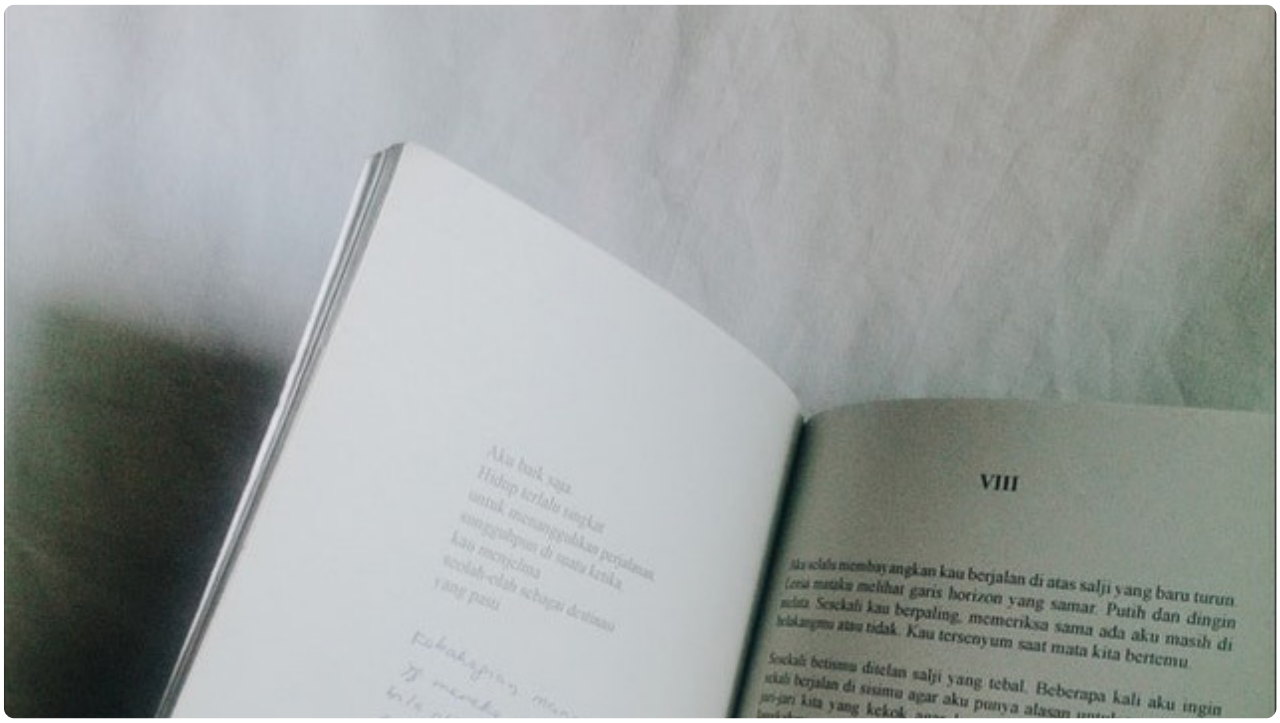
14 集合去重的正确姿势

更新时间：2019-11-18 12:12:06



> 受苦的人，没有悲观的权利。——尼采

1. 前言

《手册》的第 11 页 关于集合处理的章节有这样的描述 1：

> 【参考】利用 Set 元素唯一的特性，可以快速对一个集合进行去重操作，避免使用 List 的 contains 方法进行遍历、对比、去重操作。
>
> 【强制】关于 hashCode 和 equals 的处理，遵循如下规则:
>
> 1. 只要覆写 equals，就必须覆写 hashCode；
> 2. 因为 Set 存储的是不重复的对象，依据 hashCode 和 equals 进行判断，所以 Set 存储的对象必须覆写这两个方法；
> 3. 如果自定义对象作为 Map 的键，那么必须覆写 hashCode 和 equals。
>
>   说明：String 已覆写 hashCode 和 equals 方法，所以我们可以愉快地使用 String 对象作为 key 来使用。

可能很多人会认为工作之后不会有人通过 `List` 的 `contains` 函数来去重，然而，最可怕的是真的有...

