问题

（1）集合（Collection）和集合（Set）有什么区别?

（2）HashSet怎么保证添加元素不重复?

（3）HashSet是否允许null元素?

（4）HashSet是有序的吗?

（5）HashSet是同步的吗?

（6）什么是fail-fast?

简介

集合，这个概念有点模糊。

广义上来讲，java中的集合是指 `java.util` 包下面的容器类，包括和Collection及Map相关的所有类。

中义上来讲，我们一般说集合特指java集合中的Collection相关的类，不包含Map相关的类。

狭义上来讲，数学上的集合是指不包含重复元素的容器，即集合中不存在两个相同的元素，在java里面对应Set。

具体怎么来理解还是要看上下文环境。

比如，面试别人让你说下java中的集合，这时候肯定是广义上的。

再比如，下面我们讲的把另一个集合中的元素全部添加到Set中，这时候就是中义上的。

HashSet是Set的一种实现方式，底层主要使用HashMap来确保元素不重复。

源码分析

属性

```
1.      // 内部使用HashMap
2.      private transient HashMap<E,Object> map;
3.
4.      // 虚拟对象，用来作为value放到map中
5.      private static final Object PRESENT = new Object();
6.
```

构造方法

```
1.  public HashSet() {
2.      map = new HashMap<>();
3.  }
4.
5.  public HashSet(Collection<? extends E> c) {
6.      map = new HashMap<>(Math.max((int) (c.size()/.75f) + 1, 16));
7.      addAll(c);
8.  }
9.
10. public HashSet(int initialCapacity, float loadFactor) {
11.     map = new HashMap<>(initialCapacity, loadFactor);
12. }
```

```
13.
14.     public HashSet(int initialCapacity) {
15.         map = new HashMap<>(initialCapacity);
16.     }
17.
18.     // 非public，主要是给LinkedHashSet使用的
19.     HashSet(int initialCapacity, float loadFactor, boolean dummy) {
20.         map = new LinkedHashMap<>(initialCapacity, loadFactor);
21.     }
22.
```

构造方法都是调用HashMap对应的构造方法。

最后一个构造方法有点特殊，它不是public的，意味着它只能被同一个包或者子类调用，这是LinkedHashSet专属的方法。

添加元素

直接调用HashMap的put()方法，把元素本身作为key，把PRESENT作为value，也就是这个map中所有的value都是一样的。

```
1.     public boolean add(E e) {
2.         return map.put(e, PRESENT)==null;
3.     }
4.
```

删除元素

直接调用HashMap的remove()方法，注意map的remove返回是删除元素的value，而Set的remov返回的是boolean类型。

这里要检查一下，如果是null的话说明没有该元素，如果不是null肯定等于PRESENT。

```
1.     public boolean remove(Object o) {
2.         return map.remove(o)==PRESENT;
3.     }
4.
```

查询元素

Set没有get()方法哦，因为get似乎没有意义，不像List那样可以按index获取元素。

这里只要一个检查元素是否存在的方法contains()，直接调用map的containsKey()方法。

```
1.     public boolean contains(Object o) {
2.         return map.containsKey(o);
3.     }
4.
```

遍历元素

直接调用map的keySet的迭代器。

```
1.     public Iterator<E> iterator() {
2.         return map.keySet().iterator();
3.     }
4.
```

