# 课程目标

1、掌握命令模式在源码中的应用，知其所以然。

# 内容定位

听说过迭代器模式，但并不知其所以然的人群。

2、

3、自己手写迭代器

4、掌握迭代器模式和命令模式在源码中的应用，知其所以然。

# 内容定位

听说过迭代器模式和命令模式，但并不知其所以然的人群。

# 迭代器模式

迭代器模式（Iterator Pattern）又称为游标模式(Cursor Pattern)，它提供一种顺序访问集合/容器对象元素的方法，而又无须暴露集合内部表示。迭代器模式可以为不同的容器提供一致的遍历行为，而不用关心容器内容元素组成结构，属于行为型模式。

> 原文：Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
>
> 解释：提供一种顺序访问集合/容器对象元素的方法，而又无须暴露集合内部表示。

迭代器模式的本质是抽离集合对象迭代行为到迭代器中，提供一致访问接口。

**迭代器模式的应用场景**

迭代器模式在我们生活中应用的得也比较广泛，比如物流系统中的传送带，不管传送的是什么物品，都被打包成一个一个的箱子并且有一个统一的二维码。这样我们不需要关心箱子里面是啥，我们在分发时只需要一个一个检查发送的目的地即可。再比如，我们平时乘坐交通工具，都是统一刷卡或者刷脸进站，而不需要关心是男性还是女性、是残疾人还是正常人等个性化的信息。

