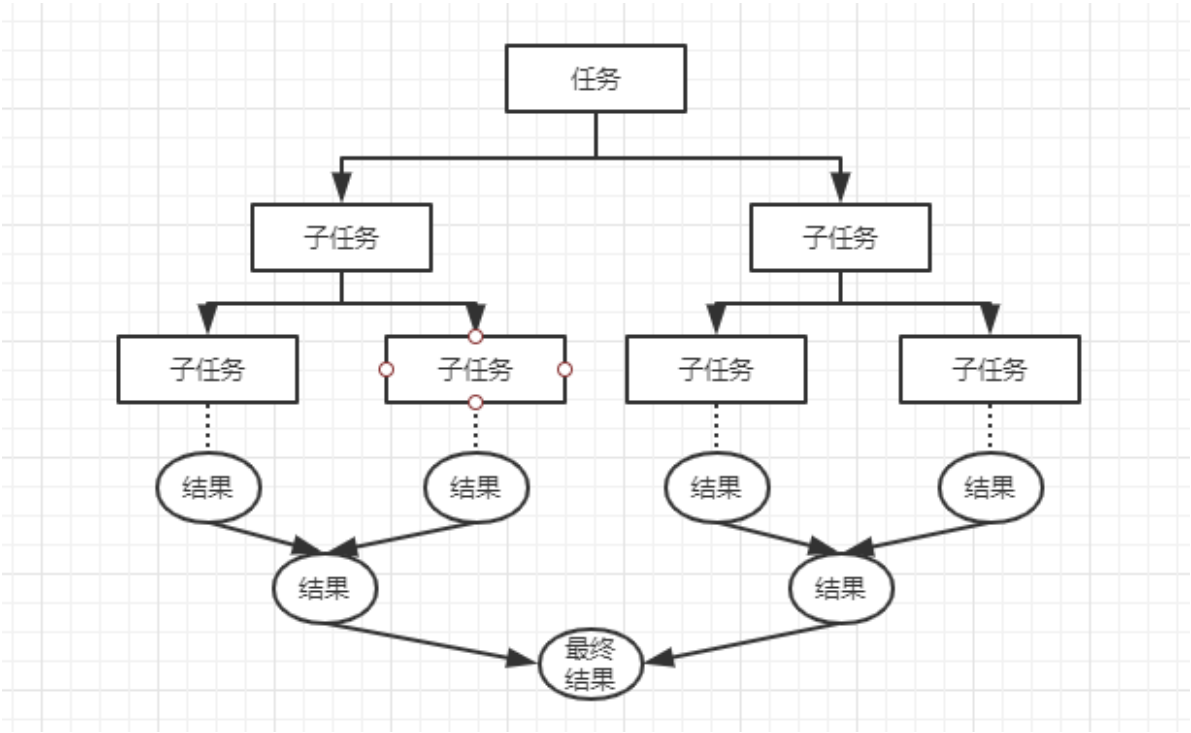
> 什么是ForkJoin

从JDK1.7开始，Java提供Fork/Join框架用于并行执行任务，它的思想就是讲一个大任务分割成若干小任务，最终汇总每个小任务的结果得到这个大任务的结果。

这种思想和MapReduce很像（input --> split --> map --> reduce --> output）

主要有两步：

- 第一、任务切分；
- 第二、结果合并

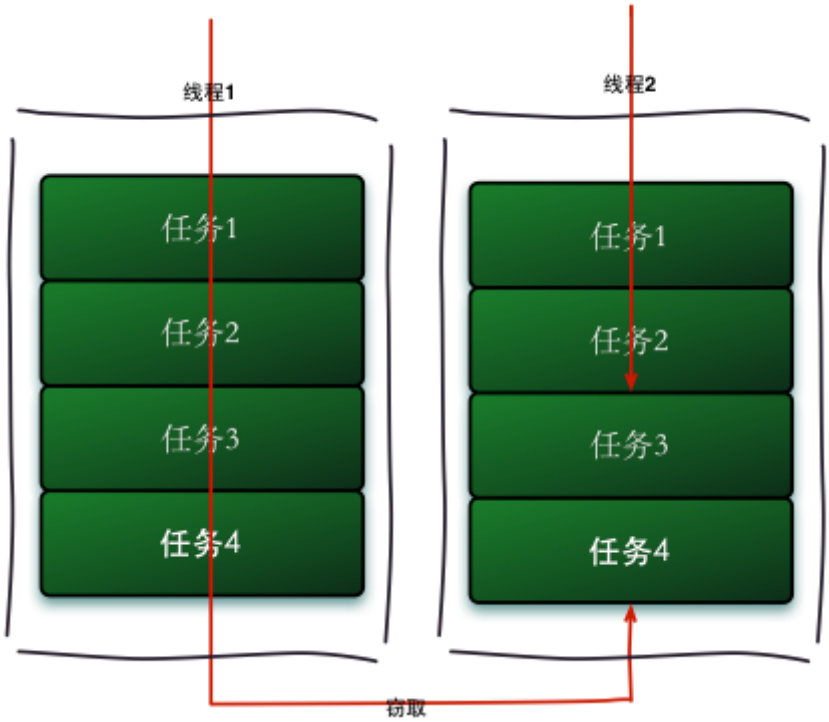


它的模型大致是这样的：线程池中的每个线程都有自己的工作队列（PS：这一点和ThreadPoolExecutor不同，ThreadPoolExecutor是所有线程公用一个工作队列，所有线程都从这个工作队列中取任务），当自己队列中的任务都完成以后，会从其它线程的工作队列中偷一个任务执行，这样可以充分利用资源。

> 工作窃取

另外，forkjoin有一个工作窃取的概念。简单理解，就是一个工作线程下会维护一个包含多个子任务的双端队列。而对于每个工作线程来说，会从头部到尾部依次执行任务。这时，总会有一些线程执行的速度较快，很快就把所有任务消耗完了。那这个时候怎么办呢，总不能空等着吧，多浪费资源啊。

工作窃取（work-stealing）算法是指某个线程从其他队列里窃取任务来执行。工作窃取的运行流程图如下：



那么为什么需要使用工作窃取算法呢?

假如我们需要做一个比较大的任务，我们可以把这个任务分割为若干互不依赖的子任务，为了减少线程间的竞争，于是把这些子任务分别放到不同的队列里，并为每个队列创建一个单独的线程来执行队列里的任务，线程和队列一一对应，比如A线程负责处理A队列里的任务。但是有的线程会先把自己队列里的任务干完，而其他线程对应的队列里还有任务等待处理。干完活的线程与其等着，不如去帮其他线程干活，于是它就去其他线程的队列里窃取一个任务来执行。而在这时它们会访问同一个队列，所以为了减少窃取任务线程和被窃取任务线程之间的竞争，通常会使用双端队列，被窃取任务线程永远从双端队列的头部拿任务执行，而窃取任务的线程永远从双端队列的尾部拿任务执行。

工作窃取算法的优点是充分利用线程进行并行计算，并减少了线程间的竞争，其缺点是在某些情况下还是存在竞争，比如双端队列里只有一个任务时。并且消耗了更多的系统资源，比如创建多个线程和多个双端队列。

于是，先做完任务的工作线程会从其他未完成任务的线程尾部依次获取任务去执行。这样就可以充分利用CPU的资源。这个非常好理解，就比如有个妹子程序员做任务比较慢，那么其他猿就可以帮她分担一些任务，这简直是双赢的局面啊，妹子开心了，你也开心了。

> 核心类

ForkJoinPool

WorkQueue是一个ForkJoinPool中的内部类，它是线程池中线程的工作队列的一个封装，支持任务窃取。

什么叫线程的任务窃取呢？就是说你和你的一个伙伴一起吃水果，你的那份吃完了，他那份没吃完，那你就偷偷的拿了他的一些水果吃了。存在执行2个任务的子线程，这里要讲成存在A,B两个个WorkQueue在执行任务，A的任务执行完了，B的任务没执行完，那么A的WorkQueue就从B的WorkQueue的ForkJoinTask数组中拿走了一部分尾部的任务来执行，可以合理的提高运行和计算效率。

每个线程都有一个WorkQueue，而WorkQueue中有执行任务的线程（ForkJoinWorkerThread owner），还有这个线程需要处理的任务（ForkJoinTask<?>[] array）。那么这个新提交的任务就是加到array中。

ForkJoinTask

ForkJoinTask代表运行在ForkJoinPool中的任务。

主要方法：

- fork()　在当前线程运行的线程池中安排一个异步执行。简单的理解就是再创建一个子任务。
- join()　当任务完成的时候返回计算结果。
- invoke()　开始执行任务，如果必要，等待计算完成。

子类：　**Recursive：递归**

- RecursiveAction　一个递归无结果的ForkJoinTask（没有返回值）
- RecursiveTask　一个递归有结果的ForkJoinTask（有返回值）

> 代码验证

核心代码

```
package FORKJOIN;

import java.util.concurrent.RecursiveTask;

public class ForkJoinWork extends RecursiveTask<Long> {

    private Long start;//起始值
    private Long end;//结束值
    public static final  Long critical = 10000L;//临界值

    public ForkJoinWork(Long start, Long end) {
        this.start = start;
        this.end = end;
    }

    @Override
    protected Long compute() {
        //判断是否是拆分完毕
        Long lenth = end - start;
        if(lenth<=critical){
            //如果拆分完毕就相加
            Long sum = 0L;
            for (Long i = start;i<=end;i++){
                sum += i;
            }
            return sum;
        }else {
```

```
            //没有拆分完毕就开始拆分
            Long middle = (end + start)/2;//计算的两个值的中间值
            ForkJoinWork right = new ForkJoinWork(start,middle);
            right.fork();//拆分，并压入线程队列
            ForkJoinWork left = new ForkJoinWork(middle+1,end);
            left.fork();//拆分，并压入线程队列

            //合并
            return right.join() + left.join();
        }
    }
}
```

