Руководство по TreeSet в Java

Java Java Collections

1. Обзор

В этой статье мы рассмотрим неотъемлемую часть Java Collections Framework и **одну из самых популярных реализаций *Set* - *TreeSet*** .

2. Введение в *TreeSet*

Проще говоря, *TreeSet* это отсортированная коллекция, которая расширяет класс *AbstractSet* и реализует интерфейс *NavigableSet*

Вот краткий обзор наиболее важных аспектов этой реализации:

- Хранит уникальные элементы
- Не сохраняет порядок вставки элементов
- Сортирует элементы в порядке возрастания
- Это не потокобезопасный
- В этой реализации объекты сортируются и хранятся в порядке возрастания в соответствии с их естественным порядком ** . *TreeSet* использует самобалансирующееся двоичное дерево поиска, более конкретно https://en.wikipedia.org/wiki/Red%E2%80%93black *tree[a* Red-Black__ tree].

Проще говоря, будучи самобалансирующимся двоичным деревом поиска, каждый узел двоичного дерева содержит дополнительный бит, который используется для определения цвета узла, который является красным или черным. Во время последующих вставок и удалений эти «цветные» биты помогают обеспечить более или менее сбалансированное дерево.

Итак, давайте создадим экземпляр *TreeSet* :

```java
Set<String> treeSet = new TreeSet<>();
```

2.1. TreeSet с параметром компаратора конструктора

При желании мы можем создать *TreeSet* с помощью конструктора, который позволяет нам определять порядок сортировки элементов с помощью *Comparable* или *Comparator:*

```java
Set<String> treeSet = new TreeSet<>(Comparator.comparing(String::length));
```

- Хотя *TreeSet* не является потокобезопасным, его можно синхронизировать извне с помощью оболочки *Collections.synchronizedSet ()* : **

```java
Set<String> syncTreeSet = Collections.synchronizedSet(treeSet);
```

Хорошо, теперь, когда у нас есть четкое представление о том, как создать экземпляр *TreeSet* , давайте посмотрим на общие операции, которые мы имеем в наличии.

3. *TreeSet add ()*

Как и ожидалось, метод *add ()* можно использовать для добавления элементов в *TreeSet* . Если элемент был добавлен, метод возвращает *true,* в противном случае - *false*.

- В контракте метода указано, что элемент будет добавлен только тогда, когда того же самого еще нет в *Set* . **

Давайте добавим элемент в *TreeSet* :

```java
@Test
public void whenAddingElement__shouldAddElement() {
    Set<String> treeSet = new TreeSet<>();

    assertTrue(treeSet.add("String Added"));
 }
```

- Метод *add* чрезвычайно важен, так как детали реализации метода иллюстрируют внутреннюю работу *TreeSet* ** , как он использует метод *TreeMap's put* для хранения элементов:

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

Переменная *m* ссылается на внутреннюю основу *TreeMap* (обратите внимание, что *TreeMap* реализует *NavigateableMap*):

```java
private transient NavigableMap<E, Object> m;
```

Следовательно, *TreeSet* внутренне зависит от вспомогательного *NavigableMap* , который инициализируется с экземпляром *TreeMap* при создании экземпляра *TreeSet* :

```java
public TreeSet() {
    this(new TreeMap<E,Object>());
}
```

Подробнее об этом можно узнать по ссылке:/java-treemap[эта статья].

4. *TreeSet содержит ()*

- Метод *contains ()* используется для проверки наличия данного элемента в данном *TreeSet* . ** Если элемент найден, он возвращает true, в противном случае *false*.

Давайте посмотрим на *contains ()* в действии:

```java
@Test
public void whenCheckingForElement__shouldSearchForElement() {
    Set<String> treeSetContains = new TreeSet<>();
    treeSetContains.add("String Added");

    assertTrue(treeSetContains.contains("String Added"));
}
```

5. *TreeSet remove ()*

- Метод *remove ()* используется для удаления указанного элемента из набора, если он присутствует. **

Если набор содержит указанный элемент, этот метод возвращает *true*.

Давайте посмотрим на это в действии:

```java
@Test
public void whenRemovingElement__shouldRemoveElement() {
    Set<String> removeFromTreeSet = new TreeSet<>();
    removeFromTreeSet.add("String Added");

    assertTrue(removeFromTreeSet.remove("String Added"));
}
```

6. *TreeSet clear ()*

Если мы хотим удалить все элементы из набора, мы можем использовать метод *clear ()* :

```java
@Test
public void whenClearingTreeSet__shouldClearTreeSet() {
    Set<String> clearTreeSet = new TreeSet<>();
    clearTreeSet.add("String Added");
    clearTreeSet.clear();

    assertTrue(clearTreeSet.isEmpty());
}
```