寄件迭代分发

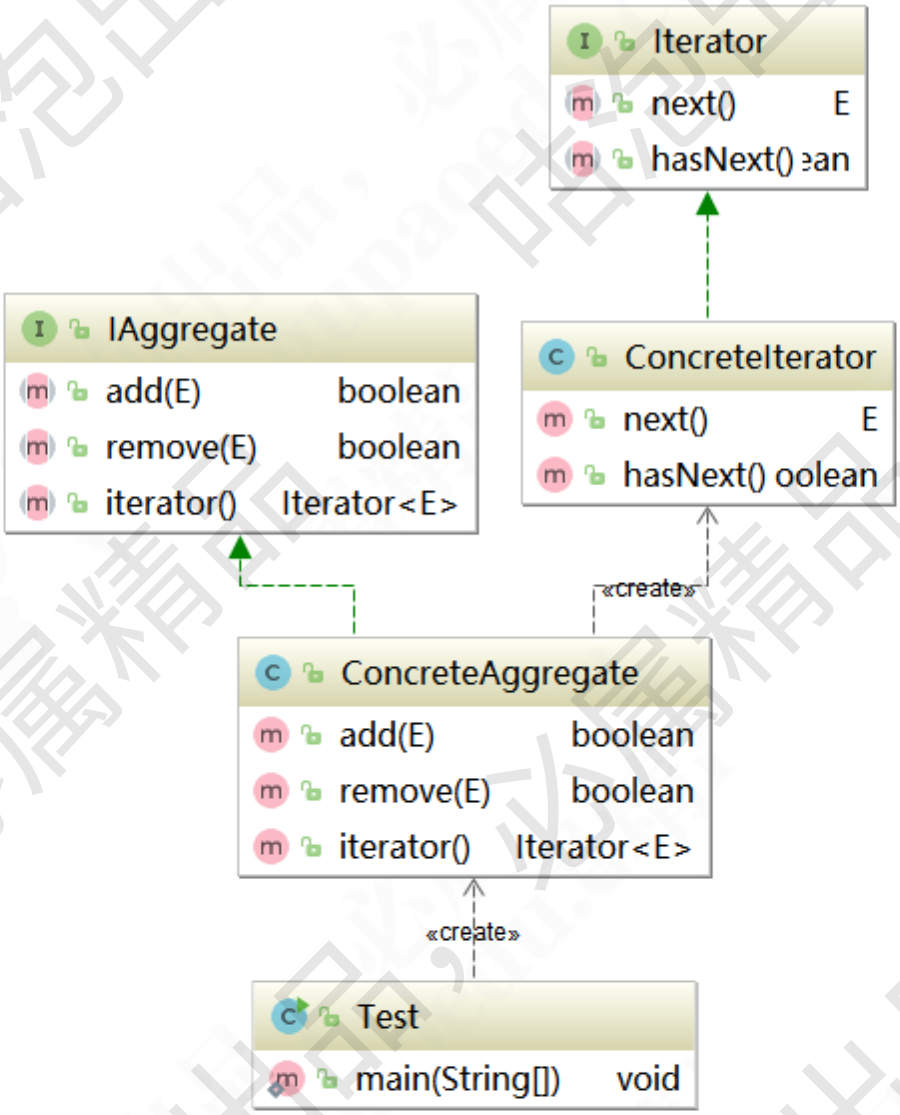

刷脸（刷卡）检票进站

我们把多个对象聚在一起形成的总体称之为集合(Aggregate)，集合对象是能够包容一组对象的容器对象。不同的集合其内部元素的聚合结构可能不同，而迭代器模式屏蔽了内部元素获取细节，为外部提供一致的元素访问行为，解耦了元素迭代与集合对象间的耦合，并且通过提供不同的迭代器，可以为同个集合对象提供不同顺序的元素访问行为，扩展了集合对象元素迭代功能，符合开闭原则。迭代器模式适用于以下场景：

1、访问一个集合对象的内容而无需暴露它的内部表示；

2、为遍历不同的集合结构提供一个统一的访问接口。

首先来看下迭代器模式的通用 UML 类图：




从 UML 类图中，我们可以看到，迭代器模式主要包含三种角色：

抽象迭代器（Iterator）：抽象迭代器负责定义访问和遍历元素的接口；

具体迭代器（ConcreteIterator）：提供具体的元素遍历行为；

抽象容器（Aggregate）：负责定义提供具体迭代器的接口；

具体容器（ConcreteAggregate）：创建具体迭代器。

## 手写自定义的迭代器

总体来说，迭代器模式还是非常简单的。我们还是以课程为例，下面我们自己创建一个课程的集合，集合中的每一个元素就是课程对象，然后自己手写一个迭代器，将每一个课程对象的

信息读出来。首先创建集合元素课程 Course 类：

```
public class Course {
    private String name;

    public Course(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

然后创建自定义迭代器 Iterator 接口：

```
public interface Iterator<E> {

    E next();

    boolean hasNext();

}
```

然后创建自定义的课程的集合 CourseAggregate 接口：

```
public interface CourseAggregate {

    void add(Course course);

    void remove(Course course);

    Iterator<Course> iterator();
}
```

然后，分别实现迭代器接口和集合接口，创建 IteratorImpl 实现类：

```
public class IteratorImpl<E> implements Iterator<E> {

    private List<E> list;
    private int cursor;
    private E element;
    public IteratorImpl(List list){
        this.list = list;
    }


    public E next() {
        System.out.print("当前位置" + cursor + ": ");
        element = list.get(cursor);
        cursor ++;
        return element;
    }


    public boolean hasNext(){
        if(cursor > list.size() - 1){
            return false;
        }
        return true;
    }
}
```

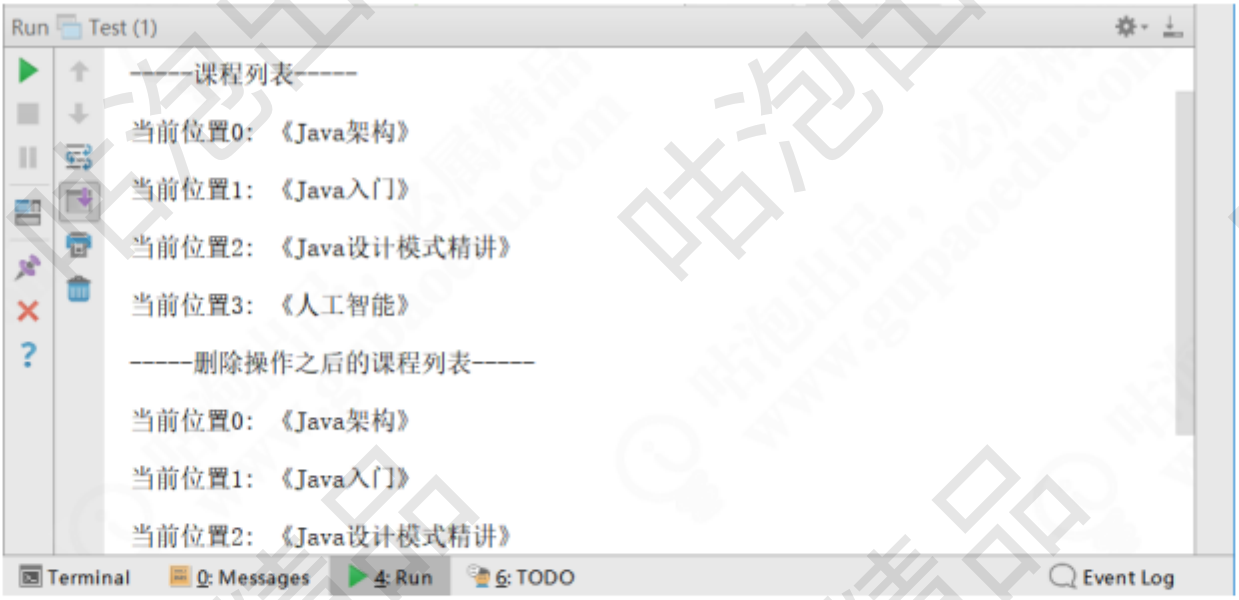
创建课程集合 CourseAggregateImpl 实现类：

