Java Collections Framework 的 Fail Fast 机制及代码导读

江苏 无锡 缪小东

本文章主要抽取了 Java Collections Framework 中的 Collection 接口、List 接口、AbstractCollection 抽象类、AbstractList 抽象类和具体的 ArrayList 的实现纵向研究了 Java Collections Framework 中的 Fail Fast 机制，通常的编程错误以及这些接口和类之间的关系，以有助于大家对 Java Collections Framework 源代码的研究。

“Fail Fast”机制－－列表的结构在其返回遍历器（Iterator）后的任何时候发生变化（这种变化不包括遍历器本身调用 remove 方法移除元素）后，将会导致遍历器抛出异常的机制。

一、Collection 接口

Collection 接口是 Java Collections Framework 中两个顶层接口（Collection 和 Map）中的一个。它表示一个集合的概念。

```
package java.util;
public interface Collection<E> extends Iterable<E> {
    // 以下是几个查询的方法
    int size();                         //查询有几个元素
    boolean isEmpty();                  //查询聚集是否为空
    boolean contains(Object o);         //查询聚集是否包含某个元素
    Iterator<E> iterator();             //返回聚集的遍历器，遍历器模式在 Java 中的典型应用

    Object[] toArray();                 //将聚集中的元素转换为对象数组
    <T> T[] toArray(T[] a);             //将聚集中的元素转换为指定类型的对象数组
                        //以上两个方法是 Collection 向数组转换的桥梁之一
                        //（另外一个，就是 Arrays.asList，将数组转换为 Collection 的子接口 List）
                        //它们构成 Collection 和 Array 转换的桥梁
    // 修改操作
    boolean add(E e);                   //增加指定的元素
    boolean remove(Object o);           //删除指定的元素
    // 批量操作
    boolean containsAll(Collection<?> c);        //查询是否包含某个聚集
    boolean addAll(Collection<? extends E> c); //将某个聚集假如该聚集
    boolean removeAll(Collection<?> c);         //从本聚集中删除指定聚集中的元素
    boolean retainAll(Collection<?> c);         //在本聚集中保存指定聚集中的元素
    void clear();                               //清除本聚集中所有的元素
    // 父类的方法
    boolean equals(Object o);                   //聚集自身的比较方法
    int hashCode();                             //聚集自身的 hash 值
}
```

二、List 接口

List 接口是 Collection 接口的三个子接口（List、Set、Queue）之一。它是各种具体列表（ArrayList、LinkedList）的公共接口。它们共同的特点是可以通过索引访问聚集中的对象。

```
package java.util;
public interface List<E> extends Collection<E> {
    //蓝色的为 List 接口中的新方法，其它的为 Collection 接口中的方法。
    //  查询操作
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator<E> iterator();
    Object[] toArray();
    <T> T[] toArray(T[] a);
    //  修改操作
    boolean add(E e);
    boolean remove(Object o);
    //  批处理操作
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean addAll(int index, Collection<? extends E> c);          //将聚集插入指定索引
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();
    //  比较和 hsah 操作
    boolean equals(Object o);
    int hashCode();
    //  位置相关的操作，只是 List 接口特有的
    E get(int index);                                              //得到指定索引处的元素
    E set(int index, E element);                                   //设置指定索引处的元素
    void add(int index, E element);                                //在指定索引处插入元素
    E remove(int index);                                           //删除指定索引处的元素
    //查找操作
    int indexOf(Object o);                                         //返回指定对象的索引
    int lastIndexOf(Object o);                                     //返回指定对象的索引，从后向前找
    //  遍历器方法
    ListIterator<E> listIterator();                                //返回列表遍历器，提供双向遍列
    ListIterator<E> listIterator(int index);                       //返回从某个位置开始的遍历器
    //  视图操作
    List<E> subList(int fromIndex, int toIndex);                   //返回该 List 的子视图
}
```

