学习目标

- 能够解释安全问题的出现的原因
  多线程访问了同一个共享的数据
- 能够使用同步代码块解决线程安全问题(必须会)
  ```
  synchronized(锁对象){
      可能出现安全问题的代码
      (访问了共享数据的代码)
  }
  ```
  注意:保证锁对象要唯一(可以是任意对象)
- 能够使用同步方法解决线程安全问题(必须会)
  1.把访问共享数据的代码,提取出来放在一个方法中
  2.在方法上增加一个同步关键字

  ```
  修饰符 synchronized 返回值类型 方法名(参数列表){
      出现了安全问题的代码
      (使用了共享数据的代码)
  }
  ```
- 能够说明volatile关键字和synchronized关键字的区别
  volatile只能修饰：变量。只能解决：可见性、有序性。不能解决原子性
  synchronized不能修饰变量，可以修饰：代码块、方法。范围要比volatile广阔。可以解决：可见性、有序性、原子性。
- 能够理解原子类的工作机制
  CAS机制(比较并交换)：(do...while)反复的比较，只有比较跟预期是一样，才会执行修改。
- 能够掌握原子类AtomicInteger的使用(必须会)
  作用:线程安全的,多个线程访问同一个整数,保证原子性
  ```
  AtomicInteger num = new AtomicInteger(0);//不写默认值0
  for(int i = 0 ; i < 10000; i++){
      num.getAndIncrement();//自增1
  }
  ```
- 能够描述ConcurrentHashMap类的作用(必须会)
  作用：线程安全的，而且比Hashtable效率高。
- 能够描述CountDownLatch类的作用
  作用：倒计时。一个线程先执行一部分操作，然后等待其它线程工作完毕，然后第一个线程再继续工作。
- 能够描述CyclicBarrier类的作用
  作用：计数器。一个线程等待其它的几个线程全部执行完毕，然后再执行
- 能够表述Semaphore类的作用
  作用：控制并发数量。可以允许几个线程同时进入执行。
- 能够描述Exchanger类的作用
  作用：两个线程的信息交换。

第一章 原子性

概述：所谓的原子性是指在一次操作或者多次操作中，要么所有的操作全部都得到了执行并且不会受到任何因素的干扰而中断，要么所有的操作都不执行，

多个操作是一个不可以分割的整体(原子)。

比如：从张三的账户给李四的账户转1000元，这个动作将包含两个基本的操作：从张三的账户扣除1000元，给李四的账户增加1000元。这两个操作必须符合原子性的要求，

要么都成功要么都失败。

1.原子类概述

概述：java从JDK1.5开始提供了java.util.concurrent.atomic包(简称Atomic包)，这个包中的原子操作类提供了一种用法简单，性能高效，线程安全地更新一个变量的方式。

1). java.util.concurrent.atomic.AtomicInteger：对int变量进行原子操作的类。

2). java.util.concurrent.atomic.AtomicLong：对long变量进行原子操作的类。

3). java.util.concurrent.atomic.AtomicBoolean：对boolean变量进行原子操作的类。

这些类可以保证对"某种类型的变量"原子操作，多线程、高并发的环境下，就可以保证对变量访问的有序性，从而保证最终的结果是正确的。

AtomicInteger原子型Integer，可以实现原子更新操作

```
构造方法:
public AtomicInteger():                    初始化一个默认值为0的原子型Integer
public AtomicInteger(int initialValue):    初始化一个指定值的原子型Integer
成员方法:
int get():                                 获取值
int getAndIncrement():                     以原子方式将当前值加1，注意，这里返回的是自增前的值。
int incrementAndGet():                     以原子方式将当前值加1，注意，这里返回的是自增后的值。
int addAndGet(int data):                   以原子方式将输入的数值与实例中的值（AtomicInteger里的value）相加，并返回结果。
int getAndSet(int value):                  以原子方式设置为newValue的值，并返回旧值。
```

