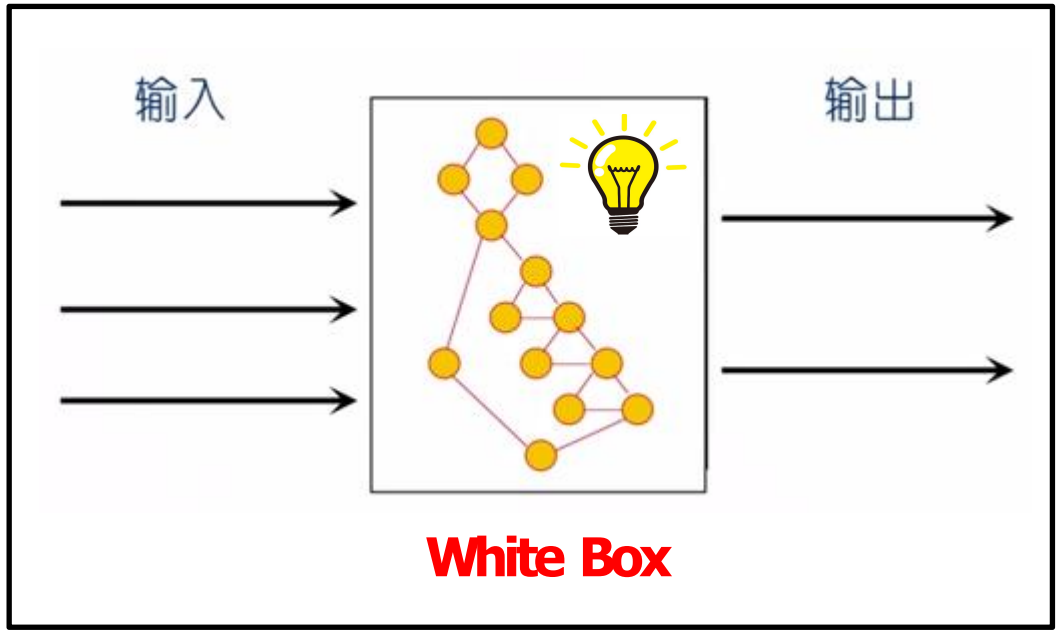
第七章：集合与策略、迭代器模式

第六章：软件测试及代码质量保障

- 1. 软件测试的定义和分类
- 2. 测试用例的定义和其设计方法
- 3. 白盒测试（定义、控制流程图、逻辑覆盖方法、测试用例设计）
- 4. 黑盒测试（定义、等价类划分、边界值分析、场景法）
- 5. 压力测试，性能测试和代码覆盖率测试
- 6. 代码质量保证

白盒测试：定义

也称作结构测试或逻辑驱动测试，它是基于程序的源代码，已知产品的内部工作过程，主要是对程序内部结构展开测试，关注程序实现细节，检验程序中的每条通路是否都有按照预定要求正确工作。



优势：

- 针对性强，可快速定位Bug
- 函数级别，Bug修复成本低
- 有助于了解测试的覆盖程度
- 有助于优化代码，预防缺陷

劣势：

- 对测试人员要求高
- 成本高

白盒测试：逻辑覆盖方法

覆盖指标	
语句覆盖	至少执行程序中所有语句一次。语句覆盖是最弱的逻辑覆盖准则。
判定覆盖	也称为分支覆盖，至少执行程序中每个分支一次，保证程序中每个判定节点取得每种可能的结果至少一次。
条件覆盖	保证程序中每个复合判定表达式中，每个简单判定条件的取真和取假情况至少执行一次。
判定条件覆盖	保证程序中每个复合判定表达式中，每个简单判定条件的取真和取假情况至少执行一次（条件覆盖），同时保证程序中每个判定节点取得每种可能的结果至少一次（判定覆盖）。
条件组合覆盖	保证程序每个判定节点中，所有简单判定条件的所有可能的取值组合情况至少执行一次。
路径覆盖	使程序中每一条可能的路径至少执行一次。该策略是最强的，但一般是不可实现的。

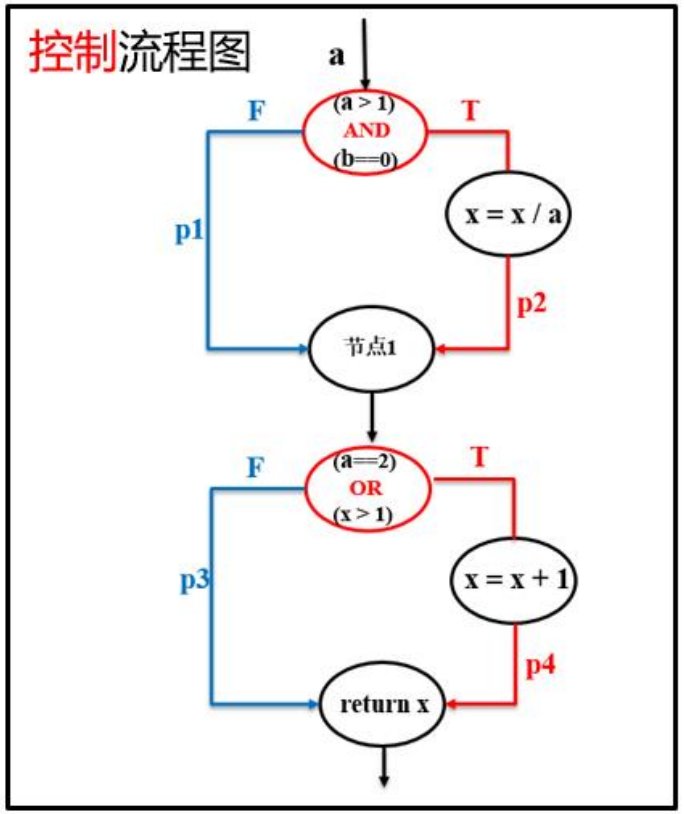
白盒测试：一个例子

- **4个基本逻辑判定条件**

T1	a > 1
T2	b==0
T3	a==2
T4	X>1

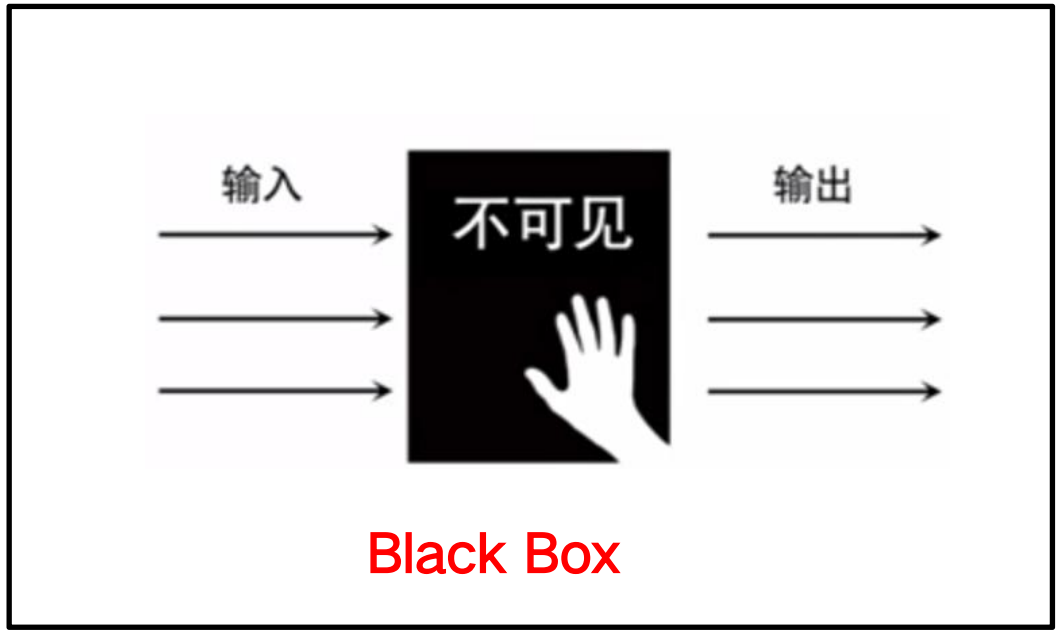
- **4条执行路径**

L13	p1->p3
L14	p1->p4
L23	p2->p3
L24	p2->p4



黑盒测试：定义

也称作功能测试或数据驱动测试，它着眼于程序外部结构，在完全不考虑程序的内部逻辑结构和内部特性的情况下，测试者在程序接口进行测试，它只检查程序功能是否按照需求规格说明书的规定正常使用，程序是否能适当的接收输入数据而产生正确的输出信息。



