Методические указания 5

Коллекции Часть 2

Интерфейсы Map и Set. Основные реализации и приемы использования, проход по элементам коллекции, сравнение и сортировка элементов коллекции.

Коллекции

Классы HashMap, LinkedHashMap, TreeMap

Класс HashMap<K, V> представляет собой хеш-таблицу для хранения пар ключ-значение (Key(K) - ключ, Value(V) - значение), и обеспечивает постоянное время выполнения методов get() и put() даже при большом количестве элементов в коллекции. Типы ключа и значения могут отличаться. Для того, чтобы понять, что такое ключ, и что такое значение, а также почему HashMap позволяет производить быстрый поиск значения по ключу, необходимо немного углубиться в структуру и логику работы HashMap.

Важно! Несмотря на то, что мы подробно рассматриваем логику работы HashMap, для работы с этим типом данных вам совершенно не обязательно все это помнить. Разбор внутренней структуры приведен для того, чтобы вы понимали почему HashMap обеспечивает быстрый поиск значения по ключу и откуда могут появиться проблемы с производительностью этой структуры данных.

У HashMap есть два основных параметра:

- **capacity** - емкость, или количество элементов(bucket) во внутренней таблице HashMap, по-умолчанию начальная емкость HashMap равна 16, и всегда равна степени 2, при попытке указать в конструкторе начальную емкость равную 28, она автоматически будет увеличена до 32;

- **loadFactor** (по-умолчанию равен 0.75, должен находиться в пределах от 0.0 до 1.0) - коэффициент, который показывает что при добавлении в HashMap количества элементов бОльшего чем capacity * loadFactor, емкость коллекции будет увеличена вдвое и произойдет рехеширование записей(что это такое будет рассказано далее).

Теперь посмотрим на структуру HashMap, которая схематично представлена на рисунке 4, для упрощения объяснения начальную емкость возьмем равной 1.

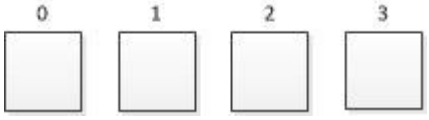

Рисунок 1 — Структура HashMap

Каждая ячейка внутренней таблицы HashMap хранит в себе список (в некоторых случаях может быть построено дерево) пар Map.Entry<K, V>. Показанный на рисунке HashMap ничем не заполнен. Для добавления элемента используется метод put(key, value), где в качестве первого аргумента передается ключ, а в качестве второго - значение. Давайте посмотрим что происходит при исполнении метода put, схема работы показана на рисунке 2.

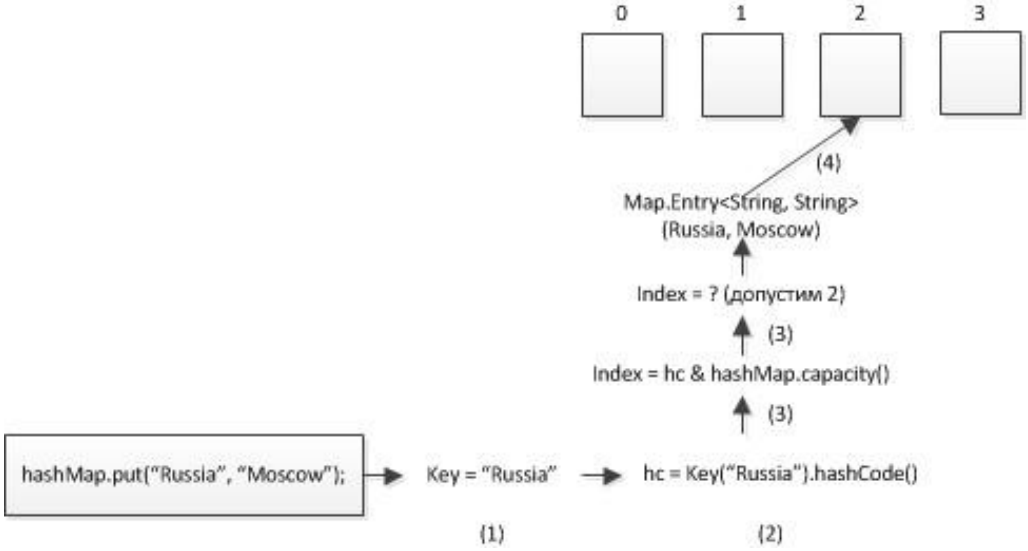

Рисунок 2 - Принцип работы метода put()

Допустим мы хотим добавить в HashMap пару страна-столица “Russia-Moscow”. На первом шаге (1), из метода put вытаскивается ключ “Russia” и по нему считается hashCode (2), при “совмещении” hashCode ключа и емкости hashMap на этапе (3) мы получаем индекс ячейки, в которую будет добавлена пара “Russia”-”Moscow” (4). ().

