迭代器模式

题目链接

迭代器模式-学生名单

基本概念

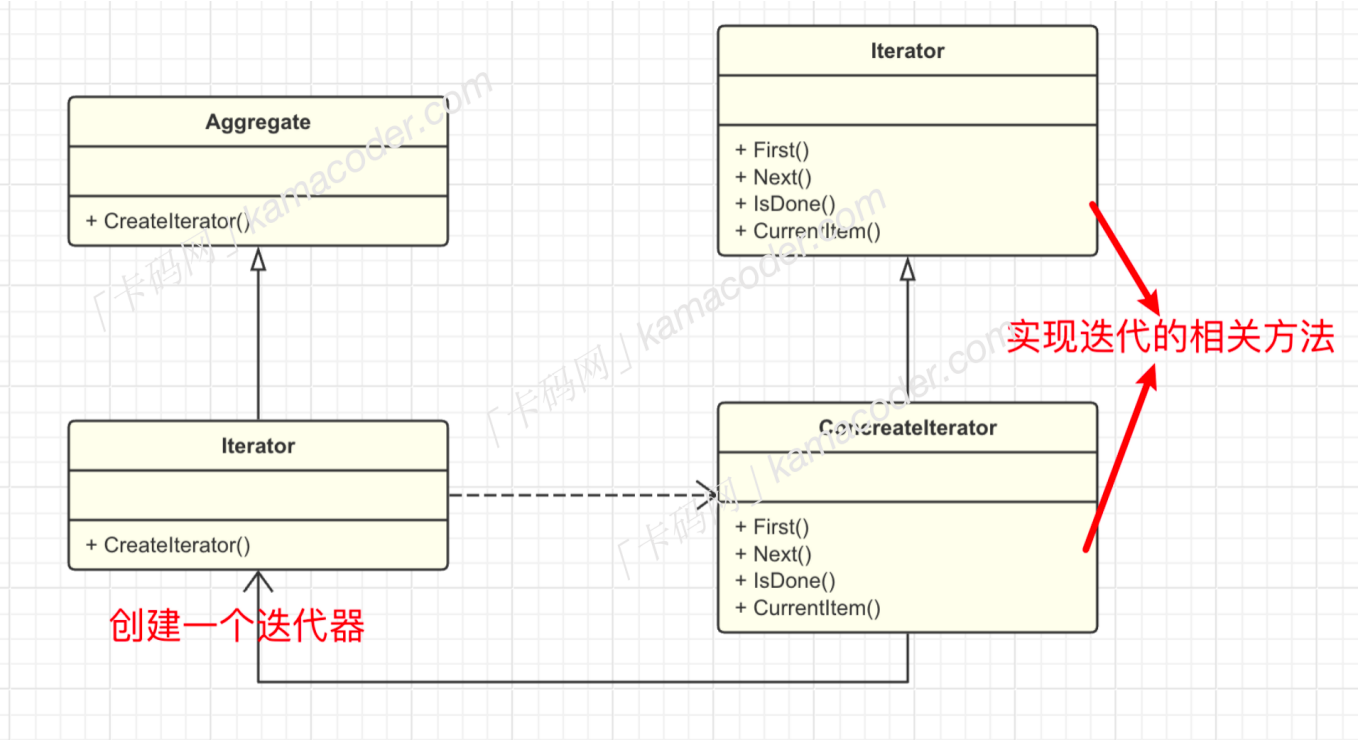
迭代器模式是一种行为设计模式，是一种使用频率非常高的设计模式，在各个语言中都有应用，其主要目的是**提供一种统一的方式来访问一个聚合对象中的各个元素，**而不需要暴露该对象的内部表示。通过迭代器，客户端可以顺序访问聚合对象的元素，而无需了解底层数据结构。