2.AtomicInteger类的基本使用

```
package com.itheima.demo01AtomicInteger;

import java.util.concurrent.atomic.AtomicInteger;

/*
     java.util.concurrent.atomic.AtomicInteger: 对int变量进行原子操作的类
        构造方法:
            public AtomicInteger():                    初始化一个默认值为0的原子型Integer
            public AtomicInteger(int initialValue):    初始化一个指定值的原子型Integer
        成员方法:
            int get():                                 获取值
            int getAndIncrement():                     以原子方式将当前值加1，注意，这里返回的是自增前的值。
            int incrementAndGet():                     以原子方式将当前值加1，注意，这里返回的是自增后的值。
            int addAndGet(int data):                   以原子方式将输入的数值与实例中的值（AtomicInteger里的value）相加，并返回结果。
            int getAndSet(int value):                  以原子方式设置为newValue的值，并返回旧值。
```

```
 */
public class Demo01AtomicInteger {
    public static void main(String[] args) {
        //创建AtomicInteger对象,初始值为10
        AtomicInteger ai = new AtomicInteger(10);

        //int get():  获取值
        int i = ai.get();
        System.out.println(i);//10

        //int getAndIncrement():  以原子方式将当前值加1，注意，这里返回的是自增前的值。
        int i2 = ai.getAndIncrement();  //i++;
        System.out.println("i2:"+i2);//i2:10
        System.out.println(ai);//11

        //int incrementAndGet(): 以原子方式将当前值加1，注意，这里返回的是自增后的值。
        int i3 = ai.incrementAndGet();// ++i
        System.out.println("i3:"+i3);//i3:12
        System.out.println(ai);//12

        //int addAndGet(int data):以原子方式将输入的数值与实例中的值（AtomicInteger里的value）
相加，并返回结果。
        int i4 = ai.addAndGet(100);// 12+100
        System.out.println("i4:"+i4);//i4:112
        System.out.println(ai);//112

        //int getAndSet(int value): 以原子方式设置为newValue的值，并返回旧值。
        int i5 = ai.getAndSet(88);
        System.out.println("i5:"+i5);//i5:112 被替换的值
        System.out.println(ai);//88
    }
}
```

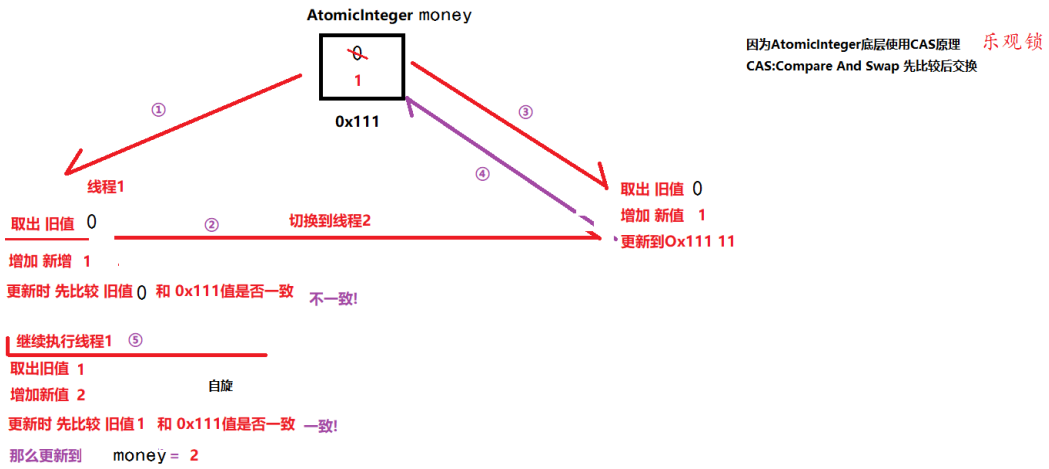
3.AtomicInteger解决变量的可见性、有序性、原子性(重点)

```
package demo07AtomicInteger;

import java.util.concurrent.atomic.AtomicInteger;

public class MyThread extends Thread{
    //不要直接使用int类型的变量,使用AtomicInteger
    public static AtomicInteger money = new AtomicInteger(0);

    @Override
    public void run() {
        System.out.println("Thread-0线程开始改变money的值!");
        for (int i = 0; i < 10000; i++) {
            //money.getAndIncrement();//先获取后自增  money++
            money.incrementAndGet();//先自增后获取  ++money
        }
```

```
            System.out.println("Thread-0线程执行完毕!");
        }
    }
```

```
package demo07AtomicInteger;

public class Demo01Thread {
    public static void main(String[] args) throws InterruptedException {
        //开启新的线程
        MyThread mt  = new MyThread();
        mt.start();

        System.out.println("main继续往下执行");
        System.out.println("main线程开始改变money的值!");
        for (int i = 0; i < 10000; i++) {
            MyThread.money.incrementAndGet();
        }
        System.out.println("main线程暂停1秒,让Thread-0执行完毕!");
        Thread.sleep(1000);
        System.out.println("最终money为:"+ MyThread.money);
    }
}
```

