final、Atomic原子类、volatile案例研究

2019-05-30 15:02:22 3 0 0

最简单的JavaBean类都不安全

```
/**
 * 不安全的类-可见性导致的线程安全问题
 */
public class UnsafeClass {

    private long count = 0;//共享变量
    //一直使用cpu1执行a线程，a线程只调用get方法
    public long getCount(){return count;};
    //一直使用cpu2执行b线程，b线程只调用set方法
    public void setCount(long count){this.count = count;}
    //在b线程中set了10次，a线程才get出来。

}
```

不加其他关键词，只要同时有get和set方法的类都不安全了？

-听上去没错，以后这个JavaBean就是判断线程安全不安全的标准了

寻找get与set的变种

```
/**
 * 变种1--实际且经常写的一个类
 * get与set隐藏的很深
 */
public class UnsafeClass1 {

    private long count = 0;//共享变量
    //虽然没有直接get、但是类似 get方法
    public long methed1(){
          long mycount = count;
          mycount = mycount/3.0;
          //将mycount数据插入数据库
        };
    //虽然没有直接set、但是类似set方法
    public void methed2(long a){
            long c = a +50;
            this.count = c;
         }


}

/**
```

```
 * 变种2--实际且经常写的一个类
 * get与set隐藏的很深
 */
public class UnsafeClass2 {

    private long count = 0;//共享变量
    //虽然没有直接set和get、但是类似set方法和get方法
    public void methed1(long a){
            long c = count + a;
            this.count = c;
        }

        //上边类似的方法
        //public void methed2(long a){
        //   long c = getCount() + a;
        //   setCount(c);
       // }

}
/**
 * 变种3--实际且经常写的一个类
 * get与set隐藏的很深
 */
public class UnsafeClass {
    public long count = 0;//public 属性、默认开放了set与get方法
}
/**
 * 变种4-安全的-实际且经常写的一个类
 */
public class SafeClass {

    //count不是共享变量、而是局部变量，即使有get和set也是线程安全的。
    public void methed1(long a){
            long count = 0;
            long c = count + a;
            count = c;
        }


}
```

有共享变量、只有get没有set的类呢？

```
/**
 * 只有get方法没有set方法 也是安全的
 * @author Administrator
 *
 */
public class SafeClass {

    private long count = 0;

    public SafeClass(){
        this.count = 1;

    }
    public long getCount() {
        return count;
    }
}
```

总结：只有get方法，不提供set方法那不就类似不可变类

不可变类一定线程安全

```
//不可变类--一定线程安全  不可被继承
public final class ImmutableClass2 {
    public final long count;// 即使发布出去该属性也不可修改

    public ImmutableClass2(long count) {
        this.count = count;
    }
    // 只允许count初始化一次 ， 该类只有初始化后才可以被多个线程使用。该类只有读操作，即多个线程只能读不可修改。
    // 所以不可变类是线程安全的

    public static void main(String[] args) {
        ImmutableClass2 tenNum = new ImmutableClass2(10);
        // tenNum.count = 100;//这一步单线程中就会报错，更不用说多线程
    }

}
```

不可变类一定是被final修饰

被final修饰一定是不可变类吗？

```

/**
 *
 * @author Administrator
 * 一个类A即使全都是用final修饰的属性也不一定是线程安全的、只有当属性是线程安全的类时候该类A才是线程安全的。（数组除外）
 * 一个类A有数组属性、不管使用不使用final修饰都不是线程安全的、final只是确保引用不能确
```

```
    保元素。
 7.  */
 8.
 9. public class MutableClass3 {
10.     public long otherCount = 0;// public属性 不安全
11.
12.     // public属性 otherCountArray引用是安全的，但引用值可以更改、不安全
13.     //要使otherCountArray的值不可更改 可以Collections.unmodifiableList
14.     //
15.     /**
16.      * public属性 otherCountArray引用是安全的，但引用值可以更改、不安全。倘若要使otherCountArray的值也不可更改
17.      *
18.      * 1. 可使用Collections.unmodifiableList 限定不可更改里面的值做到安全
19.      * 像Collections.unmodifiableList功能的还有Collections.unmodifiableMap(m)、
20.      * Collections.unmodifiableSet(s)、Collections.unmodifiableSortedMap(m)
21.      * Collections.unmodifiableSortedSet(s)、Collections.unmodifiableNavigableMap(arg0)、
22.      * Collections.unmodifiableNavigableSet(arg0)、Collections.unmodifiableCollection(c)
23.      * 2. 深拷贝一份发布出去。发布出去的数据不管外部如何修改，均并不会影响未源数据
24.      * 通过以上两种操作后otherCountArray属性也变成不可变的了
25.      */
26.     public final long[] otherCountArray = new long[10];//public属性 线程不安全
27.
28.     public final long count;// 安全的
29.     public final Long countL;// 安全的
30.
31.     private final Persion p; //线程不安全
32.
33.     public MutableClass3(long count) {
34.         this.count = count;
35.         this.countL = new Long(0);
36.
37.         this.p = new Persion();
38.     }
39.
40.     public void service(){
41.         otherCountArray[0] = 100;//但引用值可以更改、不安全
42.         Collections.unmodifiableList(Longs.asList(otherCountArray));
43.
44.     }
45.
46.     public Persion getP() {
47.         return p;
48.     }
49.
50. }
```

