# 迭代子模式

> *整理自：《java与模式》之迭代子模式*

在阎宏博士的《JAVA与模式》一书中开头是这样描述迭代子（Iterator）模式的：

**迭代子模式又叫游标(Cursor)模式，是对象的行为模式。迭代子模式可以顺序地访问一个聚集中的元素而不必暴露聚集的内部表象（internal representation）。**

---

## 聚集和JAVA聚集

多个对象聚在一起形成的总体称之为聚集(Aggregate)，聚集对象是能够包容一组对象的容器对象。聚集依赖于聚集结构的抽象化，具有复杂化和多样性。数组就是最基本的聚集，也是其他的JAVA聚集对象的设计基础。

JAVA聚集对象是实现了共同的java.util.Collection接口的对象，是JAVA语言对聚集概念的直接支持。从1.2版开始，JAVA语言提供了很多种聚集，包括Vector、ArrayList、HashSet、HashMap、Hashtable等，这些都是JAVA聚集的例子。

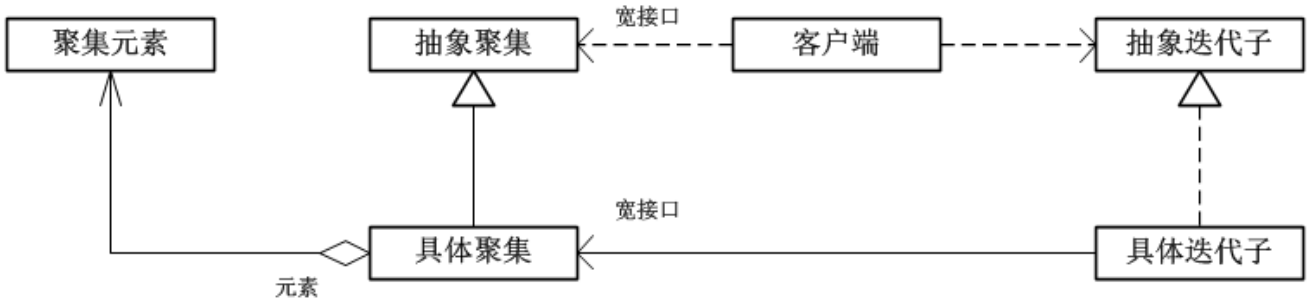
## 迭代子模式的结构

迭代子模式有两种实现方式，分别是**白箱聚集与外禀迭代子**和**黑箱聚集于内禀迭代子**。

### 白箱聚集与外禀迭代子

如果一个聚集的接口提供了可以用来修改聚集元素的方法，这个接口就是所谓的**宽接口**。

如果聚集对象为所有对象提供同一个接口，也就是宽接口的话，当然会满足迭代子模式对迭代子对象的要求。但是，这样会破坏对聚集对象的封装。这种提供宽接口的聚集叫做**白箱聚集**。聚集对象向外界提供同样的宽接口，如下图所示：