优势：

- 方法简单有效
- 可以整体测试系统行为
- 开发与测试可以并行
- 对测试人员技术要求相对较低

劣势：

- 覆盖率较低
- 直接依赖于需求规格说明书
- 入门门槛低

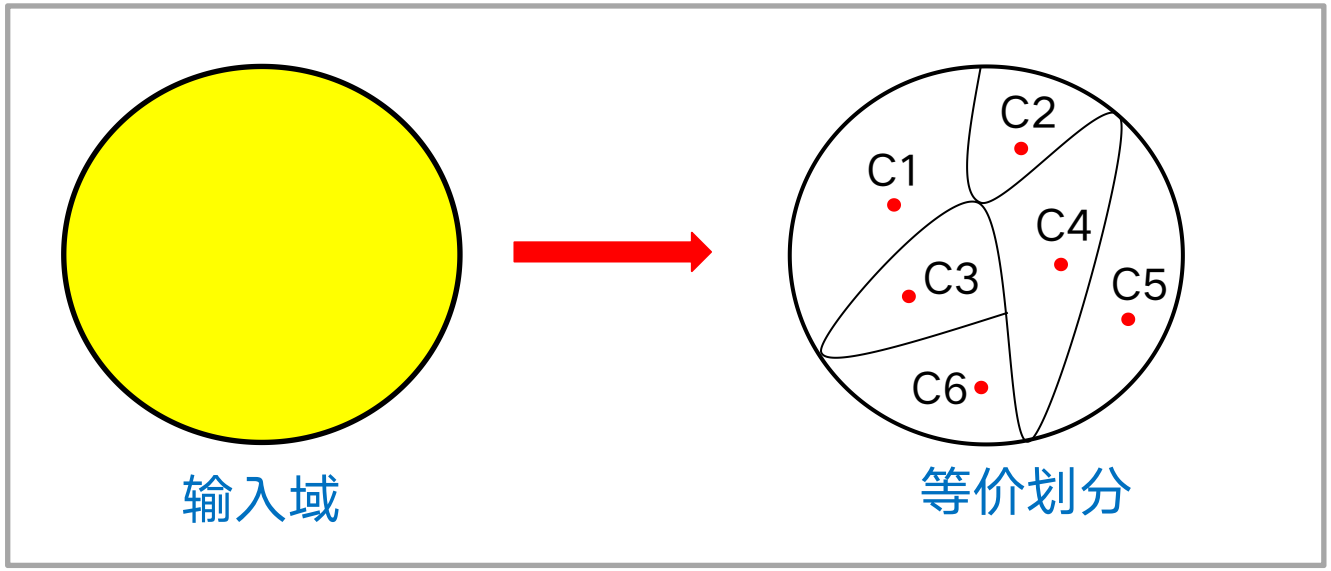
黑盒测试：等价类划分

等价类

输入域的一个子集，在该子集中，各个输入数据对揭示程序中错误是等效的。

等价类测试

将无穷多的数据缩减到有限个等价区域中，通过测试等价区域完成穷尽测试。



- 分而不交
- 合而不变
- 类内等价

黑盒测试：等价类划分

有效等价类

- 指对于软件的规格说明书而言，是合理的，有意义的输入数据所构成的集合。
- 用于检验被测系统是否能够完成指定功能。

无效等价类

- 指对于软件的规格说明书而言，是不合理的，没有意义的输入数据所构成的集合。
- 用于考察被测系统的容错性。

黑盒测试：一个例子

系统某查询条件是 1990 年 1 月到 2019 年 12 月，由 6 位数字表示，前 4 位表示年份，后 2 位表示月份。

- 确定有效等价类和无效等价类：

输入条件	有效等价类		无效等价类	
日期的类型和长度	C1	6位数字字符	C2	有非数字字符
			C3	少于6位数字字符
			C4	多于6位数字字符
年份范围	C5	1990~2019	C6	小于1990
			C7	大于2019
月份范围	C8	01~12	C9	等于00
			C10	大于12

黑盒测试：边界值分析

是一种常用的黑盒测试技术。它倾向于选择系统边界或边界附近的数据来设计测试用例，考虑了边界条件的测试用例具有更高的测试回报率。

基本原理

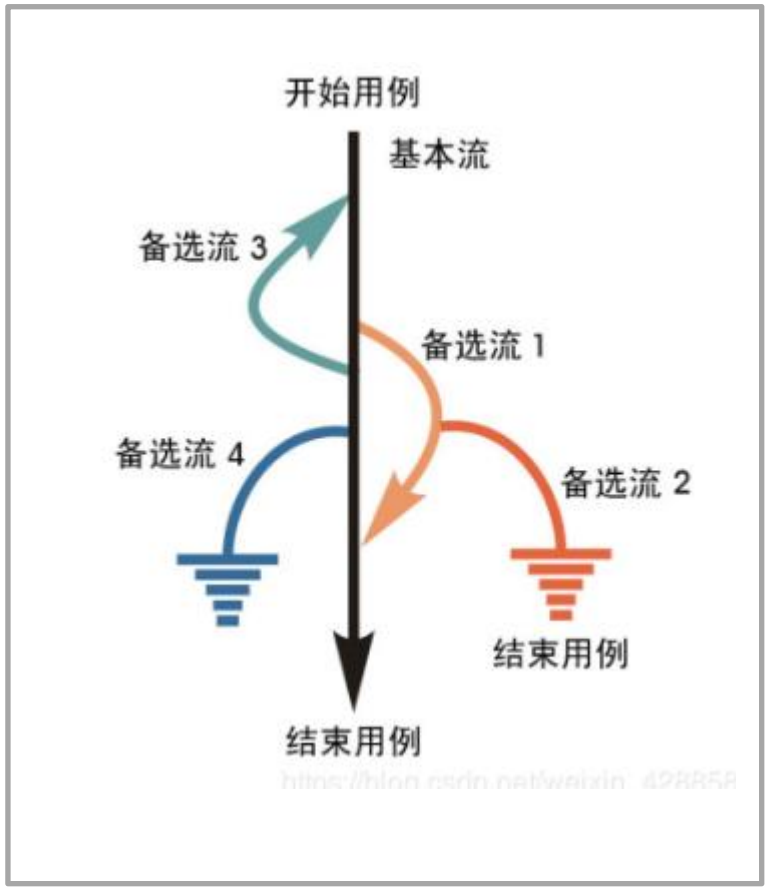
经过长期的测试工作经验表明，大量的缺陷经常发生在输入域或输出域的边界上。因此设计一些测试用例，使程序运行在边界情况附近，这样揭露错误的可能性比较大。

黑盒测试：场景法

以事件流为核心，主要目的是测试软件的主要业务流程、主要功能的正确性和异常处理能力 。

测试步骤:

- 定义基本流和备选流;
- 定义场景;
- 从场景设计测试用例;
- 输入测试数据，完善测试用例。



黑盒测试：一个例子

举个例子：ATM取款

基本流（正确取款）	
1	插入银行卡
2	验证银行卡
3	输入密码
4	验证密码
5	进入ATM机主界面
6	取款并选择金额
7	ATM机验证
8	更新账户余额并出钞
9	返回主界面

基本流

备选流（出错环节）	
1	银行卡错误
2	密码错误
3	密码3次错误
4	卡内余额不足
5	超出当日可取
6	ATM机余额不足

备选流

黑盒测试：一个例子

- 定义场景并设计测试用例

用例编号	场景	银行卡号	密码	账户余额	当日可取	ATM余额	预期结果
1	场景1：成功取款	V	V	V	V	V	成功取款
2	场景2：银行卡无效	I	N/A	N/A	N/A	N/A	提示错误、退卡
3	场景3：密码错误	V	I	N/A	N/A	N/A	提示错误
4	场景4：密码3次错误	V	I	N/A	N/A	N/A	提示错误、吞卡
5	场景5：账户余额不足	V	V	I	N/A	N/A	提示错误、退卡
6	场景6：总取款金额超出当日可取限额	V	V	V	I	N/A	提示错误、退卡
7	场景7：ATM机余额不足	V	V	V	V	I	提示错误、退卡