```
public class CourseAggregateImpl implements CourseAggregate {

    private List courseList;

    public CourseAggregateImpl() {
        this.courseList = new ArrayList();
    }

    public void add(Course course) {
        courseList.add(course);
    }

    public void remove(Course course) {
        courseList.remove(course);
    }

    public Iterator<Course> iterator() {
        return new IteratorImpl(courseList);
    }
}
```

然后，编写客户端代码：

```
public static void main(String[] args) {
        Course java = new Course("Java 架构");
        Course javaBase = new Course("Java 入门");
        Course design = new Course("Java 设计模式精讲");
        Course ai = new Course("人工智能");


        CourseAggregate courseAggregate = new CourseAggregateImpl();

        courseAggregate.add(java);
        courseAggregate.add(javaBase);
        courseAggregate.add(design);
        courseAggregate.add(ai);

        System.out.println("-----课程列表-----");
        printCourses(courseAggregate);

        courseAggregate.remove(ai);

        System.out.println("-----删除操作之后的课程列表-----");
        printCourses(courseAggregate);
    }


    public static void printCourses(CourseAggregate courseAggregate){
        Iterator<Course> iterator = courseAggregate.iterator();
        while(!iterator.hasNext()){
            Course course = iterator.next();
            System.out.println("《" + course.getName() + "》");
        }
    }
```

运行结果如下：




看到这里，小伙伴们肯定会有一种似曾相识的感觉，让人不禁想起我们每天都在用的 JDK 自带的结合迭代器。下面我们就来看看源码中是如何运用迭代器的。

## 迭代器模式在源码中的体现

先来看 JDK 中大家非常熟悉的 Iterator 源码：

```
public interface Iterator<E> {
    boolean hasNext();

    E next();

    default void remove() {
        throw new UnsupportedOperationException("remove");
    }
    default void forEachRemaining(Consumer<? super E> action) {
        Objects.requireNonNull(action);
        while (hasNext())
            action.accept(next());
    }
}
```

从上面代码中，我们看到两个主要的方法定义 hasNext()和 next()方法，和我们自己写的完全一致。

另外，从上面的代码中，我们看到 remove()方法实现似曾相识。其实是在组合模式中我们见到过。迭代器模式和组合模式，两者似乎存在一定的相似性。组合模式解决的是统一树形结构各层次访问接口，迭代器模式解决的是统一各集合对象元素遍历接口。虽然他们的适配场景不同，但核心理念是相通的。

下面接着来看 Iterator 的实现类，其实在我们常用的 ArrayList 中有一个内部实现类 Itr，它就实现了 Iterator 接口：

```
public class ArrayList<E> extends AbstractList<E>
       implements List<E>, RandomAccess, Cloneable, java.io.Serializable {
       ...
       private class Itr implements Iterator<E> {
             int cursor;       // index of next element to return
             int lastRet = -1; // index of last element returned; -1 if no such
             int expectedModCount = modCount;

             public boolean hasNext() {
                  return cursor != size;
             }

             @SuppressWarnings("unchecked")
             public E next() {
                  checkForComodification();
                  int i = cursor;
                  if (i >= size)
                      throw new NoSuchElementException();
                  Object[] elementData = ArrayList.this.elementData;
                  if (i >= elementData.length)
                      throw new ConcurrentModificationException();
                  cursor = i + 1;
                  return (E) elementData[lastRet = i];
             }
            ...
       }
      ...
}
```

其中 hasNext()方法和 next()方法实现也非常简单，我们继续往下看在 ArrayList 内部还有几个迭代器对 Itr 进行了进一步扩展，首先看 ListItr：