3.AtomicInteger的原理_CAS机制（乐观锁）



3.AtomicIntegerArray(了解)

可以用原子方式更新其元素的 `int` 数组:可以保证数组的原子性

构造方法:

AtomicIntegerArray(int length) 创建指定长度的给定长度的新 AtomicIntegerArray。
AtomicIntegerArray(int[] array) 创建与给定数组具有相同长度的新 AtomicIntegerArray，并从给定数组复制其所有元素。

成员方法:

int addAndGet(int i, int delta) 以原子方式将给定值与索引 i 的元素相加。
i:获取指定索引处的元素
delta:给元素增加的值

int get(int i) 获取指定索引处元素的值

没有使用AtomicIntegerArray数组存在原子性的问题

```
package com.itheima.demo06AtomicIntegerArray;

public class MyThread extends Thread{
    //定义一个静态所有线程共享的数组
    public static int[] arr = new int[1000];

    @Override
    public void run() {
        //遍历数组,改变数组中变量的值0-->1
        for (int i = 0; i < arr.length; i++) {
            arr[i]++;//arr[i]=arr[i]+1;
        }
    }
}
```

```
package com.itheima.demo06AtomicIntegerArray;

import java.util.Arrays;

public class Demo01AtomicIntegerArray {
    public static void main(String[] args) {
        //创建1000个线程,每个线程都把数组中的元素的值加1==>0-->1000 最终的结果
[1000,1000,1000,...]
        for (int i = 0; i < 1000; i++) {
            new MyThread().start();
        }

        //让主线程睡眠5秒钟,等待1000个线程执行完毕
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("main线程睡眠结束,遍历数组,获取数组中的元素");
        System.out.println(Arrays.toString(MyThread.arr));
```

```
    }
}
```

执行结果:

```
[1000, 1000, 999, 999, 999, 999, 999, 999, 999, 999, 999, 999, 999, 999, 999, 999, 999,
1000, 1000, 1000, 1000, 1000, 1000, 1000,]
```

使用AtomicIntegerArray数组解决数组的原子性问题

```
package com.itheima.demo07AtomicIntegerArray;

import java.util.concurrent.atomic.AtomicIntegerArray;

public class MyThread extends Thread{
    //定义一个静态所有线程共享的数组,不要使用基本数据类型的数组了,使用AtomicIntegerArray数组
    public static int[] arr = new int[1000];
    public static AtomicIntegerArray integerArray = new AtomicIntegerArray(arr);

    @Override
    public void run() {
        //遍历数组,改变数组中变量的值0-->1
        for (int i = 0; i < 1000; i++) {
            //把指定索引处的元素,加1
            integerArray.addAndGet(i,1);
        }
    }
}
```

```
package com.itheima.demo07AtomicIntegerArray;

import java.util.Arrays;

public class Demo01AtomicIntegerArray {
    public static void main(String[] args) {
        //创建1000个线程,每个线程都把数组中的元素的值加1==>0-->1000 最终的结果
[1000,1000,1000,...]
        for (int i = 0; i < 1000; i++) {
            new MyThread().start();
        }

        //让主线程睡眠5秒钟,等待1000个线程执行完毕
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("main线程睡眠结束,遍历数组,获取数组中的元素");
        for (int i = 0; i < MyThread.integerArray.length(); i++) {
            //获取指定索引处的元素
```

```
                System.out.println(MyThread.integerArray.get(i));
            }
        }
    }
```

