# java多线程学习

# volatile

关键字保证了在多线程中value的值是可见的，任何一个线程修改了value值，会将其立即写回内存当中

可以避免重排序

volatile就是只具备可见性这个原子特性的关键字，它修饰的变量能让线程每次访问时候都拿到最新的数据，意味着只要它做了修改，就马上共享给每个访问它的线程；\nsynchronized也一样具有可见性这个特点（volatile能做的工作它完全能做），不过它修饰的变量每次只能有一个线程访问（典型的互斥性吧）；\n可以肯定，用synchronized修饰的变量肯定不用volatile修饰啦；

Volatile、synchronized两者的区别联系

1.volatile本质是在告诉jvm当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取；synchronized则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。

2.volatile仅能使用在变量级别；synchronized则可以使用在变量、方法、和类级别的。

3.volatile仅能实现变量的修改可见性，不能保证原子性（线程A修改了变量还没结束时,另外的线程B可以看到已修改的值,而且可以修改这个变量,而不用等待A释放锁,因为Volatile 变量没上锁）；而synchronized则可以保证变量的修改可见性和原子性。

4.volatile不会造成线程的阻塞；synchronized可能会造成线程的阻塞和上下文切换。

5.volatile标记的变量不会被编译器优化；synchronized标记的变量可以被编译器优化。

6.在使用volatile关键字时要慎重，并不是只要简单类型变量使用volatile修饰，对这个变量的所有操作都是原子操作。当变量的值由自身决定时，如n=n+1、n++ 等，volatile关键字将失效。只有当变量的值和自身无关时对该变量的操作才是原子级别的，如n = m + 1，这个就是原级别的。所以在使用volatile关键时一定要谨慎，如果自己没有把握，可以使用synchronized来代替volatile。

7.“锁是昂贵的”，谨慎使用锁机制

使用volatile关键字修饰共享变量可以禁止重排序。若用volatile修饰共享变量，在编译时，会在指令序列中插入内存屏障来禁止特定类型的处理器重排序

　　volatile禁止指令重排序的一些规则：

　　1.当第二个操作是voaltile写时，无论第一个操作是什么，都不能进行重排序

　　2.当地一个操作是volatile读时，不管第二个操作是什么，都不能进行重排序

　　3.当第一个操作是volatile写时，第二个操作是volatile读时，不能进行重排序

## 可见性

当一个线程修改了共享变量时，另一个线程可以读取到这个修改后的值。

## 内存屏障（Memory Barriers）

处理器的一组指令，用于实现对内存操作的顺序限制。

## 缓冲行

CPU告诉缓存中可以分配的最小存储单位，处理器填写缓存行时，会加载整个缓存行。

## Lock前缀的指令

Lock前缀的指令在多核处理器下会发生两件事情：

1）将当前处理器的缓存行的数据协会到系统内存。

2）这个写回内存的操作会使其他CPU缓存了该内存的地址的数据无效。

## 缓存一致性协议

在多处理器下，为零保证各个处理器的缓存是一致的，每个处理器都会通过嗅探在总线上传播的数据来检查自己缓存的值是不是过期了。当处理器发现自己缓存行对应的地址被修改，就会将当前处理器的缓存行设置为无效状态。当处理器对这个数据进行读写的时候，会重新把数据从内存中读取到处理器缓存中。

## CAS

CompareAndSwap 比较并交换  
CAS操作需要输入两个值，一个旧值（执行CAS操作前的值，期望值）和一个新值，只有当当前值等于旧值时，才可以将当前值设置为新值，否则不设置。这是一个原子操作，由硬件保证。

CAS也是适用一些场合的，比如资源竞争小时，是非常适用的，不用进行内核态和用户态之间的线程上下文切换，同时自旋概率也会大大减少，提升性能，但资源竞争激烈时（比如大量线程对同一资源进行写和读操作）并不适用，自旋概率会大大增加，从而浪费CPU资源，降低性能；

## 重排序规则

从根本上来所，JMM 对编译器和处理器的重排序限制只有一条，只要不改变程序执行的结果（指的是单线程或者正确同步的多线程环境下），那么编译器和处理器怎么优化都可以。

## Volatile

从上面的Lock前缀指令和缓存一致性协议可以看出来，这就是volatile的实现原理了。  
实际上，valatile变量被写入时，确实加了一个Lock前缀的指定，以此来达到可见性的目的。

## Java并发容器

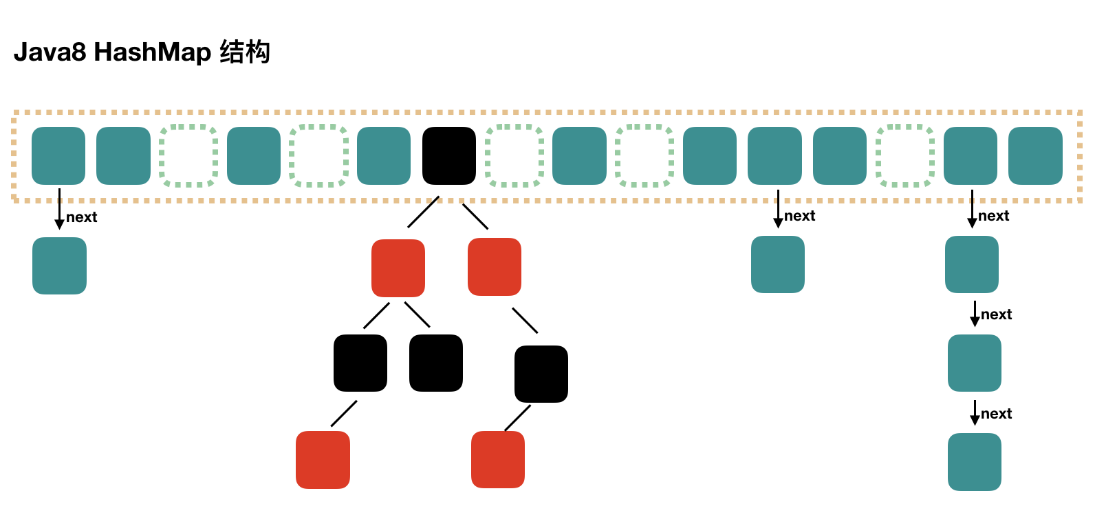
### ConcurrentHashMap

Java8 对 HashMap 进行了一些修改，最大的不同就是利用了红黑树，所以其由 数组+链表+红黑树 组成。

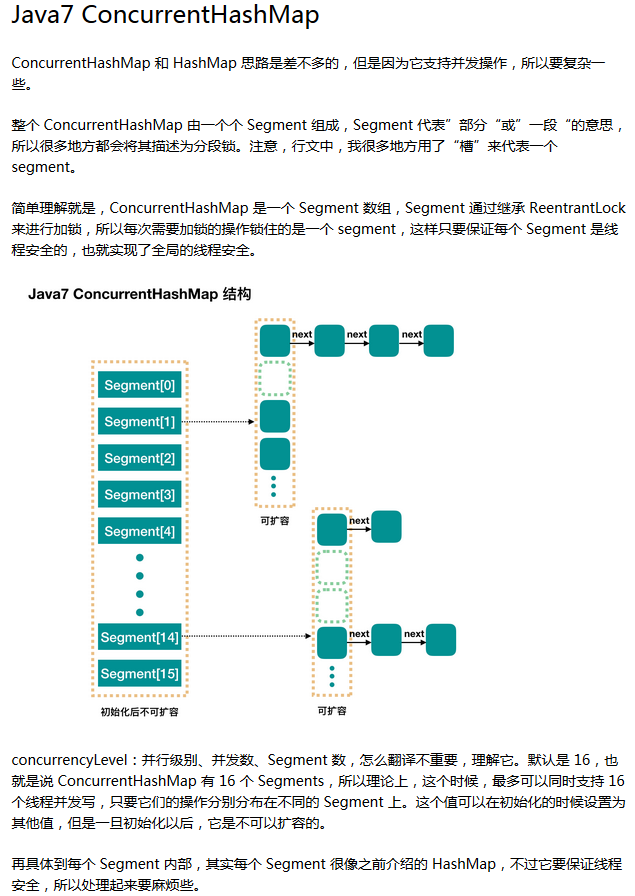
根据 Java7 HashMap 的介绍，我们知道，查找的时候，根据 hash 值我们能够快速定位到数组的具体下标，但是之后的话，需要顺着链表一个个比较下去才能找到我们需要的，时间复杂度取决于链表的长度，为 O(n)。

为了降低这部分的开销，在 Java8 中，当链表中的元素超过了 8 个以后，会将链表转换为红黑树，在这些位置进行查找的时候可以降低时间复杂度为 O(logN)。

来一张图简单示意一下吧：



注意，上图是示意图，主要是描述结构，不会达到这个状态的，因为这么多数据的时候早就扩容了。



ConcurrentHashMap比HashTable效率高的原意在于采取了分段加锁的思路，每个segment都是一个可重入锁

### ConcurrentLinkedQueue

在并发编程中，有时候需要使用线程安全的队列。如果要实现一个线程安全的队列有两  
种方式：一种是使用阻塞算法，另一种是使用非阻塞算法。使用阻塞算法的队列可以用一个锁  
（入队和出队用同一把锁）或两个锁（入队和出队用不同的锁）等方式来实现。非阻塞的实现方  
式则可以使用循环CAS的方式来实现。

ConcurrentLinkedQueue是用非阻塞的方式实现的

## 1.1传统线程

**package** cn.zz.test1;

**public** **class** TraditionalThread {

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

//继承重写run()

Thread thread1 = **new** Thread() {

@Override

**public** **void** run() {

**while**(**true**) {

System.out.println(Thread.currentThread().getName());

System.out.println(**this**.getName());

}

}

};

//thread1.start();

//实现Runnable接口

Thread thread2 = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

**while**(**true**) {

//Thread.currentThread()得到当前的线程对象

System.out.println(Thread.currentThread().getName());

//this不代表线程

//System.out.println(this.getName());

}

}

});

//thread2.start();

//第二种更能体现面向对象，线程所运行代码在对象里面

**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

System.out.println("Runnable");

}

}) {

@Override

**public** **void** run() {

System.out.println("Thread");

}

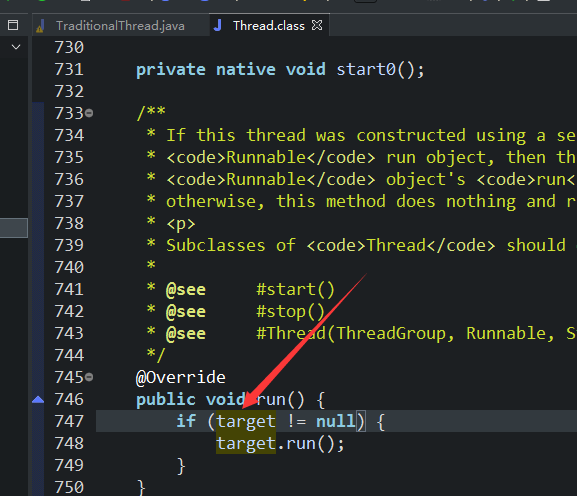
}.start();

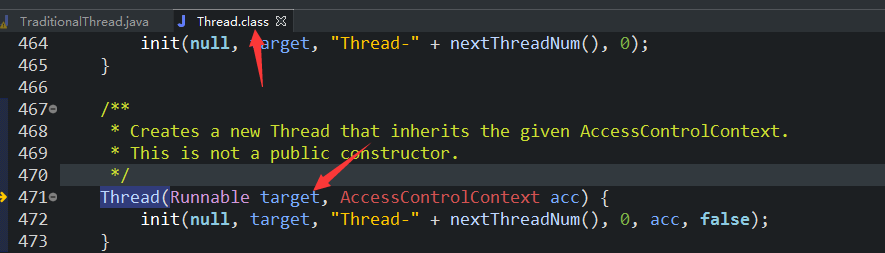
//输出Thread

}

}

原理，Thread源码说明如果target为空，就直接使用线程重写的run方法





## 1.2传统定时器

**package** cn.zz.test1;

**import** java.util.Date;

**import** java.util.Timer;

**import** java.util.TimerTask;

**public** **class** TraditionalTimer {

**private** **static** **int** count = 0;

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

/\*

\* new Timer().schedule(new TimerTask() {

\*

\* @Override public void run() { System.out.println("boom");

\*

\* } }, 10000,3000);

\*/

/\*

\* new Timer().schedule(new TimerTask() {

\*

\* @Override public void run() { System.out.println("1 boom"); new

\* Timer().schedule(new TimerTask() {

\*

\* @Override public void run() { System.out.println("2 boom");

\*

\*

\* } }, 2000);

\*

\* } }, 2000);

\*/

**class** MyTimerTask **extends** **TimerTask** {

@Override

**public** **void** run() {

count = (count+1)%2;

System.out.println("boom");

**new** Timer().schedule(**new** MyTimerTask(), 2000+2000\*count);

}

}

**new** Timer().schedule(**new** MyTimerTask(), 2000);

**while** (**true**) {

System.out.println(**new** Date().~~getSeconds~~());

**try** {

Thread.sleep(1000);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

}

## 1.3线程的同步互斥与通信

### 互斥：同一资源，同一对象，只能独自访问

#### 代码块互斥

**package** cn.zz.test1;

**public** **class** **TraditionalThreadSyn** {

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

**new** TraditionalThreadSyn().init();

}

**public** **void** **init**() {

**Outer** **outer** = **new** Outer();

**new** Thread(**new** Runnable() {

***@Override***

**public** **void** **run**() {

**while** (**true**) {

**/\*try {**

**Thread.sleep(10);**

**} catch (InterruptedException e) {**

**// TODO Auto-generated catch block**

**e.printStackTrace();**

**}\*/**

outer.log("nike");