7. *TreeSet size ()*

Метод *size ()* используется для определения количества элементов, присутствующих в *TreeSet* . Это один из фундаментальных методов в API:

```java
@Test
public void whenCheckingTheSizeOfTreeSet__shouldReturnThesize() {
    Set<String> treeSetSize = new TreeSet<>();
    treeSetSize.add("String Added");

    assertEquals(1, treeSetSize.size());
}
```

8. *TreeSet isEmpty ()*

Метод *isEmpty ()* можно использовать для определения, является ли данный экземпляр *TreeSet* пустым или нет:

```java
@Test
public void whenCheckingForEmptyTreeSet__shouldCheckForEmpty() {
    Set<String> emptyTreeSet = new TreeSet<>();

    assertTrue(emptyTreeSet.isEmpty());
}
```

9. *TreeSet iterator ()*

Метод *iterator ()* возвращает итератор, повторяющийся в возрастающем порядке по элементам *Set*. **Эти итераторы работают быстро** .

Мы можем наблюдать восходящий порядок итераций здесь:

```java
@Test
public void whenIteratingTreeSet__shouldIterateTreeSetInAscendingOrder() {
    Set<String> treeSet = new TreeSet<>();
    treeSet.add("First");
    treeSet.add("Second");
    treeSet.add("Third");
    Iterator<String> itr = treeSet.iterator();
    while (itr.hasNext()) {
        System.out.println(itr.next());
    }
}
```

Кроме того, *TreeSet* позволяет нам перебирать *Set* в порядке убывания.

Давайте посмотрим, что в действии:

```java
@Test
public void whenIteratingTreeSet__shouldIterateTreeSetInDescendingOrder() {
    TreeSet<String> treeSet = new TreeSet<>();
    treeSet.add("First");
    treeSet.add("Second");
    treeSet.add("Third");
    Iterator<String> itr = treeSet.descendingIterator();
    while (itr.hasNext()) {
        System.out.println(itr.next());
    }
}
```

- *Iterator* генерирует исключение *_ConcurrentModificationException, если набор модифицируется в любое время после создания итератора любым способом, кроме как через метод remove () _* итератора. **

Давайте создадим тест для этого:

```java
@Test(expected = ConcurrentModificationException.class)
public void whenModifyingTreeSetWhileIterating__shouldThrowException() {
    Set<String> treeSet = new TreeSet<>();
    treeSet.add("First");
    treeSet.add("Second");
    treeSet.add("Third");
    Iterator<String> itr = treeSet.iterator();
    while (itr.hasNext()) {
        itr.next();
        treeSet.remove("Second");
    }
}
```

В качестве альтернативы, если бы мы использовали метод удаления итератора, мы бы не столкнулись с исключением:

```java
@Test
public void whenRemovingElementUsingIterator__shouldRemoveElement() {

    Set<String> treeSet = new TreeSet<>();
    treeSet.add("First");
    treeSet.add("Second");
    treeSet.add("Third");
    Iterator<String> itr = treeSet.iterator();
    while (itr.hasNext()) {
        String element = itr.next();
        if (element.equals("Second"))
           itr.remove();
    }

    assertEquals(2, treeSet.size());
}
```

- Нет никаких гарантий относительно отказоустойчивого поведения итератора, поскольку невозможно сделать какие-либо жесткие гарантии при наличии несинхронизированной параллельной модификации. **

Подробнее об этом можно узнать по ссылке:/java-fail-safe-vs-fail-fast-iterator[здесь].

10. *TreeSet first ()*

Этот метод возвращает первый элемент из *TreeSet* , если он не пустой. В противном случае он генерирует исключение *NoSuchElementException* .

Давайте посмотрим на пример:

```java
@Test
public void whenCheckingFirstElement__shouldReturnFirstElement() {
    TreeSet<String> treeSet = new TreeSet<>();
    treeSet.add("First");

    assertEquals("First", treeSet.first());
}
```

11. *TreeSet last ()*

Аналогично приведенному выше примеру, этот метод вернет последний элемент, если набор не пустой:

```java
@Test
public void whenCheckingLastElement__shouldReturnLastElement() {
    TreeSet<String> treeSet = new TreeSet<>();
    treeSet.add("First");
    treeSet.add("Last");

    assertEquals("Last", treeSet.last());
}
```

12. *TreeSet subSet ()*

Этот метод возвращает элементы в диапазоне от *fromElement* до *toElement*. Обратите внимание, что *fromElement* является включающим, а *toElement* - исключительным

```java
@Test
public void whenUsingSubSet__shouldReturnSubSetElements() {
    SortedSet<Integer> treeSet = new TreeSet<>();
    treeSet.add(1);
    treeSet.add(2);
    treeSet.add(3);
    treeSet.add(4);
    treeSet.add(5);
    treeSet.add(6);

    Set<Integer> expectedSet = new TreeSet<>();
    expectedSet.add(2);
    expectedSet.add(3);
    expectedSet.add(4);
    expectedSet.add(5);

    Set<Integer> subSet = treeSet.subSet(2, 6);

    assertEquals(expectedSet, subSet);
}
```

