

# JUC

## 概念

JUC就是java.util.concurrent工具包的简称。这是一个处理线程的工具包。

## 线程与进程

- 进程：进程是**程序的一次执行**，进程是一个程序及其数据在处理机上顺序执行时所发生的活动，进程是具有独立功能的程序在一个数据集合上运行的过程，它是**系统进行资源分配和调度的一个独立单位**。
- 线程：线程作为**资源调度**的基本单位，是程序的执行单元，执行路径(单线程：一条执行路径，多线程：多条执行路径)。是程序使用CPU的最基本单位。

## java能自己开启线程吗？

```
public synchronized void start() {  
    /**  
     * This method is not invoked for the main method thread or "system"  
     * group threads created/set up by the VM. Any new functionality added  
     * to this method in the future may have to also be added to the VM.  
     *  
     * A zero status value corresponds to state "NEW".  
     */  
    if (threadStatus != 0)  
        throw new IllegalThreadStateException();  
  
    /* Notify the group that this thread is about to be started  
     * so that it can be added to the group's list of threads  
     * and the group's unstarted count can be decremented. */  
    group.add(this);  
  
    boolean started = false;  
    try {  
        start0();  
        started = true;  
    } finally {  
        try {  
            if (!started) {  
                group.threadStartFailed(this);  
            }  
        } catch (Throwable ignore) {  
            /* do nothing. If start0 threw a Throwable then  
             it will be passed up the call stack */  
        }  
    }  
}
```

//java不能自己创建线程，调用的是底层c++的native方法，java不能调硬件方面的，因为是运行在虚拟机上的。

```
private native void start0();
```

```
public static void main(String[] args) {  
    //获取CPU核数  
    System.out.println(Runtime.getRuntime().availableProcessors());  
}
```

## 线程有几个状态

```
public enum State {  
    //新建  
    NEW,  
  
    //运行  
    RUNNABLE,  
  
    //阻塞  
    BLOCKED,  
  
    //等待  
    WAITING,  
  
    //超时等待  
    TIMED_WAITING,  
  
    //终止  
    TERMINATED;  
}
```

## wait/sleep区别

### 1、来自不同的类

wait=>Object

sleep=>Thread

### 2、关于锁的释放

wait会释放锁，sleep睡着了，会抱着锁睡觉，不会释放锁。

### 3、使用的范围不同

wait必须在同步代码块中使用

sleep可以在任何地方使用

### 4、是否需要捕获异常

wait不需要捕获异常

sleep必须要捕获异常

## lock锁

synchronized锁

```
/*  
    真正的多线程开发  
    线程就是一个单独的资源类，没有任何附属操作
```

## 1、属性，方法

```
*/
public class SaleTicketDemo01 {
    public static void main(String[] args) {
        Ticket ticket = new Ticket();

        //new Thread(new Runnable(){public void run(){}}),但因为是
        //@FunctionalInterface函数式接口，所以可以用lambda表达式(参数)->{方法体代码}
        new Thread(() -> {
            for (int i = 0; i < 50; i++) {
                ticket.saleTicket();
            }
        }, "A").start();
        new Thread(() -> {
            for (int i = 0; i < 50; i++) {
                ticket.saleTicket();
            }
        }, "B").start();
        new Thread(() -> {
            for (int i = 0; i < 50; i++) {
                ticket.saleTicket();
            }
        }, "C").start();
    }
}

//资源类
class Ticket {
    //属性，方法
    //剩余总票数
    private static int number = 50;
    //已经卖出的票数
    private static int sale = 0;

    //如果用synchronized修饰后使用递归的话，会让A线程全部执行完了后才开始执行其他线程，而如果
    //A线程执行完了其他线程就不会再执行了，所以不能用递归
    public synchronized void saleTicket() {
        if (number > 0) {
            System.out.println(Thread.currentThread().getName() + "窗口卖出第" +
                (++sale) + "张票，还剩" + (--number) + "张票");
            //            saleTicket();
        }
    }
}
```

## lock接口

compact1, compact2, compact3  
java.util.concurrent.locks

### Interface Lock

所有已知实现类:

- 可重入锁 (常用) → ReentrantLock
- 读锁 → ReentrantReadWriteLock.ReadLock
- 写锁 → ReentrantReadWriteLock.WriteLock

```

public ReentrantLock() {
    sync = new NonfairSync(); // 非公平锁
}

/**
 * Creates an instance of {@code ReentrantLock} with the
 * given fairness policy.
 *
 * @param fair {@code true} if this lock should use a fair ordering policy true:公平锁, false: 非公平
 */
public ReentrantLock(boolean fair) { sync = fair ? new FairSync() : new NonfairSync(); }

```

公平锁：遵循先来后到原则

非公平锁：不公平，可以插队（默认）

```

//lock
class Tickets {
    //属性, 方法
    //剩余总票数
    private static int number = 50;
    //已经卖出的票数
    private static int sale = 0;

    Lock lock = new ReentrantLock();

    public void saleTicket() {
        lock.lock(); // 加锁

        // try 中业务代码
        try {

            if (number > 0) {
                System.out.println(Thread.currentThread().getName() + "窗口卖出第"
+ (++sale) + "张票, 还剩" + (--number) + "张票");
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock(); // 解锁
        }
    }
}

```

### Synchronized和Lock的区别:

- 1、Synchronized是关键字，Lock是java接口
- 2、Synchronized无法判断获取锁的状态，Lock可以判断是否获取到了锁
- 3、Synchronized会自动释放锁，Lock需要手动释放锁，如果不释放锁，会发生死锁
- 4、Synchronized线程1（获得锁）、线程2（等待），如果线程1阻塞，线程2就会一直等待，LOCK锁就不一定会一直等待（lock.tryLock()）
- 5、Synchronized是可重入锁，是不可以中断的，非公平。Lock，可重入锁，可以判断锁，公平与非公平可以自己设定

6、Synchronized适合少量代码同步问题，Lock是适合大量代码的同步问题

锁是什么，如何判断锁的是谁？

## 生产者和消费者问题

Synchronized版

```
//生产者消费者模式
public class test {
    public static void main(String[] args) {
        Data data = new Data();
        new Thread()->{
            for (int i = 0; i < 10; i++) {
                try {
                    data.increment();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }, "A").start();

        new Thread()->{
            for (int i = 0; i < 10; i++) {
                try {
                    data.decrement();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }, "B").start();
    }
}

//资源类
//synchronized:判断等待，业务，通知
class Data {
    private int number = 0;

    public synchronized void increment() throws InterruptedException {
        if (number != 0) {
            //等待
            this.wait();
        }
        number++;
        System.out.println(Thread.currentThread().getName() + "线程执行了++,
number=>" + number);
        this.notifyAll();
    }

    public synchronized void decrement() throws InterruptedException {
        if (number == 0) {
            //等待
            this.wait();
        }
    }
}
```

```

    }
    //业务
    number--;
    System.out.println(Thread.currentThread().getName() + "线程执行了--,
number=>" + number);
    //通知其他线程
    this.notifyAll();
}
}

```

"C:\Program Files\Java\jdk1.8.0\_131\bin\java.exe" ...

A线程执行了++, number=>1  
 B线程执行了--, number=>0  
 A线程执行了++, number=>1  
 B线程执行了--, number=>0  
 A线程执行了++, number=>1  
 B线程执行了--, number=>0  
 A线程执行了++, number=>1  
 B线程执行了--, number=>0

问题, 如果有ABCD四个线程又怎样锁呢?

o线程执行了--, number=>1  
 o线程执行了--, number=>0  
 o线程执行了++, number=>1  
 A线程执行了++, number=>2  
 o线程执行了++, number=>3  
 o线程执行了--, number=>2  
 o线程执行了--, number=>1  
 o线程执行了--, number=>0  
 o线程执行了++, number=>1

出现了问题

查看Object类中的wait()方法:

导致当前线程等待, 直到另一个线程调用该对象的notify()方法或notifyAll()方法。换句话说, 这个方法的行为就好像简单地执行调用wait(0)。

当前的线程必须拥有该对象的显示器。该线程释放此监视器的所有权, 并等待另一个线程通知等待该对象监视器的线程通过调用notify方法或notifyAll方法notifyAll。然后线程等待, 直到它可以重新获得监视器的所有权并恢复执行。

在一个参数版本中, 中断和虚假唤醒是可能的, 并且该方法应该始终在循环中使用:

如果用if的话会只判断一次, 所以会造成虚假唤醒

```

synchronized (obj) {
    while (<condition does not hold>) {
        obj.wait();
        ... // Perform action appropriate to condition
    }
}

```

采用while判断会判断多次, 即wait

该方法只能由作为该对象的监视器的所有者的线程调用。有关线程可以成为监视器所有者的方式的说明, 请参阅notify方法。

异常

IllegalMonitorStateException - 如果当前线程不是对象监视器的所有者。

将if改为while:

未出问题

```
class Data2 {
    private int number = 0;
    Lock lock = new ReentrantLock();
}
```

```

        Condition condition = lock.newCondition();

        public void increment() throws InterruptedException {
            lock.lock();
            try {
                while (number != 0) {
                    //等待
                    condition.await();
                }
                number++;
                System.out.println(Thread.currentThread().getName() + "线程执行了++,
number=>" + number);
                condition.signalAll();
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
            }
        }

        public void decrement() throws InterruptedException {
            lock.lock();
            try {
                while (number == 0 ) {
                    //等待
                    condition.await();
                }
                //业务
                number--;
                System.out.println(Thread.currentThread().getName() + "线程执行了--,
number=>" + number);
                //通知其他线程
                condition.signalAll();
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
            }
        }
    }
}

```

问题，ABCD四个线程是随机开启的，想要有序执行ABCD线程：

```

class Data3 {
    private int signal = 1;
    Lock lock = new ReentrantLock();
    Condition condition1 = lock.newCondition();
    Condition condition2 = lock.newCondition();
    Condition condition3 = lock.newCondition();

    public void increment() throws InterruptedException {
        lock.lock();
        try {

```



```

        while (signal != 1 ) {
            //等待
            condition1.await();
        }
        signal = 2;
        System.out.println(Thread.currentThread().getName() + "线程执行了");
        condition2.signal();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}

public void decrement() throws InterruptedException {
    lock.lock();
    try {
        while (signal != 2 ) {
            //等待
            condition2.await();
        }
        //业务
        signal = 3;
        System.out.println(Thread.currentThread().getName() + "线程执行了");
        //通知其他线程
        condition3.signal();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}

public void dontDoThings() throws InterruptedException {
    lock.lock();
    try {
        while (signal != 3 ) {
            //等待
            condition3.await();
        }
        //业务
        signal = 1;
        System.out.println(Thread.currentThread().getName() + "线程执行了" );
        //通知其他线程
        condition1.signal();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
}

