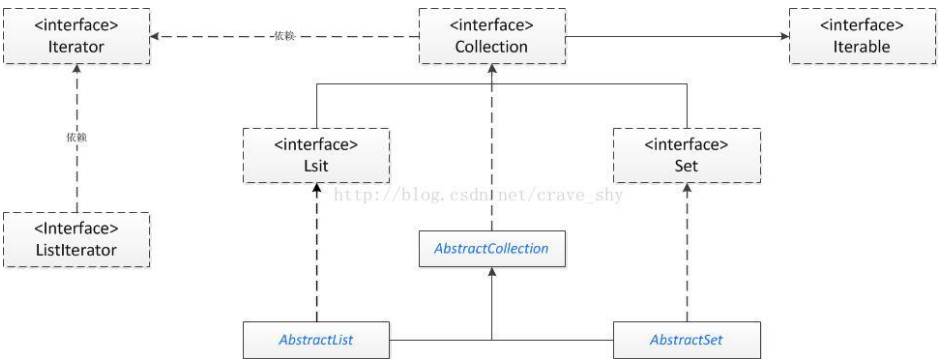


# java\_集合体系之Collection框架相关抽象类接口详解、源码——02

原创 2013年12月20日 09:14:08  
3245 0 12 编辑 (http://write.blog.csdn.net/postedit/{fileName}) 删除  
分享

## java\_集合体系之Collection框架相关抽象类接口详解、源码——02

### 一：Collection相关接口、抽象类结构图



### 二：Collection相关

#### 1、接口简介：

Collection作为队列形式存储的集合的抽象接口 Collection 是一个被高度抽象出来的接口、提供基本的操作数据的行为、属性的定义、要求子类必须提供两个构造方法、一个无参构造方法、一个以另一个Collection为基础的构造方法。

#### 2、源码：

虽然Collection是一个接口、仅有方法的声明没有方法体、但是个人觉得知道源码中对各个方法的目的或者意义的说明对后面的理解还是很有必要的、同时在这里也不再将Collection的API——列出来、会在后面的接口抽象类中标识列出、源码是一些方法的定义、



Oscar Chen (http://blog....)

+ 关注

(http://blog.csdn.net/chenghuaying)

原创 粉丝 喜欢

182 14 0

- > CentOS 集群机器之间ssh免密 (/crave\_shy/article/details/72964997)
- > JVM-内存管理-运行时数据区域 (/crave\_shy/article/details/56675052)
- > JVM-Blog目录 (/crave\_shy/article/details/56675032)
- > JVM-为什么要学JVM (/crave\_shy/article/details/56673439)

更多文章  
(http://blog.csdn.net/chenghuaying)

#### 在线课程



(http://edu.csdn.net/huiyiCourse/series\_detail?utm\_source=blog7)

【直播】机器学习&数据挖掘7周实训--韦玮

(http://edu.csdn.net/huiyiCourse/series\_detail/54?utm\_source=blog7)



(http://edu.csdn.net/combo/detail/471?utm\_source=blog7)

【套餐】系统集成项目管理工程师顺利通关--徐朋

(http://edu.csdn.net/combo/detail/471?utm\_source=blog7)

```

package com.chy.collection.core;

import java.util.Iterator;
/**
 * Collection 是一个被高度抽象出来的接口、是一个独立元素的序列、这些独立元素服从一条或者多条规则
 * 本身提供一类集合所具有的共有的方法、所有作为其子类的集合都不能直接实现他、而是应该实现他的子接口、
 * 并且必须要提供两个构造方法、一个无参构造方法、一个以一个集合为基础的有参构造方法。具体有什么规则
 * 在后面的具体的类的时候会有说明。
 *
 * @author andyChen
 */
public interface Collection<E> extends Iterable<E> {

    /** 返回当前集合的大小、如果超过int的范围、则返回int的最大值*/
    int size();

    /** 标识当前Collection是否有元素、是返回true、否返回false*/
    boolean isEmpty();

    /** 当前Collection是否包含o、*/
    boolean contains(Object o);

    /** 返回当前Collection的包含所有元素的一个迭代器 */
    Iterator<E> iterator();

    /** 将当前Collection以一个Object数组的形式返回 */
    Object[] toArray();

    /** 将当前Collection中所有元素以一个与T相同类型的数组的形式返回*/
    <T> T[] toArray(T[] a);

    // Modification Operations

    /** 确保此 collection 包含指定的元素*/
    boolean add(E e);

    /** 从此 collection 中移除指定元素的单个实例，如果存在的话*/
    boolean remove(Object o);

    // Bulk Operations

    /** 如果此 collection 包含指定 collection 中的所有元素，则返回 true*/
    boolean containsAll(Collection<?> c);

    /** 将指定 collection 中的所有元素都添加到此 collection 中*/
    boolean addAll(Collection<? extends E> c);

    /** 移除此 collection 中那些也包含在指定 collection 中的所有元素（可选操作）*/
    boolean removeAll(Collection<?> c);

    /** 仅保留此 collection 中那些也包含在指定 collection 的元素、即求两个Collection的交集 */
    boolean retainAll(Collection<?> c);

    /** 移除此 collection 中的所有元素*/
    void clear();

    // Comparison and hashing

    /** 比较此 collection 与指定对象是否相等*/
    boolean equals(Object o);

    /** 返回此 collection 的哈希码值*/
    int hashCode();
}

