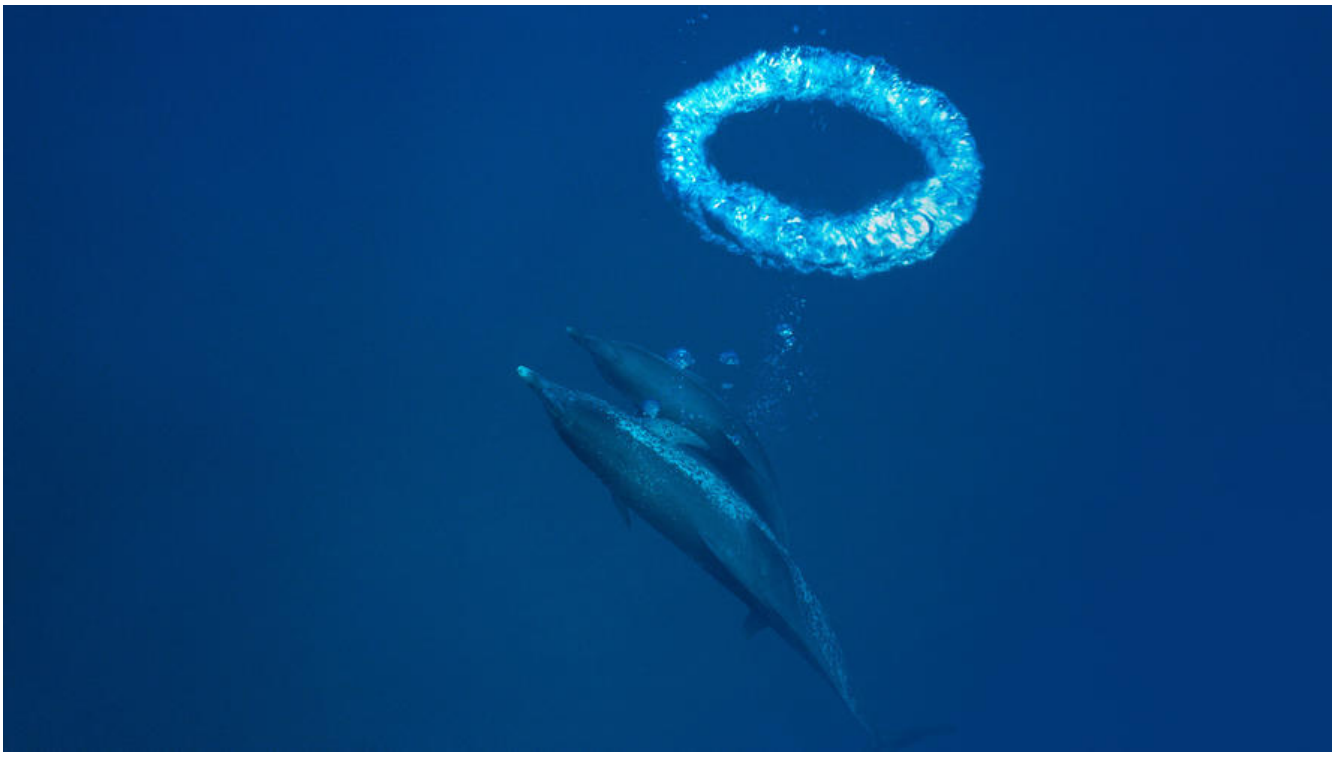
链滴

[阅读] 敏捷软件开发 —— 薪水支付案例研究（一）

作者：lizhongyue248

原文链接：https://ld246.com/article/1554268298675

来源网站：链滴



这一部分与前面的两部分不太一样，他通过实际操作来完成一个完整的案例。并且引入一些常见的设模式，开篇介绍了薪水支付系统的初步规格说明，相当于需求分析。

- 对于钟点工，按照他们雇员记录中 **每小时报酬字段**的值对他们进行支付。每天提交工作时间卡，中记录了**日期以及工作小时数**。如果一天超过八小时，超过的部分会按照**正常报酬的 1.5 倍**进行支付，**周五**对他们进交付。
- 有些雇员完全以 **月薪**进行交付。**每个月的最后一个工作日**对他们进行交付，在他们的雇员记录中一个**月薪字段**。
- 带薪雇员，根据销售情况，支付 **一定量的酬金**。他们会提交销售凭条，其中记录了**销售的日期**和**量**，在他们的雇员记录中有一个**酬金字段**。**每隔一周的周五**对他们进行交付。
- 加入协会的雇员，有一个 **每周应付款项字段**。将会从他们**薪水中扣除**。协会有时也会**针对单个协成员征收服务费用**。协会每周会提交这些**服务费用**，服务费用必须要从**相应雇员的下个月的薪水总额扣除**。
- 应用程序 **每个工作日运行一次**，并在**当天**为相应的雇员进行交付，系统会被告知雇员的**支付日期**这样他会计算从雇员**上次支付日期到规定的本次支付日期间应支付的数额**。

COMMAND 模式和 ACTIVE OBJECT 模式

没有人天生就具有命令他人的权利。

先从简单的入手，COMMAND 模式是最简单、最优雅但同时也是适用性最广的设计模式。

COMMAND 模式

他非常简单，看一下下面的接口

```
public interface Command {
   //  书中为  do()，但是  do  是  java  的关键字，无法作为函数名
   public void execute();
```

```
}
```

正如看到的一样，他只是封装了一个没有任何变量的函数。从严格的面向对象的意义上来说，这种做是被强烈反对的————因为他具有功能分解的味道，他把函数层面的任务提升到了类的层面。然而正是他有趣的地方。

作者举了一个复印机软件的例子，其中通过对依稀的一些的简单的 command 的封装。解除了系统逻辑互联关系和实际链接的设备之间的耦合。

另外一个例子是**创建和执行事务操作**，例如在对数据库进行操作之前，对数据进行 validate 操作。他好的解除了从用户获取数据代码、验证并操作数据的代码以及业务对象本身之间的耦合关系。

第三个例子是回退，可以对某个命令进行撤销操作。

ACTIVE OBJECT 模式

他是实现多线程的一项古老的技术，他可以自动完成动作或改变状态，隔离了方法执行和方法调用的程，提高了并行性，对内部拥有控制线程的主动对象，降低了异步访问的复杂性。

我们需要一个例子，看看以下的代码：

```
public interface Command {
    /**
     * 执行
     */
    public void execute();
}
```

建立一个ActiveObjectEngine

```
/**
 * 维护一个 Command 对象的链表
 *
 * @author echo
 */
public class ActiveObjectEngine {