三种测试

```
package com.kuang;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ForkJoinTask;
import java.util.stream.LongStream;

public class ForkJoinWorkDemo {

    public static void main(String[] args) throws ExecutionException, InterruptedException {
        test(); //15756  // 14414  // 203
    }

    // forkjoin这个框架针对的是大任务执行，效率才会明显的看出来有提升，于是我把总数调大到20亿。
    public static void test() throws ExecutionException, InterruptedException {
        //ForkJoin实现
        long l = System.currentTimeMillis();

        ForkJoinPool forkJoinPool = new ForkJoinPool();//实现ForkJoin 就必须有ForkJoinPool的支持
        ForkJoinTask<Long> task = new ForkJoinWork(0L,2000000000L);//参数为起始值与结束值
        ForkJoinTask<Long> result = forkJoinPool.submit(task);
        Long aLong = result.get();

        long l1 = System.currentTimeMillis();
        System.out.println("invoke = " + aLong +"  time: " + (l1-l));
    }

    public static void test2(){
        //普通线程实现
        Long x = 0L;
        Long y = 2000000000L;
        long l = System.currentTimeMillis();
        for (Long i = 0L; i <= y; i++) {
            x+=i;
        }
        long l1 = System.currentTimeMillis();
        System.out.println("invoke = " + x+"  time: " + (l1-l));

```

```
    }

    public static void test3(){
        //Java 8 并行流的实现
        long l = System.currentTimeMillis();
        long reduce = LongStream.rangeClosed(0,
2000000000L).parallel().reduce(0, Long::sum);
        long l1 = System.currentTimeMillis();
        System.out.println("invoke = " + reduce+"  time: " + (l1-l));

    }

}
```

打个比方，假设一个酒店有400个房间，一共有4名清洁工，每个工人每天可以打扫100个房间，这样，4个工人满负荷工作时，400个房间全部打扫完正好需要1天。

Fork/Join的工作模式就像这样：首先，工人甲被分配了400个房间的任务，他一看任务太多了自己一个人不行，所以先把400个房间拆成两个200，然后叫来乙，把其中一个200分给乙。

紧接着，甲和乙再发现200也是个大任务，于是甲继续把200分成两个100，并把其中一个100分给丙，类似的，乙会把其中一个100分给丁，这样，最终4个人每人分到100个房间，并发执行正好是1天。

15、异步回调

> 概述

Future设计的初衷：对将来某个时刻会发生的结果进行建模。

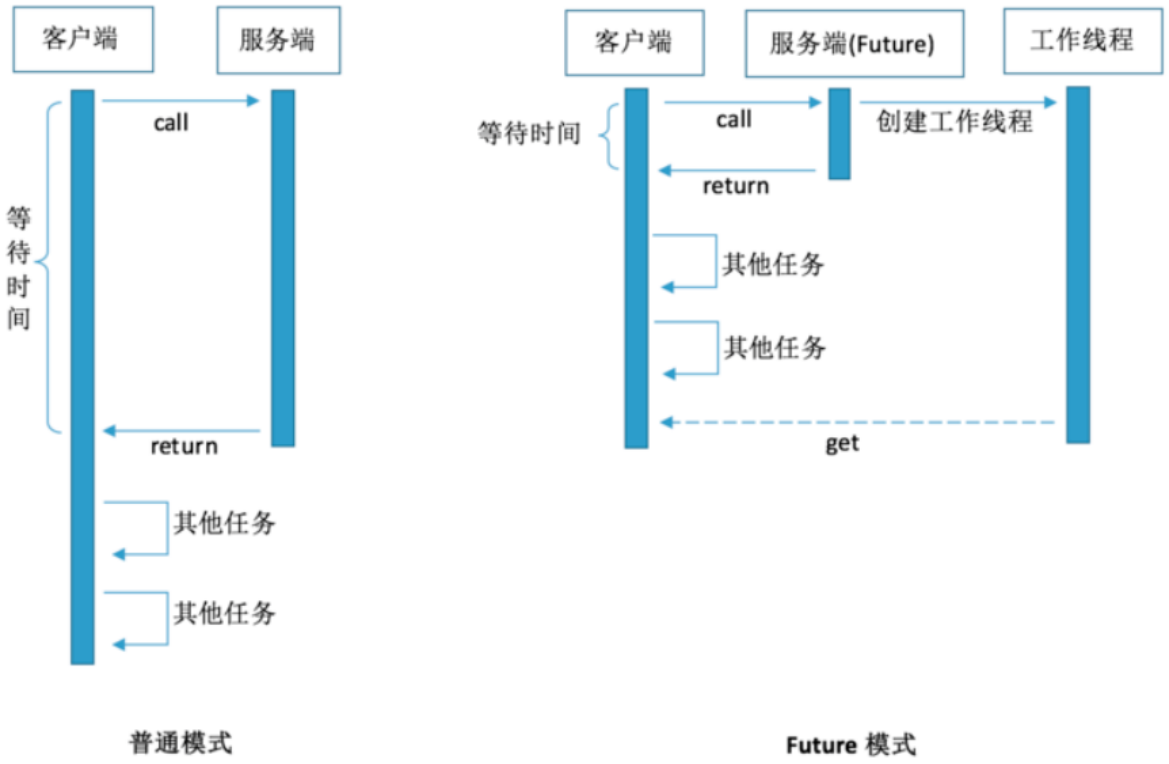
当我们需要调用一个函数方法时。如果这个函数执行很慢,那么我们就要进行等待。但有时候,我们可能并不急着要结果。

因此,我们可以让被调用者立即返回,让他在后台慢慢处理这个请求。对于调用者来说,则可以先处理一些其他任务,在真正需要数据的场合再去尝试获取需要的数据。

它建模了一种异步计算，返回一个执行运算结果的引用，当运算结束后，这个引用被返回给调用方。在Future中出发那些潜在耗时的操作把调用线程解放出来，让它能继续执行其他有价值的工作，不再需要等待耗时的操作完成。

Future的优点：比更底层的Thread更易用。要使用Future，通常只需要将耗时的操作封装在一个Callable对象中，再将它提交给ExecutorService。

为了让程序更加高效，让CPU最大效率的工作，我们会采用异步编程。首先想到的是开启一个新的线程去做某项工作。再进一步，为了让新线程可以返回一个值，告诉主线程事情做完了，于是乎Future粉墨登场。然而Future提供的方式是主线程主动问询新线程，要是有个回调函数就爽了。所以，为了满足Future的某些遗憾，强大的CompletableFuture随着Java8一起来了。



普通模式　　Future 模式

实例

```
package com.kuang;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.TimeUnit;

public class CompletableFutureDemo {
    public static void main(String[] args) throws Exception {
        //没有返回值的 runAsync 异步调用
        CompletableFuture<Void> completableFuture =
CompletableFuture.runAsync(() -> {
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName() + "没有返回,
update mysql ok");
        });
        System.out.println("111111"); // 先执行
        completableFuture.get();

        //有返回值的  供给型参数接口
        CompletableFuture<Integer> completableFuture2 =
CompletableFuture.supplyAsync(() -> {
            System.out.println(Thread.currentThread().getName() +
"completableFuture2");
            int i = 10/0;
            return 1024;
        });

```

```
27          System.out.println(completableFuture2.whenComplete((t, u) -> { //编译
    完成，正常结束输出
28              System.out.println("===t:" + t);  //正常结果
29              System.out.println("===u:" + u);  //报错的信息
30          }).exceptionally(e -> {  //结果异常，非正常结束
31              System.out.println("=======exception:" + e.getMessage());
32              return 555;
33          }).get());
34
35      }
36  }
```