```
private class ListItr extends Itr implements ListIterator<E> {
       ListItr(int index) {
           super();
           cursor = index;
       }

       public boolean hasPrevious() {
           return cursor != 0;
       }

       public int nextIndex() {
           return cursor;
       }

       public int previousIndex() {
           return cursor - 1;
       }
      ...
}
```

它增加了 hasPrevious()方法是否还有上一个等这样的判断。另外还有 SubList 对子集合的

迭代处理。

当然，迭代器模式在 MyBatis 中也是必不可少的，来看一个 DefaultCursor 类：

```
public class DefaultCursor<T> implements Cursor<T> {

   ...

   private final CursorIterator cursorIterator = new CursorIterator();
}
```

首先它实现了 Cursor 接口，而且定义了一个成员变量 cursorIterator ，我继续查看 CursorIterator 的源代码发现，它是 DefaultCursor 的一个内部类，并且实现了 JDK 中的 Iterator 接口。

**迭代器模式的优缺点**

优点：

1、多态迭代：为不同的聚合结构提供一致的遍历接口，即一个迭代接口可以访问不同的集合对象；

2、简化集合对象接口：迭代器模式将集合对象本身应该提供的元素迭代接口抽取到了迭代器中，使集合对象无须关心具体迭代行为；

3、元素迭代功能多样化：每个集合对象都可以提供一个或多个不同的迭代器，使的同种元素聚合结构可以有不同的迭代行为；

4、解耦迭代与集合：迭代器模式 封装了具体的迭代算法，迭代算法的变化，不会影响到集合对象的架构。

缺点：

1、对于比较简单的遍历（像数组或者有序列表），使用迭代器方式遍历较为繁琐。

在日常开发当中，我们几乎不会自己写迭代器。除非我们需要定制一个自己实现的数据结构对应的迭代器，否则，开源框架提供给我们的 API 完全够用。

# 命令模式

命令模式（Command Pattern）是对命令的封装，每一个命令都是一个操作：请求的一方发出请求要求执行一个操作；接收的一方收到请求，并执行操作。命令模式解耦了请求方和接收方，请求方只需请求执行命令，不用关心命令是怎样被接收，怎样被操作以及是否被执行…等。命令模式属于行为型模式。