}

}

}).start();

**new** Thread(**new** Runnable() {

***@Override***

**public** **void** **run**() {

**while** (**true**) {

**/\*try {**

**Thread.sleep(10);**

**} catch (InterruptedException e) {**

**// TODO Auto-generated catch block**

**e.printStackTrace();**

**}\*/**

outer.log("lining");

}

}

}).start();

}

**//互斥**

**class** **Outer** {

**String** syn = "";

**public** **void** **log**(**String** log) {

**//同一个对象**

**synchronized** (syn) {

**for** (**int** **i** = 0; i < log.length(); i++) {

**System**.***out***.print(log.charAt(i));

}

**System**.***out***.println();

}

**//用this也可以**

**synchronized** (**this**) {

**for** (**int** **i** = 0; i < log.length(); i++) {

**System**.***out***.print(log.charAt(i));

}

**System**.***out***.println();

}

}

}

}

#### 方法互斥

**package** cn.zz.test1;

**public** **class** **TraditionalThreadSyn2** {

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

**new** TraditionalThreadSyn2().init();

}

**public** **void** **init**() {

**Outer** **outer** = **new** Outer();

**new** Thread(**new** Runnable() {

***@Override***

**public** **void** **run**() {

**while** (**true**) {

**/\***

**\* try { Thread.sleep(10); } catch (InterruptedException e)**

**\* { // TODO Auto-generated catch block e.printStackTrace();**

**\* }**

**\*/**

outer.log("nike");

}

}

}).start();

**new** Thread(**new** Runnable() {

***@Override***

**public** **void** **run**() {

**while** (**true**) {

**/\***

**\* try { Thread.sleep(10); } catch (InterruptedException e)**

**\* { // TODO Auto-generated catch block e.printStackTrace();**

**\* }**

**\*/**

outer.log("lining");

}

}

}).start();

}

**// 互斥 同一个对象**

**class** **Outer** {

**//直接在方法上加互斥也可以**

**public** **synchronized** **void** **log**(**String** log) {

**for** (**int** **i** = 0; i < log.length(); i++) {

**System**.***out***.print(log.charAt(i));

}

**System**.***out***.println();

}

}

}

#### 静态方法互斥

**package** cn.zz.test1;

**public** **class** **TraditionalThreadSyn3class** {

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

**new** TraditionalThreadSyn3class().init();

}

**public** **void** **init**() {

**Outer** **outer** = **new** Outer();

**new** Thread(**new** Runnable() {

***@Override***

**public** **void** **run**() {

**while** (**true**) {

**/\***

**\* try { Thread.sleep(10); } catch (InterruptedException e)**

**\* { // TODO Auto-generated catch block e.printStackTrace();**

**\* }**

**\*/**

outer.log("nike");

}

}

}).start();

**new** Thread(**new** Runnable() {

***@Override***

**public** **void** **run**() {

**while** (**true**) {

**/\***

**\* try { Thread.sleep(10); } catch (InterruptedException e)**

**\* { // TODO Auto-generated catch block e.printStackTrace();**

**\* }**

**\*/**

outer.*log2*("lining");

}

}

}).start();

}

**// 互斥 同一个对象 类**

**static** **class** **Outer** {

**//直接在方法上加互斥也可以**

**public** **void** **log**(**String** log) {

**synchronized** (**Outer**.**class**) {

**for** (**int** **i** = 0; i < log.length(); i++) {

**System**.***out***.print(log.charAt(i));

}

**System**.***out***.println();

}

}

**public** **static** **synchronized** **void** **log2**(**String** log) {

**for** (**int** **i** = 0; i < log.length(); i++) {

**System**.***out***.print(log.charAt(i));

}

**System**.***out***.println();

}

}

}

### 总结：

要用到共同数据（包括同步锁）的若干的方法应该归在同一个类身上，这种设计正好体现了类聚和程序的健壮性。

两个线程执行的代码要实现同步互斥的效果，他们必须用同一个lock对象。锁是上在代表要操作的资源的类的内部方法中，而不是线程代码中。

例子：实现一个主线程执行10次，子线程执行5次，主次切换，循环50次；

思路：先写一个类，分别执行主线程，子线程，做互斥

**package** cn.zz.test1;

**public** **class** **TraditionalThreadCommicute** {

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

**Outer** **outer** = **new** Outer();

**new** Thread(**new** Runnable() {

***@Override***

**public** **void** **run**() {

**for** (**int** **i** = 0; i < 50; i++) {

outer.sub();

}

}

}).start();

**for** (**int** **i** = 0; i < 50; i++) {

outer.main();

}

}

}

**class** **Outer** {

**private** **Boolean** runMain = **true**;

**public** **synchronized** **void** **main**() {

**while** (!runMain) {

**try** {

**this**.wait();

} **catch** (**InterruptedException** **e**) {

**// TODO Auto-generated catch block**

e.printStackTrace();

}

}

**for** (**int** **i** = 0; i < 10; i++) {

**System**.***out***.println("main " + i);

}

runMain = **false**;

**this**.notify();

}

**public** **synchronized** **void** **sub**() {

**while** (runMain) {

**try** {

**this**.wait();

} **catch** (**InterruptedException** **e**) {

**// TODO Auto-generated catch block**

e.printStackTrace();

}

}

**for** (**int** **i** = 0; i < 5; i++) {

**System**.***out***.println("sub " + i);

}

runMain = **true**;

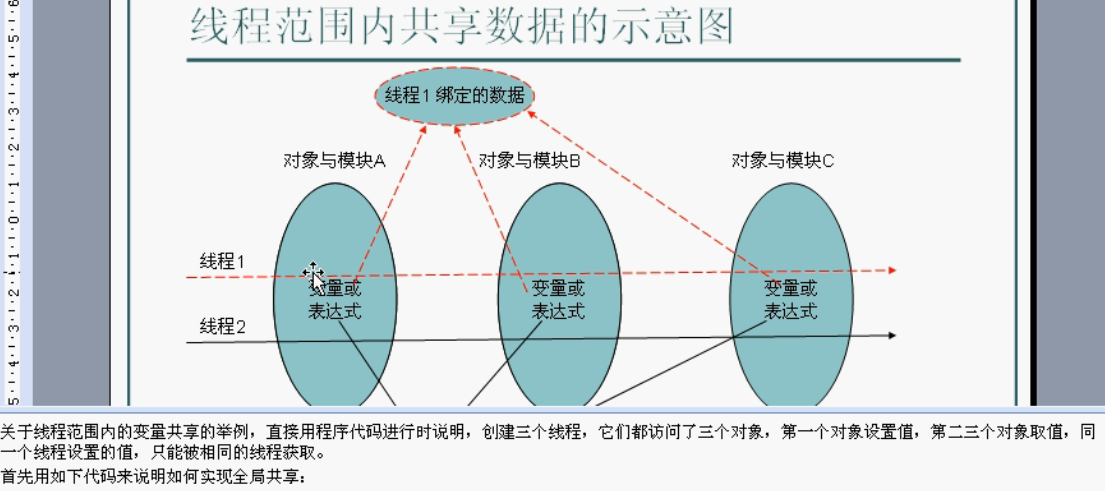
**this**.notify();

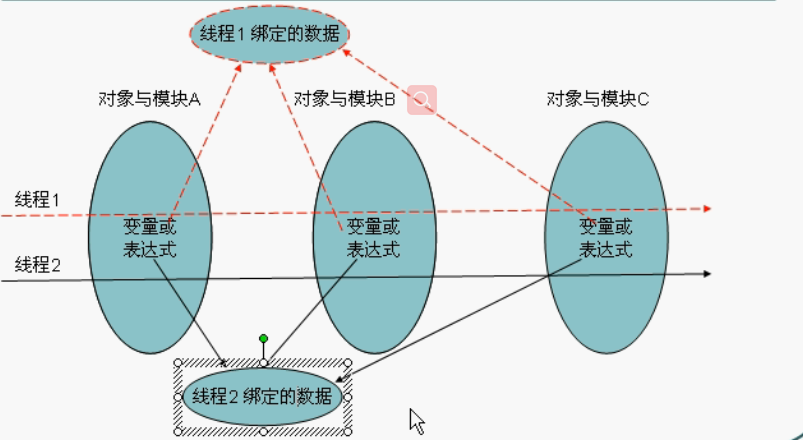
}

}

### 同步

## 2.1ThreadLocal实现线程范围的共享数据





一个ThreadLocal只能放一个变量，n个变量要定义n个ThreadLocal

ThreadLocal是一个map有多少个线程就有多少个ThreadLocal,线程结束时，底层会自动回收该线程的ThreadLocal

**package** cn.zz.test1;

**import** java.util.Random;

**public** **class** **ThreadLocalTest** {

**private** **static** **ThreadLocal**<Integer> *x* = **new** ThreadLocal<Integer>();

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

**for**(**int** **i**=0;i<3; i++){

**new** Thread(**new** Runnable() {

***@Override***

**public** **void** **run**() {

**int** **data** = **new** Random().nextInt();

**System**.***out***.println(**Thread**.*currentThread*().getName()+

": "+data);

*x*.set(data);

**new** A().get();

**new** B().get();

}

}).start();

}

}

**static** **class** **A**{

**public** **void** **get**(){

**int** **data** = *x*.get();

**System**.***out***.println("A "+**Thread**.*currentThread*().getName()+

": "+data);

}

}

**static** **class** **B**{

**public** **void** **get**(){

**int** **data** = *x*.get();

**System**.***out***.println("B "+**Thread**.*currentThread*().getName()+

": "+data);

}

}

}

# 传统线程技术回顾

线程就是程序的一条执行线索

## 创建线程的两种传统方式

1. 在Thread子类覆盖的run方法中编写运行代码

希望代码长期运行下去就编写在一个循环里面

涉及一个以往知识点：能否在run方法声明上抛出InterruptedException异常，以便省略run方法内部对Thread.sleep()语句的try…catch处理？

不行，子类不能抛出比父类更多的异常

2. 在传递给Thread对象的Runnable对象的run方法中编写代码

总结：查看Thread类的run()方法的源代码，可以看到其实这两种方式都是在调用Thread对象的run方法，如果Thread类的run方法没有被覆盖，并且为该Thread对象设置了一个Runnable对象，该run方法会调用Runnable对象的run方法。

问题：如果在Thread子类覆盖的run方法中编写了运行代码，也为Thread子类对象传递了一个Runnable对象，那么，线程运行时的执行代码是子类的run方法的代码？还是Runnable对象的run方法的代码？

子类的，因为子类已经把父类的run方法覆盖了，会以子类的run方法为准。为什么没有执行runnable对象？因为执行该对象的代码在父类中，现在父类Thread的代码被子类覆盖已经没有了，所以不可能执行父类中找runnable对象的方法

涉及到的一个以往知识点：匿名内部类对象的构造方法如何调用父类的非默认构造方法。

多线程机制会提高程序的运行效率吗？

不会，会更慢，因为CPU资源有限

为什么会有多线程下载呢？

是为了抢夺服务器带宽

# 传统定时器技术回顾

## Timer类与TimerTask类

void schedule(TimerTask task, long delay, long period); //使用相对时间，delay表示多长时间后执行任务

void schedule(TimerTask task, Date firstTime, long period); //使用绝对时间

启动定时器的代码，过10秒钟后启动定时器，然后每过1秒定时器执行一次：

1. new Timer().schedule(new TimerTask(){
2. public void run() {
3. System.out.println(Thread.currentThread().getName());
4. }
5. },
6. 10000, //过十秒启动定时器
7. 1000); //执行周期

代码实例：

1. package ThreadDemo;
2. import java.util.\*;
3. public class TimerDemo {
4. public static void main(String[] args) {
5. //5秒之后启动定时器并执行代码
6. new Timer().schedule(new TimerTask(){
7. @Override
8. public void run() {
9. System.out.println("bombing!");
10. }
12. }, 5000);
14. //while循环按打印当前秒数，每过一秒打印一次。
15. while(true){
16. System.out.println(new Date().getSeconds());
17. try {
18. Thread.sleep(1000);
19. } catch (InterruptedException e) {
20. e.printStackTrace();
21. }
22. }
23. }
24. }

## 定时器嵌套应用

需求：两秒启动一个炸弹，然后四秒启动一个炸弹，交替运行

方法一：静态全局变量

1. package ThreadDemo;
2. import java.util.\*;
3. public class TimerDemo {
4. //定义一个全局静态变量（不能定义在内部类里）
5. private static int count = 0;
6. public static void main(String[] args) {
7. //自定义一个内部类继承TimerTask
8. class MyTimerTask extends TimerTask{
9. @Override
10. public void run() {
11. count = (count+1)%2;//让count在0和1之间交替
12. System.out.println("bombing!");
13. new Timer().schedule(new MyTimerTask(),2000+2000\*count);//多长时间后执行是一个动态值
14. }
15. }
16. new Timer().schedule(new MyTimerTask(),2000);
18. while(true){
19. System.out.println(new Date().getSeconds());
20. try {
21. Thread.sleep(1000);
22. } catch (InterruptedException e) {
23. e.printStackTrace();
24. }
25. }
26. }
27. }

方法二：两个自定义定时器交替运行

1. package ThreadDemo;
2. import java.util.\*;
3. //自定义一个类继承TimerTask，内部调用另一个自定义的TimerTask对象
4. class MyTimerTask extends TimerTask{
5. @Override
6. public void run() {
7. System.out.println("bombing!");
8. new Timer().schedule(new MyTimerTask2(),2000);//多长时间后执行是一个动态值
9. }
10. }
11. //自定义一个类继承TimerTask，内部调用另一个自定义的TimerTask对象
12. class MyTimerTask2 extends TimerTask{
13. @Override
14. public void run() {
15. System.out.println("bombing!");
16. new Timer().schedule(new MyTimerTask(),4000);
17. }
18. }
19. public class TimerDemo {
20. public static void main(String[] args) {
21. new Timer().schedule(new MyTimerTask(),4000);
22. while(true){
23. System.out.println(new Date().getSeconds());
24. try {
25. Thread.sleep(1000);
26. } catch (InterruptedException e) {
27. e.printStackTrace();
28. }
29. }
30. }
31. }