注：V 表示有效的数值；I 表示无效的数值；N/A表示不适用。

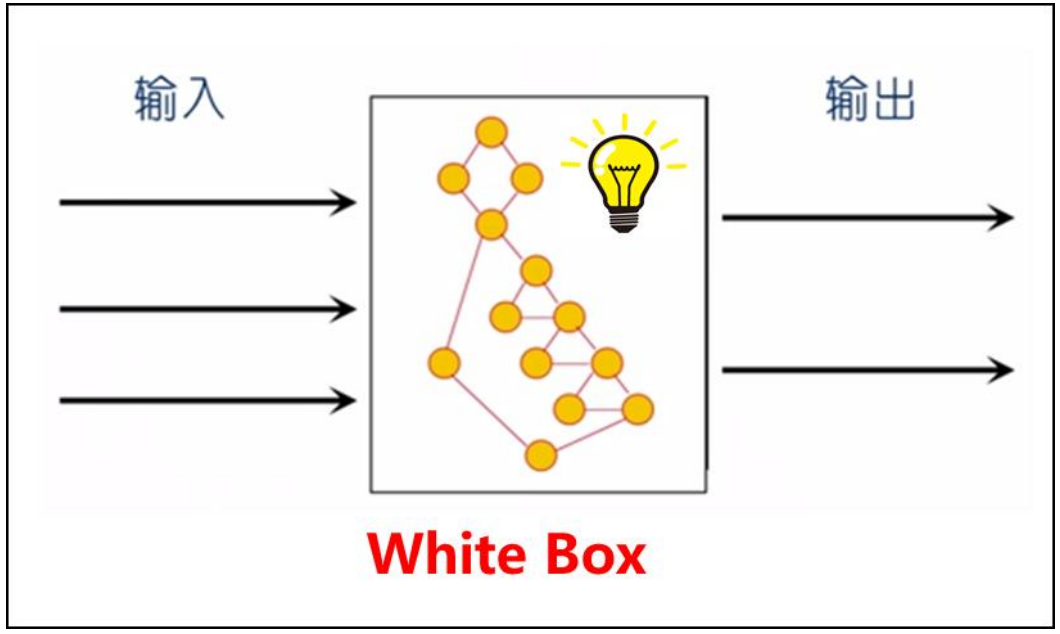
黑盒测试：一个例子

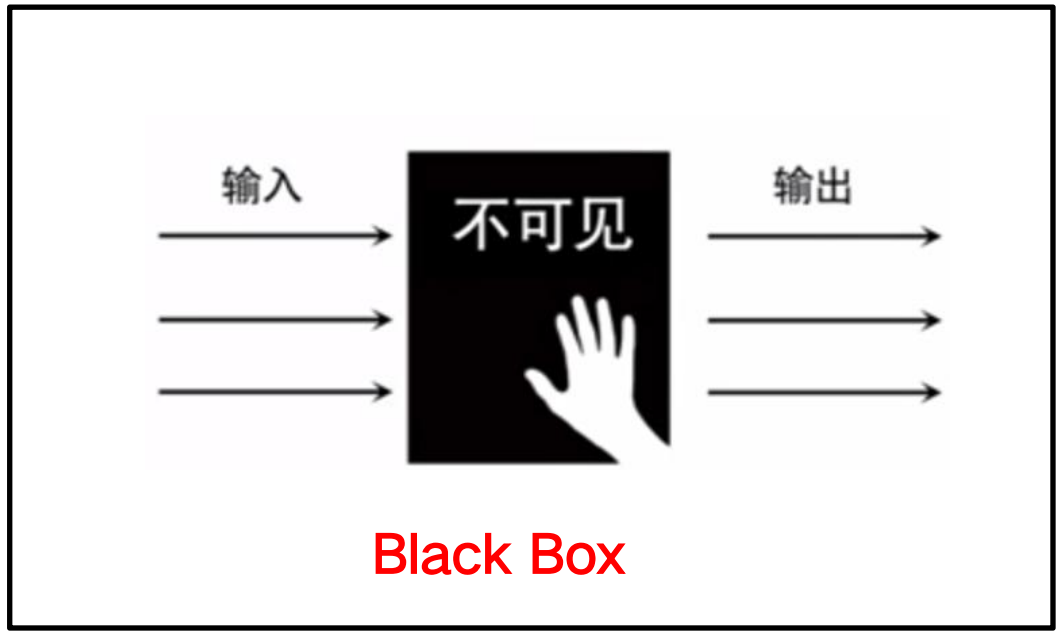
➢ **输入测试数据，完善测试用例**

用例编号	对应场景	前提条件	操作步骤	预期结果
C1	**场景1**	① ATM机余额5000元 ② 银行卡号：1234567812345678 ③ 密码：123456 ④ 卡内余额：2000元	① 插入银行卡 ② 输入正确密码 ③ 进入主页后选择取款1000元	① ATM机输出1000元，提示用户取走并返回主界面 ② ATM余额4000元 ③ 用户卡内余额1000元
C2	**场景2**	ATM机余额4000元	插入无效银行卡	提示银行卡无效并退卡
C3	**场景3**	① ATM机余额4000元 ② 银行卡号：1234567812345678 ③ 密码：123456 ④ 卡内余额：1000元	① 插入银行卡 ② 输入错误密码：654321	提示密码错误，并清空密码
C4	**场景3**	① ATM机余额4000元 ② 银行卡号：1234567812345678 ③ 密码：123456 ④ 卡内余额：1000元	在C3基础上再次输入错误密码：987654	提示密码错误，并清空密码
C5	**场景4**	① ATM机余额4000元 ② 银行卡号：1234567812345678 ③ 密码：123456 ④ 卡内余额：1000元	在C4基础上再次输入错误密码：876543	提示密码错误，并没收该卡

对比总结

- **白盒测试**（结构测试）：主要检测软件编码过程中的错误。
- **黑盒测试**（功能测试）：主要检测功能是否能够按照规格说明书正常使用。





- 对测试人员要求高
- 发现代码中的问题

- 相对简单
- 从用户角度测试功能

第七章：集合与策略、迭代器模式

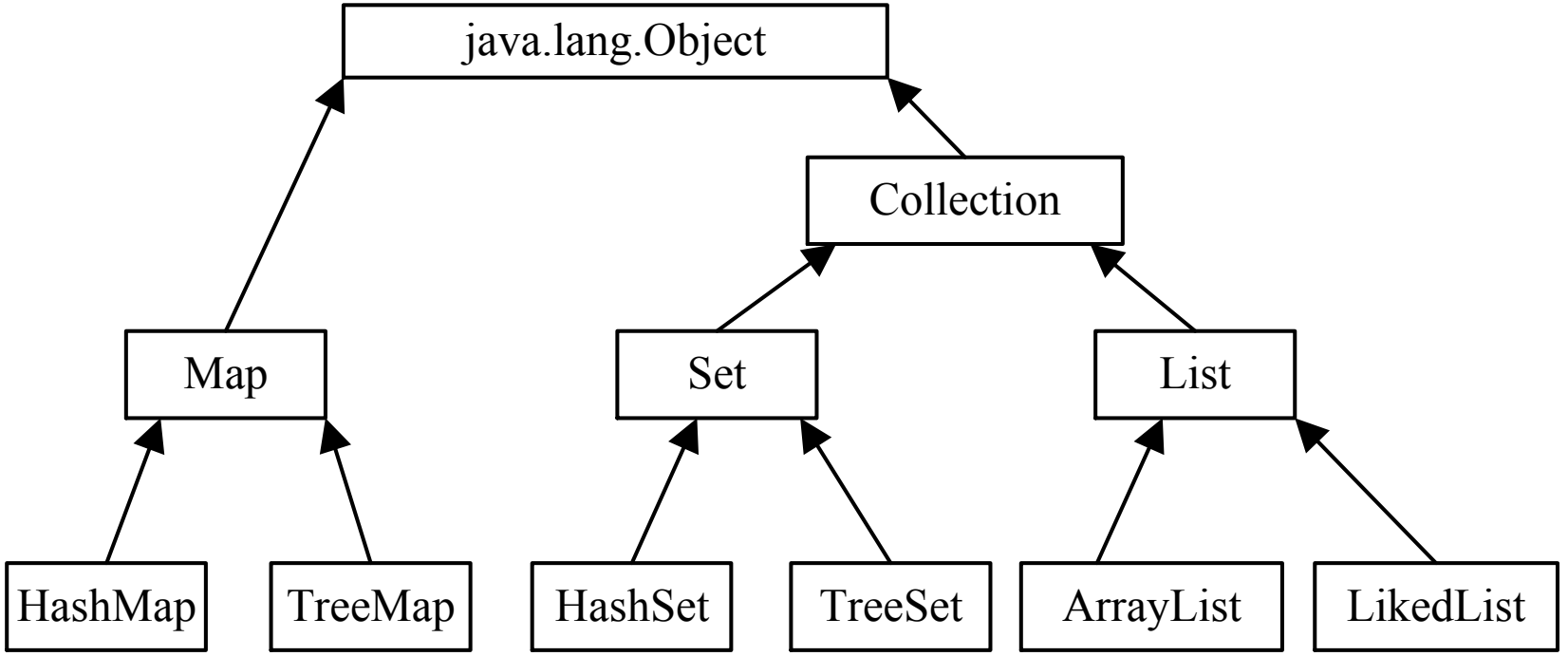
- 集合类概述
- List接口及其标准实现
- 类ArrayList与LinkedList
- Set与Map接口
- 策略模式
- 迭代器模式

什么是集合

- Java语言的java.util包中提供了一些集合类，这些集合类又被称为容器。提到容器难免会想到数组，集合类与数组不同之处是，**数组**的长度是**固定**的，**集合**的长度是**可变**的；
- 集合中接口决定了集合API中各个类的基本特性。具体类仅仅是提供了标准接口的不同实现。

什么是集合

- 最基本的接口是Collection接口，该接口定义了操作数据的基本方法。
- 常用的集合有List集合、Set集合、Map集合，其中List与Set实现了Collection接口。各接口还提供了不同的实现类。



第七章：集合与策略、迭代器模式

- 集合类概述
- List接口及其标准实现
- 类ArrayList与LinkedList
- Set与Map接口
- 策略模式
- 迭代器模式

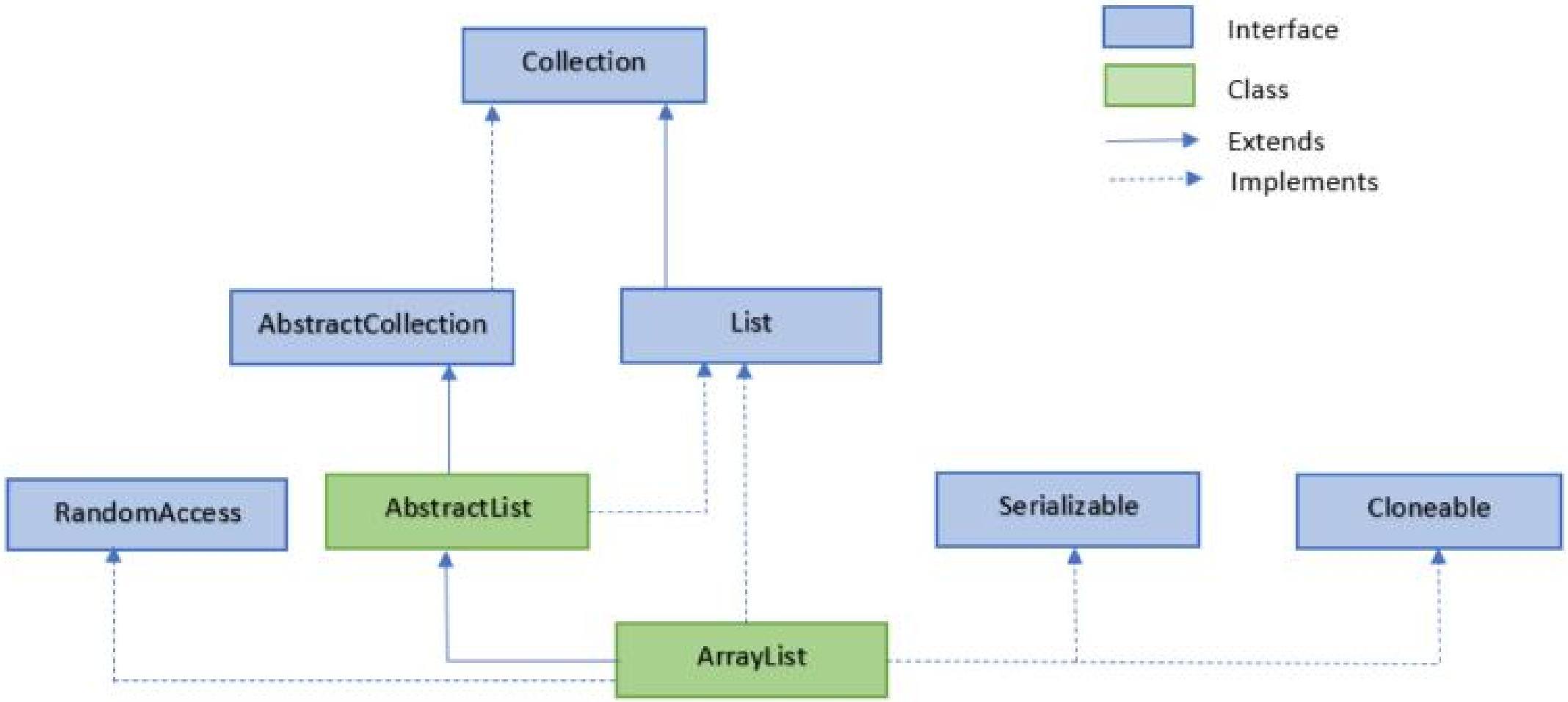
List接口

- List接口定义了一个有序的对象集合，允许重复元素存在。
- List类似于动态数组或变长数组，List中存放的元素的数量(List的容量)可以随着插入操作自动调整。

