77-开源实战一（下）：通过剖析JavaJDK源码学习灵活应用设计模式

上一节课，我们讲解了工厂模式、建造者模式、装饰器模式、适配器模式在Java JDK中的应用，其中，Calendar类用到了工厂模式和建造者模式，Collections类用到了装饰器模式、适配器模式。学习的重点是让你了解，在真实的项目中模式的实现和应用更加灵活、多变，会根据具体的场景做实现或者设计上的调整。

今天，我们继续延续这个话题，再重点讲一下模板模式、观察者模式这两个模式在JDK中的应用。除此之外，我还会对在理论部分已经讲过的一些模式在JDK中的应用做一个汇总，带你一块回忆复习一下。

话不多说，让我们正式开始今天的学习吧！

模板模式在Collections类中的应用

我们前面提到，策略、模板、职责链三个模式常用在框架的设计中，提供框架的扩展点，让框架使用者，在不修改框架源码的情况下，基于扩展点定制化框架的功能。Java中的Collections类的sort()函数就是利用了模板模式的这个扩展特性。

首先，我们看下Collections.sort()函数是如何使用的。我写了一个示例代码，如下所示。这个代码实现了按照不同的排序方式（按照年龄从小到大、按照名字字母序从小到大、按照成绩从大到小）对students数组进行排序。

```
public class Demo {
  public static void main(String[] args) {
    List<Student> students = new ArrayList<>();
    students.add(new Student("Alice", 19, 89.0f));
    students.add(new Student("Peter", 20, 78.0f));
    students.add(new Student("Leo", 18, 99.0f));

    Collections.sort(students, new AgeAscComparator());
    print(students);

    Collections.sort(students, new NameAscComparator());
    print(students);

    Collections.sort(students, new ScoreDescComparator());
    print(students);
  }

  public static void print(List<Student> students) {
    for (Student s : students) {
      System.out.println(s.getName() + " " + s.getAge() + " " + s.getScore());
    }
  }

  public static class AgeAscComparator implements Comparator<Student> {
    @Override
    public int compare(Student o1, Student o2) {
      return o1.getAge() - o2.getAge();
    }
  }

  public static class NameAscComparator implements Comparator<Student> {
    @Override
    public int compare(Student o1, Student o2) {
```

```
      return o1.getName().compareTo(o2.getName());
    }
  }

  public static class ScoreDescComparator implements Comparator<Student> {
    @Override
    public int compare(Student o1, Student o2) {
      if (Math.abs(o1.getScore() - o2.getScore()) < 0.001) {
        return 0;
      } else if (o1.getScore() < o2.getScore()) {
        return 1;
      } else {
        return -1;
      }
    }
  }
}
```

结合刚刚这个例子，我们再来看下，为什么说Collections.sort()函数用到了模板模式？

Collections.sort()实现了对集合的排序。为了扩展性，它将其中“比较大小”这部分逻辑，委派给用户来实现。如果我们把比较大小这部分逻辑看作整个排序逻辑的其中一个步骤，那我们就可以把它看作模板模式。不过，从代码实现的角度来看，它看起来有点类似之前讲过的JdbcTemplate，并不是模板模式的经典代码实现，而是基于Callback回调机制来实现的。

不过，在其他资料中，我还看到有人说，Collections.sort()使用的是策略模式。这样的说法也不是没有道理的。如果我们并不把“比较大小”看作排序逻辑中的一个步骤，而是看作一种算法或者策略，那我们就可以把它看作一种策略模式的应用。

不过，这也不是典型的策略模式，我们前面讲到，在典型的策略模式中，策略模式分为策略的定义、创建、使用这三部分。策略通过工厂模式来创建，并且在程序运行期间，根据配置、用户输入、计算结果等这些不确定因素，动态决定使用哪种策略。而在Collections.sort()函数中，策略的创建并非通过工厂模式，策略的使用也非动态确定。

观察者模式在JDK中的应用

在讲到观察者模式的时候，我们重点讲解了Google Guava的EventBus框架，它提供了观察者模式的骨架代码。使用EventBus，我们不需要从零开始开发观察者模式。实际上，Java JDK也提供了观察者模式的简单框架实现。在平时的开发中，如果我们不希望引入Google Guava开发库，可以直接使用Java语言本身提供的这个框架类。