    private LinkedList<Command> itsCommands = new LinkedList<>();

    /**
     * 添加命令
     *
     * @param command 命令
     */
    public void addCommand(Command command) {
        itsCommands.add(command);
    }

    /**
     * 遍历链表，执行并去除每个命令
     */
    public void run() {
        while (!itsCommands.isEmpty()){
            Command command = itsCommands.getFirst();
            itsCommands.removeFirst();
```

```
            command.execute();
        }
    }
}
```

以及一个实现

```
/**
 * sleep 命令
 * 等待指定数目的毫秒，然后执行 wakeup 命令
 *
 * @author echo
 */
public class SleepCommand implements Command {
    private Command wakeupCommand;
    private ActiveObjectEngine engine;
    private long sleepTime;
    private long startTime = 0;
    private boolean started = false;

    public SleepCommand(int milliseconds, ActiveObjectEngine engine, Command wakeupCo
mand) {
        this.sleepTime = milliseconds;
        this.engine = engine;
        this.wakeupCommand = wakeupCommand;
    }

    /**
     * 执行时，检查自己是以前是已经执行过
     * 如果没有，记录下开始时间
     * 如果没有过延迟时间，就把自己再加到 ActiveObjectEngine 中
     * 如果过了延迟时间，就把 wakeup 命令对象加到 ActiveObjectEngine 中
     */
    @Override
    public void execute() {
        long currentTime = System.currentTimeMillis();
        if (!started) {
            // 未开始时
            started = true;
            startTime = currentTime;
            engine.addCommand(this);
        } else if ((currentTime - startTime) < sleepTime) {
            // 没有过延迟时间
            engine.addCommand(this);
        } else {
            // 过了延迟时间,添加 -> 执行
            engine.addCommand(wakeupCommand);
        }
    }
}
```