ArrayList类

- ArrayList 类是一个可以动态修改的数组，与普通数组的区别就是它是没有固定大小的限制，我们可以添加或删除元素。
- ArrayList 继承了 AbstractList ，并实现了 List 接口。

ArrayList类


Interface
Class
Extends
Implements
Collection
AbstractCollection
List
RandomAccess
AbstractList
Serializable
Cloneable
ArrayList

ArrayList类

ArrayList 类位于 java.util 包中，使用前需要引入它，语法格式如下：

```
import java.util.ArrayList; // 引入 ArrayList 类

ArrayList<E> objectName =new ArrayList<E>();     // 初始化
```

- E: 泛型数据类型，用于设置 objectName 的数据类型。

ArrayList 是一个数组队列，提供了相关的添加、删除、修改、遍历等功能。

ArrayList类

实例：

```
import java.util.ArrayList;

public class ArrayListTest {
    public static void main(String[] args) {
        ArrayList<String> sites = new ArrayList<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        sites.add("Weibo");
        System.out.println(sites);
    }
}
```

输出为：[Google, Amazon, Taobao, Weibo]

ArrayList类

在列表中间增加元素：

```
import java.util.ArrayList;

public class ArrayListTest {
    public static void main(String[] args) {
        ArrayList<String> sites = new ArrayList<String>();
        sites.add(“Google”);
        sites.add(“Amazon”);
        sites.add(“Taobao”);
        sites.add(1, "Weibo");
        System.out.println(sites);
    }
}
```

输出为：　[Google, Weibo, Amazon, Taobao]

ArrayList类

删除元素：如果要删除 ArrayList 中的元素可以使用 remove() 方法

```
import java.util.ArrayList;

public class ArrayListTest {
    public static void main(String[] args) {
        ArrayList<String> sites = new ArrayList<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        sites.add("Weibo");
        sites.remove(3); // 删除第四个元素
        System.out.println(sites);
    }
}
```

输出为：　[Google, Amazon, Taobao]

ArrayList类

修改元素： 访问 ArrayList 中的元素可以使用 set() 方法

```
import java.util.ArrayList;

public class ArrayListTest {
    public static void main(String[] args) {
        ArrayList<String> sites = new ArrayList<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        sites.add("Weibo");
        sites.set(2, "Wiki"); // 第一个参数为索引位置，第二个为要修改的值
        System.out.println(sites);
    }
}
```

输出为： [Google, Amazon, Wiki, Weibo]

ArrayList类

访问元素： 访问 ArrayList 中的元素可以使用 get() 方法

```
import java.util.ArrayList;

public class ArrayListTest {
    public static void main(String[] args) {
        ArrayList<String> sites = new ArrayList<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        sites.add("Weibo");
        System.out.println(sites.get(1));
    }
}
```

输出为： Amazon

ArrayList类

计算大小： 如果要计算 ArrayList 中的元素数量可以使用 size() 方法

```
import java.util.ArrayList;

public class ArrayListTest {
   public static void main(String[] args) {
      ArrayList<String> sites = new ArrayList<String>();
      sites.add("Google");
      sites.add("Amazon");
      sites.add("Taobao");
      sites.add("Weibo");
      System.out.println(sites.size());
   }
}
```

输出为： 4

ArrayList类

迭代数组列表： 可以使用 for 来迭代数组列表中的元素

```
import java.util.ArrayList;
public class ArrayListTest {
   public static void main(String[] args) {
      ArrayList<String> sites = new ArrayList<String>();
      sites.add("Google");
      sites.add("Amazon");
      sites.add("Taobao");
      sites.add("Weibo");
       for (int i = 0; i < sites.size(); i++) {
         System.out.println(sites.get(i));
       }
   }
}
```

输出为：

```
Google
Amazon
Taobao
Weibo
```

ArrayList类

迭代数组列表：也可以使用 for-each 来迭代元素

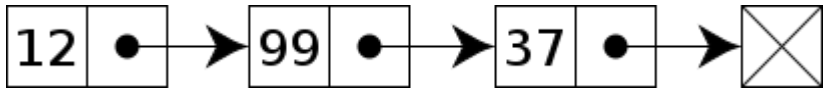
```
import java.util.ArrayList;
public class ArrayListTest {
   public static void main(String[] args) {
      ArrayList<String> sites = new ArrayList<String>();
      sites.add("Google");
      sites.add("Amazon");
      sites.add("Taobao");
      sites.add("Weibo");
       for (String i : sites) {
         System.out.println(i);
      }
   }
}
```

输出为：

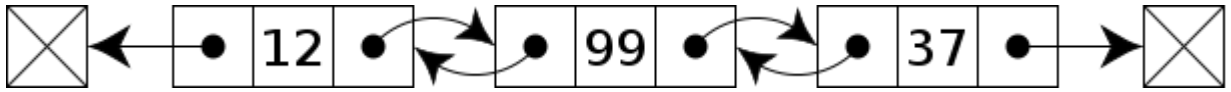
```
Google
Amazon
Taobao
Weibo
```

LinkedList类

- 链表（Linked list）是一种常见的基础数据结构，是一种线性表，但是并不会按线性的顺序存储数据，而是在每一个节点里存到下一个节点的地址。链表可分为单向链表和双向链表。
- 一个单向链表包含两个值: 当前节点的值和一个指向下一个节点的链接。



- 一个双向链表有三个整数值: 数值、向后的节点链接、向前的节点链接。



LinkedList类

- Java LinkedList（链表） 类似于 ArrayList，是一种常用的数据容器。

- 与 ArrayList 相比，LinkedList 的增加和删除的操作效率更高，而查找和修改的操作效率较低。一个单向链表包含两个值: 当前节点的值和一个指向下一个节点的链接。

LinkedList类

以下情况使用 ArrayList：

- 频繁访问列表中的某一个元素。
- 只需要在列表末尾进行添加和删除元素操作。

以下情况使用 LinkedList：

- 需要频繁的在列表开头、中间或指定位置进行添加和删除元素操作。

LinkedList类

LinkedList 类位于 java.util 包中，使用前需要引入它，语法格式如下：

```
// 引入 LinkedList 类
import java.util.LinkedList;
LinkedList<E> objectName = new LinkedList<E>(); // 创建方法
```

E: 泛型数据类型，用于设置 objectName 的数据类型。

LinkedList类

实例：

```
import java.util.LinkedList;

public class LinkedListTest {
    public static void main(String[] args) {
        LinkedList<String> sites = new LinkedList<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        sites.add("Weibo");
        System.out.println(sites);
    }
}
```

输出为： [Google, Amazon, Taobao, Weibo]

LinkedList类

在列表开头添加元素：

```
// 引入 LinkedList 类
import java.util.LinkedList;

public class LinkedListTest {
    public static void main(String[] args) {
        LinkedList<String> sites = new LinkedList<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        // 使用 addFirst() 在头部添加元素
        sites.addFirst("Wiki");
        System.out.println(sites);
    }
}
```

输出为：　[Wiki,Google, Amazon, Taobao]

LinkedList类

在列表中间添加元素：

```
// 引入 LinkedList 类
import java.util.LinkedList;

public class LinkedListTest {
    public static void main(String[] args) {
        LinkedList<String> sites = new LinkedList<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        // 使用 add (index, element) 在中间添加元素
        sites.add (1, "Wiki");
        System.out.println(sites);
    }
}
```

输出为：　[Google, Wiki, Amazon, Taobao]

LinkedList类

注意： 在列表中间添加元素时，是否永远都是LinkedList比ArrayList快？

一般来说，由于ArrayList底层是数组，添加数据需要移动后面的数据，而LinkedList使用的是链表，直接移动指针就行，所以应该是LinkedList更快.

然而插入位置的选取对LinkedList有很大的影响，因为LinkedList在插入时需要先移动指针到指定节点，才能开始插入，一旦要插入的位置比较远，LinkedList就需要一步一步的移动指针，直到移动到插入位置，所以插入节点值越大，LinkedList花费时间越长，因为指针移动需要时间。而ArrayList是数组结构，可以根据下标直接获得位置，这就省去了查找特定节点的时间，所以对ArrayList的影响不是特别大。

总结：虽然有上述情况存在，可是因为ArrayList可以使用下标直接获取数据，所以在使用查询的时候一般选择ArrayList，而进行删除和增加时，LinkedList比较方便，所以一般还是使用LinkedList比较多。

LinkedList类

在列表结尾添加元素：

```
// 引入 LinkedList 类
import java.util.LinkedList;
public class LinkedListTest {
    public static void main(String[] args) {
        LinkedList<String> sites = new LinkedList<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        // 使用 addLast() 在尾部添加元素
        sites.addLast("Wiki");
        System.out.println(sites);
    }
}
```

输出为：　[Google, Amazon, Taobao,Wiki]

LinkedList类

在列表开头移除元素：

```
// 引入 LinkedList 类
import java.util.LinkedList;
public class LinkedListTest {
    public static void main(String[] args) {
        LinkedList<String> sites = new LinkedList<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
       sites.add("Weibo");
        // 使用 removeFirst() 移除头部元素
        sites.removeFirst();
        System.out.println(sites);
    }
}
```

输出为：　[Amazon, Taobao,Weibo]

LinkedList类

在列表结尾移除元素：

```
// 引入 LinkedList 类
import java.util.LinkedList;
public class LinkedListTest {
    public static void main(String[] args) {
        LinkedList<String> sites = new LinkedList<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        sites.add("Weibo");
        // 使用 removeLast() 移除尾部元素
        sites.removeLast();
        System.out.println(sites);
    }
}
```

输出为：　[Google,Amazon, Taobao]

LinkedList类

获取列表开头的元素:

使用 getFirst() 获取头部元素

获取列表结尾的元素:

