This repository Search Pull requests Issues Gist

huyilong / OOP-Design

Unwatch 1 Unstar 1 Fork 0

Code Issues 0 Pull requests 0 Wiki Pulse Graphs Settings

Branch: master OOP-Design / ****Iterator&&SerializeSummary.java

Find file Copy path

huyilong SUMMARY - ALLLL **Iterator&&Serialize&&OOP** Problems 7bdf4eb 10 seconds ago

1 contributor

524 lines (417 sloc) 15.8 KB

Raw Blame History

```

Implement an iterator to flatten a 2d vector.
For example,
Given 2d vector =

[
  [1,2],
  [3],
  [4,5,6]
]

By calling next repeatedly until hasNext returns false,
the order of elements returned by next should be: [1,2,3,4,5,6].

How many variables do you need to keep track?
Two variables is all you need. Try with x and y.
Beware of empty rows. It could be the first few rows.
To write correct code, think about the invariant to maintain. What is it?
The invariant is x and y must always point to a valid point in the 2d vector. Should you maintain your invariant ahead of time or right whe
Not sure? Think about how you would implement hasNext(). Which is more complex?
Common logic in two different places should be refactored into a common method.

class Vector2D {
        private Iterator<List<Integer>> row = null;
    private Iterator<Integer> col = null;
    public Vector2D(List<List<Integer>> vec2d) {
        row = vec2d.iterator();
        if(row.hasNext())
            col = row.next().iterator();
    }

    public int next() {
        int lastValue = col.next();
        return lastValue;
    }

    public boolean hasNext() {
        if(col == null) {
            return false;
        }
        if(col.hasNext()) {
            return true;
        } else {
            while(row.hasNext()) {
                col = row.next().iterator();
                if(col.hasNext())
                    return true;
            }
            return false;
        }
    }
}

/**
 * Your Vector2D object will be instantiated and called as such:
 * Vector2D i = new Vector2D(vec2d);
 * while (i.hasNext()) v[f()] = i.next();
 */

// Java Iterator interface reference:
// https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html
class PeekingIterator implements Iterator<Integer> {
    //Here is an example. Assume that the iterator is initialized to the beginning of the list: [1, 2, 3].
    private Iterator<Integer> iter = null;
    private int nextValue = 0;
    //we need a end variable to make sure it is not end right now
    private boolean end = false;


    public PeekingIterator(Iterator<Integer> iterator) {
        // initialize any member here.
         this.iter = iterator;
         //check if there is any element
         if(iterator.hasNext()) {
             //initilize the nextValue in constructor as well
             nextValue = iterator.next();
         }else {
             end = true;
         }

    }

    // Returns the next element in the iteration without advancing the iterator.
    public Integer peek() {
         if(end == false){
             //return the buffer for next value
             //if the boolean for this buffer mark is false
             return nextValue;
         }else{
             //throw new NoSuchElementException();
             return 0;
         }
    }

    // hasNext() and next() should behave the same as in the Iterator interface.
    // Override them if needed.
    @Override
    public Integer next() {
        //buffer the real next
        int current = nextValue;
        if(iter.hasNext()){
            //update the nextval
            //correspondingly after next() operation
            //we update nextValue in next() right after the operation
            nextValue = iter.next();
        }else{
            end = true;
        }

        return current;//return the buffer
    }

    @Override
    public boolean hasNext() {
        return end != true;
    }
}

Zigzag Iterator
Total Accepted: 964 Total Submissions: 2714 Difficulty: Medium
Given two 1d vectors, implement an iterator to return their elements alternately.
For example, given two 1d vectors:
v1 = [1, 2]
v2 = [3, 4, 5, 6]
By calling next repeatedly until hasNext returns false, the order of elements returned by next should be: [1, 3, 2, 4, 5, 6].
Follow up: What if you are given k 1d vectors? How well can your code be extended to such cases?
Clarification for the follow up question - Update (2015-09-18):
The "Zigzag" order is not clearly defined and is ambiguous for k > 2 cases. If "Zigzag" does not look right to you, replace "Zigzag" with "
[1,2,3]
[4,5,6,7]
[8,9]
It should return [1,4,8,2,5,9,3,6,7].
[思路]
iterator都放到一个list里，用一个count循环，

public class ZigzagIterator {
    List<Iterator<Integer> > iters = new ArrayList<Iterator<Integer> >();

    int count = 0;

    public ZigzagIterator(List<Integer> v1, List<Integer> v2) {
        if( !v1.isEmpty() ) iters.add(v1.iterator());
        if( !v2.isEmpty() ) iters.add(v2.iterator());
    }

    public int next() {
        int x = iters.get(count).next();
        if(!iters.get(count).hasNext()) iters.remove(count);
        else count++;

        if(iters.size()!=0) count %= iters.size();
        return x;
    }

    public boolean hasNext() {
        return !iters.isEmpty();
    }
}

/**
 * Your ZigzagIterator object will be instantiated and called as such:
 * ZigzagIterator i = new ZigzagIterator(v1, v2);
 * while (i.hasNext()) v[f()] = i.next();
 */

Design an algorithm to encode a list of strings to a string.
The encoded string is then sent over the network and is decoded back
to the original list of strings.

Machine 1 (sender) has the function:
string encode(vector<string> strs) { // ... your code return encoded_string; }
Machine 2 (receiver) has the function:
vector<string> decode(string s) { //... your code return strs; }


So Machine 1 does:
string encoded_string = encode(strs);
and Machine 2 does:
vector<string> strs2 = decode(encoded_string);
strs2 in Machine 2 should be the same as strs in Machine 1.

Note: The string may contain any possible characters out of 256 valid ascii characters.

时间 O(N) 空间 O(1)

本题难点在于如何在合并后的字符串中，区分出原来的每一个子串。这里我采取的编码方式，
是将每个子串的长度先赋在前面，然后用一个#隔开长度和子串本身。这样我们先读出长度，
就知道该读取多少个字符作为子串了。


public class Codec {

    // Encodes a list of strings to a single string.
    public String encode(List<String> strs) {
        StringBuilder output = new StringBuilder();
        for(String str : strs){
            // 对于每个子串，先把其长度放在前面，用#隔开
            output.append(String.valueOf(str.length())+"#");
            // 再把子串本身放在后面
            output.append(str);
        }
        return output.toString();
    }

    // Decodes a single string to a list of strings.
    public List<String> decode(String s) {
        List<String> res = new LinkedList<String>();
        int start = 0;
        while(start < s.length()){
            // 找到从start开始的第一个#，这个#前面是长度
            int idx = s.indexOf('#', start);
            int size = Integer.parseInt(s.substring(start, idx));
            // 根据这个长度截取子串
            res.add(s.substring(idx + 1, idx + size + 1));
            // 更新start为子串后面一个位置
            start = idx + size + 1;
        }
        return res;
    }
}

