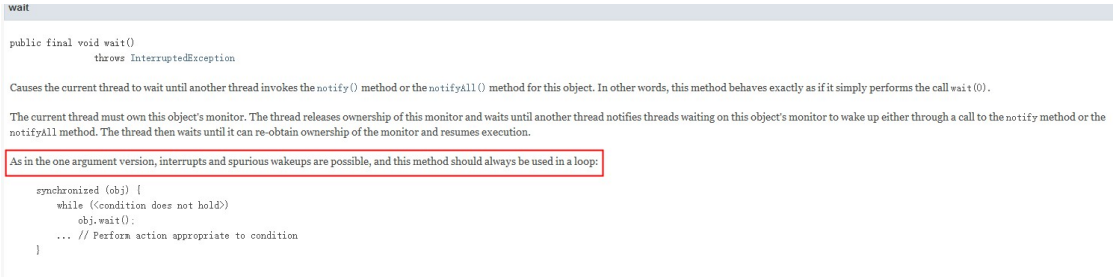
# 一、多线程并发编程 - 企业级套路 + 模板

> 1. 高内聚低耦合前提下，线程->操作->资源类
> 2. 判断/干活/通知
> 3. 多线程交互中（wait，notify，await，signal），必须要防止多线程的虚假唤醒，也即（判断只能使用 while，不能使用 if）



wait

public final void wait()
throws InterruptedException

Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object. In other words, this method behaves exactly as if it simply performs the call wait(0).

The current thread must own this object's monitor. The thread releases ownership of this monitor and waits until another thread notifies threads waiting on this object's monitor to wake up either through a call to the notify method or the notifyAll method. The thread then waits until it can re-obtain ownership of the monitor and resumes execution.

As in the one argument version, interrupts and spurious wakeups are possible, and this method should always be used in a loop:

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

> 4. 标志位，

## 1. 线程 -> 操作 -> 资源类

> 题目：三个售票员同时出售30张票问题

### 1.1 synchronized 版

```
// 资源类 -> 使用同步 synchronized 来处理并发（老版本）
class Ticket{
    private int number = 30;
    public synchronized void saleTicket() {
        if (number > 0) {
            System.out.println(Thread.currentThread().getName() + "卖出第：" +
number-- + "剩下：" + number);
        }
    }
}
```

```
// 线程操作资源类
Ticket t = new Ticket();
new Thread(new Runnable(){
    public void run() {
        for (int i = 0; i < 40; i++) {
            t.saleTicket();
        }
    }
},"A").start();
new Thread(new Runnable(){
    public void run() {
        for (int i = 0; i < 40; i++) {
            t.saleTicket();
        }
    }
},"B").start();
new Thread(new Runnable(){
    public void run() {
        for (int i = 0; i < 40; i++) {
            t.saleTicket();
        }
```

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

### 1.2 Lock版及lambda表达式

```
// 资源类 -> 使用Lock
class Ticket{
    private int number = 30;
    private Lock lock = new ReentrantLock();
    public void saleTicket() {
        lock.lock();
        try{
            if (number > 0) {
            System.out.println(Thread.currentThread().getName() + "卖出第：" +
number-- + "剩下："  + number);
        }
        } finally {
            lock.unlock();
        }
    }
}
```

```
// 线程操作资源类
Ticket t = new Ticket();
new Thread(() -> {for(int i=0;i<40;i++) t.saleTicket();},"A").start();
new Thread(() -> {for(int i=0;i<40;i++) t.saleTicket();},"B").start();
new Thread(() -> {for(int i=0;i<40;i++) t.saleTicket();},"C").start();
```

> 知识点复习：
>
> 1. 线程状态有哪些?
>
> Thread.State枚举中定义：
>
> - NEW 新建状态
> - RUNNABLE 可运行状态
> - BLOCKED 阻塞状态
> - WAITING 等待状态
> - TIMED_WAITING 超时等待
> - TERMINATED 终结状态
>
> 2. 启动一个线程多次start可以吗?
>
> 不能多次start，多次start会报 IllegalThreadStateException 异常
>
> ```
> public synchronized void start() {
>         if (threadStatus != 0)
>             throw new IllegalThreadStateException();
>         group.add(this);
>
>         boolean started = false;
>         try {
>             start0();
>             started = true;
>         } finally {
>             try {
>                 if (!started) {
> ```

```
                    group.threadStartFailed(this);
                }
            } catch (Throwable ignore) {
            }
        }
    }
```

## 2. Lambda表达式及函数式接口

### 2.1 Lambda表达式

> **口诀：拷贝小括号，写死右箭头，落地大括号**

```
// 接口中有且只有一个未实现的方法，方可使用lambda表达式
new Thread(() ->{System.out.println("only one method...");},"A").start();
```

```
// Runnable接口源码
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}
```

### 2.2 @FunctionalInterface（函数式接口） 注解

- 自定义函数式接口

```
@FunctionalInterface
interface Foo{
 // 有参数，有返回值的情况，其他情况更简单
 public int add(int x,int y);
}
```

- 使用

```
Foo foo = (x,y) -> {return x+y;};
System.out.println(foo.add(3,5));
```

### 2.3 default 方法

接口中可以定义多个 default 方法

```
@FunctionalInterface
interface Foo{
 // 有参数，有返回值的情况，其他情况更简单
 public int add(int x,int y);
 // default方法
 default int div(int x,int y) {
     return x/y;
 }
 default int mul(int x,int y){
     return x*y;
 }
}
```

### 2.4 静态方法实现

> 接口中可以定义多个静态方法实现

```
@FunctionalInterface
interface Foo{
 // 有参数，有返回值的情况，其他情况更简单
 public int add(int x,int y);
 // default方法
 default int div(int x,int y) {
     return x/y;
 }
 default int mul(int x,int y){
     return x*y;
 }
 // 静态方法实现
 public static void say(){
     System.out.println("hello static...")
 }
 public static int getDouble(int x){
     return x*2;
 }
}
```

### 3. 多线程交互 - 判断/干活/通知（wait，notify，notifyAll，await，signal，signalAll）及防止虚假唤醒

> 案例：两个线程，操作一个初始值为0的变量，一个线程加一，一个线程减一，彼此交互来10次

#### 3.1 同步 synchronized 版

> - this.wait()
> - this.notifyAll()

```
// 资源类
class Cake{
 private int number = 0;
 public synchronized void increment(){
     // 1.判断（TODO），这里的判断 if 是坑，2个线程看不出来，多几个线程就会出问题
     // if (number != 0){
     while(number != 0){
         this.wait();
     }
     // 2.干活
     number++;
     System.out.println(Thread.currentThread().getName() + ":" +number);
     // 3.通知
     this.notifyAll();
 }
 public synchronized void decrement(){
     // 1.判断（TODO），这里的判断 if 是坑，2个线程看不出来，多几个线程就会出问题
     // if (number == 0){
     while(number == 0){
         this.wait();
     }
     // 2.干活
     number--;
     System.out.println(Thread.currentThread().getName() + ":" +number);
```

```
        // 3.通知
        this.notifyAll();
    }
}
```

