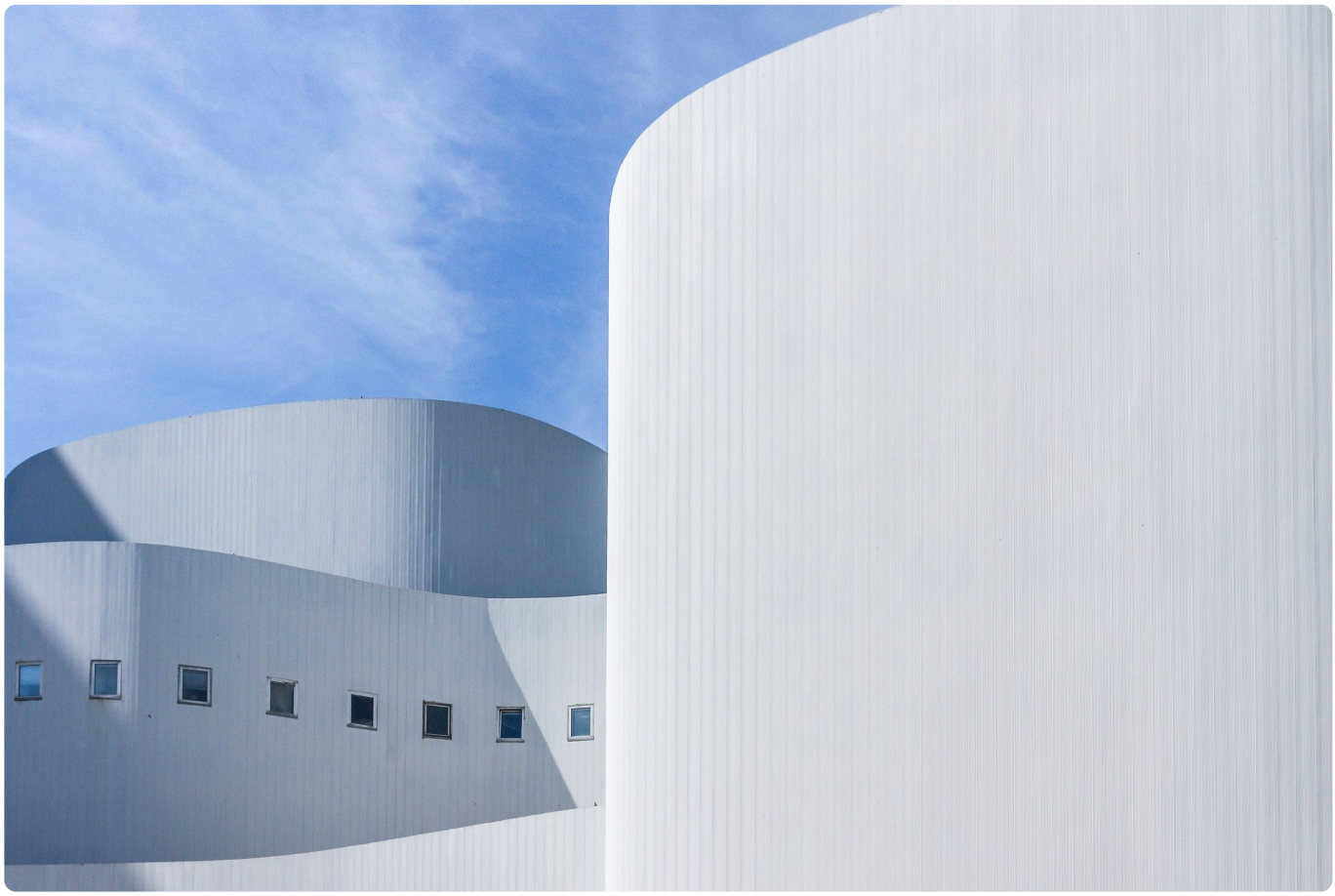
77 | 开源实战一（下）：通过剖析Java JDK源码学习灵活应用设计模式

2020-04-29 王争

设计模式之美 进入课程 >



讲述：冯永吉
时长 11:24 大小 10.45M

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。

复制代码

```
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) {
```

```
34        return o1.getName().compareTo(o2.getName());
35      }
36    }
37
38    public static class ScoreDescComparator implements Comparator<Student> {
39      @Override
40      public int compare(Student o1, Student o2) {
41        if (Math.abs(o1.getScore() - o2.getScore()) < 0.001) {
42          return 0;
43        } else if (o1.getScore() < o2.getScore()) {
44          return 1;
45        } else {
46          return -1;
47        }
48      }
49    }
50  }
```

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。

复制代码

```
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);
```

```
41      }
42
43      public synchronized void deleteObservers() {
44          obs.removeAllElements();
45      }
46
47      protected synchronized void setChanged() {
48          changed = true;
49      }
50
51      protected synchronized void clearChanged() {
52          changed = false;
53      }
54  }
```

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。

复制代码

```
/**
 * 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。

上一篇　76 | 开源实战一（上）：通过剖析Java JDK源码学习灵活应用设计模式

下一篇　加餐一 | 用一篇文章带你了解专栏中用到的所有Java语法

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。