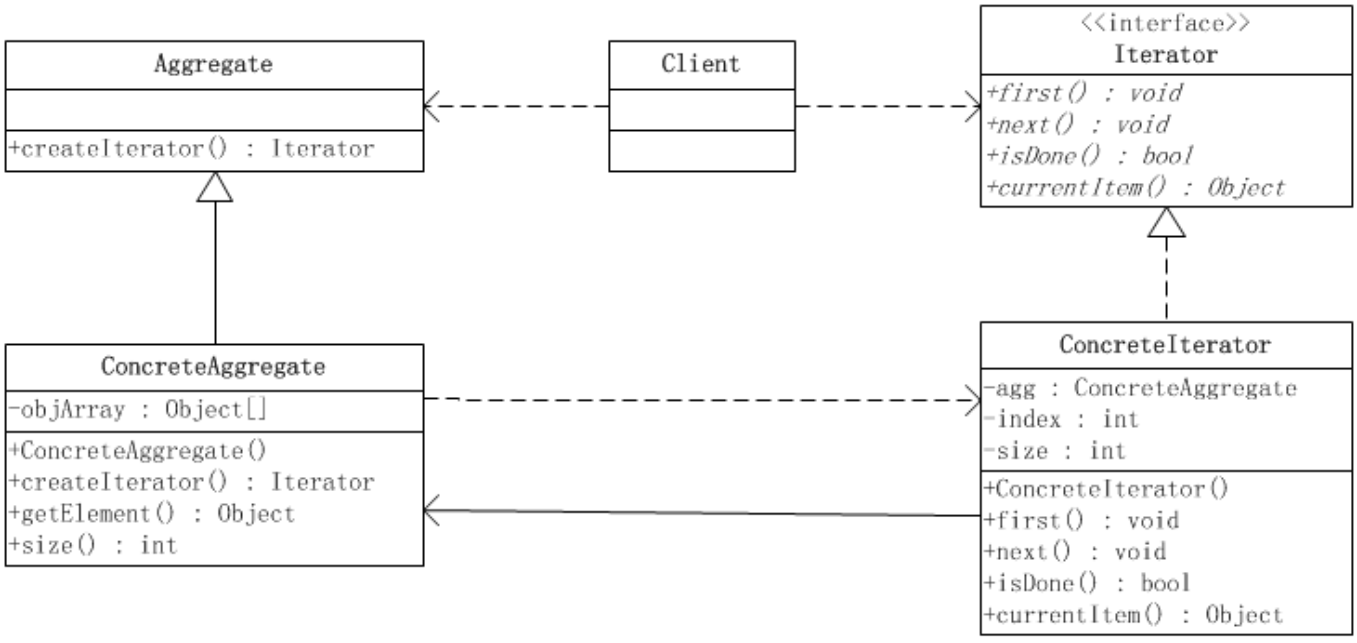



由于聚集自己实现迭代逻辑，并向外部提供适当的接口，使得迭代子可以从外部控制聚集元素的迭代过程。这样一来迭代子所控制的仅仅是一个游标而已，这种迭代子叫做**游标迭代子（Cursor Iterator）**。由于迭代子是在聚集结构之外的，因此这样的迭代子又叫做**外禀迭代子（Extrinsic Iterator）**。

现在看一看白箱聚集与外禀迭代子的实现。一个白箱聚集向外界提供访问自己内部元素的接口（称作遍历方法或者Traversing Method），从而使外禀迭代子可以通过聚集的遍历方法实现迭代功能。

因为迭代的逻辑是由聚集对象本身提供的，所以这样的外禀迭代子角色往往仅仅保持迭代的游标位置。

一个典型的由白箱聚集与外禀迭代子组成的系统如下图所示，在这个实现中具体迭代子角色是一个外部类，而具体聚集角色向外界提供遍历聚集元素的接口。




迭代子模式涉及到以下几个角色：

- **抽象迭代子(Iterator)角色：**此抽象角色定义出遍历元素所需的接口。

- **具体迭代子(ConcreteIterator)角色：**此角色实现了Iterator接口，并保持迭代过程中的游标位置。
- **聚集(Aggregate)角色：**此抽象角色给出创建迭代子(Iterator)对象的接口。
- **具体聚集(ConcreteAggregate)角色：**实现了创建迭代子(Iterator)对象的接口，返回一个合适的具体迭代子实例。
- **客户端(Client)角色：**持有对聚集及其迭代子对象的引用，调用迭代子对象的迭代接口，也有可能通过迭代子操作聚集元素的增加和删除。

**源代码**

抽象聚集角色类，这个角色规定出所有的具体聚集必须实现的接口。迭代子模式要求聚集对象必须有一个工厂方法，也就是createIterator()方法，以向外界提供迭代子对象的实例。

```
public abstract class Aggregate {
    /**
     * 工厂方法，创建相应迭代子对象的接口
     */
    public abstract Iterator createIterator();
}
```

具体聚集角色类，实现了抽象聚集角色类所要求的接口，也就是createIterator()方法。此外，还有方法getElement()向外界提供聚集元素，而方法size()向外界提供聚集的大小等。

```
public class ConcreteAggregate extends Aggregate {

    private Object[] objArray = null;
    /**
     * 构造方法，传入聚合对象的具体内容
     */
    public ConcreteAggregate(Object[] objArray){
        this.objArray = objArray;
    }

    @Override
    public Iterator createIterator() {

        return new ConcreteIterator(this);
    }
```

```
    /**
     * 取值方法：向外界提供聚集元素
     */
    public Object getElement(int index){

        if(index < objArray.length){
            return objArray[index];
        }else{
            return null;
        }
    }
    /**
     * 取值方法：向外界提供聚集的大小
     */
    public int size(){
        return objArray.length;
    }
}
```

