发件人：方堃 fangkun119@icloud.com
主题：CH03 Immutable Object
日期：2017年5月2日 上午11:59
收件人：

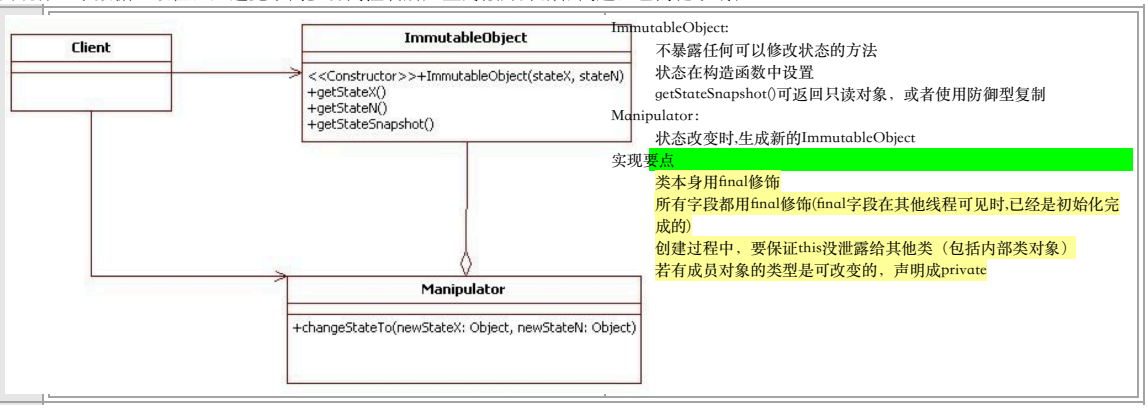
CH03 Immutable Object

2017年4月21日 星期五
上午9:42

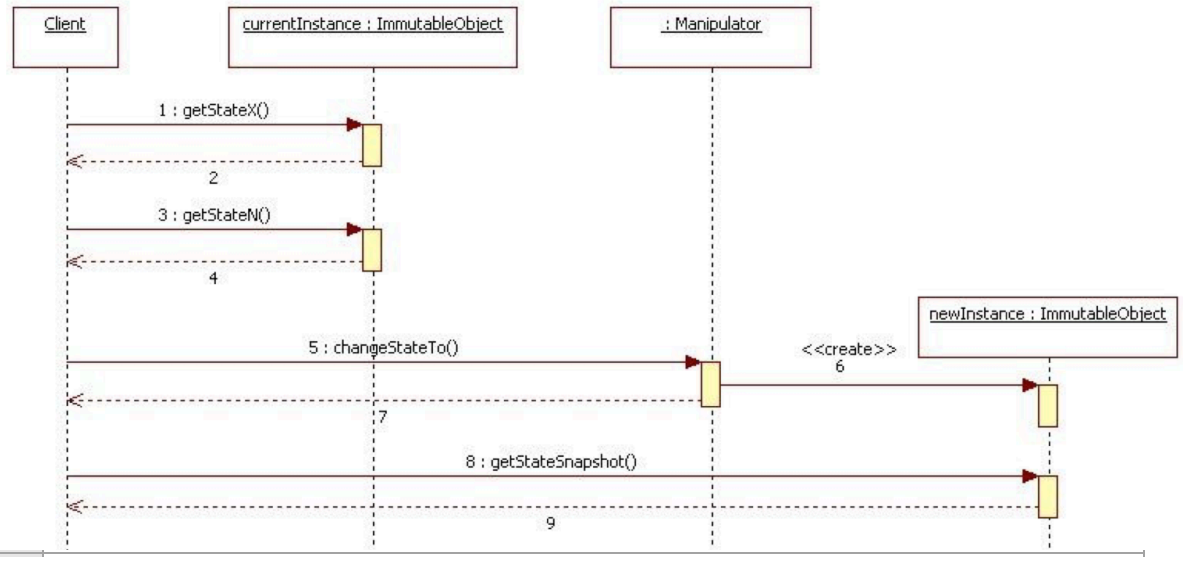
用途

多线程环境中，一个对象常常会被多个线程共享。这种情况下，如果存在多个线程并发地修改该对象的状态或者一个线程访问该对象的状态而另外一个线程试图修改该对象的状态，我们不得不做一些同步访问控制以保证数据一致性。而这些同步访问控制，如显式锁（Explicit Lock）和CAS（Compare and Swap）操作，会带来额外的开销和问题，如上下文切换、等待时间和ABA问题等。Immutable Object模式的意图是通过使用对外可见的状态不可变的对象（即Immutable Object），使得被共享对象“天生”具有线程安全性，而无须额外地同步访问控制。从而既保证了数据一致性，又避免了同步访问控制所产生的额外开销和问题，也简化了编程。