需求：每天凌晨三点发邮件（需要使用绝对时间）

需求：工作日周一至周五发邮件，周末不发（使用开源工具Quartz）

# 传统线程互斥技术

静态方法中不能直接创建内部类的实例对象，因为内部类可以访问外部类的成员变量，而静态方法存在时还没有外部类的对象存在。

线程安全问题可以用银行转账来解释使用synchronized代码块及其原理使用synchronized方法分析静态方法所使用的同步监视器对象是什么？是所属类的字节码对象

不论是同步代码块还是同步方法（包括静态同步方法），只要他们使用的锁是同一个对象就可以实现互斥，即同步

# synchronized的三种应用方式

synchronized关键字最主要有以下3种应用方式，下面分别介绍

* 修饰实例方法，作用于当前实例加锁，进入同步代码前要获得当前实例的锁
* 修饰静态方法，作用于当前类对象加锁，进入同步代码前要获得当前类对象的锁
* 修饰代码块，指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。

# 传统线程同步通信技术

wait与notify实现线程间的通信

面试题：子线程循环10次，接着主线程循环100，接着又回到子线程循环10次，接着再回到主线程又循环100，如此循环50次，请写出程序。

最初写出来的代码如下：

1. package ThreadDemo;
2. public class ThreadTest {
3. private static boolean bShouldMain = false;//这里相当于定义了控制该谁执行的一个信号灯
4. public static void main(String[] args) {
5. new Thread(
6. new Runnable(){
7. public void run() {
8. for(int i=1;i<=50;i++){
9. //这里使用类的字节码对象作为锁进行同步，但是当需要对同步进行分组时就不科学了
10. synchronized(ThreadTest.class){
11. if(bShouldMain){
12. try {
13. ThreadTest.class.wait();
14. } catch (InterruptedException e) {
15. e.printStackTrace();
16. }
17. }
18. for(int j=1;j<=10;j++){
19. System.out.println("sub thread sequence of "+j+", loop of "+i);
20. }
21. bShouldMain = true;
22. ThreadTest.class.notify();
23. }
24. }
25. }
26. }
27. ).start();
29. //main方法本身是一个线程，这里是主线程的运行代码
30. for(int i=1;i<=50;i++){
31. synchronized(ThreadTest.class){
32. if(!bShouldMain){
33. try {
34. ThreadTest.class.wait();
35. } catch (InterruptedException e) {
36. e.printStackTrace();
37. }
38. }
39. for(int j=1;j<=100;j++){
40. System.out.println("main thread sequence of "+j+", loop of "+i);
41. }
42. bShouldMain = false;
43. ThreadTest.class.notify();
44. }
45. }
46. }
47. }

问题在于两个线程的代码要参照同一个变量，即bShouldMain，即这两个线程的代码要共享数据，所以，把这两个线程的执行代码搬到同一个类中去（注意循环50次不属于各自的业务执行代码）：

1. package ThreadDemo;
2. public class ThreadTest {
3. private static boolean bShouldMain = false;//这里相当于定义了控制该谁执行的一个信号灯
4. public static void main(String[] args) {
5. //因为用到了匿名内部类，内部类访问的局部变量需要用final修饰
6. final Business business = new Business();
7. new Thread(
8. new Runnable(){
9. public void run() {
10. for(int i=1;i<=50;i++){
11. //这里使用类的字节码对象作为锁进行同步，但是当需要对同步进行分组时就不科学了
12. business.sub(i);
13. }
14. }
15. }
16. ).start();
18. //main方法本身是一个线程，这里是主线程的运行代码
19. for(int i=1;i<=50;i++){
20. business.main(i);
21. }
22. }
23. }
25. class Business {
26. private boolean bShouldSub = true;//最开始该子线程走
27. public synchronized void sub(int i){
28. while(!bShouldSub){//用while而不是if线程醒来还会再次进行判断，防止代码被伪唤醒，代码更健壮。还可以防止生产者消费者问题
29. try {
30. this.wait();
31. } catch (InterruptedException e) {
32. e.printStackTrace();
33. }
34. }
35. for(int j=1;j<=10;j++){
36. System.out.println("sub thread sequence of "+j+", loop of "+i);
37. }
38. bShouldSub = false;
39. this.notify();
40. }
41. public synchronized void main(int i){
42. while(bShouldSub){
43. try {
44. this.wait();//这里的锁是this
45. } catch (InterruptedException e) {
46. e.printStackTrace();
47. }
48. }
49. for(int j=1;j<=100;j++){
50. System.out.println("main thread sequence of "+j+", loop of "+i);
51. }
52. bShouldSub = true;
53. this.notify();
54. }
55. }

经验：

1. 要用到共同数据（包括同步锁）或共同算法的若干方法应该归在同一个类身上，这种设计正好体现了高类聚和程序的健壮性
2. 锁是上在要操作的资源的类的内部方法中，而不是线程代码中！好处是以后该类交给任何线程自然就同步了，而不需要考虑互斥同步的问题。
3. Eclipse中将运行结果保存至文件的操作：Run as-->Run Configuration-->Common-->File处打钩然后选择一个文件

# 线程范围内的共享数据

## 线程范围内共享变量的概念与作用

线程范围内共享数据的示意图

全局变量会被所有的线程都共享，现在需要实现同一个线程内不同模块间变量的共享

关于线程范围内的变量共享的举例，直接用程序代码进行时说明，创建两个线程，它们都访问了两个模块，两个模块都取值，同一个线程设置的值，只能被相同的线程获取。

1. package cn.itcast.heima;
3. import java.util.HashMap;
4. import java.util.Map;
5. import java.util.Random;
7. public class ThreadScopeShareData {
8. private static Map<Thread,Integer> threadData = new HashMap<Thread,Integer>();//定义一个Map,为每一个线程存放独立数据
9. public static void main(String[] args) {
10. //使用循环创建两个线程
11. for(int i=0;i<2;i++){
12. new Thread(new Runnable(){
13. public void run() {
14. int data = new Random().nextInt();//线程产生一个数据
15. System.out.println(Thread.currentThread().getName()+" has put data: "+data);
16. threadData.put(Thread.currentThread(),data);
17. new A().get();//A模块取出数据
18. new B().get();//B模块取出数据
19. }
20. }).start();
21. }
22. }
24. //模块A
25. static class A{
26. public void get(){
27. int data = threadData.get(Thread.currentThread());//取出当前线程中变量的值
28. System.out.println("A from "+Thread.currentThread().getName()+" has put data: "+data);
29. }
30. }
32. //模块B
33. static class B{
34. public void get(){
35. int data = threadData.get(Thread.currentThread());
36. System.out.println("B from "+Thread.currentThread().getName()+" has put data: "+data);
37. }
38. }
40. }

运行结果：

应用：账户的转入转出。在同一个线程中有一个转入模块和一个转出模块，如果刚刚把钱转入时程序崩溃，转出模块还没有执行，就需要撤销之前的转入操作。并且只能提交己方线程的转出请求，而不能提交其他线程的转出请求。

## ThreadLocal实现线程范围的共享变量

ThreadLocal类就相当于一个Map，用于实现线程内的数据共享，即对于相同的程序代码，多个模块在同一个线程中运行时要共享一份数据，而在另外线程中运行时又共享另外一份数据。

每个线程调用全局ThreadLocal对象的set方法，就相当于往其内部的map中增加一条记录，key分别是各自的线程，value是各自的set方法传进去的值。在线程结束时可以调用ThreadLocal.clear()方法，这样会更快释放内存，不调用也可以，因为线程结束后也可以自动释放相关的ThreadLocal变量。

怎样得到线程结束的通知呢，或者是监听线程死亡的事件？比如监听虚拟机退出，Runtime类代表虚拟机，其addShutdownHook(Thread hook)方法会在虚拟机停止前运行传入线程的代码。那么要得到线程结束的通知，也会用到同样的思想。

注：api中包名以com.sun打头的是属于底层不被程序员调用的  
ThreadLocal的应用场景：

* 订单处理包含一系列操作：减少库存量、增加一条流水台账、修改总账，这几个操作要在同一个事务中完成，通常也即同一个线程中进行处理，如果累加公司应收款的操作失败了，则应该把前面的操作回滚，否则，提交所有操作，这要求这些操作使用相同的数据库连接对象，而这些操作的代码分别位于不同的模块类中。
* 银行转账包含一系列操作： 把转出帐户的余额减少，把转入帐户的余额增加，这两个操作要在同一个事务中完成，它们必须使用相同的数据库连接对象，转入和转出操作的代码分别是两个不同的帐户对象的方法。
* 例如Strut2的ActionContext，同一段代码被不同的线程调用运行时，该代码操作的数据是每个线程各自的状态和数据，对于不同的线程来说，getContext方法拿到的对象都不相同，对同一个线程来说，不管调用getContext方法多少次和在哪个模块中getContext方法，拿到的都是同一个。

实验案例：定义一个全局共享的ThreadLocal变量，然后启动多个线程向该ThreadLocal变量中存储一个随机值，接着各个线程调用另外其他多个类的方法，这多个类的方法中读取这个ThreadLocal变量的值，就可以看到多个类在同一个线程中共享同一份数据。  
实现对ThreadLocal变量的封装，让外界不要直接操作ThreadLocal变量。

* 对基本类型的数据的封装，这种应用相对很少见。
* 对对象类型的数据的封装，比较常见，即让某个类针对不同线程分别创建一个独立的实例对象。

总结：一个ThreadLocal代表一个变量，故其中里只能放一个数据，你有两个变量都要线程范围内共享，则要定义两个ThreadLocal对象。如果有一个百个变量要线程共享呢？那请先定义一个实体对象来装这一百个变量，然后在ThreadLocal中存储这一个对象。比如定义一个学生实体，存放姓名，年龄等变量。

如何设计线程范围内的共享对象？

* 第一种不优雅的实现方式：
  1. package cn.itcast.heima;
  2. import java.util.Random;
  3. public class ThreadLocalTest {
  5. //定义一个ThreadLocal变量存储线程内共享变量
  6. private static ThreadLocal<Integer> x = new ThreadLocal<Integer>();
  7. //定义一个ThreadLocal变量存储线程内的实体对象
  8. private static ThreadLocal<MyThreadScopeData> myThreadScopeData = new ThreadLocal<MyThreadScopeData>();
  9. public static void main(String[] args) {
  10. for(int i=0;i<2;i++){
  11. new Thread(new Runnable(){
  12. public void run() {
  13. int data = new Random().nextInt();
  14. System.out.println(Thread.currentThread().getName()+" has put data: "+data);
  15. x.set(data);//存入数据，并且数据已与当前线程关联
  17. //创建实体对象，并向实体对象中存入数据
  18. MyThreadScopeData myData = new MyThreadScopeData();
  19. myData.setName("name"+data);
  20. myData.setAge(data);
  21. //将实体存入ThreadLocal变量中
  22. myThreadScopeData.set(myData);
  24. new A().get();
  25. new B().get();
  26. }
  27. }).start();
  28. }
  29. }
  31. //模块A
  32. static class A{
  33. public void get(){
  34. int data = x.get();//不用指定线程号就取出当前线程中变量的值
  35. System.out.println("A from "+Thread.currentThread().getName()+" has put data: "+data);
  37. //获取线程内存入的实体对象并获取对象中存储的数据
  38. MyThreadScopeData myData = myThreadScopeData.get();
  39. System.out.println("A from "+Thread.currentThread().getName()+" getMyData: "+myData.getName()+","+myData.getAge());
  40. }
  41. }
  43. //模块B
  44. static class B{
  45. public void get(){
  46. int data = x.get();
  47. System.out.println("B from "+Thread.currentThread().getName()+" has put data: "+data);
  48. }
  49. }
  51. }
  53. //定义一个实体
  54. class MyThreadScopeData{
  55. private String name;
  56. private int age;
  57. public String getName() {
  58. return name;
  59. }
  60. public void setName(String name) {
  61. this.name = name;
  62. }
  63. public int getAge() {
  64. return age;
  65. }
  66. public void setAge(int age) {
  67. this.age = age;
  68. }
  69. }
* 第二种优雅的实现方式：
  1. package cn.itcast.heima;
  2. import java.util.Random;
  3. public class ThreadLocalTest {
  5. //定义一个ThreadLocal变量存储线程内共享变量
  6. private static ThreadLocal<Integer> x = new ThreadLocal<Integer>();
  8. public static void main(String[] args) {
  9. for(int i=0;i<2;i++){
  10. new Thread(new Runnable(){
  11. public void run() {
  12. int data = new Random().nextInt();
  13. System.out.println(Thread.currentThread().getName()+" has put data: "+data);
  14. x.set(data);//存入数据，并且数据已与当前线程关联
  16. /\*
  17. //创建实体对象，并向实体对象中存入数据
  18. MyThreadScopeData myData = new MyThreadScopeData();
  19. myData.setName("name"+data);
  20. myData.setAge(data);
  21. //将实体存入ThreadLocal变量中
  22. myThreadScopeData.set(myData);\*/
  24. //拿到与本线程相关的实例对象，然后向实体对象中存入数据
  25. MyThreadScopeData.getThreadInstance().setName("name"+data);;
  26. MyThreadScopeData.getThreadInstance().setAge(data);;
  27. new A().get();
  28. new B().get();
  29. }
  30. }).start();
  31. }
  32. }
  34. //模块A
  35. static class A{
  36. public void get(){
  37. int data = x.get();//不用指定线程号就取出当前线程中变量的值
  38. System.out.println("A from "+Thread.currentThread().getName()+" has put data: "+data);
  40. /\*//获取线程内存入的实体对象并获取对象中存储的数据
  41. MyThreadScopeData myData = myThreadScopeData.get();
  42. System.out.println("A from "+Thread.currentThread().getName()+" getMyData: "+myData.getName()+","+myData.getAge());
  43. \*/
  44. //获取本线程相关的实体对象并获取对象中存储的数据
  45. MyThreadScopeData myData = MyThreadScopeData.getThreadInstance();
  46. System.out.println("A from "+Thread.currentThread().getName()+" getMyData: "+myData.getName()+","+myData.getAge());
  48. }
  49. }
  51. //模块B
  52. static class B{
  53. public void get(){
  54. int data = x.get();
  55. System.out.println("B from "+Thread.currentThread().getName()+" has put data: "+data);
  57. MyThreadScopeData myData = MyThreadScopeData.getThreadInstance();
  58. System.out.println("B from "+Thread.currentThread().getName()+" getMyData: "+myData.getName()+","+myData.getAge());
  60. }
  61. }
  63. }
  65. //定义一个实体
  66. class MyThreadScopeData{
  68. //将该类设计成类似单例模式
  69. private MyThreadScopeData(){}
  70. private static ThreadLocal<MyThreadScopeData> map = new ThreadLocal<MyThreadScopeData>();
  71. public static MyThreadScopeData getThreadInstance(){//这里不需要同步，因为ThreadLocal变量与当前线程相关，各个线程会获取各自的实例
  72. MyThreadScopeData instance = map.get();
  73. if(instance==null){//如果实例没有，就创建实例并保存
  74. instance = new MyThreadScopeData();//保证返回的变量不为null
  75. map.set(instance);//存入的实体与当前线程相关，所以该方法不需要加同步
  76. }
  77. return instance;
  78. }
  80. private String name;
  81. private int age;
  82. public String getName() {
  83. return name;
  84. }
  85. public void setName(String name) {
  86. this.name = name;
  87. }
  88. public int getAge() {
  89. return age;
  90. }
  91. public void setAge(int age) {
  92. this.age = age;
  93. }
  94. }
  95. 优点：
      + 与前一种实现方式相比，这种方式向调用者隐藏ThreadLocal变量
      + 调用者只需要调用实体类中的方法就可以获得与当前线程相关的实例对象了
      + 单例是在任意地方都只能获取同一个对象，这里是在线程内调用获取的是同一个对象