16、JMM

> 问题：请你谈谈你对volatile的理解

volitile 是 Java 虚拟机提供的轻量级的同步机制，三大特性：

1、保证可见性

2、不保证原子性

3、禁止指令重排

> 什么是JMM

JMM 本身是一种抽象的概念，并不真实存在，它描述的是一组规则或者规范~

JMM 关于同步的规定：

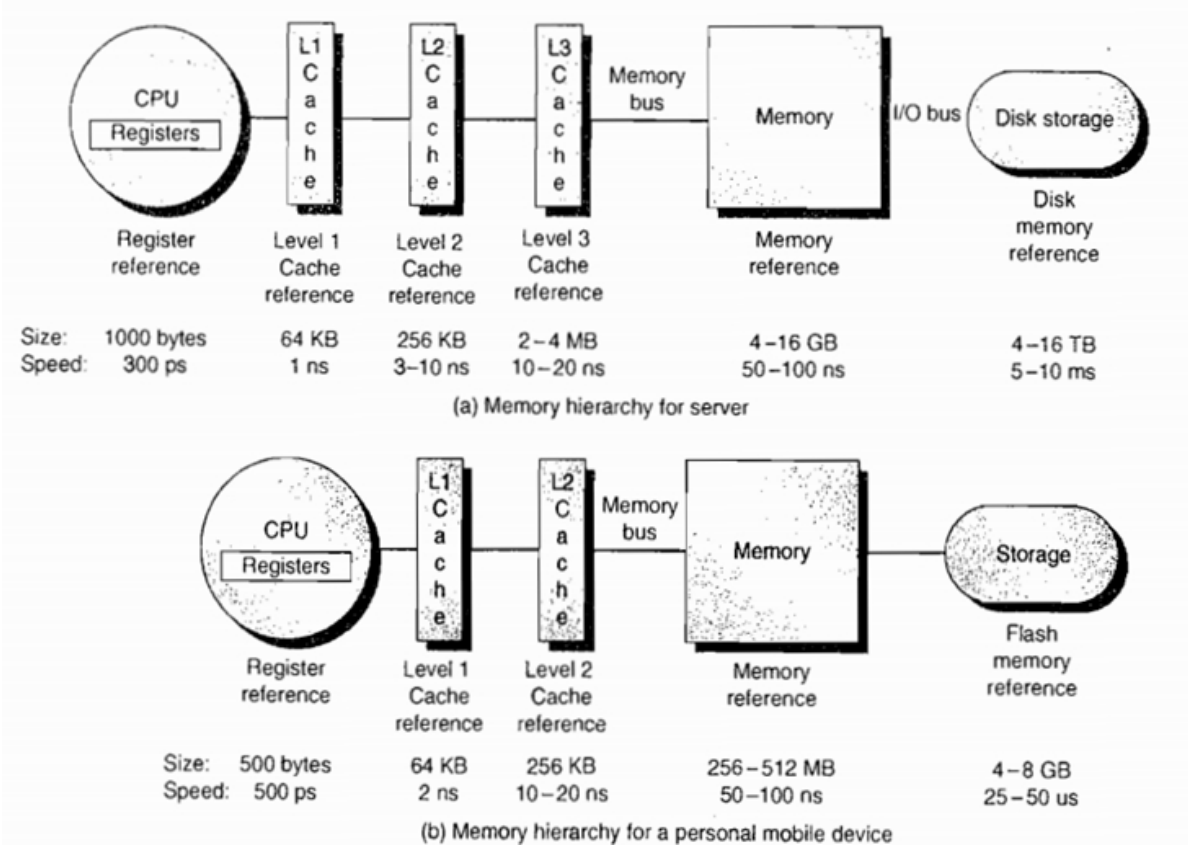
1、线程解锁前，必须把共享变量的值刷新回主内存

2、线程加锁前，必须读取主内存的最新值到自己的工作内存

3、加锁解锁是同一把锁

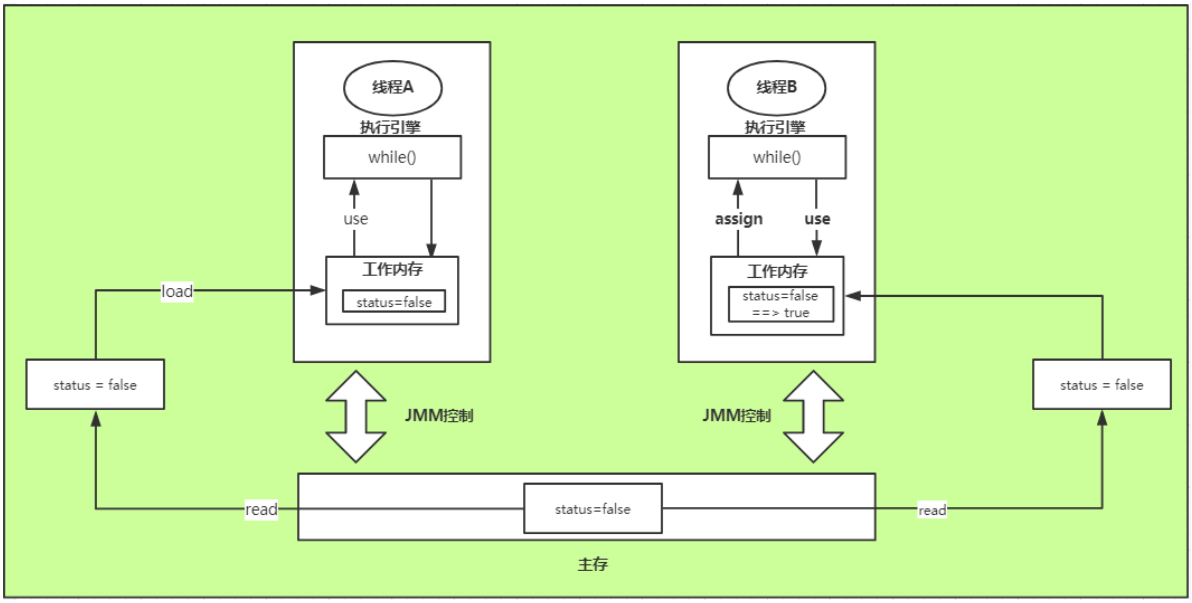
JMM即为JAVA 内存模型（java memory model）。因为在**不同的硬件生产商和不同的操作系统**下，内存的访问逻辑有一定的差异，结果就是当你的代码在某个系统环境下运行良好，并且线程安全，但是换了个系统就出现各种问题。Java内存模型，就是为了屏蔽系统和硬件的差异，让一套代码在不同平台下能到达相同的访问结果。JMM从java 5开始的JSR-133发布后，已经成熟和完善起来。

JMM规定了内存主要划分为**主内存**和**工作内存**两种。此处的主内存和工作内存跟JVM内存划分（堆、栈、方法区）是在不同的层次上进行的，如果非要对应起来，主内存对应的是Java堆中的对象实例部分，工作内存对应的是栈中的部分区域，从更底层的来说，主内存对应的是硬件的物理内存，工作内存对应的是寄存器和高速缓存。



JVM在设计时候考虑到，如果JAVA线程每次读取和写入变量都直接操作主内存，对性能影响比较大，所以每条线程拥有各自的工作内存，工作内存中的变量是主内存中的一份拷贝，线程对变量的读取和写入，直接在工作内存中操作，而不能直接去操作主内存中的变量。但是这样就会出现一个问题，当一个线程修改了自己工作内存中变量，对其他线程是不可见的，会导致线程不安全的问题。因为JMM制定了一套标准来保证开发者在编写多线程程序的时候，能够控制什么时候内存会被同步给其他线程。

> JMM的内存模型



线程A感知不到线程B操作了值的变化！如何能够保证线程间可以同步感知这个问题呢？只需要使用Volatile关键字即可！volatile 保证线程间变量的可见性，简单地说就是当线程A对变量X进行了修改后，在线程A后面执行的其他线程能看到变量X的变动，更详细地说是要符合以下两个规则：

- 线程对变量进行修改之后，要立刻回写到主内存。
- 线程对变量读取的时候，要从主内存中读，而不是缓存。

各线程的工作内存间彼此独立，互不可见，在线程启动的时候，虚拟机为每个内存分配一块工作内存，不仅包含了线程内部定义的局部变量，也包含了线程所需要使用的共享变量（非线程内构造的对象）的副本，即，为了提高执行效率。

内存交互操作

内存交互操作有8种，虚拟机实现必须保证每一个操作都是原子的，不可在分的（对于double和long类型的变量来说，load、store、read和write操作在某些平台上允许例外）

- lock （锁定）：作用于主内存的变量，把一个变量标识为线程独占状态
- unlock （解锁）：作用于主内存的变量，它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）：作用于主内存变量，它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，它把read操作从主存中变量放入工作内存中
- use （使用）：作用于工作内存中的变量，它把工作内存中的变量传输给执行引擎，每当虚拟机遇到一个需要使用到变量的值，就会使用到这个指令
- assign （赋值）：作用于工作内存中的变量，它把一个从执行引擎中接受到的值放入工作内存的变量副本中
- store （存储）：作用于主内存中的变量，它把一个从工作内存中一个变量的值传送到主内存中，以便后续的write使用
- write （写入）：作用于主内存中的变量，它把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

JMM对这八种指令的使用，制定了如下规则：

- 不允许read和load、store和write操作之一单独出现。即使用了read必须load，使用了store必须write
- 不允许线程丢弃他最近的assign操作，即工作变量的数据改变了之后，必须告知主存
- 不允许一个线程将没有assign的数据从工作内存同步回主内存
- 一个新的变量必须在主内存中诞生，不允许工作内存直接使用一个未被初始化的变量。就是怼变量实施use、store操作之前，必须经过assign和load操作
- 一个变量同一时间只有一个线程能对其进行lock。多次lock后，必须执行相同次数的unlock才能解锁
- 如果对一个变量进行lock操作，会清空所有工作内存中此变量的值，在执行引擎使用这个变量前，必须重新load或assign操作初始化变量的值
- 如果一个变量没有被lock，就不能对其进行unlock操作。也不能unlock一个被其他线程锁住的变量
- 对一个变量进行unlock操作之前，必须把此变量同步回主内存

JMM对这八种操作规则和对**volatile的一些特殊规则**就能确定哪里操作是线程安全，哪些操作是线程不安全的了。但是这些规则实在复杂，很难在实践中直接分析。所以一般我们也不会通过上述规则进行分析。更多的时候，使用java的happen-before规则来进行分析。

happens-before字面翻译过来就是先行发生，A happens-before B 就是A先行发生于B？

不准确！在Java内存模型中，happens-before 应该翻译成：前一个操作的结果可以被后续的操作获取。讲白点就是前面一个操作把变量a赋值为1，那后面一个操作肯定能知道a已经变成了1。

我们再来看看为什么需要这几条规则？

因为我们现在电脑都是多CPU,并且都有缓存，导致多线程直接的可见性问题。详情可以看我之前的文章面试官：你知道并发Bug的源头是什么吗？

所以为了解决多线程的可见性问题，就搞出了happens-before原则，让线程之间遵守这些原则。编译器还会优化我们的语句，所以等于是给了编译器优化的约束。不能让它优化的不知道东南西北了！

17、volatile

volatile是不错的机制，但是也不能保证原子性。

> 代码验证可见性

```
//Volatile 用来保证数据的同步，也就是可见性
public class JMMVolatileDemo01 {
    // volatile 不加volatile没有可见性
    // 不加 volatile 就会死循环，这里给大家将主要是为了面试，可以避免指令重排
    private volatile static int num = 0;

    public static void main(String[] args) throws InterruptedException {
        new Thread(()->{
            while (num==0){ //此处不要编写代码,让计算机忙的不可开交

            }
        }).start();

        Thread.sleep(1000);
        num = 1;
        System.out.println(num);
    }
}
```