抽象迭代子角色类

```
public interface Iterator {
    /**
     * 迭代方法：移动到第一个元素
     */
    public void first();
    /**
     * 迭代方法：移动到下一个元素
     */
    public void next();
    /**
     * 迭代方法：是否为最后一个元素
     */
    public boolean isDone();
    /**
     * 迭代方法：返还当前元素
     */
    public Object currentItem();
}
```

具体迭代子角色类

```
public class ConcreteIterator implements Iterator {
    //持有被迭代的具体的聚合对象
    private ConcreteAggregate agg;
    //内部索引，记录当前迭代到的索引位置
    private int index = 0;
    //记录当前聚集对象的大小
    private int size = 0;

    public ConcreteIterator(ConcreteAggregate agg){
        this.agg = agg;
        this.size = agg.size();
        index = 0;
    }
    /**
     * 迭代方法：返还当前元素
     */
    @Override
    public Object currentItem() {
        return agg.getElement(index);
    }
    /**
     * 迭代方法：移动到第一个元素
     */
    @Override
    public void first() {

        index = 0;
    }
    /**
     * 迭代方法：是否为最后一个元素
     */
    @Override
    public boolean isDone() {
        return (index >= size);
    }
    /**
     * 迭代方法：移动到下一个元素
     */
```

```
    @Override
    public void next() {

        if(index < size)
        {
            index ++;
        }
    }

}
```

客户端类

```
public class Client {

    public void operation(){
        Object[] objArray = {"One","Two","Three","Four","Five","Six"};
        //创建聚合对象
        Aggregate agg = new ConcreteAggregate(objArray);
        //循环输出聚合对象中的值
        Iterator it = agg.createIterator();
        while(!it.isDone()){
            System.out.println(it.currentItem());
            it.next();
        }
    }
    public static void main(String[] args) {

        Client client = new Client();
        client.operation();
    }

}
```

上面的例子首先创建了一个聚集类实例，然后调用聚集对象的工厂方法createIterator()以得到一个迭代子对象。在得到迭代子的实例后，客户端开始迭代过程，打印出所有的聚集元素。

**外禀迭代子的意义**

一个常常会问的问题是：既然白箱聚集已经向外界提供了遍历方法，客户端已经可以自行进行迭代了，为什么还要应用迭代子模式，并创建一个迭代子对象进行迭代呢？

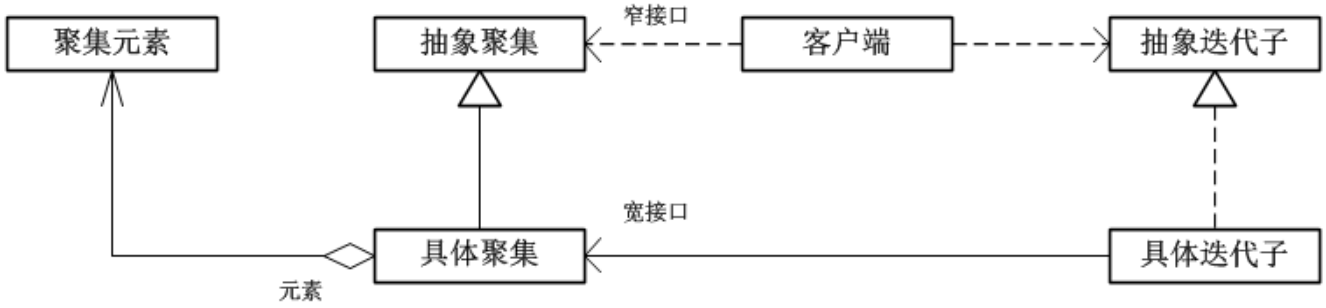
客户端当然可以自行进行迭代，不一定非得需要一个迭代子对象。但是，迭代子对象和迭代模式会将迭代过程抽象化，将作为迭代消费者的客户端与迭代负责人的迭代子责任分隔开，使得两者可以独立的演化。在聚集对象的种类发生变化，或者迭代的方法发生改变时，迭代子作为一个中介层可以吸收变化的因素，而避免修改客户端或者聚集本身。