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

### 3.2 Lock 版

- condition.await()
- condition.signalAll()

```
// 资源类
class Cake{
 private int number = 0;
 private Lock lock = new ReentrantLock();
 private Condition condition = lock.newCondition();
 public void increment(){
     lock.lock();
     try{
         // 1.判断（TODO），这里的判断 if 是坑，2个线程看不出来，多几个线程就会出问题
         // if (number != 0){
         while(number != 0){
             condition.await();
         }
         // 2.干活
         number++;
         System.out.println(Thread.currentThread().getName() + ":"
+number);
         // 3.通知
         condition.signalAll();
     } finalyy {
         lock.unlock();
     }
 }
 public void decrement(){
     lock.lock();
     try{
         // 1.判断（TODO），这里的判断 if 是坑，2个线程看不出来，多几个线程就会出问题
         // if (number == 0){
         while(number == 0){
             condition.await();
         }
```

```
            // 2.干活
            number--;
            System.out.println(Thread.currentThread().getName() + ":"
+number);
            // 3.通知
            condition.signalAll();
        }finally {
            lock.unlock();
        }
    }
}
```

知识点：

1. 线程sleep()和wait()有何区别？

   > sleep()不会释放锁，wait()会释放锁

2. 有了synchronized 为何还会出一个Lock？

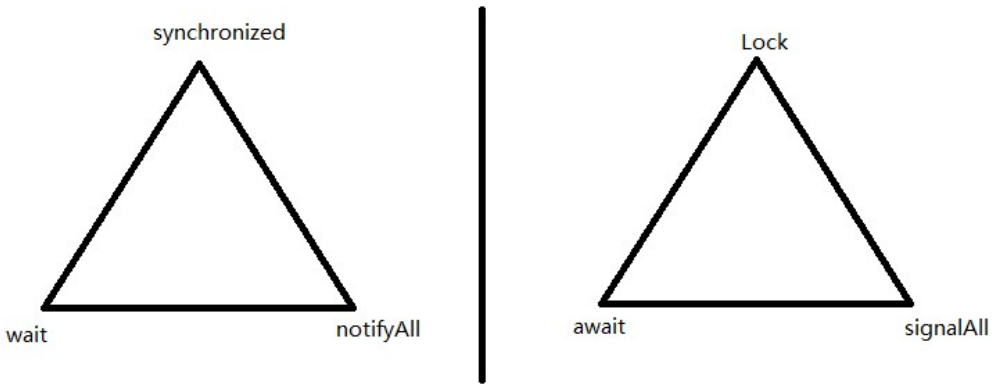
   > Lock能够更细粒度的操作控制线程

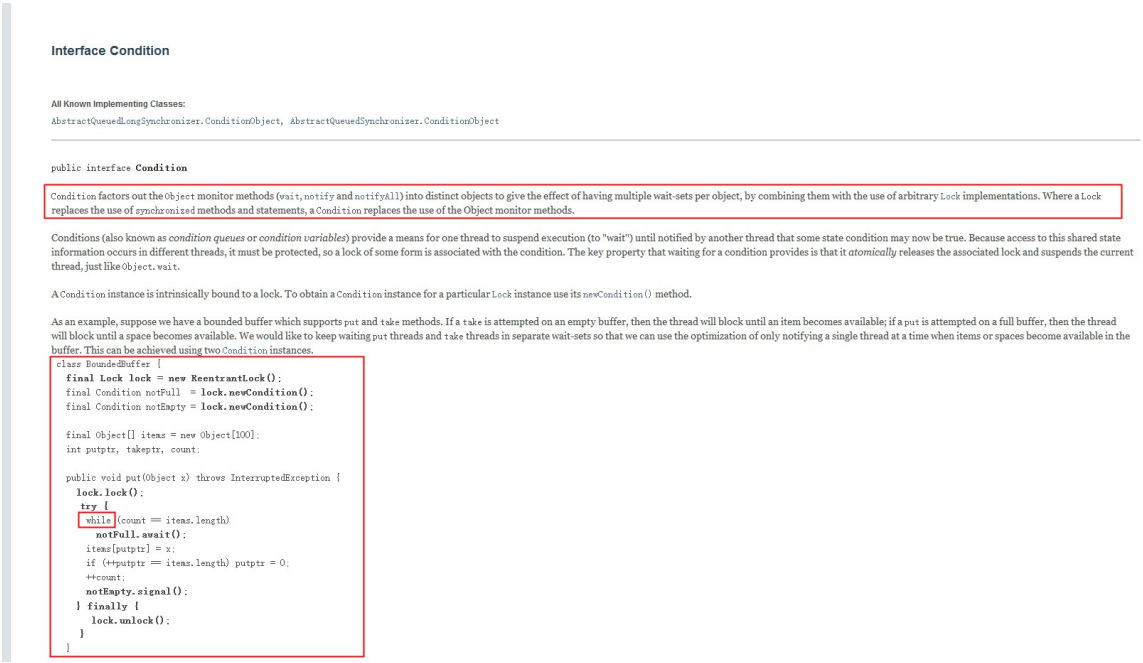
3. wait()和notifyAll是属于Thread类的吗？

   > 不是，是Object的方法

4. 比较记忆法：

   - synchronized -> Lock
   - wait() -> await()
   - notify() -> signal()
   - notifyAll() -> signalAll()





Interface Condition

All Known Implementing Classes:

AbstractQueuedLongSynchronizer.ConditionObject, AbstractQueuedSynchronizer.ConditionObject

public interface Condition

Condition factors out the Object monitor methods (wait, notify and notifyAll) into distinct objects to give the effect of having multiple wait-sets per object, by combining them with the use of arbitrary Lock implementations. Where a Lock replaces the use of synchronized methods and statements, a Condition replaces the use of the Object monitor methods.

Conditions (also known as *condition queues* or *condition variables*) provide a means for one thread to suspend execution (to "wait") until notified by another thread that some state condition may now be true. Because access to this shared state information occurs in different threads, it must be protected, so a lock of some form is associated with the condition. The key property that waiting for a condition provides is that it *atomically* releases the associated lock and suspends the current thread, just like Object.wait.

A Condition instance is intrinsically bound to a lock. To obtain a Condition instance for a particular Lock instance use its newCondition() method.