> *迭代器模式应用广泛，但是大多数语言都已经内置了迭代器接口，不需要自己实现。*

基本结构

迭代器模式包括以下几个重要角色

- 迭代器接口`Iterator`：定义访问和遍历元素的接口，通常会包括`hasNext()`方法用于检查是否还有下一个元素，以及`next()`方法用于获取下一个元素。有的还会实现获取第一个元素以及获取当前元素的方法。
- 具体迭代器`ConcreateIterator`：实现迭代器接口，实现遍历逻辑对聚合对象进行遍历。
- 抽象聚合类：定义了创建迭代器的接口，包括一个`createIterator`方法用于创建一个迭代器对象。
- 具体聚合类：实现在抽象聚合类中声明的`createIterator()`方法，返回一个与具体聚合对应的具体迭代器



简易实现

1. 定义迭代器接口：通常会有检查是否还有下一个元素以及获取下一个元素的方法。

```
// 迭代器接口
public interface Iterator{
    // 检查是否还会有下一个元素
    boolean hasNext();
    // 获取下一个元素
    Object next();
}
```

2. 定义具体迭代器：实现迭代器接口，遍历集合。

```
public class ConcreteIterator implements Iterator {
    private int index;
    private List<Object> elements;

    // 构造函数初始化迭代器
    public ConcreteIterator(List<Object> elements) {
        this.elements = elements;
        this.index = 0;
    }

    @Override
    public boolean hasNext() {
        return index < elements.size();
```

```
    }

    @Override
    public Object next() {
        if (hasNext()) {
            return elements.get(index++);
        }
        return null;
    }
}
```

3. 定义聚合接口：通常包括`createIterator()`方法，用于创建迭代器

```
public interface Iterable {
    Iterator createIterator();
}
```

4. 实现具体聚合：创建具体的迭代器

```
// 具体聚合
public class ConcreteIterable implements Iterable {
    private List<Object> elements;

    // 构造函数初始化可迭代对象
    public ConcreteIterable(List<Object> elements) {
        this.elements = elements;
    }

    @Override
    public Iterator createIterator() {
        return new ConcreteIterator(elements);
    }
}
```

5. 客户端使用

```
import java.util.ArrayList;
import java.util.List;

public class IteratorPatternExample {
    public static void main(String[] args) {
        List<Object> elements = new ArrayList<>();
        elements.add("Element 1");
        elements.add("Element 2");
        elements.add("Element 3");
```

```
        Iterable iterable = new ConcreteIterable(elements);
        Iterator iterator = iterable.createIterator();

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

使用场景

迭代器模式是一种通用的设计模式，其封装性强，简化了客户端代码，客户端不需要知道集合的内部结构，只需要关心迭代器和迭代接口就可以完成元素的访问。但是引入迭代器模式会增加额外的类，每增加一个集合类，都需要增加该集合对应的迭代器，这也会使得代码结构变得更加复杂。

许多编程语言和框架都使用了这个模式提供一致的遍历和访问集合元素的机制。下面是几种常见语言迭代器模式的实现。

1. Java语言：集合类（如ArrayList、LinkedList), 通过`Iterator`接口，可以遍历集合中的元素。

```
List<String> list = new ArrayList<>();
list.add("Item 1");
list.add("Item 2");
list.add("Item 3");

Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

2. Python语言：使用迭代器和生成器来实现迭代模式，`iter()`和`next()`函数可以用于创建和访问迭代器。

```
elements = ["Element 1", "Element 2", "Element 3"]
iterator = iter(elements)

while True:
    try:
        element = next(iterator)
        print(element)
    except StopIteration:
        break
```

3. C++语言：C++中的STL提供了迭代器的支持，`begin()`和`end()`函数可以用于获取容器的起始和结束迭代器。

```
#include <iostream>
#include <vector>

int main() {
    std::vector<std::string> elements = {"Element 1", "Element 2", "Element 3"};

    for (auto it = elements.begin(); it != elements.end(); ++it) {
        std::cout << *it << std::endl;
    }

    return 0;
}
```

4. JavaScript语言：ES6中新增了迭代器协议，使得遍历和访问集合元素变得更加方便。

```
// 可迭代对象实现可迭代协议
class IterableObject {
  constructor() {
    this.elements = [];
  }
  addElement(element) {
    this.elements.push(element);
  }
  [Symbol.iterator]() {
    let index = 0;
    // 迭代器对象实现迭代器协议
    return {
      next: () => {
        if (index < this.elements.length) {
          return { value: this.elements[index++], done: false };
        } else {
          return { done: true };
        }
      }
    };
  }
}

// 使用迭代器遍历可迭代对象
const iterableObject = new IterableObject();
iterableObject.addElement("Element 1");
```

```
iterableObject.addElement("Element 2");
iterableObject.addElement("Element 3");

for (const element of iterableObject) {
  console.log(element);
}
```

