

## 初识CopyOnWriteArrayList

第一次见到CopyOnWriteArrayList，是在研究JDBC的时候，每一个数据库的Driver都是维护在一个CopyOnWriteArrayList中的，为了证明这一点，贴两段代码，第一段在com.mysql.jdbc.Driver下，也就是我们写Class.forName(...)中的内容：



```
public class Driver extends NonRegisteringDriver
    implements java.sql.Driver
{
    public Driver()
        throws SQLException
    {
    }


    static
    {
        try
        {
            DriverManager.registerDriver(new Driver());
        } catch (SQLException E) {
            throw new RuntimeException("Can't register driver!");
        }
    }
}
```



看到com.mysql.jdbc.Driver调用了DriverManager的registerDriver方法，这个类在java.sql.DriverManager下：



```
public class DriverManager
{
    private static final CopyOnWriteArrayList<DriverInfo> registeredDrivers = new CopyOnWriteArrayList();
    private static volatile int loginTimeout = 0;
    private static volatile PrintWriter logWriter = null;
    private static volatile PrintStream logStream = null;
    private static final Object logSync = new Object();
    static final SQLPermission SET_LOG_PERMISSION = new SQLPermission("setLog");
    ...
}
```

```
}  

```

看到所有的DriverInfo都在CopyOnWriteArrayList中。既然看到了CopyOnWriteArrayList，我自然免不了要研究一番为什么JDK使用的是这个List。

首先提两点：

- 1、CopyOnWriteArrayList位于java.util.concurrent包下，可想而知，这个类是为并发而设计的
- 2、CopyOnWriteArrayList，顾名思义，Write的时候总是要Copy，也就是说**对于CopyOnWriteArrayList，任何可变的操作（add、set、remove等等）都是伴随复制这个动作的**，后面会解读CopyOnWriteArrayList的底层实现机制

四个关注点在CopyOnWriteArrayList上的答案


关 注 点	结 论
CopyOnWriteArrayList是否允许空	允许
CopyOnWriteArrayList是否允许重复数据	允许
CopyOnWriteArrayList是否有序	有序
CopyOnWriteArrayList是否线程安全	线程安全

如何向CopyOnWriteArrayList中添加元素

对于CopyOnWriteArrayList来说，增加、删除、修改、插入的原理都是一样的，所以用增加元素来分析一下CopyOnWriteArrayList的底层实现机制就可以了。先看一段代码：

```
1 public static void main(String[] args)  
2 {  
3     List<Integer> list = new CopyOnWriteArrayList<Integer>();  
4     list.add(1);  
5     list.add(2);  
6 }
```

看一下这段代码做了什么，先是第3行的实例化一个新的CopyOnWriteArrayList：

```
  
public class CopyOnWriteArrayList<E>
```

```

implements List<E>, RandomAccess, Cloneable, java.io.Serializable {
private static final long serialVersionUID = 8673264195747942595L;

/** The lock protecting all mutators */
transient final ReentrantLock lock = new ReentrantLock();

/** The array, accessed only via getArray/setArray. */
private volatile transient Object[] array;

...
}

```



```

public CopyOnWriteArrayList() {
    setArray(new Object[0]);
}

```


```

final void setArray(Object[] a) {
    array = a;
}

```

看到，对于CopyOnWriteArrayList来说，底层就是一个Object[] array，然后实例化一个CopyOnWriteArrayList，用图来表示非常简单：