```
原文：Encapsulate a request as an object,thereby letting you parameterize clients with different requests,queue or log requests,and support undoable operations.
解释：将一个请求封装成一个对象，从而让你使用不同的请求把客户端参数化，对请求排队或者记录请求日志，可以提供命令的撤销和恢复功能。
```

在软件系统中，行为请求者与行为实现者通常是一种紧耦合关系，因为这样的实现简单明了。但紧耦合关系缺乏扩展性，在某些场合中，当需要为行为进行记录，撤销或重做等处理时，只能修改源码。而命令模式通过为请求与实现间引入一个抽象命令接口，解耦了请求与实现，并且中间件是抽象的，它可以有不同的子类实现，因此其具备扩展性。所以，命令模式的本质是解耦命令请求与处理。

**命令模式的应用场景**

当系统的某项操作具备命令语义时，且命令实现不稳定（变化），那么可以通过命令模式解耦请求与实现，利用抽象命令接口使请求方代码架构稳定，封装接收方具体命令实现细节。接收方与抽象命令接口呈现弱耦合（内部方法无需一致），具备良好的扩展性。命令模式适用于以下应用场景：

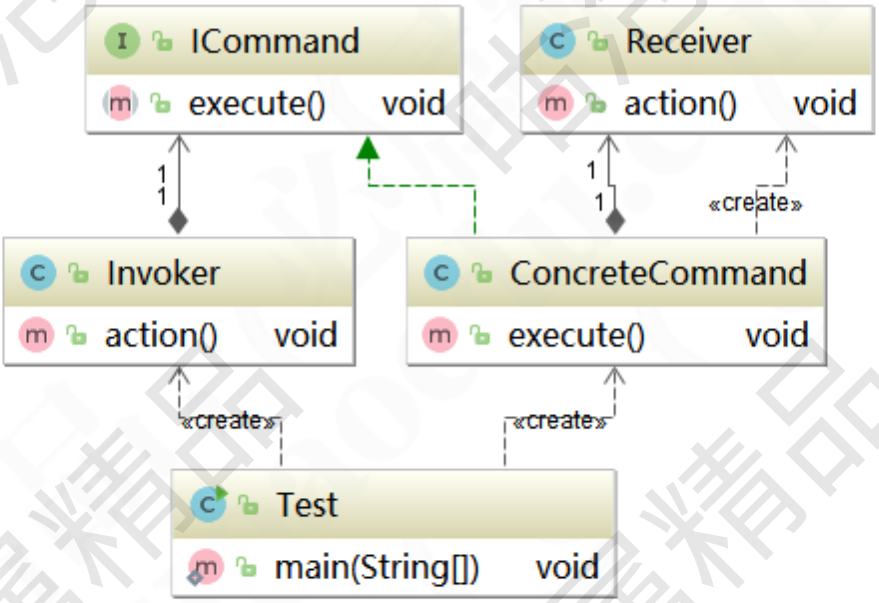
1、现实语义中具备“命令”的操作（如命令菜单，shell 命令…）；

2、请求调用者和请求的接收者需要解耦，使得调用者和接收者不直接交互；

3、需要抽象出等待执行的行为，比如撤销(Undo)操作和恢复(Redo)等操作；

4、需要支持命令宏（即命令组合操作）。

首先看下命令模式的通用 UML 类图：




从 UML 类图中，我们可以看到，命令模式 主要包含四种角色：

接收者角色（Receiver）：该类负责具体实施或执行一个请求；

命令角色（Command）：定义需要执行的所有命令行为；

具体命令角色( ConcreteCommand ):该类内部维护一个接收者( Receiver ),在其 execute() 方法中调用 Receiver 的相关方法；

请求者角色（Invoker）：接收客户端的命令，并执行命令。

从命令模式的 UML 类图中，其实可以很清晰地看出：Command 的出现就是作为 Receiver 和 Invoker 的中间件，解耦了彼此。而之所以引入 Command 中间件，我觉得是以下两方面原因：

解耦请求与实现：即解耦了 Invoker 和 Receiver，因为在 UML 类图中，Invoker 是一个具体的实现，等待接收客户端传入命令( 即 Invoker 与客户端耦合 )，Invoker 处于业务逻辑区域，应当是一个稳定的结构。而 Receiver 是属于业务功能模块，是经常变动的，如果没有 Command，则 Invoker 紧耦合 Receiver，一个稳定的结构依赖了一个不稳定的结构，就会导致整个结构都

不稳定了。这也就是 Command 引入的原因：不仅仅是解耦请求与实现，同时稳定（Invoker）依赖稳定（Command），结构还是稳定的。

扩展性增强：扩展性体现在两个方面：

1、Receiver 属于底层细节，可以通过更换不同的 Receiver 达到不同的细节实现；

2、Command 接口本身就是抽象的，本身就具备扩展性；而且由于命令对象本身就具备抽象，如果结合装饰器模式，功能扩展简直如鱼得水。

注：在一个系统中，不同的命令对应不同的请求，也就是说无法把请求抽象化，因此命令模式中的 Receiver 是具体实现；但是如果在某一个模块中，可以对 Receiver 进行抽象，其实这就变相使用到了桥接模式（Command 类具备两个变化的维度：Command 和 Receiver），这样子的扩展性会更加优秀。

举个生活中的例子，相信 80 后的小伙伴应该都经历过普及黑白电视机的那个年代。黑白电视机要换台那简直不容易，需要人跑上前去用力掰动电视机上那个切换频道的旋钮，一顿“啪啪啪”折腾下来才能完成一次换台。如今时代好了，我们只需躺沙发上按一下遥控器就完成了换台。这就是用到了命令模式，将换台命令和换台处理进行了分离。

另外，就是餐厅的点菜单，一般是后厨先把所有的原材料组合配置好了，客户用餐前只需要点菜即可，将需求和处理进行了解耦。

遥控器


餐厅点菜单

## 命令模式在业务场景中的应用

假如我们自己开发一个播放器，播放器有播放功能、有拖动进度条功能、停止播放功能、暂停功能，我们自己去操作播放器的时候并不是直接调用播放器的方法，而是通过一个控制条去

传达指令给播放器内核，那么具体传达什么指令，会被封装为一个一个的按钮。那么每个按钮的就相当于是对一条命令的封装。用控制条实现了用户发送指令与播放器内核接收指令的解耦。

下面来看代码，首先创建播放器内核 GPlayer 类：

```
public class GPlayer {
    public void play(){
        System.out.println("正常播放");
    }

    public void speed(){
        System.out.println("拖动进度条");
    }

    public void stop(){
        System.out.println("停止播放");
    }

    public void pause(){
        System.out.println("暂停播放");
    }
}
```

创建命令接口 IAction 类：