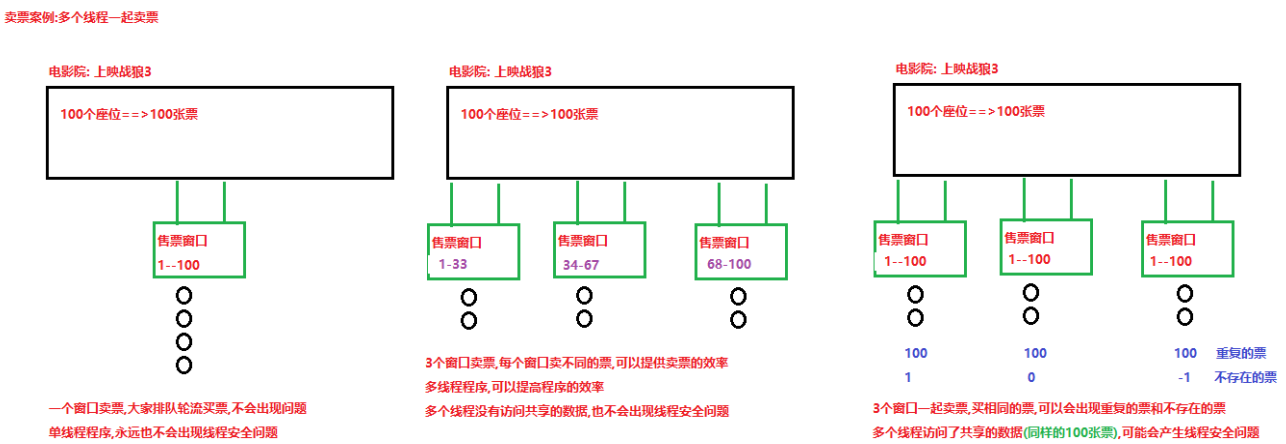
第二章 线程安全

之前我们讲过的AtomicInteger可以对“int类型的变量”做原子操作。但如果需要将“很多行代码”一起作为“原子性”执行——一个线程进入后，必须将所有代码行执行完毕，其它线程才能进入，可以使用synchronized关键字——重量级的同步关键字。

AtomicInteger:只能解决一个变量的原子性

synchronized:可以解决一段代码的原子性

1.线程安全问题的概述(了解)



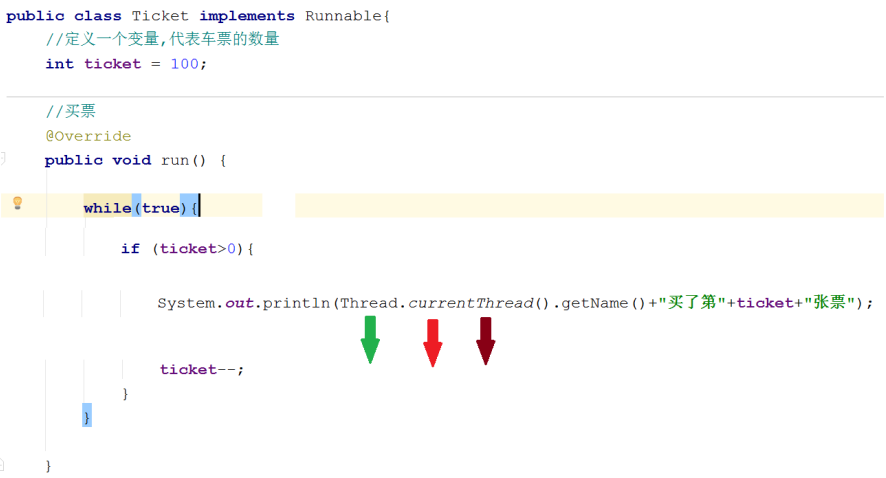
2.线程安全问题的代码实现(重点)

```
package com.itheima.demo10payTicket;

/*
    卖票案例
 */
public class RunnableImpl implements Runnable{
    //定义共享的票源
    private int ticket = 100;

    //线程任务:卖票
    @Override
    public void run() {
        //让卖票重复执行
        while (true){
            //票大于0,就进行卖票
```

```
            if(ticket>0){
                //让线程睡眠10毫秒,提供安全问题出现的几率
                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread().getName()+"正在卖第"+ticket+"张
票!");
                ticket--;
            }
        }
    }
}
```

```
package com.itheima.demo10payTicket;

/*
    创建3个线程进行买相同的100张票
 */
public class Demo01PayTticket {
    public static void main(String[] args) {
        RunnableImpl r = new RunnableImpl();
        Thread t0 = new Thread(r);
        Thread t1 = new Thread(r);
        Thread t2 = new Thread(r);
        t0.start();
        t1.start();
        t2.start();
    }
}
```

3.线程安全问题的产生原理(了解)