> 验证 volatile 不保证原子性

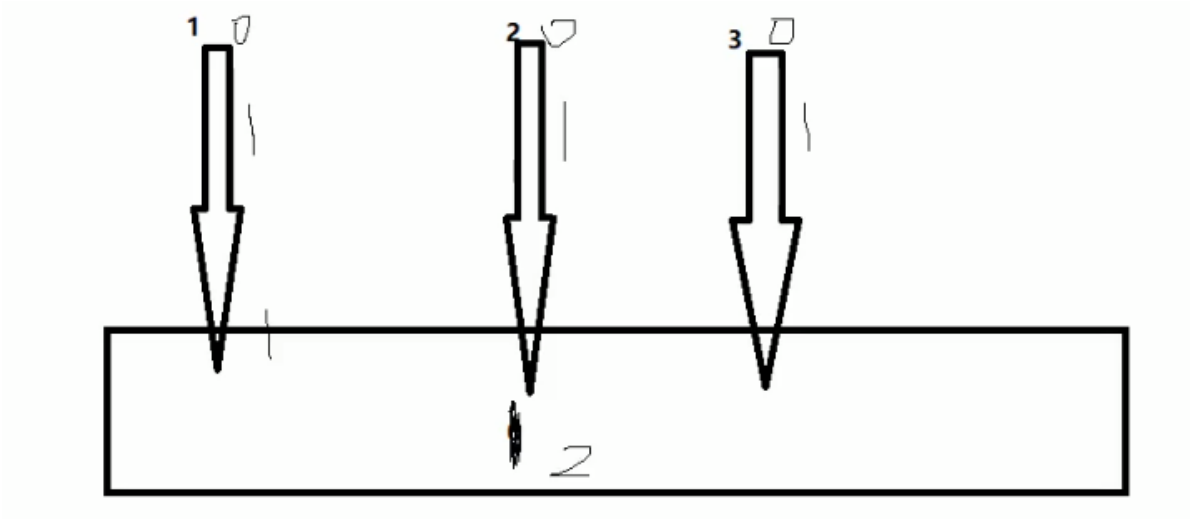
原子性理解：

不可分割，完整性，也就是某个线程正在做某个具体的业务的时候，中间不可以被加塞或者被分割，需要整体完整，要么同时成功，要么同时失败。

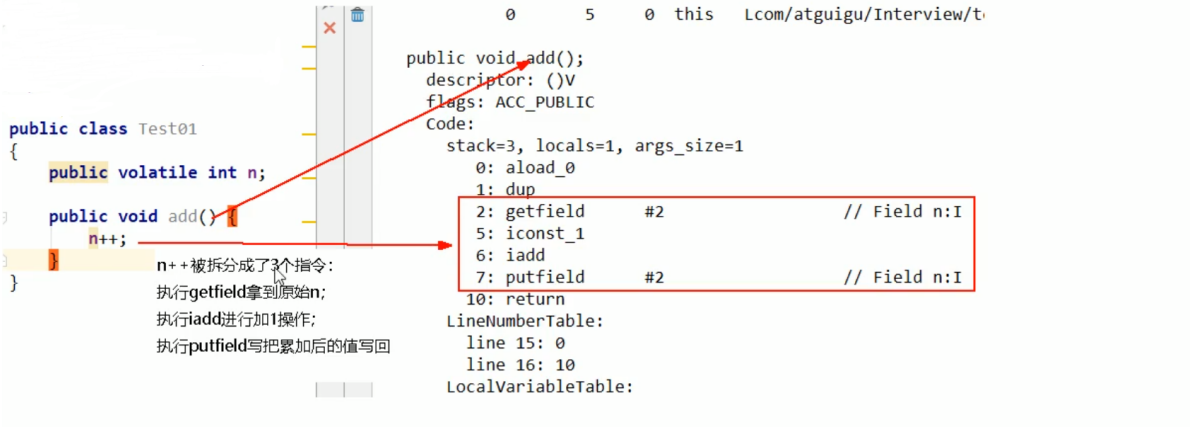
```
public class JMMVolatileDemo02 {

    private volatile static int num = 0;

    public static void add(){
        num++;
    }

    // 结果应该是 num 为 2万，测试看结果
    public static void main(String[] args) throws InterruptedException {
        for (int i = 1; i <= 20; i++) {
            new Thread(()->{
                for (int j = 1; j <= 1000; j++) {
                    add();
                }
            },String.valueOf(i)).start();
        }
```

```
18          // 需要等待上面20个线程都全部计算完毕，看最终结果
19          while (Thread.activeCount()>2){ // 默认一个 main线程  一个 gc 线程
20              Thread.yield();
21          }
22
23          System.out.println(Thread.currentThread().getName()+" "+num);
24      }
25
26  }
```

因为我们的 add 方法没有加锁，但是加了 volatile ，说明 volatile 不能保证原子性；画图解释，数值被覆盖



命令行查看底层字节码代码实现： `javap -c JMMVolatileDemo02.class`



num++ 在多线程下是非线程安全的，如何不加 synchronized解决?

查看原子包下的类，分析方法!

测试：

```java
public class JMMVolatileDemo02 {

    private volatile static AtomicInteger num = new AtomicInteger();

    public static void add(){
        num.getAndIncrement(); // 等价 num++
    }

    // 结果应该是 num 为 2万，测试看结果
    public static void main(String[] args) throws InterruptedException {
        for (int i = 1; i <= 20; i++) {
            new Thread(()->{
                for (int j = 1; j <= 1000; j++) {
                    add();
                }
            },String.valueOf(i)).start();
        }
        // 需要等待上面20个线程都全部计算完毕，看最终结果
        while (Thread.activeCount()>2){ // 默认一个 main线程  一个 gc 线程
            Thread.yield();
        }

        System.out.println(Thread.currentThread().getName()+" "+num);
    }

}
```

你的同学在学习，你的对手在磨刀，你的闺蜜在减肥，隔壁老王在练腰

> 指令重排讲解

计算机在执行程序时，为了提高性能，编译器和处理器的常常会对指令做重排，一般分以下3种：

单线程环境里面确保程序最终执行结果和代码顺序执行的结果一致。

处理器在进行重排序时必须要考虑指令之间的数据依赖性。

多线程环境中线程交替执行，由于编译器优化重排的存在，两个线程中使用的变量能否保证一致性是无法确定的，结果无法预测。

重排理解测试1：

```
public class TestHappensBefore {
    public static void main(String[] args) {
        int x = 11;  // 语句1
        int y = 12;  // 语句2
        x = x + 5;   // 语句3
        y = x * x;   // 语句4
    }
    // 指令顺序预测：  1234   2134  1324
    // 问题：请问语句4可以重排后变成第一条吗？ 答案：不可以

}
```

重排理解测试2：

int a,b,x,y = 0;

线程1	线程2
x = a;	y = b;
b = 1;	a = 2;
x = 0 y = 0	

如果编译器对这段程序代码执行重排优化后，可能出现下列情况

线程1	线程2
b = 1;	a = 2;
x = a;	y = b;
x = 2 y = 1	

案例：

```
// 多线程环境中线程交替执行，由于编译器优化重排的存在
// 两个线程中使用的变量能否保证一致性是无法确定的，结果无法预测。
public class TestHappensBefore {

    int a = 0;
    boolean flag = false;

    public void m1(){
        a = 1;          // 语句1
        flag = true;    // 语句2
    }

    public void m2(){
        if (flag){
            a = a + 5; // 语句3
```

```
16              System.out.println("m2=>"+a);
17          }
18      }
19
20  }
```

指令重排小结：

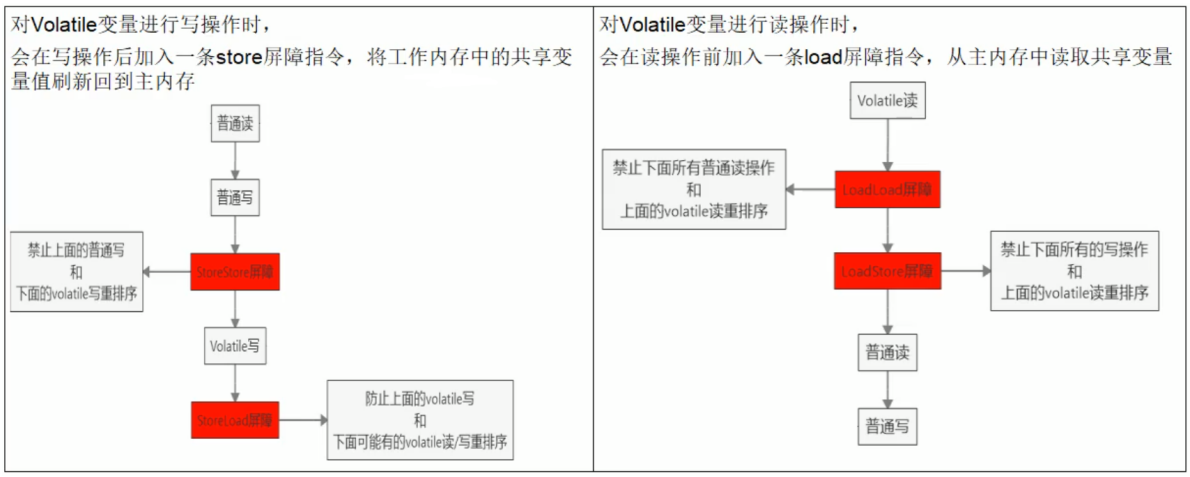
volatile 实现了禁止指令重排优化，从而避免 多线程环境下程序出现乱序执行的现象。

先了解一个概念，内存屏障（Memory Barrier）又称内存栅栏，是一个CPU 指令，它的作用有两个：

1、保证特定操作的执行顺序。

2、保证某些变量的内存可见性（利用该特性实现volatile的内存可见性）。

由于编译器和处理器都能执行指令重排优化。如果在指令间插入一条 Memory Barrier 则会告诉编译器和CPU，不管什么指令都不能和这条 Memory Barrier 指令重排序，也就是说，通过插入内存屏障禁止在内存屏障前后的指令执行重排序优化。内存屏障另外一个作用是强制刷出各种CPU的缓存数据，因此任何CPU上的线程都能读取到这些数据的最新版本。