```

### 三：AbstractCollection

#### 1、抽象类简介：

AbstractCollection是一个实现Collection接口的抽象类、本身没有提供任何额外的方法、所有想要实现Collection接口的实现类都必须从AbstractCollection继承、他存在的意义就是大大减少编码的工作量、AbstractCollection中的方法都是从Collection接口中继承、除两个关键方法 abstract Iterator<E> iterator();abstract int size();方法外都提供了简单的实现（这里要注意add()

只是抛出异常、要求子类必须去实现、因为不同数据结构添加元素的方式不一致 )、其他的方法都是通过Iterator的方法来实现的、所以在子类中要实现Iterator的方法、讲到Iterator才具体列举有哪些方法、这就是此类为什么能减少代码量的原因。

## 2、源码：

API与Collection的完全相同、只是有简单实现、具体如何实现可见源码

```

package com.chy.collection.core;

import java.util.Arrays;
import java.util.Iterator;

/**
 * 此类提供 Collection 接口的骨干实现，以最大限度地减少了实现此接口所需的工作。
 *
 * 子类要继承此抽象类、
 * 1、必须提供两个构造方法、一个无参一个有参。
 * 2、必须重写抽象方法 Iterator<E> iterator();方法体中必须有hasNext()和next()两个方法、同时必须实现
remove方法 用于操作集合中元素。
 * 3、一般要重写add方法、否则子类调用add方法会抛UnsupportedOperationException异常。
 * 子类通常要重写此类中的方法已得到更有效的实现、
 */

@SuppressWarnings("unchecked")
public abstract class AbstractCollection<E> implements Collection<E> {
    /**
     * 基础构造器
     */
    protected AbstractCollection() {
    }

    // Query Operations

    public abstract Iterator<E> iterator();

    public abstract int size();

    public boolean isEmpty() {
        return size() == 0;
    }

    public boolean contains(Object o) {
        Iterator<E> e = iterator();
        if (o==null) {
            while (e.hasNext())
                if (e.next()==null)
                    return true;
        } else {
            while (e.hasNext())
                if (o.equals(e.next()))
                    return true;
        }
        return false;
    }

    public Object[] toArray() {
        // Estimate size of array; be prepared to see more or fewer elements
        Object[] r = new Object[size()];
        Iterator<E> it = iterator();
        for (int i = 0; i < r.length; i++) {
            if (! it.hasNext()) // fewer elements than expected
                return Arrays.copyOf(r, i);
            r[i] = it.next();
        }
        return it.hasNext() ? finishToArray(r, it) : r;
    }

    public <T> T[] toArray(T[] a) {
        // Estimate size of array; be prepared to see more or fewer elements
        int size = size();
        T[] r = a.length >= size ? a :
            (T[])java.lang.reflect.Array
                .newInstance(a.getClass().getComponentType(), size);
        Iterator<E> it = iterator();

        for (int i = 0; i < r.length; i++) {
            if (! it.hasNext()) { // fewer elements than expected
                if (a != r)
                    return Arrays.copyOf(r, i);
                r[i] = null; // null-terminate
                return r;
            }
            r[i] = (T)it.next();
        }
        return it.hasNext() ? finishToArray(r, it) : r;
    }
}

