软件构造实验报告七

实验名称：迭代器模式编程实现

实验时间：2019.5.22

学号：E21614061

姓名：徐奕

所在院系：计算机科学与技术学院

所在专业：软件工程

【实验目的和要求】

实现一个链表类（自己定义数据结构，不要使用语言自带的数据结构），链表内部每个节点存放一个 int 变量和一个 double 变量。

利用迭代器模式，实现一个外部迭代器，可以从头到尾和从尾到头两种方式遍历这个链表，打印各个节点中的数据。

【实验原理】

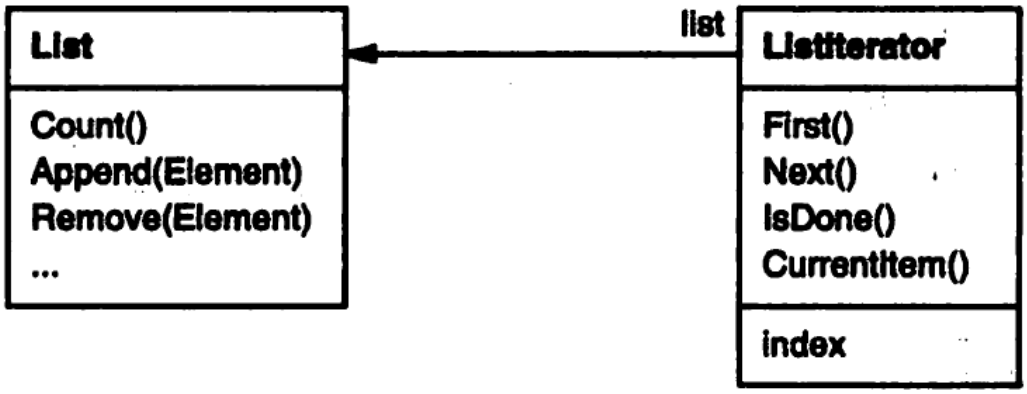
迭代器模式

简介

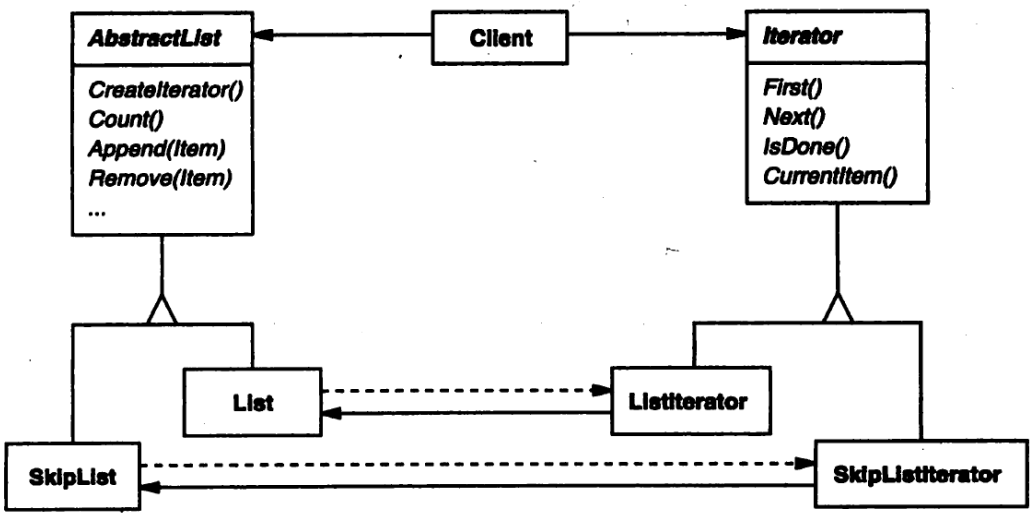
- 提供一种方法顺序访问一个聚合对象中各个元素，而又不暴露该对象的内部表示。

动机

- 一个聚合对象，如列表（List）应提供一种方法来让别人可以访问它的元素，而又不需要暴露它的内部结构。
- 针对不同的需要，可能要以不同方式遍历这个列表。
- 迭代器模式可以解决这些问题。这一模式的关键思想是将对列表的访问和遍历从列表对象中分离出来并放入一个迭代器（iterator）对象中。迭代器类定义了一个访问该列表元素的接口。
- 例如，一个列表类（List）可能需要一个列表迭代器类，它们之间的关系如下图：