本题代码

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

// 可迭代对象接口
interface StudentCollection {
    java.util.Iterator<Student> iterator();
}

// 具体可迭代对象
class ConcreteStudentCollection implements StudentCollection {
    private List<Student> students = new ArrayList<>();

    public void addStudent(Student student) {
        students.add(student);
    }

    @Override
    public java.util.Iterator<Student> iterator() {
        return new ConcreteStudentIterator(students);
    }
}

// 迭代器接口
interface Iterator<T> {
    boolean hasNext();

    T next();
}

// 具体迭代器
class ConcreteStudentIterator implements java.util.Iterator<Student> {
    private List<Student> students;
    private int currentIndex = 0;

    public ConcreteStudentIterator(List<Student> students) {
        this.students = students;
```

```
    }

    @Override
    public boolean hasNext() {
        return currentIndex < students.size();
    }

    @Override
    public Student next() {
        if (hasNext()) {
            Student student = students.get(currentIndex);
            currentIndex++;
            return student;
        }
        return null;
    }
}

// 学生类
class Student {
    private String name;
    private String studentId;

    public Student(String name, String studentId) {
        this.name = name;
        this.studentId = studentId;
    }

    public String getName() {
        return name;
    }

    public String getStudentId() {
        return studentId;
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // 读取学生数量
        int n = scanner.nextInt();
        scanner.nextLine(); // 读取换行符

        // 创建具体可迭代对象
```

```
        ConcreteStudentCollection studentCollection = new
ConcreteStudentCollection();

        // 读取学生信息并添加到集合
        for (int i = 0; i < n; i++) {
            String[] input = scanner.nextLine().split(" ");
            if (input.length == 2) {
                String name = input[0];
                String studentId = input[1];
                Student student = new Student(name, studentId);
                studentCollection.addStudent(student);
            } else {
                System.out.println("Invalid input");
                return;
            }
        }

        // 使用迭代器遍历学生集合
        java.util.Iterator<Student> iterator =
studentCollection.iterator();
        while (iterator.hasNext()) {
            Student student = iterator.next();
            System.out.println(student.getName() + " " +
student.getStudentId());
        }
    }
}
```

其他版本代码

Java

Java 内置的 List 接口的 iterator() 方法实现。

```
import java.util.*;

// 学生类
class Student {
    String name;
    String id;

    Student(String name, String id) {
        this.name = name;
        this.id = id;
```

```
    }

    String getInfo() {
        return name + " " + id;
    }
}

// 学生列表系统
class StudentListSystem {
    private List<Student> students = new ArrayList<>();

    void addStudent(String name, String id) {
        students.add(new Student(name, id));
    }

    Iterator<Student> iterator() {
        return students.iterator();
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        StudentListSystem studentList = new StudentListSystem();

        int n = scanner.nextInt();
        scanner.nextLine();

        for (int i = 0; i < n; i++) {
            String[] s = scanner.nextLine().split(" ");
            studentList.addStudent(s[0], s[1]);
        }

        Iterator<Student> iterator = studentList.iterator();
        while (iterator.hasNext()) {
            Student student = iterator.next();
            System.out.println(student.getInfo());
        }
        scanner.close();
    }
}
```

C++

```cpp
#include <iostream>
#include <vector>

// 学生类
class Student {
public:
    Student(const std::string& name, const std::string& studentId) :
name(name), studentId(studentId) {}

    std::string getName() const {
        return name;
    }

    std::string getStudentId() const {
        return studentId;
    }

private:
    std::string name;
    std::string studentId;
};

// 可迭代对象接口
class StudentCollection {
public:
    virtual ~StudentCollection() = default;

    virtual std::vector<Student>::iterator begin() = 0;
    virtual std::vector<Student>::iterator end() = 0;
};