# 多个线程访问共享对象和数据的方式

如果每个线程执行的代码相同，可以使用同一个Runnable对象，这个Runnable对象中有那个共享数据，例如，买票系统就可以这么做。

**package** cn.zz.test1;

**public** **class** **MultiThreadShareData** {

**private** **static** **ShareData1** *data1* = **new** ShareData1();

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

**ShareData1** **data1** = **new** ShareData1();

**new** Thread(data1).start();

**new** Thread(data1).start();

}

}

**class** **ShareData1** **implements** Runnable {

**private** **int** count = 100;

***@Override***

**public** **void** **run**() {

**// TODO Auto-generated method stub**

**while** (**true**) {

count--;

**System**.***out***.println(**Thread**.*currentThread*().getName()+" "+count);

}

}

}  
如果每个线程执行的代码不同，这时候需要用不同的Runnable对象，有如下两种方式来实现这些Runnable对象之间的数据共享：

* 将共享数据传递给Runnable实现类：将共享数据封装在另外一个对象中，然后将这个对象通过Runnable接口实现类的构造方法逐一传递给各个Runnable对象。每个线程对共享数据的操作方法也分配到那个对象身上去完成，这样容易实现针对该数据进行的各个操作的互斥和通信。
* **package** cn.zz.test1;
* **public** **class** **MultiThreadShareData2** {
* **public** **static** **void** **main**(**String**[] args) {
* **ShareData** **shareData** = **new** ShareData();
* **new** Thread(**new** Runnable() {
* ***@Override***
* **public** **void** **run**() {
* **while**(**true**){
* **try** {
* **Thread**.*sleep*(1000);
* } **catch** (**InterruptedException** **e**) {
* **// TODO Auto-generated catch block**
* e.printStackTrace();
* }
* shareData.inc();
* }
* }
* }).start();
* **new** Thread(**new** Runnable() {
* ***@Override***
* **public** **void** **run**() {
* **while**(**true**){
* **try** {
* **Thread**.*sleep*(1000);
* } **catch** (**InterruptedException** **e**) {
* **// TODO Auto-generated catch block**
* e.printStackTrace();
* }
* shareData.des();
* }
* }
* }).start();
* }
* }
* **class** **ShareData** {
* **private** **int** j = 10;
* **public** **synchronized** **void** **inc**() {
* j++;
* **System**.***out***.println(j);
* }
* **public** **synchronized** **void** **des**() {
* j--;
* **System**.***out***.println(j);
* }

}

* 让Runnable实现类去访问共享数据：将这些Runnable对象作为某一个类中的内部类，共享数据作为这个外部类中的成员变量，每个线程对共享数据的操作方法也分配给外部类，以便实现对共享数据进行的各个操作的互斥和通信，作为内部类的各个Runnable对象调用外部类的这些方法。也可以将成员变量作为局部变量加上final。
* 上面两种方式的组合：将共享数据封装在另外一个对象中，每个线程对共享数据的操作方法也分配到那个对象身上去完成，对象作为这个外部类中的成员变量或方法中的局部变量，每个线程的Runnable对象作为外部类中的成员内部类或局部内部类。
* 总之，要同步互斥的几段代码最好是分别放在几个独立的方法中，这些方法再放在同一个类中，这样比较容易实现它们之间的同步互斥和通信。

极端且简单的方式，即在任意一个类中定义一个static的变量，这将被所有线程共享。

面试题示例：

设计4个线程，其中两个线程每次对j增加1，另外两个线程对j每次减少1。写出程序。

方式一（上述第三种方式）：

1. public class MultiThreadShareData {
2. private static ShareData1 data1 = new ShareData1();
3. public static void main(String[] args) {
4. new Thread(new Runnable(){
5. public void run() {
6. while(true){
7. data1.decrement();
8. }
9. }
11. }).start();
12. new Thread(new Runnable(){
13. public void run() {
14. while(true){
15. data1.increment();
16. }
17. }
19. }).start();
20. }
21. }
23. //定义共享数据类，谁拥有数据，就提供操作该数据的方法
24. class ShareData1 implements Runnable{
25. private int j=0;
26. public synchronized void increment(){
27. j++;
28. }
29. public synchronized void decrement(){
30. j--;
31. }
32. public void run() {
34. }
35. }

方式二（上述第二种方式）：

1. public class ThreadTest1
2. {
3. private int j;
4. public static void main(String args[]){
5. ThreadTest1 tt=new ThreadTest1();
6. Inc inc=tt.new Inc();
7. Dec dec=tt.new Dec();
8. for(int i=0;i<2;i++){
9. Thread t=new Thread(inc);
10. t.start();
11. t=new Thread(dec);
12. t.start();
13. }
14. }
16. private synchronized void inc(){
17. j++;
18. System.out.println(Thread.currentThread().getName()+"-inc:"+j);
19. }
20. private synchronized void dec(){
21. j--;
22. System.out.println(Thread.currentThread().getName()+"-dec:"+j);
23. }
25. class Inc implements Runnable{
26. public void run(){
27. for(int i=0;i<100;i++){
28. inc();
29. }
30. }
31. }
33. class Dec implements Runnable{
34. public void run(){
35. for(int i=0;i<100;i++){
36. dec();
37. }
38. }
39. }
40. }

# Java5中的线程并发库

看java.util.concurrent包及子包的API帮助文档

* 看concurrent包的帮助文档页面，对并发库中涉及的内容有一个总体上的介绍：在并发编程中很常用的实用工具类。
* 如何看包的API帮助文档：可以先找到该包下的某个类的帮助页面，然后在该页面的顶部单击package超链接。

java.util.concurrent.atomic包

* 查看atomic包文档页下面的介绍：类的小工具包，支持在单个变量上解除锁的线程安全编程。
* 可以对基本数据，对数组中的基本数据，对类中的基本数据等进行操作
* 通过如下两个方法快速理解atomic包的意义：
  + AtomicInteger类的boolean compareAndSet(expectedValue, updateValue);
  + AtomicIntegerArray类的int addAndGet(int i, int delta);
* 顺带解释volatile类型的作用，需要查看java语言规范。Volatile的意思是说：在jvm中，一个线程更新了共享变量i，另外一个线程立即去读取共享区中的i时，读到的可能不是刚才另外那个线程更新过的结果，这就类似数据库中的事务隔离级别中的read uncommited，volatile就是解决这个问题的。

了解java.util.concurrent.lock包

* 在下面通过案例详细讲解

## 原子性操作类的应用

对类中的基本数据进行操作

### AtomicInteger类

java.util.concurrent.atomic包下的AtomicInteger类可以解决**多线程访问同一个整数**的问题

首先通过构造函数AtomicInteger(int initialValue)创建一个给定值的原子整数

然后比如调用对象上的addAndGet(int delta)方法返回对象中的数和delta的和（传入负数就是相减），该方法的调用过程中别的线程无法参与进来，因此自动实现了线程同步。还有其他的方法比如decrementAndGet()和incrementAndGet()

一般用于定义被多线程访问的成员变量而不是局部变量，因为局部变量会在每个线程中都有一份

类似的有AtomicBoolean、AtomicLong类操作不同的数据

### AtomicIntegerArray类

java.util.concurrent.atomic包下的AtomicIntegerArray类用于解决**多线程操作数组中的一个整数**的问题

addAndGet(int i, int delta) 以原子方式将给定值与索引 i 的元素相加。

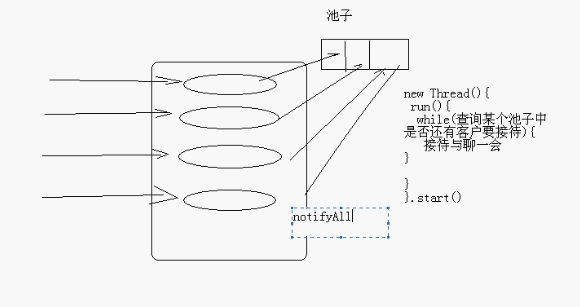
### AtomicIntegerFieldUpdater类

解决**多线程操作类的对象中存储的整数**的问题

static <U> AtomicIntegerFieldUpdater<U> **newUpdater**(Class<U> tclass, String fieldName) 使用给定字段为对象创建和返回一个更新器

int **addAndGet**(T obj, int delta) 以原子方式将给定值添加到此更新器管理的给定对象的字段当前值

## 线程池



线程池的概念

* 首先介绍在Tcp服务器编程模型的原理，每一个客户端连接用一个单独的线程为之服务，当与客户端的会话结束时，线程也就结束了，即每来一个客户端连接，服务器端就要创建一个新线程。
* 线程池首先创建一些线程，它们的集合称为线程池，当服务器接受到一个客户请求后，就从线程池中取出一个空闲的线程为之服务，服务完后不关闭该线程，而是将该线程还回到线程池中。
* 任务是提交给整个线程池，一个线程同时只能执行一个任务，但可以同时向一个线程池提交多个任务，池中的线程会各自同时执行一个任务，其他任务会排队等待
* 所有任务执行完后线程还存在，程序不会结束，需要手动关闭线程池
* 不需要和传统方式一样把任务交给特定的线程执行，而是把任务交给线程池，如果池中有空闲线程，就执行该任务，否则任务就等待被执行

Executors类的应用

线程池里的一个线程只会服务与当前的一个任务，当前任务结束，该线程才会服务其他线程

* 创建固定大小的线程池

执行任务的线程数固定

**package** cn.zz.pool;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**public** **class** **ThreadPoolGuDingDe** {

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

**//固定的线程池，三个线程**

ExecutorService **threadPool** = **Executors**.*newFixedThreadPool*(3);

**for**(**int** **i** =1;i<=10;i++){

**final** **int** **task** = i;

threadPool.execute(**new** Runnable() {

***@Override***

**public** **void** **run**() {

**try** {

**Thread**.*sleep*(20);

} **catch** (**InterruptedException** **e**) {

**// TODO Auto-generated catch block**

e.printStackTrace();

}

**for**(**int** **j**=1;j<=10;j++){

**System**.***out***.println(

**Thread**.*currentThread*().getName()+" "+

j+" task "+task);

}

}

});

}

**System**.***out***.println("十个任务全部提交");

**//所有的任务执行完了结束线程池**

threadPool.shutdown();

**System**.***out***.println("所有的任务执行完了结束线程池，这句话在任务结束前已经打印");

}

}

* 创建缓存线程池//线程数可随需求变化

**public** **class** **ThreadPoolCachepool** {

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

**//缓存线程池 根据任务的数量创建线程数，不确定有多少**

ExecutorService **threadPool** = **Executors**.*newCachedThreadPool*();

**for**(**int** **i** =1;i<=50;i++){

**final** **int** **task** = i;

threadPool.execute(**new** Runnable() {

***@Override***

**public** **void** **run**() {

**try** {

**Thread**.*sleep*(20);

} **catch** (**InterruptedException** **e**) {

**// TODO Auto-generated catch block**

e.printStackTrace();

}

**for**(**int** **j**=1;j<=10;j++){

**System**.***out***.println(

**Thread**.*currentThread*().getName()+" "+

j+" task "+task);

}

}

});