13. *TreeSet headSet ()*

Этот метод вернет элементы *TreeSet* , которые меньше указанного элемента:

```java
@Test
public void whenUsingHeadSet__shouldReturnHeadSetElements() {
    SortedSet<Integer> treeSet = new TreeSet<>();
    treeSet.add(1);
    treeSet.add(2);
    treeSet.add(3);
    treeSet.add(4);
    treeSet.add(5);
    treeSet.add(6);

    Set<Integer> subSet = treeSet.headSet(6);

    assertEquals(subSet, treeSet.subSet(1, 6));
}
```

14. *TreeSet tailSet ()*

Этот метод вернет элементы *TreeSet* , которые больше или равны указанному элементу:

```java
@Test
public void whenUsingTailSet__shouldReturnTailSetElements() {
    NavigableSet<Integer> treeSet = new TreeSet<>();
    treeSet.add(1);
    treeSet.add(2);
    treeSet.add(3);
    treeSet.add(4);
    treeSet.add(5);
    treeSet.add(6);

    Set<Integer> subSet = treeSet.tailSet(3);

    assertEquals(subSet, treeSet.subSet(3, true, 6, true));
}
```

15. Хранение *Null* элементов

- До Java 7 можно было добавлять элементы *null* в пустой TreeSet. **

Однако это считалось ошибкой. Поэтому ***TreeSet* больше не поддерживает добавление *null* .**

Когда мы добавляем элементы в *TreeSet,* элементы сортируются в соответствии с их естественным порядком или в соответствии с указаниями *comparator*. Следовательно, добавление *null,* по сравнению с существующими элементами, приводит к *NullPointerException* , поскольку *null* нельзя сравнивать с любым значением :

```java
@Test(expected = NullPointerException.class)
public void whenAddingNullToNonEmptyTreeSet__shouldThrowException() {
    Set<String> treeSet = new TreeSet<>();
    treeSet.add("First");
    treeSet.add(null);
}
```

Элементы, вставленные в *TreeSet* , должны либо реализовывать интерфейс *Comparable* , либо, по крайней мере, быть приняты указанным компаратором. **Все такие элементы должны быть взаимно сопоставимы, *i.e. E1.compareTo (e2)* или *comparator.compare (e1, e2)* не должны вызывать *ClassCastException* .**

Давайте посмотрим на пример:

```java
class Element {
    private Integer id;

   //Other methods...
}

Comparator<Element> comparator = (ele1, ele2) -> {
    return ele1.getId().compareTo(ele2.getId());
};

@Test
public void whenUsingComparator__shouldSortAndInsertElements() {
    Set<Element> treeSet = new TreeSet<>(comparator);
    Element ele1 = new Element();
    ele1.setId(100);
    Element ele2 = new Element();
    ele2.setId(200);

    treeSet.add(ele1);
    treeSet.add(ele2);

    System.out.println(treeSet);
}
```

16. Производительность *TreeSet*

По сравнению с *HashSet* производительность *TreeSet* находится на нижней стороне. Такие операции, как *add* , *remove* и *search* , занимают *O (log n)* время, в то время как такие операции, как печать *n* элементов в отсортированном порядке, требуют *O (n)* время.

TreeSet должен быть нашим основным выбором, если мы хотим, чтобы наши записи сортировались как *TreeSet* , к которым можно обращаться и проходить в порядке возрастания или убывания, а выполнение восходящих операций и представлений, вероятно, будет быстрее, чем у нисходящих.

Принцип локальности - это термин, обозначающий явление, при котором часто обращаются к одним и тем же значениям или связанным местам хранения в зависимости от схемы доступа к памяти.

Когда мы говорим, местность:

- Подобные данные часто доступны из приложения с аналогичными

частота ** Если две записи находятся рядом с указанием порядка, *TreeSet* размещает их

рядом друг с другом в структуре данных, и, следовательно, в памяти

TreeSet является структурой данных с большей локальностью, поэтому мы можем сделать вывод в соответствии с Принципом локальности, что мы должны отдавать предпочтение *TreeSet* , если у нас мало памяти и если мы хотим получить доступ к элементам, которые находятся относительно близко друг другу в соответствии с их естественным порядком.

В случае, если данные должны быть прочитаны с жесткого диска (который имеет большую задержку, чем данные, прочитанные из кеша или памяти), тогда предпочитайте *TreeSet* , поскольку он имеет большую локальность

17. Заключение

В этой статье мы сосредоточимся на понимании того, как использовать стандартную реализацию *TreeSet* в Java. Мы увидели его назначение и эффективность в отношении удобства использования, учитывая его способность избегать дублирования и сортировки элементов.

Как всегда, фрагменты кода можно найти over на GitHub .