此外，如果系统需要同时针对几个不同的聚集对象进行迭代，而这些聚集对象所提供的遍历方法有所不同时，使用迭代子模式和一个外界的迭代子对象是有意义的。具有同一迭代接口的不同迭代子对象处理具有不同遍历接口的聚集对象，使得系统可以使用一个统一的迭代接口进行所有的迭代。

## 黑箱聚集与内禀迭代子

如果一个聚集的接口没有提供修改聚集元素的方法，这样的接口就是所谓的**窄接口**。

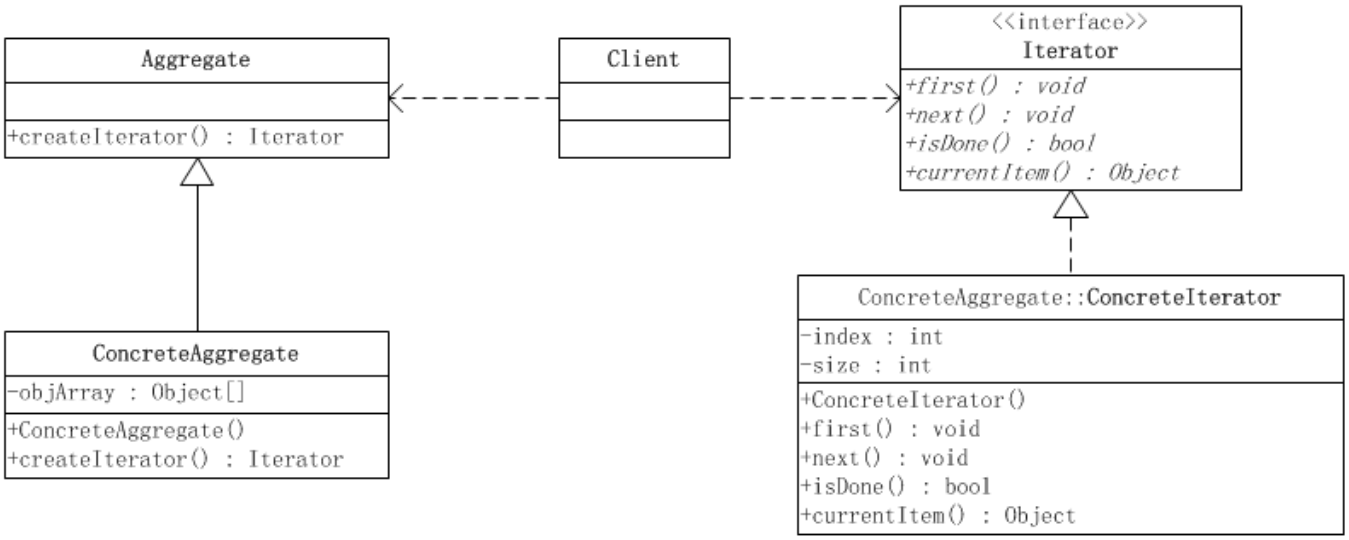
聚集对象为迭代子对象提供一个宽接口，而为其他对象提供一个窄接口。换言之，聚集对象的内部结构应当对迭代子对象适当公开，以便迭代子对象能够对聚集对象有足够的了解，从而可以进行迭代操作。但是，聚集对象应当避免向其他的对象提供这些方法，因为其他对象应当经过迭代子对象进行这些工作，而不是直接操控聚集对象。




在JAVA语言中，实现双重接口的办法就是将迭代子类设计成聚集类的内部成员类。这样迭代子对象将可以像聚集对象的内部成员一样访问聚集对象的内部结构。下面给出一个示意性的实现，说明这种双重接口的结构时怎么样产生的，以及使用了双重接口结构之后迭代子模式的实现方案。这种同时保证聚集对象的封装和迭代子功能的实现的方案叫做**黑箱实现方案**。