As an example, suppose we have a bounded buffer which supports put and take methods. If a take is attempted on an empty buffer, then the thread will block until an item becomes available; if a put is attempted on a full buffer, then the thread will block until a space becomes available. We would like to keep waiting put threads and take threads in separate wait-sets so that we can use the optimization of only notifying a single thread at a time when items or spaces become available in the buffer. This can be achieved using two Condition instances.

```
class BoundedBuffer {
  final Lock lock = new ReentrantLock();
  final Condition notFull  = lock.newCondition();
  final Condition notEmpty = lock.newCondition();

  final Object[] items = new Object[100];
  int putptr, takeptr, count;

  public void put(Object x) throws InterruptedException {
    lock.lock();
    try {
      while (count == items.length)
        notFull.await();
      items[putptr] = x;
      if (++putptr == items.length) putptr = 0;
      ++count;
      notEmpty.signal();
    } finally {
      lock.unlock();
    }
  }
```

## 4. Lock精确通知顺序访问

> **synchronized 不能做到精确顺序通知，所以Lock出现**

> 案例：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：
>
> A打印5次，B打印10次，C打印15次，
>
> 接着顺序来10轮

```java
// 资源类
class Resource{
    // 标志位
    private int number = 1;
    // Lock
    private Lock lock = new ReentrantLock();
    // Condition
    private Condition condition1 = lock.newCondition();
    private Condition condition2 = lock.newCondition();
    private Condition condition3 = lock.newCondition();
    public void prints(int count) throws InterruptedException {
        lock.lock();
        try {
            switch(count){
                case 5:
                    // 1.判断
                    while(number != 1){
                        condition1.await();
                    }
                    // 2.干活
                    for(int i=1;i<= count;i++){
                        System.out.println(Thread.currentThread().getName());
                    }
                    // 3.通知
                    // 3.1 修改标志位
                    number = 2;
                    // 3.2 通知线程2
                    condition2.signal();
```

```
                    break;
                case 10:
                    // 1.判断
                    while(number != 2){
                        condition2.await();
                    }
                    // 2.干活
                    for(int i=1;i<= count;i++){
                        System.out.println(Thread.currentThread().getName());
                    }
                    // 3.通知
                    // 3.1 修改标志位
                    number = 3;
                    // 3.2 通知线程3
                    condition3.signal();
                    break;
                case 15:
                    // 1.判断
                    while(number != 3){
                        condition3.await();
                    }
                    // 2.干活
                    for(int i=1;i<= count;i++){
                        System.out.println(Thread.currentThread().getName());
                    }
                    // 3.通知
                    // 3.1 修改标志位
                    number = 1;
                    // 3.2 通知线程1
                    condition1.signal();
                    break;
            }
        }finally {
            lock.unlock();
        }
    }
}
```

```
Resource resource = new Resource();
new Thread(() -> {
    for (int i = 1; i <=10 ; i++) {
        try {
            resource.prints(5);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
},"A").start();
new Thread(() -> {
    for (int i = 1; i <=10 ; i++) {
        try {
            resource.prints(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
},"B").start();
```

```
new Thread(() -> {
    for (int i = 1; i <=10 ; i++) {
        try {
            resource.prints(15);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
},"C").start();
```

## 二、8锁

> 多线程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4s，请问先打印邮件还是短信?
> 3. 新增一个普通方法hello，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，两部手机，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，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，两部手机，请问先打印邮件还是短信?

```
// 资源类
class Phone{

 //public static synchronized void sendEmail() throws Exception {
 public synchronized void sendEmail() throws Exception {
     //TimeUnit.SECONDS.sleep(4);
     System.out.println("----sendEmail");
 }

 //public static synchronized void sendSms() throws Exception {
 public synchronized void sendSms() throws Exception {
     System.out.println("----sendSms");
 }

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

}
```

```
Phone phone1 = new Phone();
Phone phone2 = new Phone();
new Thread(() -> {
 try{
     phone1.sendEmail();
 }catch(Exception e){
     e.printStackTrace();
 }
},"A").start();

Thread.sleep(100);
```

```
new Thread(() -> {
 try{
     phone1.sendSms();
      //phone1.hello();
     //phone2.sendSms();
 }catch(Exception e){
     e.printStackTrace();
 }
},"B").start();
```

> 8锁解释：synchronized 实现同步的基础，Java中的每一个对象都可以作为锁。
>
> 1. 锁的是当前对象this，synchronized 方法 只能有一个线程可以访问，先打印邮件，后打印短信
> 2. 锁的是当前对象this，synchronized 方法 只能有一个线程可以访问，暂停4s后先打印邮件，后再打印短信
> 3. 普通方法，不会存在并发争抢问题，先打印hello，后打印邮件
> 4. phone1锁的是当前对象phone1，phone2锁的是phone2当前对象，没有使用同一把锁，先打印短信，后打印邮件
> 5. 静态同步方法，锁的是当前Class对象，先打印邮件，后打印短信
> 6. 静态同步方法，锁的是当前Class对象，先打印邮件，后打印短信
> 7. 普通同步方法锁的是当前对象this，静态同步方法锁的当前Class对象，锁不同，先打印短信，后打印邮件
> 8. 普通同步方法锁的是当前对象this，静态同步方法锁的当前Class对象，锁不同，先打印短信，后打印邮件

## 三、list不安全

> 1. **故障现象**
>
> > `java.util.ConcurrentModificationException`
>
> 2. **导致原因**
>
> > 多线程情况下，并发修改，ArrayList 线程不安全，导致并发修改异常
>
> 3. **解决方案**
>
> - Vector
>
> > 原理：synchronized
>
> - Collections.synchronizedList(new ArrayList<>())
> - CopyOnWriteArrayList
>
> > 原理：写时复制，读写分离思想
> >
> > `CopyOnWrite`容器即写时复制的容器。往一个容器添加元素的时候，不直接往当前`Object[]`添加，而是先将当前容器`Object[]`进行`copy`，复制出一个新的容器`Object[] newElements`， 然后新的容器`Object[] newElements`里添加元素，添加完元素之后，再将原容器的引用指向新的容器`setArray(newElements)`；这样做的好处是可以对`CopyOnWrite`容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。所以`CopyOnWrite`容器也是一种读写分离的思想，读和写不同的容器

```
public boolean add(E e) {
    final ReentrantLock lock = this.lock;  // 加锁
    lock.lock();
    try {
        Object[] elements = getArray(); // 得到原数组
        int len = elements.length;      // 原数组大小
        Object[] newElements = Arrays.copyOf(elements, len +
1);//扩容+1
        newElements[len] = e;  // 放入元素
        setArray(newElements);      // 重新设置数组
        return true;
    } finally {
        lock.unlock();
    }
}
```

4. **经典面试题**

- 当我们new ArrayList 的时候，底层new了一个什么东西?

  > new 了一个数组

- 数组是什么类型的?

  > Object 类型的数组

- 默认初始大小是多少?

  > 默认大小是 10 的Object类型的数组

- 使用默认的，我需要存25个元素怎么办?