三、Iterator 接口

Iterator 是 Java Collection Framework 中一个比较有用的接口，它的作用是完成集合内部对象的遍列，是一个典型的遍历器模式的应用。

下面是 JDK 中 Iterator 类的源代码：

```
package java.util;
public interface Iterator<E> {
    boolean hasNext();      //查询聚集中是否还有元素
    E next();               //返回聚集中的下一个元素
    void remove();
    //从聚集中删除当前元素。该方法在每次调用 next 方法后可以被使用。
    //在调用该方法时，若其它线程改变聚集中的元素会抛出 ConcurrentModificationException 异常。
    //聚集不支持此方法，会抛出 UnsupportedOperationException 异常。
    //next 方法没有调用前，就调用此方法会抛出 IllegalStateException 异常。
}
```

Iterator 接口是 Java 集合框架中新的接口，Enumeration 是老的遍列接口。新接口支持 remove 方法，同时新接口的方法名称也比老接口短了。不过 Collections 类提供了它们之间转换的方法，阎宏的《Java 与模式》中使用了适配器模式（Adapter）完成两者的转换。

四、AbstractCollection 抽象类

AbstractCollection 抽象类是 Collection 接口的抽象实现，它利用 Collection 中的方法，完成其中的某些方法。

```
package java.util;
public abstract class AbstractCollection<E> implements Collection<E> {
    protected AbstractCollection() {    }          //默认的构造器，抽象类是有构造器的哦

    public abstract Iterator<E> iterator();
    public abstract int size();

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


    public boolean contains(Object o) {            //查询聚集中是否包含某元素
        Iterator<E> e = iterator();                //利用接口的 iterator 接口，遍历比较
        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() {                          //将聚集中的元素，转换为数组
        Object[] result = new Object[size()];        //主要利用接口中的 size、iterator 方法
        Iterator<E> e = iterator();
        for (int i=0; e.hasNext(); i++)
                result[i] = e.next();
        return result;
}

public <T> T[] toArray(T[] a) {                      //将聚集中的元素，转换为指定类型的数组
        int size = size();
        if (a.length < size)        //当指定数组长度小于聚集中元素的数目时，创建指定类型的数组
                a = (T[])java.lang.reflect.Array.newInstance(a.getClass().getComponentType(), size);
        //利用 java 的反射机制，根据输入数组的类型，创建和聚集相同容量的指定类型的数组
        //然后将聚集中的内容，利用 iterator 接口将其放入数组中
        Iterator<E> it=iterator();
        Object[] result = a;
        for (int i=0; i<size; i++)
                result[i] = it.next();
        if (a.length > size)        a[size] = null;       //这是干什么的啊？仔细想想吧！有什么问题吗
        return a;
}
//以上方法中返回数组的长度至少等于聚集类元素的个数
//当指定数组的长度小于聚集内元素的个数时，创建与其元素个数相同的指定类型的数组
//放入元素，并返回之；当指定数组的长度大于聚集中元素的数目时，作者让第 n+1 个元素为 null，
//是否合理呢!!
//若聚集内部有 n 个元素，指定数组长度大于 n，且指定数组在以前的操作中，全放满了元素，
//调用该方法后会是怎样呢？？首先是指定元素中的前 n 个元素被聚集中的 n 个元素覆盖，
//第 n+1 个元素，被设置为 null，即丢弃了，其它后面的保存原来的！怪不怪啊！
//一般使用此方法时，指定数组为只有一个元素的数组，该数组仅仅为了指定返回数组的类型


//  修改操作
public boolean add(E e) {                    //消极实现增加元素的方法
        throw new UnsupportedOperationException();
}

public boolean remove(Object o) {       //删除指定的元素
        Iterator<E> e = iterator();          //主要依靠 iterator 接口，完成遍列
        if (o==null) {
                while (e.hasNext()) {
                        if (e.next()==null) {       //聚集中可以包含 null 值哦
                                e.remove();