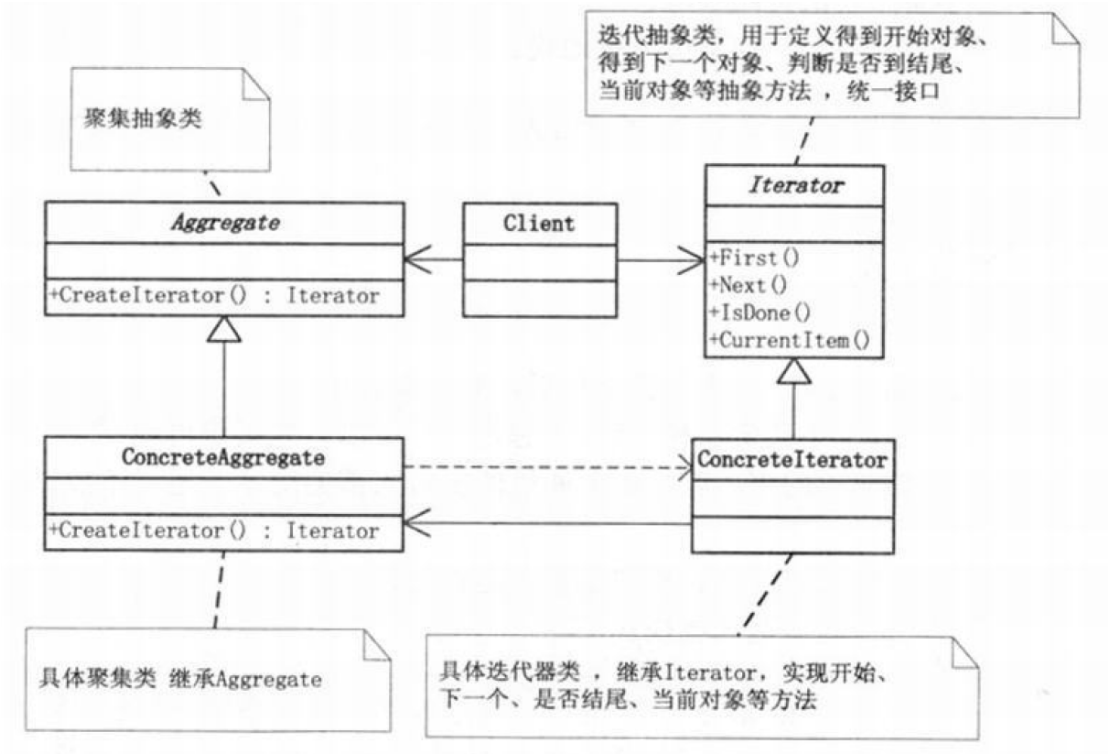
- 注意迭代器和列表是耦合在一起的，客户对象必须知道遍历是一个列表而不是其他聚合结构。
- 最好能有一种方法使得不需要改变客户代码即可改变聚合类。可以通过将迭代器概念推广到多态迭代来达到这个目标。



适用性

- 访问一个聚合对象的内部而无需暴露它的内部表示。
- 支持对聚合对象的多种遍历。
- 为遍历不同的聚合结构提供一个统一的接口。

结构



参与者

- Iterator
    - 迭代器定义访问和遍历元素的接口
- ConcreteIterator
    - 具体迭代器实现迭代器接口
    - 对该聚合遍历时跟踪当前位置
- Aggregate
    - 聚合定义创建相应迭代器对象的接口
- ConcreteAggregate