```

```

private static <T> T[] finishToArray(T[] r, Iterator<?> it) {
    int i = r.length;
    while (it.hasNext()) {
        int cap = r.length;
        if (i == cap) {
            int newCap = ((cap / 2) + 1) * 3;
            if (newCap <= cap) { // integer overflow
                if (cap == Integer.MAX_VALUE)
                    throw new OutOfMemoryError(
                        "Required array size too large");
                newCap = Integer.MAX_VALUE;
            }
            r = Arrays.copyOf(r, newCap);
        }
        r[i++] = (T)it.next();
    }
    // trim if overallocated
    return (i == r.length) ? r : Arrays.copyOf(r, i);
}

// Modification Operations

public boolean add(E e) {
    throw new UnsupportedOperationException();
}

public boolean remove(Object o) {
    Iterator<E> e = iterator();
    if (o==null) {
        while (e.hasNext()) {
            if (e.next()==null) {
                e.remove();
                return true;
            }
        }
    } else {
        while (e.hasNext()) {
            if (o.equals(e.next())) {
                e.remove();
                return true;
            }
        }
    }
    return false;
}

// Bulk Operations

public boolean containsAll(Collection<?> c) {
    Iterator<?> e = c.iterator();
    while (e.hasNext())
        if (!contains(e.next()))
            return false;
    return true;
}

public boolean addAll(Collection<? extends E> c) {
    boolean modified = false;
    Iterator<? extends E> e = c.iterator();
    while (e.hasNext()) {
        if (add(e.next()))
            modified = true;
    }
    return modified;
}

public boolean removeAll(Collection<?> c) {
    boolean modified = false;
    Iterator<?> e = iterator();
    while (e.hasNext()) {
        if (c.contains(e.next())) {
            e.remove();
            modified = true;
        }
    }
    return modified;
}

public boolean retainAll(Collection<?> c) {

```

```

        boolean modified = false;
        Iterator<E> e = iterator();
        while (e.hasNext()) {
            if (!c.contains(e.next())) {
                e.remove();
                modified = true;
            }
        }
        return modified;
    }

    public void clear() {
        Iterator<E> e = iterator();
        while (e.hasNext()) {
            e.next();
            e.remove();
        }
    }

    // String conversion

    public String toString() {
        Iterator<E> i = iterator();
        if (! i.hasNext())
            return "[]";

        StringBuilder sb = new StringBuilder();
        sb.append('[');
        for (;;) {
            E e = i.next();
            sb.append(e == this ? "(this Collection)" : e);
            if (! i.hasNext())
                return sb.append(']').toString();
            sb.append(", ");
        }
    }
}

```

## 四：List

### 1、接口简介：

作为Collection的一个子接口、也是集合的一种、List是有序的队列、其中存放的每个元素都有索引、第一个元素的索引从0开始、与Set相比、List允许存放重复的元素。因其具有索引这一概念、所以相比于Collection接口、List多了根据索引操作元素的方法、具体可为“增删改查”、从前向后查找某元素的索引值、从后向前查找某元素的索引值、List特有的ListIterator和在指定的索引处添加一个Collection集合。

官方版：A List is a collection which maintains an ordering for its elements. Every element in the List has an index. Each element can thus be accessed by its index, with the first index being zero. Normally, Lists allow duplicate elements, as compared to Sets, where elements have to be unique.

### 2、相对于Collection多提供的方法的源码

```

public interface List<E> extends Collection<E> {

    // Bulk Modification Operations

    /**
     * 将指定 collection 中的所有元素都插入到列表中的指定位置
     */
    boolean addAll(int index, Collection<? extends E> c);

    // Positional Access Operations

    /** 返回列表中指定位置的元素。*/
    E get(int index);

    /** 用指定元素替换列表中指定位置的元素*/
    E set(int index, E element);

    /** 在列表的指定位置插入指定元素*/
    void add(int index, E element);

    /** 移除列表中指定位置的元素*/
    E remove(int index);

    // Search Operations

    /** 返回此列表中第一次出现的指定元素的索引；如果此列表不包含该元素，则返回 -1*/
    int indexOf(Object o);

    /** 返回此列表中最后一次出现的指定元素的索引；如果此列表不包含该元素，则返回 -1*/
    int lastIndexOf(Object o);

    // List Iterators

    /** 返回此列表元素的列表迭代器*/
    ListIterator<E> listIterator();

    /** 返回列表中元素的列表迭代器（按适当顺序），从列表的指定位置开始*/
    ListIterator<E> listIterator(int index);

    // View

    /** 返回列表中指定的 fromIndex（包括）和 toIndex（不包括）之间的部分视图*/
    List<E> subList(int fromIndex, int toIndex);
}

```

## 五：AbstractList

### 1、抽象类简介：

同AbstractCollection定义相同、每个需要继承List的方法都不能从List直接继承、而是要通过AbstractList来实现、因List实现了Collection接口、所以为进一步简化编码、使得AbstractList继承AbstractCollection、这样AbstractList只需提供AbstractCollection中必须实现的方法的实现（具体是Iterator、add、size必须实现的方法、和一些为具有自己特色而重写的方法）List相对于Collection多提供的方法的实现即可。对于AbstractList的源码中、需要理解获取Iterator、ListIterator、