那么我们思考下面几个问题：

- Set 是怎样保证数据的唯一性呢？
- Set 存储的是不重复的对象，是不是根据 hashCode 和 equals 来判断是否重复的呢？

- Set 和 List 的去重性能差距是多大呢？

本节将重点研究这几个问题。

2. 唯一性保证

开发中常见到使用 `Set` 去重的代码如下：

```
public static <T> Set<T> removeDuplicateBySet(List<T> data) {

    if (CollectionUtils.isEmpty(data)) {
        return new HashSet<>();
    }
    return new HashSet<>(data);
}
```

> 注：Set 自身可以保证不重复，不需要先通过 contains 判断不存在再添加元素。

我们先看 `java.util.HashSet` 的类注释 （注释内容省略，具体请大家看源码）中的一些要点：

> 实现了 `Set` 接口。
>
> 基于 `HashMap` 来实现核心功能。
>
> 允许存入 `null` 元素，不保证顺序。
>
> 该类的方法并没同步，如果想要同步需要外部处理，可以构造一个同步对象，也可以使用 `Collections#synchronizedSet` ，最佳实践：
>
> `Set s = Collections.synchronizedSet(new HashSet(...));`
>
> 迭代方法返回的迭代器是 “fail-fast” 的，迭代器创建后如果调用除了迭代器自己的 `remove` 函数外的其他修改方法，会抛出：`ConcurrentModificationException`。

我们再看看 `HashSet` 对应的构造函数 `java.util.HashSet#HashSet(java.util.Collection<? extends E>)` 源码：

```
/**
 * Constructs a new set containing the elements in the specified
 * collection.  The <tt>HashMap</tt> is created with default load factor
 * (0.75) and an initial capacity sufficient to contain the elements in
 * the specified collection.
 *
 * @param c the collection whose elements are to be placed into this set
 * @throws NullPointerException if the specified collection is null
 */
public HashSet(Collection<? extends E> c) {
    map = new HashMap<>(Math.max((int) (c.size()/.75f) + 1, 16));
    addAll(c);
}
```

从这里我们看到了底层的确是通过 `HashMap` 支持的，根据参数集合的长度构造对应默认容量的 `HashMap`。

然后调用父类的 java.util.AbstractCollection#addAll （添加所有元素的函数）：

```
public boolean addAll(Collection<? extends E> c) {
    boolean modified = false;
    for (E e : c)
        if (add(e))
            modified = true;
    return modified;
}
```

从这里可以看出，通过 for-each 语法糖对集合进行迭代并调用 add 函数将元素依次添加到 HashSet 中。

```
// Dummy value to associate with an Object in the backing Map
private static final Object PRESENT = new Object();

/**
 * Adds the specified element to this set if it is not already present.
 * More formally, adds the specified element <tt>e</tt> to this set if
 * this set contains no element <tt>e2</tt> such that
 * <tt>(e==null ? e2==null : e.equals(e2))</tt>.
 * If this set already contains the element, the call leaves the set
 * unchanged and returns <tt>false</tt>.
 *
 * @param e element to be added to this set
 * @return <tt>true</tt> if this set did not already contain the specified
 * element
 */
public boolean add(E e) {
    return map.put(e, PRESENT)==null;
}
```

通过这个函数的注释，我们可以看到：

> 该函数的功能是添加 set 中没添加过的元素。
>
> 更准确地说，如果想将元素 e 添加到此集合中，那么集合中不能存在元素 e2 满足：
>
> **(e== null ? e2 ==null : e.equals(e2))** 。
>
> 如果已经包含了该元素，那么集合将不会发生改变，将返回 false。

从这里我们还看到，为了保持 HashMap 的用法，这里给底层的 Map 的值传入一个傀儡对象（PRESENT）。

我们进入更底层源码 java.util.HashMap#put ：

```
public V put(K key, V value) {
    return putVal(hash(key), key, value, false, true);
}
```