```

```
                    return true;
                }
            }
        } else {
            while (e.hasNext()) {
                if (o.equals(e.next())) {
                    e.remove();
                    return true;
                }
            }
        }
        return false;
}

// 批量处理方法
public boolean containsAll(Collection<?> c) {                    //本聚集是否包含指定聚集中所有的元素
        Iterator<?> e = c.iterator();                             //类似与集合论中的包含关系
        while (e.hasNext())
            if (!contains(e.next()))    return false;
        return true;
}

public boolean addAll(Collection<? extends E> c) {               //将指定聚集中的所有元素假如本聚集
        boolean modified = false;                                 //类似与集合论中的并集关系
        Iterator<? extends E> e = c.iterator();
        while (e.hasNext()) {                    //指定聚集中没有元素返回 false，有元素则假如返回 true
            if (add(e.next()))     modified = true;
        }
        return modified;
}

public boolean removeAll(Collection<?> c) {                      //从本聚集中删除指定聚集中的元素
        boolean modified = false;
        Iterator<?> e = iterator();
        while (e.hasNext()) {
            if (c.contains(e.next())) {
                e.remove();
                modified = true;
            }
        }
        return modified;
}
//该方法为：遍列本聚集中所有的元素，查询指定聚集 c 中是否包含本聚集中的某个元素
//c 中存在，则会调用本聚集遍历器的 remove 方法删除，否则保存该元素
//以上方法可能存在一定的效率问题。
//当本聚集中的元素比指定聚集中的元素少得多时很有利，反之则不然
```

```
    public boolean retainAll(Collection<?> c) {        //保存指定聚集中的所有本聚集中存在的元素
        boolean modified = false;                      //类似与集合论中的“交集”关系
        Iterator<E> e = iterator();
        while (e.hasNext()) {
            if (!c.contains(e.next())) {
                e.remove();
                modified = true;
            }
        }
        return modified;
    }
    //该方法为：遍列本聚集中所有的元素，查询指定聚集 c 中是否包含本聚集中的某个元素
    //c 中不存在，则会调用本聚集遍历器的 remove 方法删除，否则保存该元素
    //以上方法可能存在一定的效率问题。
    //当本聚集中的元素比指定聚集中的元素少得多时很有利，反之则不然

    public void clear() {                          //利用 Collection 接口的 iterator 方法，完成元素的清空
        Iterator<E> e = iterator();                //主要有两个步骤：
        while (e.hasNext()) {                      //①遍列
            e.next();
            e.remove();                            //②删除－－调用 iterator 接口的 remove 方法
        }
    }

    // 覆盖父类的 toString 方法，其中使用聚集内部对象的 toString 方法
    public String toString() {
        Iterator<E> i = iterator();
        if (! i.hasNext())      return "[]";       //没有元素返回"[]"，即括号中没有任何元素
        StringBuilder sb = new StringBuilder();
        sb.append('[');
        for (;;) {
            E e = i.next();
            sb.append(e == this ? "(this Collection)" : e);      //不能重复迭代自身，同时添加元素
            if (! i.hasNext())
                return sb.append(']').toString();                //继续迭代
            sb.append(", ");
        }
    }

}
```