4.解决线程安全问题的第一种方式使用同步代码块(重点)

```
/*
   1.同步代码
   2.格式:
      synchronize代码块
      synchronize(任意对象){
         代码
      }
   3.进入代码块->拿到锁,其他线程就拿不到锁,不能运行,等着
     出了代码块->释放锁,只有还锁,其他线程才有资格拿到锁去执行线程任务
   4.注意:想要实现线程同步,锁对象就必须是同一把锁
 */
public class MyTicket1 implements Runnable{
    //有100张票
    int ticket = 100;

   Object obj = new Object();

    @Override
    public void run() {
        while(true){
            synchronized (obj){
                try {
                    Thread.sleep(100L);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                if (ticket>0){
                    System.out.println(Thread.currentThread().getName()+"买到了
第"+ticket+"张票");//买完了
                    ticket--;
                }
            }

        }

    }
}
```

```java
public class MyMain {
    public static void main(String[] args) {
        //创建线程类对象
        MyTicket1 myTicket = new MyTicket1();

        Thread thread1 = new Thread(myTicket,"刘能");
        Thread thread2 = new Thread(myTicket,"赵四");
        Thread thread3 = new Thread(myTicket,"广坤");
        thread1.start();
        thread2.start();
        thread3.start();
    }
}
```

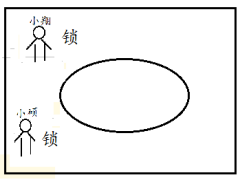
5.同步的原理(了解)

```java
public class Ticket implements Runnable{
    //定义一个变量,代表车票的数量
    int ticket = 100;

    //买票
    @Override
    public void run() {
        while(true){
            锁
            if (ticket>0){

                System.out.println(Thread.currentThread().getName()+"买了第"+ticket+"张票");

                ticket--;
            }
        }
    }
```

100人拿100个橘子
1. 排队拿安全,还是抢安全
   排队
2. 排队拿效率高,还是抢效率高
   抢

结论:
线程安全的,效率低
线程不安全,效率高

StringBuilder
线程不安全　　拼接字符串效率高



把资源上锁,一个线程拿到锁去访问资源,其他线程等待,什么时候锁释放了,其他的线程才有可能去访问资源,但是第一个线程拿到锁执行完毕,第一个线程还有可能继续拿锁去访问资源->线程同步问题
问题:线程同步,需要锁
但是需要几把锁呢?
1把

```java
//买票
@Override
public void run() {
    //循环买票
    while(true){
        //同步代码块
        synchronized (obj){
            //票大于0,就买票
            if (ticket>0){
                try {
                    Thread.sleep( millis: 100L);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread().getName()+"买了第"+ticket+"张票");
                ticket--;
            }
        }

    }
```

6.解决线程安全问题的第二种方式:使用同步方法(重点)

```java
/*
```

```
  同步方法:
    1.格式:
       public synchronize 返回值类型 方法名(参数){
          可能会产生线程安全问题的代码
       }
    2.默认的锁->this
 */
public class MyTicket1 implements Runnable{
    //有100张票
    int ticket = 100;


    @Override
    public void run() {
        while(true){
           //调用同步方法
            method();
        }

    }

   /* public synchronized void method(){
        try {
            Thread.sleep(100L);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        if (ticket>0){
            System.out.println(Thread.currentThread().getName()+"买到了第"+ticket+"张
票");//买完了
            ticket--;
        }
    }*/
   public void method(){
       synchronized (this){
           try {
               Thread.sleep(100L);
           } catch (InterruptedException e) {
               e.printStackTrace();
           }
           if (ticket>0){
               System.out.println(Thread.currentThread().getName()+"买到了第"+ticket+"张
票");//买完了
               ticket--;
           }
       }
   }
}
```

```
public class MyMain {
    public static void main(String[] args) {
        //创建线程类对象
        MyTicket1 myTicket = new MyTicket1();
```

```
        Thread thread1 = new Thread(myTicket,"刘能");
        Thread thread2 = new Thread(myTicket,"赵四");
        Thread thread3 = new Thread(myTicket,"广坤");
        thread1.start();
        thread2.start();
        thread3.start();
    }
}
=======================================================
```

```
/*
  静态同步方法:
    1.格式:
       public static synchronize 返回值类型 方法名(参数){
          可能会产生线程安全问题的代码
       }
    2.默认的锁->(类名.class)
 */
public class MyTicket1 implements Runnable{
    //有100张票
   static int ticket = 100;


    @Override
    public void run() {
        while(true){
           //调用同步方法
            method();
        }

    }

   /* public static synchronized void method(){
        try {
            Thread.sleep(100L);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        if (ticket>0){
            System.out.println(Thread.currentThread().getName()+"买到了第"+ticket+"张
票");//买完了
            ticket--;
        }
    }*/
   public static void method(){
       synchronized (MyTicket1.class){
           try {
               Thread.sleep(10L);
           } catch (InterruptedException e) {
               e.printStackTrace();
           }
           if (ticket>0){
```

```
                System.out.println(Thread.currentThread().getName()+"买到了第"+ticket+"张
票");//买完了
                ticket--;
            }
        }
    }
}
```

```
public class MyMain {
    public static void main(String[] args) {
        //创建线程类对象
        MyTicket1 myTicket = new MyTicket1();

        Thread thread1 = new Thread(myTicket,"刘能");
        Thread thread2 = new Thread(myTicket,"赵四");
        Thread thread3 = new Thread(myTicket,"广坤");
        thread1.start();
        thread2.start();
        thread3.start();
    }
}
```