通过这里我们看到，除了传入 key 和 value 外，第一个哈希值的参数 (hash) 是通过 HashMap 的 hash 函数实现的。

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

可以看到如果 key 为 null，哈希值为 0，否则将 key 通过自身 hashCode 函数计算的的哈希值和其右移 16 位进行异或运算得到最终的哈希值。

在 java.util.HashMap#putVal 中，直接通过 (n - 1) & hash 来得到当前元素在节点数组中的位置。如果不存在，直接构造新节点并存储到该节点数组的对应位置。如果存在，则通过下面逻辑：

```
p.hash == hash && ((k = p.key) == key || (key != null && key.equals(k)))
```

来判断元素是否相等。

如果相等则用新值替换旧值，否则添加红黑树节点或者链表节点。

这就是前言中第 2 和第 3 条规定的依据。

最终如果存在 key 则返回旧值，不存在则返回 null。

此时，我们再回看 java.util.HashSet#add 源码:

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

一切就非常清晰了。

通过 HashMap 的 key 的唯一性保证 HashSet 的元素的唯一性。

我们再看 HashSet 的迭代器 java.util.HashSet#iterator:

```
/**
     * Returns an iterator over the elements in this set.  The elements
     * are returned in no particular order.
     *
     * @return an Iterator over the elements in this set
     * @see ConcurrentModificationException
     */
    public Iterator<E> iterator() {
        return map.keySet().iterator();
    }
```

我们发现，其实 HashSet 的元素是存放在 HashMap 的 keySet 中。

大家可以进入 HashSet 的其他方法中查看，可以发现几乎 HashSet 的所有核心函数都是通过 HashMap 支撑的。

由于 HashSet 底层采用 HashMap 实现，通过上述分析，我们可知其 “去重” 的时间复杂度是 O (n)。

另外我们回答前言中 “关于 hashCode 和 equals 的处理” 的第 1 条：** 只要覆写 equals，就必须覆写 hashCode”。** 这个问题。

除了上面讲到的判断重复的依据外，从其源码 java.lang.Object#equals 的注释中也可以得到更本质的原因：

> Note that it is generally necessary to override the {@code hashCode} method whenever this method is overridden, so as to maintain the general contract for the {@code hashCode} method, which states that equal objects must have equal hash codes.
>
> 只要重写 equals 方法就要重新 hashCode 方法，来维持 hashCode 的约定，即 **equals 的对象的哈希值必须相等**。

3. 性能差异的原因

前面讲到“由于 `HashSet` 底层采用了 `HashMap` 实现，因此去重的时间复杂度是 O (n)”。

那么，通过 `List` 的 `contains` 函数来去重，原理又是怎样的呢？时间复杂度是多少呢？

且看下面基于 `List` 的 `contains` 函数来去重示例代码：

```
public static <T> List<T> removeDuplicateByList(List<T> data) {

    if (CollectionUtils.isEmpty(data)) {
        return new ArrayList<>();

    }
    List<T> result = new ArrayList<>(data.size());
    for (T current : data) {
        if (!result.contains(current)) {
            result.add(current);
        }
    }
    return result;
}
```

其实 `HashSet` 和 `ArrayList` 去重性能差异的核心在于 `contains` 函数性能对比。

我们分别查看 `java.util.HashSet#contains` 和 `java.util.ArrayList#contains` 的实现。

`java.util.HashSet#contains` 源码：

```
/**
 * Returns <tt>true</tt> if this set contains the specified element.
 * More formally, returns <tt>true</tt> if and only if this set
 * contains an element <tt>e</tt> such that
 * <tt>(o==null&amp;nbsp;?&amp;nbsp;e==null&amp;nbsp;:&amp;nbsp;o.equals(e))</tt>.
 *
 * @param o element whose presence in this set is to be tested
 * @return <tt>true</tt> if this set contains the specified element
 */
public boolean contains(Object o) {
    return map.containsKey(o);
}
```