五、AbstractList 抽象类

Java Collection Framework 中提供了很多以 Abstract 开头命名实现相应接口的抽象类，一般都是**利用**

接口中的 iterator 提供的方法，完成无需子类支持的功能。抽象类是有代码的，是实现某些功能的，不是所有的抽象类都使用空代码甚至抛出异常实现的。

（群里的朋友仔细读懂这一点：不是任何设计都需要一个接口和一个抽象类。甚至提供什么标识接口和抽象类的设计）

以下是 JDK 中 AbstractList 的源代码：

```
package java.util;
public abstract class AbstractList<E> extends AbstractCollection<E> implements List<E> {
    protected transient int modCount = 0;          //① AbstractList 中唯一的属性；
                                                   //并且它是一个供序列化使用的 transient 的关键词

    protected AbstractList() {    }       //默认构造器，被子类调用。在 Reflection 中很有用

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

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

    public boolean add(E e) {      //将指定的元素添加到聚集的最后面 ，
        add(size(), e);            //只要子类覆盖的 add(int index, E element)方法，该方法就可以了
        return true;               //
    }

    public int indexOf(Object o) {                    //查找某个对象，返回该对象的索引
        ListIterator<E> e = listIterator();           //利用接口的 listIterator 方法，子类只要实现
        if (o==null) {                                //listIterator 方法，就具有此查找功能
            while (e.hasNext())
                if (e.next()==null)      return e.previousIndex();
        } else {
            while (e.hasNext())
                if (o.equals(e.next()))      return e.previousIndex();
        }
        return -1;
    }

    //与上面的方法基本一致，只是反向遍历而已
    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;
    }
```

```
public void clear() {                          //清除聚集中所有的元素，使用 removeRange 方法
    removeRange(0, size());
}

protected void removeRange(int fromIndex, int toIndex) {  //从聚集中删除，某两个位置间的元素
    ListIterator<E> it = listIterator(fromIndex);         //利用下面的内部类 listIterator(fromIndex)
    for (int i=0, n=toIndex-fromIndex; i<n; i++) {        //在此可往复遍列上删除，指定个数的元素
        it.next();
        it.remove();
    }
}

//从指定索引位置开始，添加其它聚集所有的元素，该索引先后的元素会被依次移动，
//主要利用 Collection 的 iterator 方法和 AbstractList 类的 add add(int,E)方法。
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() {                //返回一个该聚集的遍历器对象
    return new Itr();                          //Itr 为内部类
}

private class Itr implements Iterator<E> {     //Itr 的实现
    int cursor = 0;          //next 方法将要返回的元素在聚集中的游标
    int lastRet = -1;              //①      最近调用 next 方法返回元素的索引，
                                   //-1 代表该元素被调用 remove 方法删除了
    int expectedModCount = modCount;       //②
    //很重要的变量 expectedModCount，
    //外界调用 AbstractList 任何具体子类的 iterator 方法时，该子类会调用父类的 iterator 方法
    //父类中创建此 Itr 内部类，
    //注意：创建时将此时父类中的 modCount 赋给，内部类的 expectedModCount 变量。
    //这样就形成以下关系：AbstractList 的具体子类在改变其内部元素时，modCount 会递增 1
    //调用 iterator 方法时，返回的 Itr 的实例，保存了此时 modCount 的值，
    //此后任何改变聚集中内部元素的方法都会增加 modCount，
    //这样就造成此时内部元素和调用 iterator 是的内部元素不一致（关键所在），
    //此时外部使用遍历器遍历时－－调用 next 方法返回后一个元素就会检查状态是否一致，
    //检查状态是否一致通过 checkForComodification()方法完成，
    //就是检查聚集的 modCount  和 Itr 创建时 expectedModCount 是否相同!!!!!
```

```
        public boolean hasNext() {              //判断游标是否指到聚集元素的末端
            return cursor != size();
        }

        public E next() {
            checkForComodification();           //③得到元素前先检查，聚集内部的状态是否改变
            try {
                E next = get(cursor);           //得到游标所指的元素
                lastRet = cursor++;             //②下一个返回值的索引是此时游标的后一个位置
                return next;                    //返回此值
            } catch (IndexOutOfBoundsException e) {
                checkForComodification();
                throw new NoSuchElementException();
            }
        }

        public void remove() {
            if (lastRet == -1)    throw new IllegalStateException();     //③
            checkForComodification();
            try {
                AbstractList.this.remove(lastRet);          //④删除游标所在位置的元素
                if (lastRet < cursor)       cursor--;       //
                lastRet = -1;                               //⑤下一次返回的索引-1，表示刚刚删除过
                expectedModCount = modCount;
            } catch (IndexOutOfBoundsException e) {
                throw new ConcurrentModificationException();