```

# 八锁现象

## 深刻理解什么是锁

```
/*
 * 1、同一个对象调用两个同步锁方法
 * 2、不同的对象调用两个同步锁方法
 * */
public class Test01 {
    public static void main(String[] args) throws InterruptedException {
        Phone phone1 = new Phone();
        Phone phone2 = new Phone();
        new Thread()->{
            phone1.sendMes();
        }.start();

        TimeUnit.SECONDS.sleep(1);

        new Thread()->{phone2.call();}.start();
    }
}

class Phone{
    //synchronized 锁的对象是方法的调用者! (phone)
    //两个方法用的是同一个锁, 谁先拿到谁先执行
    public synchronized void sendMes() {
        try {
            TimeUnit.SECONDS.sleep(4);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("发消息!");
    }

    public synchronized void call() {
        System.out.println("打电话!");
    }
}

/*
 * 3、同一个对象分别调用同步锁方法和普通方法
 * 4、两个对象, 两个同步方法
 * */
public class Test02 {
    public static void main(String[] args) throws InterruptedException {
        Phone2 phone = new Phone2();
        new Thread()->{
            phone.sendMes();
        }.start();

        TimeUnit.SECONDS.sleep(1);

        new Thread()->{phone.hello();}.start();
    }
}
```

```

class Phone2{
    //synchronized 锁的对象是方法的调用者! (phone)
    //两个方法用的是同一个锁, 谁先拿到谁先执行
    public synchronized void sendMes() {
        try {
            TimeUnit.SECONDS.sleep(4);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("发消息!");
    }

    public synchronized void call() {
        System.out.println("打电话!");
    }

    //普通方法, 这里没有锁, 不是同步方法, 不受锁的影响
    public void hello() {
        System.out.println("hello!");
    }
}

/*
 * 5、一个对象, 两个静态方法
 * 6、两个对象, 两个静态方法
 * */
public class Test03 {
    public static void main(String[] args) throws InterruptedException {
        Phone3 phone = new Phone3();
        new Thread()->{
            phone.sendMes();
        }.start();

        TimeUnit.SECONDS.sleep(1);

        new Thread()->{phone.call();}.start();
    }
}

class Phone3{
    //synchronized 锁的对象是方法的调用者! (phone)
    //如果调用的是static方法, 方法则会在类加载的时候就有了, 所以如果使用了static则锁的是
    Class, 即Phone3.Class
    public static synchronized void sendMes() {
        try {
            TimeUnit.SECONDS.sleep(4);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("发消息!");
    }

    public static synchronized void call() {
        System.out.println("打电话!");
    }
}

/*

```

```

* 7、一个对象，一个同步方法，一个静态同步方法
* 8、两个对象，一个同步方法，一个静态方法
* */
public class Test04 {
    public static void main(String[] args) throws InterruptedException {
        Phone4 phone1 = new Phone4();
        Phone4 phone2 = new Phone4();
        new Thread()->{
            phone1.sendMes();
        }.start();

        TimeUnit.SECONDS.sleep(1);

        new Thread()->{phone2.call();}.start();
    }
}

class Phone4{
    //synchronized 锁的对象是方法的调用者! (phone)
    public synchronized void sendMes() {
        try {
            TimeUnit.SECONDS.sleep(4);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("发消息!");
    }

    //锁的是Class类模板
    public static synchronized void call() {
        System.out.println("打电话!");
    }
}

```

## 小结

new的对象锁的是this，具体的一个调用者

static 锁的是类模板

## 集合类不安全

### List不安全

```

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

        List<String> list = new CopyOnWriteArrayList(new ArrayList<>());

        for (int i = 0; i < 10; i++) {
            new Thread()->{ //java.util.ConcurrentModificationException 并发修改
                异常
            }.start();
            /*
            * 解决办法:
            * 1、List<String> list = new Vector<>();
            */
        }
    }
}

```

```

        * 2、List<String> list = Collections.synchronizedList(new
ArrayList<>());
        * 3、List<String> list = new CopyOnWriteArrayList(new ArrayList<>
());

        * CopyOnWrite写入时复制， COW  计算机程序设计领域的一种优化策略
        * 多个线程调用的时候，list读取的时候是固定的，写入的时候先复制再写入
        */
        list.add(String.valueOf((int)(Math.random() * 100) ));
        System.out.println(list);
        },String.valueOf(i + 1)).start();
    }
}
}

```

```

        * @since 1.2
        */
        public synchronized boolean add(E e) {
            modCount++;
            ensureCapacityHelper( minCapacity: elementCount + 1);
            elementData[elementCount++] = e;
            return true;
        }

        */
        public boolean add(E e) {
            final ReentrantLock lock = this.lock;
            lock.lock();
            try {
                Object[] elements = getArray();
                int len = elements.length;
                Object[] newElements = Arrays.copyOf(elements, newLength: len + 1);
                newElements[len] = e;
                setArray(newElements);
                return true;
            } finally {
                lock.unlock();
            }
        }
    }
}

```

Vector是用synchronized关键字进行锁

CopyOnWrite是用的juc中手动加锁

复制

Set不安全

```

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

        // Set<String> set = new HashSet<>();
        // Set<String> set = Collections.synchronizedSet(new HashSet<>());
        Set<String> set = new CopyOnWriteArraySet<>(new HashSet<>());
        /* public CopyOnWriteArraySet() {
            al = new CopyOnWriteArrayList<E>();CopyOnWriteArraySet的构造方法还是调用了CopyOnWriteArrayList构造方法
        }*/

        /*
        * ConcurrentModificationException并发修改异常
        */
    }
}

```

```

* 解决方案:
* 1、Set<String> set = Collections.synchronizedSet(new HashSet<>());
* 2、Set<String> set = new CopyOnWriteArraySet<>(new HashSet<>());
* */
for (int i = 0; i < 10; i++) {
    new Thread()->{
        set.add(UUID.randomUUID().toString().substring(1,6));
        System.out.println(set);
    },String.valueOf(i + 1)).start();
}
}
}

```

## HashSet底层

```

public HashSet() {
    map = new HashMap<>();
}

//其add方法就是map的put方法，因为map的key不能重复，所以set不能重复，PRESENT是一个不变的值
public boolean add(E e) {
    return map.put(e, PRESENT)==null;
}

```

## HashMap不安全

```

public class UnsafeMap {
    public static void main(String[] args) {
        // Map<String, Object> map = new HashMap<>();
        // Map<String, Object> map = Collections.synchronizedMap(new HashMap<>());
        Map<String, Object> map = new ConcurrentHashMap<>();

        /*
        * ConcurrentModificationException
        * 解决方法
        * 1、Map<String, Object> map = Collections.synchronizedMap(new HashMap<>());
        * 2、Map<String, Object> map = new ConcurrentHashMap<>();
        * */

        for (int i = 0; i < 10; i++) {
            new Thread()->{
                map.put(Thread.currentThread().getName(),
                UUID.randomUUID().toString().substring(0,5));
                System.out.println(map);
            },String.valueOf(i + 1)).start();
        }
    }
}

```

# Callable

```
@FunctionalInterface
public interface Callable<V>
```

返回结果并可能引发异常的任务。实现者定义一个没有参数的单一方法，称为call。

Callable接口类似于Runnable，因为它们都是为其实例可能由另一个线程执行的类设计的。然而，Runnable不返回结果，也不能抛出被检查的异常。

该Executors类包含的实用方法，从其他普通形式转换为Callable类。

- 1、可以有返回值
- 2、可以抛出异常
- 3、方法不同，call()

```
@FunctionalInterface
public interface Callable<V> {

    /**
     * Computes a result, or throws an exception if unable to do so.
     *
     * @return computed result
     * @throws Exception if unable to compute a result
     */
    V call() throws Exception;
}
```

这里的泛型就是call方法的返回值类型

查看Thread类构造方法：

Constructor and Description	
<code>Thread()</code>	分配一个新的 Thread对象。
<code>Thread(Runnable target)</code>	分配一个新的 Thread对象。 可以是无参构造或者是Runnable参数，不能用Callable为参数
<code>Thread(Runnable target, String name)</code>	分配一个新的 Thread对象。
<code>Thread(String name)</code>	分配一个新的 Thread对象。
<code>Thread(ThreadGroup group, Runnable target)</code>	分配一个新的 Thread对象。
<code>Thread(ThreadGroup group, Runnable target, String name)</code>	分配一个新的 Thread对象，使其具有 target作为其运行对象，具有指定的 name作为其名称，属于 group引用的线程组。
<code>Thread(ThreadGroup group, Runnable target, String name, long stackSize)</code>	分配一个新的 Thread对象，以便它具有 target作为其运行对象，将指定的 name正如其名，以及属于该线程组称作 group，并具有指定的 堆栈大小。
<code>Thread(ThreadGroup group, String name)</code>	分配一个新的 Thread对象。

因为不能直接用Callable作构造参数，所以要找一个中间的类将Callable转为Runnable接口的实现类中，其中FutureTask是实Runnable的实现类，且其中有构造方法可以是Callable参数：

```
public FutureTask(Callable<V> callable)
```

创建一个 `FutureTask`，它将在运行时执行给定的 `Callable`。

参数

`callable` - 可调用任务

异常

`NullPointerException` - 如果可调用为`null`

```
public class CallableTest01 {
    public static void main(String[] args) throws ExecutionException,
        InterruptedException {
        //      new Thread(new Runnable() {@Override public void run() { }}).start();
        //      new Thread(new FutureTask<>()).start();
        MyThread thread = new MyThread();
        FutureTask<String> futureTask = new FutureTask<>(thread);
        new Thread(futureTask, "A").start(); //callable的结果会被缓存，执行两次start也
        只会打印一次"执行了call方法"
        new Thread(futureTask, "B").start();
        String str = (String)futureTask.get();//get方法返回call方法的返回值，如果返回
        的结果在call方法中需要等待，则可能会造成阻塞，通常将get方法放在最后或者使用异步通信来处理
        System.out.println("返回值: " + str);
    }
}

class MyThread implements Callable<String> {

    @Override
    public String call() throws Exception {
        System.out.println("执行了call方法");
        return "hello";
    }
}
```

```
"C:\Program Files\Java\jdk1.8.0_131\bin\java.exe" ...
```