经过，可见性，原子性，指令重排的话，线程安全性获得保证：

工作内存与主内存同步延迟现象导致的可见性问题，可以使用 synchronized 或 volatile 关键字解决，它们都可以使一个线程修改后的变量立即对其他线程可见。

对于指令重排导致的可见性问题 和 有序性问题，可以利用 volatile 关键字解决，因为 volatile 的另外一个作用就是禁止重排序优化。

18、深入单例模式

单例模式可以说只要是一个合格的开发都会写，但是如果要深究，小小的单例模式可以牵扯到很多东西，比如 多线程是否安全，是否懒加载，性能等等。还有你知道几种单例模式的写法呢？如何防止反射破坏单例模式？今天，我们来探究单例模式。

关于单例模式的概念，在这里就不在阐述了，相信每个小伙伴都了如指掌。我们直接进入正题：

1、饿汉式

```
public class Hungry {
    private Hungry() {
    }
    private final static Hungry hungry = new Hungry();

    public static Hungry getInstance() {
        return hungry;
    }
}
```

饿汉式是最简单的单例模式的写法，保证了线程的安全，在很长的时间里，我都是饿汉模式来完成单例的，因为够简单，后来才知道饿汉式会有一点小问题，看下面的代码：

```
public class Hungry {
    private byte[] data1 = new byte[1024];
    private byte[] data2 = new byte[1024];
    private byte[] data3 = new byte[1024];
    private byte[] data4 = new byte[1024];

    private Hungry() {
    }

    private final static Hungry hungry = new Hungry();

    public static Hungry getInstance() {
        return hungry;
    }
}
```

在Hungry类中，我定义了四个byte数组，当代码一运行，这四个数组就被初始化，并且放入内存了，如果长时间没有用到getInstance方法，不需要Hungry类的对象，这不是一种浪费吗？我希望的是 只有用到了 getInstance方法，才会去初始化单例类，才会加载单例类中的数据。所以就有了 第二种单例模式：懒汉式。

2、懒汉式

正常的 懒汉式单例：

```
public class LazyMan {
    private LazyMan() {
        System.out.println(Thread.currentThread().getName()+"Start");
    }

    private static LazyMan lazyMan;

    public static LazyMan getInstance() {
        if (lazyMan == null) {
            lazyMan = new LazyMan();
        }
        return lazyMan;
    }

    // 测试并发环境，发现单例失效
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            new Thread(()->{
                LazyMan.getInstance();
```

```
            }).start();
        }
    }
}
```

多加一层检测可以避免问题，也就是DCL懒汉式！

```
public class LazyMan {
    private LazyMan() {
    }

    private static LazyMan lazyMan;

    public static LazyMan getInstance() {
        if (lazyMan == null) {
            synchronized (LazyMan.class) {
                if (lazyMan == null) {
                    lazyMan = new LazyMan();
                }
            }
        }
        return lazyMan;
    }
}
```

DCL懒汉式的单例，保证了线程的安全性，又符合了懒加载，只有在用到的时候，才会去初始化，调用效率也比较高，但是这种写法在极端情况还是可能会有一定的问题。因为

```
lazyMan = new LazyMan();
```

不是原子性操作，至少会经过三个步骤：

1. 分配对象内存空间
2. 执行构造方法初始化对象
3. 设置instance指向刚分配的内存地址，此时instance ！ =null；

由于指令重排，导致A线程执行 lazyMan = new LazyMan();的时候，可能先执行了第三步（还没执行第二步），此时线程B又进来了，发现lazyMan已经不为空了，直接返回了lazyMan，并且后面使用了返回的lazyMan，由于线程A还没有执行第二步，导致此时lazyMan还不完整，可能会有一些意想不到的错误，所以就有了下面一种单例模式。

这种单例模式只是在上面DCL单例模式增加一个volatile关键字来避免指令重排：

```
public class LazyMan {
    private LazyMan() {
    }

    private volatile static LazyMan lazyMan;

    public static LazyMan getInstance() {
        if (lazyMan == null) {
            synchronized (LazyMan.class) {
                if (lazyMan == null) {
                    lazyMan = new LazyMan();
                }
            }
        }
        return lazyMan;
```

```
    }
}
```

3、静态内部类

还有这种方式是第一种饿汉式的改进版本，同样也是在类中定义static变量的对象，并且直接初始化，不过是移到了静态内部类中，十分巧妙。既保证了线程的安全性，同时又满足了懒加载。

```
public class Holder {
    private Holder() {
    }

    public static Holder getInstance() {
        return InnerClass.holder;
    }

    private static class InnerClass {
        private static final Holder holder = new Holder();
    }
}
```

4、万恶的反射

万恶的反射登场了，反射是一个比较霸道的东西，无视private修饰的构造方法，可以直接在外面newInstance，破坏我们辛辛苦苦写的单例模式。

```
public static void main(String[] args) {
    try {
        LazyMan lazyMan1 = LazyMan.getInstance();
        Constructor<LazyMan> declaredConstructor =
LazyMan.class.getDeclaredConstructor(null);
        declaredConstructor.setAccessible(true);
        LazyMan lazyMan2 = declaredConstructor.newInstance();
        System.out.println(lazyMan1.hashCode());
        System.out.println(lazyMan2.hashCode());
        System.out.println(lazyMan1 == lazyMan2);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

我们分别打印出lazyMan1，lazyMan2的hashcode，lazyMan1是否相等lazyMan2，结果显而易见，**不相等；**

那么，怎么解决这种问题呢？

```
public class LazyMan {
    private LazyMan() {
        synchronized (LazyMan.class) {
            if (lazyMan != null) {
                throw new RuntimeException("不要试图用反射破坏单例模式");
            }
        }
    }

    private volatile static LazyMan lazyMan;

```

```
    public static LazyMan getInstance() {
        if (lazyMan == null) {
            synchronized (LazyMan.class) {
                if (lazyMan == null) {
                    lazyMan = new LazyMan();
                }
            }
        }
        return lazyMan;
    }
}
```

在私有的构造函数中做一个判断，如果lazyMan不为空，说明lazyMan已经被创建过了，如果正常调用getInstance方法，是不会出现这种事情的，所以直接抛出异常！

但是这种写法还是有问题：

上面我们是先正常的调用了getInstance方法，创建了LazyMan对象，所以第二次用反射创建对象，私有构造函数里面的判断起作用了，反射破坏单例模式失败。但是如果破坏者干脆不先调用getInstance方法，一上来就直接用反射创建对象，我们的判断就不生效了：

```
public static void main(String[] args) {
    try {
        Constructor<LazyMan> declaredConstructor =
LazyMan.class.getDeclaredConstructor(null);
        declaredConstructor.setAccessible(true);
        LazyMan lazyMan1 = declaredConstructor.newInstance();
        LazyMan lazyMan2 = declaredConstructor.newInstance();
        System.out.println(lazyMan1.hashCode());
        System.out.println(lazyMan2.hashCode());
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

那么如何防止这种反射破坏呢?

```
public class LazyMan {
    private static boolean flag = false;
    private LazyMan() {
        synchronized (LazyMan.class) {
            if (flag == false) {
                flag = true;
            } else {
                throw new RuntimeException("不要试图用反射破坏单例模式");
            }
        }
    }
    private volatile static LazyMan lazyMan;
    public static LazyMan getInstance() {
        if (lazyMan == null) {
            synchronized (LazyMan.class) {
                if (lazyMan == null) {
                    lazyMan = new LazyMan();
                }
            }
        }
        return lazyMan;
```

```
    }
}
```

在这里，我定义了一个boolean变量flag，初始值是false，私有构造函数里面做了一个判断，如果flag=false，就把flag改为true，但是如果flag等于true，就说明有问题了，因为正常的调用是不会第二次跑到私有构造方法的，所以抛出异常。

看起来很美好，但是还是不能完全防止反射破坏单例模式，因为可以利用反射修改flag的值。

```
class Demo02{
    public static void main(String[] args) {
        try {
            // 通过反射创建对象
            Constructor<LazyMan> declaredConstructor =
LazyMan.class.getDeclaredConstructor(null);
            Field field = LazyMan.class.getDeclaredField("flag");
            field.setAccessible(true);

            // 通过反射实例化对象
            declaredConstructor.setAccessible(true);
            LazyMan lazyMan1 = declaredConstructor.newInstance();
            System.out.println(field.get(lazyMan1));
            System.out.println(lazyMan1.hashCode());

            //通过反射，修改字段的值！
            field.set(lazyMan1,false);
            LazyMan lazyMan2 = declaredConstructor.newInstance();
            System.out.println(field.get(lazyMan2));
            System.out.println(lazyMan2.hashCode());

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

并没有一个很好的方案去避免反射破坏单例模式，所以轮到我们的枚举登场了。

5、枚举

枚举类型是Java 5中新增特性的一部分，它是一种特殊的数据类型，之所以特殊是因为它既是一种类(class)类型却又比类类型多了些特殊的约束，但是这些约束的存在也造就了枚举类型的简洁性、安全性以及便捷性。

```
public enum EnumSingleton {
    INSTANCE;
    public EnumSingleton getInstance(){
        return INSTANCE;
    }
}

class Demo04{
    public static void main(String[] args) {
        EnumSingleton singleton1=EnumSingleton.INSTANCE;
        EnumSingleton singleton2=EnumSingleton.INSTANCE;
        System.out.println("正常情况下，实例化两个实例是否相同："+
(singleton1==singleton2));
```

```
    }
}
```

枚举是目前最推荐的单例模式的写法，因为足够简单，不需要开发自己保证线程的安全，同时又可以有效的防止反射破坏我们的单例模式，我们可以看下newInstance的源码：

重点就是红框中圈出来的部分，如果枚举去newInstance就直接抛出异常了。

反编译查看下枚举的源码