使用 getLast() 获取尾部元素

LinkedList类

迭代元素：我们可以使用 for 来迭代列表中的元素

```
import java.util.LinkedList;
public class LinkedListTest {
    public static void main(String[] args) {
        LinkedList<String> sites = new LinkedList<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        sites.add("Weibo");
        for (int size = sites.size(), i = 0; i < size; i++) {
            System.out.println(sites.get(i));
        }
    }
}
```

输出为：

```
Google
Amazon
Taobao
Weibo
```

LinkedList类

迭代元素：也可以使用 for-each 来迭代元素：

```
import java.util.LinkedList;
public class LinkedListTest {
    public static void main(String[] args) {
        LinkedList<String> sites = new LinkedList<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        sites.add("Weibo");
        for (String i : sites) {
            System.out.println(i);
        }
    }
}
```

输出为：

```
Google
Amazon
Taobao
Weibo
```

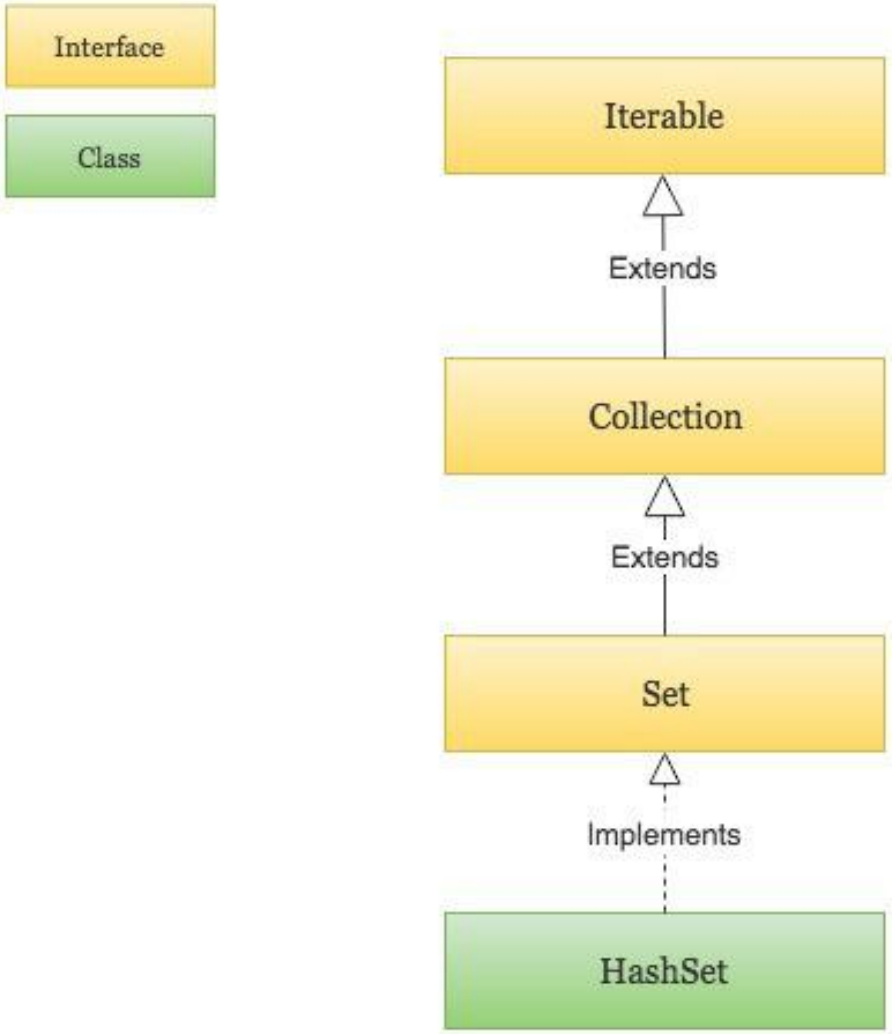
第七章：集合与策略、迭代器模式

- 集合类概述
- List接口及其标准实现
- 类ArrayList与LinkedList
- Set与Map接口
- 策略模式
- 迭代器模式

Set集合

- Set集合中的对象不按特定的方式排序，只是简单地把对象加入集合中，**但Set集合中不能包含重复对象**。
- Set集合由Set接口和Set接口的实现类组成。Set接口继承了Collection接口，因此包含Collection接口的所有方法。

Java HashSet



HashSet类

- HashSet 类位于 java.util 包中，使用前需要引入它，语法格式如下：

```
import java.util.HashSet; // 引入 HashSet 类
HashSet<E> sites = new HashSet<E>();
```

E: 泛型数据类型，用于设置 objectName 的数据类型，只能为引用数据类型。

HashSet类

添加元素：HashSet 类提供类很多有用的方法，添加元素可以使用 add() 方法：

```
// 引入 HashSet 类
import java.util.HashSet;
public class HashsetTest {
    public static void main(String[] args) {
    HashSet<String> sites = new HashSet<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        sites.add("Zhihu");
        sites.add("Amazon");  // 重复的元素不会被添加
        System.out.println(sites);
    }
}
```

输出为：　[Google, Amazon, Taobao, Zhihu]

HashSet类

判断元素是否存在：我们可以使用 contains() 方法来判断元素是否存在于集合当中

```
// 引入 HashSet 类
import java.util.HashSet;
public class HashsetTest {
    public static void main(String[] args) {
    HashSet<String> sites = new HashSet<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        sites.add("Zhihu");
        sites.add("Amazon"); // 重复的元素不会被添加
        System.out.println(sites.contains("Taobao"));
    }
}
```

输出为：　true

HashSet类

删除元素:

我们可以使用 remove() 方法来删除集合中的元素:

```
sites.remove("Taobao"); // 删除元素，删除成功返回 true，否则为 false
```

删除集合中所有元素可以使用 clear 方法:

```
sites.clear();
```

HashSet类

计算大小:

如果要计算 HashSet 中的元素数量可以使用 size() 方法:

```
sites.size()
```

HashSet类

迭代 **HashSet**： 可以使用 for-each 来迭代 HashSet 中的元素。

```
// 引入 HashSet 类
import java.util.HashSet;
public class HashsetTest {
    public static void main(String[] args) {
    HashSet<String> sites = new HashSet<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        sites.add("Zhihu");
        sites.add("Amazon"); // 重复的元素不会被添加
        for (String i : sites) {
            System.out.println(i);
        }
    }
}
```

输出为：

```
Google
Amazon
Taobao
Zhihu
```

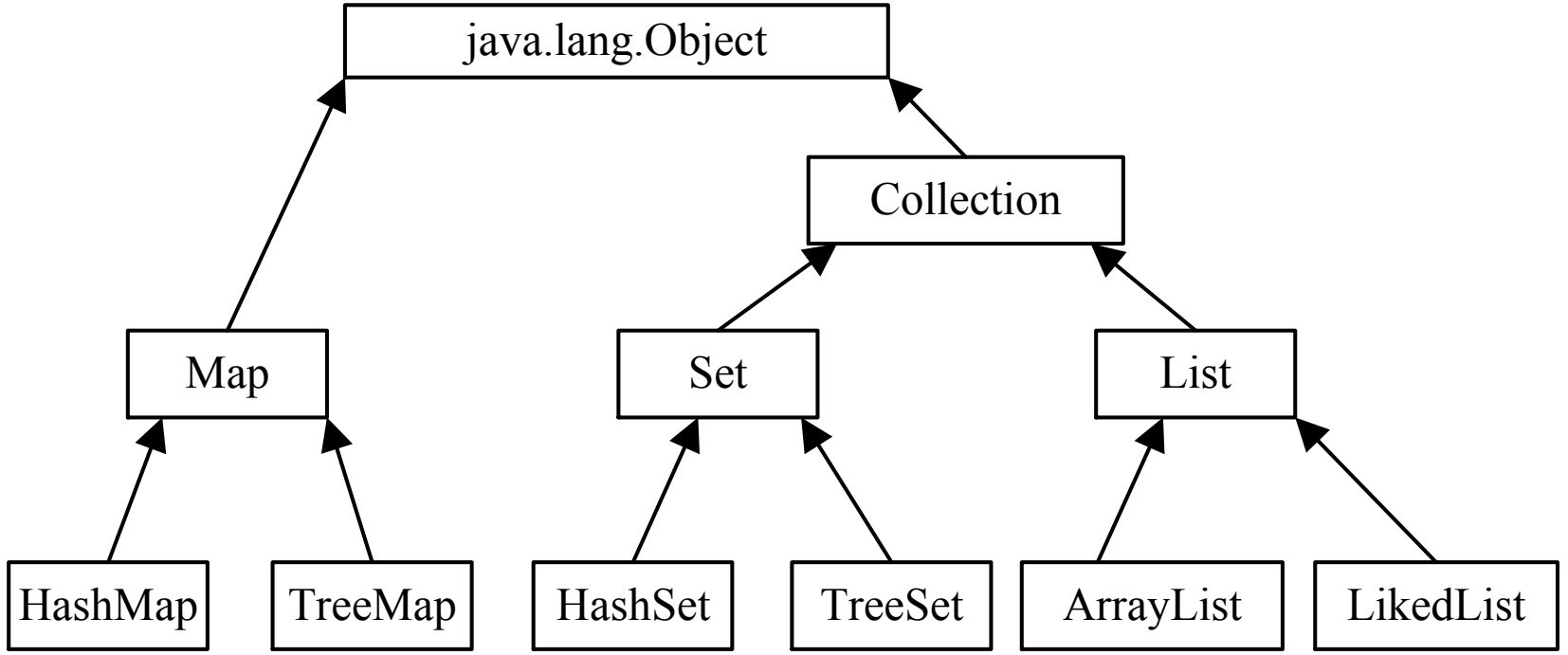
Map集合

- Map接口提供了将key映射到value的对象。一个映射不能包含重复的key，每个key最多只能映射到一个value 。Map接口中同样提供了集合的常用方法，除此之外还包括如下表所示的常用方法。

方　　法	功 能 描 述
put(K key, V value)	向集合中添加指定的 key 与 value 的映射关系
containsKey(Object key)	如果此映射包含指定 key 的映射关系，则返回 true
containsValue(Object value)	如果此映射将一个或多个 key 映射到指定值，则返回 true
get(Object key)	如果存在指定的 key 对象，则返回该对象对应的值，否则返回 null
keySet()	返回该集合中的所有 key 对象形成的 Set 集合
values()	返回该集合中所有值对象形成的 Collection 集合