然后一个测试

```
class TestSleepCommand {
```

```
    private boolean commandExecuted = false;

    @Test
    void testSleep() {
        Command wakeup = () -> commandExecuted = true;
        ActiveObjectEngine activeObjectEngine = new ActiveObjectEngine();
        SleepCommand sleepCommand = new SleepCommand(1000, activeObjectEngine, wake
p);
        activeObjectEngine.addCommand(sleepCommand);
        long start = System.currentTimeMillis();
        activeObjectEngine.run();
        long stop = System.currentTimeMillis();
        long sleepTime = stop - start;
        assertTrue(commandExecuted, "Command executed!");
        System.out.println("SleepTime " + sleepTime);
    }
}
```

具体的已经添加注释，在处理事件的时候，他不进行阻塞，常常在不符合执行条件的时候，他就把自再次放回到 ActiveObjectEngine 之中。

采用该技术的变体去构建多线程系统已经是很常见的实践，这种类型的线程被称为 run-to-completio （RTC），意味着 command 不会阻塞。

我们来模拟一个例子

```
public class DelayedTyper implements Command {
    private int itsDelay;
    private char itsChar;
    private static ActiveObjectEngine engine = new ActiveObjectEngine();
    private static boolean stop = false;

    public DelayedTyper(int delay, char c) {
        itsDelay = delay;
        itsChar = c;
    }

    public static void main(String[] args) {
        // 进行循环
        engine.addCommand(new DelayedTyper(100, '1'));
        engine.addCommand(new DelayedTyper(300, '3'));
        engine.addCommand(new DelayedTyper(500, '5'));
        engine.addCommand(new DelayedTyper(700, '7'));
        // 设置 stop，停止循环
        Command startCommand = () -> stop = true;
        // 最后一个命令
        engine.addCommand(new SleepCommand(20000, engine, startCommand));
        engine.run();
    }

    /**
     * 打印在构造时传入的字符
     */
```

```
    @Override
    public void execute() {
        System.out.print(itsChar);
        if (!stop) {
            delayAndRepeat();
        }
    }
}
```

COMMAND 模式的简单性掩盖了他的多功能性，但是它可能是不符合面向对象的思维范式的，因为对函数的关注查过了类，但是在实际开发中，他确实是非常有用的。

TEMPLATE METHOD 模式和 STRATEGY 模式：继承与委托

业精于勤

“业精于勤，荒于嬉；行成于思，毁于随。”一本外国的书引入了我们中国的《进学解》，让人意外这句话很好理解，学业由于勤奋而专精，由于玩乐而荒废；德行由于独立思考而有所成就，由于因循俗而败坏。在软件开发中，使用继承我们可以基于差异编程，通过集成可以建立完整的软件结构分类。

但是继承的过度使用是非常糟糕的，代价十分昂贵

优先使用对象组合而不是类继承

TEMPLATE METHOD 模式

华氏度转摄氏度

他是通过继承来解决问题。现在我们有一个程序如下：

```
/**
 * 输入华氏度转化为摄氏度
 *
 * @author echo
 * @version 1.0
 * @date 19-4-3 09:01
 */
public class Ftocraw {
    public static void main(String[] args) throws Exception {
        // 初始化流
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        boolean done = false;
        // 主循环中完成工作
        while (!done) {
            String fabrString = br.readLine();
            if (fabrString == null || fabrString.length() == 0) {
                done = true;
            } else {
                double fahr = Double.parseDouble(fabrString);
                double celcius = 5.0 / 9.0 * (fahr - 32);
                System.out.println("F=" + fahr + ", C=" + celcius);
```

```
            }
        }
        System.out.println("ftoc exit!");
    }
}
```