```

```

CHAR ARR -> String
String.valueOf(char_arr)

valueOf(boolean b): Returns the string representation of the boolean argument.

valueOf(char c): Returns the string representation of the char argument.

valueOf(char[] data): Returns the string representation of the char array argument.

/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */

 import java.util.StringTokenizer;
public class Codec {
//Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memor


    // Encodes a tree to a single string.
    public String serialize(TreeNode root) {
         StringBuilder s = new StringBuilder();
         if(root == null){
             return s.toString();
         }

         helper1(s, root);
         return s.toString();
    }
    public void helper1(StringBuilder s, TreeNode n){
        if(n==null){
            s.append("#,");
        }else{
            s.append(n.val).append(",");
            helper1(s, n.left);
             helper1(s, n.right);
        }

        // s.append(n.val).append(",");
        // if(n.left !=null){
        //     helper1(s, n.left);

        // }

        // if(n.right!=null){
        //     helper2(s, n.right);
        // }
    }

    // Decodes your encoded data to tree.
    public TreeNode deserialize(String data) {
        if(data ==null || data.length() ==0){
            return null;
        }

        StringTokenizer t = new StringTokenizer(data, ",");
        return helper2(t);
    }

    //this should finally return the root of the tree
    public TreeNode helper2(StringTokenizer t){
        if(! t.hasMoreTokens()) return null;
        //You almost always want to useObjects.equals(). In the rare situation where you know you're dealing with interned strings, you can


        String val = t.nextToken();//delimited by the ","
        if(val.equals("#")){
            return null;
        }

        TreeNode root = new TreeNode(Integer.parseInt(val));
        root.left = helper2(t);
        root.right = helper2(t);

        //return the root of the entire tree

        return root;
    }
}

// Your Codec object will be instantiated and called as such:
// Codec codec = new Codec();
// codec.deserialize(codec.serialize(root));


Private Members in a Superclass

A subclass does not inherit the private members of its parent class.
However, if the superclass has public or protected methods for accessing
its private fields, these can also be used by the subclass.

A nested class has access to all the private members of its enclosing
class—both fields and methods. Therefore, a public or protected nested
class inherited by a subclass has indirect access to all of the private
members of the superclass.

only inherits the protected fields and variables

Here is the sample code for a possible implementation of a Bicycle
 class that was presented in the Classes and Objects lesson:

public class Bicycle {

    // the Bicycle class has three fields
    public int cadence;
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    // the Bicycle class has four methods
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }

}

public class MountainBike extends Bicycle {
        Bicycle is not abstract and all the fields in it should be protected
    // the MountainBike subclass adds one field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int startHeight,
                        int startCadence,
                        int startSpeed,
                        int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}

MountainBike inherits all the fields and methods of Bicycle and adds
```

```
the field seatHeight and a method to set it. Except for the constructor,
it is as if you had written a new MountainBike class entirely from scratch,
with four fields and five methods. However, you didn't have to do all the work.
This would be especially valuable if the methods in the Bicycle class
were complex and had taken substantial time to debug.


An animal shelter holds only dogs and cats, and operate FIFO - people must get the oldest one
Design a queue for the shelter that could hold both dogs and cats

public abstract class Animal{
    private int order;
    protected String name;
    public Animal(String n){
        this.name = n;
        //name=n;
    }

    public void setOrder(int ord){
        order = ord;
    }

    public int getOrder(){
        return order;
    }

    //or we could use a priority queue
    public boolean isOlderThan(Animal o){
        //order is private so when we implement compareTo
        //we must use o.getOrder()
        return this.order < o.getOrder();
    }
}

public class Dog extends Animal{
    public Dog(String s){
        super(s);
    }
}

public class Cat extends Animal{
    public Cat(String s){
        super(s);
    }
}

Casting Objects

We have seen that an object is of the data type of the class from
which it was instantiated. For example, if we write

public MountainBike myBike = new MountainBike();
then myBike is of type MountainBike.

MountainBike is descended from Bicycle and Object. Therefore, a MountainBike
is a Bicycle and is also an Object, and it can be used wherever Bicycle or
Object objects are called for.

The reverse is not necessarily true: a Bicycle may be a MountainBike, but
it isn't necessarily. Similarly, an Object may be a Bicycle or a MountainBike,
but it isn't necessarily.

Casting shows the use of an object of one type in place of another type,
among the objects permitted by inheritance and implementations. For example,
if we write

Object obj = new MountainBike();

then obj is both an Object and a MountainBike (until such time as obj is assigned
 another object that is not a MountainBike). This is called IMPLICIT casting.

If, on the other hand, we write

MountainBike myBike = obj;
we would get a compile-time error because obj is not known to the compiler to
be a MountainBike. However, we can tell the compiler that we promise to assign
a MountainBike to obj by EXPLICT casting:

MountainBike myBike = (MountainBike)obj;