### 2、源码

```

package com.chy.collection.core;

import java.util.AbstractList;
import java.util.AbstractSequentialList;
import java.util.ConcurrentModificationException;
import java.util.Iterator;
import java.util.ListIterator;
import java.util.NoSuchElementException;
import java.util.RandomAccess;
import java.util.AbstractList.Itr;

/**
 * 此类提供 List 接口的骨干实现，以最大限度地减少实现“随机访问”数据存储（如数组）支持的该接口所需的工作。对于连续的访问数据（如链表），应优先使用 AbstractSequentialList，而不是此类。
 *
 * 1、若子类要使用set(int index, E e)则必须自己重新实现
 * 2、若子类要使用add(int index, E e)则必须自己重新实现
 * 3、若子类要使用remove(int index)则必须自己重新实现
 * 4、内部实现了iterator()方法、返回一个包含hasNext()、next()、remove()方法的Iterator。
 * 许多方法都是由内部类来实现的
 */

public abstract class AbstractList<E> extends AbstractCollection<E> implements List<E> {

    protected AbstractList() {

    }

    /**
     * 将指定元素追加到List集合结尾处
     */
    public boolean add(E e) {
        add(size(), e);
        return true;
    }

    abstract public E get(int index);

    public E set(int index, E element) {
        throw new UnsupportedOperationException();
    }

    public void add(int index, E element) {
        throw new UnsupportedOperationException();
    }

    public E remove(int index) {
        throw new UnsupportedOperationException();
    }

    // Search Operations

    public int indexOf(Object o) {
        ListIterator<E> e = listIterator();
        if (o==null) {
            while (e.hasNext())
                if (e.next()==null)
                    return e.previousIndex();
        } else {
            while (e.hasNext())
                if (o.equals(e.next()))
                    return e.previousIndex();
        }
        return -1;
    }

    public int lastIndexOf(Object o) {
        ListIterator<E> e = listIterator(size());
        if (o==null) {
            while (e.hasPrevious())
                if (e.previous()==null)
                    return e.nextIndex();
        } else {
            while (e.hasPrevious())
                if (o.equals(e.previous()))
                    return e.nextIndex();
        }
        return -1;
    }
}

```



```

// Bulk Operations

public void clear() {
    removeRange(0, size());
}

public boolean addAll(int index, Collection<? extends E> c) {
    boolean modified = false;
    Iterator<? extends E> e = c.iterator();
    while (e.hasNext()) {
        add(index++, e.next());
        modified = true;
    }
    return modified;
}

// Iterators

public Iterator<E> iterator() {
    return new Itr();
}

public ListIterator<E> listIterator() {
    return listIterator(0);
}

public ListIterator<E> listIterator(final int index) {
    if (index<0 || index>size())
        throw new IndexOutOfBoundsException("Index: "+index);

    return new ListItr(index);
}

//fail-fast机制
private class Itr implements Iterator<E> {
    /**
     * 游标的位置、用于获取元素和标识是否遍历完毕
     */
    int cursor = 0;

    /**
     * 最后一次调用next或者previous返回的元素的下标、如果元素被删除则返回-1
     */
    int lastRet = -1;

    /**
     * 用于后面检测在执行Iterator过程中是否被动手脚的变量
     */
    int expectedModCount = modCount;

    //是否还有下一个、从判断标准中可以看出、下边每遍历出一个元素cursor都会++
    public boolean hasNext() {
        return cursor != size();
    }

    public E next() {
        //类私有方法、检测是否此类中元素被动过
        checkForComodification();
        try {
            //获取下一个元素、将返回的index元素的下标记录下来、同时cursor自增1
            E next = get(cursor);
            lastRet = cursor++;
            return next;
        } catch (IndexOutOfBoundsException e) {
            checkForComodification();
            throw new NoSuchElementException();
        }
    }

    //移除当前遍历到的元素
    public void remove() {
        if (lastRet == -1)
            throw new IllegalStateException();
        checkForComodification();
        try {
            //调用当前的remove删除最后被遍历到的元素、在next()方法中记录了此元素的idnex
            AbstractList.this.remove(lastRet);

```

```

        //将cursor--、继续遍历下一个
        if (lastRet < cursor)
            cursor--;
        lastRet = -1;
        expectedModCount = modCount;
    } catch (IndexOutOfBoundsException e) {
        throw new ConcurrentModificationException();
    }
}
/**
 * 检测是否触发fail-fast机制
 */
final void checkForComodification() {
    if (modCount != expectedModCount)
        throw new ConcurrentModificationException();
}
}

private class ListItr extends Itr implements ListIterator<E> {

    //构造方法、初始化第一个被读取的元素的下标
    ListItr(int index) {
        cursor = index;
    }

    //当前listIterator中cursor指向的元素是否还有上一个元素
    public boolean hasPrevious() {
        return cursor != 0;
    }

    //获取当前ListIterator中cursor指向的元素
    public E previous() {
        checkForComodification();
        try {
            //获取cursor上一个元素的
            int i = cursor - 1;
            E previous = get(i);
            lastRet = cursor = i;
            return previous;
        } catch (IndexOutOfBoundsException e) {
            checkForComodification();
            throw new NoSuchElementException();
        }
    }

    public int nextIndex() {
        return cursor;
    }

    public int previousIndex() {
        return cursor-1;
    }

    public void set(E e) {
        if (lastRet == -1)
            throw new IllegalStateException();
        checkForComodification();

        try {
            AbstractList.this.set(lastRet, e);
            expectedModCount = modCount;
        } catch (IndexOutOfBoundsException ex) {
            throw new ConcurrentModificationException();
        }
    }

    public void add(E e) {
        checkForComodification();

        try {
            AbstractList.this.add(cursor++, e);
            lastRet = -1;
            expectedModCount = modCount;
        } catch (IndexOutOfBoundsException ex) {
            throw new ConcurrentModificationException();
        }
    }
}

public List<E> subList(int fromIndex, int toIndex) {