不过，它比EventBus要简单多了，只包含两个类：java.util.Observable和java.util.Observer。前者是被观察者，后者是观察者。它们的代码实现也非常简单，为了方便你查看，我直接copy-paste到了这里。

```
public interface Observer {
    void update(Observable o, Object arg);
}

public class Observable {
    private boolean changed = false;
    private Vector<Observer> obs;
```

```
    public Observable() {
        obs = new Vector<>();
    }

    public synchronized void addObserver(Observer o) {
        if (o == null)
            throw new NullPointerException();
        if (!obs.contains(o)) {
            obs.addElement(o);
        }
    }

    public synchronized void deleteObserver(Observer o) {
        obs.removeElement(o);
    }

    public void notifyObservers() {
        notifyObservers(null);
    }

    public void notifyObservers(Object arg) {
        Object[] arrLocal;

        synchronized (this) {
            if (!changed)
                return;
            arrLocal = obs.toArray();
            clearChanged();
        }

        for (int i = arrLocal.length-1; i>=0; i--)
            ((Observer)arrLocal[i]).update(this, arg);
    }

    public synchronized void deleteObservers() {
        obs.removeAllElements();
    }

    protected synchronized void setChanged() {
        changed = true;
    }

    protected synchronized void clearChanged() {
        changed = false;
    }
}
```

对于Observable、Observer的代码实现，大部分都很好理解，我们重点来看其中的两个地方。一个是changed成员变量，另一个是notifyObservers()函数。

我们先来看changed成员变量。

它用来表明被观察者（Observable）有没有状态更新。当有状态更新时，我们需要手动调用setChanged()函数，将changed变量设置为true，这样才能在调用notifyObservers()函数的时候，真正触发观察者（Observer）执行update()函数。否则，即便你调用了notifyObservers()函数，观察者的update()函数也不会被执行。

也就是说，当通知观察者被观察者状态更新的时候，我们需要依次调用setChanged()和notifyObservers()两个函数，单独调用notifyObservers()函数是不起作用的。你觉得这样的设计是不是多此一举呢？这个问题留给你思考，你可以在留言区说说你的看法。

我们再来看notifyObservers()函数。

为了保证在多线程环境下，添加、移除、通知观察者三个操作之间不发生冲突，Observable类中的大部分函数都通过synchronized加了锁，不过，也有特例，notifyObservers()这函数就没有加synchronized锁。这是为什么呢？在JDK的代码实现中，notifyObservers()函数是如何保证跟其他函数操作不冲突的呢？这种加锁方法是否存在问题？又存在什么问题呢？

notifyObservers()函数之所以没有像其他函数那样，一把大锁加在整个函数上，主要还是出于性能的考虑。notifyObservers()函数依次执行每个观察者的update()函数，每个update()函数执行的逻辑提前未知，有可能会很耗时。如果在notifyObservers()函数上加synchronized锁，notifyObservers()函数持有锁的时间就有可能会很长，这就会导致其他线程迟迟获取不到锁，影响整个Observable类的并发性能。

我们知道，Vector类不是线程安全的，在多线程环境下，同时添加、删除、遍历Vector类对象中的元素，会出现不可预期的结果。所以，在JDK的代码实现中，为了避免直接给notifyObservers()函数加锁而出现性能问题，JDK采用了一种折中的方案。这个方案有点类似于我们之前讲过的让迭代器支持”快照“的解决方案。

在notifyObservers()函数中，我们先拷贝一份观察者列表，赋值给函数的局部变量，我们知道，局部变量是线程私有的，并不在线程间共享。这个拷贝出来的线程私有的观察者列表就相当于一个快照。我们遍历快照，逐一执行每个观察者的update()函数。而这个遍历执行的过程是在快照这个局部变量上操作的，不存在线程安全问题，不需要加锁。所以，我们只需要对拷贝创建快照的过程加锁，加锁的范围减少了很多，并发性能提高了。

为什么说这是一种折中的方案呢？这是因为，这种加锁方法实际上是存在一些问题的。在创建好快照之后，添加、删除观察者都不会更新快照，新加入的观察者就不会被通知到，新删除的观察者仍然会被通知到。这种权衡是否能接受完全看你的业务场景。实际上，这种处理方式也是多线程编程中减小锁粒度、提高并发性能的常用方法。

单例模式在Runtime类中的应用

JDK中java.lang.Runtime类就是一个单例类。这个类你有没有比较眼熟呢？是的，我们之前讲到Callback回调的时候，添加shutdown hook就是通过这个类来实现的。

每个Java应用在运行时会启动一个JVM进程，每个JVM进程都只对应一个Runtime实例，用于查看JVM状态以及控制JVM行为。进程内唯一，所以比较适合设计为单例。在编程的时候，我们不能自己去实例化一个Runtime对象，只能通过getRuntime()静态方法来获得。