最终是通过 `java.util.HashMap#getNode` 来判断的（和 `java.util.HashMap#putVal` 的一些判断非常相似）：

```
/**
 * Implements Map.get and related methods.
 *
 * @param hash hash for key
 * @param key the key
 * @return the node, or null if none
 */
final Node<K,V> getNode(int hash, Object key) {
    Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (first = tab[(n - 1) & hash]) != null) {
        if (first.hash == hash && // always check first node
            ((k = first.key) == key || (key != null && key.equals(k))))
            return first;
        if ((e = first.next) != null) {
            if (first instanceof TreeNode)
                return ((TreeNode<K,V>)first).getTreeNode(hash, key);
            do {
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    return e;
            } while ((e = e.next) != null);
        }
    }
    return null;
}
```

先通过计算过的 hash 值找到 table 对应索引的第一个元素进行比较，如果相等则返回第一个元素。

如果是树节点，从红黑树中查找该元素，否则在链表中查找该元素。

如果 hash 冲突不是极其严重（大多数都没怎么有哈希冲突），n 个元素依次判断并插入到 Set 的时间复杂度接近于 O (n)。

接下来我们看 java.util.ArrayList#contains 的源码：

```
/**
 * Returns <tt>true</tt> if this list contains the specified element.
 * More formally, returns <tt>true</tt> if and only if this list contains
 * at least one element <tt>e</tt> such that
 * <tt>(o==null&amp;nbsp;?&amp;nbsp;e==null&amp;nbsp;:&amp;nbsp;o.equals(e))</tt>.
 *
 * @param o element whose presence in this list is to be tested
 * @return <tt>true</tt> if this list contains the specified element
 */
public boolean contains(Object o) {
    return indexOf(o) >= 0;
}
```

其功能实现依赖于 java.util.ArrayList#indexOf：

```
/**
 * Returns the index of the first occurrence of the specified element
 * in this list, or -1 if this list does not contain the element.
 * More formally, returns the lowest index <tt>i</tt> such that
 * <tt>(o==null ? get(i)==null : o.equals(get(i)))</tt>,
 * or -1 if there is no such index.
 */
public int indexOf(Object o) {
    if (o == null) {
        for (int i = 0; i < size; i++)
            if (elementData[i]==null)
                return i;
    } else {
        for (int i = 0; i < size; i++)
            if (o.equals(elementData[i]))
                return i;
    }
    return -1;
}
```

发现其核心逻辑为：如果为 null，则遍历整个集合判断是否有 null 元素；否则遍历整个列表，通过 o.equals(当前遍历到的元素) 判断与当前元素是否相等，相等则返回当前循环的索引。

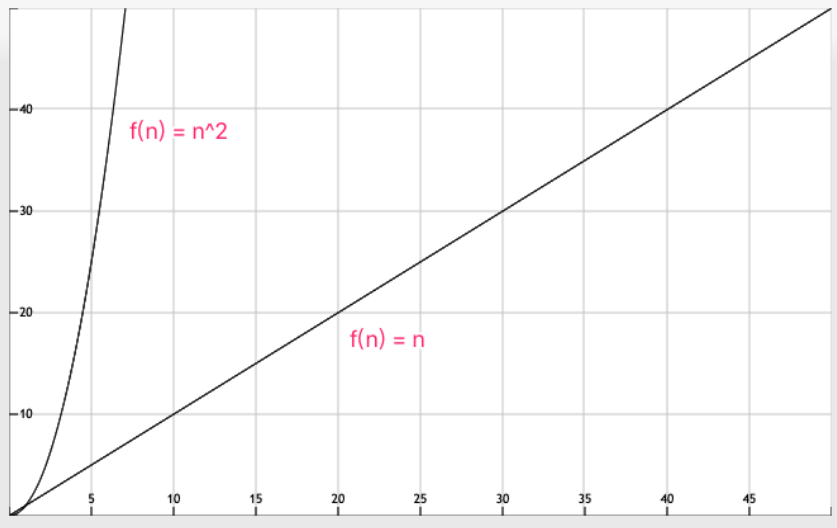
所以， n 个元素依次通过 java.util.ArrayList#contains 判断并插入到 Set 的时间复杂度接近于 O (n^2)。