  > 不够存时，使用Arrays.copyOf()方法进行扩容

- 底层使用什么进行扩容?

  > Arrays.copyOf()方法

- 如何扩容?

  > 扩容为原值的一半，10 -> 15 ; 15 -> 22

- ArrayList 是否线程安全?

  > ArrayList是线程不安全的

- ArrayList 线程不安全报什么异常?

  > java.util.ConcurrentModificationException

- 如何解决线程不安全?

  > - Vector
  > - Collections.synchronizedList(new ArrayList<>())
  > - CopyOnWriteArrayList

- 写一个线程不安全的例子?

```
List<String> list = new ArrayList<>();
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();
}
```

## 四、set不安全

1. **故障现象**

   ```
   java.util.ConcurrentModificationException
   ```

2. **导致原因**

   并发修改

3. **解决方案**

   - Collections.synchronizedSet(new HashSet<>())
   - CopyOnWriteArraySet

4. **经典面试题**

   - new HashSet时底层是new了一个什么东西?

     new 了一个 HashMap

   - 初始大小是多少?

     就是HashMap的默认大小 16

   - 默认到多少开始扩容?

     HashMap的默认负载因子是0.75，默认初始大小 16，所以到 12 的时候会开始扩容

   - HashMap 需要 k，v存值，HashSet的add只有一个元素，如何解释?

     HashSet的add方法就是调用了HashMap的put方法，k就是add的元素，v是一个Object类型的常量，固定写死

   - HashSet是线程安全的吗?

     HashSet是线程不安全的

   - HashSet线程不安全报何异常?

     java.util.ConcurrentModificationException

   - 如何解决HashSet线程不安全问题?

     - Collections.synchronizedSet(new HashSet<>())
     - CopyOnWriteArraySet

   - 写一个线程不安全例子?

```
Set<String> set = new HashSet<>();
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();
}
```

## 五、map不安全

1. **故障现象**

   ```
   java.util.ConcurrentModificationException
   ```

2. **导致原因**

   并发修改

3. **解决方案**

   - Collections.synchronizedMap(new HashMap<>())
   - ConcurrentHashMap

4. **经典面试题**

   - HashMap底层是什么?

     Node的节点数组 + Node节点单向链表 + 红黑树

   - HashMap 的默认初始大小多少?

     默认大小是 16

   - 默认从多少开始扩容?

     默认负载因子是 0.75，默认初始大小是 16，因此到 12 之后开始扩容

   - 如何扩容?

     原值的一倍，即 16 -> 32

   - 如何优化HashMap?

     如果能够知道数据的范围，设置默认值，这样可以节约扩容带来的时间花费

   - HashMap是线程安全的吗?

     HashMap是线程不安全的

   - 如何解决线程不安全的问题?

     - Collections.synchronizedMap(new HashMap<>())
     - ConcurrentHashMap

   - 写一个线程不安全的例子?

```
Map<String,String> map = new HashMap<>();
for (int i = 1; i <= 30; i++) {
 new Thread(() -> {
map.put(UUID.randomUUID().toString().substring(0,8),Thread.curre
ntThread().getName());
     System.out.println(map);
 },String.valueOf(i)).start();
}
```

- HashMap是有序的吗?

  > 无序的

- 有序的Map有吗?

  > **TreeMap和LinkedHashMap是有序的（TreeMap默认 Key 升序，LinkedHashMap则记录了插入顺序）**

- TreeMap和LinkedHashMap是如何实现有序的及二者的区别?

  > TreeMap默认是升序的，如果我们需要改变排序方式，则需要使用比较器：Comparator。
  >
  > LinkedHashMap保存了记录的插入顺序

- 有没有更好的方式实现有序?

  >

## 六、多线程的实现方式

### 1. 3种实现方式

- Thread 类

```
class MyThread extends Thread {
 @Override
 public void run() {

 }
}
```

- Runnable接口

```
class MyThread implements Runnable {
    @Override
public void run() {

}
}
```

- Callable 接口

```
class MyThread implements Callable<Integer> {

@Override
public Integer call() throws Exception {
    return null;
}
}
```

> Runnable与Callable 的区别：
>
> 1. 是否有返回值，Callable接口有返回值
> 2. 是否有异常，Callable接口有异常
> 3. 落地方法不一样，Runnable接口为run方法，Callable接口为call方法

**2. Callable接口使用细节，使用 FutureTask**

```
class MyThread implements Callable<Integer> {

    @Override
    public Integer call() throws Exception {
        System.out.println("callable...");
        return 1024;
    }
}
```

```
FutureTask futureTask = new FutureTask(new MyThread());
new Thread(futureTask,"A").start();
new Thread(futureTask,"B").start(); // 多次执行任务，callable...只会打印一次，结果会复用
// 马上获取的话，会一直等待，阻塞主线程
System.out.println(futureTask.get());
// 判断任务是否完成
while (!futureTask.isDone()) {
    System.out.println("任务还在运行中...");
}
// 获取返回值，一般情况放在最后
System.out.println(futureTask.get());
```

## 七、JUC多线程辅助工具类

**1. CountDownLatch（倒计数）-> 自习室楼管关门**

```
CountDownLatch countDownLatch = new CountDownLatch(6);

for (int i = 1; i <= 6; i++) {
    new Thread(() -> {
        System.out.println(Thread.currentThread().getName() + "离开了");
        // 离开一个减少一个
        countDownLatch.countDown();
    },String.valueOf(i)).start();
}
// 阻塞等待计数减到0
countDownLatch.await();
System.out.println("主线程最后撤离....");
```

> 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），
>
> 当计数器的值变为0时，因await方法阻塞的线程会被唤醒，继续执行。

**2. CyclicBarrier （顺计数）->七龙珠召唤神龙**

```
CyclicBarrier cyclicBarrier = new CyclicBarrier(7,() -> {System.out.println("召唤神龙");});

for (int i = 1; i <=7 ; i++) {
    final int temp = i;
    new Thread(() -> {
        System.out.println(Thread.currentThread().getName()+"集齐第"+temp+"颗龙珠");
        try {
            cyclicBarrier.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (BrokenBarrierException e) {
            e.printStackTrace();
        }
    },String.valueOf(i)).start();
}
```

> 原理：CyclicBarrier 的await方法会阻塞构造方法中的线程，直到子线程数执行到设置的数时才唤醒构造方法中的线程，继续执行。

**3. Semaphore （信号灯）->抢车位**

```
Semaphore semaphore = new Semaphore(3); //模拟资源类，有3个空车位
for (int i = 1; i <=6 ; i++) {
    new Thread(() -> {
        try {
            // 占位
            semaphore.acquire();
            System.out.println(Thread.currentThread().getName()+"抢占了车位");
            try {
                TimeUnit.SECONDS.sleep(3);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
```

```
            System.out.println(Thread.currentThread().getName()+"离开车位");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            // 释放
            semaphore.release();
        }

    },String.valueOf(i)).start();
}
```