Заметка. Алгоритм совмещения hashCode и capacity не приводится, так как может меняться от версии к версии jdk **.**

Поскольку hashCode для объекта не уникален, и количество ячеек во внутренней таблице HashMap ограничено, может случиться так, что в одну ячейку упадет несколько разных Map.Entry<>, такая

ситуация называется коллизией. Пример показан на рисунке 3 (на рисунке показан лишь пример, на самом деле эти записи могут оказаться в разных ячейках, в зависимости от hashCode ключей).

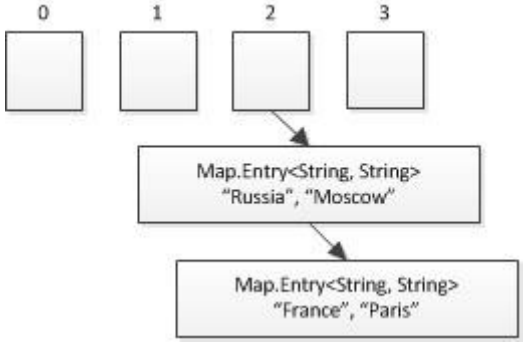

Рисунок 3 - Коллизия для пар “Russia”-”Moscow” и “France”-”Paris”

Как видите, разные записи складываются по порядку добавления в одну ячейку (ячейка может содержать в себе LinkedList<Map.Entry> или древовидную структуру).

При необходимости достать из HashMap значение по ключу происходит обратный процесс. При выполнении метода get(“France”), у указанного ключа находится hashCode, после чего он “совмещается” с емкостью HashMap и тем самым мы находим ячейку, в которой лежит искомая запись. Останется только пройтись по коллекции в этой ячейке и найти Map.Entry у которой ключ совпадает по equals() с искомым. Если бы мы хранили только значения без ключей, то невозможно было бы понять какое значение к какому ключу относится.

При создании HashMap можно использовать несколько перегруженных конструкторов:

HashMap()	Создает пустой HashMap с начальной емкостью 16
HashMap(Map<? extends K, ? extends V> m)	Создает HashMap, инициализируемый элементами из Map m
HashMap(int initialCapacity)	Создает HashMap, с указанной начальной емкостью
HashMap(int initialCapacity, float loadFactor)	Создает HashMap, с указанной начальной емкостью и коэффициент заполнения

Следует иметь в виду, что HashMap не гарантирует порядок расположения своих элементов, соответственно порядок их перебора может не соответствовать порядку их добавления. (Такая “хаотичность” связана со способом хранения записей). В следующем примере программы демонстрируется применение класса HashMap:

```
public  static  void main(String args[]) {
    HashMap<String, String> hm =  new HashMap<>();
    hm.put( "Russia" ,  "Moscow" );     hm.put( "France" ,
"Paris" );     hm.put( "Germany" ,  "Berlin" );     hm.put(
"Norway" ,  "Oslo" );      for (Map.Entry<String, String> o
: hm.entrySet()) {         System.out.println(o.getKey() +
": " + o.getValue());
    }
    hm.put( "Germany" ,  "Berlin2" );
    System.out.println( "New Germany Entry: " + hm.get( "Germany"
)); }

// Результат:
// Norway: Oslo
// France: Paris
// Germany: Berlin
// Russia: Moscow
// New Germany Entry: Berlin2
```

Выполнение данной программы начинается с создания HashMap<String, String> hm и добавления в него стран и столиц. Далее его содержимое с помощью цикла foreach выводится в консоль. Ключи и значения выводятся в результате вызова методов getKey() и getValue(), определенных в Map.Entry. Обратите особое внимание на порядок изменения записи Germany/Berlin. Метод put() автоматически заменяет ранее существовавшее значение на новое при совпадении ключей, то есть не может быть нескольких значений под одним ключом. Таким образом, после обновления записи Germany/Berlin на Germany/Berlin2 HashMap по-прежнему содержит только одну пару ключ/значение Germany/Berlin2.

Классы LinkedHashMap и TreeMap расширяют класс HashMap. LinkedHashMap сохраняет порядок добавления записей, а TreeMap хранит пары «ключ-значение» в отсортированном порядке (в порядке возрастания ключей).

