# 锁



# synchronized与lock

1. 多线程交互中，必须防止多线程的虚假唤醒，也即（ 判断只用while，不能用if ）。

   什么是虚假唤醒：

   　　当一个条件满足时，很多线程都被唤醒了，但是只有其中部分是有用的唤醒，其它的唤醒都是无用功。

   为什么 if会出现虚假唤醒：

   　　因为if只会执行一次，执行完会接着向下执行if（）外边的，

   　　而while不会，直到条件满足才会向下执行while（）外边的。

2. synchronized的新写法为lock

```Java
private Lock lock = new ReentrantLock();
private Condition condition = lock.newCondition();
其中 wait()被condition.await()代替，notifyAll()被condition.signalAll()代替.
```

   lock的使用方式：

```java
Lock lock = new ReentrantLock();
lock.lock();
try{
    //处理任务
}catch(Exception ex){

}finally{
    lock.unlock();   //释放锁
}
```

3. 为什么要用lock代替synchronized?

   lock可以定义多个condition（可以理解为一把锁有多个钥匙），精准通知，精准唤醒。

   通过设置多个condition精确通知需要唤醒的线程，可以指定线程顺序调用。

# 8锁理解

① 非静态方法的默认锁是this， 静态方法的默认锁是class类。

②某一时刻内，只能有一个线程有锁，无论几个方法。

Java 复制代码

```java
/*
1、标准的访问情况下，先执行 sendEmail 还是 sendSMS

   答案：sendEmail
   被 synchronized 修饰的方法，锁的对象是方法的调用者也就是实际new的对象，所以说这里两个方法调用的对象是同一个
   先调用的先执行！
 */
public class LockDemo01 {
    public static void main(String[] args) throws InterruptedException {
        Phone phone = new Phone();

        new Thread(()->{phone.sendEmail();},"A").start();
        TimeUnit.SECONDS.sleep(2);
        new Thread(()->{phone.sendSMS();},"B").start();
    }
}
class Phone{
    public synchronized void sendEmail(){
        System.out.println("sendEmail");
    }
    public synchronized void sendSMS(){
        System.out.println("sendSMS");
    }
}
```

Java 复制代码

```java
/*
2、sendEmail休眠4秒后 ， 先执行 sendEmail 还是 sendSMS

   答案：sendEmail
   被 synchronized 修饰的方法，锁的对象是方法的调用者，所以说这里两个方法调用的对象是同一个。 先调用的先执行！执行sleep（）方法的线程并不会释放锁。

   阳哥笔记：
   一个对象里面如果有多个synchronized方法，某一个时刻内，只要一个线程去调用其中的一个synchronized方法了，其他线程都只能等待，换句话说，某一个时刻内，只能有唯一一个线程去访问这些synchronized方法，锁的是当前对象this，被锁定后，其他线程都不能进入到当前对象的其他的synchronized方法。
 */
public class LockDemo02 {
    public static void main(String[] args) throws InterruptedException {
        Phone2 phone = new Phone2();

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

        //Thread.sleep(200);
        TimeUnit.SECONDS.sleep(2);

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

class Phone2{
    public synchronized void sendEmail() throws InterruptedException {
        TimeUnit.SECONDS.sleep(4);
        System.out.println("sendEmail");
    }

    public synchronized void sendSMS(){
        System.out.println("sendSMS");
    }
}
```

Java 复制代码

```java
/*
3、增加一个普通方法，请问先打印那个 sendEmail 还是 hello

   答案：hello
   新增加的这个方法没有 synchronized 修饰，不是同步方法，不受锁的影响！
 */
public class LockDemo03 {
    public static void main(String[] args) throws InterruptedException {
        Phone3 phone = new Phone3();

        new Thread(()->{
            try {
                phone.sendEmail();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        },"A").start();
        TimeUnit.SECONDS.sleep(1);

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

class Phone3{
    public synchronized void sendEmail() throws InterruptedException {
        TimeUnit.SECONDS.sleep(4);
        System.out.println("sendEmail");
    }

    // 没有 synchronized 没有 static 就是普通方法
    public void hello(){
        System.out.println("hello");
    }
}
```