> 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，它要么通过成功获取信号量（信号量减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加1，然后唤醒等待的线程。
>
> **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。**

## 八、ReadWriteLock 读写锁

> Lock为了进一步的细化，读操作共享，写操作独占
>
> 多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，读取共享资源应该是可以同时进行的
>
> 但是，如果有一个线程想去写共享资源类，就不应该再有其他线程可以对该资源进行读或写
>
> 总结：
>
> - 读-读 能共存
> - 读-写不能共存
> - 写-写不能共存

```
// 资源类
class MyCache{
    private volatile Map<String,Object> map = new HashMap<>();
    // 读写锁 ReentrantReadWriteLock
    private ReadWriteLock rwLock = new ReentrantReadWriteLock();

    // 写操作
    public void put(String k, Object v) {
        // 获取写锁
        rwLock.writeLock().lock();
        try{
            System.out.println(Thread.currentThread().getName()+"写入数据"+k);
            try {
                TimeUnit.MILLISECONDS.sleep(300);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            map.put(k,v);
            System.out.println(Thread.currentThread().getName()+"写入完成");
        } finally {
            rwLock.writeLock().unlock();
```

```
        }
    }
    // 读操作
    public void get(String k) {
        // 获取读锁
        rwLock.readLock().lock();
        try{
            System.out.println(Thread.currentThread().getName()+"开始读取");
            try {
                TimeUnit.MILLISECONDS.sleep(300);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            Object o = map.get(k);
            System.out.println(Thread.currentThread().getName()+"读取完成"+o);
        }finally {
            rwLock.readLock().unlock();
        }
    }
}
```

```
MyCache myCache = new MyCache();
for (int i = 1; i <=5 ; i++) {
    final int temp = i;
    new Thread(() -> {
        myCache.put(String.valueOf(temp),UUID.randomUUID().toString());
    },String.valueOf(i)).start();
}

Thread.sleep(300);

for (int i = 1; i <=5 ; i++) {
    final int temp = i;
    new Thread(() -> {
        myCache.get(String.valueOf(temp));
    },String.valueOf(i)).start();
}
```

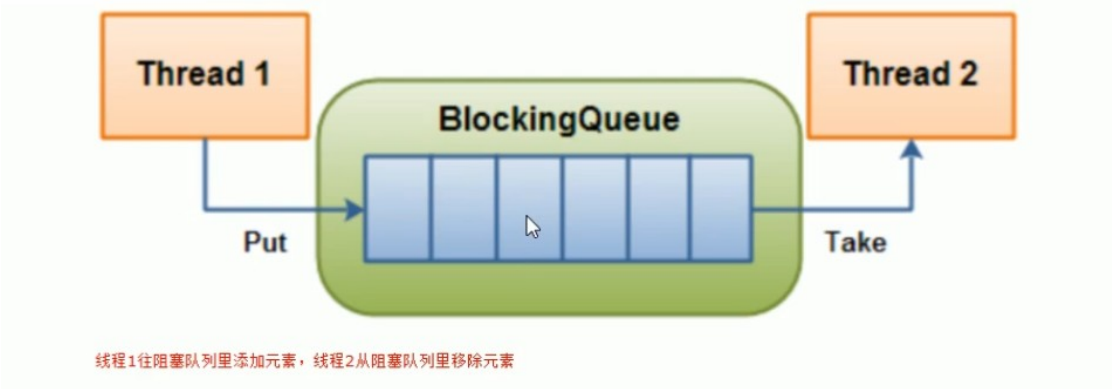
## 九、BlockingQueue 阻塞队列

### 1. 栈与队列

> 栈：　FILO 先进后出
>
> 队列：FIFO 先进先出

### 2. 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，结构如下：




**当队列是空时，从队列中获取元素的操作将会被阻塞**

**当队列是满时，往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，直到其他线程往空队列中插入新的元素

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。

### 3. 阻塞队列的用处

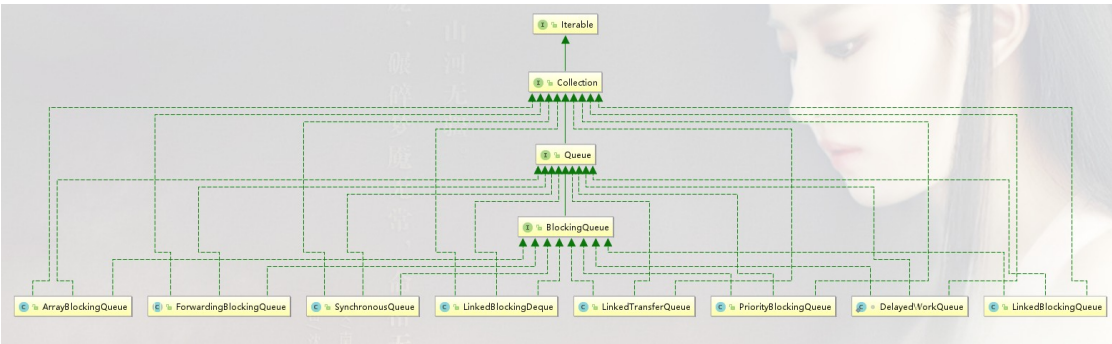
在多线程领域：所谓阻塞，在某些情况下会挂起线程（即阻塞），一旦条件满足，被挂起的线程又会自动被唤醒

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去自己控制这些细节，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。

### 4. 架构梳理，种类分析




- **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（但大小默认值为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

### 5. BlockingQueue核心方法

BlockingQueue核心方法

| 方法类型 | 抛出异常 | 特殊值 | 阻塞 | 超时 |
| --- | --- | --- | --- | --- |
| 插入 | add(e) | offer(e) | put(e) | offer(e,time,unit) |
| 移除 | remove() | poll() | take() | poll(time,unit) |
| 检查 | element() | peek() | 不可用 | 不可用 |

| | |
| --- | --- |
| 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<br>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<br>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<br>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 |