Runtime类的的代码实现如下所示。这里面只包含部分相关代码，其他代码做了省略。从代码中，我们也可以看出，它使用了最简单的饿汉式的单例实现方式。

```
/**
 * Every Java application has a single instance of class
```

```
  * <code>Runtime</code> that allows the application to interface with
  * the environment in which the application is running. The current
  * runtime can be obtained from the <code>getRuntime</code> method.
  * <p>
  * An application cannot create its own instance of this class.
  *
  * @author  unascribed
  * @see     java.lang.Runtime#getRuntime()
  * @since   JDK1.0
  */
public class Runtime {
  private static Runtime currentRuntime = new Runtime();

  public static Runtime getRuntime() {
    return currentRuntime;
  }

  /** Don't let anyone else instantiate this class */
  private Runtime() {}

  //....
  public void addShutdownHook(Thread hook) {
    SecurityManager sm = System.getSecurityManager();
    if (sm != null) {
       sm.checkPermission(new RuntimePermission("shutdownHooks"));
    }
    ApplicationShutdownHooks.add(hook);
  }
  //...
}
```

其他模式在JDK中的应用汇总

实际上，我们在讲解理论部分的时候，已经讲过很多模式在Java JDK中的应用了。这里我们一块再回顾一下，如果你对哪一部分有所遗忘，可以再回过头去看下。

在讲到模板模式的时候，我们结合Java Servlet、JUnit TestCase、Java InputStream、Java AbstractList四个例子，来具体讲解了它的两个作用：扩展性和复用性。
在讲到享元模式的时候，我们讲到Integer类中的-128~127之间的整型对象是可以复用的，还讲到String类型中的常量字符串也是可以复用的。这些都是享元模式的经典应用。

在讲到职责链模式的时候，我们讲到Java Servlet中的Filter就是通过职责链来实现的，同时还对比了Spring中的interceptor。实际上，拦截器、过滤器这些功能绝大部分都是采用职责链模式来实现的。

在讲到的迭代器模式的时候，我们重点剖析了Java中Iterator迭代器的实现，手把手带你实现了一个针对线性数据结构的迭代器。

重点回顾

好了，今天的内容到此就讲完了。我们一块来总结回顾一下，你需要重点掌握的内容。

这两节课主要剖析了JDK中用到的几个经典设计模式，其中重点剖析的有：工厂模式、建造者模式、装饰器模式、适配器模式、模板模式、观察者模式，除此之外，我们还汇总了其他模式在JDK中的应用，比如：单例模式、享元模式、职责链模式、迭代器模式。

实际上，源码都很简单，理解起来都不难，都没有跳出我们之前讲解的理论知识的范畴。学习的重点并不是表面上去理解、记忆某某类用了某某设计模式，而是让你了解我反复强调的一点，也是标题中突出的一点，在真实的项目开发中，如何灵活应用设计模式，做到活学活用，能够根据具体的场景、需求，做灵活的设计和实现上的调整。这也是模式新手和老手的最大区别。

课堂讨论

针对Java JDK中观察者模式的代码实现，我有两个问题请你思考。

1. 每个函数都加一把synchronized大锁，会不会影响并发性能？有没有优化的方法？
2. changed成员变量是否多此一举？

欢迎留言和我分享你的想法，如果有收获，也欢迎你把这篇文章分享给你的朋友。

精选留言：

- 小晏子 2020-04-29 13:53:13
  思考题：
  1. 每个函数加一把Synchronized锁，在并发激烈的时候是会影响性能的，优化的方式的话确实是可以使用CopyOnWriteList，copyOnWriteList是个并发安全的List，并且它不是基于锁实现的，而且又因为Oberser 中的List很少被修改经常被遍历的特点，所以使用CopyOnWriteList性能会提升。
  2. changed成员变量还是必须的，这么做的好处是可以将“跟踪变化”和“通知观察者”两步分开，处理一些复杂的逻辑， [4赞]

- Darren 2020-04-29 23:49:30
  1、肯定会影响性能，但是因为保存观察者对象的必须是线程安全的，所以是不可避免，根据实际业务场景，如果很少被修改，可以使用CopyOnWriteArrayList来实现，但是如果修改频繁，CopyOnWriteArrayList 本质是写时复制，所以比较消耗内存，不建议使用，可以使用别的，比如ConcurrentSkipListSet等；
  2、change是必须的，有些场景下（比如报警），状态发生变化其实是不报警，持续一定的时间菜报警，所以，把被观察者的对象是否发生变化独立出来，是可以做很多自己业务的事情；可以接单的理解为对变化抽象，提高可扩展性。 [2赞]