```
javap -p EnumSingleton.class

Compiled from "EnumSingleton.java"
public final class 单例模式.EnumSingleton extends java.lang.Enum<单例模式.EnumSingleton> {
  public static final 单例模式.EnumSingleton INSTANCE;
  private static final 单例模式.EnumSingleton[] $VALUES;
  public static 单例模式.EnumSingleton[] values();
  public static 单例模式.EnumSingleton valueOf(java.lang.String);
  private 单例模式.EnumSingleton();
  public 单例模式.EnumSingleton getInstance();
  static {};
}
```

这个看的不清楚，我们可以下 jad 进行反编译，我们的素材中也都有！

```
jad -sjava EnumSingleton.class
# 会生成一个java文件
Parsing EnumSingleton.class... Generating EnumSingleton.java
```

我们点开里面的源码

```
// Decompiled by Jad v1.5.8g. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.kpdus.com/jad.html
// Decompiler options: packimports(3)
// Source File Name:   EnumSingleton.java

package 53554F8B6A215F0F;


public final class EnumSingleton extends Enum
{

    public static EnumSingleton[] values()
    {
        return (EnumSingleton[])$VALUES.clone();
    }

    public static EnumSingleton valueOf(String name)
    {
        return (EnumSingleton)Enum.valueOf(53554F8B6A215F0F/EnumSingleton, name);
    }

    private EnumSingleton(String s, int i)
    {
```

```
        super(s, i);
    }

    public EnumSingleton getInstance()
    {
        return INSTANCE;
    }

    public static final EnumSingleton INSTANCE;
    private static final EnumSingleton $VALUES[];

    static
    {
        INSTANCE = new EnumSingleton("INSTANCE", 0);
        $VALUES = (new EnumSingleton[] {
            INSTANCE
        });
    }
}
```

再次尝试破坏看一下!

```
package 单例模式;

import java.lang.reflect.Constructor;

public enum EnumSingleton {
    INSTANCE;
    public EnumSingleton getInstance(){
        return INSTANCE;
    }
}

class Demo04{
    public static void main(String[] args) throws Exception {
        EnumSingleton singleton1=EnumSingleton.INSTANCE;
        EnumSingleton singleton2=EnumSingleton.INSTANCE;
        System.out.println("正常情况下，实例化两个实例是否相同："+
(singleton1==singleton2));
        //Constructor<EnumSingleton> constructor =
EnumSingleton.class.getDeclaredConstructor(); //自身的类没有无参构造方法
        Constructor<EnumSingleton> constructor =
EnumSingleton.class.getDeclaredConstructor(String.class,int.class);
        constructor.setAccessible(true);
        EnumSingleton enumSingleton = constructor.newInstance();
    }
}
```

试图破坏，真的破坏不了!

假如有人问你单例模式，再也不用害怕了。

19、深入理解CAS

> CAS : 比较并交换

前言：互联网缩招之下，初级程序员大量过剩，高级程序员重金难求，除非你不吃这碗饭，否则就要逼自己提升！

用代码理解下什么是CAS：

```
package com.kuang;

import java.util.concurrent.atomic.AtomicInteger;

/**
 * CAS : 比较并交换  compareAndSet
 *
 * 参数：期望值，更新值
 * public final boolean compareAndSet(int expect, int update) {
 *     return unsafe.compareAndSwapInt(this, valueOffset, expect, update);
 * }
 * @author 狂神说Java 24736743@qq.com
 */
public class CASDemo {
    public static void main(String[] args) {
        AtomicInteger atomicInteger = new AtomicInteger(5);

        // main do somethings...

        // 期望的是5，后面改为 2020 ，  所以结果为 true，2020
        System.out.println(atomicInteger.compareAndSet(5,
2020)+"=>"+atomicInteger.get());

        // 期望的是5，后面改为 1024 ，  所以结果为 false，2020
        System.out.println(atomicInteger.compareAndSet(5,
1024)+"=>"+atomicInteger.get());

    }
}
```

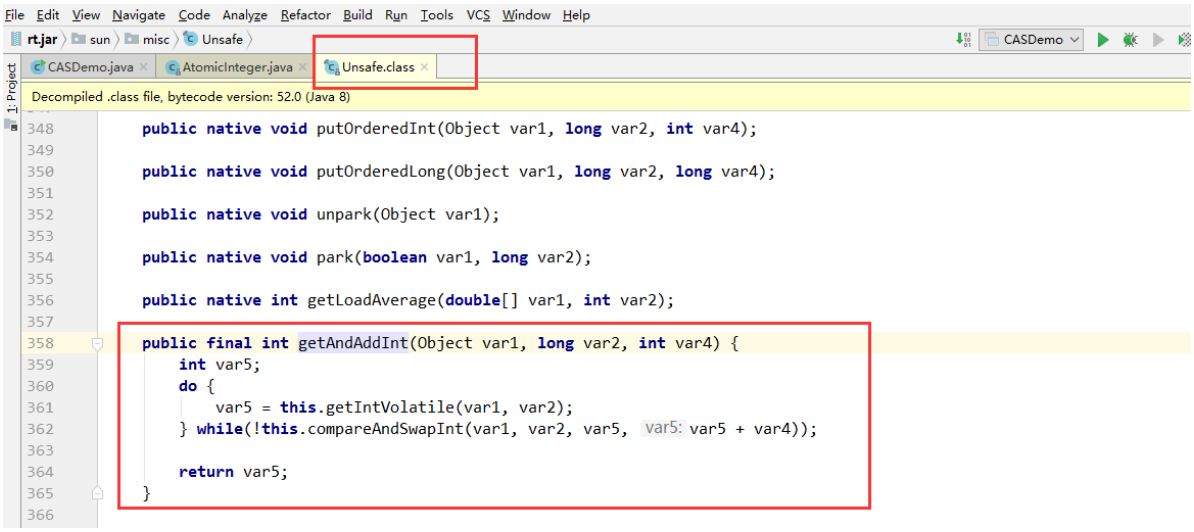
一句话：真实值和期望值相同，就修改成功，真实值和期望值不同，就修改失败！

> CAS 底层原理？如果知道，谈谈你对UnSafe的理解？

`atomicInteger.getAndIncrement();` 这里的自增 + 1怎么实现的！

```
atomicInteger.getAndIncrement(); // 分析源码，如何实现的 i++ 安全的问题

public final int getAndIncrement() { // 继续走源码
    // this 当前对象
    // valueOffset 内存偏移量，内存地址
    // 1.固定写死
    return unsafe.getAndAddInt(this, valueOffset, 1);
}
```

发现到了 字节码文件，我们已经无法在这里操作了！



需要去到JDK安装目录下的 rt.jar 包下寻找了！而且这个类中的方法大部分都是 native 的方法了！

问题：这个UnSafe类到底是什么？ 可以看到AtomicInteger源码中也是它！

 1

1、UnSafe

UnSafe是CAS的核心类，由于Java方法无法直接访问底层系统，需要通过本地（native）方法来访问，UnSafe相当于一个后门，基于该类可以直接操作特定内存的数据，Unsafe类存在于 sun.misc包中，其内部方法操作可以像C的指针一样直接操作内存，因为Java中CAS操作的执行依赖于Unsafe类的方法。

注意：Unsafe类中的所有方法都是Native修饰的，也就是说Unsafe类中的方法都直接调用操作系统底层资源执行相应任务

2、变量valueOffset

表示该变量值在内存中的偏移地址，因为Unsafe就是根据内存偏移地址获取数据的。

3、变量 value用volatile修饰，保证了多线程之间的内存可见性

> 最后解释CAS 是什么

CAS 的全称为 Compare-And-Swap，它是一条CPU并发原语。

它的功能是判断内存某个位置的值是否为预期值，如果是则更改为新的值，这个过程是原子的。

CAS并发原语体现在JAVA语言中就是 sun.misc.Unsafe 类中的各个方法。调用UnSafe类中的CAS方法，JVM会帮我们实现出CAS汇编指令。这是一种完全依赖于硬件的功能，通过它实现了原子操作。再次强调，由于CAS是一种系统原语，原语属于操作系统用于范畴，是由若干条指令组成的，用于完成某个功能的一个过程，并且原语的执行必须是连续的，在执行过程中不允许被中断，也就是说CAS是一条CPU的原子指令，不会造成所谓的数据不一致问题。

分析源码：

```java
public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        // 获取传入对象的地址
        var5 = this.getIntVolatile(var1, var2);
        // 比较并交换，如果var1，var2 还是原来的 var5，就执行内存偏移+1； var5 + var4
    } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));

    return var5;
}
```

> 汇编层面理解

Unsafe 类中的 compareAndSwapint，是一个本地方法，该方法的实现位于 unsafe.cpp 中；