```

```

        return (this instanceof RandomAccess ?
            new RandomAccessSubList<E>(this, fromIndex, toIndex) :
            new SubList<E>(this, fromIndex, toIndex));
    }

    // Comparison and hashing

    public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof List))
            return false;

        ListIterator<E> e1 = listIterator();
        ListIterator e2 = ((List) o).listIterator();
        while (e1.hasNext() && e2.hasNext()) {
            E o1 = e1.next();
            Object o2 = e2.next();
            if (!(o1==null ? o2==null : o1.equals(o2)))
                return false;
        }
        return !(e1.hasNext() || e2.hasNext());
    }

    public int hashCode() {
        int hashCode = 1;
        Iterator<E> i = iterator();
        while (i.hasNext()) {
            E obj = i.next();
            hashCode = 31*hashCode + (obj==null ? 0 : obj.hashCode());
        }
        return hashCode;
    }

    protected void removeRange(int fromIndex, int toIndex) {
        ListIterator<E> it = listIterator(fromIndex);
        for (int i=0, n=toIndex-fromIndex; i<n; i++) {
            it.next();
            it.remove();
        }
    }

    protected transient int modCount = 0;
}

class SubList<E> extends AbstractList<E> {
    private AbstractList<E> l;
    private int offset;
    private int size;
    private int expectedModCount;

    SubList(AbstractList<E> list, int fromIndex, int toIndex) {
        if (fromIndex < 0)
            throw new IndexOutOfBoundsException("fromIndex = " + fromIndex);
        if (toIndex > list.size())
            throw new IndexOutOfBoundsException("toIndex = " + toIndex);
        if (fromIndex > toIndex)
            throw new IllegalArgumentException("fromIndex(" + fromIndex +
                ") > toIndex(" + toIndex + ")");

        l = list;
        offset = fromIndex;
        size = toIndex - fromIndex;
        expectedModCount = l.modCount;
    }

    public E set(int index, E element) {
        rangeCheck(index);
        checkForComodification();
        return l.set(index+offset, element);
    }

    public E get(int index) {
        rangeCheck(index);
        checkForComodification();
        return l.get(index+offset);
    }

    public int size() {
        checkForComodification();
        return size;
    }

```

```

}

public void add(int index, E element) {
    if (index < 0 || index > size)
        throw new IndexOutOfBoundsException();
    checkForComodification();
    l.add(index+offset, element);
    expectedModCount = l.modCount;
    size++;
    modCount++;
}

public E remove(int index) {
    rangeCheck(index);
    checkForComodification();
    E result = l.remove(index+offset);
    expectedModCount = l.modCount;
    size--;
    modCount++;
    return result;
}

protected void removeRange(int fromIndex, int toIndex) {
    checkForComodification();
    l.removeRange(fromIndex+offset, toIndex+offset);
    expectedModCount = l.modCount;
    size -= (toIndex-fromIndex);
    modCount++;
}

public boolean addAll(Collection<? extends E> c) {
    return addAll(size, c);
}

public boolean addAll(int index, Collection<? extends E> c) {
    if (index < 0 || index > size)
        throw new IndexOutOfBoundsException(
            "Index: "+index+", Size: "+size);
    int cSize = c.size();
    if (cSize == 0)
        return false;

    checkForComodification();
    l.addAll(offset+index, c);
    expectedModCount = l.modCount;
    size += cSize;
    modCount++;
    return true;
}

public Iterator<E> iterator() {
    return listIterator();
}

public ListIterator<E> listIterator(final int index) {
    checkForComodification();
    if (index < 0 || index > size)
        throw new IndexOutOfBoundsException(
            "Index: "+index+", Size: "+size);

    return new ListIterator<E>() {
        private ListIterator<E> i = l.listIterator(index+offset);

        public boolean hasNext() {
            return nextIndex() < size;
        }

        public E next() {
            if (hasNext())
                return i.next();
            else
                throw new NoSuchElementException();
        }

        public boolean hasPrevious() {
            return previousIndex() >= 0;
        }

        public E previous() {
            if (hasPrevious())

