# JAVA多线程

**一、synchronized关键字**

**1.初始代码：**

**ThreadSafeDemo1.java**

package NoSafeThread;

import java.util.Iterator;

import java.util.List;

import java.util.concurrent.TimeUnit;

public class ThreadSafeDemo1 implements Runnable{

private List<String> list;

public ThreadSafeDemo1(List<String> list){

this.list = list;

}

@Override

public void run() {

System.out.println( Thread.currentThread().getName()+"开始!");

//synchronized(this) 修饰代码块，作用于调用的对象

synchronized(this){

Iterator<String> iterator = list.iterator();

int cnt = 0;

//移除list中的所有元素

while(iterator.hasNext()){

iterator.next();

iterator.remove();

cnt++;

try {

TimeUnit.MILLISECONDS.sleep(10);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

System.out.println(Thread.currentThread().getName()+"结束!"+"移除元素的数量："+cnt);

}

}

}

**ThreadMain.java**

package NoSafeThread;

import java.util.ArrayList;

import java.util.List;

import java.util.Vector;

public class ThreadMain {

public static void main(String[] args) {

List<String> list = new ArrayList<>(1000);

for(int cnt=0;cnt<1000;cnt++){

list.add("abc");

}

//创建线程

ThreadSafeDemo1 demo1 = new ThreadSafeDemo1(list);

ThreadSafeDemo1 demo2 = new ThreadSafeDemo1(list);

Thread thread1 = new Thread(demo1);

Thread thread2 = new Thread(demo2);

//设置线程名

thread1.setName("A");

thread2.setName("B");

//启动线程

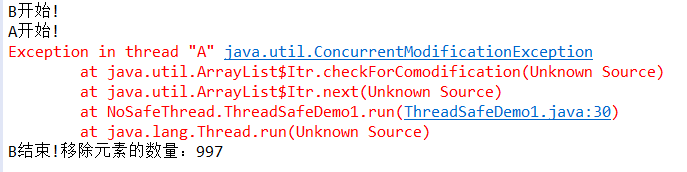
thread1.start();

thread2.start();

}

}

**2.问题描述：**



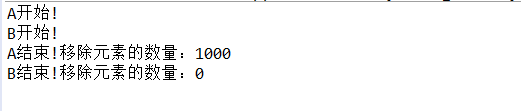
**3.问题分析：**

可以看到，线程B已经结束，线程A抛出异常。因为ArrayList是线程不安全的，所以对同一个list进行访问时，同时对一个数据进行修改，就可能会造成数据的不一致性。 当一个线程的时间片结束，而结果还没有写回，下次执行时再写回，就会覆盖掉之前的内容，使得数据不一致。运行结果中显示移除的元素数量为997而不是1000，就是这个道理。

解决方法是给共享的资源加把锁，保证每个资源变量每时每刻至多被一个线程占用。此例虽然使用了synchronized(this)声明同步语句块，但只作用于当前对象，不同对象无法同步。

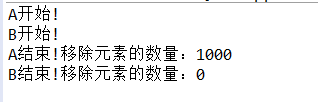
这里共享的是list,所以将synchronized(this)改为synchronized(list),即可解决问题。

**4.正确结果：**

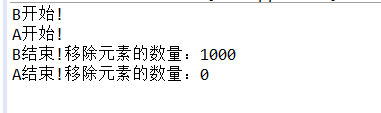


**5.其他解决办法：**

（1）将synchronized(this)改为synchronized(ThreadSafeDemo1.class)，对当前对应的\*.Class进行持锁。



（2） 在ThreadSafeDemo1类中添加静态成员变量private static Object o=new Object();再将synchronized(this)改为synchronized(o),也是对当前对应的\*.Class进行持锁。



（3）synchronized修饰静态方法，也是对当前对应的\*.Class进行持锁。

**ThreadSafeDemo1.java**

package NoSafeThread;

import java.util.Iterator;

import java.util.List;

import java.util.concurrent.TimeUnit;

public class ThreadSafeDemo1 implements Runnable{

private List<String> list;

public ThreadSafeDemo1(List<String> list){

this.list = list;

}

public synchronized static void runService(List<String> list) {

Iterator<String> iterator = list.iterator();

int cnt = 0;