### 6. 代码

```
// 定义 队列大小为 3
BlockingQueue<String> blockingQueue = new ArrayBlockingQueue<>(3);
// 1.抛出异常
System.out.println(blockingQueue.add("a"));
System.out.println(blockingQueue.add("b"));
System.out.println(blockingQueue.add("c"));
// 加入第4个时，java.lang.IllegalStateException: Queue full
//System.out.println(blockingQueue.add("x"));

System.out.println(blockingQueue.remove());
System.out.println(blockingQueue.remove());
System.out.println(blockingQueue.remove());
// 取出第4个时，java.util.NoSuchElementException
//System.out.println(blockingQueue.remove());

System.out.println(blockingQueue.element());
System.out.println(blockingQueue.element());
System.out.println(blockingQueue.element());
// 查看队首元素，取出第4个时，java.util.NoSuchElementException
//System.out.println(blockingQueue.element());

// 2.特殊值 true false
System.out.println(blockingQueue.offer("a"));
System.out.println(blockingQueue.offer("b"));
System.out.println(blockingQueue.offer("c"));
// false
System.out.println(blockingQueue.offer("x"));

System.out.println(blockingQueue.poll());
System.out.println(blockingQueue.poll());
System.out.println(blockingQueue.poll());
// null
System.out.println(blockingQueue.poll());

System.out.println(blockingQueue.peek());
System.out.println(blockingQueue.peek());
System.out.println(blockingQueue.peek());
// 检查队首的元素
System.out.println(blockingQueue.peek());

// 3.阻塞
blockingQueue.put("a");
```

```
blockingQueue.put("b");
blockingQueue.put("c");
// 被阻塞
//blockingQueue.put("x");
System.out.println(blockingQueue.take());
System.out.println(blockingQueue.take());
System.out.println(blockingQueue.take());
// 被阻塞
//System.out.println(blockingQueue.take());

// 4.超时
System.out.println(blockingQueue.offer("a"));
System.out.println(blockingQueue.offer("b"));
System.out.println(blockingQueue.offer("c"));
// 3秒后输出 false
System.out.println(blockingQueue.offer("x",3L, TimeUnit.SECONDS));
```

## 十、ThreadPool 线程池

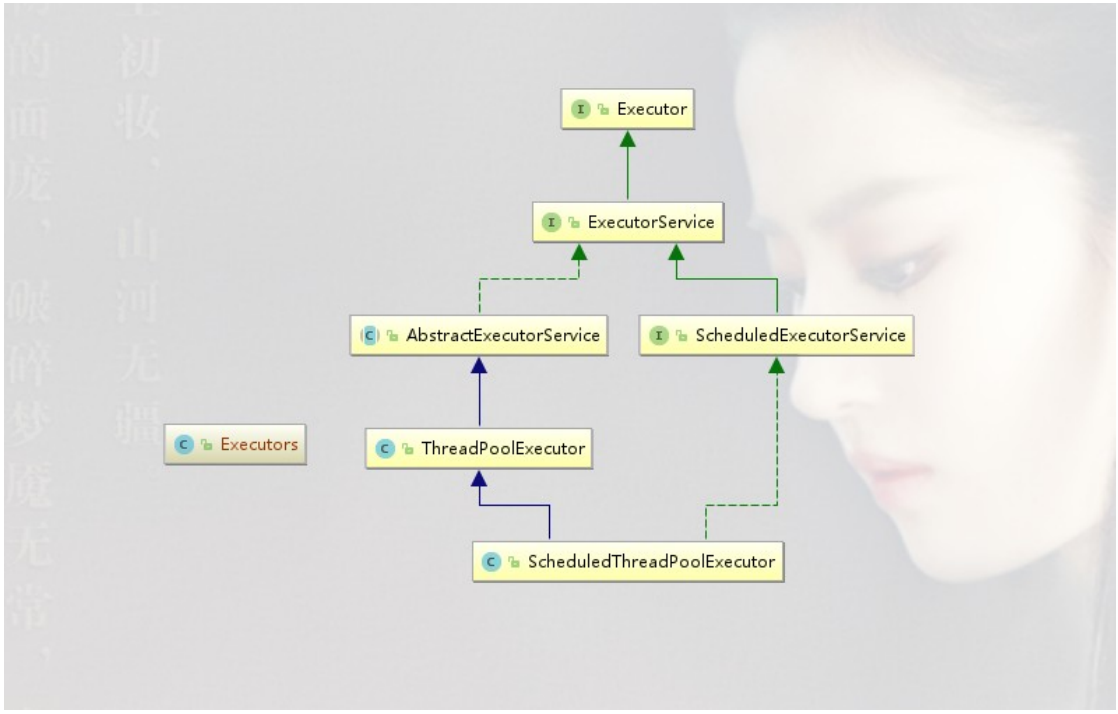
### 1. 为什么用线程池

> 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**，然后再线程创建后启动这些任务，**如果线程数量过了最大出数量，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
>
> 主要特点：**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的分配，调优和监控

### 2. 线程池如何使用

#### 2.1 架构说明

> Java 中的线程池是提供 Executor 框架实现的，该框架中用到了 Executor，Executors，ExecutorService，ThreadPoolExecutor 这几个类




### 2.2 编码实现

2.2.1 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**

```
ExecutorService threadPool = Executors.newFixedThreadPool(5);
try {
 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();
}
```

2.2.2 Executors.newSingleThreadExecutor()

> **一个任务一个任务的执行，一池一线程**

```
ExecutorService threadPool = Executors.newSingleThreadExecutor();
try {
 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();
}
```

2.2.3 Executors.newCachedThreadPool()

> **执行很多短期异步任务，线程池根据需要创建新线程，但在先前构建的线程可用时将重用它们，可扩容，遇强则强**

```
ExecutorService threadPool = Executors.newCachedThreadPool();
try {
 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();
}
```

### 2.3 ThreadPoolExecutor底层原理

> 以上三种获取线程池的方法源码

```
public static ExecutorService newFixedThreadPool(int nThreads) {
 return new ThreadPoolExecutor(nThreads, nThreads,
                               0L, TimeUnit.MILLISECONDS,
                               new LinkedBlockingQueue<Runnable>());
}
```

```
public static ExecutorService newSingleThreadExecutor() {
 return new FinalizableDelegatedExecutorService
     (new ThreadPoolExecutor(1, 1,
                             0L, TimeUnit.MILLISECONDS,
                             new LinkedBlockingQueue<Runnable>()));
}
```

```
public static ExecutorService newCachedThreadPool() {
 return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                               60L, TimeUnit.SECONDS,
                               new SynchronousQueue<Runnable>());
}
```

> 分析可知：底层都是使用了 ThreadPoolExecutor

## 3. 线程池几个重要参数

- corePoolSize：线程池中的常驻核心线程数
- maximumPoolSize：线程池中能够容纳同时执行的最大线程数，此值必须大于等于1
- keepAliveTime：多余的空闲线程的存活时间，当前池中线程数量超过corePoolSize时，当空闲时间达到keepAliveTime时，多余线程会被销毁直到只剩下corePoolSize个线程为止
- unit：keepAliveTime的单位
- workQueue：任务队列，被提交但尚未执行的任务
- threadFactory：表示生成线程池中工作线程的线程工厂，用于创建线程，**一般默认即可**
- handler：**拒绝策略，表示当队列满了，并且工作线程大于等于线程池的最大线程数（maximumPoolSize）时如何来拒绝请求执行的runnable的策略**