由于迭代子是聚集的内部类，迭代子可以自由访问聚集的元素，所以迭代子可以自行实现迭代功能并控制对聚集元素的迭代逻辑。由于迭代子是在聚集的结构之内定义的，因此这样的迭代子又叫做**内禀迭代子（Intrinsic Iterator）**。

为了说明黑箱方案的细节，这里给出一个示意性的黑箱实现。在这个实现里，聚集类ConcreteAggregate含有一个内部成员类ConcreteIterator，也就是实现了抽象迭代子接口的具体迭代子类，同时聚集并不向外界提供访问自己内部元素的方法。




**源代码**

抽象聚集角色类，这个角色规定出所有的具体聚集必须实现的接口。迭代子模式要求聚集对象必须有一个工厂方法，也就是createIterator()方法，以向外界提供迭代子对象的实例。

```
public abstract class Aggregate {
    /**
     * 工厂方法，创建相应迭代子对象的接口
     */
    public abstract Iterator createIterator();
}
```

抽象迭代子角色类

```
public interface Iterator {
    /**
     * 迭代方法：移动到第一个元素
     */
    public void first();
    /**
     * 迭代方法：移动到下一个元素
     */
    public void next();
```

```
    /**
     * 迭代方法：是否为最后一个元素
     */
    public boolean isDone();
    /**
     * 迭代方法：返还当前元素
     */
    public Object currentItem();
}
```

具体聚集角色类，实现了抽象聚集角色所要求的接口，也就是createIterator()方法。此外，聚集类有一个内部成员类ConcreteIterator，这个内部类实现了抽象迭代子角色所规定的接口；而工厂方法createIterator()所返还的就是这个内部成员类的实例。

```
public class ConcreteAggregate extends Aggregate {

    private Object[] objArray = null;
    /**
     * 构造方法，传入聚合对象的具体内容
     */
    public ConcreteAggregate(Object[] objArray){
        this.objArray = objArray;
    }

    @Override
    public Iterator createIterator() {

        return new ConcreteIterator();
    }
    /**
     * 内部成员类，具体迭代子类
     */
    private class ConcreteIterator implements Iterator
    {
        //内部索引，记录当前迭代到的索引位置
        private int index = 0;
        //记录当前聚集对象的大小
        private int size = 0;
        /**
         * 构造函数
```

```
         */
        public ConcreteIterator(){

            this.size = objArray.length;
            index = 0;
        }
        /**
         * 迭代方法：返还当前元素
         */
        @Override
        public Object currentItem() {
            return objArray[index];
        }
        /**
         * 迭代方法：移动到第一个元素
         */
        @Override
        public void first() {

            index = 0;
        }
        /**
         * 迭代方法：是否为最后一个元素
         */
        @Override
        public boolean isDone() {
            return (index >= size);
        }
        /**
         * 迭代方法：移动到下一个元素
         */
        @Override
        public void next() {

            if(index < size)
            {
                index ++;
            }
        }
    }
}
```

客户端类

```
public class Client {

    public void operation(){
        Object[] objArray = {"One","Two","Three","Four","Five","Six"};
        //创建聚合对象
        Aggregate agg = new ConcreteAggregate(objArray);
        //循环输出聚合对象中的值
        Iterator it = agg.createIterator();
        while(!it.isDone()){
            System.out.println(it.currentItem());
            it.next();
        }
    }
    public static void main(String[] args) {

        Client client = new Client();
        client.operation();
    }

}
```

上面的例子首先创建了一个聚集类实例，然后调用聚集对象的工厂方法createIterator()以得到一个迭代子对象。在得到迭代子的实例后，客户端开始迭代过程，打印出所有的聚集元素。

## 主动迭代子和被动迭代子

主动迭代子和被动迭代子又称作外部迭代子和内部迭代子。

所谓主动（外部）迭代子，指的是由客户端来控制迭代下一个元素的步骤，客户端会明显调用迭代子的next()等迭代方法，在遍历过程中向前进行。

所谓被动（内部）迭代子，指的是由迭代子自己来控制迭代下一个元素的步骤。因此，如果想要在迭代的过程中完成工作的话，客户端就需要把操作传递给迭代子，迭代子在迭代的时候会在每个元素上执行这个操作，类似于JAVA的回调机制。