执行了call方法

返回值: hello

Process finished with exit code 0

## 常用的辅助类

### CountDownLatch (理解: 减法计数器)



```
public class CountdownLatch
extends Object
```

491164 陈万强 10

允许一个或多个线程等待直到在其他线程中执行的一组操作完成的同步辅助。

当计数器变成0之后，await方法被唤醒，才会执行await方法后面的代

A CountdownLatch用给定的初始值初始化。await方法阻塞，直到由于countDown()方法的调用而导致当前计数达到零，之后所有等待线程被释放，并且任何后续的await调用立即返回。这是一个一次性的现象 - 计数无法重置。如果您需要重置计数的版本，请考虑使用CyclicBarrier。

A CountdownLatch是一种通用的同步工具，可用于多种用途。一个CountDownLatch为一个计数的CountDownLatch用作一个简单的开/关锁存器，或者门：所有线程调用await在门口等待，直到被调用countDown()的线程打开。一个CountDownLatch初始化N可以用来做一个线程等待，直到N个线程完成某项操作，或某些动作已经完成N次。

CountDownLatch一个有用的属性是，它不要求调用countDown线程等待计数到达零之前继续，它只是阻止任何线程通过await，直到所有线程可以通过。

示例用法：这是一组类，其中一组工作线程使用两个倒计时锁存器：

- 第一个是启动信号，防止任何工作人员进入，直到驾驶员准备好继续前进；
- 第二个是完成信号，在连司机完成所有的工作人员完成

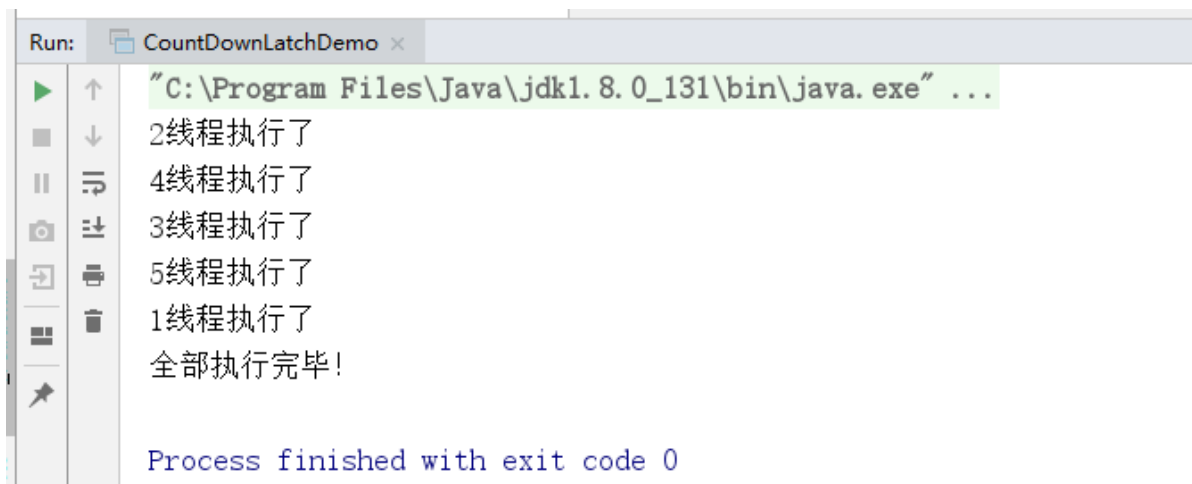
```
//计数器
public class CountdownLatchDemo{
    public static void main(String[] args) throws InterruptedException {
        //总计数的线程是5，一般在必须执行任务的时候再使用
        CountdownLatch countDownLatch = new CountdownLatch(5);

        for (int i = 0; i < 5 ; i++) {
            new Thread(()->{
                System.out.println(Thread.currentThread().getName() + "线程执行
了");

                countDownLatch.countDown();//总数减一
            },String.valueOf(i + 1)).start();
        }

        countDownLatch.await();//只有计数器为0的时候才会唤醒await方法，再继续向下执行，否
则就会一直等待

        System.out.println("全部执行完毕！");
    }
}
```



```
Run: CountdownLatchDemo x
"C:\Program Files\Java\jdk1.8.0_131\bin\java.exe" ...
2线程执行了
4线程执行了
3线程执行了
5线程执行了
1线程执行了
全部执行完毕!
Process finished with exit code 0
```

## 小结:

- 1、CountDownLatch()的参数是需要执行线程个数的计数器
- 2、countDownLatch.countDown();//总数减一
- 3、countDownLatch.await();//只有计数器为0的时候才会唤醒await方法，再继续向下执行，否则就会一直等待

## CyclicBarrier (理解: 加法计数器)

```
public class CyclicBarrier
extends Object
```

允许一组线程全部等待彼此达到共同屏障点的同步辅助。循环阻塞在涉及固定大小的线程方的程序中很有用, 这些线程必须偶尔等待彼此。屏障被称为循环, 因为它可以在等待的线程被释放之后重新使用。

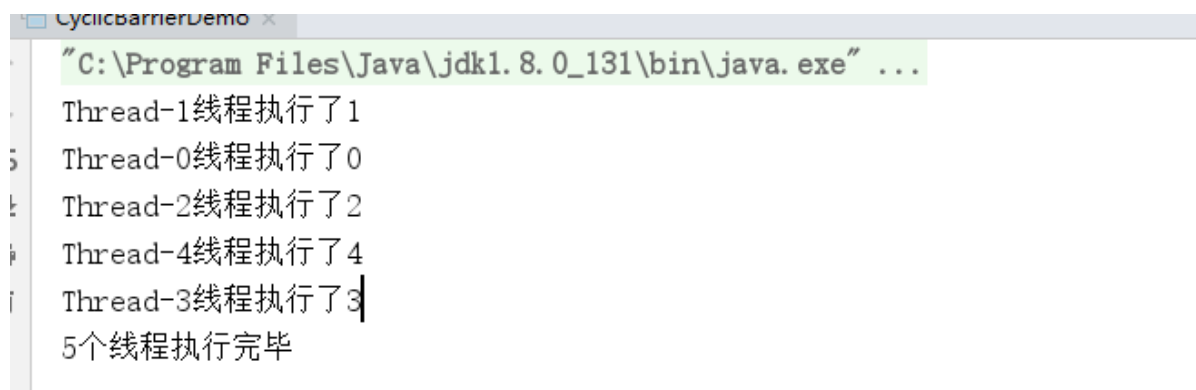
A CyclicBarrier 支持一个可选的 Runnable 命令, 每个屏障点运行一次, 在派对中的最后一个线程到达之后, 但在任何线程释放之前。在任何一方继续进行之前, 此屏障操作对更新共享状态很有用。

示例用法: 以下是在并行分解设计中使用障碍的示例:

```
class Solver { final int N; final float[][] data; final CyclicBarrier barrier; class Worker implements Runnable { int myRow; Worker(int row) { myRow
```

```
//加法计数器
public class CyclicBarrierDemo {
    public static void main(String[] args) {
        CyclicBarrier cyclicBarrier = new CyclicBarrier(5,()->{
            System.out.println("5个线程执行完毕");
        });

        for (int i = 0; i < 5; i++) {
            final int temp = i;
            new Thread()->{
                System.out.println(Thread.currentThread().getName() + "线程执行了"
+ temp); //这里要想取到i的值, 不能够直接拿到, 需要用final关键字修饰的才能拿到
            } try {
                cyclicBarrier.await(); //当线程达到了5才会执行CyclicBarrier构造方法
                //中的Runnable接口的run方法
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (BrokenBarrierException e) {
                e.printStackTrace();
            }
        }
    }
}
```



```
CyclicBarrierDemo x
"C:\Program Files\Java\jdk1.8.0_131\bin\java.exe" ...
Thread-1线程执行了1
Thread-0线程执行了0
Thread-2线程执行了2
Thread-4线程执行了4
Thread-3线程执行了3
5个线程执行完毕
```

理解: cyclicBarrier.await(); : 如果线程没有到5个, 那么就不会唤醒CyclicBarrier构造方法中的Runnable接口的run方法

## Semaphore (信号量)

```
public class Semaphore
extends Object
implements Serializable
```

即设置初始容量，当获取了通行证才可以执行，如果容量满了，就需要等待，等待其他线程释放通行证才能再次拿到通行证

一个计数信号量。在概念上，信号量维持一组许可证。如果有必要，每个`acquire()`都会阻塞，直到许可证可用，然后才能使用它。每个`release()`添加许可证，潜在地释放阻塞获取方。但是，没有使用实际的许可证对象；`Semaphore`只保留可用数量的计数，并相应地执行。

信号量通常用于限制线程数，而不是访问某些（物理或逻辑）资源。例如，这是一个使用信号量来控制对一个项目池的访问的类：

```
class Pool { private static final int MAX_AVAILABLE = 100; private final Semaphore available = new Semaphore(MAX_AVAILABLE, true); public Object getI
```

在获得项目之前，每个线程必须从信号量获取许可证，以确保某个项目可用。当线程完成该项目后，它将返回到池中，并将许可证返回到信号量，允许另一个线程获取该项目。请注意，当调用`acquire()`时，不会保持阻塞，因为这将阻止某个项目返回到池中。信号量封装了限制对池的访问所需的同步，与保持池本身一致性所需的任何同步分开。

```
public class SemaphoreDemo {
    public static void main(String[] args) {
        //设置初始容量只能有3个线程。
        Semaphore semaphore = new Semaphore(3);
        for (int i = 0; i < 8; i++) { //总共8个线程要开启
            new Thread()->{
                //acquire() 获取许可证
                //release() 释放许可证
                try {
                    semaphore.acquire(); //获取许可证
                    System.out.println(Thread.currentThread().getName() + "线程取
得了许可证");

                    TimeUnit.SECONDS.sleep(2); //等待两秒
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    System.out.println(Thread.currentThread().getName() + "线程释
放了许可证");
                    semaphore.release(); //释放
                }
            }, String.valueOf(i + 1)).start();
        }
    }
}
```

## 理解

`semaphore.acquire()`: 获得通信证，如果已经满了，则等待有通行证呗释放了为止

`semaphore.release()`: 释放通行证，会将当前信号量 +1 然后唤醒等待中的线程

作用：多个共享资源互斥使用！并发限流，控制最大的线程数！

## 读写锁（ReadWriteLock）



所有已知实现类:  
ReentrantReadWriteLock

只有一个实现类

理解: 读锁: 可以由多个线程同时进行, 写锁: 只能有一个线程操作

public interface ReadWriteLock

A ReadWriteLock 维护一对关联的 locks, 一个用于只读操作, 一个用于写入, read lock 可以由多个阅读器线程同时进行, 只要没有作者。 write lock 是独家的。