//移除list中的所有元素

while(iterator.hasNext()){

iterator.next();

iterator.remove();

cnt++;

try {

TimeUnit.MILLISECONDS.sleep(10);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

System.out.println(Thread.currentThread().getName()+"结束!"+"移除元素的数量："+cnt);

}

@Override

public void run() {

System.out.println( Thread.currentThread().getName()+"开始!");

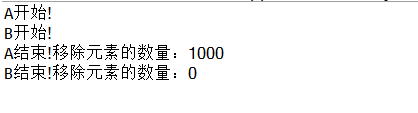
runService(this.list);

}

}

**ThreadMain.java不变**

**运行结果：**



**6.其他尝试：**

（1）synchronized修饰方法，作用于调用的对象。不同对象，无法同步。

**ThreadSafeDemo1.java**

package NoSafeThread;

import java.util.Iterator;

import java.util.List;

import java.util.concurrent.TimeUnit;

public class ThreadSafeDemo1 implements Runnable{

private List<String> list;

public ThreadSafeDemo1(List<String> list){

this.list = list;

}

public synchronized void runService() {

Iterator<String> iterator = list.iterator();

int cnt = 0;

//移除list中的所有元素

while(iterator.hasNext()){

iterator.next();

iterator.remove();

cnt++;

try {

TimeUnit.MILLISECONDS.sleep(10);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

System.out.println(Thread.currentThread().getName()+"结束!"+"移除元素的数量："+cnt);

}

@Override

public void run() {

System.out.println( Thread.currentThread().getName()+"开始!");

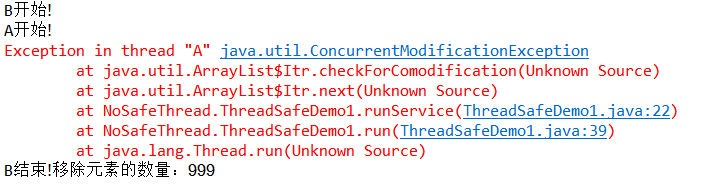
runService();

}

}

**ThreadMain.java不变**

**运行结果：**



（2）synchronized（任意对象），作用于调用的对象。不同对象，无法同步。

**ThreadSafeDemo1.java**

package NoSafeThread;

import java.util.Iterator;

import java.util.List;

import java.util.concurrent.TimeUnit;

public class ThreadSafeDemo1 implements Runnable{

private List<String> list;

public ThreadSafeDemo1(List<String> list){

this.list = list;

}

@Override

public void run() {

System.out.println( Thread.currentThread().getName()+"开始!");

Object o=new Object();

synchronized(o){

Iterator<String> iterator = list.iterator();

int cnt = 0;

//移除list中的所有元素

while(iterator.hasNext()){

iterator.next();

iterator.remove();

cnt++;

try {

TimeUnit.MILLISECONDS.sleep(10);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

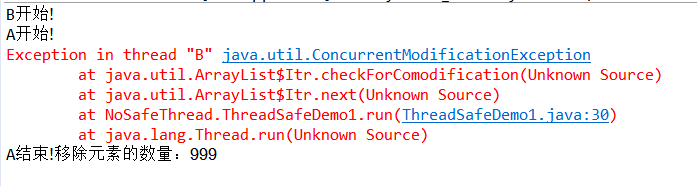
System.out.println(Thread.currentThread().getName()+"结束!"+"移除元素的数量："+cnt);

}

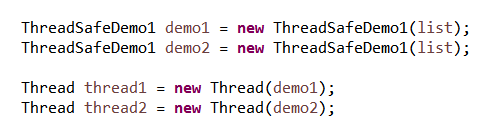
}

}

**ThreadMain.java不变**

**运行结果：**

**7.小结：**



本例中是两个不同对象，共享资源是list。而ArrayList又是线程不安全的，所以要想实现线程安全，可以采用synchronized标识的同步代码块来实现对共享资源的互斥访问。

如果有一块代码(或方法)可能被多个线程同时访问，共享的数据可能因为不同线程的修改操作而不一致的时候，使用synchronized锁定这块代码，确保同时只有一个线程访问这个代码块。