            }
        }

        final void checkForComodification() {           //③检查聚集的内部状态是否改变
            if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
        }
    }

    public ListIterator<E> listIterator() {         //方法 listIterator 返回实现 ListIterator 接口的内部类
        return listIterator(0);
    }

    public ListIterator<E> listIterator(final int index) {//从指定索引返回可往复的遍列
        if (index<0 || index>size())
          throw new IndexOutOfBoundsException("Index: "+index);
        return new ListItr(index);
    }

//-----------------------------------------------------------------------------------------------------------------------
//以下是一个可往复的遍列的实现，它是一个内部类。所谓可往复就是游标可以向前、向后移动罢了
```

```
//注意一下带颜色的部分就可以了。
    private class ListItr extends Itr implements ListIterator<E> {
        ListItr(int index) {
            cursor = index;
        }

        public boolean hasPrevious() {
            return cursor != 0;
        }

        public E previous() {
            checkForComodification();
            try {
                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();
                }
            }
        }
//-------------------------------------------------------------------------------------------------------------------------------

    //根据子类实现接口的不同，返回相应的两指定位置间所有元素，以 List 接口返回哦
    public List<E> subList(int fromIndex, int toIndex) {
        return (this instanceof RandomAccess ?
                new RandomAccessSubList<E>(this, fromIndex, toIndex) :
                new SubList<E>(this, fromIndex, toIndex));
    }

    //简单的覆盖 Object 的 equal 方法。
    //当两个 AbstractList 子类中，相同索引上的元素完全相同时，才代表完全相同
    //记住：只要所有元素相同，无所谓索引时，最好覆盖此方法哦
    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());
    }

    //简单的覆盖 Object 的 hashCode 方法。
    //  利用 iterator 方法，遍列所有元素，前面所有元素的 hash 值乘 31 加当前元素的 hash 值
    //为什么这样做呢？？？？
    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;
    }

}//类  AbstractList 结束
```

```
//奇怪哦！下面的类可不是什么 AbstractList 的内部类哦，它可在 AbstractList 的外部哦！
//这是一个供 AbstractList 使用的辅助类哦！若不想给其它类使用，只要定义为内部的私有类就可以了
//简单的装饰器模式的使用！（怎么又是模式啊！呵呵，JDK 就是一个宝藏，有很多值得学习的设计理念）
//学习模式吧！要不然你永远是个门外汉，永远不能挖掘 JDK 这个宝藏，为几所用！
class SubList<E> extends AbstractList<E> {       //为什么继承 AbstractList 啊？
        //装饰器模式的用意：对某个对象增加额外的功能（所谓加强版）且提供相同的接口
    private AbstractList<E> l;            //装饰器模式的被装饰对象
    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;                //被装饰对象一般通过构造方法的参数传入，不信你研究 IO 包
        offset = fromIndex;
        size = toIndex - fromIndex;
        expectedModCount = l.modCount;  //看看 AbstractList 的属性 modCount 是不是 protected 的啊
    }

    public E set(int index, E element) {
        rangeCheck(index);            //辅助方法，检查索引是否超出范围
        checkForComodification();   //辅助方法，检查聚集的内部状态是否改变，同 Itr 内部类哦！
                                    //为什么啊？
        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();               //检查状态是否改变
        l.add(index+offset, element);           //调用此方法内部状态肯定改变，l.modCount 递增
```

```
        expectedModCount = l.modCount;      //本 SubList 对象改变的状态，自己可以接收！
        size++;
        modCount++;                         //改变了聚集的状态，当然 modCount 要递增一个
                                            //前面的 l.add(index+offset, element)不是加了吗？怎么由加啊！
    }
    //自己改变，自己能接收！那别人能不能接收呢？当然不能啦！别人也自认它自己的

    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++;
            }
        };  //匿名内部类，该内部类又有自己的 ListIterator 属性，又存在所谓状态问题
    }

    public List<E> subList(int fromIndex, int toIndex) {
        return new SubList<E>(this, fromIndex, toIndex);      //在装饰器上继续使用装饰器
    }

    private void rangeCheck(int index) {          //检查索引是否越界的辅助方法
        if (index<0 || index>=size)
            throw new IndexOutOfBoundsException("Index: "+index+",Size: "+size);
    }

    private void checkForComodification() {     //检查状态是否改变的辅助方法
        if (l.modCount != expectedModCount)
            throw new ConcurrentModificationException();
    }
}

//供 AbstractList 使用的辅助类！
//还是装饰器模式吧！只是没有提供复杂的额外功能罢了
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);
    }
}
```