什么是集合

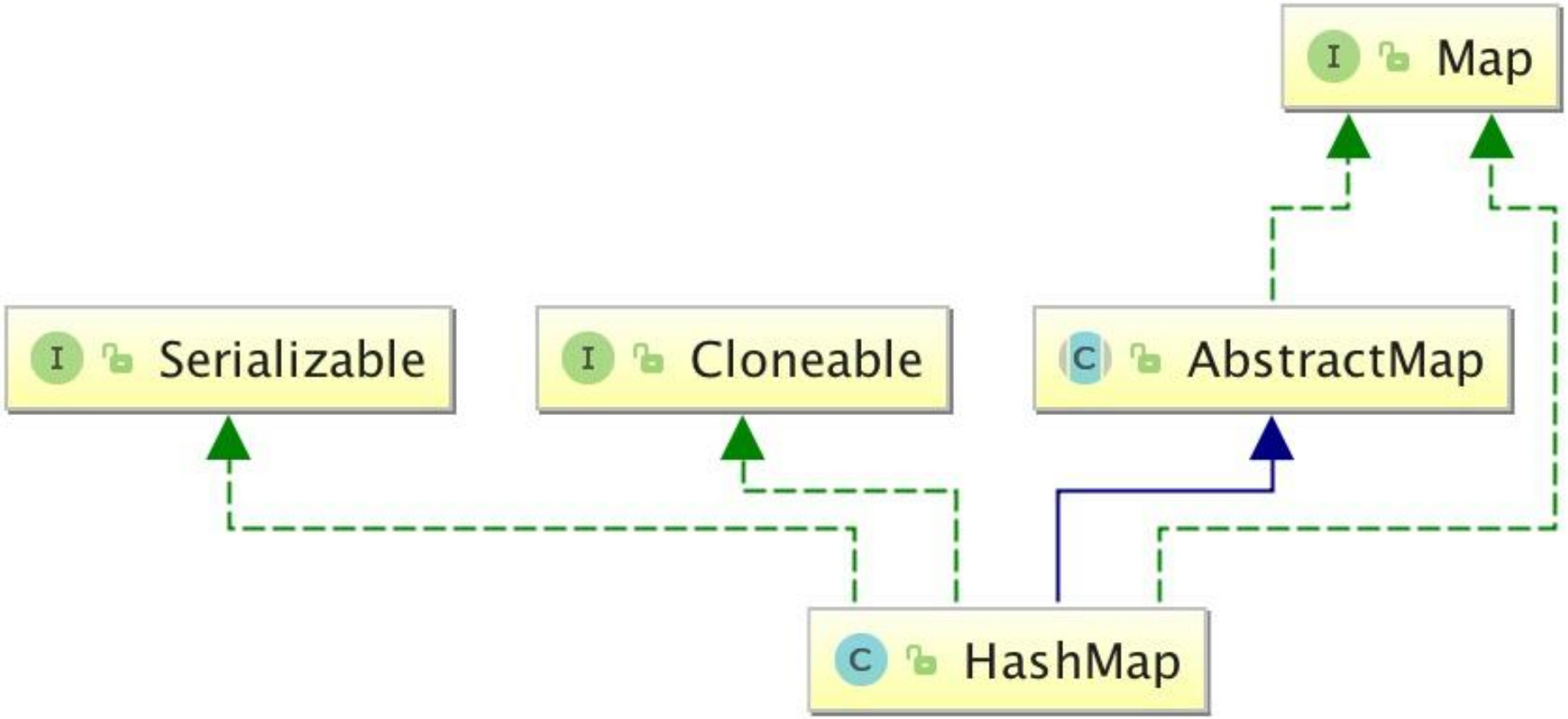
- 最基本的接口是Collection接口，该接口定义了操作数据的基本方法。
- 常用的集合有List集合、Set集合、Map集合，其中List与Set实现了Collection接口。各接口还提供了不同的实现类。



HashMap

- HashMap 是一个散列表，它存储的内容是键值对(key-value)映射。
- HashMap 实现了 Map 接口，根据键的 HashCode 值存储数据，具有很快的访问速度，最多允许一条记录的键为 null，不支持线程同步。
- HashMap 是无序的，即不会记录插入的顺序。
- HashMap 继承于AbstractMap，实现了 Map、Cloneable、java.io.Serializable 接口。

HashMap



HashMap

- HashMap 的 key 与 value 类型可以相同也可以不同，可以是字符串（String）类型的 key 和 value，也可以是整型（Integer）的 key 和字符串（String）类型的 value。
- HashMap 类位于 java.util 包中，使用前需要引入它，语法格式如下：

```
import java.util.HashMap; // 引入 HashMap 类
```

- 以下实例我们创建一个 HashMap 对象 Sites，整型（Integer）的 key 和字符串（String）类型的 value：

```
HashMap<Integer, String> Sites = new HashMap<Integer, String>();
```

HashMap

添加元素： HashMap 类提供了很多有用的方法， 添加键值对(key-value)可以使用 put() 方法：

```
// 引入 HashMap 类
import java.util.HashMap;
public class HashMapTest {
    public static void main(String[] args) {
        // 创建 HashMap 对象 Sites
        HashMap<Integer, String> Sites = new HashMap<Integer, String>();
        // 添加键值对
        Sites.put(1, "Google");
        Sites.put(2, “Amazon");
        Sites.put(3, "Taobao");
        Sites.put(4, "Zhihu");
        System.out.println(Sites);
    }
}
```

输出为： {1=Google, 2=Amazon, 3=Taobao, 4=Zhihu}

HashMap

访问元素：我们可以使用 get(key) 方法来获取 key 对应的 value:

```
// 引入 HashMap 类
import java.util.HashMap;
public class HashMapTest {
    public static void main(String[] args) {
        // 创建 HashMap 对象 Sites
        HashMap<Integer, String> Sites = new HashMap<Integer, String>();
        // 添加键值对
        Sites.put(1, "Google");
        Sites.put(2, “Amazon");
        Sites.put(3, "Taobao");
        Sites.put(4, "Zhihu");
        System.out.println(Sites.get(3));
    }
}
```

输出为：Taobao

删除元素:

我们可以使用 remove(key) 方法来删除 key 对应的键值对(key-value):

```
Sites.remove(4);
```

删除集合中所有元素可以使用 clear 方法:

```
sites.clear();
```

计算大小:

如果要计算 HashMap 中的元素数量可以使用 size() 方法:

```
sites.size()
```

HashMap

迭代 **HashMap**：可以使用 for-each 来迭代 HashMap 中的元素。

```
// 引入 HashMap 类
import java.util.HashMap;
public class HashMapTest {
    public static void main(String[] args) {
        // 创建 HashMap 对象 Sites
        HashMap<Integer, String> Sites = new HashMap<Integer, String>();
        // 添加键值对
        Sites.put(1, "Google");
        Sites.put(2, “Amazon");
        Sites.put(3, "Taobao");
        Sites.put(4, "Zhihu");
```

HashMap

迭代 **HashMap**：可以使用 for-each 来迭代 HashMap 中的元素。

```
        // 输出 key 和 value（如果你只想获取 key，可以使用 keySet() 方法）
        for (Integer i : Sites.keySet()) {
            System.out.println("key: " + i + " value: " + Sites.get(i));
        }
        // 输出 value 值（如果你只想获取 value，可以使用 values() 方法）
        for(String value: Sites.values()) {
          System.out.print(value + ", ");
        }
    }
}
```

输出为：

```
key: 1 value: Google
key: 2 value: Amazon
key: 3 value: Taobao
key: 4 value: Zhihu
Google, Amazon, Taobao, Zhihu,
```

第七章：集合与策略、迭代器模式

- 集合类概述
- List接口及其标准实现
- 类ArrayList与LinkedList
- Set与Map接口
- 策略模式
- 迭代器模式

策略模式

- **策略模式（Strategy Pattern）**

在有多种算法相似的情况下，使用多个 if-else 语句会使代码变得复杂和难以维护，而策略模式允许策略随着对象改变而改变。

- 很多时候我们可以有多种策略来实现目的
  - 诸葛亮的锦囊妙计，每一个锦囊就是一个策略，可以视情况使用不同策略
  - 旅行的出游方式，选择骑自行车、坐汽车，每一种旅行方式都是一个策略
  - ...

策略模式

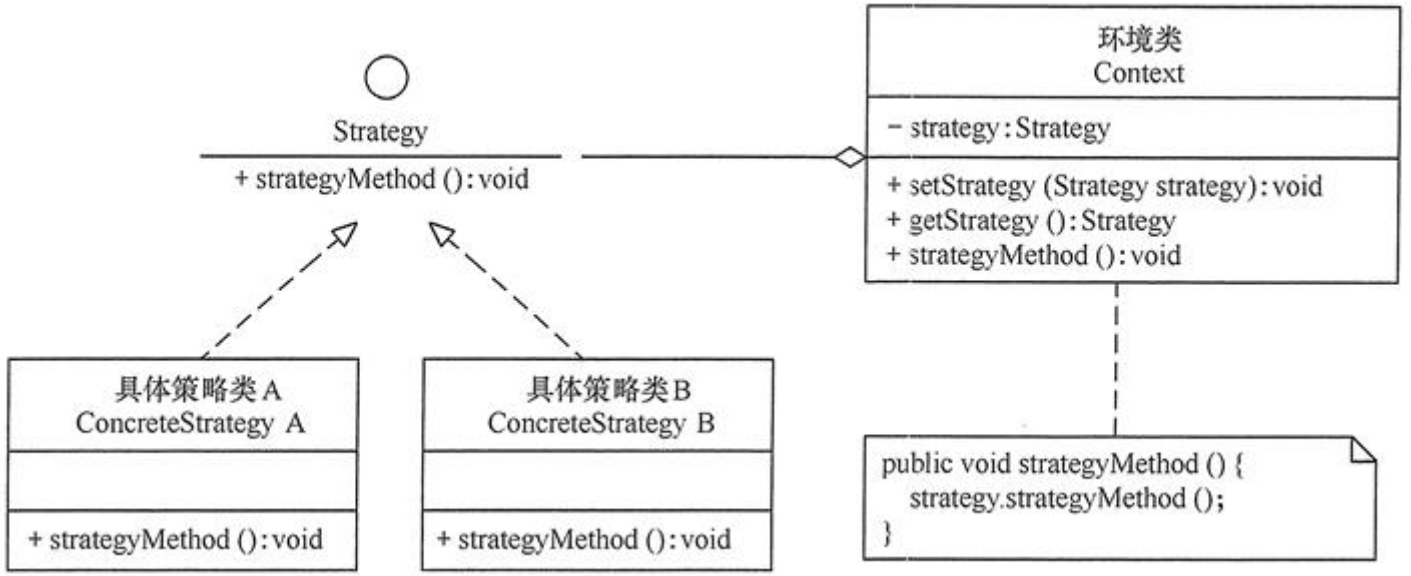
如何让算法和对象分开来，使得算法可以独立于使用它的客户而变化？

定义一系列的算法,把每一个算法封装起来, 并且使它们可相互替换, 从而使得算法可独立于使用它的客户而变化。并将逻辑判断移到Client中去（即由客户端决定在什么情况下使用什么具体策略）。