他是一个简单且正常运行的主循环结构。我们可以应用 TEMPLATE METHOD 模式把这个基本结构从 foc 程序中分离出来。

我们把所有的通用代码放入一个抽象基类的实现方法中，这个实现方法完成这个通用算法，但是将所的实现细节都交付给该积累的抽象方法，修改后的如下：

```
/**
 * 描绘了一个通用的主循环应用程序
 *
 * @author echo
 * @version 1.0
 * @date 19-4-3 09:10
 */
public abstract class Application {
    private boolean isDone = false;
    protected abstract void init();
    protected abstract void idle();
    protected abstract void cleanup();

    public void run() {
        // 初始化
        init();
        // 执行
        while (!done()) {
            idle();
        }
        // 清除
        cleanup();
    }

    protected boolean done() {
        return isDone;
    }

    protected void setDone(boolean done) {
        isDone = done;
    }
}
```

现在，我们可以通过继承 Application 来重写 floc 类，只需要实现抽象方法即可，修改后的如下：

```
/**
 * @author echo
 * @version 1.0
 * @date 19-4-3 09:15
 */
public class FtocTemplateMethod extends Application {
```

```
    private InputStreamReader inputStreamReader;
    private BufferedReader bufferedReader;

    public static void main(String[] args) {
        new FtocTemplateMethod().run();
    }

    @Override
    protected void init() {
        inputStreamReader = new InputStreamReader(System.in);
        bufferedReader = new BufferedReader(inputStreamReader);
    }

    @Override
    protected void idle() {
        String fahrString = readLineAndReturnNullIfError();
        if (fahrString == null || fahrString.length() == 0) {
            setDone();
        } else {
            double fahr = Double.parseDouble(fahrString);
            double celcius = 5.0 / 9.0 * (fahr - 32);
            System.out.println("F=" + fahr + ", C=" + celcius);
        }
    }

    /**
     * 异常处理
     *
     * @return 读取结果
     */
    private String readLineAndReturnNullIfError() {
        String s;
        try {
            s = bufferedReader.readLine();
        } catch (IOException e) {
            e.printStackTrace();
            s = null;
        }
        return s;
    }

    @Override
    protected void cleanup() {
        System.out.println("ftoc exit!");
    }
}
```

这就是 TEMPLATE METHOD 模式的一个简单应用。在这个特定的简单的程序中，我们很容易理解但是我们真的需要这样吗？**其实，上面的这个是一个滥用模式的好礼自，在这个特定程序中，使用 TMPLATE METHOD 模式是荒谬的，他使得程序变得复杂庞大，他的意义不大，因为俄他的代价高他所带来的好处。**

冒泡排序

设计模式是很好的东西，但是并不意味着必须要经常使用它们，下面我们看一个设计模式稍微有用些例子。

```
/**
 * 冒泡排序
 *
 * @author echo
 * @version 1.0
 * @date 19-4-3 09:29
 */
public class BubbleSorter {
    static int operations = 0;

    public static int sort(int[] array) {
        operations = 0;
        if (array.length <= 1) {
            return operations;
        }
        for (int nextToLast = array.length - 2; nextToLast >= 0; nextToLast--) {
            for (int index = 0; index <= nextToLast; index++) {
                compareAndSwap(array, index);
            }
        }
        return operations;
    }

    private static void compareAndSwap(int[] array, int index) {
        if (array[index] > array[index + 1]) {
            swap(array, index);
        }
        operations++;
    }

    private static void swap(int[] array, int index) {
        int tmp = array[index];
        array[index] = array[index + 1];
        array[index + 1] = tmp;
    }
}
```

这是一个非常普通的冒泡排序算法，现在我们使用 TEMPLATE METHOD 模式，把冒泡排序算法分出来，放到一个抽象类中，如下：

```
/**
 * 抽象
 *
 * @author echo
 * @version 1.0
 * @date 19-4-3 09:35
 */
public abstract class BubbleSorterAbstract {
    private int operations = 0;
    protected int length = 0;
```

```
    protected int doSort() {
        operations = 0;
        if (length <= 1) {
            return operations;
        }
        for (int nextToLast = length - 2; nextToLast >= 0; nextToLast--) {
            for (int index = 0; index <= nextToLast; index++) {
                if (outOfOrder(index)) {
                    swap(index);
                }
                operations++;
            }
        }
        return operations;
    }

    protected abstract void swap(int index);
    protected abstract boolean outOfOrder(int index);
}
```

