课程目标

1、了解迭代器模式和命令的应用场景。

2、自己手写迭代器

3、掌握迭代器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。

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

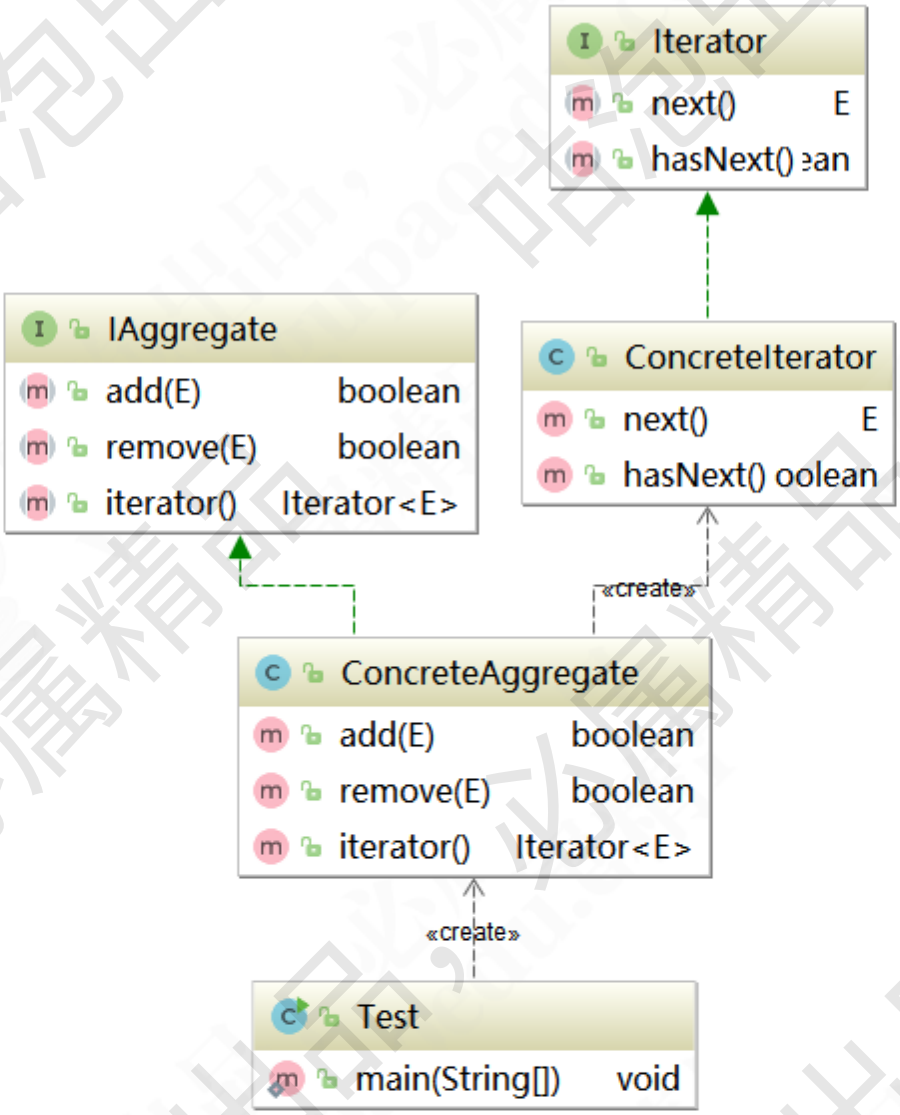
刷脸（刷卡）检票进站

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;
    }
}
```

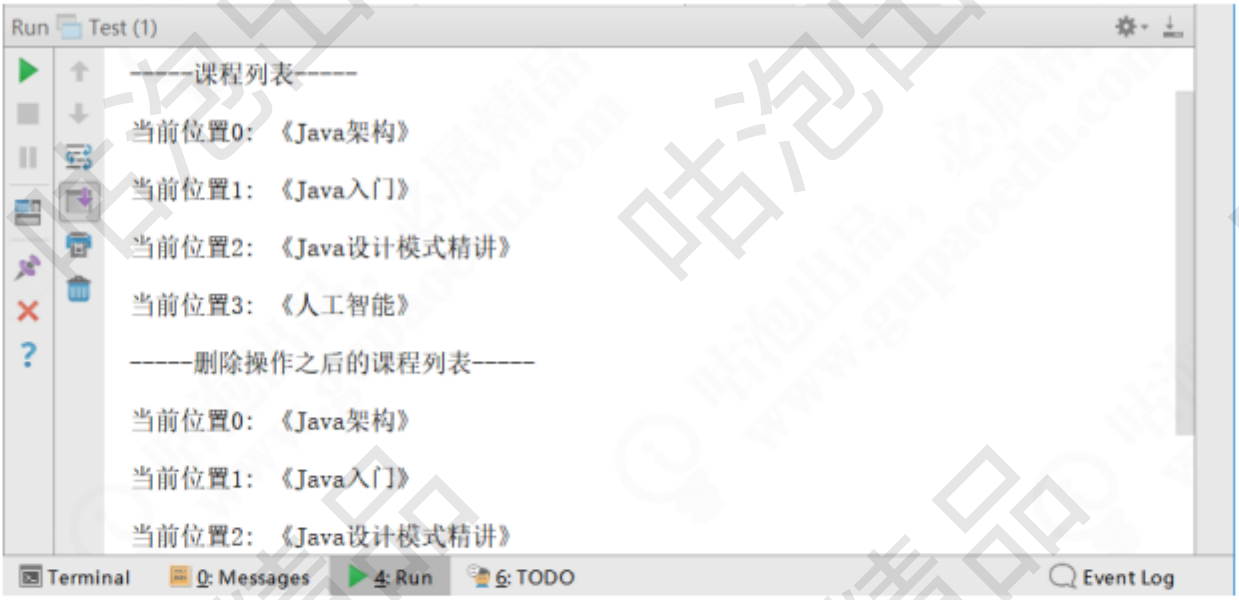
创建课程集合 CourseAggregateImpl 实现类：

```
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。