总结一下上面的代码：

1. AbstractList 抽象类，实现了 List 等接口，利用以下 4 个抽象方法或“消息实现”的方法，完成其实现接口所定义的其它方法，子类只要实现这 4 个方法即可。

   ```
   abstract public E get(int index)
   public E set(int index, E element)
   public void add(int index, E element)
   public E remove(int index)
   ```

2. JDK 中所谓的 FailFast 机制，就是在错误发生后，其它对象立即会有所反应。不会留待以后处理，或者忽略不计。可以参阅 2004 年《IEEE Software》中 Martin Fowler 所写的《Fail Faster》。
3. 一般书中所将的以下代码是错误的，可以参阅**蓝色**代码①-⑤。

   ```
   Collection c = new ArrayList();
   Iterator it = c.iterator();
   ```

```
while ( it.hasNext() ){
    Object o1 = it.next();
    Object o2 = it.next();
}          //具体原因看代码好了！
```

4. AbstractList 中的 FailFast 机制在 Itr 内部类，以及 SubList 辅助类就可以初见端倪！总之，创建遍历器或者其它对象时，保存此时聚集内部的属性 modCount，在要进行相应的操作时检查此属性是否已经改变。要知道这些属性什么时候真正改变请继续下面的 ArrayList。

六、ArrayList 类

ArrayList 是一个内部使用数组作为容器的 AbstractList 抽象类的具体实现。由于其内部使用数组存储，同时 java 中提供操作数组的方法，因此该类中覆盖了不少其父类的方法。下面是源码：

```
package java.util;
import java.util.*; // for javadoc (till 6280605 is fixed)

public class ArrayList<E> extends AbstractList<E>
                                    implements List<E>, RandomAccess, Cloneable, java.io.Serializable {
    private static final long serialVersionUID = 8683452581122892189L;
    private transient E[] elementData;              //保存元素的数组
    private int size;                               //数组中已有元素的个数

    public ArrayList(int initialCapacity) {         //提供元素个数的构造器
        super();
        if (initialCapacity < 0)    throw new IllegalArgumentException("Illegal Capacity: "+ initialCapacity);
        this.elementData = (E[])new Object[initialCapacity];         //元素的数组被创建
    }

    public ArrayList() {            this(10);       }     //默认构造器，默认只有 10 个元素

    public ArrayList(Collection<? extends E> c) {           //由 Collection 具体创建此 ArrayList 对象
        size = c.size();
        int capacity = (int) Math.min((size*110L)/100, Integer.MAX_VALUE);          //扩容 10%
        elementData = (E[]) c.toArray(new Object[capacity]);
        //将 Collection 中的元素拷贝至 ArrayList 中
        //可以使用 Collection 的 iterator 方法，但是没有 toArray 方法效率高
        //其次，toArray 方法也架起 Collection 向 Array 转换的桥梁
    }

    public void trimToSize() {
        modCount++;                     //记得吧！父类 AbstractList 的属性，FastFail 机制的核心
        int oldCapacity = elementData.length;
        if (size < oldCapacity) {
            elementData = Arrays.copyOf(elementData, size);
        }
        //该方法没有改变聚集内部的元素啊！怎么 modCount 要变呢？？？
```

```
        //数组中的个数没有变，内容没有变！
        //但是数组的大小变了啊－－末尾不存在元素的位置被 trim 了啊！！！
    }

    public void ensureCapacity(int minCapacity) {
        modCount++;
        int oldCapacity = elementData.length;
        if (minCapacity > oldCapacity) {
            Object oldData[] = elementData;
            int newCapacity = (oldCapacity * 3)/2 + 1;
            if (newCapacity < minCapacity)          newCapacity = minCapacity;
                // minCapacity is usually close to size, so this is a win:
            elementData = Arrays.copyOf(elementData, newCapacity);
        }
        //扩容方法，modCount 肯定要变啦！
        //扩容策略为：1.扩容小于原来数组长度时，不睬你－－越扩越小啊！
        //                  2.系统推荐你至少扩容 1.5 倍加 1，比建议小你就扩容到建议大小；
        //                      比建议大，就是想扩的大小。
        //英名吗！要是我们领导也这样，总部要加 20%的工资，他说不行，
        //大家太辛苦了至少加 30%就好了。唉！不奢求，本本分分，改多少就多少已经心满意足了
    }

    public int size() {          //返回聚集中元素的数目
        return size;
    }

    public boolean isEmpty() {    //判断聚集是否为空
        return size == 0;
    }

    public boolean contains(Object o) {      //使用下面的方法，查询是否存在指定元素
        return indexOf(o) >= 0;
    }

    public int indexOf(Object o) {       //返回指定元素的索引，不存在返回-1
        if (o == null) {
            for (int i = 0; i < size; i++)
                if (elementData[i]==null)       return i;         //聚集内部可以包含 null 哦，为什么啊？
                                                    //数组后面没有存入对象的部分不就是 null 吗