```
UNSAFE_ENTRY(jboolean, Unsafe_CompareAndSwapInt(JNIEnv *env, jobject unsafe, jobject obj, jlong offset, jint e, jint x))
  UnsafeWrapper("Unsafe_CompareAndSwapInt");
  oop p = JNIHandles::resolve(obj);
  jint* addr = (jint *) index_oop_from_field_offset_long(p, offset);
  return (jint)(Atomic::cmpxchg(x, addr, e)) == e;
UNSAFE_END
//先想办法拿到变量value在内存中的地址。
//通过Atomic::cmpxchg实现比较替换，其中参数x是即将更新的值，参数e是原内存的值。
```

> 总结

CAS（CompareAndSwap）

比较当前工作内存中的值和主内存中的值，如果相同则执行规定操作，否则继续比较直到主内存和工作内存中的值一致为止。

CAS 应用

CAS 有3个操作数，内存值V，旧的预期值A，要修改的更新值B。且仅当预期值A 和 内存值 V 相同时，将内存值 V 修改为B，否则什么都不做。

CAS 的缺点

1、循环时间长开销很大。

- 可以看到源码中存在 一个 do...while 操作，如果CAS失败就会一直进行尝试。

2、只能保证一个共享变量的原子操作。

- 当对一个共享变量执行操作时，我们可以使用循环CAS的方式来保证原子操作。但是：
- 对多个共享变量操作时，循环CAS就无法保证操作的原子性，这时候就可以用锁来保证原子性。

3、引出来 ABA 问题？？？

20、原子引用

> 原子类 AtomicInteger 的ABA问题谈谈？原子更新引用知道吗？

CAS => UnSafe => CAS 底层思想 => ABA => 原子引用更新 => 如何规避ABA问题

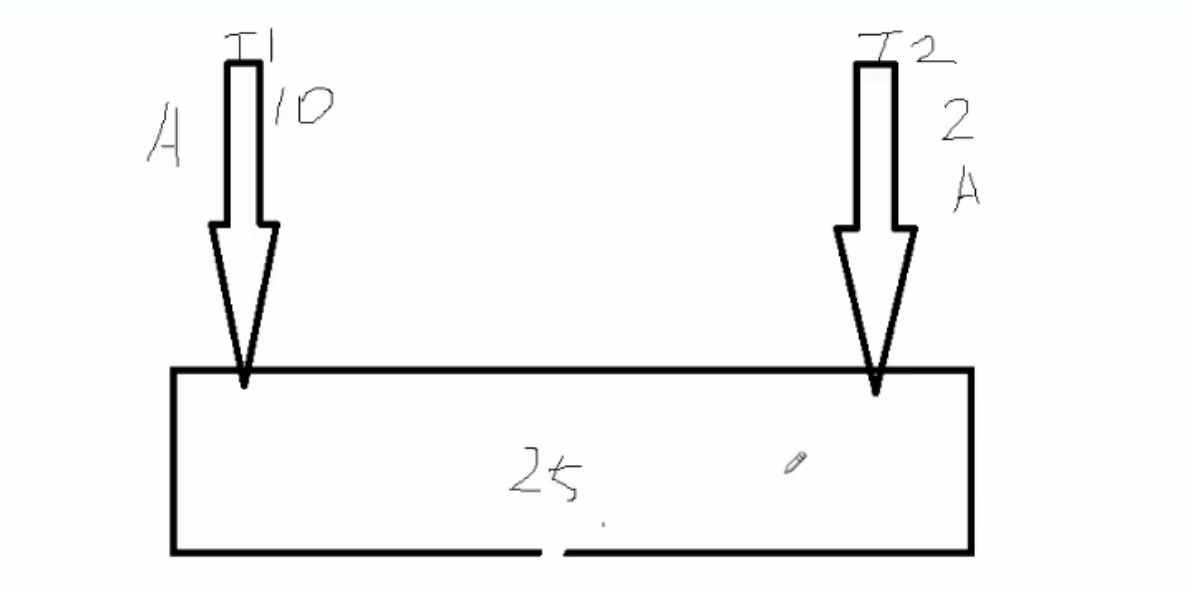
ABA问题怎么产生的?

CAS会导致 “ABA问题”。狸猫换太子

CAS算法实现一个重要前提：需要取出内存中某时刻的数据并在当下时刻比较并交换，那么在这个时间差内会导致数据的变化

比如说一个线程one从内存位置V中取出A，这个时候另一个线程two也从内存中取出A，并且线程two进行了一些操作将值变成了B，然后线程two又将 V位置的数据变成A，这时候线程one进行CAS操作发现内存中仍然是A，然后线程one操作成功。

尽管线程one的CAS操作成功，但是不代表这个过程就是没有问题的。



> 原子引用 AtomicReference

```java
package com.kuang;

import java.util.concurrent.atomic.AtomicReference;

public class AtomicReferenceDemo {
    public static void main(String[] args) {
        User zhangsan = new User("zhangsan", 22);
        User lisi = new User("lisi", 25);

        AtomicReference<User> atomicReference = new AtomicReference<>();
        atomicReference.set(zhangsan); // 设置

        System.out.print(atomicReference.compareAndSet(zhangsan,lisi));
        System.out.println(atomicReference.get().toString());

        System.out.print(atomicReference.compareAndSet(zhangsan,lisi));
        System.out.println(atomicReference.get().toString());
    }
}
```

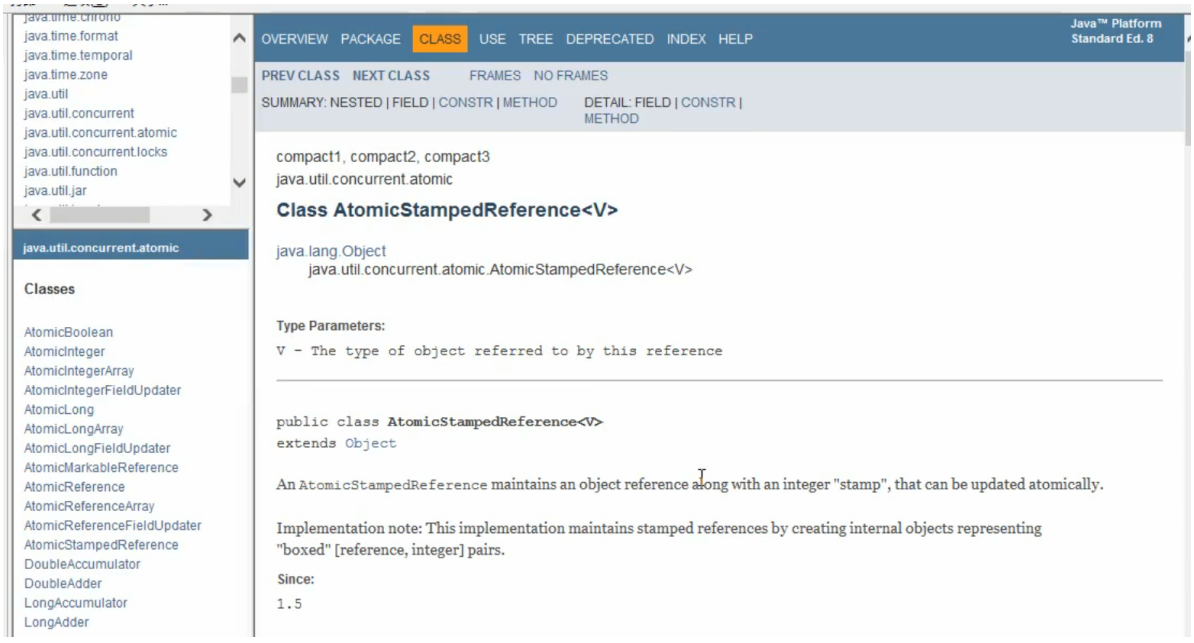
```
20
21  class User{
22      String username;
23      int age;
24
25      public User(String username, int age) {
26          this.username = username;
27          this.age = age;
28      }
29
30      @Override
31      public String toString() {
32          return "User{" +
33                  "username='" + username + '\'' +
34                  ", age=" + age +
35                  '}';
36      }
37  }
```

要解决ABA问题，我们就需要加一个版本号

> 版本号原子引用，类似乐观锁

T1 100 1

T2 100 1 => 101 2 => 100 3



演示ABA问题：

```
1  /**
2   * ABA 问题的解决    AtomicStampedReference
3   */
4  public class ABADemo {
5
6      static AtomicReference<Integer> atomicReference = new AtomicReference<>
   (100);
7
8      public static void main(String[] args) {
```

```
        new Thread(()->{
            atomicReference.compareAndSet(100,101);
            atomicReference.compareAndSet(101,100);
        },"T1").start();

        new Thread(()->{
            // 暂停一秒钟，保证上面线程先执行
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(atomicReference.compareAndSet(100, 2019)); //
修改成功!
            System.out.println(atomicReference.get());
        },"T2").start();
    }
}
```

解决方案：

```
package com.kuang;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicStampedReference;

/**
 * ABA 问题的解决    AtomicStampedReference
 * 注意变量版本号修改和获取问题。不要写错
 */
public class ABADemo {

    static AtomicStampedReference<Integer> atomicStampedReference = new
AtomicStampedReference<>(100,1);

    public static void main(String[] args) {
        new Thread(()->{
            int stamp = atomicStampedReference.getStamp(); // 获得版本号
            System.out.println("T1 stamp 01=>"+stamp);

            // 暂停2秒钟，保证下面线程获得初始版本号
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            atomicStampedReference.compareAndSet(100, 101,
atomicStampedReference.getStamp()
                                                  ,
atomicStampedReference.getStamp()+1);

            System.out.println("T1 stamp
02=>"+atomicStampedReference.getStamp());

            atomicStampedReference.compareAndSet(101, 100,
atomicStampedReference.getStamp()
```

```
                                                ,
atomicStampedReference.getStamp()+1);

            System.out.println("T1 stamp
03=>"+atomicStampedReference.getStamp());
        },"T1").start();

        new Thread(()->{

            int stamp = atomicStampedReference.getStamp(); // 获得版本号
            System.out.println("T2 stamp 01=>"+stamp);
            // 暂停3秒钟，保证上面线程先执行
            try {
                TimeUnit.SECONDS.sleep(3);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            boolean result = atomicStampedReference.compareAndSet(100, 2019,
stamp, stamp + 1);
            System.out.println("T2 是否修改成功 =>"+ result);
            System.out.println("T2 最新stamp
=>"+atomicStampedReference.getStamp());
            System.out.println("T2 当前的最新值
=>"+atomicStampedReference.getReference());

        },"T2").start();
    }
}
```

21、Java锁

1、公平锁非公平锁

> 是什么

公平锁：是指多个线程按照申请锁的顺序来获取锁，类似排队打饭，先来后到。

非公平锁：是指多个线程获取锁的顺序并不是按照申请锁的顺序，有可能后申请的线程比现申请的线程优先获取锁，在高并发的情况下，有可能会造成优先级反转或者饥饿现象。