---

```
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.corePoolSize = corePoolSize;
    this.maximumPoolSize = maximumPoolSize;
    this.workQueue = workQueue;
    this.keepAliveTime = unit.toNanos(keepAliveTime);
    this.threadFactory = threadFactory;
    this.handler = handler;
}
```

**4. 线程池底层工作原理**

- 使用corePoolSize常驻线程处理任务
- 任务大于corePoolSize时，多余任务进入BlockingQueue阻塞队列
- BlockingQueue队列也满了，进行扩容到 maximumPoolSize进行处理
- 如果线程数已经扩容到maximumPoolSize，且BlockingQueue也满了，启用拒绝策略
- 随着时间的推移，任务慢慢少了，这时多余的线程将根据keepAliveTime和unit参数进行线程销毁直到corePoolSize个线程

**5. 线程池用哪个？生产中如何设置合理参数?**

**5.1 线程池的拒绝策略**

5.1.1 是什么

> **等待队列已经排满了，再也塞不下新任务了，同时，线程池中的max线程也达到了，无法继续为新任务服务。**
>
> 这个时候我们就需要拒绝策略机制合理的处理这个问题

5.1.2 JDK内置的拒绝策略

- **AbortPolicy（默认）**：直接抛出java.util.concurrent.RejectedExecutionException异常阻止系统正常运行
- **CallerRunsPolicy**：“调用者运行”一种调节机制，该策略既不会抛弃任务，也不会抛出异常，而是将某些任务退回到调用者，从而降低新任务的流量
- **DiscardOldestPolicy**：抛弃队列中等待最久的任务，然后把当前任务加入队列中尝试再次提交当前任务
- **DiscardPolicy**：该策略默默地丢弃无法处理的任务，不予任何处理也不抛出异常。如果允许任务丢失，这是最好的一种策略

5.1.3 以上内置拒绝策略均实现了RejectedExecutionHandler接口

**5.2 在工作中单一的/固定的/可变的三种常见线程池的方法哪个用的多?**

> **答案是一个都不用，我们工作中只能使用自定义的**

> 为什么不用?


```
// newFixedThreadPool 这里的 LinkedBlockingQueue 默认大小是
Integer.MAX_VALUE
public static ExecutorService newFixedThreadPool(int nThreads) {
 return new ThreadPoolExecutor(nThreads, nThreads,
                               0L, TimeUnit.MILLISECONDS,
                               new LinkedBlockingQueue<Runnable>());//这
里的问题
}
// newSingleThreadExecutor 这里的 LinkedBlockingQueue 默认大小是
Integer.MAX_VALUE
public static ExecutorService newSingleThreadExecutor() {
 return new FinalizableDelegatedExecutorService
     (new ThreadPoolExecutor(1, 1,
                             0L, TimeUnit.MILLISECONDS,
                             new LinkedBlockingQueue<Runnable>()));//这里
的问题
}
// Integer.MAX_VALUE
public LinkedBlockingQueue() {
 this(Integer.MAX_VALUE);
}
/**********************************************************************
**********/
// newCachedThreadPool 的 maximumPoolSize大小是 Integer.MAX_VALUE
public static ExecutorService newCachedThreadPool() {
 return new ThreadPoolExecutor(0, Integer.MAX_VALUE, // 这里的问题
                               60L, TimeUnit.SECONDS,
                               new SynchronousQueue<Runnable>());
}
// ScheduledThreadPoolExecutor 的 maximumPoolSize大小是 Integer.MAX_VALUE
public ScheduledThreadPoolExecutor(int corePoolSize) {
 super(corePoolSize, Integer.MAX_VALUE, // 这里的问题
       0, NANOSECONDS,
       new DelayedWorkQueue());
}
```

### 5.3 在工作中如何使用线程池，是否自定义过线程池

> 自定义线程池:

```
// JDK内置四大拒绝策略
// ThreadPoolExecutor.AbortPolicy()
// ThreadPoolExecutor.CallerRunsPolicy()
```

```
// ThreadPoolExecutor.DiscardPolicy()
// ThreadPoolExecutor.DiscardOldestPolicy()
ExecutorService threadPool = new ThreadPoolExecutor(
 2,
 5,
 2L,
 TimeUnit.SECONDS,
 new LinkedBlockingDeque<>(3),
 Executors.defaultThreadFactory(),
 new ThreadPoolExecutor.DiscardPolicy()
);
try {
 for (int i = 1; i <=9 ; i++) {
     threadPool.execute(() -> {
         System.out.println(Thread.currentThread().getName()+"办理业务");
     });
 }
} catch (Exception e) {
 e.printStackTrace();
} finally {
 threadPool.shutdown();
}
```

面试题：

**工作中如何设置线程池参数?**

- CPU密集型

  CPU密集的意思是该任务需要大量的运算，而没有阻塞，CPU一直全速运行。

  CPU密集任务只有在真正的多核CPU上才能得到加速（通过多线程）

  第一步：**Runtime.getRuntime().availableProcessors() 获取CPU核数**

  第二步：**一般公式：CPU核数 + 1 个线程的线程池**

  ```
  int maximumPoolSize = Runtime.getRuntime().availableProcessors()
  + 1;
  ExecutorService threadPool = new ThreadPoolExecutor(
   2,
   maximumPoolSize,
   2L,
   TimeUnit.SECONDS,
   new LinkedBlockingDeque<>(3),
   Executors.defaultThreadFactory(),
   new ThreadPoolExecutor.AbortPolicy()
  );
  ```

- IO密集型

  IO密集型，即该任务需要大量的IO，即大量的阻塞

  在单线程上运行IO密集型的任务会导致浪费大量的CPU运算能力浪费在等待。

  所以在IO密集型任务中使用多线程可以大大的加速程序运行，即使在单核CPU上，这种加速主要加速利用了被浪费掉的阻塞数据。

  IO密集型时，大部分线程阻塞，故需要多配置线程数：

1. 方式一：由于IO密集型任务线程并不是一直在执行任务，则应配置尽可能多的线程，如：**CPU核数*2**
2. 方式二：参考公式：CPU核数 / 1-阻塞系数

阻塞系数在0.8~0.9之间

比如8核CPU：8/1-0.9 = 80个线程数

```
int maximumPoolSize = Runtime.getRuntime().availableProcessors()
/(1-0.9);
ExecutorService threadPool = new ThreadPoolExecutor(
    2,
    maximumPoolSize,
    2L,
    TimeUnit.SECONDS,
    new LinkedBlockingDeque<>(3),
    Executors.defaultThreadFactory(),
    new ThreadPoolExecutor.AbortPolicy()
);
```

## 十一、Java8链式编程+流式计算

### 1. 链式编程

```
package com.yidao.promote.http.wangqin.stream;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.experimental.Accessors;

/**
 * @Classname Book
 * @Description TODO
 * @Date 2019/12/1 20:38
 * @Created by 伊人
*/
@Data
@AllArgsConstructor
@NoArgsConstructor
@Accessors(chain = true)
public class Book {
 private Integer id;
 private String name;
 private Integer age;
}
```

```
Book book = new Book();
book.setId(1).setName("shadow").setAge(28);
```