总体来说外部迭代器比内部迭代器要灵活一些，因此我们常见的实现多属于主动迭代

子。

## 静态迭代子和动态迭代子

- 静态迭代子由聚集对象创建，并持有聚集对象的一份快照(snapshot)，在产生后这个快照的内容就不再变化。客户端可以继续修改原聚集的内容，但是迭代子对象不会反映出聚集的新变化。

静态迭代子的好处是它的安全性和简易性，换言之，静态迭代子易于实现，不容易出现错误。但是由于静态迭代子将原聚集复制了一份，因此它的短处是对时间和内存资源的消耗。

- 动态迭代子则与静态迭代子完全相反，在迭代子被产生之后，迭代子保持着对聚集元素的引用，因此，任何对原聚集内容的修改都会在迭代子对象上反映出来。

完整的动态迭代子不容易实现，但是简化的动态迭代子并不难实现。大多数JAVA设计师遇到的迭代子都是这种简化的动态迭代子。为了说明什么是简化的动态迭代子，首先需要介绍一个新的概念：Fail Fast。

## Fail Fast

如果一个算法开始之后，它的运算环境发生变化，使得算法无法进行必需的调整时，这个算法就应当立即发出故障信号。这就是Fail Fast的含义。

如果聚集对象的元素在一个动态迭代子的迭代过程中发生变化时，迭代过程会受到影响而变得不能自恰。这时候，迭代子就应当立即抛出一个异常。这种迭代子就是实现了Fail Fast功能的迭代子。

### Fail Fast在JAVA聚集中的使用

JAVA语言以接口java.util.Iterator的方式支持迭代子模式，Collection接口要求提供iterator()方法，此方法在调用时返还一个Iterator类型的对象。而作为Collection接口的子类型，AbstractList类的内部成员类Itr便是实现Iterator接口的类。

Itr类的源代码如下所示

```
private class Itr implements Iterator<E> {
        /**
         * Index of element to be returned by subsequent call to next.
         */
        int cursor = 0;
```

```
        /**
         * Index of element returned by most recent call to next or
         * previous.  Reset to -1 if this element is deleted by a call
         * to remove.
         */
        int lastRet = -1;

        /**
         * The modCount value that the iterator believes that the backin
g
         * List should have.  If this expectation is violated, the itera
tor
         * has detected concurrent modification.
         */
        int expectedModCount = modCount;

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

        public E next() {
                checkForComodification();
            try {
            E next = get(cursor);
            lastRet = cursor++;
            return next;
            } catch (IndexOutOfBoundsException e) {
            checkForComodification();
            throw new NoSuchElementException();
            }
        }

        public void remove() {
            if (lastRet == -1)
            throw new IllegalStateException();
                checkForComodification();

            try {
            AbstractList.this.remove(lastRet);
            if (lastRet < cursor)
```

```
                cursor--;
            lastRet = -1;
            expectedModCount = modCount;
            } catch (IndexOutOfBoundsException e) {
            throw new ConcurrentModificationException();
            }
        }

        final void checkForComodification() {
            if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
        }
    }
```

从Itr类的源代码中可以看到，方法checkForComodification()会检查聚集的内容是否刚刚被外界直接修改过(不是通过迭代子提供的方法修改的)。如果在迭代开始后，聚集的内容被外界绕过迭代子对象而直接修改的话，这个方法会立即抛出ConcurrentModificationException()异常。

这就是说，AbstractList.Itr迭代子是一个Fail Fast的迭代子。

## 迭代子模式的优点

（1）迭代子模式简化了聚集的接口。迭代子具备了一个遍历接口，这样聚集的接口就不必具备遍历接口。

（2）每一个聚集对象都可以有一个或多个迭代子对象，每一个迭代子的迭代状态可以是彼此独立的。因此，一个聚集对象可以同时有几个迭代在进行之中。

（3）由于遍历算法被封装在迭代子角色里面，因此迭代的算法可以独立于聚集角色变化。