```
// 无参
public ReentrantLock() {
    sync = new NonfairSync();
}
// 有参
public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}
```

> 两者区别

并发包中的 ReentrantLock 的创建可以指定构造函数 的 boolean类型来得到公平锁或者非公平锁，默认是非公平锁！

公平锁：就是很公平，在并发环境中，每个线程在获取到锁时会先查看此锁维护的等待队列，如果为空，或者当前线程是等待队列的第一个，就占有锁，否则就会加入到等待队列中，以后会按照FIFO的规则从队列中取到自己。

非公平锁：非公平锁比较粗鲁，上来就直接尝试占有锁，如果尝试失败，就会采用类似公平锁那种方式。

Java ReentrantLock 而言，通过构造函数指定该锁是否是公平锁，默认是非公平锁。非公平锁的优点在于吞吐量比公平锁大。

对于Synchronized而言，也是一种非公平锁。

2、可重入锁

> 是什么

可重入锁（也叫递归锁）

指的是同一线程外层函数获得锁之后，内层递归函数仍然能获取该锁的代码，在同一个线程在外层方法获取锁的时候，在进入内层方法会自动获取锁。

也就是说，线程可以进入任何一个它已经拥有的锁，所同步着的代码块。 好比家里进入大门之后，就可以进入里面的房间了；

ReentrantLock、Synchronized 就是一个典型的可重入锁；

可重入锁最大的作用就是避免死锁

测试一：Synchronized

```
package com.kuang;

/**
 * 可重入锁（也叫递归锁）
 * 指的是同一线程外层函数获得锁之后，内层递归函数仍然能获取该锁的代码
 * 在同一个线程在外层方法获取锁的时候，在进入内层方法会自动获取锁。
 */
public class ReentrantLockDemo {
    public static void main(String[] args) throws Exception {
        Phone phone = new Phone();

        // T1 线程在外层获取锁时，也会自动获取里面的锁
        new Thread(()->{
            phone.sendSMS();
        },"T1").start();

        new Thread(()->{
            phone.sendSMS();
        },"T2").start();

    }
```

```
}

class Phone{
    public synchronized void sendSMS(){
        System.out.println(Thread.currentThread().getName()+" sendSMS");
        sendEmail();
    }

    public synchronized void sendEmail(){
        System.out.println(Thread.currentThread().getName()+" sendEmail");
    }
}
```

测试二：ReentrantLock

```
package com.kuang;

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * 可重入锁（也叫递归锁）
 * 指的是同一线程外层函数获得锁之后，内层递归函数仍然能获取该锁的代码
 * 在同一个线程在外层方法获取锁的时候，在进入内层方法会自动获取锁。
 */
public class ReentrantLockDemo {
    public static void main(String[] args) throws Exception {
        Phone phone = new Phone();

        // T1 线程在外层获取锁时，也会自动获取里面的锁
        new Thread(phone,"T1").start();

        new Thread(phone,"T2").start();

    }
}

class Phone implements Runnable{

    Lock lock = new ReentrantLock();

    @Override
    public void run() {
        get();
    }
    public void get(){
        lock.lock();
        // lock.lock(); 锁必须匹配，如果两个锁，只有一个解锁就会失败
        try {
            System.out.println(Thread.currentThread().getName()+" get()");
            set();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
            // lock.lock();
        }

```

```
    }
    public void set(){
        lock.lock();
        try {
            System.out.println(Thread.currentThread().getName()+" set()");
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }

    }

}
```

3、自旋锁

自旋锁（spinlock）

是指尝试获取锁的线程不会立即阻塞，而是采用循环的方式去尝试获取锁，这样的好处是减少线程上下文切换的消耗，缺点是循环会消耗CPU。

```
unsafe.getAndAddInt()
public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        // 获取传入对象的地址
        var5 = this.getIntVolatile(var1, var2);
        // 比较并交换，如果var1，var2 还是原来的 var5，就执行内存偏移+1； var5 + var4
    } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));

    return var5;
}
```

测试代码：

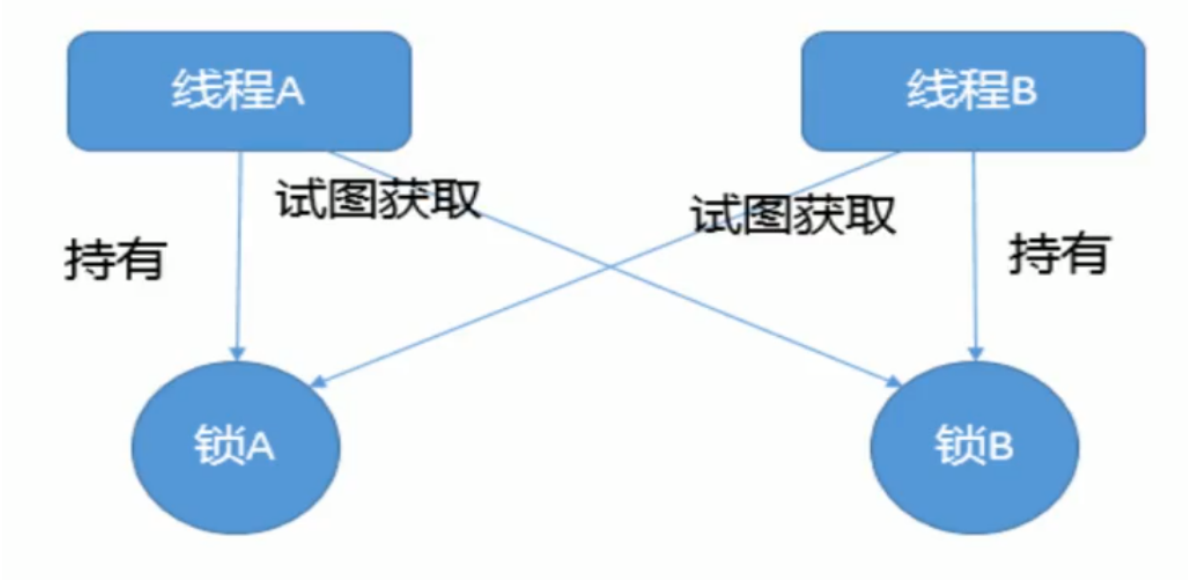
```
package com.kuang;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicReference;

public class SpinLockDemo {
    // 原子引用线程，没写参数，引用类型默认为null
    AtomicReference<Thread> atomicReference = new AtomicReference<>();

    //上锁
    public void myLock(){
        Thread thread = Thread.currentThread();
        System.out.println(Thread.currentThread().getName()+"==>mylock");

        // 自旋
        while (!atomicReference.compareAndSet(null,thread)){

        }
    }
```

```
20
21      //解锁
22      public void myUnlock(){
23          Thread thread = Thread.currentThread();
24          atomicReference.compareAndSet(thread,null);
25          System.out.println(Thread.currentThread().getName()+"==>myUnlock");
26      }
27
28      // 测试
29      public static void main(String[] args) {
30
31          SpinLockDemo spinLockDemo = new SpinLockDemo();
32
33          new Thread(()->{
34              spinLockDemo.myLock();
35              try {
36                  TimeUnit.SECONDS.sleep(5);
37              } catch (InterruptedException e) {
38                  e.printStackTrace();
39              }
40              spinLockDemo.myUnlock();
41          },"T1").start();
42
43          try {
44              TimeUnit.SECONDS.sleep(1);
45          } catch (InterruptedException e) {
46              e.printStackTrace();
47          }
48
49          new Thread(()->{
50              spinLockDemo.myLock();
51
52              try {
53                  TimeUnit.SECONDS.sleep(1);
54              } catch (InterruptedException e) {
55                  e.printStackTrace();
56              }
57
58              spinLockDemo.myUnlock();
59          },"T2").start();
60
61      }
62
63  }
```

4、死锁

> 死锁是什么

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，若无外力干涉那它们都将无法推进下去，如果系统资源充足，进程的资源请求都能够得到满足，死锁出现的可能性就很低，否者就会因为争夺有限的资源而陷入死锁。



产生死锁主要原因：

1、系统资源不足

2、进程运行推进的顺序不合适

3、资源分配不当

> 测试

```
import java.util.concurrent.TimeUnit;

public class DeadLockDemo {
    public static void main(String[] args) {
        String lockA = "lockA";
        String lockB = "lockB";

        new  Thread(new HoldLockThread(lockA,lockB),"T1").start();
        new  Thread(new HoldLockThread(lockB,lockA),"T2").start();
    }
}

class HoldLockThread implements Runnable{

    private String lockA;
    private String lockB;

    public HoldLockThread(String lockA, String lockB) {
        this.lockA = lockA;
        this.lockB = lockB;
    }

    public void run() {
        synchronized (lockA){

 System.out.println(Thread.currentThread().getName()+"lock:"+lockA+"=>get"+lockB);

```

```
            try {
                TimeUnit.SECONDS.sleep(2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            synchronized (lockB){

 System.out.println(Thread.currentThread().getName()+"lock:"+lockB+"=>get"+lockA);
            }

        }
    }
}
```

> 解决

拓展java自带工具操作：

1、查看JDK目录的bin目录

2、使用 `jps -l` 命令定位进程号

3、使用 `jstack 进程号` 找到死锁查看

结果：

```
Java stack information for the threads listed above:
===================================================
"T2":
        at com.kuang.HoldLockThread.run(DeadLockDemo.java:43)
        - waiting to lock <0x00000000d5b87298> (a java.lang.String)
        - locked <0x00000000d5b872d0> (a java.lang.String)
        at java.lang.Thread.run(Thread.java:748)
"T1":
        at com.kuang.HoldLockThread.run(DeadLockDemo.java:43)
        - waiting to lock <0x00000000d5b872d0> (a java.lang.String)
        - locked <0x00000000d5b87298> (a java.lang.String)
        at java.lang.Thread.run(Thread.java:748)

Found 1 deadlock.
```

问10个人，9个说看日志，还有一个分析堆栈信息，这一步，他就已经赢了！