表、栈和队列

2019年4月3日

18:57

**Collection接口**

public interface Collection<AnyType> extends Iterable<AnyType>

{

int size( );

boolean isEmpty( );

void clear( );

boolean contains(AnyType x );

boolean add(AnyType x );

boolean remove(AnyType x );

java.util.Iterator<AnyType> iterator( );

}

size返回集合中的项数；

isEmpty返回true当且仅当集合的大小为0；

如果x在集合中，则contains返回true；这个接口并不直接规定集合如何决定x是否属于该集合——这要由实现该Collection接口的具体的类来确定；

add和remove从集合中添加和删除x，如果操作成功则返回true，如果因某个看似有理(非异常(能通过编译))的原因失败则返回false。

Collection接口扩展了Iterable接口。实现Iterable接口的那些类可以拥有增强的for循环

public static <AnyType> void print (Collection<AnyType> coll)

{

for(AnyType item : coll)

System.out.println(item);

}

**简单链表**

为了避免插入和删除的线性开销，我们需要保证表可以不连续存储，否则表的每个部分都可能需要整体移动。

链表由一系列节点组成，这些节点不必存在内存中相连。每一个节点均含有表元素和到包含该元素后继元的节点的链。我们称之为next链。最后一个单元的next链引用null.

remove通过修改一个next引用来实现。

insert方法需要使用new操作符从系统取得一个新节点，此后执行两次引用的调整。

**Iterator接口**

实现Iterable接口的集合必须提供一个称为iterator的方法，该方法返回一个Iterator类型的对象。

Iterator接口的思路是，通过iterator方法，每个集合均可创建并返回给客户一个实现Iterator接口的对象，并将当前位置的概念在对象内部存储下来。

每次对next的调用都给出集合的下一项。因此，第一次调用next给出第一项(数组第一个，下标为0)，第二次调用给出第二项，等等。

hasNext给出是否存在下一项。当编译器见到一个正在用于Iterable的对象的增强的for循环的时候，它用对iterator方法的那些调用代替增强for循环以得到一个Iterator对象，然后调用next和hasNext。因此，前面看到的print由编译器重写.

public static <AnyType> void print (Collection<AnyType> coll)

{

Iterator<AnyType> itr = col.iterator( );

while( itr.hasNext( ))

{

AnyType item = itr.next( );

System.out.println(item);

}

}

Iterator接口还包含一个方法，叫做remove。该方法可以删除由next最新返回的项(此后，我们不能再调用remove，直到对next再一次调用以后)

**List接口、ArrayList类和LinkedList类**

public interface List<AnyType> extends Collection<AnyType>

{

AnyType get (int idx);

AnyType set(int idx,AnyType newVal);

void add(int idx,AnyType x);

void remove(int idx);

ListIterator<AnyType> listIterator(int pos);

}

get和set可以访问或改变通过由位置索引idx给定的表中指定位置上的项。索引0位于表的前端。索引size( )-1代表表中的最后一项，而索引size( )则表示新添加的项可以被放置的位置。

add使得在位置idx处置入一个新的项，并把其后的项向后推移一个位置。

除以AnyType作为参数的标准的remove外，remove还被重载以删除指定位置上的项。

List ADT有两种流行的实现方式。ArrayList类提供List ADT的一种可增长数组的实现。使用ArrayList的优点在于，对get和set的调用花费常数时间。其缺点是新项的插入和现有项的删除代价昂贵，除非变动是在ArrayList的末端进行。LinkedList类则提供了List ADT的双链表实现。使用LinkedList的优点在于，新项的插入和现有项的删除均开销很小。使用LinkedList的缺点是它不容易索引，因此对get的调用是昂贵的，除非调用非常接近表的端点。

**ArrayList类的实现**

为避免与类库中的类相混，这里将把我们的类叫做MyArrayList,存在以下主要细节。

1.MyArrayList将保持基础数组，数组的容量，以及存储在MyArrayList中的当前项数

2.MyArrayList将提供一种机制以改变基础数组的容量。通过获得一个新的数组，将老数组拷贝到新数组中来改变数组的容量，允许虚拟机回收老数组。

3.MyArrayList将提供get和set的实现

4.MyArrayList将提供基本的例程，如size、isEmpty和clear，还提供remove，以及两种不同版本的add.