7.解决线程安全问题的第三方式:使用Lock锁(了解)

```
/*
   Lock:
     1.概述:接口
     2.使用:通过实现类去使用-->ReentranLock
     3.方法:
         void lock() ->获取锁
         void unlock() ->释放锁
 */
public class MyTicket01 implements Runnable{
    int ticket = 100;
    //获取Lock对象
    Lock lock = new ReentrantLock();

    @Override
    public void run() {
        while(true){
            //获取锁
            lock.lock();
            try {
                Thread.sleep(100L);
                if (ticket>0){
                    System.out.println(Thread.currentThread().getName()+"买到了
第"+ticket+"张票");
                    ticket--;
                }
            } catch (InterruptedException e) {
```

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

```
public class MyMain {
    public static void main(String[] args) {
        //创建实现类对象
        MyTicket01 myTicket01 = new MyTicket01();
        Thread thread1 = new Thread(myTicket01, "刘能");
        Thread thread2 = new Thread(myTicket01, "赵四");
        Thread thread3 = new Thread(myTicket01, "广坤");
        //开启线程
        thread1.start();
        thread2.start();
        thread3.start();
    }
}
```

8.CAS与Synchronized

CAS和Synchronized都可以保证多线程环境下共享数据的安全性。那么他们两者有什么区别?

Synchronized是从悲观的角度出发:

总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁

（**共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程**）。因此Synchronized我们也将其称之为悲观锁。jdk中的ReentrantLock也是一种悲观锁。

CAS是从乐观的角度出发:

总是假设最好的情况，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据。

CAS这种机制我们也可以将其称之为乐观锁。

第三章 并发包

在JDK的并发包java.util.concurrent里提供了几个非常有用的并发容器和并发工具类。供我们在多线程开发中进行使用。这些集合和工具类都可以保证高并发的线程安全问题.

1.并发List集合_CopyOnWriteArrayList(重点)

1).java.util.concurrent.CopyOnWriteArrayList(类)：它是一个“线程安全”的ArrayList，我们之前学习的java.utils.ArrayList不是线程安全的。 2).如果是多个线程，并发访问同一个ArrayList，我们要使用：CopyOnWriteArrayList

```
public class Test01 {
    public static void main(String[] args) throws InterruptedException {
        ArrayList<Integer> list = new ArrayList<>();
        //CopyOnWriteArrayList<Integer> list = new CopyOnWriteArrayList<>();

        //加1000个
        Runnable r1 = new Runnable() {
            @Override
            public void run() {
                for(int i = 0 ; i < 1000;i++){
                    list.add(i);
                }
            }
        };
        new Thread(r1).start();
        //加1000个
        Runnable r2 = new Runnable() {
            @Override
            public void run() {
                for(int i = 0 ; i < 1000;i++){
                    list.add(i);
                }
            }
        };

        new Thread(r2).start();
        //等待0.1秒，让2个线程全部运行完毕
        Thread.sleep(100);
        //打印集合长度，线程安全集合应该是2000
        System.out.println(list.size());
    }
}
```