所有 ReadWriteLock 实现必须保证的存储同步效应 writeLock 操作 (如在指定 lock 接口) 也保持相对于所述相关联的 readLock。也就是说, 一个线程成功获取读锁定会看到在之前发布的写锁所做的所有更新。

读写锁允许访问共享数据时的并发性高于互斥锁所允许的并发性。它利用了这样一个事实: 一次只有一个线程 (写入线程) 可以修改共享数据, 在许多情况下, 任何数量的线程都可以同时读取数据 (因此减少线程)。从理论上讲, 通过使用读写锁允许的并发性增加将导致性能改进超过使用互斥锁。实际上, 并发性增加只能在多处理器上完全实现, 然后只有在共享数据的访问模式是合适的时才可以。

读写锁是否会提高使用互斥锁的性能取决于数据被读取的频率与被修改的频率相比, 读取和写入操作的持续时间以及数据的争用 - 即是, 将尝试同时读取或写入数据的线程数。例如, 最初填充

```
/**
 * 独占锁 (写锁): 一次只能被一个线程占用
 * 共享锁 (读锁): 可以被多个线程共同占用
 * 读 - 读: 可以共存
 * 读 - 写: 可以共存
 * 写 - 写: 不能共存
 */
public class ReadWriteLockDemo {
    public static void main(String[] args) {
        // MyCache cache = new MyCache();
        MyCacheLock cache = new MyCacheLock();

        //写入
        for (int i = 0; i < 5; i++) {
            final int temp = i + 1;
            new Thread(() -> {
                cache.put(temp + "", temp + "");
            }, String.valueOf(i + 1)).start();
        }

        //读取
        for (int i = 0; i < 5; i++) {
            final int temp = i + 1;
            new Thread(() -> {
                cache.get(temp + "");
            }, String.valueOf(i + 1)).start();
        }
    }
}

//自定义缓存类
class MyCache {
    private volatile Map<String, Object> map = new HashMap<>();

    public void put(String key, Object value) {
        System.out.println(Thread.currentThread().getName() + "线程写入" + value);
        map.put(key, value);
        System.out.println(Thread.currentThread().getName() + "线程完毕");
    }

    public void get(String key) {
        System.out.println(Thread.currentThread().getName() + "线程读出" +
        map.get(key));
        System.out.println(Thread.currentThread().getName() + "线程读完");
    }
}
```

```

    }
}

//添加读写锁的缓存类
class MyCacheLock {
    private volatile Map<String, Object> map = new HashMap<>();
    //new 读写锁对象
    private ReadWriteLock readWriteLock = new ReentrantReadWriteLock();

    public void put(String key, Object value) {
        //写锁上锁
        readWriteLock.writeLock().lock();
        try {
            System.out.println(Thread.currentThread().getName() + "线程写入" +
value);
            map.put(key, value);
            System.out.println(Thread.currentThread().getName() + "线程完毕");
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            //写锁解锁
            readWriteLock.writeLock().unlock();
        }
    }

    public void get(String key) {
        //读锁上锁
        readWriteLock.readLock().lock();
        try {
            System.out.println(Thread.currentThread().getName() + "线程读出" +
map.get(key));
            System.out.println(Thread.currentThread().getName() + "线程读完毕");
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            //读锁解锁
            readWriteLock.readLock().unlock();
        }
    }
}
}

```

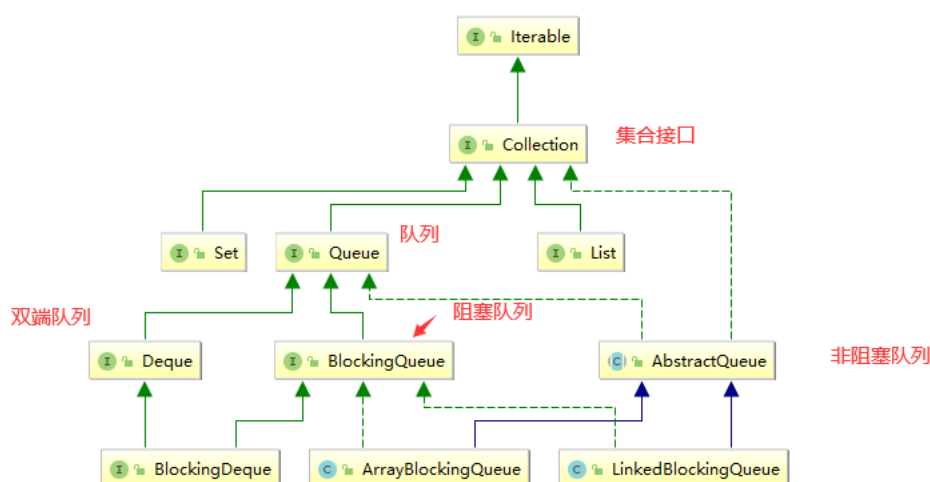
↓  
2线程写入2  
2线程完毕  
1线程写入1  
1线程完毕  
3线程写入3  
3线程完毕  
4线程写入4  
4线程完毕  
5线程写入5  
5线程完毕  
1线程读出1  
1线程读完毕  
2线程读出2  
2线程读完毕  
3线程读出3  
3线程读完毕  
5线程读出5  
4线程读出4  
4线程读完毕  
5线程读完毕

写入的时候是锁住的，只能一个线程一个线程操作，而读的时候是共享的，可以同时多个线程同时操作

## 阻塞队列 (BlockingQueue)

**阻塞队列**是这样的一种数据结构，它是一个队列（类似于一个List），可以存放0到N个元素。我们可以对这个队列执行插入或弹出元素操作，弹出元素操作就是获取队列中的第一个元素，并且将其从队列中移除；而插入操作就是将元素添加到队列的末尾。当队列中没有元素时，对这个队列的弹出操作将会被阻塞，直到有元素被插入时才会被唤醒；当队列已满时，对这个队列的插入操作就会被阻塞，直到有元素被弹出后才会被唤醒。

在线程池中，往往就会用阻塞队列来保存那些暂时没有空闲线程可以直接执行的任务，等到线程空闲之后再从阻塞队列中弹出任务来执行。一旦队列为空，那么线程就会被阻塞，直到有新任务被插入为止。



## 4组API

方式	抛出异常	有返回值, 不抛出异常	阻塞等待	超时等待
添加	add(E)	offer(E)	put(E)	offer(time(等待时间),TimeUnit(等待单位))
移除	remove()	poll()	take()	poll(long time, TimeUnit unit)
获取队首元素	element()	peek()	-	-

```
/**
 * 抛出异常
 */
public static void test01() {
    //其中3代表队列固定容量为3
    BlockingQueue<Object> blockingQueue = new ArrayBlockingQueue<>(3);
    //插入队列
    System.out.println(blockingQueue.add("A"));
    System.out.println(blockingQueue.add("B"));
    System.out.println(blockingQueue.add("C"));
    //获取队首元素
    System.out.println(blockingQueue.element());
    //插入超出容量的元素报错 java.lang.IllegalStateException: Queue full
    //
    System.out.println(blockingQueue.add("D"));
    System.out.println("=====");
    System.out.println(blockingQueue.remove());
    System.out.println(blockingQueue.remove());
    System.out.println(blockingQueue.remove());
    //移除空队列中的元素报错 java.util.NoSuchElementException
    System.out.println(blockingQueue.remove());
}

/**
 * 不抛出异常, 有返回值
 */
public static void test02() {
    //其中3代表队列固定容量为3
    BlockingQueue<Object> blockingQueue = new ArrayBlockingQueue<>(3);
    //插入队列
    System.out.println(blockingQueue.offer("A"));
    System.out.println(blockingQueue.offer("B"));
    System.out.println(blockingQueue.offer("C"));
    //插入超出容量元素: 返回false
    System.out.println(blockingQueue.offer("D"));

    System.out.println("=====");
    //获取队首元素
    System.out.println(blockingQueue.peek());
    //取出元素
    System.out.println(blockingQueue.poll());
    System.out.println(blockingQueue.poll());
    System.out.println(blockingQueue.poll());
}
```

```

        //取出空队列的元素：返回null
        System.out.println(blockingQueue.poll());
    }

    /**
     * 阻塞等待，条件不满足会一直等待
     */
    public static void test03() throws InterruptedException {
        //其中3代表队列固定容量为3
        BlockingQueue<Object> blockingQueue = new ArrayBlockingQueue<>(3);

        //插入元素
        blockingQueue.put("A");
        blockingQueue.put("B");
        blockingQueue.put("C");
        //插入已满队列的元素，如果期间没有元素弹出则会一直等待
        //    blockingQueue.put("D");
        System.out.println("=====");
        //取出元素
        System.out.println(blockingQueue.take());
        System.out.println(blockingQueue.take());
        System.out.println(blockingQueue.take());
        //取出空队列的元素，如果一直为空队列，则也会一直等待
        //    system.out.println(blockingQueue.take());

    }

    /**
     * 超时等待，设置等待时间，如果时间到了就继续向下执行而不在等待
     */
    public static void test04() throws InterruptedException {
        //其中3代表队列固定容量为3
        BlockingQueue arrayBlockingQueue = new ArrayBlockingQueue<>(3);
        //添加元素
        System.out.println(arrayBlockingQueue.offer("A"));
        System.out.println(arrayBlockingQueue.offer("B"));
        System.out.println(arrayBlockingQueue.offer("C"));
        //超时等待，添加满队列元素，并且设置等待时间为3秒
        System.out.println(arrayBlockingQueue.offer("D", 3, SECONDS));
        System.out.println("=====");
        System.out.println(arrayBlockingQueue.poll());
        System.out.println(arrayBlockingQueue.poll());
        System.out.println(arrayBlockingQueue.poll());
        //超时等待，获取空队列元素并设置等待时间为2秒
        System.out.println(arrayBlockingQueue.poll(2, SECONDS));
    }
}

```

## SynchronousQueue 同步队列

Serializable , Iterable <E>, Collection <E>, BlockingQueue <E>, Queue <E>

```

public class SynchronousQueue<E>
    extends AbstractQueue<E>
    implements BlockingQueue<E>, Serializable