```

```

        return i.previous();
    } else {
        throw new NoSuchElementException();
    }

    public int nextIndex() {
        return i.nextIndex() - offset;
    }

    public int previousIndex() {
        return i.previousIndex() - offset;
    }

    public void remove() {
        i.remove();
        expectedModCount = l.modCount;
        size--;
        modCount++;
    }

    public void set(E e) {
        i.set(e);
    }

    public void add(E e) {
        i.add(e);
        expectedModCount = l.modCount;
        size++;
        modCount++;
    }
};
}

public List<E> subList(int fromIndex, int toIndex) {
    return new SubList<E>(this, fromIndex, toIndex);
}

private void rangeCheck(int index) {
    if (index<0 || index>=size)
        throw new IndexOutOfBoundsException("Index: "+index+
            ",Size: "+size);
}

private void checkForComodification() {
    if (l.modCount != expectedModCount)
        throw new ConcurrentModificationException();
}
}

class RandomAccessSubList<E> extends SubList<E> implements RandomAccess {
    RandomAccessSubList(AbstractList<E> list, int fromIndex, int toIndex) {
        super(list, fromIndex, toIndex);
    }

    public List<E> subList(int fromIndex, int toIndex) {
        return new RandomAccessSubList<E>(this, fromIndex, toIndex);
    }
}
}

```

## 六：Iterable、Iterator、ListIterator

### 1、Iterable接口说明及源码

Iterable接口中只有一个方法就是返回一个有序元素序列的迭代器Iterator

```

public interface Iterable<T> {

    /** 返回一个元素序列的迭代器 */
    Iterator<T> iterator();
}

```

### 2、Iterator接口说明及源码

是集合依赖的接口、每个集合类或者其父类都必须实现获取Iterator的方法、同时要提供Iterator中方法的实现、用于遍历和作为一些方法的实现。

```
public interface Iterator<E> {

    boolean hasNext();

    E next();

    void remove();

}
```

### 3、ListIterator接口说明及源码

ListIterator继承Iterator、用于提供额外的几个方法、用于迭代List集合中的元素。

```
public interface ListIterator<E> extends Iterator<E> {
    // Query Operations

    /**
     * 以正向遍历列表时，如果列表迭代器有多个元素，则返回 true（换句话说，如果 next 返回一个元素而
     * 不是抛出异常，则返回 true）。
     */
    boolean hasNext();

    /**
     * 返回列表中的下一个元素。
     */
    E next();

    /**
     * 如果以逆向遍历列表，列表迭代器有多个元素，则返回 true。
     */
    boolean hasPrevious();

    /**
     * 返回列表中的前一个元素。
     */
    E previous();

    /**
     * 返回对 next 的后续调用所返回元素的索引。
     */
    int nextIndex();

    /**
     * 返回对 previous 的后续调用所返回元素的索引。
     */
    int previousIndex();

    // Modification Operations

    /**
     * 从列表中移除由 next 或 previous 返回的最后一个元素（可选操作）。
     */
    void remove();

    void set(E e);

    void add(E e);
}
```

## 六：Set

### 1、接口简介：

Set、作为Collection的一个子类、与List相比、其不允许有重复元素出现、存放的元素没有索引一说、其API与Collection完全相同、只是对方法的定义不同

### 2、源码不再乱贴

## 七：AbstractSet

### 1、抽象类简介：

同AbstractList很相似、因Set继承与Collection接口、其也继承AbstractCollection抽象类、由于Set的特性、他不像AbstractList一样新增了有关索引的操作方法、他没有添加任何方法、仅仅是实现了equals、hashCode、removeAll(Collection<?> c)方法的实现。**注意**：他不像AbstractList、内部提供了实现获取Iterator的方法、他要求继承他的子类去实现获取Iterator的方法。

## 2、源码

```
package com.chy.collection.core;

import java.util.Iterator;

/**
 * 此类提供 Set 接口的骨干实现，从而最大限度地减少了实现此接口所需的工作。
 * 此类并没有重写 AbstractCollection 类中的任何实现。它仅仅添加了 equals 和 hashCode 的实现。
 */
@SuppressWarnings("unchecked")
public abstract class AbstractSet<E> extends AbstractCollection<E> implements Set<E> {
    protected AbstractSet() {
    }

    // Comparison and hashing

    public boolean equals(Object o) {
        if (o == this)
            return true;

        if (!(o instanceof Set))
            return false;
        Collection c = (Collection) o;
        if (c.size() != size())
            return false;
        try {
            return containsAll(c);
        } catch (ClassCastException unused) {
            return false;
        } catch (NullPointerException unused) {
            return false;
        }
    }

    public int hashCode() {
        int h = 0;
        Iterator<E> i = iterator();
        while (i.hasNext()) {
            E obj = i.next();
            if (obj != null)
                h += obj.hashCode();
        }
        return h;
    }

    public boolean removeAll(Collection<?> c) {
        boolean modified = false;

        if (size() > c.size()) {
            for (Iterator<?> i = c.iterator(); i.hasNext(); )
                modified |= remove(i.next());
        } else {
            for (Iterator<?> i = iterator(); i.hasNext(); ) {
                if (c.contains(i.next())) {
                    i.remove();
                    modified = true;
                }
            }
        }
        return modified;
    }
}
```