通过继承这个类，就可以完成一些变化的排序如下：

```
/**
 * int 类型冒泡排序
 *
 * @author echo
 * @version 1.0
 * @date 19-4-3 09:39
 */
public class IntBubbleSorter extends BubbleSorterAbstract {
    private int[] array = null;

    public int sort(int[] theArray) {
        array = theArray;
        length = array.length;
        return doSort();
    }

    @Override
    protected void swap(int index) {
        int tmp = array[index];
        array[index] = array[index + 1];
        array[index + 1] = tmp;
    }

    @Override
    protected boolean outOfOrder(int index) {
        return array[index] > array[index+1];
    }
}
```

```
/**
 * double 类型冒泡排序
 *
 * @author echo
 * @version 1.0
 * @date 19-4-3 09:41
 */
public class DoubleBubbleSorter extends BubbleSorterAbstract {
    private double[] array = null;
    public int sort(double[] theArray) {
        array = theArray;
        length = array.length;
        return doSort();
    }

    @Override
    protected void swap(int index) {
        double tmp = array[index];
        array[index] = array[index + 1];
        array[index + 1] = tmp;
    }

    @Override
    protected boolean outOfOrder(int index) {
        return array[index] > array[index+1];
    }
}
```

通过 TEMPLATE METHOD 模式的继承，把通用方法放在基类中，并且通过继承在不同的上下文中现该通用算法。但是继承是一种非常强的关系，派生类不得不和基类绑定在一起。例如，其他类型的序算法也要重新实现 outOfOrder 和 swap。

java 中有范型能够很好的解决上面的例子的问题。

不过，STRATEGY 提供了一种可选的方案。

STRATEGY 模式

华氏度转摄氏度

我们考虑刚才 华氏度转摄氏度 的例子，我们不再将通用的应用算法放进一个抽象基类中，而是放到个 ApplicationRunner 具体类中。我们把通用算法必须要调用的抽象方法定义在一个 Application 口中，再从这个接口中派生出 FlocStrategy，再传给具 ApplicationRunner，之后，就可以把具体作委托给接口去完成。

先来写 Application 接口

```
/**
 * @author echo
 * @version 1.0
 * @date 19-4-3 10:00
 */
```

```
public interface Application {
    public void init();
    public void idle();
    public void cleanup();
    public boolean done();
}
```

看看 ApplicationRunner

```
/**
 * @author echo
 * @version 1.0
 * @date 19-4-3 09:59
 */
public class ApplicationRunner {
    public Application itsAppliction = null;

    public ApplicationRunner(Application itsAppliction) {
        this.itsAppliction = itsAppliction;
    }

    public void run() {
        itsAppliction.init();
        while (!itsAppliction.done()) {
            itsAppliction.idle();
        }
        itsAppliction.cleanup();
    }
}
```

然后使用 STRATEGY 模式来进行改造 华氏度转摄氏度 的例子

```
/**
 * @author echo
 * @version 1.0
 * @date 19-4-3 10:02
 */
public class FtocStrategy implements Application {
    private InputStreamReader inputStreamReader;
    private BufferedReader bufferedReader;
    private boolean isDone = false;

    public static void main(String[] args) {
        new ApplicationRunner(new FtocStrategy()).run();
    }

    @Override
    public void init() {
        inputStreamReader = new InputStreamReader(System.in);
        bufferedReader = new BufferedReader(inputStreamReader);
    }
```

```
    @Override
    public void idle() {
        String fahrString = readLineAndReturnNullIfError();
        if (fahrString == null || fahrString.length() == 0) {
            isDone = true;
        } else {
            double fahr = Double.parseDouble(fahrString);
            double celcius = 5.0 / 9.0 * (fahr - 32);
            System.out.println("F=" + fahr + ", C=" + celcius);
        }
    }

    @Override
    public void cleanup() {
        System.out.println("ftoc exit!");
    }

    @Override
    public boolean done() {
        return isDone;
    }

    /**
     * 异常处理
     *
     * @return 读取结果
     */
    private String readLineAndReturnNullIfError() {
        String s;
        try {
            s = bufferedReader.readLine();
        } catch (IOException e) {
            e.printStackTrace();
            s = null;
        }
        return s;
    }

}
```