```

A blocking queue 其中每个插入操作必须等待另一个线程相应的删除操作，反之亦然。同步队列没有任何内部容量，甚至没有一个容量。你不能peek在同步队列，因为一个元素，当您尝试删除它才存在；您无法插入元素（使用任何方法），除非另有线程正在尝试删除它；你不能迭代，因为没有什么可以迭代。队列的头是第一个排队的插入线程尝试添加到队列中的元素；如果没有这样排队的线程，那么没有元素可用于删除，并且poll()将返回null。为了其他Collection方法（例如contains）的目的，SynchronousQueue充当空集合。此队列不允许null元素。

同步队列类似于CSP和Ada中使用的会合通道。它们非常适用于切换设计，其中运行在一个线程中的对象必须与在另一个线程中运行的对象同步，以便交付一些信息，事件或任务。

此类支持可选的公平策略，用于订购等待的生产者和消费者线程。默认情况下，此订单不能保证。然而，以公平设置为true的队列以FIFO顺序授予线程访问权限。



```

/**
 * 同步队列和其他的BlockingQueue不一样，SynchronousQueue不会存储元素
 * 其中put了一个元素，想要再次put就会进入等待，必须要先take取出了才可以继续put元素，否则不能再
put进去值
 */
public class SynchronousQueueDemo {
    public static void main(String[] args) {
        //同步队列，没有初始化容量，容量只有1
        BlockingQueue<String> blockingQueue = new SynchronousQueue<>();

        new Thread(() -> {
            try {
                System.out.println(Thread.currentThread().getName() + "=> put
A");

                blockingQueue.put("A");
                System.out.println(Thread.currentThread().getName() + "=> put
B");

                blockingQueue.put("B");
                System.out.println(Thread.currentThread().getName() + "=> put
C");

                blockingQueue.put("C");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, "Thread1").start();
        new Thread(() -> {
            try {
                TimeUnit.SECONDS.sleep(2);
                System.out.println(Thread.currentThread().getName() + "=> "
+blockingQueue.take());
                TimeUnit.SECONDS.sleep(2);
                System.out.println(Thread.currentThread().getName() + "=> "
+blockingQueue.take());
                TimeUnit.SECONDS.sleep(2);
                System.out.println(Thread.currentThread().getName() + "=> "
+blockingQueue.take());
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, "Thread2").start();
    }
}

```

## 线程池

三大方法，七大参数，四种拒绝策略

池化技术

程序的运行本质：占用系统资源，需要优化资源的使用！==>池化技术

例如线程池，连接池，内存池。。。

## 线程池的好处

- 1、降低资源消耗
- 2、提高响应的速度
- 3、方便管理

(线程复用，可控最大并发数，管理线程)

```
//Executors工具类，三大方法
public class Demo01 {
    public static void main(String[] args) {
        // ExecutorService threadPool = Executors.newSingleThreadExecutor();//单个线程
        ExecutorService threadPool = Executors.newFixedThreadPool(5);//创建一个固定的线程池大小,5个线程
        // ExecutorService threadPool = Executors.newCachedThreadPool();//可伸缩的线程池

        try {
            //使用了线程池后就用线程池来创建线程
            for (int i = 0; i < 10; i++) {
                threadPool.execute(() -> {
                    System.out.println(Thread.currentThread().getName() + "执行");
                });
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            //关闭线程池
            threadPool.shutdown();
        }
    }
}
```

4. **【强制】**线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。

说明：Executors 返回的线程池对象的弊端如下：

- 1) **FixedThreadPool** 和 **SingleThreadPool**：

允许的请求队列长度为 Integer.MAX\_VALUE，可能会堆积大量的请求，从而导致 OOM。

- 2) **CachedThreadPool**：

允许的创建线程数量为 Integer.MAX\_VALUE，可能会创建大量的线程，从而导致 OOM。

**【强制】**线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。说明：Executors 返回的线程池对象的弊端如下：1) FixedThreadPool 和 SingleThreadPool：允许的请求队列长度为 Integer.MAX\_VALUE，可能会堆积大量的请求，从而导致 OOM。2) CachedThreadPool：允许的创建线程数量为 Integer.MAX\_VALUE，可能会创建大量的线程，从而导致 OOM。

源码分析：

```

public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(1, 1, //初始核心线程和最大线程都为1
                                0L, TimeUnit.MILLISECONDS,
                                new LinkedBlockingQueue<Runnable>()));
}

public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads, //初始线程和核心线程都为
nThreads数
                                0L, TimeUnit.MILLISECONDS,
                                new LinkedBlockingQueue<Runnable>());
}

public static ExecutorService newCachedThreadPool(ThreadFactory threadFactory) {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE, //初始线程数为0, 最大线程
数为约21亿
                                60L, TimeUnit.SECONDS,
                                new SynchronousQueue<Runnable>(),
                                threadFactory);
}

//可以看到不管是哪种创建线程池的方法都是创建了ThreadPoolExecutor类
//查看ThreadPoolExecutor类
public ThreadPoolExecutor(int corePoolSize, //初始核心线程数
                           int maximumPoolSize, //最大线程数
                           long keepAliveTime, //等待时间（开辟了新的线程后，如果在等待
时间内线程没有再执行任务，开启的线程就会自动结束）
                           TimeUnit unit, //等待单位
                           BlockingQueue<Runnable> workQueue, //阻塞队列
                           ThreadFactory threadFactory //拒绝策略（4种） {
    this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
          threadFactory, defaultHandler);
}

```

## 测试创建自定义线程池

```

public class ThreadPoolDemo02 {
    public static void main(String[] args) {
        ExecutorService threadPool = new ThreadPoolExecutor(
            3, //初始线程为3个
            5, //最大线程数为5个
            3, //等待时间
            TimeUnit.SECONDS, //等待单位
            new ArrayBlockingQueue<>(3), //设置阻塞队列，并且阻塞队列的默认大小为3
            // new ThreadPoolExecutor.AbortPolicy() //拒绝策略,这里是如果队列满了，
            // 就不处理并且抛出异常（默认）
            // new ThreadPoolExecutor CallerRunsPolicy() //拒绝策略,这里是如果队列
            // 满了，就将新进来的线程丢回来源的线程即主线程执行
            // new ThreadPoolExecutor.DiscardPolicy() //拒绝策略,这里是如果队列满
            // 了，就不处理并且不会抛出异常
            new ThreadPoolExecutor.DiscardOldestPolicy() //拒绝策略,这里是如果队
            // 列满了，就会尝试争夺最前面的线程资源，如果没有抢到也会不处理并且不会抛出异常
        );

        try {
            //使用了线程池后就用线程池来创建线程

```

```

        for (int i = 0; i < 9; i++) { //根据上面开启的线程池，当线程池小于等于3时会
直接执行，
            // 当线程大于3小于等于6时多余的线程会进入阻塞队列等待，
            // 当大于6小于等于8时会开启新的线程，8等于最大线程数加上阻塞队列数
            // 当大于8的时候会出发拒绝策略
            threadPool.execute(() -> {
                System.out.println(Thread.currentThread().getName() + "执
行");
            });
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        //关闭线程池
        threadPool.shutdown();
    }
}
}

```

#### 4种拒绝策略

- 1、new ThreadPoolExecutor.AbortPolicy(): 如果队列满了，就不处理并且抛出异常（默认）
- 2、new ThreadPoolExecutor.CallerRunsPolicy(): 如果队列满了，就将新进来的线程丢回来源的线程即主线程执行
- 3、new ThreadPoolExecutor.DiscardPolicy(): 如果队列满了，就不处理并且不会抛出异常
- 4、new ThreadPoolExecutor.DiscardOldestPolicy(): 如果队列满了，就会尝试争夺最前面的线程资源，如果没有抢到也会不处理并且不会抛出异常（有可能会被执行）

#### 问题：创建线程池的时候最大线程数如何定义？

- 1、CPU密集型：判断CPU的核数，几核就是几个最大线程数，可以保持CPU的效率最高
- 2、IO密集型：线程数 > 程序中十分耗资源的线程，因为IO是十分耗资源的

```

//获取CPU核数
Runtime.getRuntime().availableProcessors()

```

## 四大函数式接口

lambda表达式、函数式接口、链式编程、Stream流式计算

函数式接口：只有一个方法的接口

```

@FunctionalInterface //简化编程模型，新版框架底层大量应用
public interface Runnable {
    public abstract void run();
}

```

**Interfaces**

*BiConsumer*  
*BiFunction*  
*BinaryOperator*  
*BiPredicate*  
*BooleanSupplier*  
***Consumer***  
*DoubleBinaryOperator*  
*DoubleConsumer*  
*DoubleFunction*  
*DoublePredicate*  
*DoubleSupplier*  
*DoubleToIntFunction*  
*DoubleToLongFunction*  
***DoubleUnaryOperator***  
***Function***  
*IntBinaryOperator*  
*IntConsumer*  
*IntFunction*  
*IntPredicate*  
*IntSupplier*  
*IntToDoubleFunction*  
*IntToLongFunction*  
*IntUnaryOperator*  
*LongBinaryOperator*  
*LongConsumer*  
*LongFunction*  
*LongPredicate*  
*LongSupplier*  
*LongToDoubleFunction*  
*LongToIntFunction*  
*LongUnaryOperator*  
*ObjDoubleConsumer*  
*ObjIntConsumer*  
*ObjLongConsumer*  
***Predicate***  
***Supplier***  
*ToDoubleBiFunction*  
*ToDoubleFunction*  
*ToIntBiFunction*  
*ToIntFunction*  
*ToLongBiFunction*  
*ToLongFunction*  
*UnaryOperator*

四大函数式接口

函数型接口：两个参数，一个为apply的参数，一个为apply的返回值

```

    */
    @FunctionalInterface
    public interface Function<T, R> {

        /**
         * Applies this function to the given argument.
         *
         * @param t the function argument
         * @return the function result
         */
        R apply(T t);

        // ...
    }

```

```

/**
 * Function 函数型接口，一个输入参数一个输出参数
 * 只要是函数式接口就可以用lambda表达式简化
 */
public class Demo01 {
    public static void main(String[] args) {
        //可以作为工具类
        /*Function<String, String> function = new Function<String, String>() {
            @Override
            public String apply(String s) {
                return s;
            }
        };*/

        //      Function<String, String> function = (s)->{return s;};
        Function<String, String> function = s -> s; //最简洁

        System.out.println(function.apply("hello world!"));
    }
}