更多内容：[java\\_集合体系之总体目录——00](http://blog.csdn.net/crave_shy/article/details/1741679)  
([http://blog.csdn.net/crave\\_shy/article/details/1741679](http://blog.csdn.net/crave_shy/article/details/1741679))

版权声明：本文为博主原创文章，未经博主允许不得转载。



标签：Collection (<http://so.csdn.net/so/search/s.do?q=Collection&t=blog>) /  
java (<http://so.csdn.net/so/search/s.do?q=java&t=blog>) /  
List (<http://so.csdn.net/so/search/s.do?q=List&t=blog>) /  
set (<http://so.csdn.net/so/search/s.do?q=set&t=blog>) /  
java集合框架图 (<http://so.csdn.net/so/search/s.do?q=java集合框架图&t=blog>) /

0条评论

 qq\_36596145 ([http://my.csdn.net/qq\\_36596145](http://my.csdn.net/qq_36596145))




发表评论

暂无评论

## 相关文章推荐


### Node初学者入门，一本全面的NodeJS教程 (/qi1271199790/article/details/61441338)

关于 本书致力于教会你如何用Node.js来开发应用，过程中会传授你所有所需的“高级”JavaScript知识。本书绝不是一本“Hello World”的教程。 状态 你正在...

 qi1271199790 2017-03-11 23:17 👁 117

### elasticsearch 分布式环境搭建 (/qq\_20679251/article/details/76038884)

elasticsearch 分布式环境搭建es下载及目录介绍es官网地址：<https://www.elastic.co/downloads/elasticsearch>  
本集群：192.168.190....

 qq\_20679251 2017-07-24 22:17 👁 65


### [并行计算] 1. 并行计算简介 (/magicbean2/article/details/75174859)

这篇帖子旨在为并行计算这一广泛而宏大的话题提供一个非常快速的概述，作为随后教程的先导。因此，它只涵盖了并行计算的基础知识，实用于刚刚开始熟悉该主题的初学者。

 magicbean2 2017-07-20 15:30 👁 2177

### JAVA中的进程和线程 (/qq\_36074113/article/details/74857209)

一：进程与线程 概述：几乎任何的操作系统都支持运行多个任务，通常一个任务就是一个程序，而一个程序就是一个进程。当一个进程运行时，内部可能包括多个顺序执行流，每个顺序执行流就是一个线程。 进程：进程...

 qq\_36074113 2017-07-09 09:13 👁 73

### C++虚函数表深入解析 (一) (/jxz\_dz/article/details/47998869)

本篇文章大概从三个角度解析虚函数表：A：虚函数调用方式 B：深入解析虚函数 C：打印虚函数表 有问题一起交流  
! A：虚函数调用方式 关于函数调用方式,在此指的是直接调用与...





jxz\_dz 2015-08-26 09:48 395

### Java中的最常犯的错误Top10 (/qq\_27035123/article/details/75330836)

1. 数组转ArrayList为了实现在把一个数组转换成一个ArrayList，很多java程序员会使用如下的代码：List list = Arrays.asList(arr); Arrays.asLi...



qq\_27035123 2017-07-18 21:19 1554

### 收录各种猥琐的Java笔试/面试题 (/smcwwh/article/details/7315041)

本文收录各种猥琐的Java笔试/面试题，一些比较容易忘记的，不定期更新。也希望大家在底下留言，贴出自己碰到或看到的各种猥琐笔试、面试题。J2EE基础部分 1、运算符优先级问题，下面代码的结果是...