因此，通过时间复杂度的比较，性能差距就不言而喻了。

4. 性能对比

上面我们分别对性能的差异原因， 时间复杂度进行了分析。

我们分别将两个时间复杂度函数进行作图， 两者增速对比非常明显：



实践是检验真理的标准，因此我们写一段代码粗略对比一下他们的性能差异：

```java
@Slf4j
public class SetDemo {

    public static void main(String[] args) {

        List<Integer> lengthList = new LinkedList<>();
        int base = 1;
        for (int i = 1; i <= 6; i++) {
            base *= 10;
            lengthList.add(base);
        }

        StringRandomizer stringRandomizer = new StringRandomizer(10, 100, 1000);

        for (Integer length : lengthList) {
            log.debug("----------长度为 {} 时-------", length);
            ListRandomizer<String> listRandomizer = new ListRandomizer<>(stringRandomizer, length);
            List<String> data = listRandomizer.getRandomValue();

            StopWatch stopWatch = new StopWatch();
            stopWatch.start();
            Set<String> resultBySet = CollectionUtil.removeDuplicateBySet(data);
            log.debug("set去重耗时：{} ms", stopWatch.getTime());

            stopWatch = new StopWatch();
            stopWatch.start();
            List<String> resultByList = CollectionUtil.removeDuplicateByList(data);
            log.debug("list去重耗时：{} ms", stopWatch.getTime());
        }

    }
}
```

最终得到下面的数据：

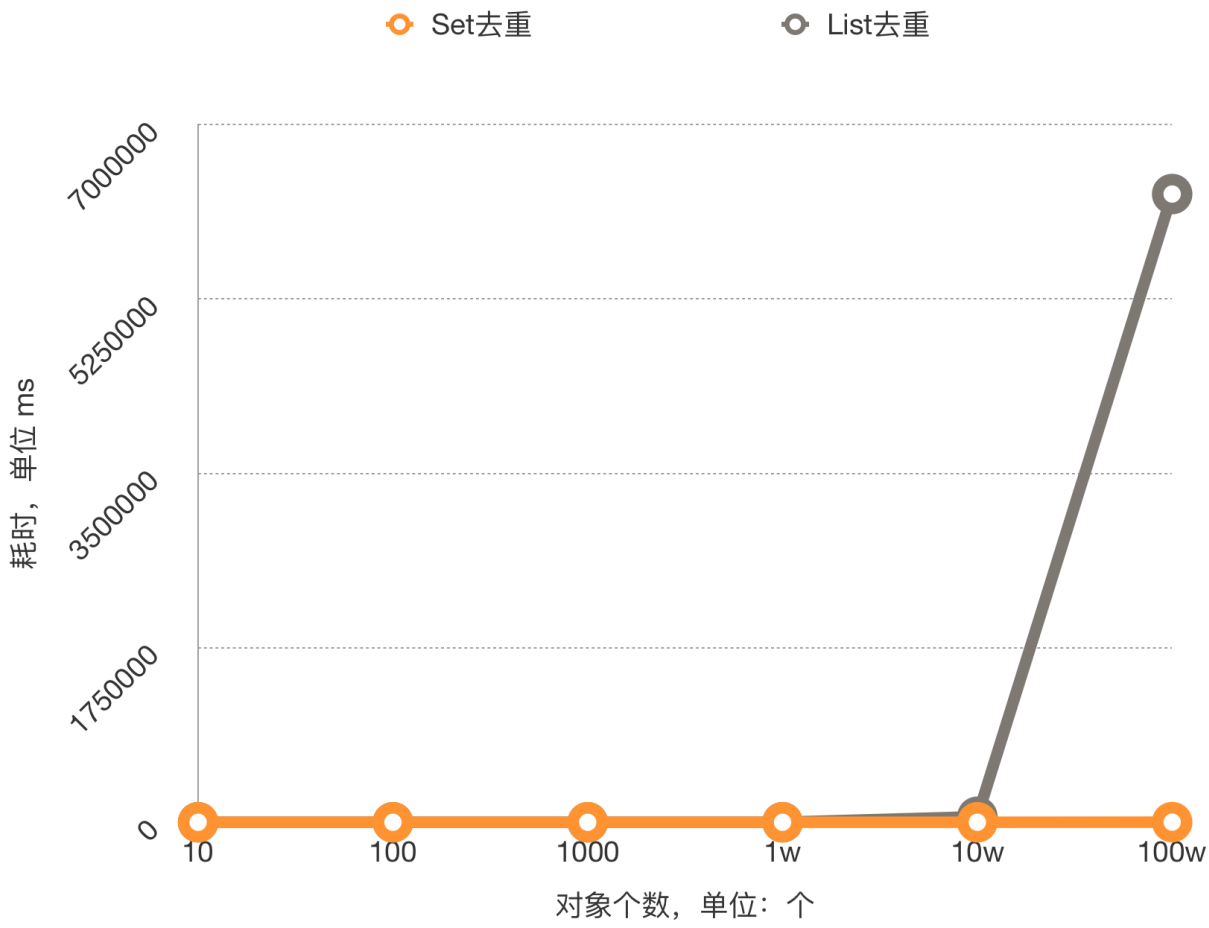
Set 和 List 去重耗时表，单位： 毫秒(ms)