他和 TEMPLATE METHOD 实现的相比怎么样呢？很明显，STRATEGY 模式代价更高一点，涉及到多数量的类和间接层次，ApplicationRunner 委托指针的使用会造成比继承稍微多一点的运行时间和据空间开销。**但是另一方面，如果有许多不同的程序妖运行，就可以重用 ApplicationRunner 实例并把许多不同的 Application 实现传给他，从而减小了通用算法和该算法所控制的及具体细节之间耦合。**

冒泡排序

不过最烦人的问题是 STRATEGY 模式需要很多额外的类，我们来考虑一下使用 STRATEGY 模式实现冒泡排序 看看。

排序接口

```
/**
 * @author echo
 * @version 1.0
 * @date 19-4-3 10:13
 */
public interface SortHandle {
    public void swap(int index);
    public boolean outOfOrder(int index);
    public int length();
    public void setArray(Object array);
}
```

他的一个实现

```
/**
 * @author echo
 * @version 1.0
 * @date 19-4-3 10:13
 */
public class IntSortHandle implements SortHandle {
    private int[] array = null;

    @Override
    public void swap(int index) {
        int tmp = array[index];
        array[index] = array[index + 1];
        array[index + 1] = tmp;
    }

    @Override
    public boolean outOfOrder(int index) {
        return (array[index] > array[index + 1]);
    }

    @Override
    public int length() {
        return array.length;
    }

    @Override
    public void setArray(Object array) {
        this.array = (int[]) array;
    }
}
```

排序算法

```
/**
 * @author echo
 * @version 1.0
 * @date 19-4-3 10:12
 */
public class BubbleSorter {
```

```
    private int operations = 0;
    private int length = 0;
    private SortHandle itsSortHandle = null;

    public BubbleSorter(SortHandle itsSortHandle) {
        this.itsSortHandle = itsSortHandle;
    }

    public int sort(Object array) {
        itsSortHandle.setArray(array);
        length = itsSortHandle.length();
        operations = 0;
        if (length <= 1) {
            return operations;
        }
        for (int nextToLast = length - 2; nextToLast >= 0; nextToLast--) {
            for (int index = 0; index <= nextToLast; index++) {
                if (itsSortHandle.outOfOrder(index)) {
                    itsSortHandle.swap(index);
                }
                operations++;
            }
        }
        return operations;
    }
}
```

和 TEMPLATE METHOD 不同的是，接口实现 IntSortHandle 对 BubbleSorter 一无所知，布依赖任何实现方式。在 TEMPLATE METHOD 中，swap 和 outOfOrder 的实现依赖于冒泡排序算法，所他部分违反了 DIP，而 STRATEGY 总不包含这样的依赖。所以可以在 BubbleSorter 之外的其他任从 SortHandle 派生出来的类。

冒泡排序的变体

现在，我们创建冒泡排序的一个变体，如果他在一次对于数组的遍历中发现数组的元素已经是按序排的话，就提前结束。创建一个 QuickBubbleSorter

```
/**
 * @author echo
 * @version 1.0
 * @date 19-4-3 10:27
 */
public class QuickBubbleSorter {
    private int operations = 0;
    private int length = 0;
    private SortHandle itsSortHandle = null;

    public QuickBubbleSorter(SortHandle itsSortHandle) {
        this.itsSortHandle = itsSortHandle;
    }

    public int sort(Object array) {
        itsSortHandle.setArray(array);
```

```
        length = itsSortHandle.length();
        operations = 0;
        if (length <= 1) {
            return operations;
        }
        boolean thisPassInOrder = false;
        for (int nextToLast = length - 2; nextToLast >= 0 && !thisPassInOrder; nextToLast--) {
            thisPassInOrder = true;
            for (int index = 0; index <= nextToLast; index++) {
                if (itsSortHandle.outOfOrder(index)) {
                    itsSortHandle.swap(index);
                    thisPassInOrder = false;
                }
                operations++;
            }
        }
        return operations;
    }
}
```