不可变类有没有变种？

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.Date;
import java.util.List;

//不可变类-变种--一定线程安全  不可被继承
public final class ImmutableOtherClass4 {
    public final long count;// 即使发布出去该属性也不可修改 --正常的不可变类属性
    private final List<Long> otherCountArray = new ArrayList<>();// private属性
    // otherCountArray通过Collections.unmodifiableList变种后仍然达到不可变的效果
    private final List<Long> unModifileotherCountArray = Collections.unmodifiableList(otherCountArray);

    private final Date start;// Date是可变类、通过拷贝的方式赋值给start，通过拷贝的形式发布出去。
    private final Date end;// Date是可变类、通过拷贝的方式赋值给end，通过拷贝的形式发布出去。

    public ImmutableOtherClass4(long count, Date start, Date end) {
        this.count = count;
        this.start = new Date(start.getTime());
        this.end = new Date(end.getTime());
    }

    // 只允许count初始化一次 ，该类只有初始化后才可以被多个线程使用。该类只有读操作，即多个线程只能读不可修改。
    // 所以不可变类是线程安全的
    public long getCount() {
        return count;
    }

    public List<Long> getOtherCountArray() {
        return otherCountArray;
    }

    public List<Long> getUnModifileotherCountArray() {
        return unModifileotherCountArray;
    }

    public Date getStart() {
        return new Date(start.getTime());
    }

    public Date getEnd() {
        return new Date(end.getTime());
    }

}
```

保证可见性与禁止重排序的volatile关键字

```
1.  public class volatileClass5 {
2.      /**
3.       * JMM规定两个线程通讯必须把数据放在主内存。
4.       * 这里使用volatile后
5.       * set后cpu一定将数据放在主内存
6.       * get方法后cpu一定从主内存中读取
7.       */
8.      private  volatile long count;//线程安全的
9.      private  volatile Long countL;//仍然线程安全的、因为Long是线程安全的不可变类
10.
11.     //从主内存获得值后放到工作内存中--不存在缓存
12.     public long getCount() {
13.         return count;
14.     }
15.     //从工作内存修改值后放到主内存中--不存在缓存
16.     public void setCount(long count) {
17.         this.count = count;
18.     }
19.
20.     public Long getCountL() {
21.         return countL;
22.     }
23.     public void setCountL(Long countL) {
24.         this.countL = countL;
25.     }
26.
27. }
```

volatile关键字能保证线程安全吗？

```
public class volatileUnsafeClass6 {
    /**
     * JMM规定两个线程通讯必须把数据放在主内存。
     * 这里使用volatile后
     * set后cpu一定将数据放在主内存
     * get方法后cpu一定从主内存中读取
     */
    private  volatile long count;

    //从主内存获得值后放到工作内存中--不存在缓存
    public long getCount() {
        return count;
    }
    //从工作内存修改值后放到主内存中--不存在缓存
    public void setCount(long count) {
        this.count = count;
    }