```

Predicate 断定型接口：一个输入参数，返回的是boolean类型

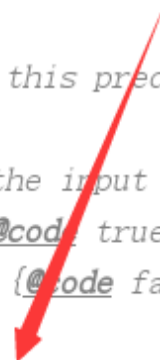
```

    */
    @FunctionalInterface 函数式接口
    public interface Predicate<T> {

        /**
         * Evaluates this predicate on the given argument.
         *
         * @param t the input argument
         * @return {@code true} if the input argument matches the pr
         * otherwise {@code false}
         */
        boolean test(T t);
    }

```

只会返回boolean类型



```

/**
 * 断定型接口：一个为输入参数，返回的只有一个boolean类型
 */
public class Demo02 {
    public static void main(String[] args) {
        //判断字符串是否为空
        /*Predicate<String> predicate = new Predicate<String>() {
            @Override
            public boolean test(String s) {
                if (s.isEmpty())
                    return true;
                else return false;
            }
        };*/

        /*Predicate<String> predicate = (str)->{
            if (str.isEmpty())
                return true;
            else return false;
        };*/

        Predicate<String> predicate = str -> str.isEmpty();

        System.out.println(predicate.test("asdadsd"));
    }
}

```

Consumer 消费型接口

```
    */
    @FunctionalInterface
    public interface Consumer<T> {

        /**
         * Performs this operation on the given argument.
         *
         * @param t the input argument
         */
        void accept(T t);
    }
}
```

没有返回值

```
/**
 * Consumer: 消费型接口，一个参数，没有返回值
 */
public class Demo03 {
    public static void main(String[] args) {
        /*Consumer<String> consumer = new Consumer<String>() {
            @Override
            public void accept(String string) {
                System.out.println(string);
            }
        };*/

        // Consumer<String> consumer = (string) -> {System.out.println(string)};
        Consumer<String> consumer = string -> System.out.println(string);

        consumer.accept("hello world!!");
    }
}
```

#### Supplier 供给型接口

```
@FunctionalInterface
public interface Supplier<T> {

    /**
     * Gets a result.
     *
     * @return a result
     */
    T get();
}
```

没有参数，只有返回值

```
/**
 * Supplier : 供给型接口，没有参数，只有返回值

```



```

*/
public class Demo04 {
    public static void main(String[] args) {
        /*Supplier<String> supplier = new Supplier<String>() {
            @Override
            public String get() {
                return "你好 世界!!! ";
            }
        };*/

        //      Supplier<String> supplier = () -> {return "你好 世界!!! ";};
        Supplier<String> supplier = () -> "你好 世界!!! ";
        System.out.println(supplier.get());
    }
}

```

## Stream流式计算

### 什么是Stream流式计算

集合、mysql本质就是存储数据，计算都应该交给流来操作！

```

/**
 * 问题：
 * 1、ID必须是偶数
 * 2、年龄大于23岁
 * 3、用户名转为大写字母
 * 4、用户名字倒序排序
 * 5、只输出一个用户（分页）
 */
public class Test {
    public static void main(String[] args) {
        User user1 = new User(1, "a", 21);
        User user2 = new User(2, "b", 22);
        User user3 = new User(3, "c", 23);
        User user4 = new User(4, "d", 24);
        User user5 = new User(5, "e", 25);
        User user6 = new User(6, "f", 26);
        //转为集合，存储
        List<User> users = Arrays.asList(user1, user2, user3, user4, user5,
user6);

        //流用来计算
        users.stream().filter((user) -> {return user.getId() % 2 == 0;}) //返回Id
为偶数的用户

        .filter((user) -> {return user.getAge() > 23;}) //返回年龄大于23的
用户

        .map((user) -> {return user.getName().toUpperCase();})//名字转为大
写字母

        .sorted((u1, u2) -> {return u2.compareTo(u1);})//倒序
        .limit(1)//分页
        .forEach(System.out :: println);
    }
}

```

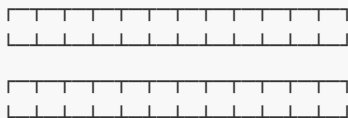
# Forkjoin

Java 7开始引入了一种新的Fork/Join线程池，它可以执行一种特殊的任务：把一个大任务拆成多个小任务并行执行。

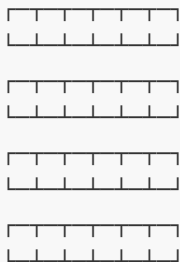
我们举个例子：如果要计算一个超大数组的和，最简单的做法是用一个循环在一个线程内完成：



还有一种方法，可以把数组拆成两部分，分别计算，最后加起来就是最终结果，这样可以用两个线程并行执行：



如果拆成两部分还是很大，我们还可以继续拆，用4个线程并行执行：



这就是Fork/Join任务的原理：判断一个任务是否足够小，如果是，直接计算，否则，就分拆成几个小任务分别计算。这个过程可以反复“裂变”成一系列小任务。

```
public class ForkJoinDemo extends RecursiveTask<Long> {
    // private Long start;
    private long start;
    // private Long end;
    private long end;
    //临界值
    // private Long temp = 10000L;
    private long temp = 10000L;

    public ForkJoinDemo(long start, long end) {
        this.start = start;
        this.end = end;
    }
    /*public ForkJoinDemo(Long start, Long end) {
        this.start = start;
        this.end = end;
    }*/

    @Override
    protected Long compute() {
        if ((end - start) >= temp) {
```

```

//      Long mid = (start + end) / 2;
      long mid = (start + end) / 2;
      ForkJoinDemo task1 = new ForkJoinDemo(start, mid);
      task1.fork();
      ForkJoinDemo task2 = new ForkJoinDemo(mid + 1, end);
      task2.fork();
      return task1.join() + task2.join();
    }else {
//      Long sum = 0L;
      long sum = 0L;
      for (Long i = start; i <= end; i++) {
        sum += i;
      }
      return sum;
    }
  }
}

```

---

```

public class UseDemo {
    public static void main(String[] args) throws ExecutionException,
        InterruptedException {
//      test1(); //10974 如果直接使用基本类型long为3733
      test2(); //5378 如果直接使用基本类型long为2350
//      test3(); //232

    }

    public static void test1() {
        long start = System.currentTimeMillis();

        Long sum = 0L;
        /*for (Long i = 1L; i <= 10_0000_0000L; i++) { //如果用long的包装类Long会慢
大约3倍
            sum += i;
        }*/
        for (long i = 1L; i <= 10_0000_0000L; i++) {
            sum += i;
        }
        long end = System.currentTimeMillis();
        System.out.println("sum=" + sum + ",usedtimes:" + (end - start));
    }

    public static void test2() throws ExecutionException, InterruptedException {
        long start = System.currentTimeMillis();
        ForkJoinPool forkJoinPool = new ForkJoinPool();
//      forkJoinPool.execute();//执行没有返回值
        ForkJoinTask<Long> task = new ForkJoinDemo(0L, 10_0000_0000L);
        ForkJoinTask<Long> submit = forkJoinPool.submit(task);//提交有返回值
        long sum = submit.get();
        long end = System.currentTimeMillis();
        System.out.println("sum=" + sum + ",usedtimes:" + (end - start));
    }

    public static void test3() {
        long start = System.currentTimeMillis();

```

```

        //stream并行流
        //range() rangeClosed()
        long sum = LongStream.rangeClosed(0L,
10_0000_0000L).parallel().reduce(0, Long::sum);
        long end = System.currentTimeMillis();

        System.out.println("sum=" + sum + ",usedtimes:" + (end - start));
    }

}

```

## 异步回调

compact1, compact2, compact3  
java.util.concurrent

**Class CompletableFuture<T>**

java.lang.Object  
java.util.concurrent.CompletableFuture<T>

All Implemented Interfaces:

CompletionStage <T>, Future <T>

```

public class CompletableFuture<T>
extends Object
implements Future<T>, CompletionStage<T>

```

甲Future可以明确地完成（设定其值和状态），并且可以被用作CompletionStage，支持有关的功能和它的完成时触发动作。

当两个或多个线程试图complete，completeExceptionally，或cancel一个CompletableFuture，只有一个成功。

除了直接操作状态和结果的这些和相关方法外，CompletableFuture还实现了接口CompletionStage，具有以下策略：

- 为异步方法的依赖完成提供的操作可以由完成当前CompletableFuture的线程或完成方法的任何其他调用者执行。
- 所有不使用显式Executor参数的异步方法都使用ForkJoinPool.commonPool()执行（除非它不支持至少两个并行级别，在这种情况下，使用新的线程）。为了简化监视，调试和跟踪，所有生成的异步任务都是标记接口CompletableFuture.AsynchronousCompletionTask的实例。
- 所有CompletionStage方法都是独立于其他公共方法实现的，因此一个方法的行为不会受到子类中其他方法的覆盖的影响。

CompletableFuture还实施Future，具有以下政策：

```

public class FutureDemo {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        //没有返回值
        /*CompletableFuture<Void> completableFuture =
CompletableFuture.runAsync(() -> {
            try {
                TimeUnit.SECONDS.sleep(2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName() + "==> runAsync
return void");
        });

        System.out.println("执行");
        completableFuture.get();//获取结果*/

        //有返回值
        CompletableFuture<String> completableFuture =
CompletableFuture.supplyAsync(() -> {
            System.out.println(Thread.currentThread().getName() + "==> runAsync
return String");
        });
        //            int i = 1 / 0;
        return "hello world! ";
    });
}

```

阿里云-服务器优惠  
腾讯云-服务器优惠

```

completableFuture.whenComplete((t,u) -> { //正常执行
    System.out.println("t =>" + t); //正常执行返回结果
    System.out.println("u =>" + u); //异常执行返回信息
}).exceptionally((e) -> { //异常执行
    System.out.println(e.getMessage());
    return "exception"; //如果错误了可以获取到错误的返回结果
}).get();

//      System.out.println(completableFuture.get());
    }
}

```

## JMM

### 什么是JMM

JMM: java内存模型, 不存在的东西, 概念, 约定!