        } else {
            for (int i = 0; i < size; i++)
                if (o.equals(elementData[i]))         return i;
        }
        return -1;
    }

    public int lastIndexOf(Object o) {        //上面方法的变体，从后向前遍列吗！！！
```

```
        if (o == null) {
            for (int i = size-1; i >= 0; i--)
                if (elementData[i]==null)       return i;
        } else {
            for (int i = size-1; i >= 0; i--)
                if (o.equals(elementData[i]))       return i;
        }
        return -1;
    }

    public Object clone() {         //克隆该对象！将其元素取出，放入新的克隆对象中
        try {
            ArrayList<E> v = (ArrayList<E>) super.clone();
            v.elementData = Arrays.copyOf(elementData, size);
            v.modCount = 0;
            return v;
        } catch (CloneNotSupportedException e) {
            // this shouldn't happen, since we are Cloneable
            throw new InternalError();
        }
    }

    public Object[] toArray() {    //ArrayList 对象转换为数组对象，Collection 与 Array 转换的一部分
        return Arrays.copyOf(elementData, size);
    }

    public <T> T[] toArray(T[] a) {
        if (a.length < size)        return (T[]) Arrays.copyOf(elementData, size, a.getClass());
        //当指定数组的元素的个数小于，实际聚集中元素的个数时，
        //调用 Arrays 类的类方法将元素拷入指定类型的数组中。
        //一般地使用只包含一个元素的数组，主要是用于确定返回数组的类型！
        //高效而无需类型转换
        System.arraycopy(elementData, 0, a, 0, size);
        if (a.length > size)        a[size] = null;
        return a;
    }

    public E get(int index) {               //返回指定索引处的元素
        RangeCheck(index);
        return elementData[index];
    }

    public E set(int index, E element) {            //在指定索引处，设置新元素，同时返回已有元素
        RangeCheck(index);
        E oldValue = elementData[index];
        elementData[index] = element;
        return oldValue;
```

```
}

public boolean add(E e) {            //在末端增加新元素
    ensureCapacity(size + 1);       // 先扩容
    elementData[size++] = e;      //再增加新元素
    return true;
}

public void add(int index, E element) {              //在指定位置增加新元素
    if (index > size || index < 0)
        throw new IndexOutOfBoundsException("Index: "+index+", Size: "+size);

    ensureCapacity(size+1);  // Increments modCount!!
    System.arraycopy(elementData, index, elementData, index + 1, size - index);
    //将指定位置及以后的元素向后挪一位，然后在指定位置增加新元素
    elementData[index] = element;
    size++;
}

public E remove(int index) {        //删除指定位置处的元素
    RangeCheck(index);
    modCount++;              //删除内部元素，自然改变内部状态
    E oldValue = elementData[index];
    int numMoved = size - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, index+1, elementData, index, numMoved);
    //将指定位置以后的元素，向前挪动一个位置
    elementData[--size] = null; // Let gc do its work
    //原先最后一个位置的元素还是存在的，现在必须清除。断绝该数组最后元素指向其的指针
    //垃圾回收器会在恰当的时候将此没有被引用的对象回收
    return oldValue;
}

public boolean remove(Object o) {       //删除某个对象
    if (o == null) {
        for (int index = 0; index < size; index++)
            if (elementData[index] == null) {         //ArrayList 中的元素可以为 null 哦
                fastRemove(index);
                return true;
            }
    } else {
        for (int index = 0; index < size; index++)
            if (o.equals(elementData[index])) {
                //注意此方法，假如聚集中的元素一般要覆盖父类的 equals 方法
                fastRemove(index);
                return true;
            }
```

```
        }
        return false;
    }

    private void fastRemove(int index) {           //快速移除某个位置的元素
        modCount++;                                //聚集的内部状态改变
        int numMoved = size - index - 1;
        if (numMoved > 0)
            System.arraycopy(elementData, index+1, elementData, index, numMoved);
        elementData[--size] = null; // Let gc do its work
    }

    public void clear() {              //清除聚集中的内容