```

```
This cast inserts a runtime check that obj is assigned a MountainBike so
that the compiler can safely assume that obj is a MountainBike. If obj is not
a MountainBike at runtime, an exception will be thrown.

Note: You can make a logical test as to the type of a particular object using
the instanceof operator. This can save you from a runtime error owing to an
improper cast. For example:
if (obj instanceof MountainBike) {
    MountainBike myBike = (MountainBike)obj;
}
Here the instanceof operator verifies that obj refers to a MountainBike so that we can make the cast with knowledge that there will be no r


public class AnimalQueue{
    LinkedList<Dog> dogs = new LinkedList<>();
    LinkedList<Cat> cats = new LinkedList<>();
    Cannot do LinkedList<Animal> because it is abstract
    however we could also make dog and cat extend a non-abstract class

    private int order = 0;

    public void enqueue( Animal o ){
        o.setOrder(order);
        order++;//timestamp

        if(o instanceof Dog){
            EXPLICT casting here otherwise compile error
            dogs.addLast((Dog) o);
        }

        if(o instanceof Cat){
            cats.addLast((Cat) o);
        }

    }

    public Animal dequeueAny(){
        IMPORTANT
        if(dogs.size() == 0){
            return dequeueCats();
        }else if(cats.size() == 0){
            return dequeueDogs();
        }
        not empty
        Dog dog = dogs.peek();
        Cat cat = cats.peek();
        if(dog.isOlderThan(cat)){
            return dequeueDogs();
        }else{
            return dequeueCats;
        }
    }

    public Dog dequeueDogs(){
        return dogs.poll();
    }

    public Cat dequeueCats(){
        return cats.poll();
    }
}
```