全部源码

```
package java.util;

import java.io.InvalidObjectException;
import sun.misc.SharedSecrets;


public class HashSet<E>
    extends AbstractSet<E>
    implements Set<E>, Cloneable, java.io.Serializable
{
    static final long serialVersionUID = -5024744406713321676L;

    // 内部元素存储在HashMap中
    private transient HashMap<E,Object> map;

    // 虚拟元素，用来存到map元素的value中的，没有实际意义
    private static final Object PRESENT = new Object();

    // 空构造方法
    public HashSet() {
        map = new HashMap<>();
    }

    // 把另一个集合的元素全都添加到当前Set中
    // 注意，这里初始化map的时候是计算了它的初始容量的
    public HashSet(Collection<? extends E> c) {
        map = new HashMap<>(Math.max((int) (c.size()/.75f) + 1, 16));
        addAll(c);
    }

    // 指定初始容量和装载因子
    public HashSet(int initialCapacity, float loadFactor) {
        map = new HashMap<>(initialCapacity, loadFactor);
    }

    // 只指定初始容量
    public HashSet(int initialCapacity) {
        map = new HashMap<>(initialCapacity);
    }

    // LinkedHashSet专用的方法
    // dummy是没有实际意义的，只是为了跟上上面那个操持方法签名不同而已
    HashSet(int initialCapacity, float loadFactor, boolean dummy) {
        map = new LinkedHashMap<>(initialCapacity, loadFactor);
    }

    // 迭代器
    public Iterator<E> iterator() {
        return map.keySet().iterator();
    }

    // 元素个数
    public int size() {
        return map.size();
    }

    // 检查是否为空
    public boolean isEmpty() {
        return map.isEmpty();
    }

    // 检查是否包含某个元素
    public boolean contains(Object o) {
```

```
64.             return map.containsKey(o);
65.         }
66.
67.         // 添加元素
68.         public boolean add(E e) {
69.             return map.put(e, PRESENT)==null;
70.         }
71.
72.         // 删除元素
73.         public boolean remove(Object o) {
74.             return map.remove(o)==PRESENT;
75.         }
76.
77.         // 清空所有元素
78.         public void clear() {
79.             map.clear();
80.         }
81.
82.         // 克隆方法
83.         @SuppressWarnings("unchecked")
84.         public Object clone() {
85.             try {
86.                 HashSet<E> newSet = (HashSet<E>) super.clone();
87.                 newSet.map = (HashMap<E, Object>) map.clone();
88.                 return newSet;
89.             } catch (CloneNotSupportedException e) {
90.                 throw new InternalError(e);
91.             }
92.         }
93.
94.         // 序列化写出方法
95.         private void writeObject(java.io.ObjectOutputStream s)
```

```
96.             throws java.io.IOException {
97.             // 写出非static非transient属性
98.             s.defaultWriteObject();
99.
100.            // 写出map的容量和装载因子
101.            s.writeInt(map.capacity());
102.            s.writeFloat(map.loadFactor());
103.
104.            // 写出元素个数
105.            s.writeInt(map.size());
106.
107.            // 遍历写出所有元素
108.            for (E e : map.keySet())
109.                s.writeObject(e);
110.        }
111.
112.        // 序列化读入方法
113.        private void readObject(java.io.ObjectInputStream s)
114.            throws java.io.IOException, ClassNotFoundException {
115.            // 读入非static非transient属性
116.            s.defaultReadObject();
117.
118.            // 读入容量，并检查不能小于0
119.            int capacity = s.readInt();
120.            if (capacity < 0) {
121.                throw new InvalidObjectException("Illegal capacity: " +
122.                                                 capacity);
123.            }
124.
125.            // 读入装载因子，并检查不能小于等于0或者是NaN(Not a Number)
126.            // java.lang.Float.NaN = 0.0f / 0.0f;
127.            float loadFactor = s.readFloat();
```

```
            if (loadFactor <= 0 || Float.isNaN(loadFactor)) {
                throw new InvalidObjectException("Illegal load factor: " +
                                                 loadFactor);
            }

            // 读入元素个数并检查不能小于0
            int size = s.readInt();
            if (size < 0) {
                throw new InvalidObjectException("Illegal size: " +
                                                 size);
            }
            // 根据元素个数重新设置容量
            // 这是为了保证map有足够的容量容纳所有元素，防止无意义的扩容
            capacity = (int) Math.min(size * Math.min(1 / loadFactor, 4.0f),
                    HashMap.MAXIMUM_CAPACITY);

            // 再次检查某些东西，不重要的代码忽视掉
            SharedSecrets.getJavaOISAccess()
                         .checkArray(s, Map.Entry[].class, HashMap.tableSizeFor(capacity));

            // 创建map，检查是不是LinkedHashSet类型
            map = (((HashSet<?>)this) instanceof LinkedHashSet ?
                   new LinkedHashMap<E,Object>(capacity, loadFactor) :
                   new HashMap<E,Object>(capacity, loadFactor));

            // 读入所有元素，并放入map中
            for (int i=0; i<size; i++) {
                @SuppressWarnings("unchecked")
                    E e = (E) s.readObject();
                map.put(e, PRESENT);
            }
        }

        // 可分割的迭代器，主要用于多线程并行迭代处理时使用
        public Spliterator<E> spliterator() {
            return new HashMap.KeySpliterator<E,Object>(map, 0, -1, 0, 0);
        }
    }
```

总结

（1）HashSet内部使用HashMap的key存储元素，以此来保证元素不重复；

（2）HashSet是无序的，因为HashMap的key是无序的；

（3）HashSet中允许有一个null元素，因为HashMap允许key为null；

（4）HashSet是非线程安全的；

（5）HashSet是没有get()方法的；

彩蛋

（1）阿里手册上有说，使用java中的集合时要自己指定集合的大小，通过这篇源码的分析，你知道初始化HashMap的时候初始容量怎么传吗？

我们发现有下面这个构造方法，很清楚明白地告诉了我们怎么指定容量。

假如，我们预估HashMap要存储n个元素，那么，它的容量就应该指定为((n/0.75f) + 1)，如果这个值小于16，那就直接使用16得了。

初始化时指定容量是为了减少扩容的次数，提高效率。

```
    public HashSet(Collection<? extends E> c) {
        map = new HashMap<>(Math.max((int) (c.size()/.75f) + 1, 16));
        addAll(c);
    }

```

（2）什么是fail-fast?

fail-fast机制是java集合中的一种错误机制。

当使用迭代器迭代时，如果发现集合有修改，则快速失败做出响应，抛出ConcurrentModificationException异常。

这种修改有可能是其它线程的修改，也有可能是当前线程自己的修改导致的，比如迭代的过程中直接调用remove()删除元素等。

另外，并不是java中所有的集合都有fail-fast的机制。比如，像最终一致性的ConcurrentHashMap、CopyOnWriterArrayList等都是没有fast-fail的。

那么，fail-fast是怎么实现的呢?

细心的同学可能会发现，像ArrayList、HashMap中都有一个属性叫 `modCount` ，每次对集合的修改这个值都会加1，在遍历前记录这个值到 `expectedModCount` 中，遍历中检查两者是否一致，如果出现不一致就说明有修改，则抛出ConcurrentModificationException异常。