策略模式（Strategy Pattern）

策略模式涉及到三个角色：

- 抽象策略（Strategy）角色：这是一个抽象角色，通常由一个接口或抽象类实现。此角色给出所有的具体策略类所需的接口。
- 具体策略（ConcreteStrategy）角色：包装了相关的算法或行为。
- 环境（Context）角色：持有一个Strategy的引用。



策略模式（Strategy Pattern）

例子：假设现在要设计一个卖书的电子商务网站，本网站可能对所有的高级会员提供每本20%的促销折扣；对中级会员提供每本10%的促销折扣；对初级会员没有折扣。

根据描述，折扣是根据以下的几个算法中的一个进行的：

- 算法一：对初级会员没有折扣。
- 算法二：对中级会员提供10%的促销折扣。
- 算法三：对高级会员提供20%的促销折扣。

策略模式（Strategy Pattern）

抽象折扣类:

```
public interface MemberStrategy {
    /**
     * 计算图书的价格
     * @param booksPrice    图书的原价
     * @return    计算出打折后的价格
     */
    public double calcPrice(double booksPrice);
}
```

初级会员折扣类:

```
public class PrimaryMemberStrategy implements Member
Strategy {
    @Override
    public double calcPrice(double booksPrice) {
        System.out.println("对于初级会员的没有折扣");
        return booksPrice;
    }
}
```

策略模式（Strategy Pattern）

中级会员折扣类:

```
public class IntermediateMemberStrategy implements
MemberStrategy {
    @Override
    public double calcPrice(double booksPrice) {
        System.out.println("对于中级会员的折扣为10%"
);
        return booksPrice * 0.9;
    }
}
```

高级会员折扣类:

```
public class AdvancedMemberStrategy implements Mem
berStrategy {
    @Override
    public double calcPrice(double booksPrice) {
        System.out.println("对于高级会员的折扣为20%");
        return booksPrice * 0.8;
    }
}
```

策略模式（Strategy Pattern）

网站类:

```
public class Network { //持有一个具体的策略对象
    private MemberStrategy strategy;
    /**
    * 构造函数，传入一个具体的策略对象
    * @param strategy 具体的策略对象
    */
    public Network(MemberStrategy strategy){
        this.strategy = strategy;
    }
    /**
      *计算图书的价格
    * @param booksPrice 图书的原价
    * @return 计算出打折后的价格
    */
    public double quote(double booksPrice){
        return this.strategy.calcPrice(booksPrice);
    }
}
```

策略模式（Strategy Pattern）

客户端类:

```
public class Client { public static void main(String[] args) {
    //选择并创建需要使用的策略对象
    MemberStrategy strategy = new AdvancedMemberStrategy();
    //创建环境
    Network network = new Network(strategy);
    //计算价格
    double quote = network.quote(300);
    System.out.println("图书的最终价格为: " + quote); }
}
```

策略模式（Strategy Pattern）

回顾：

- 从上面的示例可以看出，策略模式封装了算法
- 方便新的算法插入到已有系统中，以及老算法从系统中“退休”，实现方法替换
- 策略模式并不决定在何时使用何种算法。在什么情况下使用什么算法是由客户端决定的。

策略模式（Strategy Pattern）

补充：

- 策略模式的重心：策略模式的重心不是如何实现算法，而是如何组织、调用这些算法，从而让程序结构更灵活，具有更好的维护性和扩展性。

- 算法的**平等性**：策略模式一个很大的特点就是各个策略算法的平等性。对于一系列具体的策略算法，大家的地位是完全一样的，正因为这个平等性，才能实现算法之间可以相互替换。所有的策略算法在实现上也是相互独立的，相互之间是没有依赖的。

策略模式（Strategy Pattern）

补充：

- 运行时策略的**唯一性**：运行期间，策略模式在每一个时刻只能使用一个具体的策略实现对象，虽然可以动态地在不同的策略实现中切换，但是同时只能使用一个。

- 公有的行为：经常见到的是，所有的具体策略类都有一些公有的行为。这时候，就应当把这些公有的行为放到共同的抽象策略角色Strategy类里面。当然这时候抽象策略角色必须要用Java抽象类实现，而不能使用接口。

策略模式（Strategy Pattern）

优点：

- 提供了一种替代继承的方法，而且既保持了继承的优点（代码重用）还比继承更灵活（算法独立，可以任意扩展）。

- 它把采取哪一种算法或采取哪一种行为的逻辑与算法本身分离，避免程序中使用多重条件转移语句，使系统更灵活，并易于扩展。

- 遵守大部分设计原则，高内聚、低偶合。

策略模式（Strategy Pattern）

缺点：

- 客户端必须知道所有的策略类，并自行决定使用哪一个策略类。这就意味着客户端必须理解这些算法的区别，以便适时选择恰当的算法类。换言之，策略模式只适用于客户端知道算法或行为的情况。

- 由于策略模式把每个具体的策略实现都单独封装成为类，如果备选的策略很多的话，那么对象的数目就会很可观。

策略模式

策略模式总结:

要素	描述
模式名 Pattern Name	策略模式（**Strategy Pattern**）
目的 Intent	在有多种算法相似的情况下，避免多个 if-else 语句所带来的复杂和难以维护。
问题 Problem	如何让算法和对象分开来，使得算法可以独立于使用它的客户而变化。
解决方案 Solution	定义一系列的算法,把每一个算法封装起来, 并且使它们可相互替换，从而使得算法可独立于使用它的客户而变化。
效果 Consequence	优点：1、算法可以自由切换。 2、避免使用多重条件判断。 3、扩展性良好。 缺点： 1、策略类会增多。 2、所有策略类都需要对外暴露。

第七章：集合与策略、迭代器模式

- 集合类概述
- List接口及其标准实现
- 类ArrayList与LinkedList
- Set与Map接口
- 策略模式
- 迭代器模式

迭代器模式

- **迭代器模式（Iterator Pattern）**

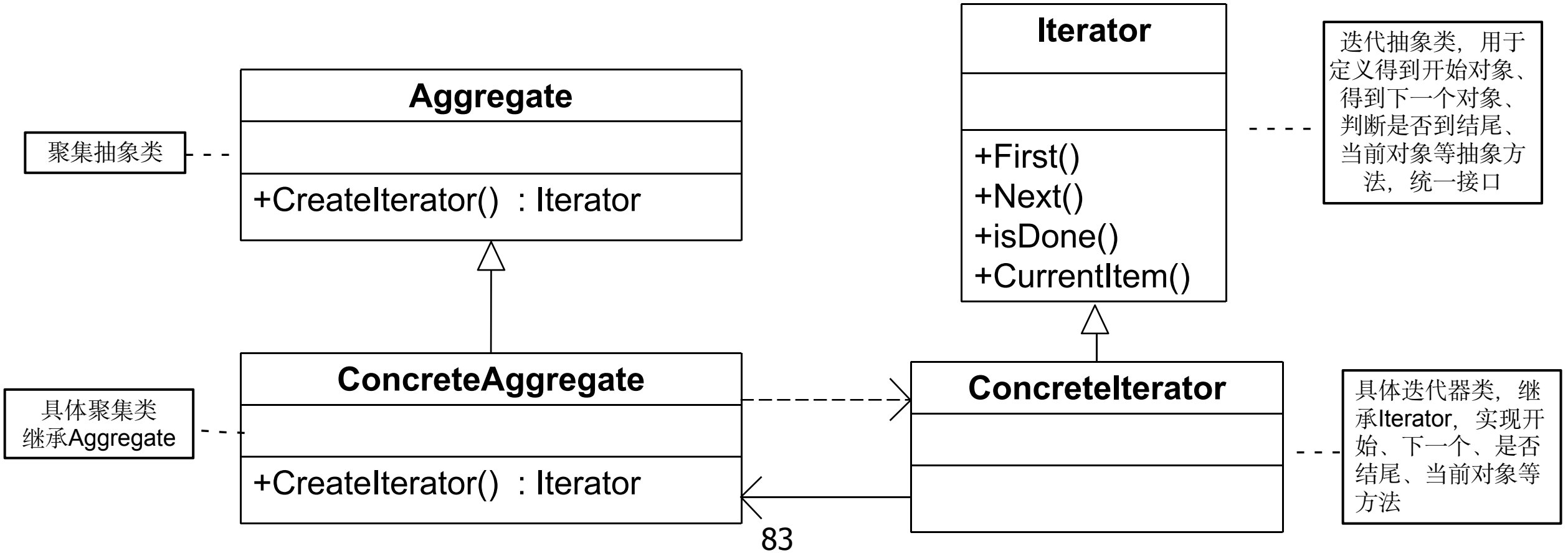
在客户访问类与聚合类之间插入一个迭代器，这分离了聚合对象与其遍历行为，对客户也隐藏了其内部细节，且满足“单一职责原则”和“开闭原则”。

- 迭代器模式在生活中应用的比较广泛
  - 物流系统中的传送带，不管传送什么物品，都打包成一个个箱子，并且有一个统一的二维码，在分发时只需要一个个检查发送的目的地即可。
  - 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信息。
  - ...