}

**System**.***out***.println("十个任务全部提交");

**//所有的任务执行完了结束线程池**

threadPool.shutdown();

**System**.***out***.println("所有的任务执行完了结束线程池，这句话在任务结束前已经打印");

}

}

* 创建单一线程池（实现线程死掉之后重新启动）

**public** **class** **ThreadPoolSinglePool** {

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

**//单线程池，只有一个线程在执行**

**//当一个线程死掉后，才会启动另一个线程**

ExecutorService **threadPool** = **Executors**.*newSingleThreadExecutor*();

**for**(**int** **i** =1;i<=50;i++){

**final** **int** **task** = i;

threadPool.execute(**new** Runnable() {

***@Override***

**public** **void** **run**() {

**try** {

**Thread**.*sleep*(20);

} **catch** (**InterruptedException** **e**) {

**// TODO Auto-generated catch block**

e.printStackTrace();

}

**for**(**int** **j**=1;j<=10;j++){

**System**.***out***.println(

**Thread**.*currentThread*().getName()+" "+

j+" task "+task);

}

}

});

}

**System**.***out***.println("十个任务全部提交");

**//所有的任务执行完了结束线程池**

threadPool.shutdown();

**System**.***out***.println("所有的任务执行完了结束线程池，这句话在任务结束前已经打印");

}

}

关闭线程池

* shutdown与shutdownNow的比较

用线程池启动定时器

* 调用ScheduledExecutorService的schedule方法，返回的ScheduleFuture对象可以取消任务。

**public** **class** **ThreadPoolSinglePoolScheduled** {

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

**//启动三个线程，只设置一个定时任务，那个线程有空那个线程执行，1秒后执行,之后每2秒执行一次**

**Executors**.*newScheduledThreadPool*(3).scheduleAtFixedRate(**new** Runnable() {

***@Override***

**public** **void** **run**() {

**System**.***out***.println("定时任务 "+**Thread**.*currentThread*().getName());

}

}, 1,2, *TimeUnit*.***SECONDS***);

}

}

* 支持间隔重复任务的定时方式（scheduleAtFixedRate）。

**public** **class** **ThreadPoolSinglePoolScheduledFix** {

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

**//设置一个定时任务，1秒后执行**

**Executors**.*newScheduledThreadPool*(1).schedule(**new** Runnable() {

***@Override***

**public** **void** **run**() {

**System**.***out***.println("定时任务1");

}

}, 1, *TimeUnit*.***SECONDS***);

}

}

* 所有的 schedule 方法都接受相对（相对现在） 延迟和周期作为参数，而不是绝对的时间或日期。将以 [Date](https://blog.csdn.net/joshchen0805/article/details/44776575) 所表示的绝对时间转换成要求的形式很容易。例如，要安排在某个以后的 Date 运行，可以使用：schedule(task, date.getTime() - System.currentTimeMillis(), TimeUnit.MILLISECONDS)。

1. package cn.itcast.heima;
3. import java.util.concurrent.ExecutorService;
4. import java.util.concurrent.Executors;
5. import java.util.concurrent.TimeUnit;
7. public class ThreadPool {
8. public static void main(String[] args) {
9. //创建一个固定大小的线程池，里面有3个线程
10. ExecutorService threadPool = Executors.newFixedThreadPool(3);
11. //创建一个缓存的线程池，池中线程不够，会自动创建新的线程
12. //ExecutorService threadPool = Executors.newCachedThreadPool();
13. //创建单一线程池，好处是始终保证池中有一个线程存在，如果线程死了，会再创建一个
14. //ExecutorService threadPool = Executors.newSingleThreadExecutor();
16. //通过循环向池中添加10个任务，但是同时只有3个任务被池中的3个线程执行，只有这3个任务都执行结束了才会执行其余任务
17. for(int i=1;i<=10;i++){
18. final int task = i;
19. threadPool.execute(new Runnable(){
20. public void run() {
21. //任务循环10遍，线程池中的某一个线程就会执行该任务
22. for(int j=1;j<=10;j++){
23. try {
24. Thread.sleep(20);
25. } catch (InterruptedException e) {
26. // TODO Auto-generated catch block
27. e.printStackTrace();
28. }
29. System.out.println(Thread.currentThread().getName()+" is looping of "+j+" for task "+task);
30. }
31. }
32. });
33. }
34. System.out.println("all of 10 tasks has committed!");
35. threadPool.shutdown();//关闭线程池（当所有的任务执行完毕，所有的线程都空闲下来时，结束池中的所有线程）
36. //threadPool.shutdownNow();//关闭线程池（立即结束池中的所有线程，即使还有任务没有执行完毕）
38. //创建一个调度线程池
39. Executors.newScheduledThreadPool(3).schedule(
40. new Runnable(){
41. public void run() {
42. System.out.println("bombing!");
43. }
44. },
45. 3,//delay
46. TimeUnit.SECONDS);
47. }
48. }

## Callable&Future

使用Callable<T>接口编写的线程中的任务会在线程运行结束会返回一个结果，该结果类型为Future<V>

Future通过get()方法取得返回结果，线程没有结束get方法会一直等待。

返回的结果类型和Callable返回的结果类型必须一致，这是通过泛型来实现的。  
Callable要采用ExecutorSevice的submit方法提交，而不是execute方法，因为execute方法没有返回值，返回的future对象可以取消任务。

**public** **class** **CallableAndFuture** {

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

ExecutorService **threadPool** = **Executors**.*newSingleThreadExecutor*();

Future<String> **future** =

threadPool.submit(

**new** Callable<String>() {

**public** **String** **call**() **throws** **Exception** {

**Thread**.*sleep*(2000);

**return** "hello";

};

}

);

**System**.***out***.println("等待结果");

**try** {

**//只有等到线程执行return后，这句话才会执行**

**System**.***out***.println("拿到结果：" + future.get());

} **catch** (**InterruptedException** **e**) {

**// TODO Auto-generated catch block**

e.printStackTrace();

} **catch** (**Exception** **e**) {

**// TODO Auto-generated catch block**

e.printStackTrace();

}

}

}  
CompletionService接口用于提交一组Callable任务，其take方法返回已完成的一个Callable任务对应的Future对象。

* 好比我同时种了几块地的麦子，然后就等待收割。收割时，则是哪块先成熟了，则先去收割哪块麦子。

应用：既然苦苦等待线程运行完毕返回的结果，还不如直接调用一个方法，运行完成就返回一个结果。

**public** **class** **CallableAndFutureCompletionService** {

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

ExecutorService **threadPool** = **Executors**.*newFixedThreadPool*(10);

CompletionService<Integer> **completionService** =

**new** ExecutorCompletionService<>(threadPool);

**for**(**int** **i**=1;i<=10;i++){

**final** **int** **seq** = i;

completionService.submit(**new** Callable<Integer>() {

***@Override***

**public** **Integer** **call**() **throws** **Exception** {

**Thread**.*sleep*(**new** Random().nextInt(5000));

**return** seq;

}

});

}

**for** (**int** **i**=0;i<10;i++){

**try** {

**System**.***out***.println(completionService.take().get());

} **catch** (**InterruptedException** | **ExecutionException** **e**) {

**// TODO Auto-generated catch block**

e.printStackTrace();

}

}

}

}

1. package cn.itcast.heima;
3. import java.util.Random;
4. import java.util.concurrent.Callable;
5. import java.util.concurrent.CompletionService;
6. import java.util.concurrent.ExecutionException;
7. import java.util.concurrent.ExecutorCompletionService;
8. import java.util.concurrent.ExecutorService;
9. import java.util.concurrent.Executors;
10. import java.util.concurrent.Future;
12. public class CallableAndFuture {
13. public static void main(String[] args) {
14. ExecutorService threadPool1 = Executors.newSingleThreadExecutor();
15. //使用Callable编写任务，Future接受返回结果
16. Future<String> future = threadPool1.submit(
17. new Callable<String>(){
18. public String call() throws Exception {
19. Thread.sleep(2000);
20. return "hello";
21. }
22. });
23. System.out.println("等待结果");
24. try {
25. System.out.println("拿到结果: "+future.get());
26. } catch (InterruptedException e) {
27. e.printStackTrace();
28. } catch (ExecutionException e) {
29. e.printStackTrace();
30. }
32. ExecutorService threadPool2 = Executors.newFixedThreadPool(10);
34. CompletionService<Integer> completionService = new ExecutorCompletionService<Integer>(threadPool2);
35. //向CompletionService中提交10个任务
36. for(int i=1;i<=10;i++){
37. final int sequence = i;//记录任务序号
38. completionService.submit(
39. new Callable<Integer>(){
40. public Integer call() throws Exception {
41. Thread.sleep(new Random().nextInt(5000));
42. return sequence;//返回的是当前任务的序号
43. }
44. });
45. }
46. //获取结果
47. for (int i = 0; i < 10; i++) {
48. try {
49. System.out.println(completionService.take().get());
50. } catch (InterruptedException e) {
51. e.printStackTrace();
52. } catch (ExecutionException e) {
53. e.printStackTrace();
54. }
55. }
56. }
57. }

## java5的线程锁技术Lock&Condition实现线程同步通信

### Lock

Lock比传统线程模型中的synchronized方式更加面向对象，与生活中的锁类似，锁本身也应该是一个对象。两个线程执行的代码片段要实现**同步互斥**的效果，它们必须用同一个Lock对象。锁是上在要操作的资源的类的内部方法中，而不是线程代码中！

**public** **class** **LockTest** {

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

**new** LockTest().init();

}

**private** **void** **init**(){

**final** **Outputer** **outputer** = **new** Outputer();

**new** Thread(**new** Runnable(){

***@Override***

**public** **void** **run**() {

**while**(**true**){

**try** {

**Thread**.*sleep*(10);

} **catch** (**InterruptedException** **e**) {

**// TODO Auto-generated catch block**

e.printStackTrace();

}

outputer.output("zhangxiaoxiang");

}

}

}).start();

**new** Thread(**new** Runnable(){

***@Override***

**public** **void** **run**() {

**while**(**true**){

**try** {

**Thread**.*sleep*(10);

} **catch** (**InterruptedException** **e**) {

**// TODO Auto-generated catch block**

e.printStackTrace();

}

outputer.output("lihuoming");

}

}

}).start();

}

**static** **class** **Outputer**{

**//lock必须写在这里，Outputer对象必须是同一个，当前lock才会生效**

Lock lock = **new** ReentrantLock();

**public** **void** **output**(**String** name){

**//写在这里相当于每次都创建一个新锁，不能到达互斥**

**//Lock lock = new ReentrantLock();**

**int** **len** = name.length();

lock.lock();

**try**{

**for**(**int** **i**=0;i<len;i++){

**System**.***out***.print(name.charAt(i));

}

**System**.***out***.println();

}**finally**{

lock.unlock();

}

}

**public** **synchronized** **void** **output2**(**String** name){

**int** **len** = name.length();

**for**(**int** **i**=0;i<len;i++){

**System**.***out***.print(name.charAt(i));

}

**System**.***out***.println();

}

**public** **static** **synchronized** **void** **output3**(**String** name){

**int** **len** = name.length();

**for**(**int** **i**=0;i<len;i++){

**System**.***out***.print(name.charAt(i));

}

**System**.***out***.println();

}

}

}

### 读写锁ReadWriteLock

分为读锁和写锁，多个读锁不互斥，读锁与写锁互斥，写锁与写锁互斥，这是由jvm自己控制的，你只要上好相应的锁即可。如果你的代码只读数据，可以很多人同时读，但不能同时写，那就上读锁；如果你的代码修改数据，只能有一个人在写，且不能同时读取，那就上写锁。读与读不互斥，读与写互斥，写与写也互斥。总之，读的时候上读锁，写的时候上写锁！

读写锁技能提高性能，又能实现互斥。如果只是使用Lock，那么读和读的线程之前也会互斥

代码示例：创建3个读线程，3个写线程：

**package** cn.zz.pool;

**import** java.util.Random;

**import** java.util.concurrent.locks.ReadWriteLock;

**import** java.util.concurrent.locks.ReentrantReadWriteLock;

**import** javax.swing.plaf.SliderUI;

**public** **class** **ReadWriteLockTest** {

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

**Queue** **queue** = **new** Queue();

**for**(**int** **i**=0;i<30;i++){

**new** Thread(**new** Runnable() {

***@Override***

**public** **void** **run**() {

**try** {

**Thread**.*sleep*(**new** Random().nextInt(1000));

} **catch** (**InterruptedException** **e**) {

**// TODO Auto-generated catch block**

e.printStackTrace();

}

queue.get();

}

}).start();

**new** Thread(**new** Runnable() {

***@Override***

**public** **void** **run**() {

**try** {

**Thread**.*sleep*(**new** Random().nextInt(1000));

} **catch** (**InterruptedException** **e**) {

**// TODO Auto-generated catch block**

e.printStackTrace();

}

queue.put(**new** Random().nextInt(5000));

}

}).start();

}

}

}

**class** **Queue**{

**private** **Object** data;

**private** ReadWriteLock lock = **new** ReentrantReadWriteLock();

**/\***

**\* 读数据的方法，加读锁**

**\*/**

**public** **void** **get**(){

lock.readLock().lock();

**try**{

**System**.***out***.println(**Thread**.*currentThread*().getName() + " be ready to read data!");

**Thread**.*sleep*((**long**)(**Math**.*random*()\*1000));

**System**.***out***.println(**Thread**.*currentThread*().getName() + "have read data :" + data);

}**catch**(**Exception** **e**){

e.printStackTrace();

}

**finally**{

lock.readLock().unlock();

}

}

**/\***

**\* 写数据方法，设置写锁**

**\*/**

**public** **void** **put**(**Object** data){

lock.writeLock().lock();