```
public interface IAction {
    void execute();
}
```

然后分别创建操作播放器可以接受的指令，播放指令 PlayAction 类：

```
public class PlayAction implements IAction {
    private GPlayer gplayer;

    public PlayAction(GPlayer gplayer) {
        this.gplayer = gplayer;
    }

    public void execute() {
        gplayer.play();
    }
}
```

暂停指令 PauseAction 类：

```
public class PauseAction implements IAction {
    private GPlayer gplayer;

    public PauseAction(GPlayer gplayer) {
        this.gplayer = gplayer;
    }

    public void execute() {
        gplayer.pause();
    }
}
```

拖动进度条指令 SpeedAction 类：

```
public class SpeedAction implements IAction {
```

```
    private GPlayer gplayer;

    public SpeedAction(GPlayer gplayer) {
        this.gplayer = gplayer;
    }

    public void execute() {
        gplayer.speed();
    }
}
```

停止播放指令 StopAction 类：

```
public class StopAction implements IAction {
    private GPlayer gplayer;

    public StopAction(GPlayer gplayer) {
        this.gplayer = gplayer;
    }

    public void execute() {
        gplayer.stop();
    }
}
```

最后，创建控制条 Controller 类：

```
public class Controller {
    private List<IAction> actions = new ArrayList<IAction>();
    public void addAction(IAction action){
        actions.add(action);
    }

    public void execute(IAction action){
        action.execute();
    }

    public void executes(){
        for(IAction action : actions){
            action.execute();
        }
        actions.clear();
    }
}
```

从上面代码来看，控制条可以执行单条命令，也可以批量执行多条命令。下面来看客户端测试代码：

```
    public static void main(String[] args) {

        GPlayer player = new GPlayer();
        Controller controller = new Controller();
        controller.execute(new PlayAction(player));

        controller.addAction(new PauseAction(player));
        controller.addAction(new PlayAction(player));
        controller.addAction(new StopAction(player));
        controller.addAction(new SpeedAction(player));
        controller.executes();
    }
```

由于控制条已经与播放器内核解耦了，以后如果想扩展新命令，只需增加命令即可，控制条的结构无需改动。

## 命令模式在源码中的体现

首先来看 JDK 中的 Runnable 接口，实际上 Runnable 就相当于是命令的抽象，只要是实现了 Runnable 接口的类都被认为是一个线程。

```
public interface Runnable {
    public abstract void run();
}
```

实际上调用线程的 start()方法之后，就有资格去抢 CPU 资源，而不需要我们自己编写获得 CPU 资源的逻辑。而线程抢到 CPU 资源后，就会执行 run()方法中的内容，用 Runnable 接口把用户请求和 CPU 执行进行了解耦。

然后，再看一个大家非常熟悉的 junit.framework.Test 接口：

```
package junit.framework;

public interface Test {
    public abstract int countTestCases();

    public abstract void run(TestResult result);
}
```

Test 接口中有两个方法,第一个是 countTestCases()方法用来统计当前需要执行的测试用例总数。第二个是 run()方法就是用来执行具体的测试逻辑，其参数 TestResult 是用来返回测试结果的。实际上我们在平时编写测试用例的时候，只需要实现 Test 接口即便认为就是一个测试用例，那么在执行的时候就会自动识别。实际上我们平时通常做法都是继承 TestCase 类，我们不妨来看一下 TestCase 的源码：

```
public abstract class TestCase extends Assert implements Test {
    ...
    public void run(TestResult result) {
       result.run(this);
    }
    ...
}
```

实际上 TestCase 类它也实现了 Test 接口。我们继承 TestCase 类，相当于也实现了 Test 接口，自然也就会被扫描成为一个测试用例。

## 命令模式的优缺点

优点：

1、通过引入中间件（抽象接口），解耦了命令请求与实现；

2、扩展性良好，可以很容易地增加新命令；

3、支持组合命令，支持命令队列；

4、可以在现有命令的基础上，增加额外功能（比如日志记录…，结合装饰器模式更酸爽）。

缺点：

1、具体命令类可能过多；

2、命令模式的结果其实就是接收方的执行结果，但是为了以命令的形式进行架构，解耦请求与实现，引入了额外类型结构（引入了请求方与抽象命令接口），增加了理解上的困难（不过这也是设计模式带来的一个通病，抽象必然会引入额外类型；抽象肯定比紧密难理解）。