迭代器模式

如何将聚合对象与其遍历行为分离开?

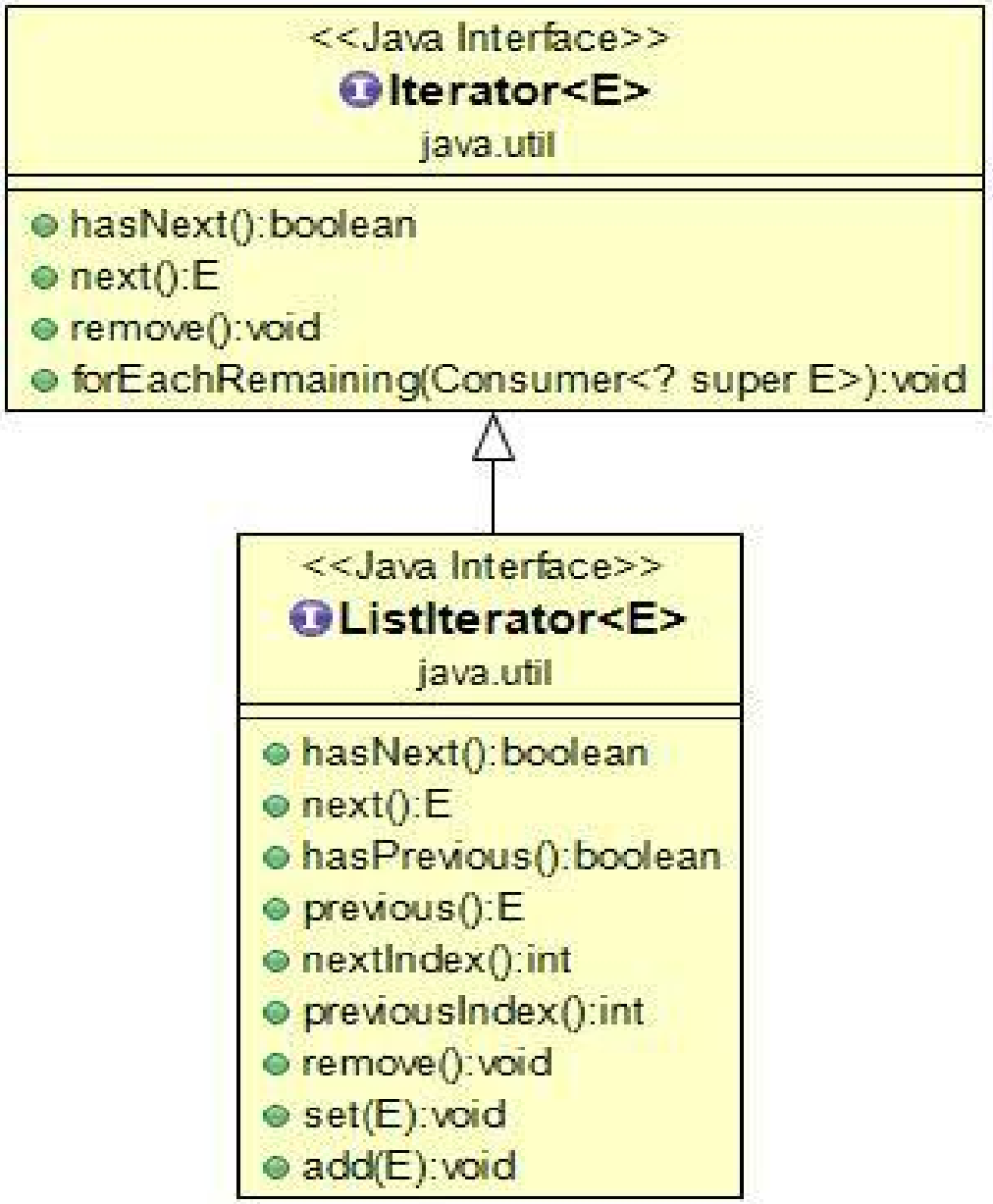
迭代器模式（Iterator Pattern）结构图



迭代器模式

- Java Iterator（迭代器）不是一个集合，它是一种用于访问集合的方法，可用于迭代 ArrayList和 HashSet 等集合。
- Iterator 是 Java 迭代器最简单的实现，ListIterator 是 Collection API 中的接口， 它扩展了 Iterator 接口。

迭代器模式


<<Java Interface>>
Iterator<E>
java.util
hasNext():boolean
next():E
remove():void
forEachRemaining(Consumer<? super E>):void
<<Java Interface>>
ListIterator<E>
java.util
hasNext():boolean
next():E
hasPrevious():boolean
previous():E
nextIndex():int
previousIndex():int
remove():void
set(E):void
add(E):void

迭代器模式

- 迭代器 it 的三个基本操作是 next 、hasNext 和 remove。
- 调用 it.next() 会返回迭代器的下一个元素，并且更新迭代器的状态。
- 调用 it.hasNext() 用于检测集合中是否还有元素。
- 调用 it.remove() 将迭代器返回的元素删除。
- Iterator 类位于 java.util 包中，使用前需要引入它，语法格式如下：

```
import java.util.Iterator; // 引入 Iterator 类
```

迭代器模式

实例：

```
// 引入 ArrayList 和 Iterator 类
import java.util.ArrayList;
import java.util.Iterator;
public class IteratorTest {
    public static void main(String[] args) {
        // 创建集合
        ArrayList<String> sites = new ArrayList<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        sites.add("Zhihu");
        // 获取迭代器
        Iterator<String> it = sites.iterator();
        // 输出集合中的第一个元素
        System.out.println(it.next());
    }
}
```

输出为：Google

迭代器模式

循环集合元素:让迭代器 it 逐个返回集合中所有元素最简单的方法是使用 while 循环:

```
while(it.hasNext()) {
        System.out.println(it.next());
}
```

输出为:

```
Google
Amazon
Taobao
Weibo
```

思考题

- 你会几种for循环?

```
for(int i = 0; i < myCollection.size(); i++) {
        Item element = myCollection.get(i);
        ...
}
```

```
for(Item element : myCollection) { ... }
```

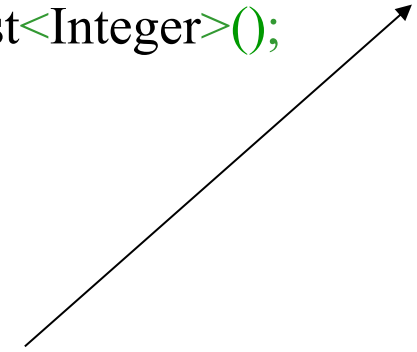
```
Iterator<Item> iterator = myCollection.iterator();
while(iterator.hasNext()) {
 Item element = iterator.next();
 ...
 }
```

迭代器模式

删除元素:要删除集合中的元素可以使用 remove() 方法。

以下实例我们删除集合中小于 10 的元素：

```
// 引入 ArrayList 和 Iterator 类
import java.util.ArrayList;
import java.util.Iterator;
public class IteratorTest {
    public static void main(String[] args) {
        ArrayList<Integer> numbers = new ArrayList<Integer>();
        numbers.add(12);
        numbers.add(8);
        numbers.add(2);
        numbers.add(23);
        Iterator<Integer> it = numbers.iterator();
```



```
        while(it.hasNext()) {
            Integer i = it.next();
            if(i < 10) {
                // 删除小于 10 的元素
                it.remove();
            }
        }
        System.out.println(numbers);
    }}
```

输出为：　[12, 23]

单选题 1分

设置

下面哪个关于for循环与迭代器iterator的描述是错误的?

A for循环可以支持快速获取指定位置的元素

B Iterator采用的是顺序访问的方法

C LinkedList里不适合使用iterator

D ArrayList里适合使用for循环

提交

单选题 1分

设置

下面哪个关于for循环与迭代器iterator的描述是错误的?

A for循环可以支持快速获取指定位置的元素

B Iterator采用的是顺序访问的方法

C LinkedList里不适合使用iterator

D ArrayList里适合使用for循环

提交

迭代器模式

迭代器模式总结:

要素	描述
模式名 Pattern Name	迭代器模式（**Iterator Pattern**）
目的 Intent	提供一种方法顺序访问一个聚合对象中各个元素，而又无须暴露该对象的内部表示。
问题 Problem	1、访问一个聚合对象的内容而无须暴露它的内部表示。 2、需要为聚合对象提供多种遍历方式。 3、为遍历不同的聚合结构提供一个统一的接口。
解决方案 Solution	把在元素之间游走的责任交给迭代器，而不是聚合对象。
效果 Consequence	优点：1）它支持以不同的方式遍历一个聚合对象。 2）迭代器简化了聚合类。 3）在同一个聚合上可以有多个遍历。 4）在迭代器模式中，增加新的聚合类和迭代器类都很方便，无须修改原有代码。

迭代器讨论

- 迭代器是一种模式，它可以使得对于序列类型的数据结构的遍历行为与被遍历的对象分离，即我们无需关心该序列的底层结构是什么样子的
  - 案例：如果使用 Iterator 来遍历集合中元素，一旦不再使用 List 转而使用 Set 来组织数据，那遍历元素的代码不用做任何修改；如果使用 for 来遍历，那所有遍历此集合的算法都得做相应调整,因为List有序,Set无序,结构不同,他们的访问算法也不一样
- 在循环的时候，采用Iterator可以删除元素；而无法在for循环中删除元素

比较

- 迭代器优点
  - 支持多种集合遍历
  - 封装性好，用户只需要得到迭代器就可以遍历，而对于遍历算法则不用去关心
- 迭代器缺点
  - 对于比较简单的遍历（数组或者有序列表），使用迭代器方式遍历较为繁琐而且遍历效率不高
  - 使用迭代器的方式比较适合那些底层以链表形式实现的集合

第七章：集合与策略、迭代器模式

- 集合类概述
- List接口及其标准实现
- 类ArrayList与LinkedList
- Set与Map接口
- 策略模式
- 迭代器模式