死磕 java集合之HashSet源码分析

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。

Set特点：无序不可重复

源码分析

属性

```
    // 内部使用HashMap
    private transient HashMap<E,Object> map;

    // 虚拟对象，用来作为value放到hashmap中,所有放进set中的对象，都是作为key放入hashmap中
    private static final Object PRESENT = new Object();
```

构造方法

```
public HashSet() {
    map = new HashMap<>();HashSet内部使用HashMap存储Set(value)中的value数值
}

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

public HashSet(int initialCapacity, float loadFactor) {
    map = new HashMap<>(initialCapacity, loadFactor);
}

public HashSet(int initialCapacity) {
    map = new HashMap<>(initialCapacity);
}

// 非public，主要是给LinkedHashSet使用的
HashSet(int initialCapacity, float loadFactor, boolean dummy) {
    map = new LinkedHashMap<>(initialCapacity, loadFactor);
}
```

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。

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

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。

```
public boolean remove(Object o) {
    return map.remove(o)==PRESENT;
```

```
}
```

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。

```
public boolean contains(Object o) {
    return map.containsKey(o);
}
```

遍历元素

直接调用map的keySet的迭代器。

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

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) {
        return map.containsKey(o);
    }

    // 添加元素
    public boolean add(E e) {
        return map.put(e, PRESENT)==null;
    }

    // 删除元素
    public boolean remove(Object o) {
        return map.remove(o)==PRESENT;
    }

    // 清空所有元素
    public void clear() {
        map.clear();
    }

    // 克隆方法
    @SuppressWarnings("unchecked")
    public Object clone() {
        try {
            HashSet<E> newSet = (HashSet<E>) super.clone();
            newSet.map = (HashMap<E, Object>) map.clone();
            return newSet;
        } catch (CloneNotSupportedException e) {
            throw new InternalError(e);
        }
    }
```

```
    // 序列化写出方法
    private void writeObject(java.io.ObjectOutputStream s)
        throws java.io.IOException {
        // 写出非static非transient属性
        s.defaultWriteObject();

        // 写出map的容量和装载因子
        s.writeInt(map.capacity());
        s.writeFloat(map.loadFactor());

        // 写出元素个数
        s.writeInt(map.size());

        // 遍历写出所有元素
        for (E e : map.keySet())
            s.writeObject(e);
    }

    // 序列化读入方法
    private void readObject(java.io.ObjectInputStream s)
        throws java.io.IOException, ClassNotFoundException {
        // 读入非static非transient属性
        s.defaultReadObject();

        // 读入容量，并检查不能小于0
        int capacity = s.readInt();
        if (capacity < 0) {
            throw new InvalidObjectException("Illegal capacity: " +
                                             capacity);
        }

        // 读入装载因子，并检查不能小于等于0或者是NaN(Not a Number)
        // java.lang.Float.NaN = 0.0f / 0.0f;
        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。

如何理解Set的无序性和不可重复性？

无序性（hashcode有关）：不等于随机性。HashSet存储的数据在底层数组中并非按照数组索引的顺序添加，而是根据数据的hashcode计算出的索引，添加到对应的数组索引或者数组索引指向的链表/红黑树中
不可重复性（equals有关）：保证添加的元素按照equals判断时候，不能返回true，即被添加对象的equals判断相等的两个，不能都添加到set中。

Set放进的对象没有重写hashcode和equals方法

```
public class TestSet {
    public static void main(String[] args) {
        Set set=new HashSet();
        set.add(123);
        set.add("ABC");
        set.add(new Person(1,12,"Tom"));
        set.add(new Person(1,12,"Tom"));
        Iterator iterator = set.iterator();
        while (iterator.hasNext()){
            System.out.println(iterator.next());
        }
    }
}
```

```
class Person{
    private int id;
    private int age;
    private String name;
    public Person(int id, int age, String name) {
        this.id = id;
        this.age = age;
        this.name = name;
    }
    @Override
    public String toString() {
        return "Person{" +
                "id=" + id +
                ", age=" + age +
                ", name='" + name + '\'' +
                '}';
    }
}
```

解释：Person未重写hashcode和equals，因此放入set集合时候根据Peron1的hashcode计算int数值，放入set中，然后Person2在计算hashcode，由于Person1,Person2的hashcode极大概率不相等，极大概率放在Set底层的HashMap的底层的数组的不同位置的key中，因此Person1,Person2根本就是放在数组的不同index位置，因此更用不到判断equals。因此Person1,Person2都放入了HashSet中。如果规则两个Person对象相同的规则是，id相同的两个对象认为是同一个人，不能同时放入集合，此时就需要重写hashcode和equals方法。

需求：两个Person对象的id一样，认为是同一个Person,不允许同时放入set(重写hashcode和equals)

重写Person的hashcode和equasl

```
@Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Person person = (Person) o;
    return id == person.id;两个Person的id相同则认为两个Person是equals的
  }

  @Override
  public int hashCode() {
    return Objects.hash(id);
  }
```

此时打印输出：（只打印第一个Person对象）

```
Person{id=1, age=12, name='Tom'}
ABC
123
```

解释：第二个Person对象加入set时候，根据id计算出来了hashcode，这个hashcode等同于第一个Person对象的hashcode，因此第二个对象放入set时候计算出的索引和第一个对象一样，因此需要比较第二个Person的equals方法是否等于第一个放进去的Person对象，此时是相等的，因此第二个Person不放入set中。

hashcode和equals重写原则

如果两个对象的equals相等，则必须hashcode此时也是相等。equals -> hashcode
但是如果两个对象的hashcode相等，不必要求equals一定相等，同一个对象多次执行hashcode必须返回同一个值。

对象不重写hashcode和equasl可能有什么问题?

Set:假设我们逻辑上认为两个Person的id相同就认为是同一个对象，那么两个id属性一样的Person可以放到同一个set中。

```
Set set=new HashSet();
set.add(new Person(1,12,"Tom"));
set.add(new Person(1,12,"Tom"));两个Person都能放进去Set中
```

Map:假设我们逻辑上认为两个Person的id相同就认为是同一个对象，并且Person作为Map的key,则导致导致放进Map的value通过get取不出来，导致内存泄漏

```
Map map=new HashMap();
map.put(new Person(1,12,"Tom")," I am Tom");
System.out.println(map.get(new Person(1,12,"Tom"))); get null，放进map的“I am Tom”不能通过get获得，导致map的value内存泄漏。原因：Person未重写hashcode,则两个Person的hashcode不相等，第二个Person计算出来的index肯定和第一个不一样，导致get不到第一个Person对应的value "I am Tom"
```

修改equals为什么要重写hashcode?（只重写equals没重写hashcode）

重写equals说明，我们判断两个对象是否相等的规则发生了变化，比如重写equals后根据对象的id判断两个对象是否相等，如果只重写了equals没有重写hashcode，则导致Person仍然使用的是Object的hashcode，同样导致上面两个问题，set可以放进去两个id相同的Person,Map集合Person作为key放进去的value取不出来。

只重写hashcode没有重写equals 会发生什么

1. set中，多个Person对象即使hashcode重写了，根据id,name,age重写的hashcode，多个Person(id,name,age)属性相同的对象都可以放到set集合中，因为equals没重写，使用==判断，多个Person不相等，因此都可以放进set
2. map中，作为key的Person放入map,则map.put(new Person(id,name,age),"value")，之后再map.get(new Person(id,name,age))返回null,仍然是导致map的value对象内存泄漏