### 2.函数式接口

题目：请按照给出的数据，找出同时满足以下条件的用户，也即以下条件全　　部满足：

偶数ID且年龄大于24且用户名称转为大写且用户名字字母倒排序，只　　输出一个用户名字

```
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);
```

**Java内置的四大函数式接口：**

| 函数式接口 | 参数类型 | 返回类型 | 用途 |
| --- | --- | --- | --- |
| Consumer<T><br>消费型接口 | T | void | 对类型为T的对象应用操作，包含方法：<br>void accept(T t) |
| Supplier<T><br>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><br>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><br>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); |

- **Function<T, R>**

  > 函数型接口，输入一个T类型参数返回一个R类型值

```
// 源码
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

```
// 使用
// lambda表达式
Function<String,Integer> function1 = (String s) -> {return s.length();};
Function<String,Integer> function2 = (s) -> {return s.length();};
Function<String,Integer> function3 = s -> {return s.length();};
Function<String,Integer> function4 = s -> s.length();
System.out.println(function1.apply("abc"));
System.out.println(function2.apply("1"));
System.out.println(function3.apply("22"));
System.out.println(function4.apply("hello"));
```

- **Predicate**

  > 断定型接口，输入一个T类型参数，返回一个boolean值

```
@FunctionalInterface
public interface Predicate<T> {
   boolean test(T t);
}
```

```
// 使用
Predicate<String> predicate1 = (String s) -> {return s.isEmpty();};
Predicate<String> predicate2 = (s) -> {return s.isEmpty();};
Predicate<String> predicate3 = s -> {return s.contains("a");};
Predicate<String> predicate4 = s ->  s.contains("a");
System.out.println(predicate1.test("a"));
System.out.println(predicate2.test(""));
System.out.println(predicate3.test("lambda"));
System.out.println(predicate4.test("www"));
```

- **Consumer**

  > 消费型接口，输入一个T类型参数，无返回

```
@FunctionalInterface
public interface Consumer<T> {
   void accept(T t);
}
```

```
// 使用
Consumer<String> consumer1 = (String s) -> {System.out.println(s);};
Consumer<String> consumer2 =  (s) -> {System.out.println(s);};
Consumer<String> consumer3 = s -> {System.out.println(s);};
Consumer<String> consumer4 = s -> System.out.println(s);
consumer1.accept("a");
consumer2.accept("b");
consumer3.accept("cde");
consumer4.accept("qww");
```

- **Supplier**

  > 供给型接口，无输入参数，返回一个T类型的值

```
@FunctionalInterface
public interface Supplier<T> {
   T get();
}
```

```
// 使用
Supplier<String> supplier1 = () -> {return "supplier";};
Supplier<String> supplier2 = () ->  "supplier...";
System.out.println(supplier1.get());
System.out.println(supplier2.get());
```

### 3. Stream流

- 流（Stream）到底是什么?

  > 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

  1. Stream自己不会存储元素
  2. Stream不会改变源对象。相反，他们会返回一个持有结果的新Stream
  3. 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

- 使用

  1. 创建一个Stream：一个数据源（数组，集合）
  2. 中间操作：一个中间操作，处理数据源操作
  3. 终止操作：一个终止操作，执行中间操作链，产生结果

  源头 ==> 中间流水线 ==> 结果

**4. 实际操作**

```
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);

// 排序
List<User> collect = list.stream().sorted((t1, t2) -> {
 return u1.getId().compareTo(u2.getAge());
}).collect(Collectors.toList());

// 偶数ID 且年龄大于24 且用户名称转为大写 且用户名字字母倒排序，只输出一个用户名字
list.stream()  // to stream
 .filter(u -> {return u.getId()%2==0;}) // 过滤
 .filter(u -> {return u.getAge() > 24;})
 .map(u -> {return u.getUserName().toUpperCase();}) // 映射，这里已经是 String
啦
 .sorted((s1,s2) -> {return s2.compareTo(s1);}) // 排序
 .limit(1); // 截取

// 极简模式
list.stream()
 .filter(u -> u.getId()%2==0)
 .filter(u -> u.getAge() > 24)
 .map(u -> u.getUserName().toUpperCase()) // 这里已经是 String 啦
 .sorted((s1,s2) -> s2.compareTo(s1))
 .limit(1);
```

> stream(): list ==> stream
>
> collect() : stream ==> list

## 十二、分支合并框架

> ForkJoinPool，ForkJoinTask，RecursiveTask（递归任务）

```
// 资源类
// 递归任务
class MyTask extends RecursiveTask<Integer> {
```

```
    private int DEFAULT_VALUE = 10;
    private int start;
    private int end;
    private int result;
    @Override
    protected Integer compute() {
        if ((end - start) <= DEFAULT_VALUE) {
            for (int i = start; i <=end ; i++) {
                result = result + i;
            }
        } else {
            int middle = (end+start)/2;
            MyTask myTask1 = new MyTask(start,middle);
            MyTask myTask2 = new MyTask(middle+1,end);
            myTask1.fork();  // 分支fork
            myTask2.fork();
            result = myTask1.join() + myTask2.join(); // 合并join
        }
        return result;
    }
    public MyTask() {
    }
    public MyTask(Integer start, Integer end) {
        this.start = start;
        this.end = end;
    }
}
```

```
MyTask myTask = new MyTask(0,100);

ForkJoinPool forkJoinPool = new ForkJoinPool();
ForkJoinTask<Integer> forkJoinTask = forkJoinPool.submit(myTask);
System.out.println(forkJoinTask.get());
forkJoinPool.shutdown();
```

## 十三、异步回调

```
CompletableFuture<T>
CompletableFuture.runAsync() // 无返回值
CompletableFuture.supplyAsync() //有返回值
```

- 案例

```
CompletableFuture<Void> voidCompletableFuture =
CompletableFuture.runAsync(() ->{
 System.out.println(Thread.currentThread().getName()+"没有返回值的
CompletableFuture.runAsync...");
});
voidCompletableFuture.get();

CompletableFuture<Integer> integerCompletableFuture =
CompletableFuture.supplyAsync(() ->{
 System.out.println("有返回值的CompletableFuture.supplyAsync...");
 //int i = 10/0;
 return 1024;
});
```

```
integerCompletableFuture.whenComplete((t, u) -> {
 System.out.println("****t:" + t);
 System.out.println("****u:" + u);
}).exceptionally(t -> { // 处理异常情况
 System.out.println("异常监听exceptionally..."+t.getMessage());
 return 2048;
}).get();
```

2019-12学习笔记<王钦>