简述

迭代器模式又称为游标模式，提供一种方法顺序访问一个聚集对象中的各种元素，而又不暴露该对象的内部表示

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Vector、ArrayList、HashSet、HashMap、Hashtable等，这些都是JAVA聚集的例子

适用场景

对聚集做只读操作，不能通过集合添加、删除集合的元素

迭代器模式中的几个角色

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

代码实现

迭代器接口类（抽象迭代器角色）

```
/**
 * 迭代器接口类
 */
public interface Iterator {
    boolean hasNext();
    Object next();
}
```

聚集接口（抽象聚集角色）

```
/**
 * 聚集接口
 */
public interface Aggregate {

    /**
     * 提供一个迭代集合数据的方法
     * @return 迭代器
     */
     Iterator iterator();
}
```

图书类

```
import lombok.Data;

/**
 * 图书类
 */

@Data
public class Book {
    private String name;

     public Book(String name){
         this.name = name;
     }
}
```

图书-具体聚集类（具体聚集角色）

```
/**
 * 图书 具体聚合类
 */
public class BookAggregate implements Aggregate{

    private Book[] books;
```

```
7
8       // 记录最后一本书的下标，初始化为 -1
9       private int last;
10
11      public int getLength(){
12          return last + 1;
13      }
14
15      public void addBook(Book book){
16          last++;
17          books[last] = book;
18      }
19
20      public Book getBook(int index){
21          return books[index];
22      }
23
24      public BookAggregate(int maxSize){
25          books = new Book[maxSize];
26          this.last = -1;
27      }
28
29      @Override
30      public Iterator iterator() {
31          return new BookIterator(this);
32      }
33  }
```

图书-迭代器（具体迭代器）

```
1   /**
2    * 图书 迭代器
3    */
4   public class BookIterator implements Iterator{
5
6       private BookAggregate aggregate;
7       private int last;
8
9       public BookIterator(BookAggregate bookAggregate){
10          this.last = -1;
11          this.aggregate = bookAggregate;
12      }
13
14      @Override
15      public boolean hasNext() {
16          if(last < aggregate.getLength() - 1){
17              last++;
18              return true;
19          }
20          return false;
21      }
22
23      @Override
24      public Object next() {
25          return aggregate.getBook(last);
```

```
    }
}
```

测试类

```
public class Main {

    public static void main(String[] args) {
        Book book1 = new Book("三国");
        Book book2 = new Book("西游记");
        Book book3 = new Book("水浒");

        BookAggregate bookAggregate = new BookAggregate(10);
         bookAggregate.addBook(book1);
         bookAggregate.addBook(book2);
         bookAggregate.addBook(book3);

         Iterator iterator = bookAggregate.iterator();
         while (iterator.hasNext()){
             System.out.println(iterator.next());
         }
    }
}
```

总结

1. 这里的循环遍历，不依赖 BookAggregate ， 而是依赖与 BookIterator
2. 这里可以访问 BookAggregate 中的元素，但是又不暴露 BookAggregate中的其他方法

扩展

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 Fa

在java中的使用 - modCount详解

在ArrayList,LinkedList,HashMap等等的内部实现增，删，改中总能看到modCount的身影，modCount字面意思就是修改次数，但为什

有一个公共特点，所有使用modCount属性的全是线程不安全的，这是为什么呢?

以hashMap为例

```
final Node<K,V> nextNode() {
    Node<K,V>[] t;
    Node<K,V> e = next;
    if (modCount != expectedModCount)
        throw new ConcurrentModificationException();
    if (e == null)
```

```
7           throw new NoSuchElementException();
8       if ((next = (current = e).next) == null && (t = table) != null) {
9           do {} while (index < t.length && (next = t[index++]) == null);
10       }
11       return e;
12   }
```

迭代遍历的时候，会初始化expectedModCount=modCount，这时候对HashMap进行修改操作，modCount会+1，继续遍历的时候e
继而抛出java.util.ConcurrentModificationException异常