- Geek_54edc1 2020-04-29 15:09:05
  1、方案一：使用性能更好的线程安全的容器，来替换vector；方案二：如果没有多线程添加、删除观察者的操作，而是在程序启动时就定义好了观察者，以后也不会变更的话，就不用给相关函数加锁了。
  2、changed成员不是多此一举，如果没有这个成员，notifyObservers()函数在多线程场景下，会出现重复通知观察者的情况。 [2赞]

- test 2020-04-29 11:04:00
  1.会影响，如果要优化，可以使用CopyOnWriteArrayList；
  2.有必要，如果没有change，则需要观察者知道被观测者什么时候会有状态改变。 [2赞]

- 汝林外史 2020-04-29 11:43:28
  为什么说Vector不是线程安全的类呢？？ Vector的方法不都加了synchronize关键字实现串行化并发安全了吗，应该是线程安全的类啊。 [1赞]

- Heaven 2020-04-29 11:39:23

1.肯定降低了性能,而通常优化的手段,是更小粒度的锁或者使用乐观锁,在这个方法中已经将notifyObservers方法原本的大锁,利用一个复制技术缩小到一小点了,也是一种版本控制的方式,这里先给出一个尝试优化,使用原子类Boolean来替换setChanged这个大锁,并且使用copyonwriteArrayList来替换我们的数组
2.如果没有多并发的任何情况,changed的设计就是多此一举了,但是如果出现了高并发,那么直接去尝试直接执行更新操作可能会是一个非常漫长的等待,于是利用一个简单的标识位,并加上了锁来进行了修改,在高并发的情况下,无可厚非 [1赞]

- 罗乾林 2020-04-29 10:07:58
1、会影响并发性能,synchronized主要保证Vector线程安全，高并发下会影响加入集合的速度，可以使用并发性好的无锁化容器
2、当多个线程同时发起notifyObservers时保证只通知Observer一次 [1赞]

- 成楠Peter 2020-04-29 08:45:28
思考题

1，是否能用异步观察者模式，减少并发压力。

2，change必须，如果没有change，那在notifyObservers同步拷贝观察者对象进行通知时，如果这时候有新的变更，那被观察者又会被通知一次。 [1赞]

- 超威丶 2020-04-29 08:26:12
先解决好并发问题，后续影响性能再做优化，没必要一上来就优化，优化也是对于锁的粒度优化 [1赞]

- jinjunzhu 2020-04-30 09:09:02
1.每个函数加synchronized，肯定会有性能影响的，尤其是高并发的情况下，会有大量现场阻塞在入口等待队列。对于非线程安全的操作，加锁不一定要在方法级别，可以在变量级别加锁，也可以用并发包下的一些安全类来取代synchronized
2.changed变量主要好处就是当通知的时候如果没有改变这个变量值，可以直接return。但是我觉得如果通知的时候，忘了set这个变量的值，那不是就相当于通知失败了吗？去掉changed我觉得也可以，保证被观察这在通知的时候，确实是有新消息到来或者有真实事件发生

- jaryoung 2020-04-29 23:03:47
课后习题：
1. 大量并发的时候会影响，但是在少量的并发的时候，其他影响会比较小，毕竟优化后的synchronized不是默认就是重量级锁。优化方案：更换为一些线程安全的集合类，changed 也可以更换为线程安全的AtomicBoolean，简单一句话，就是缩小锁的范围。
2. changed 算一个巧妙的设置吧，可能会存在需求暂时屏蔽某些主播（Observable）。

- Jxin 2020-04-29 13:45:45
1.会，写多场景可以采用分治思想降低锁冲突，数据量不大且写少场景就采用cow拿空间换时间。

2.有这个change字段可能导致丢失通知的情况。并发多个线程发送通知，保障至少一个线程发送通知的场景可以用。

- 不能忍的地精 2020-04-29 11:05:52
1. 加同步关键字的方法操作内容简单,都是对容器进行操作和更改状态,所以影响有限,优化的方法可以是线程隔离.避免多线程操作共享变量的问题

2. changed变量不是多此一举,存在一种情况,就是被观察者行动了,但是条件不满足,但是不需要通知观察者

的情况

- Demon.Lee 2020-04-29 09:29:50

  1. 每个函数都加一把 synchronized 大锁，会不会影响并发性能？有没有优化的方法？
  ---查询资料，vector是jdk很早之前就有的，实现了线程安全，但性能很差，我觉得可以换成CopyOnWriteArrayList会好些，毕竟读多，写少一些。

  2. changed 成员变量是否多此一举
  ---没想明白，不知道是不是防止滥用notifyObservers()方法，必须先设置标志位，然后再通知？看到有hasChanged()方法，难道是让Observer可以主动来检测数据是否变化了？