描述	10	100	1000	1W	10W	100W
Set去重耗时	5	0	0	16	17	1333
List去重耗时	0	0	16	328	55539	6302916

我们重点观察最后两种情况：

> 长度为 10 万时使用 List 去重耗时接近 1 分钟，而使用 Set 去重则只需要 17 毫秒；
>
> 而集合长度为 100 万时，使用 List 去重，耗时则约为 1.7 小时，使用 Set 去重则只需要 1.33 秒。

对上述结果进行绘图如下：



通过此图，大家就可以非常直观地感受到两种去重方式的性能差异。

我们**发现当元素较少时两者耗时差距很小，随着元素的增多耗时差距越来越大**。

如果数据量不大时采用 `List` 去重勉强可以接受，但是数据量增大后，接口响应时间会超慢，这是难以忍受的，甚至造成大量线程阻塞引发故障。

> 在工作中一次排查慢接口时，查到了一个函数耗时较长，最终定位到是通过 List 去重导致的。
>
> 由于测试环境还有线上早期数据较少，这个接口的性能问题没有引起较大关注，后面频繁超时，才引起重视。

因此我们要养成好的编程习惯，尽可能地提高接口性能，避免因知识盲区导致故障。

5. 为什么有人会这么用？

最后我们思考一下：为什么有人会用 List 的 contains 方法进行遍历、对比然后去重呢？

无非就是两个原因：

- 基础不扎实，不了解这种操作的时间复杂度；
- 为了维持返回值的类型。

对于第一个问题，基础不扎实我们要加强学习，多注意代码规范和代码的运行效率。

第二个问题是一种直线思维，是一种偷懒的表现。

比如某种特殊场景下需要的返回值类型为 List，“因此” 有些朋友就会声明一个 List，通过 contains 方法进行遍历、对比、去重，然后将其作为返回值返回。

其实，这种情况可以分两步走，先去重，然后通过 ArrayList 参数为集合的构造方法创建 List 对象来实现类型“转换”，示例代码如下：

```
// 数据
List<String> data = listRandomizer.getRandomValue();
// 先去重
Set<String> resultBySet = CollectionUtil.removeDuplicateBySet(data);

// 再转换格式
ArrayList<String> result = new ArrayList<>(resultBySet);
```

6. 总结

本节主要讲述集合去重的正确姿势，主要要点有：

- HashSet 元素唯一性是通过 HashMap 的 key 唯一性来实现的；
- 性能的差距是元素查找函数的时间复杂度不同导致的；
- 元素较少时两者耗时差距很小，随着元素的增多耗时差距越来越大。

下一节我们将学习如何学习线程池。

参考资料

阿里巴巴与 Java 社区开发者.《 Java 开发手册 1.5.0》华山版. 2019. 11 □□

}

13 你真得了解可变参数吗?

15 学习线程池的正确姿势