Java 复制代码

```java
/*
4、两个手机，请问先执行sendEmail 还是 sendSMS
    答案：sendSMS
    被 synchronized  修饰的方式，锁的对象是调用者；我们这里有两个调用者，两个方法在这里是两个锁
 */
public class LockDemo04 {
    public static void main(String[] args) throws InterruptedException {
        Phone4 phone1 = new Phone4();
        Phone4 phone2 = new Phone4();

        new Thread(()->{
            try {
                phone1.sendEmail();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        },"A").start();

        //Thread.sleep(200);
        TimeUnit.SECONDS.sleep(1);

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

class Phone4{
    public synchronized void sendEmail() throws InterruptedException {
        TimeUnit.SECONDS.sleep(3);
        System.out.println("sendEmail");
    }

    public synchronized void sendSMS(){
        System.out.println("sendSMS");
    }
}
```

Java 复制代码

```java
/*
5、两个静态同步方法，同一个手机请问先执行sendEmail 还是 sendSMS

    答案：sendEmail
    只要方法被 static 修饰，锁的对象就是 Class模板对象,这个则全局唯一！所以说这里是同一个锁
    并不是因为synchronized
 */
public class LockDemo05 {
    public static void main(String[] args) throws InterruptedException {
        Phone5 phone = new Phone5();


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

        //Thread.sleep(200);
        TimeUnit.SECONDS.sleep(1);

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

class Phone5{

    public static synchronized void sendEmail() throws InterruptedException {
        TimeUnit.SECONDS.sleep(3);
        System.out.println("sendEmail");
    }

    public static synchronized void sendSMS(){
        System.out.println("sendSMS");
    }

}
```

Java 复制代码

```java
/*
6、两个静态同步方法，两个手机，请问先执行sendEmail 还是 sendSMS

    答案：sendEmail
    只要方法被 static 修饰，锁的对象就是 Class模板对象,这个则全局唯一！所以说这里是同一个锁
    并不是因为synchronized
 */
public class LockDemo06 {
    public static void main(String[] args) throws InterruptedException {
        Phone6 phone = new Phone6();
        Phone6 phone2 = new Phone6();

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

        //Thread.sleep(200);
        TimeUnit.SECONDS.sleep(1);

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

class Phone6{

    public static synchronized void sendEmail() throws InterruptedException {
        TimeUnit.SECONDS.sleep(3);
        System.out.println("sendEmail");
    }

    public static synchronized void sendSMS(){
        System.out.println("sendSMS");
    }

}
```

Java 复制代码

```java
/*
7、一个普通同步方法，一个静态同步方法，只有一个手机，请问先执行sendEmail 还是 sendSMS

    答案：sendSMS
    synchronized 锁的是这个调用的对象
    static 锁的是这个类的Class模板
    这里是两个锁！互不影响
 */
public class LockDemo07 {
    public static void main(String[] args) throws InterruptedException {
        Phone7 phone = new Phone7();

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

        //Thread.sleep(200);
        TimeUnit.SECONDS.sleep(1);

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

class Phone7{

    public static synchronized void sendEmail() throws InterruptedException {
        TimeUnit.SECONDS.sleep(3);
        System.out.println("sendEmail");
    }

    public synchronized void sendSMS(){
        System.out.println("sendSMS");
    }

}
```

Java 复制代码

```java
/*
8、一个普通同步方法，一个静态同步方法，两个手机，请问先执行sendEmail 还是 sendSMS

    答案：sendSMS
    synchronized 锁的是这个调用的对象
    static 锁的是这个类的Class模板
    这里是两个锁！
 */
public class LockDemo08 {
    public static void main(String[] args) throws InterruptedException {
        Phone8 phone = new Phone8();
        Phone8 phone2 = new Phone8();

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

        //Thread.sleep(200);
        TimeUnit.SECONDS.sleep(1);

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

class Phone8{

    public static synchronized void sendEmail() throws InterruptedException {
        TimeUnit.SECONDS.sleep(3);
        System.out.println("sendEmail");
    }

    public synchronized void sendSMS(){
        System.out.println("sendSMS");
    }

}
```

**8锁小结**

1、new this 调用的这个对象，是一个具体的对象！

2、static class 唯一的一个模板！

在我们编写多线程程序得时候，只需要搞明白这个到底锁的是什么就不会出错了！

```
synchronized(Demo.class){
}//等同于static synchronized 方法

synchronized(this){
}//等同于synchronized普通方法
```

# List不安全

**举例说明list线程不安全**

注意：ArrayList不是为并发情况而设计的集合类

举例：多个线程操作ArraList，边写边读，会出现并发修改异常.