UML



交互



场景

- **被建模对象的状态变化不频繁：**正如本章案例所展示的，这种场景下可以设置一个专门的线程（Manipulator参与者所在的线程）用于在被建模对象状态变化时创建新的不可变对象。而其他线程则只是读取不可变对象的状态。此场景下的一个小技巧是Manipulator对不可变对象的引用采用volatile关键字修饰，既可以避免使用显式锁（如synchronized），又可以保证多线程间的内存可见性。
- **同时对一组相关的数据进行写操作，因此需要保证原子性：**此场景为了保证操作的原子性，通常的做法是使用显式锁。但若采用Immutable Object模式，将这一组相关的数据“组合”成一个不可变对象，则对这一组数据的操作就可以无须加显式锁也能保证原子性，这既简化了编程，又提高了代码运行效率。本章开头所举的车辆位置跟踪的例子正是这种场景。
- **使用某个对象作为安全的HashMap的Key：**我们知道，一个对象作为HashMap的Key被“放入”HashMap之后，若该对象状态变化导致了其Hash Code的变化，则会导致后面在用同样的对象作为Key去get的时候无法获取关联的值。尽管该HashMap中的确存在以该对象为Key的条目。相反，由于不可变对象的状态不变，因此其Hash Code也不变。这使得不可变对象非常适于用作HashMap的Key。

注意问题

Immutable Object模式实现时需要注意以下几个问题。

- **被建模对象的状态变更比较频繁：**此时也不见得不能使用Immutable Object模式。只是这意味着频繁创建新的不可变对象，因此会增加JVM垃圾回收（Garbage Collection）的负担和CPU消耗，我们需要综合考虑：被建模对象的规模、代码目标运行环境的JVM内存分配情况、系统对吞吐率和响应性的要求。若这几个方面因素综合考虑都能满足要求，那么使用不可变对象建模也未尝不可。
- **使用等效或者近似的不可变对象：**有时创建严格意义上的不可变对象比较难，但是尽量向严格意义上的不可变对象靠拢也有利于发挥不可变对象的好处。
- **防御性复制：**如果不可变对象本身包含一些状态需要对外暴露，而相应的字段本身又是可变的（如HashMap），那么返回这些字段的方法还是需要做防御性复制，以避免外部代码修改了其内部状态。正如清单3-4的代码中的getRouteMap方法所展示的。

案例

场景	但是这些数据的变化频率并不高。因此，即使是为了保证线程安全，我们也不希望对这些数据的访问进行加锁等并发访问控制，以免产生不必要的开销和问题。这时，Immutable Object模式就派上用场了。
代码	电子书643（以下只是部分代码，deepCopy()是作者实现的函数）

```
public final class MMSCRouter {
```

```
    // 用volatile修饰，保证多线程环境下该变量的可见性
    private static volatile MMSCRouter instance = new MMSCRouter();
            //维护手机号码前缀到彩信中心之间的映射关系
    private final Map<String, MMSCInfo> routeMap;

    public MMSCRouter()  {
     // 将数据库表中的数据加载到内存，存为Map
     this.routeMap = MMSCRouter.retrieveRouteMapFromDB();
    }
     ……
      public static MMSCRouter getInstance()  {

        return instance;
      }
     ……
    public static void setInstance(MMSCRouter newInstance) {
     instance = newInstance;
    }
     ……
    public Map<String, MMSCInfo> getRouteMap()  {
     //做防御性复制
     return Collections.unmodifiableMap(deepCopy(routeMap));
    }

}
```

JDK案例

为了保证线程安全而在遍历时对集合对象进行加锁，但这在某些情形下可能并不合适，比如系统中对该集合的插入和删除的操作频率远比遍历操作的频率要高。JDK 1.5中引入的类java.util.concurrent.CopyOnWriteArrayList应用了Immutable Object模式，使得对CopyOnWriteArrayList实例进行遍历时不用加锁也能保证线程安全。当然，CopyOnWriteArrayList也不是“万能”的，它是专门针对遍历操作的频率比添加和删除操作更加频繁的场景设计的。CopyOnWriteArrayList的源码（骨架）如清单3-8所示。

清单3-8. JDK类CopyOnWriteArrayList的源码（骨架）

```
public class CopyOnWriteArrayList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable {

    //省略其他代码

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

    /**
     * Gets the array.  Non-private so as to also be accessible
     * from CopyOnWriteArraySet class.
     */
    final Object[]  getArray()  {
        return array;
    }

    /**
     * Sets the array.
       */
     final void setArray(Object[]  a) {
         array = a;
}

//省略其他代码

//遍历集合
    public Iterator<E> iterator()  {
        return new COWIterator<E>(getArray(), 0);
    }

    //添加元素
```

```
    public boolean add(E e) {
final ReentrantLock lock = this.lock;
lock.lock();
try {
    Object[]  elements = getArray();
    int len = elements.length;

            //复制原数组，并在此基础上将新数组的最后一个元素设置为要添加的元素
    Object[]  newElements = Arrays.copyOf(elements, len + 1);
    newElements[len] = e;

             //直接将array变量设置为新的数组
    setArray(newElements);
    return true;
} finally {
        lock.unlock();
  }
}

//省略其他代码
}
```

使用 Microsoft OneNote for Mac 创建。