**try** {

**System**.***out***.println(**Thread**.*currentThread*().getName() + " be ready to write data!");

**Thread**.*sleep*((**long**)(**Math**.*random*()\*1000));

**this**.data = data;

**System**.***out***.println(**Thread**.*currentThread*().getName() + " have write data: " + data);

} **catch** (**InterruptedException** **e**) {

e.printStackTrace();

}**finally**{

lock.writeLock().unlock();

}

}

}

#### 缓存对象和缓存系统

在Hibernate中，获取数据库数据实体有两种方式：

1. User user = session.get(id,User.class);//如果数据对象不存在，返回null
2. User user = session.load(id,User.class);//如果数据对象不存在，则加载的是一个缓存代理对象
3. User$Proxy extends User{
4. private Integer id = id;
5. User realUser = null;
6. getName(){
7. if(realUser==null){
8. realUser = session.get(id);
9. if(realUser==null)
10. throw new Exception();
11. }
12. return realUser.getName();
13. }
14. }

API文档中对ReentrantReadWriteLock示例代码：

1. /\*
2. \* 示例用法。下面的代码展示了如何利用重入来执行升级缓存后的锁降级（为简单起见，省略了异常处理）：
3. \* 当有许多读的线程访问一个数据实体时，如果有一个线程发现该数据实体为空，该线程就需要将读锁释放，上写锁，写入数据
4. \* 然后在释放写锁前重新上读锁
5. \*/
6. class CachedData {
7. Object data;
8. volatile boolean cacheValid;//判断实体中是否有值
9. ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
11. void processCachedData() {
12. rwl.readLock().lock();
13. if (!cacheValid) {
14. // Must release read lock before acquiring write lock
15. rwl.readLock().unlock();
16. rwl.writeLock().lock();
17. // Recheck state because another thread might have acquired
18. // write lock and changed state before we did.
19. if (!cacheValid) {
20. data = ...//加载数据
21. cacheValid = true;
22. }
23. // Downgrade by acquiring read lock before releasing write lock
24. rwl.readLock().lock();
25. rwl.writeLock().unlock(); // Unlock write, still hold read
26. }
27. use(data);
28. rwl.readLock().unlock();
29. }
30. }

面试题：请设计一个**缓存系统**

以上是对单个对象进行缓存，缓存系统就是可以装很多个对象。

要拿对象别直接找数据库，而是访问缓存系统。当缓存系统的数据被访问时，就应该检查内部是否有该数据，如果有，就直接返回，如果没有就查询数据库，然后把数据存入内存中，下次就直接返回该数据不需要再查数据库了。

1. package cn.itcast.heima;
2. import java.util.HashMap;
3. import java.util.Map;
4. import java.util.concurrent.locks.ReadWriteLock;
5. import java.util.concurrent.locks.ReentrantReadWriteLock;
6. public class CacheDemo {
7. //定义一个Map，存储多个对象
8. private Map<String, Object> cache = new HashMap<String, Object>();
9. public static void main(String[] args) {}
10. private ReadWriteLock rwl = new ReentrantReadWriteLock();
11. //获取对象的方法，当多个线程来读（即获取对象时）不需要互斥，只有写的时候才需要互斥，因此需要使用读写锁
12. public Object getData(String key){
13. rwl.readLock().lock();//先上读锁，这样多个线程都可以来读
14. Object value = null;
15. try{
16. value = cache.get(key);
17. //如果线程读的时候发现数据为空，那么就把读锁释放，禁止数据的读取，然后上写锁
18. if(value == null){
19. rwl.readLock().unlock();
20. rwl.writeLock().lock();
21. try{
22. //再次判断数据是否存在，因为当一个线程获取写锁完成了数据填充并释放了写锁，另外一个线程也可能获取了写锁，会重复填充数据。
23. if(value==null){
24. value = "aaaa";//实际是去queryDB();即查询数据库
25. }
26. }finally{
27. rwl.writeLock().unlock();//数据填充完，释放写锁
28. }
29. rwl.readLock().lock();//恢复为读锁
30. }
31. }finally{
32. rwl.readLock().unlock();
33. }
34. return value;
35. }
36. }

### Condition

Lock和Condition的关系：Lock只能实现**互斥**（一个线程持有锁，另外的线程不能访问该锁），但是不能实现**通信**（即使获取了CPU的执行权，但是也可以让出执行权通知另外的线程执行），一个是哥们你不能干，一个是哥们你可以干。

Condition的功能类似于在传统线程计数中的Object.wait和Object.notify的功能。在等待 Condition 时，允许发生“虚假唤醒”，这通常作为对基础平台语义的让步。对于大多数应用程序，这带来的实际影响很小，因为 Condition 应该总是在一个循环中被等待，并测试正被等待的状态声明。某个实现可以随意移除可能的虚假唤醒，但建议应用程序程序员总是假定这些虚假唤醒可能发生，因此总是在一个循环中等待。  
Condition和 wait相似，那么为什么要用Condition 因为Condition允许多路等待

**package** cn.zz.pool;

**import** java.util.concurrent.locks.Condition;

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

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

/\*\*

\* 实现主线程和子线程依次执行，主线程执行完，通知子线程执行， 子线程执行完，通知主线程执行

\* **@author** Administrator

\*

\*/

**public** **class** ConditionCommunication {

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

BUS bus = **new** BUS();

**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

**for**(**int** i=1;i<=50;i++){

bus.sub(i);

}

}

}).start();

**for**(**int** i=1;i<=50;i++){

bus.main(i);

}

}

}

**class** BUS {

//设置一个锁

Lock lock = **new** ReentrantLock();

Condition condition = lock.newCondition();

**private** **boolean** bShouldSub = **false**;

//子线程逻辑

**public** **void** sub (**int** i) {

//lock保证互斥

lock.lock();

**try** {

//防止虚假唤醒

**while** (!bShouldSub) {

**try** {

condition.await();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**for**(**int** j=1;j<=10;j++){

System.out.println("sub thread sequence of " + j + ",loop of " + i);

}

bShouldSub = **false**;

condition.signal();

}**finally** {

lock.unlock();

}

}

//主线程逻辑

**public** **void** main(**int** i) {

lock.lock();

**try** {

**while**(bShouldSub) {

**try** {

condition.await();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**for**(**int** j=1;j<=10;j++){

System.out.println("main thread sequence of " + j + ",loop of " + i);

}

bShouldSub = **true**;

condition.signal();

}**finally** {

lock.unlock();

}

}

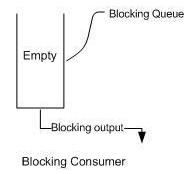
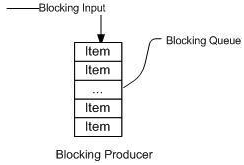
}

1. package cn.itcast.heima;
3. import java.util.concurrent.locks.Condition;
4. import java.util.concurrent.locks.Lock;
5. import java.util.concurrent.locks.ReentrantLock;
7. public class ConditionCommunication {
8. public static void main(String[] args) {
10. final Business business = new Business();
11. new Thread(
12. new Runnable() {
13. public void run() {
14. for(int i=1;i<=50;i++){
15. business.sub(i);
16. }
17. }
18. }
19. ).start();
21. for(int i=1;i<=50;i++){
22. business.main(i);
23. }
24. }
26. static class Business {//解决两个java文件中都有该类的问题：这里的完整类名是ConditionCommunicaiton.Business，另一个是Business
27. Lock lock = new ReentrantLock();
28. Condition condition = lock.newCondition();//获取该锁的condition对象
29. private boolean bShouldSub = true;
30. public void sub(int i){
31. lock.lock();
32. try{
33. while(!bShouldSub){//while循环防止虚假唤醒（线程被唤醒但是运行条件并不满足）
34. try {
35. condition.await();//调用condition对象特有的await方法，调用wait方法不报错，因为condition也是一个object，但是不是预期的方法。
36. } catch (Exception e) {
37. e.printStackTrace();
38. }
39. }
40. for(int j=1;j<=10;j++){
41. System.out.println("sub thread sequence of " + j + ",loop of " + i);
42. }
43. bShouldSub = false;
44. condition.signal();//调用condition对象特有的signal方法
45. }finally{
46. lock.unlock();
47. }
48. }
50. public void main(int i){
51. lock.lock();
52. try{
53. while(bShouldSub){
54. try {
55. condition.await();
56. } catch (Exception e) {
57. e.printStackTrace();
58. }
59. }
60. for(int j=1;j<=100;j++){
61. System.out.println("main thread sequence of " + j + ",loop of " + i);
62. }
63. bShouldSub = true;
64. condition.signal();
65. }finally{
66. lock.unlock();
67. }
68. }
69. }
70. }

## ****1.什么是阻塞队列****

阻塞队列常用于生产者和消费者的场景，生产者是往队列里添加元素的线程，消费者是从队列里拿元素的线程。阻塞队列就是生产者存放元素的容器，而消费者也只从容器里拿元素。

**BlockingQueue有两个常见阻塞场景**

1. 当队列中没有数据的情况下，消费者端的所有线程都会被自动阻塞（挂起），直到有数据放入队列。   
   
2. 当队列中填满数据的情况下，生产者端的所有线程都会被自动阻塞（挂起），直到队列中有空的位置，线程被自动唤醒。   
   

那么支持以上两种阻塞场景的队列我们称之为阻塞队列。

一个锁内部可以有多个Condition，即有**多路等待和通知**，可以参看jdk1.5提供的Lock与Condition实现的**可阻塞队列**的应用案例（见下面代码），从中除了要体味算法，还要体味面向对象的封装。在传统的线程机制中一个监视器对象上只能有一路等待和通知，要想实现多路等待和通知，必须嵌套使用多个同步监视器对象。（如果只用一个Condition，两个存放的线程都在等，一旦一个放的线程进去了，那么它通知**可能**会导致另一个放的线程接着往下走。）

可阻塞队列起到缓冲的效果，应用：使用该队列实现寻呼信息的存放和取出，当队列满时，存放线程等待，当队列空时，取出线程等待。

1. class BoundedBuffer {
2. final Lock lock = new ReentrantLock();
3. final Condition notFull = lock.newCondition();
4. final Condition notEmpty = lock.newCondition();
6. final Object[] items = new Object[100];
7. int putptr, takeptr, count;
9. public void put(Object x) throws InterruptedException {
10. lock.lock();
11. try {
12. while (count == items.length) //如果缓冲区中存入数据的数量等于缓冲区的长度，即缓冲区满了，那么存入线程就进入等待
13. notFull.await();
14. items[putptr] = x;
15. if (++putptr == items.length) putptr = 0;//将存的指针指向缓冲区第一个坐标
16. ++count;
17. notEmpty.signal();//只换醒取出线程
18. } finally {
19. lock.unlock();
20. }
21. }
23. public Object take() throws InterruptedException {
24. lock.lock();
25. try {
26. while (count == 0) //缓冲区中没有数据可取了，取出线程就进入等待
27. notEmpty.await();
28. Object x = items[takeptr];
29. if (++takeptr == items.length) takeptr = 0;//将取的指针指向第一个坐标
30. --count;
31. notFull.signal();//只换醒存入线程
32. return x;
33. } finally {
34. lock.unlock();
35. }
36. }
37. }

并法库包中的ArrayBlockingQueue 类提供了这项功能，因此没有理由去实现这个示例类

可以实现多个线程按顺序执行。下面代码实现主线程，子线程1，子线程2轮流交替运行：

**package** cn.zz.pool;

**import** java.util.concurrent.locks.Condition;

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

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

/\*\*

\* sub1唤醒sub2,sub2唤醒sub3,sub3唤醒sub1

\* 注意 sub1 sub2 sub3的线程数量必须相等

\* **@author** Administrator

\*

\*/

**public** **class** ThreeConditionCommunication2 {

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

Three three = **new** Three();

**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

**for**(**int** i=1;i<=50;i++){

three.sub1(i);

}

}

}).start();

**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

**for**(**int** i=1;i<=50;i++){

three.sub2(i);

}

}

}).start();

**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

**for**(**int** i=1;i<=50;i++){

three.sub3(i);

}

}

}).start();

}

}

**class** Three {

//设置一个锁

Lock lock = **new** ReentrantLock();

Condition condition1 = lock.newCondition();

Condition condition2 = lock.newCondition();

Condition condition3 = lock.newCondition();

**private** **int** subNum = 1;

//子线程逻辑

**public** **void** sub1 (**int** i) {

//lock保证互斥

lock.lock();

**try** {

//防止虚假唤醒

**while** (subNum != 1) {

**try** {

condition1.await();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**for**(**int** j=1;j<=10;j++){

System.out.println("subNum:"+subNum+" sub thread sequence of " + j + ",loop of " + i);

}

subNum = 2;

condition2.signal();

}**finally** {

lock.unlock();

}

}

**public** **void** sub2 (**int** i) {

//lock保证互斥

lock.lock();

**try** {

//防止虚假唤醒

**while** (subNum != 2) {

**try** {

condition2.await();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**for**(**int** j=1;j<=10;j++){

System.out.println("subNum:"+subNum+" sub thread sequence of " + j + ",loop of " + i);

}

subNum = 3;

condition3.signal();

}**finally** {

lock.unlock();

}

}

**public** **void** sub3 (**int** i) {

//lock保证互斥

lock.lock();

**try** {

//防止虚假唤醒

**while** (subNum != 3) {

**try** {

condition3.await();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**for**(**int** j=1;j<=10;j++){

System.out.println("subNum:"+subNum+" sub thread sequence of " + j + ",loop of " + i);

}

subNum = 1;

condition1.signal();

}**finally** {

lock.unlock();