5.MyArrayList将提供一个实现Iterator接口的类。这个类存储迭代序列中的下一项的下标，并提供next、hasNext、和remove等方法的实现。MyArrayList的迭代器方法直接返回实现Iterator接口的该类的新构造的实例。

public class MyArrayList<AnyType> implements Iterable<AnyType>

{

private static final int DEFAULT\_CAPACITY = 10;

private int theSize;//属性列表长度

private AnyType [ ] theItems;

public MyArrayList( )//初始化方法

{doClear( );}

public void clear( )//clear方法

{doClear( );}

private void doClear( )//实际clear操作，令长度为0，缩短数组长度为默认长度10

{theSize = 0;ensureCapacity(DEFAULT\_CAPACITY);}

public int size( )//获取长度方法，返回属性theSize

{return theSize;}

public boolean isEmpty( )//判读数组为空方法，调用size( )方法，如果size( )返回值为0则返回true否则返回false

{return size( ) == 0;}

public void trimToSize( )

{ensureCapacity( size( ) );}

public AnyType get( int idx )//通过下标获取元素方法，如果超出数组长度则抛出异常，否则返回对应下标元素

{

if( idx < 0 || idx >= size( ) )

throw new ArrayIndexOutOfBounsException( );

return theItems[ idx ];

}

public AnyType set( int idx, AnyType newVal )//改变元素值方法，如果超出数组长度则抛出异常，否则改变对应下标元素

{

if( idx < 0 || idx >= size( ) )

throw new ArrayIndexOutOfBounsException( );

AnyType old = theItems[ idx ];

theItems[ idx ] = newVal;

return old;

}

public void ensureCapacity( int newCapacity )//收缩基础数组方法，

{

if( newCapacity < theSize )

return;

AnyType [ ] old = theItems;

theItems = (AnyType [ ]) new Object[ newCapacity ];

for( int i = 0; i < size( ); i++ )

theItems[ i ] = old[ i ];

}

public boolean add( AnyType x )//在列表末端添加元素方法

{

add( size( ), x );

return true;

}

public void add( int idx, AnyType x )//可以增加列表长度的添加元素方法，新数组长度变为原来的两倍，+1是为了防止长度为0

{

if( theItems.length == size( ) )

ensureCapacity( size( ) \* 2 + 1 );

for( int i = theSize; i > idx; i-- )

theItems[ i ] = theItems[ i - 1 ];

theItems[ idx ] = x;

theSize++;

}

public AnyType remove( int idx ) //删除列表元素方法，并把要删除元素之后的元素位置向前移动一个位置

{

AnyType removedItem = theItems[ idx ];

for( int i = idx; i < size( ) - 1; i++)

theItems[ i ] = theItems[ i + 1 ];

theSize--;

return removedItem;

}

public java.util.Iterator<AnyType> iterator( )//返回ArrayListIterator类的一个实例

{ return new ArrayListIterator( ); }

private class ArrayListIterator implements java.util.Iterator<AnyType>//实现存储当前位置，hasNext,next和remove方法的实现

{

private int current = 0;

public boolean hasNext( )

{ return current < size( ); }

public AnyType next( )

{

if( !hasNext( ) )

throw new java.util.NoSuchElementException( );

return theItems[ current++ ];

}

public void remove( )

{ MyArrayList.this.remove( --current ); }

}

}

**迭代器、Java嵌套类和内部类**

ArrayListIterator使用一个复杂Java结构，叫作内部类。

当我们让ArrayListIterator为一个嵌套类时，该类将被放入另一个类(MyArrayList)的内部，这个类叫做外部类。我们必须用static来表示它是嵌套的；若无static，将得到一个内部类。嵌套类可以被设计成private,这很好，此时该嵌套类除能够被外部类MyArrayList访问外，其他是不可访问的。

public class MyArrayList<AnyType> implements Iterable<AnyType>

{

private int theSize;

private AnyType [ ] theItems;

···

public java.util.Iterator<AnyType> iterator( )

{ return new ArrayListIterator<AnyType>( this ); }

private static class ArrayListIterator<AnyType> implements java.util.Iterator<AnyType>

{

private int current = 0;

private MyArrayList<AnyType> theList;

···

public ArrayListIterator( MyArrayList<AnyType> list )

{ theList = list; }

public boolean hasNext( )

{ return current < theList.size( ); }

public AnyType next( )

{ return theList.theItems[ current++ ]; }

}

}

当声明一个内部类时，编译器则添加对外部类对象的一个隐式引用，该对象引起内部类对象的构造。如果外部类的名字时Outer，则隐式引用就是Outer.this。theList.theItems的使用可以由MyArray-List.this.theItems代替。正如this.data可以简写为data一样，MyArrayList.this.theItems可以简写为theItems。

public class MyArrayList<AnyType> implements Iterable<AnyType>

{

private int theSize;

priavte AnyType [ ] theItems;

···

public class ArrayListIterator implements java.util.Iterator<AnyType>

{

private int current = 0;

public boolean hasNext( )

{ return current < size( ); }

public AnyType next( )

{ return theItems[ current++ ]; }

public void remove( )

{ MyArrayList.this.remove( --current ); }

}

}

**LinkedList类的实现**

在考虑设计方面，我们将需提供三个类：

1.MyLinkedList类本身，它包含到两端的链、表的大小以及一些方法。

2.Node类，它可能是一个私有的嵌套类。一个节点包含数据以及到前一个节点的链和到下一个节点的链，还有一些适当的构造方法。

3.LinkedListIeraor类，该类抽象了位置的概念，是一个私有类，并实现接口Iterator。它提供了方法next、hasNext和remove的实现。