**Object[] array**



就是这样，Object array指向一个数组大小为0的数组。接着看一下，第4行的add一个整数1做了什么，add的源代码是：



```

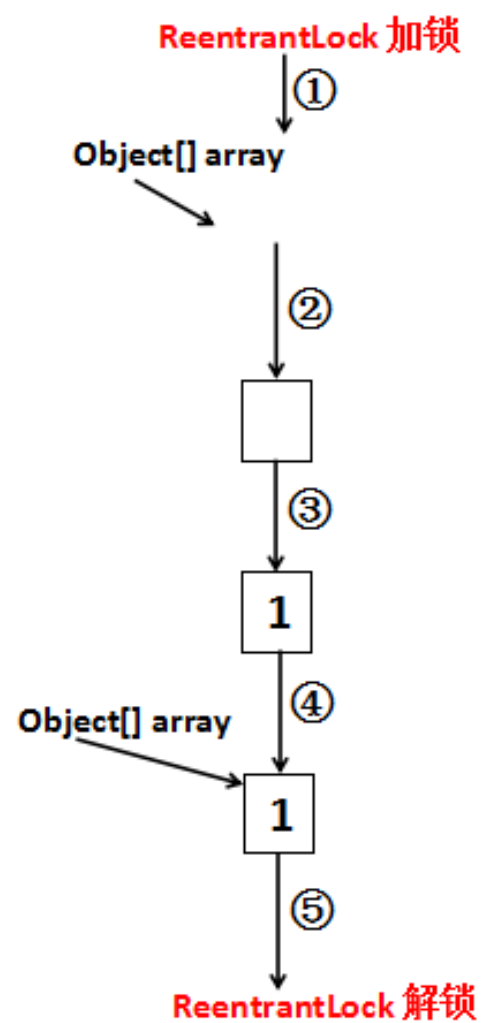
public boolean add(E e) {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        Object[] elements = getArray();
        int len = elements.length;
        Object[] newElements = Arrays.copyOf(elements, len + 1);
        newElements[len] = e;
    }
}

```

```
    setArray(newElements);  
    return true;  
} finally {  
    lock.unlock();  
}  
}
```



画一张图表示一下：

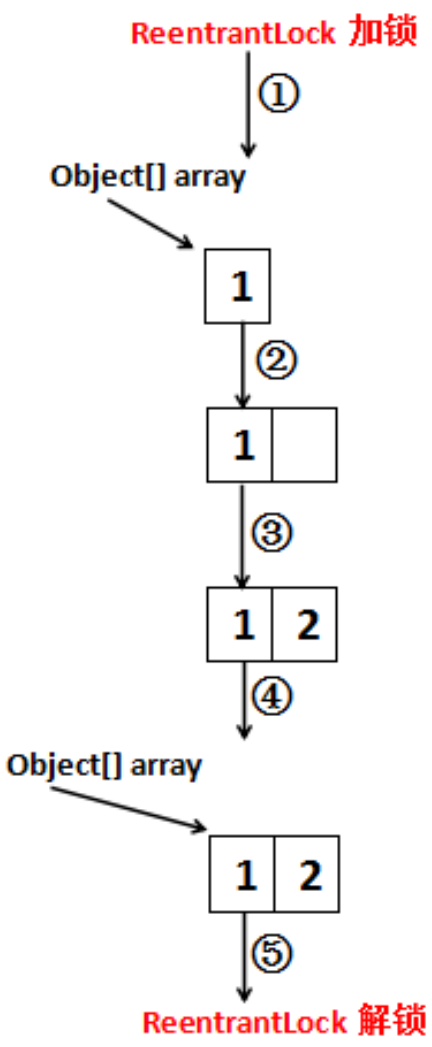


每一步都清楚地表示在图上了，一次add大致经历了几个步骤：

- 1、加锁
- 2、拿到原数组，得到新数组的大小（原数组大小+1），实例化出一个新的数组来

- 3、把原数组的元素复制到新数组中去
- 4、新数组最后一个位置设置为待添加的元素（ 因为新数组的大小是按照原数组大小+1 来的 ）
- 5、把Object array引用指向新数组
- 6、解锁

整个过程看起来比较像ArrayList的扩容。有了这个基础，我们再来看一下第5行的add了一个整数2做了什么，这应该非常简单了，还是画一张图来表示：



和前面差不多，就不解释了。

另外，插入、删除、修改操作也都是一样，每一次的操作都是以对`Object[] array`进行一次复制为基础的，如果上面的流程看懂了，那么研究插入、删除、修改的源代码应该不难。

普通List的缺陷

常用的List有ArrayList、LinkedList、Vector，其中前两个是线程非安全的，最后一个是线程安全的。我有一种场景，两个线程操作了同一个List，分别对同一个List进行迭代和删除，就如同下面的代码：



```
public static class T1 extends Thread
{
    private List<Integer> list;

    public T1(List<Integer> list)
    {
        this.list = list;
    }

    public void run()
    {
        for (Integer i : list)
        {
        }
    }
}

public static class T2 extends Thread
{
    private List<Integer> list;

    public T2(List<Integer> list)
    {
        this.list = list;
    }

    public void run()
    {
        for (int i = 0; i < list.size(); i++)
        {
            list.remove(i);
        }
    }
}
```