// 具体可迭代对象
class ConcreteStudentCollection : public StudentCollection {
public:
    void addStudent(const Student& student) {
        students.push_back(student);
    }

    std::vector<Student>::iterator begin() override {
        return students.begin();
    }

    std::vector<Student>::iterator end() override {
        return students.end();
```

```
    }

private:
    std::vector<Student> students;
};

int main() {
    int n;
    std::cin >> n;
    std::cin.ignore(); // 忽略换行符

    ConcreteStudentCollection studentCollection;

    for (int i = 0; i < n; ++i) {
        std::string name, studentId;
        std::cin >> name >> studentId;
        studentCollection.addStudent(Student(name, studentId));
    }

    // 使用迭代器遍历学生集合
    for (auto it = studentCollection.begin(); it !=
studentCollection.end(); ++it) {
        const Student& student = *it;
        std::cout << student.getName() << " " << student.getStudentId() <<
std::endl;
    }

    return 0;
}
```

Python

```
class Student:
    def __init__(self, name, student_id):
        self.name = name
        self.student_id = student_id

    def get_name(self):
        return self.name

    def get_student_id(self):
        return self.student_id


class StudentCollection:
    def __init__(self):
```

```
        self.students = []

    def add_student(self, student):
        self.students.append(student)

    def __iter__(self):
        return iter(self.students)


def main():
    n = int(input())

    student_collection = StudentCollection()

    for _ in range(n):
        inputs = input().split()
        if len(inputs) == 2:
            name, student_id = inputs
            student = Student(name, student_id)
            student_collection.add_student(student)
        else:
            print("Invalid input")
            return

    # 使用迭代器遍历学生集合
    for student in student_collection:
        print(student.get_name(), student.get_student_id())


if __name__ == "__main__":
    main()
```

Go

```
package main

import (
    "fmt"
    "bufio"
    "os"
    "strings"
)

// 可迭代对象接口
type StudentCollection interface {
    Iterator() Iterator
```

```
}

// 具体可迭代对象
type ConcreteStudentCollection struct {
    students []Student
}

func NewConcreteStudentCollection() *ConcreteStudentCollection {
    return &ConcreteStudentCollection{
        students: make([]Student, 0),
    }
}

func (c *ConcreteStudentCollection) AddStudent(student Student) {
    c.students = append(c.students, student)
}

func (c *ConcreteStudentCollection) Iterator() Iterator {
    return NewConcreteStudentIterator(c.students)
}

// 迭代器接口
type Iterator interface {
    HasNext() bool
    Next() Student
}

// 具体迭代器
type ConcreteStudentIterator struct {
    students    []Student
    currentIndex int
}

func NewConcreteStudentIterator(students []Student)
*ConcreteStudentIterator {
    return &ConcreteStudentIterator{
        students:    students,
        currentIndex: 0,
    }
}

func (i *ConcreteStudentIterator) HasNext() bool {
    return i.currentIndex < len(i.students)
}

func (i *ConcreteStudentIterator) Next() Student {
```

```
    if i.HasNext() {
        student := i.students[i.currentIndex]
        i.currentIndex++
        return student
    }
    return Student{}
}

// 学生类
type Student struct {
    Name      string
    StudentID string
}

// 主函数
func main() {
    scanner := bufio.NewScanner(os.Stdin)

    // 读取学生数量
    scanner.Scan()
    n := 0
    fmt.Sscanf(scanner.Text(), "%d", &n)

    // 创建具体可迭代对象
    studentCollection := NewConcreteStudentCollection()

    // 读取学生信息并添加到集合
    for i := 0; i < n; i++ {
        scanner.Scan()
        input := strings.Fields(scanner.Text())

        if len(input) == 2 {
            name := input[0]
            studentID := input[1]
            student := Student{Name: name, StudentID: studentID}
            studentCollection.AddStudent(student)
        } else {
            fmt.Println("Invalid input")
            return
        }
    }

    // 使用迭代器遍历学生集合
    iterator := studentCollection.Iterator()
    for iterator.HasNext() {
        student := iterator.Next()
```

```
            fmt.Printf("%s %s\n", student.Name, student.StudentID)
        }
    }
```