}

}

}

## Semaphore实现信号灯

Semaphore（信号）可以控制当前访问资源的线程个数，并提供了同步机制。例如，实现一个文件允许的并发访问数。

* Semaphore实现的功能就类似厕所有5个坑，假如有十个人要上厕所，那么同时能有多少个人去上厕所呢？同时只能有5个人能够占用，当5个人中的任何一个人让开后，其中在等待的另外5个人中又有一个可以占用了。
* 同步锁是一个坑，现在是多个坑一起参与资源的管理
* 另外等待的5个人中可以是随机获得优先机会，也可以是按照先来后到的顺序获得机会，这取决于构造Semaphore对象时传入的参数选项：public Semaphore(int permits, boolean fair)，fair为 true则保证此信号量在争用时按先进先出的顺序授予许可

单个信号量的Semaphore对象可以实现互斥锁的功能，这时这个信号量就类似一个锁，并且可以是由一个线程获得了“锁”，再由另一个线程释放“锁”，这可应用于死锁恢复的一些场合（锁只能被本线程释放，不能被其他线程释放）。

**package** cn.zz.pool;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.Semaphore;

/\*\*

\* 信号灯

\* **@author** Administrator

\*

\*/

**public** **class** SemaphoreTest {

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

ExecutorService service = Executors.newCachedThreadPool();

**final** Semaphore sp = **new** Semaphore(3);

**for**(**int** i=0;i<10;i++) {

Runnable runnable = **new** Runnable() {

@Override

**public** **void** run() {

**try** {

sp.acquire();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

System.out.println("线程" + Thread.currentThread().getName() +

"进入，当前已有" + (3-sp.availablePermits()) + "个并发");

**try** {

Thread.sleep((**long**)(Math.random()\*10000));

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.out.println("线程" + Thread.currentThread().getName() +

"即将离开");

sp.release();

//下面代码有时候执行不准确，因为其没有和上面的代码合成原子单元

System.out.println("线程" + Thread.currentThread().getName() +

"已离开，当前已有" + (3-sp.availablePermits()) + "个并发");

}

};

service.execute(runnable);

}

service.shutdown();

}

}

管理停车位，一个小的电子设备，实时性强就要semaphore。

## 其他同步工具类

### CyclicBarrier

一个同步辅助类，它允许一组线程互相等待，直到到达某个公共屏障点 (common barrier point)。  
表示大家彼此等待，大家集合好后才开始出发，分散活动后又在指定地点集合碰面，这就好比整个公司的人员利用周末时间集体郊游一样，先各自从家出发到公司集合后，再同时出发到公园游玩，在指定地点集合后再同时开始就餐，…。

Cyclic：循环的，有周期性的，Barrier:障碍物，屏障。多个线程干完各自的任务，在不同的时刻到达集合点后，就可以接着忙各自的工作去了，再到达新的集合点，再去忙各自的工作，到达集合点了用CyclicBarrier对象的await方法表示。想在什么地方集合，就在什么地方调用await方法  
**package** cn.zz.CyclicBarrierZ;

**import** java.util.Random;

**import** java.util.concurrent.BrokenBarrierException;

**import** java.util.concurrent.CyclicBarrier;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.TimeUnit;

/\*\*

\* 玩家排队，每拍够5个，进入游戏

\* **@author** Administrator

\*

\*/

**public** **class** CyclicBarrierTest {

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

ExecutorService service = Executors.newFixedThreadPool(5);

**final** CyclicBarrier barrier = **new** CyclicBarrier(5);

//这里准备了32个线程,所以最终会有两个线程一直等待

**for** (**int** i = 0; i < 32; i++) {

service.execute(**new** Player("玩家" + i, barrier));

}

service.shutdown();

}

}

**class** Player **implements** Runnable {

**private** **final** String name;

**private** **final** CyclicBarrier barrier;

**public** Player(String name, CyclicBarrier barrier) {

**this**.name = name;

**this**.barrier = barrier;

}

**public** **void** run() {

**try** {

TimeUnit.SECONDS.sleep(1 + (**new** Random().nextInt(3)));

System.out.println(name + "已准备,等待其他玩家准备...");

barrier.await();

TimeUnit.SECONDS.sleep(1 + (**new** Random().nextInt(3)));

System.out.println(name + "已加入游戏");

} **catch** (InterruptedException e) {

System.out.println(name + "离开游戏");

} **catch** (BrokenBarrierException e) {

System.out.println(name + "离开游戏");

}

}

}

### CountDownLatch

Latch：门闩，闩锁  
犹如倒计时计数器，调用CountDownLatch对象的await方法让当前线程等待计数器到0，调用countDown方法就将计数器减1，当计数到达0时，则所有等待者或单个等待者开始执行。

可以实现一个人（也可以是多个人）等待其他所有人都来通知他，可以实现一个人通知多个人的效果，类似裁判一声口令，运动员同时开始奔跑，或者所有运动员都跑到终点后裁判才可以公布结果，用这个功能做百米赛跑的游戏程序不错哦！还可以实现一个计划需要多个领导都签字后才能继续向下实施的情况。

**package** cn.zz.CountdownLatchZ;

**import** java.util.concurrent.CountDownLatch;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**public** **class** CountdownLatchTest {

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

ExecutorService service = Executors.newCachedThreadPool();

CountDownLatch order = **new** CountDownLatch(1);

CountDownLatch answer = **new** CountDownLatch(3);

**for**(**int** i=0;i<3;i++) {

Runnable runnable = **new** Runnable() {

@Override

**public** **void** run() {

**try** {

System.out.println("线程" + Thread.currentThread().getName() +

"正准备接受命令");

//主线程发 order,子线程才开始执行

order.await();

System.out.println("线程" + Thread.currentThread().getName() +

"已接受命令");

Thread.sleep((**long**)(Math.random()\*10000));

System.out.println("线程" + Thread.currentThread().getName() +

"回应命令处理结果");

//三个线程都执行了answer计数减到0,主线程继续执行

answer.countDown();

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

};

service.execute(runnable);

}

service.shutdown();

**try** {

Thread.sleep((**long**)(Math.random()\*10000));

System.out.println("线程" + Thread.currentThread().getName() +

"即将发布命令");

order.countDown();

System.out.println("线程" + Thread.currentThread().getName() +

"已发送命令，正在等待结果");

answer.await();

System.out.println("线程" + Thread.currentThread().getName() +

"已收到所有响应结果");

} **catch** (Exception e) {

e.printStackTrace();

}

}

}

### Exchanger(用于交换数据)

用于实现两个人之间的数据交换，每个人在完成一定的事务后想与对方交换数据，第一个先拿出数据的人将一直等待第二个人拿着数据到来时，才能彼此交换数据。

好比两个毒贩要进行交易，一手交钱、一手交货，不管谁先来到接头地点后，就处于等待状态了，当另外一方也到达了接头地点（所谓到达接头地点，也就是到到达了准备接头的状态）时，两者的数据就立即交换了，然后就又可以各忙各的了。  
exchange方法就相当于两手高高举着待交换物，等待人家前来交换，一旦人家到来（即人家也执行到exchange方法），则两者立马完成数据的交换

**package** cn.zzExchangerZ;

**import** java.util.concurrent.Exchanger;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**public** **class** ExchangerTest {

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

ExecutorService service = Executors.newCachedThreadPool();

Exchanger exchanger = **new** Exchanger();

service.execute(**new** Runnable() {

@Override

**public** **void** run() {

**try** {

String data1 = "money";

System.out.println("线程" + Thread.currentThread().getName() +

"正在把数据" + data1 +"换出去");

Thread.sleep((**long**)(Math.random()\*10000));

String data2 = (String)exchanger.exchange(data1);

System.out.println("线程" + Thread.currentThread().getName() +

"换回的数据为" + data2);

}**catch** (Exception e) {

// **TODO**: handle exception

}

}

});

service.execute(**new** Runnable(){

**public** **void** run() {

**try** {

String data1 = "薯条";

System.out.println("线程" + Thread.currentThread().getName() +

"正在把数据" + data1 +"换出去");

Thread.sleep((**long**)(Math.random()\*10000));

String data2 = (String)exchanger.exchange(data1);

System.out.println("线程" + Thread.currentThread().getName() +

"换回的数据为" + data2);

}**catch**(Exception e){

}

}

});

service.shutdown();

}

}

## 可阻塞的队列BlockingQueue

队列：先进先出。包含固定长度队列和不固定长度队列

什么是可阻塞队列：队列满了添加线程会阻塞等待，队列空了获取线程也会阻塞等待。非阻塞队列则会直接报错

阻塞队列的作用与实际应用，阻塞队列的实现原理。  
阻塞队列与Semaphore有些相似，但也不同，阻塞队列是一方存放数据，另一方释放数据，Semaphore通常则是由同一方设置和释放信号量。  
BlockingQueue接口的子类提供了可阻塞队列功能，ArrayBlockingQueue类是固定大小的，LinkedBlockingQueue类可以是不固定大小的（数组是连续的一片内存，链表是不连续的内存）

在其接口类BlockingQueue中提供了3个插入方法。add抛异常，offer返回真假值，put方法阻塞。3个取的方法：remove抛出异常，poll返回null，take可阻塞

只有put方法和take方法才具有阻塞功能  
用3个空间的队列来演示阻塞队列的功能和效果。

**package** cn.zz.BlockingQueueZ;

**import** java.util.concurrent.ArrayBlockingQueue;

**import** java.util.concurrent.BlockingQueue;

**public** **class** BlockingQueueTest {

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

BlockingQueue queue = **new** ArrayBlockingQueue(3);

**for**(**int** i=0;i<2;i++) {

**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

**while**(**true**) {

**try** {

Thread.sleep((**long**)(Math.random()\*1000));

System.out.println(Thread.currentThread().getName() + "准备放数据!");

queue.put(1);

System.out.println(Thread.currentThread().getName() + "已经放了数据，" +

"队列目前有" + queue.size() + "个数据");

}**catch** (Exception e) {

e.printStackTrace();

}

}

}

}).start();

}

**new** Thread(){

**public** **void** run(){

**while**(**true**){

**try** {

//将此处的睡眠时间分别改为100和1000，观察运行结果

Thread.sleep(1000);

System.out.println(Thread.currentThread().getName() + "准备取数据!");

queue.take();

System.out.println(Thread.currentThread().getName() + "已经取走数据，" +

"队列目前有" + queue.size() + "个数据");

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}.start();

}

}

用两个具有1个空间的队列来实现同步通知的功能（你一下，我一下，轮流执行）。  
在前面用Condition实现的同步通知的例子的基础上，改为用阻塞队列来实现。  
第一个线程：A.take()……..B.put()  
第二个线程：B.take()……..A.put()

1. package cn.itcast.heima;
3. import java.util.Collections;
4. import java.util.concurrent.ArrayBlockingQueue;
5. import java.util.concurrent.BlockingQueue;
6. import java.util.concurrent.atomic.AtomicInteger;
8. public class BlockingQueueCommunication {
9. public static void main(String[] args) {
10. final Business business = new Business();
11. new Thread(
12. new Runnable() {
13. public void run() {
14. for(int i=1;i<=50;i++){
15. business.sub(i);
16. }
17. }
18. }
19. ).start();
21. for(int i=1;i<=50;i++){
22. business.main(i);
23. }
24. }
26. static class Business {
27. //产生两个阻塞队列
28. BlockingQueue<Integer> queue1 = new ArrayBlockingQueue<Integer>(1);//控制子线程
29. BlockingQueue<Integer> queue2 = new ArrayBlockingQueue<Integer>(1);//控制主线程
31. //构造代码块（或匿名构造方法），对所有对象进行初始化。不能用静态代码块，因为是对成员变量进行初始化
32. {
33. Collections.synchronizedMap(null);
34. try {
35. System.out.println("xxxxxdfsdsafdsa");
36. queue2.put(1);//一开始主线程不运行，就通过构造代码块在程序运行之前在队列2中放入数据
37. } catch (InterruptedException e) {
38. e.printStackTrace();
39. }
40. }
42. public void sub(int i){//这里如果加synchronized关键字那么会出现死锁
43. try {
44. queue1.put(1);//程序一执行让子线程先运行，就先在队列1中放入元素
45. } catch (InterruptedException e) {
46. e.printStackTrace();
47. }
48. for(int j=1;j<=10;j++){
49. System.out.println("sub thread sequece of " + j + ",loop of " + i);
50. }
51. try {
52. queue2.take();//子线程运行完毕取走队列2的元素让主线程运行
53. } catch (InterruptedException e) {
54. e.printStackTrace();
55. }
56. }
58. public void main(int i){
59. try {
60. queue2.put(1);//由于程序开始执行时队列2是满的所以会阻塞在这里
61. } catch (InterruptedException e1) {
62. e1.printStackTrace();
63. }
64. for(int j=1;j<=100;j++){
65. System.out.println("main thread sequece of " + j + ",loop of " + i);
66. }
67. try {
68. queue1.take();
69. } catch (InterruptedException e) {
70. e.printStackTrace();
71. }
72. }
73. }
74. }

## 同步集合Concurrent Collections

传统集合类在并发访问（多个线程访问）时会出现问题。比如死循环。

线程不安全

* 此时就需要在读的时候不能写，写的时候不能读，但可以并发地读。
* 如果不被多个线程并发访问，就使用传统集合
  1. /\*
  2. \* 在进行元素读取的时候不要进行其他操作，会出现死循环。
  3. \* 比如hasNext正好读到最后一个元素时，其他线程进来执行remove操作，hasNext内部cursor!=count，始终返回true，成为死循环
  4. \*/
  5. count=4;
  6. while(hasNext()){
  7. next(){cursor++}//next方法内部原理
  8. }
  9. //hasNext方法内部原理
  10. hasNext(){
  11. if(cursor==count)
  12. return false;
  13. return true;
  14. }
  15. remove(){
  16. count--;count=3;
  17. }