在Java中，每个对象都自带监视锁，当多个线程要访问用synchronized修饰的方法或代码块的时候，都意味着进入这个方法或者代码块要加锁，离开要放锁。而且Synchronizd可以显式地说明对哪个对象加锁。

synchronized修饰的对象有四种：  
（1）synchronized(this)和synchronized(普通变量)，都是对当前对象加锁，不同对象不能同步；  
（2）synchronized方法，对当前对象加锁，不同对象不能同步；  
（3）synchronized修饰静态方法和synchronized(静态成员变量)，作用于所有对象；除了当前对象以外，本类的其他新建对象也会同步加锁。不同对象能够同步；  
（4）synchronized(\*.class)修饰类，作用于所有对象。Class锁对类的所有对象实例起作用。不同对象能够同步。

## 二、ArrayList换成Vector

ThreadMain.Java

public class ThreadMain {

public static void main(String[] args) {

//使用Vector集合

List<String> list = new Vector<>(1000);

for(int cnt=0;cnt<1000;cnt++){

list.add("abc");

}

ThreadSafeDemo1 demo1 = new ThreadSafeDemo1(list);

ThreadSafeDemo1 demo2 = new ThreadSafeDemo1(list);

Thread thread1 = new Thread(demo1);

Thread thread2 = new Thread(demo2);

//设置线程名

thread1.setName("A");

thread2.setName("B");

//启动线程

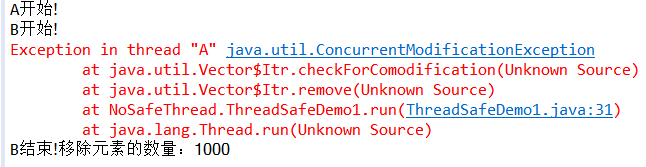
thread1.start();

thread2.start();

}

}

**运行结果：**



**结果分析：**

可以看到问题仍然存在，虽然源代码注释里面说Vector是线程安全的，因为确实很多方法都加上了同步关键字synchronized，但是对于符合操作而言，只是同步方法并没有解决线程安全的问题。要想真正达成线程安全，还需要以Vector对象为锁，来进行操作。

不管是Vector还是List容器，都可能会出现ConCurrenceModificationException的异常。这是因为几乎所有的集合类都有快速失败的（Fail-Fast）校验机制。这是为了确保集合方法一致而设立的保护措施。它的实现原理就是modCount修改计数器。如在遍历列表的时候（使用迭代器的方式），这时候会保存当前的modCount值，当每次迭代器取值的时候，会通过checkForComodification（）方法来校验modCount是否发生了改变。如果发生了改变，就会抛ConCurrenceModificationException的异常。

其实这里的异常和线程同步是两码事。因为使用迭代器遍历容器的时候，这是记录了modCount值，然后调用了remove方法，这在单线程中本来就是不允许的，和多线程同步没有关系。线程同步是为了实现线程安全，而在Vector中则是实现了线程的部分安全。  
 Vector的方法都是同步的，并且Vector明确地设计为在多线程环境中工作。但是它的线程安全性是有限制的，即在某些方法之间有状态依赖(类似地，如果在迭代过程中Vector被其他线程修改，那么由Vector.iterator()返回的iterator会抛出ConcurrentModificationException异常。

三、示例二

ThreadSafeDemo2.Java

package NoSafeThread;

import java.util.List;

public class ThreadSafeDemo2 implements Runnable{

private List<String> list;

public ThreadSafeDemo2(List<String> list){

this.list = list;

}

@Override

public void run() {

//添加元素

list.add("版权归作者所有，任何形式转载请联系作者。");

System.out.println("添加一个后尺寸："+list.size());

}

}

ThreadSafeDemo3.java

package NoSafeThread;

import java.util.List;

public class ThreadSafeDemo3 implements Runnable{

private List<String> list;

public ThreadSafeDemo3(List<String> list){

this.list = list;

}

@Override

public void run() {

if(list.size()>0){

//删除元素

list.remove(0);

System.out.println("删除一个后尺寸："+list.size());