QuickBubbleSorter 同样可以使用 IntSortHandle，或者任何其他从 SortHandle 派生出来的类。它全遵循 DIP 原则，从而允许每个具体实现都可以被多个不同的通用算法操纵。

相比起来，两个模式都可以 **用来分离高层的算法和底层的具体实现细节，都允许高层的算法独立于他具体实现细节重用。** 此外 STRATEGY 模式也允许具体实现细节独立于高层的算法重用，不过要以一额外的复杂性、内存以及运行时间开销作为代价。

FACADE 模式和 MEDIATOR 模式

尊贵的符号外表下，隐藏着卑劣的梦想。

FACADE 模式和 MEDIATOR 模式有着共同的目的，他们都把某种策略施加到另外一组对象上。

- FACADE 模式从上面施加策略，使用是明显且受限的。
    - 可以为一组具有复杂且全面的接口的对象提供一个简单且特定的接口，简单的说，就是应藏了体的内部细节，提供一个非常简单且特定的接口来完成。FACADE 对其就施加了策略。
- MEDIATOR 模式从下面施加策略，使用是不明显且不受限的。
    - 无需被施加者允许或者知晓。

SINGLETON 模式和 MONOSTATE 模式

这是对万物的祝福！除此之外再无其他

这章的两个模式，是强制对象单一性的模式。

SINGLETON 模式

SINGLETON 模式 是一个很简单的模式，通过一个一些测试用例来看看。

```
/**
 * 单例模式
 *
 * @author echo
 * @version 1.0
 * @date 19-4-3 11:08
 */
public class Singleton {
    private static Singleton theInstance = null;
    private Singleton(){}
    public static Singleton instance() {
        if (theInstance == null) {
            theInstance = new Singleton();
        }
        return theInstance;
    }
}
```

测试

```
/**
 * @author echo
 * @version 1.0
 * @date 19-4-3 11:07
 */
class TestSimpleSingleton {
    @Test
    void testCreateSingleton() {
        Singleton s1 = Singleton.instance();
        Singleton s2 = Singleton.instance();
        assert s1 == s2;
    }

    @Test
    void testNoPublicConstructors() throws ClassNotFoundException {
        Class<?> singleton = Class.forName("four.singleton.Singleton");
        Constructor[] constructors = singleton.getConstructors();
        assert constructors.length == 0;
    }
}
```

可以看出，通过私有化构造函数，实现了 SINGLETON 模式

好处：

1. 跨平台：使用合适的中间件，可以把 SINGLETON 模式扩展为跨多个 JVM 和多个计算机工作。

2. 适用于任何类：只要把一个类的构造函数私有化，并且增加相应的静态函数和变量，就可以把这个变成 SINGLETON。

3. 可以通过派生创建：给定一个类，可以创建他的一个 SINGLETON 子类。

4. 延迟求值：如果 SINGLETON 从未使用过，那么就不会创建他。

代价：

1. 摧毁方法未定义：没有好的摧毁方法去摧毁一个 SINGLETON，或者解除其职责，可能会同时存在个实例。

2. 不能继承：从 SINGLETON 类派生出来的类不是 SINGLETON 的。

3. 效率问题：每次调用 instance 都会执行 if 语句

4. 不透明性：SINGLETON 使用者指定他们在使用一个 SINGLETON，因为他们必须调用 instance 法。

MONOSTATE 模式

MONOSTATE 模式是获取单一对象的另外一种方法。它使用了一种完全不同的工作机制，看看下面一个例子：

```
/**
 * @author echo
 * @version 1.0
 * @date 19-4-3 11:20
 */
public class Monostate {
    private static int itsX = 0;
    public Monostate(){}
    public void setX(int x) {
        itsX = x;
    }
    public int getX() {
        return itsX;
    }
}
```