    //使用方法一：多个线程set、多个线程get 线程之间是不存在安全问题的

    //使用方法二：多个线程get之后计算出新值后set 情景 会出现线程安全问题

    //综上：判断该类是否是线程安全的需要根据具体的应用场景（系统）判断 。确切的说该类也不是线程安全的类

}
```

volatile修饰数组，能保证数组内部元素的可见性与禁止重排序吗？

```
public class volatileArrayUnsafeClass7 {
    /**
     * JMM规定两个线程通讯必须把数据放在主内存。
     * 这里使用volatile后
     * set后cpu一定将数据放在主内存
     * get方法后cpu一定从主内存中读取
     */
    private  volatile long count;
    //volatile只对数据的引用保持可见性、而不是他的元素
    private  volatile long[] countArray;

    //从主内存获得值后放到工作内存中--不存在缓存
    public long getCount() {
        return count;
    }
    //从工作内存修改值后放到主内存中--不存在缓存
    public void setCount(long count) {
        this.count = count;
    }


    //使用方法一：多个线程set、多个线程get 线程之间是不存在安全问题的

    //使用方法二：多个线程get之后计算出新值后set 情景 会出现线程安全问题

    //综上：判断该类是否是线程安全的需要根据具体的应用场景（系统）判断

    public long[] getCountArray() {
        return countArray;
    }
    public void setCountArray(long[] countArray) {
        this.countArray = countArray;
    }

}
```

volatile修饰普通javabean的情况

```
public class volatilePersionUnsafeClass10 {
    /**
     * JMM规定两个线程通讯必须把数据放在主内存。
     * 这里使用volatile后
     * set后cpu一定将数据放在主内存
     * get方法后cpu一定从主内存中读取
     */
    private  volatile Persion p;


    public Persion getP() {
        return p;
    }

    public void setP(Persion p) {
        this.p = p;
    }
    //即使使用方法一（多个线程读取、多个线程写）也会出现线程安全问题
    //persion里面的name变化其他线程可能读到也可能读不到，因为Persion不是线程安全的

}
```

Atomic原子类用了就安全吗？

```
 1.
 2.  public class AtomicClass8 {
 3.      /**
 4.       * JMM规定两个线程通讯必须把数据放在主内存。
 5.       * set后cpu不一定将数据放在主内存也有可能放在cpu缓存中
 6.       * get方法后cpu可能不从主内存中读取，有可能从cpu缓存中读取
 7.       * 要想使set值后数据放在主内存和get从主内存中读必须使用volatile
 8.       */
 9.      private   AtomicLong count; //同样是多线程缓存问题、这种写法会产生线程安全
10.
11.
12.      //从主内存获得值后放到工作内存中--该方法存在缓存在寄存器和其他cpu看不见的地方的可能
13.      public AtomicLong getCount() {
14.          return count;
15.      }
16.      //从工作内存修改值后放到主内存中--该方法存在缓存在寄存器和其他cpu看不见的地方的可能
17.      public void setCount(AtomicLong count) {
18.          this.count = count;
19.      }
20.
21.      /**
22.       * 也许上面的get与set方法就不应该出现在AtomicClass8中
23.       * 真正委托的用途是
24.       *
25.       public void setLong(long l){
26.          count.set(l);
27.       }
28.       public long getLong(){
29.          return count.get();
30.       }
31.       *
32.       *
33.       */
34.
35.
36.  }
```

tomic原子类怎么用才安全？

```
1.
2.  public class AtomicSafeClass9 {
3.
4.      private final AtomicLong count; // 使用final确保该类是不可变类、但该类中元素无法受到约束
5.
6.      public AtomicSafeClass9(AtomicLong count) {
7.          this.count = count;
8.      }
9.
10.     public AtomicLong getCount() {
11.         return count;
12.     }
13.
14.     public void service() {
15.         count.incrementAndGet();// 该操作是原子操作、底层通过CAS实现的。不断轮询直到更新成功。--很多时间处于忙等状态。
16.     }
17.
18.     /**
19.      * 这个才叫委托给Atomic类
20.      *
21.      * @param l
22.      */
23.     public void setLong(long l) {
24.         count.set(l);
25.     }
26.
27.     public long getLong() {
28.         return count.get();
29.     }
30.
31.     /**
32.      * 该方法仍然会出现线程安全问题-因为两个原子操作不能合并成一个原子操作、仍然有线程安全的隐患 public void
33.      * unsafeService(){ if(count.get() == 100){ count.set(200); } }
34.      **/
35.     // 综上：判断该类是否是线程安全的需要根据发布的方法是否有隐患来判断、单纯的使用AtomicLong并不代表一定是线程安全的
36.
37. }
```

加入synchronized一定是线程安全的吗？