- 具体聚合实现创建相应迭代器的接口，该操作返回 ConcreteIterator 的一个适当的实例

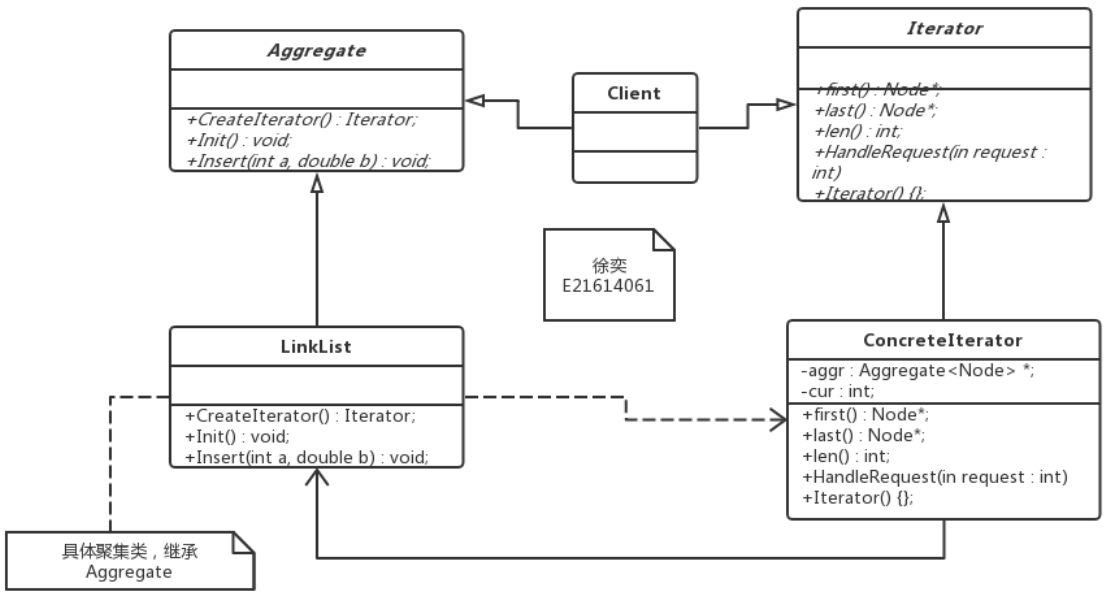
协作

- ConcreteIterator 跟踪聚合中的当前对象，并能够计算出待遍历的后继对象。

效果

- 它支持以不同方式遍历一个集合
- 迭代器简化了聚合的接口
- 在同一个聚合上可以有多个遍历

【实验 UML 图】



【实验代码与函数】

注意这里的 LinkList 即具体聚集类，继承 Aggregate。

```
#include <iostream>
using namespace std;

class Node{
private:
    int val1;
    double val2;
    Node *next;
    Node *pre;
public:
    Node(){
        next = NULL;
        pre = NULL;
    };
    Node(int x1, double x2){
        next = NULL;
        pre = NULL;
        val1 = x1;
        val2 = x2;
    }
    void SetNext(Node *n){
        next = n;
    };
    Node *GetNext(){
        return next;
    }
    void SetPre(Node *n){
        pre = n;
    }
    Node *GetPre(){
        return pre;
    }
    int getVal1(){
        return val1;
    }
    double getVal2(){
        return val2;
    }
};
```

```cpp
template<class Node>
class Iterator{
public:
    Iterator() {};
    virtual ~Iterator() {};
    virtual Node* first() = 0;
    virtual Node* last() = 0;
    virtual Node* next() = 0;
    virtual Node* pre() = 0;
    virtual Node* nowPos() = 0;
    virtual int len() = 0;
    virtual void SetDirect(bool s) = 0;
};

template<class Node>
class Aggregate{
public:
    Aggregate() {};
    virtual ~Aggregate() {};
    virtual Iterator<Node>* createIterator() = 0;
    virtual Node* GetFirst() = 0;
    virtual Node* GetLast() = 0;
    virtual int GetSum() = 0;
    virtual void Insert(int a, double b) = 0;
    virtual void Init() = 0;
};


template<class Node>
class ConcreteIterator : public Iterator<Node>{
public:
    ConcreteIterator(Aggregate<Node> *a) :aggr(a){
        cur = aggr->GetFirst()->GetNext();
    };
    ~ConcreteIterator() {};
    Node* first(){
        Node * p = new Node();
        p = aggr->GetFirst()->GetNext();
        return p;
    }

    Node* last(){
        Node * p = new Node();
        p = aggr->GetLast();
        return p;
```

```
    }

    Node* pre(){
        Node * p = cur->GetPre();
        cur = cur->GetPre();
        return p;
    }

    Node* nowPos(){
        return cur;
    }

    Node* next(){
        Node * p =  cur->GetNext();