测试用例

```
/**
 * @author echo
 * @version 1.0
 * @date 19-4-3 11:21
 */
class TestMonostate {
    @Test
    void testInstance() {
        Monostate monostate = new Monostate();
        for (int x = 0; x < 10; x++) {
            monostate.setX(x);
            assert x == monostate.getX();
        }
    }

    @Test
    void testInstanceBehaveAsOne() {
        Monostate m1 = new Monostate();
        Monostate m2 = new Monostate();
```

```
        for (int x = 0; x < 10; x++) {
            m1.setX(x);
            assert x == m2.getX();
        }
    }
}
```

两个对象共享相同的变量，itsX 是静态的，但是方法不是静态的，这一点很重要。无论创建多少个 Monostate 对象的实例，他们都表现得**像一个对象一样，甚至把当前的所有实例都销毁或者解除职责，不会丢失数据。**

好处：

1. 透明性：使用 MONOSTATE 对象和使用常规对象没有什么区别，使用者不知道对象是 MONOSTATE。
2. 可派生性：MONOSTATE 的派生类都是 MONOSTATE，事实上，MONOSTATE 的所有派生类都同一个 MONOSTATE 的一部分，他们共享相同的静态变量。
3. 多态性：由于 MONOSTATE 的方法不是静态的，所以可以在派生类中总重写。

代价

1. 不可转换性：不能通过派生类把常规类转换成 MONOSTATE 类。
2. 效率问题：因为 MONOSTATE 是真正的对象，所以会导致许多的创建和销毁开销。
3. 内存占用：即使从未使用过 MONOSTATE，他的变量也要占据内存空间。
4. 平台局限性：MONOSTATE 不能跨多个 JVM 或者多个平台工作。

两个模式，一个关注行为，一个。

- SINGLETON 模式关注结构，强制结构上的单一性。防止创建出多个对象实例。如果希望通过派生约束一个现存类，并且不介意他的所有调用这都必须要调用 instance() 方法来获取访问权，那么他是合适的。
- MONOSTATE 模式关注行为，强制行为上的单一性，而没有强加结构方面的限制。如果希望类的一性本质对使用者透明，或者希望使用单一对象的多态派生对象，那么他是最合适的。

MONOSTATE 的测试用例对 SINGLETON 类是有效的，但是 SINGLETON 的测试用例却不远不适用 MONOSTATE 类。

NULL OBJECT 模式

残缺即是完美，冷淡即是虚无，死亡即是圆满，没有即是更多。

java 中可能最常见的一个异常就是空指针异常了，使用 if 和 try/catch 都不是很优雅，现在我们俩看 NULL OBJECT 模式的例子。

先编写测试用例

```
/**
 * @author echo
 * @version 1.0
 * @date 19-4-3 11:42
```

```
 */
class TestEmployee {

    @Test
    void testNull() {
        // 不存在的数据
        Employee employee = DB.getEmployee("Bob");
        assert employee.isTimeToPay(new Date()) || Employee.NULL == employee;
    }
}
```

书写接口

```
public interface Employee {
    public boolean isTimeToPay(Date payDate);
    public void pay();
    public static final Employee NULL = new Employee() {
        @Override
        public boolean isTimeToPay(Date payDate) {
            return false;
        }

        @Override
        public void pay() {

        }
    };
}


/**
 * @author echo
 * @version 1.0
 * @date 19-4-3 11:43
 */
public class DB {
    public static Employee getEmployee(String name) {
        return Employee.NULL;
    }
}
```

我们通过使无效的数据成为一个匿名内部类是一个确保只有单一实例的方法，实际上并不存在。

我突然想到了 jdk 8 的 optional ~

总结

相比来说，前半部分较难，后面比较简单。周六看完的，但是周三才实现部分代码，理解提高了一点作业实在太多=-=脑壳疼，加油吧！