2.并发Set集合_CopyOnWriteArraySet(重点)

```
public class Test02 {
    public static void main(String[] args) throws InterruptedException {
        //HashSet<Integer> set = new HashSet<>();
        CopyOnWriteArraySet<Integer> set = new CopyOnWriteArraySet<>();

        //加1000个
        Runnable r1 = new Runnable() {
            @Override
            public void run() {
```

```
                for(int i = 0 ; i < 1000;i++){
                    set.add(i);
                }
            }
        };
        new Thread(r1).start();
        //加1000个
        Runnable r2 = new Runnable() {
            @Override
            public void run() {
                for(int i = 1000 ; i < 2000;i++){
                    set.add(i);
                }
            }
        };

        new Thread(r2).start();
        //等待0.1秒，让2个线程全部运行完毕
        Thread.sleep(100);
        //打印集合长度，线程安全集合应该是2000
        System.out.println(set.size());
    }
}
```

3.并发Map集合_ConcurrentHashMap(重点)

```
public class Test {
    public static void main(String[] args) throws InterruptedException {
        //Map<String,Integer> map = new ConcurrentHashMap<String, Integer>();
        Map<String,Integer> map = new HashMap<String, Integer>();
        //存储2000个键值对
        for(int x = 0 ; x < 2000; x++){
            map.put("count"+x,x);
        }

        //开启线程，删除前500个
        Runnable r1 = new Runnable() {
            @Override
            public void run() {
                for(int i = 0 ; i < 500;i++){
                    map.remove("count"+i);
                }
            }
        };
        new Thread(r1).start();
        //开启线程，删除1000-1500个
        Runnable r2 = new Runnable() {
            @Override
            public void run() {
                for(int i = 1000 ; i < 1500;i++){
                    map.remove("count"+i);
```

```
                }
            }
        };

        new Thread(r2).start();
        //等待1秒，让2个线程全部运行完毕
        Thread.sleep(1000);
        //打印集合长度，线程安全集合应该是1000
        System.out.println(map.size());
    }
}
```

2.4 CountDownLatch

- **CountDownLatch**的作用

  允许当前线程,等待其他线程完成某种操作之后,当前线程继续执行

- **CountDownLatch**的API

```
构造方法:
    public CountDownLatch(int count); 需要传入计数器,需要等待的线程数
成员方法:
    public void await() throws InterruptedException// 让当前线程等待 当计数器为0,等待的线程才会继续执行
    public void countDown() // 计数器进行减1
```

- **CountDownLatch**的使用案例

```
需求:
    线程1要执行打印: A和C, 线程2要执行打印: B
    我们需要这样的结果: 线程1 先打印A 线程2打印B 之后 线程1再打印C
                    A  B  C

public class TestDemo {
    public static void main(String[] args) throws InterruptedException {
        //0.创建一个CountDownLatch
        CountDownLatch latch = new CountDownLatch(1);
        //1.创建两个线程
        Thread t1  = new MyThread1(latch);

        Thread t2 = new MyThread2(latch);

        t1.start();

        Thread.sleep(5000);
        t2.start();
    }
}
public class MyThread1 extends Thread {
```

```
    private CountDownLatch latch;
    public MyThread1(CountDownLatch latch){
        this.latch = latch;
    }
    @Override
    public void run() {
        System.out.println("A....");
        try {
            latch.await();//让当前线程等待
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("C....");
    }
}

public class MyThread2 extends Thread {
    private CountDownLatch latch;

    public MyThread2(CountDownLatch latch){
        this.latch = latch;
    }

    @Override
    public void run() {
        System.out.println("B....");
        //让latch的计数器减少1
        latch.countDown();
    }
}
```

2.5 CyclicBarrier

- **CyclicBarrier**的作用

  让多个线程,都到达了某种要求之后,新的任务才能执行

- **CyclicBarrier**的API

  构造方法:
  `public CyclicBarrier(int parties, Runnable barrierAction);`
  需要多少个线程　　所有线程都满足要求了,执行的任务

  成员方法:
  `public int await();`当某个线程达到了,需要调用await()