        cur = cur->GetNext();
        return p;
    }

    void SetDirect(bool s){
        if(s){
            cur = aggr->GetFirst()->GetNext();
        }else{
            cur = aggr->GetLast();
        }
    }

    int len(){
        return aggr->GetSum();
    }

private:
    Aggregate<Node> *aggr;
    Node* cur;
};


template<class Node>
class LinkList : public Aggregate<Node>{
private:
    Node* first;
    Node* last;
    int sum;
public:
    Iterator<Node>* createIterator(){
        return new ConcreteIterator<Node>(this);
```

```
    }
    LinkList(){
        first = NULL;
        last = NULL;
        sum = 0;
    }
    Node* GetFirst(){
        return first;
    }
    Node* GetLast(){
        return last;
    }
    int GetSum(){
        return sum;
    }
    void Init(){
        first = new Node();
    }
    void Insert(int a, double b){
        Node *p = first;
        Node *s = new Node(a, b);
        for (int i = 0; i < sum; i++){
            p = p->GetNext();
        }
        s->SetNext(NULL);
        s->SetPre(p);
        p->SetNext(s);
        last = s;
        sum++;
    }
    ~LinkList(){}
};


int main(){
    Aggregate<Node> * aggr = new LinkList<Node>();
    aggr->Init();
    aggr->Insert(1, 2.1);
    aggr->Insert(2, 4.2);
    aggr->Insert(3, 6.3);
    aggr->Insert(4, 8.4);
    aggr->Insert(8, 16.5);
```

```
    Iterator<Node> * it = aggr->createIterator();

    cout<<"正序"<<endl;
    it->SetDirect(true);
    while(it->nowPos()!= NULL){
        cout<<it->nowPos()->getVal1()<<"
"<<it->nowPos()->getVal2()<<endl;
        it->next();
    }
    cout<<"倒序"<<endl;
    it->SetDirect(false);
    do{
        cout<<it->nowPos()->getVal1()<<"
"<<it->nowPos()->getVal2()<<endl;
        it->pre();
    }while(it->nowPos()!= it->first()->GetPre());
    return 0;
}
```

【实验结果】

```
XYs-MacBook-Pro:exp7 reacubeth$ cd "/Users/reacubeth/De
eacubeth/Desktop/exp7/"linklist
正序
1  2.1
2  4.2
3  6.3
4  8.4
8  16.5

倒序
8  16.5
4  8.4
3  6.3
2  4.2
1  2.1
```

【实验总结】

①由于本次实验的迭代器是链表，并且需要从头到尾或从尾到头遍历，因此这个链表实质上是一个双向链表。

②需要用迭代器进行遍历，而不是用链表本身的结构进行更新，因此迭代器需要封装，前一个，后一个，当前节点，头结点，尾节点位置即可。

③由于是链表，因此在创建链表和插入时需要给节点动态分配空间，否则无法正常操作。