```
/**
 * 同步的persion
 * 加入synchronized一定是线程安全的吗？还需要进一步保证线程安全
 */
public class SyncPersionArraySystem {
    private final Persion[] ps = new Persion[10];
    private final Object lock = new Object();
    //加synchronized保证Persion[]被修改与读取时候是线程安全的
    //使用Arrays.copyOf保证发布到外部的Persion数组里面的元素被修改均不会影响PersionArraySystem的安全性
    public Persion[] getPs(){
        synchronized (lock) {
            return Arrays.copyOf(ps, ps.length);
        }
    }

    //加synchronized保证Persion[]被修改与读取时候是线程安全的
    //使用new Persion(p)保证外部的Persion修改不会影响PersionArraySystem的安全性
    public void setPs(int index,Persion p){
        synchronized (lock) {
            this.ps[index] = new Persion(p);
        }
    }


}
```

被封装到Threadlocal中避免了线程间数据共享

```

/**
 *
 * @author Administrator
 *
 */
public class ThreadlocalSystem {

    //使用线程封闭技术
    private static final ThreadLocal<Persion> persionHolder = new ThreadLocal<Persion>(){
         protected Persion initialValue() {
             return new Persion();
         };
    };


    /**
     * Persion会被封装到线程中所以不会造成线程安全问题，确切的说"避免了线程共享"
     *
     */
    public void setPersion(Persion p){
```

```
22.          persionHolder.set(p);
23.      }
24.      public Persion getPersion(){
25.          return persionHolder.get();
26.      }
27.
28.      /**
29.       *  该类中加入该方法、会造成persionHolder这个属性发布出去
30.       *  一个线程读取、一个线程写会造成persionHolder这个属性产生线程安全
31.       *
32.          public ThreadLocal<Persion> getPersionHolder() {
33.              return persionHolder;
34.          }
35.          public void setPersionHolder(ThreadLocal<Persion> persionHolder) {
36.              this.persionHolder = persionHolder;
37.          }
38.
39.       *
40.       */
41.
42.
43.      public static void main(String[] args) {
44.          ThreadlocalSystem ccc = new ThreadlocalSystem();
45.          //一个线程对ccc执行getPersionHolder
46.          //另一个线程对ccc执行setPersionHolder
47.
48.          boolean flag = true;
49.          while (true) {
50.              //线程一旦启动，永不结束
51.              if(flag){
52.                  persionHolder.set(new Persion("张三"));
53.                  flag = false;
54.              }
55.              //...省略
56.              if(!flag){
57.                  persionHolder.set(null);//不再存储任何数据、之前存储的  Persion("张
    三")会被回收
58.              }
59.
60.
61.          }
62.      }
63.
64.  }
```

使用栈封闭技术

```
1.  public class StackSystem {
2.
3.
4.      //使用栈封闭技术
5.      public void service(){
6.
7.          Persion p = new Persion();
8.          //虽然出现了"竟态条件"但是仍然是线程安全的
9.          if(p.getCount() == 100){
10.             p.setCount(500);
11.         }
12.
13.     }
14.
15. }
```

实例封装

```java
/**
 *
 * @author Administrator
 *  实例封装
 */
public class SafePackagePersion extends Persion{
    @Override
    public int getCount() {
        synchronized(this){
            return super.getCount();
        }

    }
    @Override
    public String getName() {
    synchronized(this){ return super.getName();}
    }
    @Override
    public void setCount(int count) {
    synchronized(this){     super.setCount(count);}
    }

    @Override
    public void setName(String name) {
    synchronized(this){ super.setName(name);}
    }

    //其他方法 使用的是persion的锁
    public void otherMethed() {
        synchronized(this){
            super.setName("sss");
        }
    }

}
```

上面是优秀封装

另一种封装、增加一个新方法却是不安全的

```java
/**
 *
 * @author Administrator 实例封装
 */
public class UnSafePackagePersion<E> {

    public List<E> list = Collections.synchronizedList(new ArrayList<E>());

//使用的锁不是同一个
    public synchronized boolean putIfAbsent(E x) {
        boolean absent = !list.contains(x);
        if (absent) {
            list.add(x);
        }
        return absent;
    }

}
```