内存模型可以理解为在特定的操作协议下, 对特定的内存或者高速缓存进行读写访问的过程抽象描述, 不同架构下的物理机拥有不一样的内存模型, Java虚拟机是一个实现了跨平台的虚拟系统, 因此它也有自己的内存模型, 即Java内存模型 (Java Memory Model, JMM) 。



### 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。就是对变量实施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

## Volatile

Volatile是java虚拟机提供的轻量级的同步机制

- 1、保证可见性
- 2、不保证原子性
- 3、禁止指令重排

### 保证可见性

```
public class JMMDemo01 {
    private volatile static int i = 0; //不加volatile就会出现死循环，而加了volatile就会保证可见性

    public static void main(String[] args) throws InterruptedException { //主线程

        new Thread(() -> { //新线程对主内存的变化是不知道的
            while (i == 0){

            }
        }).start();

        TimeUnit.SECONDS.sleep(1);

        i = 1; //主线程改变i的值
        System.out.println(i);
    }
}
```

### 不保证原子性

原子性：不可分割

```
public class VolatileDemo02 {

    private volatile static int num = 0; //volatile是不保证原子性的，10个线程最终结果并不是10000

    /*public synchronized static void add() { //synchronized可以保证原子性
        num++;
    }*/

    /*static Lock lock = new ReentrantLock();

    public static void add() { //lock也可以保证原子性
        lock.lock();
        try {
```

```

        num++;
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}*/

public static void main(String[] args) {
    for (int i = 0; i < 10; i++) {
        new Thread(() -> {
            for (int j = 0; j < 1000; j++) {
                add(); //理论上10个线程执行完应该是10000
            }
        }).start();
    }

    while (Thread.activeCount() > 2) { //默认两个线程main, gc
        Thread.yield();
    }

    System.out.println("num => " + num);
}
}

```

**问题：如果不加lock和synchronized怎样保证原子性？**

java.text.spi  
java.time  
java.time.chrono  
java.time.format  
java.time.temporal  
java.time.zone  
java.util  
java.util.concurrent  
**java.util.concurrent.atomic**  
java.util.concurrent.locks  
java.util.function  
java.util.jar

保证线程原子性包

**java.util.concurrent.atomic**

**Classes**

**AtomicBoolean**  
**AtomicInteger**  
AtomicIntegerArray  
AtomicIntegerFieldUpdater  
**AtomicLong**  
AtomicLongArray  
AtomicLongFieldUpdater  
AtomicMarkableReference  
AtomicReference  
AtomicReferenceArray  
AtomicReferenceFieldUpdater  
AtomicStampedReference  
DoubleAccumulator  
DoubleAdder  
LongAccumulator  
LongAdder

概要: 嵌套 | 字段 | 构造方法 | 方法    详细信息: 字段 | 构造方法 | 方法

compact1, compact2, compact3  
java.util.concurrent.atomic

### Class AtomicInteger

java.lang.Object  
java.lang.Number  
java.util.concurrent.atomic.AtomicInteger

**All Implemented Interfaces:**  
Serializable

---

```
public class AtomicInteger
extends Number
implements Serializable
```

一个int可能原子更新的值。 有关原子变量属性的描述，请替代品。 但是，这个类确实扩展了Number以允许通过处理

**从以下版本开始:**  
1.5

**另请参见:**  
Serialized Form

**构造方法摘要**

```
public class VolatileDemo02 {
```

```
// private static int num = 0;
private static AtomicInteger num = new AtomicInteger();

/*public synchronized static void add() { //synchronized可以保证原子性
    num++;
}*/

/*static Lock lock = new ReentrantLock();

public static void add() { //lock也可以保证原子性
    lock.lock();
    try {
        num++;
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}*/

public static void add() {
//    num++; //不是一个原子性操作
    num.getAndIncrement(); //原子性操作AtomicInteger + 1方法 CAS
}

public static void main(String[] args) {
    for (int i = 0; i < 10; i++) {
        new Thread(() -> {
            for (int j = 0; j < 1000; j++) {
                add(); //理论上10个线程执行完应该是10000
            }
        }).start();
    }

    while (Thread.activeCount() > 2) { //默认两个线程main, gc
        Thread.yield();
    }

    System.out.println("num => " + num);
}
}
```

## 禁止指令重排

**什么是指令重排：程序，计算机并不是按照写的那样执行的。**

源代码-->编译器优化重排-->指令并行也可能会重排-->内存系统也会重排-->执行

处理器在进行指令重排的时候会考虑数据之间的依赖性！

## 例子

int a,b,x,y = 0



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

因为上面的代码，不存在数据的依赖性，因此编译器可能对数据进行重排

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

这样造成的结果，和最开始的就不一致了，这就是导致重排后，结果和最开始的不一样，因此为了防止这种结果出现，volatile就规定禁止指令重排，为了保证数据的一致性

内存屏障，又称内存栅栏，是一个CPU指令，它的作用有两个，一是保证特定操作的执行顺序，二是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。Memory Barrier的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。总之，volatile变量正是通过内存屏障实现其在内存中的语义，即可见性和禁止重排优化。

## 单例模式

饿汉式，懒汉式

饿汉式

```
/**
 * 饿汉式单例
 */
public class HungryMan {

    //可能会浪费空间
    private byte[] bytes1 = new byte[1024 * 1024];
    private byte[] bytes2 = new byte[1024 * 1024];
    private byte[] bytes3 = new byte[1024 * 1024];
    private byte[] bytes4 = new byte[1024 * 1024];

    private static HungryMan HUNGRYMAN = new HungryMan();

    private HungryMan(){};

    private static HungryMan getInstance() {
        return HUNGRYMAN;
    }

}
```

```
/**
 * 懒汉式单例
 */
public class LazyMan {

    //添加变量flag
    private static boolean flag = false;

    private LazyMan() {
        synchronized (LazyMan.class) {
            //通过变量判断对象是否存在
            if (flag == false) {
                flag = true;
            } else {
                throw new RuntimeException("不要试图利用反射破坏异常");
            }
            //构造方法加锁可以防止如果有一个通过反射new的对象不一致
            /*if (LAZYMAN != null) {
                throw new RuntimeException("不要试图利用反射破坏异常");
            }*/
        }
        System.out.println(Thread.currentThread().getName() + "=>ok");
    }

    private volatile static LazyMan LAZYMAN;

    //双重检测锁模式的懒汉式单例 简称DCL懒汉式 (DoubleCheckedLock)
    private static LazyMan getInstance() {
        //加锁
        if (LAZYMAN == null) {
            synchronized (LazyMan.class) {
                if (LAZYMAN == null) {
                    LAZYMAN = new LazyMan(); //不是一个原子性操作
                }
            }
        }

        return LAZYMAN;
    }

    public static void main(String[] args) throws NoSuchMethodException,
        IllegalAccessException, InvocationTargetException, InstantiationException,
        NoSuchFieldException {
        /**多线程并发，不能保证new出来的都是一个对象，这时可以加锁来保证原子性
        for (int i = 0; i < 10; i++) {
            new Thread(() -> {
                LazyMan.getInstance();
            }).start();
        }
        */
    }
}
```

```

    }*/

//    LazyMan instance1 = LazyMan.getInstance();
//反射会破坏单例模式，可以在构造方法上再次加密
Constructor<LazyMan> declaredConstructor =
LazyMan.class.getDeclaredConstructor(null);
//无视私有构造器
declaredConstructor.setAccessible(true);
//就算添加了变量flag也可以通过反射获取flag的值并再次修改
/*Field flag = LazyMan.class.getDeclaredField("flag");
flag.setAccessible(true);*/

//如果两个对象都是通过反射来new的，则无参数构造加锁又不管用了，这样可以添加一个变量
flag，通过变量来验证对象是否已经存在
LazyMan instance1 = declaredConstructor.newInstance();
/*//再次修改flag的值
flag.set(instance1, false);*/
LazyMan instance2 = declaredConstructor.newInstance();

System.out.println(instance1);
System.out.println(instance2);

    }
}

```

发现不管对其怎样加锁或验证都无法阻挡反射机制破坏原子性

解决：

枚举类型：

```

//enum的本质也是一个class
public enum EnumSingle {

    INSTANCE;

    private static EnumSingle getInstance(){
        return INSTANCE;
    }

    public static void main(String[] args) throws NoSuchMethodException,
    IllegalAccessException, InvocationTargetException, InstantiationException {
        EnumSingle instance1 = EnumSingle.getInstance();
        //Exception in thread "main" java.lang.NoSuchMethodException:
com.youxin.single.EnumSingle.<init>()
//        Constructor<EnumSingle> declaredConstructor =
EnumSingle.class.getDeclaredConstructor(null);

        //java.lang.IllegalArgumentException: Cannot reflectively create enum
objects//反射不能破坏枚举
        Constructor<EnumSingle> declaredConstructor =
EnumSingle.class.getDeclaredConstructor(String.class, int.class);
        declaredConstructor.setAccessible(true);
        EnumSingle instance2 = declaredConstructor.newInstance();
        System.out.println(instance1);
        System.out.println(instance2);

    }
}

```

# CAS

## 什么是CAS

CAS: 全称Compare and swap, 字面意思:“比较并交换”, 是一种轻量级锁。

线程在读取数据是不进行加锁, 在准备修改数据时, 先去查询原值, 操作的时候比较原值是否被修改, 若未被其他线程修改则写入数据, 若已经被修改, 就要重新执行读取流程。

```
// setup to use Unsafe.compareAndSwapInt for updates
private static final Unsafe unsafe = Unsafe.getUnsafe();
private static final long valueOffset;

static {
    try {
        valueOffset = unsafe.objectFieldOffset
            (AtomicInteger.class.getDeclaredField("value"));
    } catch (Exception ex) { throw new Error(ex); }
}
```

java无法操作内存, 但java可以操作C++, C++可以操作内存, 所以java调用native操作C++再操作内存

通过atomicInteger.getAndIncrement();理解:

```
/* @return the previous value
 */
public final int getAndIncrement() {
    return unsafe.getAndAddInt(o: this, valueOffset, i: 1);
}

/**
```

```
public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        var5 = this.getIntVolatile(var1, var2);
    } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));

    return var5;
}
```

如果当前对象的内存偏移值是var5, 则将内存加一

```
public class CASDemo01 {

    public static void main(String[] args) {
        AtomicInteger atomicInteger = new AtomicInteger(20);

        //CAS compareAndSet : 比较并交换
        //public final boolean compareAndSet(int expect, int update) 期望, 更新
        //达到期望的值就更新, 没有就返回false并不做操作 CAS是CPU的并发原语
        System.out.println(atomicInteger.compareAndSet(20, 21));
        //自增1
        atomicInteger.getAndIncrement();
        System.out.println(atomicInteger.get());

        System.out.println(atomicInteger.compareAndSet(20, 21));
    }
}
```

```

        System.out.println(atomicInteger.get());
    }
}

```

## ABA问题

### 什么是ABA问题

- 1、线程1读取了对象A，线程2也读取了对象A。
- 2、线程2比线程1快，首先将A对象CAS比较替换为B，随后马上又将B改回了A。
- 3、此时线程1拿到了对象A，但早已不是以前的对象A。
- 4、此时，线程1通过CAS比较，发现原对象是A，就改成了自己要改的值。