- **CyclicBarrier**的使用案例

  需求: 部门开会,假设部门有五个人,五个人都到达了才执行开会这个任务

  ```
  public class TestPersonThread {
  ```

```
    public static void main(String[] args) throws InterruptedException {
        //0.创建一个CyclicBarrier
        CyclicBarrier barrier = new CyclicBarrier(5, new Runnable() {
            @Override
            public void run() {
                System.out.println("人都齐了,开会吧");
            }
        });

        //1.创建五个线程
        PersonThread p1 = new PersonThread(barrier);
        PersonThread p2 = new PersonThread(barrier);
        PersonThread p3 = new PersonThread(barrier);
        PersonThread p4 = new PersonThread(barrier);
        PersonThread p5 = new PersonThread(barrier);
        //2.开启
        p1.start();
        p2.start();
        p3.start();
        p4.start();
        p5.start();
        //Thread.sleep(6000);
        //System.out.println("人都到了,开会吧...");
        //要求,人没到不开会,都到了立刻开会!!!
    }
}

public class PersonThread extends Thread {
    private CyclicBarrier barrier;
    public PersonThread(CyclicBarrier barrier){
        this.barrier = barrier;
    }
    @Override
    public void run() {
        try {
            Thread.sleep(new Random().nextInt(6)*1000);
            System.out.println(Thread.currentThread().getName() + " 到了! ");
            //调用 barrier的await 表示线程到了
            try {
                barrier.await();
            } catch (BrokenBarrierException e) {
                e.printStackTrace();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

补充:
    Math的静态方法
        public static double random(); //获取一个0(包括)到1(不包括)的正小数
```

2.6 Semaphore

- **Semaphore**的作用

  用于控制并发线程的数量!!

- **Semaphore**的API

  构造方法:
  ```
  public Semaphore(int permits); //参数 permits 表示最多允许有多少个线程并发执行
  ```
  成员方法:
  ```
  public void acquire(); //获取线程的许可证
  public void release(); //释放线程的许可证
  ```

- **Semaphore**的使用案例

```
public class MyThread extends Thread {
    private Semaphore semaphore;

    public MyThread(Semaphore semaphore) {
        this.semaphore = semaphore;
    }

    @Override
    public void run(){
        //从Semaphore获取线程的许可
        try {
            semaphore.acquire();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println(Thread.currentThread().getName() + " 进入 时间=" +
System.currentTimeMillis());
        try {
            Thread.sleep(100*new Random().nextInt(10));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() + " 结束 时间=" +
System.currentTimeMillis());
        //归还semaphore线程的许可
        semaphore.release();
    }
}

public class TestDemo {
    public static void main(String[] args) {
        //0.创建Semaphore
        Semaphore semaphore = new Semaphore(3);
```

```
        //最多的并发线程数量为1
        for (int i = 0; i < 10; i++) {
            new MyThread(semaphore).start();
        }
    }
}
```

2.7 Exchanger

- **Exchanger**的作用

  用于线程间的数据交换

- **Exchanger**的API

  构造方法:
  ```
  public Exchanger<V>();
  ```
  成员方法:
  ```
  public V exchange(V x);//参数为发给别的线程的数据,返回值别的线程发过来的数据
  ```

- **Exchanger**的使用案例

```
public class TestExchanger {
    public static void main(String[] args) throws InterruptedException {
        //0.创建一个线程间数据交互对象
        Exchanger<String> exchanger = new Exchanger<String>();

        //1.创建线程A
        ThreadA aThread = new ThreadA(exchanger);
        aThread.start();

        //休眠
        Thread.sleep(5000);

        ThreadB bThread = new ThreadB(exchanger);
        bThread.start();
    }
}

public class ThreadA extends Thread {
    private Exchanger<String> exchanger;

    public ThreadA(Exchanger<String> exchanger) {
        this.exchanger = exchanger;
    }

    @Override
    public void run() {
        System.out.println("线程A,要将礼物AAA,送给线程B...");
        //调用exchanger
        String result = null;
```

```
            try {
                result = exchanger.exchange("AAA");//如果没有别的线程返回给此处结果,此处将阻
塞
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("线程A,获取到线程B的礼物:"+result);
        }
    }

    public class ThreadB extends Thread {
        private Exchanger<String> exchanger;

        public ThreadB(Exchanger<String> exchanger) {
            this.exchanger = exchanger;
        }

        @Override
        public void run() {
            System.out.println("线程B,要将礼物BBB,送给线程A...");
            String result = null;
            try {
                result = exchanger.exchange("BBB");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("线程B,获取到线程A的礼物:"+result);
        }
    }
```

#####