我们能够在表的前端创建一个额外的节点，逻辑上代表开始的标记。这些额外的节点有时候就叫作标记节点；在前端的叫作头节点，在末端的叫作尾节点。

public class MyLinkedLis<AnyType> implements Iterable<AnyType>

{

private static class Node<AnyType>

{

public Node( AnyType d, Node<AnyType> p, Node<AnyType> n )

{ data = d; prev = p; next = n; }

public AnyType data;

public Node<AnyType> prev;

public Node<AnyType> next;

}

/\*\*

\* 初始化MyLinkedList方法，调用doClear方法

\*/

public MyLinkedList( )

{ doClear( ); }

/\*\*

\* 清空方法，调用doClear方法

\*/

public void clear( )

{ doClear( ); }

/\*\*

\* doClear方法，让头节点为空；尾节点前节点为头节点；头节点后节点为尾节点；长度为空；节点数为1

\*/

private void doClear( )

{

beginMarker = new Node<AnyType>( null, null, null);

endMarker = new Node<AnyType>(null, beginMarker, null);

beginMarker.next = endMarker;

theSize = 0;

modCount++;

}

/\*\*

\* 获取列表长度方法

\*/

public int size( )

{ return theSize;}

/\*\*

\* 检查列表是否为空方法

\*/

public boolean isEmpty( )

{ return size( ) == 0;}

/\*\*

\* 在列表末端加入元素方法

\*/

public boolean add( AnyType x )

{ add( size( ), x ); return true; }

/\*\*

\* 在列表任意位置加入元素方法

\*/

public void add( int idx, AnyType x )

{ addBefore( getNode( idx, 0, size( ) ), x ); }

/\*\*

\* 按下标获取元素值方法

\*/

public AnyType get( int idx )

{ return getNode( idx ).data; }

/\*\*

\* 改变指定位置元素值方法并返回旧的元素值

\*/

public AnyType set( int idx, AnyType newVal )

{

Node<AnyType> p = getNode( idx );

AnyType oldVal = p.data;

p.data = newVal;

return oldVal;

}

/\*\*

\* 删除指定位置元素方法

\*/

public AnyType remove( int idx )

{ return remove( getNode( idx ) ); }

/\*\*

\* 在列表任意位置加入元素方法具体实现

\*/

private void addBefore( Node<AnyType> p, AnyType x )

{

Node<AnyType> newNode = new Node<>( x, p.prev, p );//新建节点，新节点data值为参数，新节点下节点指向原节点，新节点前节点为原节点前节点

newNode.prev.next = newNode;//前节点下节点指向新节点

p.prev = newNode;//原节点前节点指向新节点

theSize++;//列表长度＋1

modCount++;//节点数+1

}

/\*\*

\* 移除元素方法具体实现

\*/

private AnyType remove( Node<AnyType> p )

{

p.next.prev = p.prev;//删除节点的下节点的前节点指向删除节点的前节点

p.prev.next = p.next;//删除节点的前节点的下节点指向删除节点的下节点

theSize--;//列表长度-1

modCount++;//节点数+1

return p.data;//返回删除节点的值

}

/\*\*

\* 获取节点方法

\*/

private Node<AnyType> getNode( int idx )

{

return getNode( idx, 0, size( ) - 1 );

}

/\*\*

\* 获取节点方法具体实现

\*/

private Node<AnyType> getNode( int idx, int lower, int upper )

{

Node<AnyType> p;

if( idx < lower || idx > upper )//判断是否下标超出

throw new IndexOutOfBoundsException( );

/\*

\* 如果下标未过半则从头开始遍历查找，否则从末尾开始遍历查找

\*/

if( idx < size( ) / 2 )

{

p = beginMarker.next;

for( int i = 0; i < idx; i++ )

p = p.next;

}

else

{

p = endMarker;

for( int i = size( ); i > idx; i-- )

p = p.prev;

}

return p;

}

public java.util.Iterator<AnyType> iterator( )

{ return new LinkedListIterator( ); }

/\*\*

\* 迭代器方法

\*/

private class LinkedListIterator implements java.util.Iterator<AnyType>

{

private Node<AnyType> current = beginMarker.next;

private int expectedModCount = modCount;

private boolean okToRemove = false;

public boolean hasNext( )

{ return current != endMarker; }

public AnyType next( )

{

if( modCount != expectedModCount )

throw new java.util.ConcurrentModificationException( );

if( !hasNext( ) )

throw new java.util.NoSuchElementException( );

AnyType nextItem = current.data;

current = current.next;

okToRemove = true;

return nextItem;

}

public void remove( )

{

if( modCount != expectedModCount )

throw new java.util.ConcurrentModificationException( );

if( !okToRemove )

throw new IllegalStateException( );

MyLInkedList.this.remove( current.prev );

expectedModCount++;

okToRemove = false;

}

}

private int theSize;

private int modCount = 0;

private Node<AnyType> beginMarker;

private Node<AnyType> endMarker;

}