**故障现象**：

java.util.**ConcurrentModificationException** 并发修改异常.

**导致原因**：

**解决方法**：

1） List<String> list = new Vector<>(); // vector 线程安全，add方法有synchronized，读写性能低,同一时间段只能一个人读/写.

2）List<String> list = Collections.synchronizedList(new ArrayList<>()); // Collections工具类提供，将ArrayList转换为线程安全.

3）List<String> list = new CopyOnWriteArrayList(); // JUC包提供的 写时复制(读写分离思想).

### CopyOnWriteArrayList浅析

和ArrayList一样，其底层数据结构也是数组，加上transient不让其被序列化，加上volatile修饰来保证多线程下的其可见性和有序性。

其修改操作是基于fail-safe机制，像我们的String一样，不在原来的对象上直接进行操作，而是复制一份对其进行修改，另外此处的修改操作是利用Lock锁进行上锁的，所以保证了线程安全问题。

### CopyOnWriteArrayList的几个要点

- 实现了List接口
- 内部持有一个ReentrantLock lock = new ReentrantLock();
- 底层是用volatile transient声明的数组 array

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

**volatile** （挥发物、易变的）：变量修饰符，只能用来修饰变量。volatile修饰的成员变量在每次被线程访问时，都强迫从共享内存中重读该成员变量的值。而且，当成员变量发生变 化时，强迫线程将变化值回写到共享内存。这样在任何时刻，两个不同的线程总是看到某个成员变量的同一个值。

**transient** （暂短的、临时的）：修饰符，只能用来修饰字段。在对象序列化的过程中，标记为transient的变量不会被序列化。

### CopyOnWrite的缺点

CopyOnWrite容器有很多优点，但是同时也存在两个问题，即内存占用问题和数据一致性问题。所以在开发的时候需要注意一下。

**内存占用问题**。因为CopyOnWrite的写时复制机制，所以在进行写操作的时候，内存里会同时驻扎两个对象的内存，旧的对象和新写入的对象（注意:在复制的时候只是复制容器里的引用，只是在写的时候会创建新对象添加到新容器里，而旧容器的对象还在使用，所以有两份对象内存）。如果这些对象占用的内存比较大，比如说200M左右，那么再写入100M数据进去，内存就会占用300M，那么这个时候很有可能造成频繁的Yong GC和Full GC。之前我们系统中使用了一个服务由于每晚使用CopyOnWrite机制更新大对象，造成了每晚15秒的Full GC，应用响应时间也随之变长。

针对内存占用问题，可以通过压缩容器中的元素的方法来减少大对象的内存消耗，比如，如果元素全是10进制的数字，可以考虑把它压缩成36进制或64进制。或者不使用CopyOnWrite容器，而使用其他的并发容器，如ConcurrentHashMap。

**数据一致性问题**。CopyOnWrite容器只能保证数据的最终一致性，不能保证数据的实时一致性。所以如果你希望写入的的数据，马上能读到，请不要使用CopyOnWrite容器。

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

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

# Set 不安全

HashSet底层数据结构是HashMap。 hashSet的add方法底层调用的就是hashMap的put方法，为什么hashMap的put是2个参数，而hashSet的add是一个参数呢，因为add进去的一个值就是put的key，value永远是一个Object类型的常量: PRSENT，固定写死的。

```
Set<String> set =  new CopyOnWriteArraySet();  // new HashSet<>();
Collections.synchronizedArraySet(new HashSet());
```

# Map 不安全

Map<String,Object> map = new ConCurrentHashMap<>();  // new HashMap<>();

Collections.synchronizedMap();