首先我在这两个线程中放入ArrayList并启动这两个线程：



```
public static void main(String[] args)
{
    List<Integer> list = new ArrayList<Integer>();

    for (int i = 0; i < 10000; i++)
    {
        list.add(i);
    }

    T1 t1 = new T1(list);
    T2 t2 = new T2(list);
    t1.start();
    t2.start();
}
```



运行结果为：

```
Exception in thread "Thread-0" java.util.ConcurrentModificationException
    at java.util.AbstractList$Itr.checkForComodification(AbstractList.java:372)
    at java.util.AbstractList$Itr.next(AbstractList.java:343)
    at com.xrq.test60.TestMain$T1.run(TestMain.java:19)
```

把ArrayList换成LinkedList，main函数的代码就不贴了，运行结果为：

```
Exception in thread "Thread-0" java.util.ConcurrentModificationException
    at java.util.LinkedList$ListItr.checkForComodification(LinkedList.java:761)
    at java.util.LinkedList$ListItr.next(LinkedList.java:696)
    at com.xrq.test60.TestMain$T1.run(TestMain.java:19)
```

可能有人觉得，这两个线程都是线程非安全的类，所以不行。其实这个问题和线程安不安全没有关系，换成Vector看一下运行结果：


```
Exception in thread "Thread-0" java.util.ConcurrentModificationException
    at java.util.AbstractList$Itr.checkForComodification(AbstractList.java:372)
    at java.util.AbstractList$Itr.next(AbstractList.java:343)
    at com.xrq.test60.TestMain$T1.run(TestMain.java:19)
```

Vector虽然是线程安全的，但是只是一种相对的线程安全而不是绝对的线程安全，它只能够保证增、删、改、查的单个操作一定是原子的，不会被打断，但是如果组合起来用，并不能保证线程安全性。比如就像上面的

线程1在遍历一个Vector中的元素、线程2在删除一个Vector中的元素一样，势必产生并发修改异常，也就是**fail-fast**。

## CopyOnWriteArrayList的作用


把上面的代码修改一下，用CopyOnWriteArrayList：



```
public static void main(String[] args)
{
    List<Integer> list = new CopyOnWriteArrayList<Integer>();

    for (int i = 0; i < 10; i++)
    {
        list.add(i);
    }

    T1 t1 = new T1(list);
    T2 t2 = new T2(list);
    t1.start();
    t2.start();
}
```



可以运行一下这段代码，是没有任何问题的。

看到我把元素数量改小了一点，因为我们从上面的分析中应该可以看出，CopyOnWriteArrayList的缺点，就是修改代价十分昂贵，每次修改都伴随着一次的数组复制；但同时优点也十分明显，就是在并发下不会产生任何的线程安全问题，也就是绝对的线程安全，这也是为什么我们要使用CopyOnWriteArrayList的原因。

另外，有两点必须讲一下。我认为CopyOnWriteArrayList这个并发组件，其实反映的是两个十分重要的分布式理念：

### （1）读写分离

我们读取CopyOnWriteArrayList的时候读取的是CopyOnWriteArrayList中的Object[] array，但是修改的时候，操作的是一个新的Object[] array，读和写操作的不是同一个对象，这就是读写分离。这种技术数据库用的非常多，在高并发下为了缓解数据库的压力，即使做了缓存也要对数据库做读写分离，读的时候使用读库，写的时候使用写库，然后读库、写库之间进行一定的同步，这样就避免同一个库上读、写的IO操作太多

### （2）最终一致

对CopyOnWriteArrayList来说，线程1读取集合里面的数据，未必是最新的数据。因为线程2、线程3、线程4四个线程都修改了CopyOnWriteArrayList里面的数据，但是线程1拿到的还是最老的那个Object[]



] array，新添加进去的数据并没有，所以线程1读取的内容未必准确。不过这些数据虽然对于线程1是不一致的，但是对于之后的线程一定是一致的，它们拿到的Object[] array一定是三个线程都操作完毕之后的Object array[]，这就是最终一致。最终一致对于分布式系统也非常重要，它通过容忍一定时间的数据不一致，提升整个分布式系统的可用性与分区容错性。当然，最终一致并不是任何场景都适用的，像火车站售票这种系统用户对于数据的实时性要求非常非常高，就必须做成强一致性的。

最后总结一点，随着CopyOnWriteArrayList中元素的增加，CopyOnWriteArrayList的修改代价将越来越昂贵，因此，**CopyOnWriteArrayList适用于读操作远多于修改操作的并发场景中。**