传统方式下用Collections工具类提供的synchronizedCollection方法来获得同步集合

* 比如Collections.synchronizedMap(new HashMap())，分析该方法的实现源码，可以到使用了代理类返回了一个同步集合，在该代理类中对被代理集合的方法都加上了synchronized关键字实现方法间的互斥
* 面试题---HashSet和HashMap的关系：HashSet的内部实现使用HashMap，只使用了HashMap的key部分，value部分不考虑（随便填写但是不使用），因为key不能重复，所以是一个Set

Java5中提供了如下一些同步集合类（通过看java.util.concurrent包下的介绍可以知道有哪些并发集合）：

* ConcurrentHashMap
* ConcurrentSkipListMap/Set：可以排序的同步集合，需要指定比较器
* CopyOnWriteArrayList
* CopyOnWriteArraySet

传统方式下的Collection在迭代集合时，不允许对集合进行修改。

* 用空中网面试的同步级线程题进行演示（见下）
* 根据AbstractList的checkForComodification方法的源码，分析产生ConcurrentModificationException异常的原因。

使用同步集合实现迭代时对集合进行修改：

1. package cn.itcast.heima;
2. import java.util.ArrayList;
3. import java.util.Collection;
4. import java.util.Iterator;
5. import java.util.concurrent.CopyOnWriteArrayList;
6. public class CollectionModifyExceptionTest {
7. public static void main(String[] args) {
8. Collection users = new CopyOnWriteArrayList();//在写的时候有一分拷贝
9. users.add(new User("张三",28));
10. users.add(new User("李四",25));
11. users.add(new User("王五",31));
12. Iterator itrUsers = users.iterator();
13. while(itrUsers.hasNext()){
14. System.out.println("aaaa");
15. User user = (User)itrUsers.next();
16. if("李四".equals(user.getName())){
17. users.remove(user);
18. //itrUsers.remove();
19. } else {
20. System.out.println(user);
21. }
22. }
23. }
24. }

## 空中网挑选实习生的面试题

### 第一题：

现有的程序代码模拟产生了16个日志对象，并且需要运行16秒才能打印完这些日志，请在程序中增加4个线程去调用parseLog()方法来分头打印这16个日志对象，程序只需要运行4秒即可打印完这些日志对象。原始代码如下：

1. package cn.itcast.heima;
2. public class Test {
3. public static void main(String[] args){
4. System.out.println("begin:"+(System.currentTimeMillis()/1000));
5. /\*模拟处理16行日志，下面的代码产生了16个日志对象，当前代码需要运行16秒才能打印完这些日志。
6. 修改程序代码，开四个线程让这16个对象在4秒钟打完。
7. \*/
8. for(int i=0;i<16;i++){ //这行代码不能改动
9. final String log = ""+(i+1);//这行代码不能改动
10. {
11. Test.parseLog(log);
12. }
13. }
14. }
16. //parseLog方法内部的代码不能改动
17. public static void parseLog(String log){
18. System.out.println(log+":"+(System.currentTimeMillis()/1000));
19. try {
20. Thread.sleep(1000);
21. } catch (InterruptedException e) {
22. e.printStackTrace();
23. }
24. }
25. }

分析：线程如何得到产生的数据？将数据存放在队列中，然后从队列中取

步骤：创建四个线程，将产生的数据存入集合，线程在集合中取数据

1. package cn.itcast.heima;
3. import java.util.concurrent.ArrayBlockingQueue;
4. import java.util.concurrent.BlockingQueue;
6. public class Test {
7. public static void main(String[] args){
8. final BlockingQueue<String> queue = new ArrayBlockingQueue<String>(1);//1个数据也可以，
9. //final BlockingQueue<String> queue = new ArrayBlockingQueue<String>(16);//可以装16个数据
10. //创建四个线程
11. for(int i=0;i<4;i++){
12. new Thread(new Runnable(){
13. @Override
14. public void run() {
15. while(true){
16. try {
17. String log = queue.take();//从阻塞队列中拿数据
18. parseLog(log);
19. } catch (InterruptedException e) {
20. e.printStackTrace();
21. }
22. }
23. }
24. }).start();
25. }
27. System.out.println("begin:"+(System.currentTimeMillis()/1000));
28. /\*模拟处理16行日志，下面的代码产生了16个日志对象，当前代码需要运行16秒才能打印完这些日志。
29. 修改程序代码，开四个线程让这16个对象在4秒钟打完。
30. \*/
31. for(int i=0;i<16;i++){ //这行代码不能改动
32. final String log = ""+(i+1);//这行代码不能改动
33. {
34. try {
35. queue.put(log);//将数据放入队列中
36. } catch (InterruptedException e) {
37. e.printStackTrace();
38. }
39. }
40. }
41. }
43. //parseLog方法内部的代码不能改动
44. public static void parseLog(String log){
45. System.out.println(log+":"+(System.currentTimeMillis()/1000));
47. try {
48. Thread.sleep(1000);
49. } catch (InterruptedException e) {
50. e.printStackTrace();
51. }
52. }
53. }

考察了阻塞队列的使用

### 第二题：

现成程序中的Test类中的代码在不断地产生数据，然后交给TestDo.doSome()方法去处理，就好像生产者在不断地产生数据，消费者在不断消费数据。请将程序改造成有10个线程来消费生成者产生的数据，这些消费者都调用TestDo.doSome()方法去进行处理，故每个消费者都需要一秒才能处理完，程序应保证这些消费者线程依次有序地消费数据，只有上一个消费者消费完后，下一个消费者才能消费数据，下一个消费者是谁都可以，但要保证这些消费者线程拿到的数据是有顺序的。原始代码如下：

1. package queue;
2. public class Test {
3. public static void main(String[] args) {
4. System.out.println("begin:"+(System.currentTimeMillis()/1000));
5. for(int i=0;i<10;i++){ //这行不能改动
6. String input = i+""; //这行不能改动
7. String output = TestDo.doSome(input);
8. System.out.println(Thread.currentThread().getName()+ ":" + output);
9. }
10. }
11. }
13. //不能改动此TestDo类
14. class TestDo {
15. public static String doSome(String input){
16. try {
17. Thread.sleep(1000);
18. } catch (InterruptedException e) {
19. e.printStackTrace();
20. }
21. String output = input + ":"+ (System.currentTimeMillis() / 1000);
22. return output;
23. }
24. }

分析：

线程如何得到产生的数据？通过队列

这里使用同步队列SynchronousQueue，它是一个阻塞队列，每一个插入操作对应一个取出操作，只有线程一旦来取了，才会插入数据。所以它不指定大小

使用lock或者semaphore实现线程互斥，一个一个执行

1. package queue;
3. import java.util.concurrent.Semaphore;
4. import java.util.concurrent.SynchronousQueue;
6. public class Test {
7. public static void main(String[] args) {
8. final Semaphore semaphore = new Semaphore(1);
9. final SynchronousQueue<String> queue = new SynchronousQueue<String>();
10. for(int i=0;i<10;i++){
11. new Thread(new Runnable(){
12. public void run() {
13. try {
14. semaphore.acquire();//当前线程获取一个信号灯，保证线程一个一个取，而不是10个线程同时取数据（这里用锁也可以）
15. String input = queue.take();
16. String output = TestDo.doSome(input);
17. System.out.println(Thread.currentThread().getName()+ ":" + output);
18. semaphore.release();
19. } catch (InterruptedException e) {
20. e.printStackTrace();
21. }
22. }
23. }).start();
24. }
26. System.out.println("begin:"+(System.currentTimeMillis()/1000));
27. for(int i=0;i<10;i++){ //这行不能改动
28. String input = i+""; //这行不能改动
29. try {
30. queue.put(input);//将数据放在队列中
31. } catch (InterruptedException e) {
32. e.printStackTrace();
33. }
34. }
35. }
36. }
38. //不能改动此TestDo类
39. class TestDo {
40. public static String doSome(String input){
41. try {
42. Thread.sleep(1000);
43. } catch (InterruptedException e) {
44. e.printStackTrace();
45. }
46. String output = input + ":"+ (System.currentTimeMillis() / 1000);
47. return output;
48. }
49. }

### 第三题：

现有程序同时启动了4个线程去调用TestDo.doSome(key, value)方法，由于TestDo.doSome(key, value)方法内的代码是先暂停1秒，然后再输出以秒为单位的当前时间值，所以，会打印出4个相同的时间值，如下所示：  
4:4:1258199615  
1:1:1258199615  
3:3:1258199615  
1:2:1258199615  
请修改代码，如果有几个线程调用TestDo.doSome(key, value)方法时，传递进去的key相等（equals比较为true），则这几个线程应互斥排队输出结果，即当有两个线程的key都是"1"时，它们中的一个要比另外其他线程晚1秒输出结果，如下所示：  
4:4:1258199615  
1:1:1258199615  
3:3:1258199615  
1:2:1258199616  
总之，当每个线程中指定的key相等时，这些相等key的线程应每隔一秒依次输出时间值（要用互斥），如果key不同，则并行执行（相互之间不互斥）。原始代码如下：

1. package syn;
3. //不能改动此Test类
4. public class Test extends Thread{
6. private TestDo testDo;
7. private String key;
8. private String value;
10. public Test(String key,String key2,String value){
11. this.testDo = TestDo.getInstance();
12. /\*常量"1"和"1"是同一个对象，下面这行代码就是要用"1"+""的方式产生新的对象，
13. 以实现内容没有改变，仍然相等（都还为"1"），但对象却不再是同一个的效果\*/
14. this.key = key+key2;
15. this.value = value;
16. }
18. public static void main(String[] args) throws InterruptedException{
19. Test a = new Test("1","","1");
20. Test b = new Test("1","","2");
21. Test c = new Test("3","","3");
22. Test d = new Test("4","","4");
23. System.out.println("begin:"+(System.currentTimeMillis()/1000));
24. a.start();
25. b.start();
26. c.start();
27. d.start();
29. }
31. public void run(){
32. testDo.doSome(key, value);
33. }
34. }
36. class TestDo {
38. private TestDo() {}
39. private static TestDo \_instance = new TestDo();
40. public static TestDo getInstance() {
41. return \_instance;
42. }
44. public void doSome(Object key, String value) {
46. // 以大括号内的是需要局部同步的代码，不能改动!
47. {
48. try {
49. Thread.sleep(1000);
50. System.out.println(key+":"+value + ":"
51. + (System.currentTimeMillis() / 1000));
52. } catch (InterruptedException e) {
53. e.printStackTrace();
54. }
55. }
56. }
57. }

分析：

需要同步，同步的锁就是传入的key，但是锁字符串内容相同，但不一定是同一对象，所以需要进行判断。

需要用到同步集合。

1. package syn;
3. import java.util.ArrayList;
4. import java.util.Iterator;
5. import java.util.concurrent.CopyOnWriteArrayList;
7. //不能改动此Test类
8. public class Test extends Thread{
10. private TestDo testDo;
11. private String key;
12. private String value;
14. public Test(String key,String key2,String value){
15. this.testDo = TestDo.getInstance();
16. /\*常量"1"和"1"是同一个对象，下面这行代码就是要用"1"+""的方式产生新的对象，
17. 以实现内容没有改变，仍然相等（都还为"1"），但对象却不再是同一个的效果\*/
18. this.key = key+key2;
19. /\* a = "1"+"";
20. b = "1"+"";//这里a和b都是常量相加，编译器会自动优化为同一对象
21. \*/
22. this.value = value;
23. }
25. public static void main(String[] args) throws InterruptedException{
26. Test a = new Test("1","","1");
27. Test b = new Test("1","","2");
28. Test c = new Test("3","","3");
29. Test d = new Test("4","","4");
30. System.out.println("begin:"+(System.currentTimeMillis()/1000));
31. a.start();
32. b.start();
33. c.start();
34. d.start();
36. }
38. public void run(){
39. testDo.doSome(key, value);
40. }
41. }
43. class TestDo {
45. private TestDo() {}
46. private static TestDo \_instance = new TestDo();
47. public static TestDo getInstance() {
48. return \_instance;
49. }
51. // private ArrayList keys = new ArrayList();//这里不能使用ArrayList集合。因为会发生ConcurrentModificationException的问题，此例中会出现一个线程迭代集合的同时另一个线程在往集合中添加元素。比如此例中a线程进入往集合中添加了一个元素，b线程进入因为元素相同就进行迭代操作，这时c线程也是同时执行的，进入后会执行添加操作。出现该错误必须是这种情况出现，否则程序运行不会报错！！
52. private CopyOnWriteArrayList keys = new CopyOnWriteArrayList();
53. public void doSome(Object key, String value) {
54. Object o = key;
55. if(!keys.contains(o)){
56. keys.add(o);//如果集合中没有该元素（实际上判断有无元素与之相等），就存入该对象
57. }else{
58. //如果有该元素，就直接遍历集合，找到该元素，并用该元素作为同步的锁
59. for(Iterator iter=keys.iterator();iter.hasNext();){
60. try {
61. Thread.sleep(20);//减慢迭代的速度（等着别的线程进行加入元素操作），增加出现并发修改问题的概率
62. } catch (InterruptedException e) {
63. e.printStackTrace();
64. }
65. Object oo = iter.next();
66. if(oo.equals(o)){
67. o = oo;
68. break;
69. }
70. }
71. }
72. synchronized(o)
73. // 以大括号内的是需要局部同步的代码，不能改动!
74. {
75. try {
76. Thread.sleep(1000);
77. System.out.println(key+":"+value + ":"
78. + (System.currentTimeMillis()/1000));//这里将毫秒数除以1000，所以看到的时间会一样
79. } catch (InterruptedException e) {
80. e.printStackTrace();
81. }
82. }
83. }