SMCwwh 2012-03-03 16:24 21425

### Elasticsearch 5.1.1搜索高亮及Java API实现 (/napoay/article/details/53910646)

5.1.1的搜索高亮和2.X有所变化，但是变化不大。下面分四步来介绍:创建索引(设置mapping/IK分词)、索引文档、REST API的搜索高亮、JAVA API的搜索高亮。注:从这篇博客开始...



napoay 2016-12-28 17:39 8071

### java菜鸟第十四课重构、js与接口的实现 (/qq\_28633249/article/details/75465656)

一、接口 接口的方法不一定必须实现的!!! 加入default，这样的方法可以不实现，如图所示 二、关于语言的特性 1. C++多继承 2. 很多编程语言JavaScript、Python支持混入...



qq\_28633249 2017-07-20 10:10 82

### 超详细的java基础知识学习 (java SE、javaEE) 笔记 核心重点! (/qq\_34477549/article/details/52803821)

标识符 Java 的标识符是由字母、数字、下划线\_、以及美元符\$组成，但是首字母不可以是数字。Java 标识符大小写敏感，长度无限制，不能是 Java 中的关键字。命名规则：要见名知意！u 变...



qq\_34477549 2016-10-13 09:12 1599

### java\_集合体系之总体目录——00 (http://810364804.iteye.com/blog/1992787)

java\_集合体系之总体目录——00 java\_集合体系之总体框架——01 <a target="\_blank" href="http://blog.csdn.net/crave\_shy



810364804 2013-12-19 15:41 66

### Java线程池 (/nanmuling/article/details/37881089)

Java线程池 线程池编程 java.util.concurrent多线程框架---线程池编程 (一) 一般的服务器都需要线程池，比如Web、FTP等服务器，不过它们一般都已经实现了线程池，比如以...





nanmuling 2014-07-16 16:44 2850

### 常见异常解析 (http://fuyou0104.iteye.com/blog/1174579)

ConcurrentHashMap与CopyOnWriteArrayList比较。博客分类：Java ConcurrentHashMap ConcurrentHashMap引入了Segment，每个Segment又是一个hashtable，相当于是两级Hash表，然后锁是在

Segment一级进行的，提高了并发性。缺点是对整个集合进行操作的方法如 size() 或 isEmpty()的实现很困难，基本无法得到精准的数据。Segment的read不加锁，只有在读到null的情况(一般不会有null的，只有在其他线程操作Map的时候，所以就用锁来等他操作完)下调用

 fuyou0104 2011-09-18 16:29  4162

---

## java\_集合体系之Collection框架相关抽象类接口详解、源码——02 (<http://810364804.iteye.com/blog/1992790>)



java\_集合体

 810364804 2013-12-20 09:14  70

---

## java\_集合体系之总体目录——00 ([/crave\\_shy/article/details/17416791](http://crave_shy/article/details/17416791))



摘要：java集合系列目录、不断更新中、、、水平有限、总有不足、误解的地方、请多包涵、也希望多提意见、多多讨论 ^\_^

 chenghuaying 2013-12-19 15:41  3210

---

## 集合框架源码分析二（抽象类篇） (<http://zhouyunan2010.iteye.com/blog/1164985>)



一。AbstractCollection [code="java"] public abstract class AbstractCollection implements Collection { /\*\* \* 唯一构造方法 \*/ protected AbstractCollection() {} // Query Operations /\*\* \* 返回集合中元素的迭代器 \*/ public abstract Iterator iterat

 zhouYunan2010 2011-09-03 17:11  1087

---

## Java基础——抽象类和接口 ([/mocarcher/article/details/76943945](http://mocarcher/article/details/76943945))

第六章 抽象类和接口第一节 抽象类一.包含抽象方法的类；抽象方法：只有方法声明，没有方法实现的方法称为“抽象方法”；抽象类是对问题领域进行分析后得出的抽象概念。抽象类和抽象方法必须使用...



 Mocarcher 2017-08-08 20:55  68

---

## Java——抽象类实现接口 (<http://fishswing.iteye.com/blog/1527166>)

在Java中，使用抽象类来实现接口，并不是毫无作用。相反，有时间有很大的作用。当你只想实现接口中的个别方法（不是所有方法）时，你可以先写一个抽象类来实现该接口，并实现除了你想要的方法之外的所有方法（方法体为空）。接着再用你的类继承这个抽象类，这个类中就只用实现你需要的方法了，这样就可以达到你的需要了。但是，如果你直接实现接口的话，你就需要实现接口的所有方法。通过下面例子，可以很好的理解：

例：有一个接口Window，有三个方法，draw(),putColor(),setPosition()

 fishswing 2012-05-14 11:32  21350

---

## 面试android开发工程师小结 ([/vae260772/article/details/72740048](http://vae260772/article/details/72740048))



5-25 今天面试了家公司，2个大神，差不多进行了1.5小时。不过最终还是没有录取，可能我的项目经验缺乏，能力不达标。今天问的问题大致如下，凭记忆写的，反正就那样吧：1、android se...

 vae260772 2017-05-25 19:01  271

---

## Java编程那些事儿68——抽象类和接口(一) (<http://xwk.iteye.com/blog/2134103>)

8.9 抽象类和接口 <span lang="EN-UK">

 阿尔萨斯 2008-12-28 13:49  29