}

}

}

ThreadMain.java

package NoSafeThread;

import java.util.ArrayList;

import java.util.List;

public class ThreadMain {

public static void main(String[] args) {

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

ThreadSafeDemo2 demo1 = new ThreadSafeDemo2(list);

ThreadSafeDemo2 demo2 = new ThreadSafeDemo2(list);

ThreadSafeDemo2 demo3 = new ThreadSafeDemo2(list);

ThreadSafeDemo2 demo4 = new ThreadSafeDemo2(list);

ThreadSafeDemo2 demo5 = new ThreadSafeDemo2(list);

ThreadSafeDemo2 demo6 = new ThreadSafeDemo2(list);

ThreadSafeDemo2 demo7 = new ThreadSafeDemo2(list);

ThreadSafeDemo2 demo8 = new ThreadSafeDemo2(list);

ThreadSafeDemo2 demo9 = new ThreadSafeDemo2(list);

ThreadSafeDemo3 demo10 = new ThreadSafeDemo3(list);

new Thread(demo1).start();

new Thread(demo2).start();

new Thread(demo3).start();

new Thread(demo4).start();

new Thread(demo5).start();

new Thread(demo6).start();

new Thread(demo7).start();

new Thread(demo8).start();

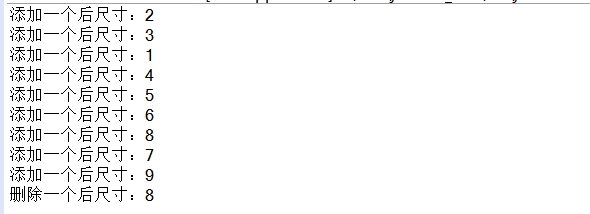
new Thread(demo9).start();

new Thread(demo10).start();

}

}

**运行结果：**



可以看出，第一行中添加一个后尺寸变为2，第三行添加一个后尺寸变为1。显然，这样的结果是不合理的，说明这里的多线程并发是不安全的。

**解决方法：**

在ThreadSafeDemo2.java中，加上同步语句块

public void run() {

synchronized(list) {

list.add("版权归作者所有，任何形式转载请联系作者。");

System.out.println("添加一个后尺寸："+list.size());

}

}

在ThreadSafeDemo3.java中，加上同步语句块

public void run() {

synchronized(list) {

if(list.size()>0){

list.remove(0);

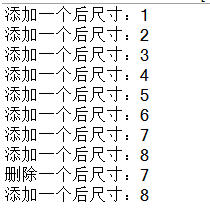
System.out.println("删除一个后尺寸："+list.size());

}

}

}

**运行结果：**



## 四、阻塞队列

阻塞队列：最常见的业务场景就是生产者不断生产任务放进阻塞队列中，消费者不断从阻塞队列中获取任务；当阻塞队列中填满数据时，所有生产者端的线程自动阻塞；当阻塞队列中数据为空时，所有消费端的线程自动阻塞。这些操作BlockingQueue都已经包办了，不用我们操心。

阻塞队列会对当前线程产生阻塞，比如一个线程从一个空的阻塞队列中取元素，此时线程会被阻塞直到阻塞队列中有了元素。当队列中有元素后，被阻塞的线程会自动被唤醒（不需要我们编写代码去唤醒）。这样提供了极大的方便性。

ArrayBlockingQueue：基于数组实现的一个阻塞队列，在创建ArrayBlockingQueue对象时必须制定容量大小。并且可以指定公平性与非公平性，默认情况下为非公平的，即不保证等待时间最长的队列最优先能够访问队列。

LinkedBlockingQueue：基于链表实现的一个阻塞队列，在创建LinkedBlockingQueue对象时如果不指定容量大小，则默认大小为Integer.MAX\_VALUE。

**ThreadMain .java**

package NoSafeThread;

import java.util.concurrent.ArrayBlockingQueue;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class ThreadMain {

public static void main(String[] args) {

//指定阻塞队列的大小为10

ArrayBlockingQueue<String> queue =new ArrayBlockingQueue<String>(10);

//使用线程池

ExecutorService service=Executors.newCachedThreadPool();

//创建生产者

ThreadSafeDemo2 demo1 = new ThreadSafeDemo2(queue);

//创建消费者

ThreadSafeDemo3 demo2 = new ThreadSafeDemo3(queue);