        modCount++;        //当然聚集内部状态改变了！
        // Let gc do its work
        for (int i = 0; i < size; i++)          //将数组所有的对对象的应用设置为 null
            elementData[i] = null;
        size = 0;                              //聚集中没有元素了，大小肯定为 0
    }

    public boolean addAll(Collection<? extends E> c) {          //覆盖了父类的方法，主要出于效率考虑
        Object[] a = c.toArray();       //将被添加聚集中的元素拷贝出，比父类中使用 iterator 效率高
        int numNew = a.length;
        ensureCapacity(size + numNew);  // 增加大量的元素，肯定要扩容了
        System.arraycopy(a, 0, elementData, size, numNew);  //将被假如的元素，拷贝入数组的最后面
        size += numNew;
        return numNew != 0;   //加入的为非空聚集返回 true，即完成假如操作。反之没有加入则返回 false
    }

    public boolean addAll(int index, Collection<? extends E> c) {    //从指定位置加入
        if (index > size || index < 0)
            throw new IndexOutOfBoundsException("Index: " + index + ", Size: " + size);
        Object[] a = c.toArray();
        int numNew = a.length;
        ensureCapacity(size + numNew);    // Increments modCount
        int numMoved = size - index;
        if (numMoved > 0)
            System.arraycopy(elementData, index, elementData, index + numNew, numMoved);
        System.arraycopy(a, 0, elementData, index, numNew);
        size += numNew;
        return numNew != 0;
    }

    protected void removeRange(int fromIndex, int toIndex) {         //删除指定索引间的元素
        //覆盖了父类的方法，还是处于效率的考虑，主要因为本 ArrayList 内部使用数组来存储元素，
        //在 java 中数组拷贝效率较高，而采用父类的 iterator 方法比较通用，但是效率不高
        modCount++;                                    //删除多个元素，肯定状态改变
```

```
        int numMoved = size - toIndex;                  //指定索引后需要移动的元素
        System.arraycopy(elementData, toIndex, elementData, fromIndex, numMoved);
        //将指定索引后的元素挪动到开始索引处
        int newSize = size - (toIndex-fromIndex);    //数组中现有元素的个数，有用的，即没有删除的
        while (size != newSize)
            elementData[--size] = null;
            //清除数组对无用元素的引用，这些没有被引用的元素留待垃圾回收器搜集
    }

    private void RangeCheck(int index) {   //简单的检查索引是否越界的辅助方法
        if (index >= size)
            throw new IndexOutOfBoundsException("Index: "+index+", Size: "+size);
    }


    //用于 ArrayList 对象序列化的方法，将实例写入流中
    private void writeObject(java.io.ObjectOutputStream s) throws java.io.IOException{
        int expectedModCount = modCount;    //写开始前确认内部状态改变属性 modCount
        s.defaultWriteObject();         //序列化机制提供的写入一般属性的默认方法，
                                        //若要了解对象序列化的细节，请查阅其它文章
        s.writeInt(elementData.length);          //写入数组元素的个数
        for (int i=0; i<size; i++)
            s.writeObject(elementData[i]);      //将数组中的元素依次写入流中
        if (modCount != expectedModCount) {       //写结束时检查此时聚集的状态是否改变
                                                  //若状态改变抛出异常，此次序列化失败
            throw new ConcurrentModificationException();
        }
    }

    private void readObject(java.io.ObjectInputStream s)
                                                  throws java.io.IOException, ClassNotFoundException {
        s.defaultReadObject();              //实现 Serializable 接口的类提供的默认方法
                                            //该方法与 java 的序列化机制有关，请查阅有关文章
        int arrayLength = s.readInt();      //从流中读入数组的长度
        Object[] a = elementData = (E[])new Object[arrayLength];            //创建该长度的数组
        for (int i=0; i<size; i++)
            a[i] = s.readObject();                  //从流中依次读取元素，将其存在数组中对应的位置
    }
    //以上两个方法与 java 的序列化机制相关！与 java IO 有关！关于其细节请参考本战的相关文章

}
```

七、其它

简单介绍一下几个模式的用意（详细的请查阅本站的其它文章）：

遍历器模式：主要用于对聚集内部元素的遍列。为聚集提供了一个面向客户的窄接口，从而不必暴露聚集内部容器的具体实现。JDK 中在集合框架中使用很多。

装饰器模式：为某个类提供装饰作用，同时向外界提供统一的接口。JDK 的 IO 包中使用很典型。

通过以上的代码分析，将有助于你研究 Java Collections Framework 中代码，通过此方法大致可以完成 60%-80%。其它的部分陆续会介绍。

感谢网友 Eric Liu 的更正！

protected transient int modCount = 0;　　//transient 是供 java 序列化使用的！

2006-11-30 创作

2006-12-01 第一次更正