虽然说线程1最后能操作成功，但是这样已经违背了CAS的初衷，数据已经被修改过了，按CAS的原则来讲，CAS是不应该修改成功的。

## 原子引用（解决ABA问题，可以理解乐观锁概念）

```

public class CASDemo02 {

    public static void main(String[] args) {
        //如果泛型是一个包装类，则注意引用
        //这里的泛型一般为类，但如果是Integer类型而大小在-128~127之外的话，期待替换就会失败，因为每一次拿到Integer就会新创建一个对象
        AtomicStampedReference<String> stampedReference = new
        AtomicStampedReference<>("hello", 1);
        int stamp = stampedReference.getStamp();
        new Thread(() -> {
            //获取当前戳（版本号）
            System.out.println("a1=>" + stampedReference.getStamp() + "--
value=>" + stampedReference.get(new int[1]).hashCode());
            //修改hello，期待对象值为hello，修改后为world，期待的版本号为1，修改后版本号
            +1
            System.out.println(stampedReference.compareAndSet("hello", "world",
            stampedReference.getStamp(), stampedReference.getStamp() + 1));
            try {
                TimeUnit.SECONDS.sleep(2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("a2=>" + stampedReference.getStamp() + "--
value=>" + stampedReference.get(new int[1]).hashCode());
            //再将hello修改回来
            System.out.println(stampedReference.compareAndSet("world", "hello",
            stampedReference.getStamp(), stampedReference.getStamp() + 1));
            //这里可以看到，将world对象修改会hello的hashCode又回到了之前的hello对象，所以
            对象引用并没有改变
            System.out.println("a3=>" + stampedReference.getStamp() + "--
value=>" + stampedReference.get(new int[1]).hashCode());
            }, "A").start();

        new Thread(() -> {
            //获取当前戳（版本号）
            System.out.println("b1=>" + stampedReference.getStamp());
            //修改hello，因为A线程已经修改了版本号，所以B线程会失败，即使需要修改的对象没有变
            System.out.println(stampedReference.compareAndSet("hello", "world",
            stamp, stamp + 1));
            try {

```

```

        TimeUnit.SECONDS.sleep(2);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("b2=>" + stampedReference.getStamp() + "--
value=>" + stampedReference.get(new int[1]).hashCode());
    }, "B").start();
}
}

```

```

"C:\Program Files\Java\jdk1.8.0_131\bin\java.exe" ...

```

```

a1=>1--value=>99162322
true
b1=>2
false
a2=>2--value=>113318802
b2=>2--value=>113318802
true
a3=>3--value=>99162322

```

注意:

【强制】所有整型包装类对象之间值的比较，全部使用 equals 方法比较。

说明：对于 Integer var = ? 在 -128 至 127 之间的赋值，Integer 对象是在 IntegerCache.cache 产生，会复用已有对象，这个区间内的 Integer 值可以直接使用 == 进行判断，但是这个区间之外的所有数据，都会在堆上产生，并不会复用已有对象，这是一个大坑，推荐使用 equals 方法进行判断。

## 各种锁的理解

### 1、公平锁，非公平锁

公平锁：不能插队，非常公平

非公平锁：可以插队，非常不公平（默认）

```

public ReentrantLock() {
    sync = new NonfairSync(); // 非公平锁
}

/**
 * Creates an instance of {@code ReentrantLock} with the
 * given fairness policy.
 *
 * @param fair {@code true} if this lock should use a fair ordering policy true:公平锁, false: 非公平
 */
public ReentrantLock(boolean fair) { sync = fair ? new FairSync() : new NonfairSync(); }

```

## 2、可重入锁

synchronized版

```
public class SynchronizedDemo {
    public static void main(String[] args) {
        Phone phone = new Phone();
        new Thread(() -> {
            phone.sms();
        }, "A").start();

        new Thread(() -> {
            phone.sms();
        }, "B").start();
    }
}

class Phone {
    public synchronized void sms() { //这里的synchronized获取锁是执行call方法也会获得call方法的锁，相当于获得了两个锁，即B线程要想执行必须等A线程执行完后才行
        call();
        System.out.println(Thread.currentThread().getName() + "==>发消息");
    }

    public synchronized void call() {
        System.out.println(Thread.currentThread().getName() + "==>打电话");
    }
}
```

lock版

```
public class LockDemo {
    public static void main(String[] args) {
        Phone2 phone2 = new Phone2();
        new Thread(() -> {
            phone2.sms();
        }, "A").start();

        new Thread(() -> {
            phone2.sms();
        }, "B").start();
    }
}

class Phone2 {
    Lock lock = new ReentrantLock();

    public void sms() {
        lock.lock(); //这里加了一次锁，同样当运行到call方法后会再得到一把锁，形成可重入锁
        try {
            System.out.println(Thread.currentThread().getName() + "==>发消息");
            call(); //这里还会得到锁，锁必须成对出现
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
}
```

```

    }
}

public void call() {
    lock.lock();
    try {
        System.out.println(Thread.currentThread().getName() + "==>打电话");
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
}
}

```

### 3、自旋锁

```

public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        var5 = this.getIntVolatile(var1, var2);
    } while(!this.compareAndSwapInt(var1, var2, var5, var5: var5 + var4));

    return var5;
}

```

典型的自旋锁，一直循环直到成功为止

```

/**
 * 自旋锁
 */
public class SpinLockDemo {
    AtomicReference<Thread> atomicReference = new AtomicReference<>();

    //自定义加锁
    public void myLock() {
        Thread thread = Thread.currentThread();
        System.out.println(thread.getName() + "==> myLock");
        //自旋锁
        while (!atomicReference.compareAndSet(null, thread)) {}
    }

    //自定义解锁
    public void myUnLock() {
        Thread thread = Thread.currentThread();
        System.out.println(thread.getName() + "==> myUnLock");
        atomicReference.compareAndSet(thread, null);
    }
}

```

```

public class SpinLockTest {
    public static void main(String[] args) throws InterruptedException {
        SpinLockDemo spinLockDemo = new SpinLockDemo();

        new Thread(() -> {

```



```

        spinLockDemo.myLock();
        try {
            TimeUnit.SECONDS.sleep(3);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            spinLockDemo.myUnLock();
        }
    }, "A").start();

//        TimeUnit.SECONDS.sleep(1);

    new Thread(() -> {
        spinLockDemo.myLock();
        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            //此时B线程在自旋，只有A线程解锁了B线程才能解锁
            spinLockDemo.myUnLock();
        }
    }, "B").start();
}
}

```

## 4、死锁

### 什么是死锁

多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。由于线程被无限期地阻塞，因此程序不可能正常终止。

**java 死锁产生的四个必要条件：**

- 1、互斥使用，即当资源被一个线程使用(占有)时，别的线程不能使用
- 2、不可抢占，资源请求者不能强制从资源占有者手中夺取资源，资源只能由资源占有者主动释放。
- 3、请求和保持，即当资源请求者在请求其他的资源的同时保持对原有资源的占有。
- 4、循环等待，即存在一个等待队列：P1占有P2的资源，P2占有P3的资源，P3占有P1的资源。这样就形成了一个等待环路。

```

public class DeadLockDemo {
    public static void main(String[] args) {
        Integer lockA = 1;
        Integer lockB = 2;
        MyThread myThread1 = new MyThread(lockA, lockB);
        MyThread myThread2 = new MyThread(lockB, lockA);
        new Thread(myThread1, "T1").start();
        new Thread(myThread2, "T2").start(); //启动后会线程T1抢夺B资源，线程T2抢夺A资源，
        然后发生死锁
    }
}

class MyThread implements Runnable{

```

```

private Integer lockA;
private Integer lockB;

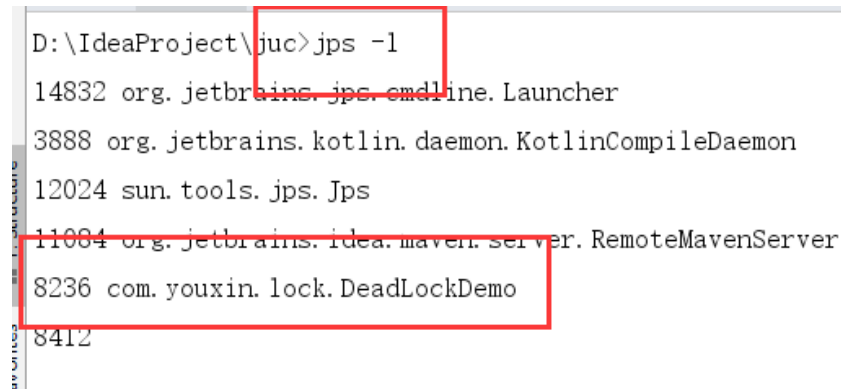
public MyThread(Integer lockA, Integer lockB) {
    this.lockA = lockA;
    this.lockB = lockB;
}

@sneakyThrows
@Override
public void run() {
    synchronized (lockA) {
        System.out.println(Thread.currentThread().getName() + "==>lockA" +
lockA + "==>getLockB" + lockB );
        TimeUnit.SECONDS.sleep(2);
        synchronized (lockB) {
            System.out.println(Thread.currentThread().getName() + "==>lockB"
+ lockB + "==>getLockA" + lockA );
        }
    }
}
}

```

## 排查解决问题 (jps)

### 1、使用 jps -l 定位问题进程号



```

D:\IdeaProject\juc>jps -l
14832 org.jetbrains.jps.cmdline.Launcher
3888 org.jetbrains.kotlin.daemon.KotlinCompileDaemon
12024 sun.tools.jps.Jps
11084 org.jetbrains.idea.maven.server.RemoteMavenServer
8236 com.youxin.lock.DeadLockDemo
8412

```

### 2、jstack 进程号 找到死锁问题

java stack information for the threads listed above:

=====

"T2":

at com.youxin.lock.MyThread.run([DeadLockDemo.java:41](#))

- waiting to lock <0x000000076b3fc190> (a java.lang.Integer) 想要锁住该对象资源

- locked <0x000000076b3fc1a0> (a java.lang.Integer) 已经锁住该资源

at java.lang.Thread.run([Thread.java:748](#))

"T1":

at com.youxin.lock.MyThread.run([DeadLockDemo.java:41](#))

- waiting to lock <0x000000076b3fc1a0> (a java.lang.Integer) 想要锁住该对象资源

- locked <0x000000076b3fc190> (a java.lang.Integer) 已经锁住该资源

at java.lang.Thread.run([Thread.java:748](#))

冲突了

Found 1 deadlock.