//启动线程

service.execute(demo1);

service.execute(demo2);

}

}

ThreadSafeDemo2.java

package NoSafeThread;

import java.util.concurrent.ArrayBlockingQueue;

import java.util.concurrent.BlockingQueue;

public class ThreadSafeDemo2 implements Runnable{

private BlockingQueue<String> queue=new ArrayBlockingQueue<String>(10);

private int queueSize = 10;

public ThreadSafeDemo2(BlockingQueue<String> queue){

this.queue = queue;

}

@Override

public void run() {

while(true){

try {//不停地插入元素

queue.put("a");

System.out.println("向队列插入一个元素，队列剩余空间："+(queueSize-queue.size()));

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

ThreadSafeDemo3.java

package NoSafeThread;

import java.util.concurrent.BlockingQueue;

public class ThreadSafeDemo3 implements Runnable{

private BlockingQueue<String> queue;

public ThreadSafeDemo3(BlockingQueue<String> queue){

this.queue = queue;

}

@Override

public void run() {

while(true){

try {//不停地移除首元素

queue.take();

System.out.println("从队列取走一个元素，队列剩余"+queue.size()+"个元素");

} catch (InterruptedException e) {

e.printStackTrace();

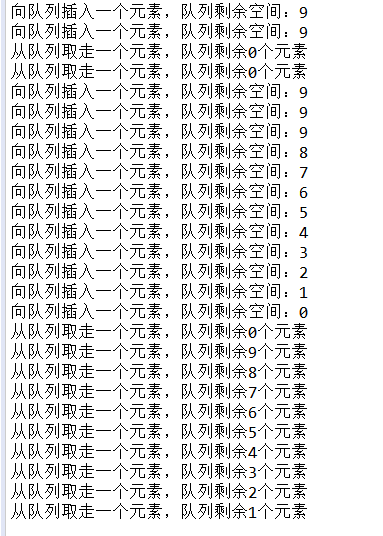
}

}

}

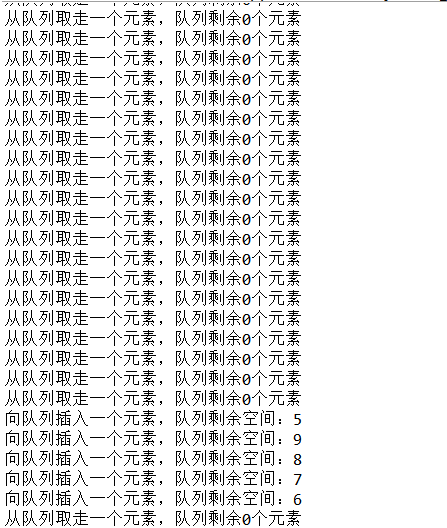
}

**运行结果:**



可以看到如果队列为空，消费者就会阻塞，不会进行消费直到有数据被生产出来。

下图为使用poll时的非阻塞效果。若阻塞队列为空，poll返回null.



## 五、上机总结

通过此次上机，动手实现了一些代码，对线程的理解更加深刻。从发现问题、思考问题

最后解决问题这一系列过程中，看了很多有关线程的资料，可以说收获很多。

改正一些错误的同时，对线程安全和线程同步等概念有了更清楚的认识。并且学会了线程的创建和启动，掌握了同步语句块和线程池的使用方法，通过实现阻塞队列也学习了经典的生产者-消费者模型。

对以前学的集合类有了更进一步的认识，ArrayList是线程不安全的，Vector实现了线程的部分安全。

明确了线程安全的概念后，可以设想到线程安全主要应用在后台服务程序中。例如：自动作业处理：比如定期备份日志、数据库。页面异步处理：比如大批量数据的核对工作。数据库的数据分析（待分析的数据太多）、迁移等。多步骤任务的处理，可根据步骤特征选用不同个数和特征的线程来协作处理。

使用多线程虽然可以加快处理数据的速度，但线程数要适量，因为大量的线程会影响性能，操作系统需要在它们之间切换。所以线程并不是越多越好。

感觉对线程的认识还很浅显，有更多的知识需要学习。我会怀抱着学习的热情，去走好接下来的每一步。