Важно! Что нужно помнить при работе с HashMap:

- HashMap предоставляет возможность быстрого поиска значения по ключу;
- Для того, чтобы ваши собственные типы данных (классы) могли использоваться в качестве ключей HashMap, необходимо корректно реализовать методы hashCode() и equals();
- Метод put() используем для добавления пары ключ-значение, get() для получения значения по ключу;

Классы HashSet, LinkedHashSet, TreeSet

HashSet. Класс HashSet служит для создания коллекции, содержащей только уникальные элементы (особое внимание необходимо уделить словосочетанию “только уникальные элементы”) и основанной на использовании внутренней хеш-таблицы. Преимущество хеширования заключается в том, что оно обеспечивает постоянство времени выполнения методов add(), contains(), remove() и size(). В классе HashSet определены следующие конструкторы:

HashSet()	Создает пустой HashSet с начальной ёмкостью 16
HashSet(Collection<? extends E> collection)	Создает HashSet, инициализируемый элементами из заданной коллекции collection

HashSet(int initialCapacity)	Создает HashSet, имеющий указанную начальную емкость
HashSet(int initialCapacity, float loadFactor)	Создает HashSet, имеющий указанную начальную емкость и коэффициент заполнения

Назначение capacity и loadFactor такое же, как и в случае с HashMap. В классе HashSet не определяется никаких дополнительных методов, помимо тех, что предоставляют его суперклассы и интерфейсы. Следует также иметь в виду, что класс HashSet не гарантирует упорядоченности элементов, поскольку процесс хеширования сам по себе обычно не приводит к созданию отсортированных множеств. Ниже приведён пример, демонстрирующий применение класса HashSet.

```
public  static  void main(String
args[]) {     Set<String> set =  new
HashSet<>();      set.add( "Альфа" );
set.add( "Бета" );      set.add( "Альфа"
);      set.add( "Эта" );      set.add(
"Гамма" );      set.add( "Эпсилон" );
set.add( "Омега" );      set.add(
"Гамма" );      System.out.println(set);
}

// Результат:
// [Гамма, Эпсилон, Бета, Эта, Омега, Альфа]
```

Как видите, в коде объекты “Альфа” и “Гамма” были добавлены дважды, однако HashSet сохранил только по одному варианту этих объектов.

LinkedHashSet. Класс LinkedHashSet<E> расширяет класс HashSet, не добавляя никаких новых методов. У этого класса такие же конструкторы, как и у класса HashSet. Класс LinkedHashSet использует связный список для сохранения порядка добавления в него элементов. Следовательно, при переборе элементов они будут извлекаться в том порядке, в каком были добавлены. Пример:

```
public  static  void main (String args[]) {     Set<String> set =
new LinkedHashSet<>();
    set.add( "Бета" );      set.add( "Альфа" );      set.add( "Эта"
);      set.add( "Гамма" );      set.add( "Эпсилон" );      set.add(
"Омега" );      System.out.println(set);
}

// Результат:
// [Бета, Альфа, Эта, Гамма, Эпсилон, Омега]
```

TreeSet. Класс TreeSet создаёт коллекцию, где для хранения элементов применяет древовидная структура. Объекты сохраняются в отсортированном порядке по возрастанию. Время доступа и извлечения элементов достаточно мало, благодаря чему класс TreeSet оказывается отличным выбором для хранения больших объемов отсортированных данных.

В классе TreeSet определены следующие конструкторы:

- TreeSet ().
- TreeSet (Collection<? extends E> collection).
- TreeSet (Comparator<? super E> comparator).
- TreeSet (SortedSet<E> s).

В первой форме конструктора создаётся пустое древовидное множество. Во второй — древовидное множество, содержащее элементы заданной коллекции collection. В третьей — пустое древовидное множество, элементы которого будут отсортированы заданным компаратором. И, наконец, в четвёртой форме создаётся древовидное множество, содержащее элементы заданного отсортированного множества s. В приведённом ниже примере программы демонстрируется применение класса TreeSet.

```
public  static  void main(String args[])
{     Set<String> set =  new TreeSet<>();
    set.add( "C" );
set.add( "A" );
set.add( "B" );
set.add( "E" );
set.add( "F" );
set.add( "D" );
    System.out.println(set);
}

// Результат:
// [A, B, C, D, E, F]
```

Элементы такого множества автоматически располагаются в отсортированном порядке.

Итераторы

Итератор позволяет обойти все элементы коллекции. Для работы с итераторами служит интерфейс Iterator. Для получения объекта этого типа, необходимо вызвать метод iterator() у коллекции.

```
public  static  void main(String[] args) {
List<String> list =  new ArrayList<>();
   Iterator<String> iter = list.iterator();     }
```

Давайте рассмотрим три основных метода интерфейса Iterator: hasNext(), next(), remove().

- **hasNext()** проверяет наличие элементов в коллекции, которые мы еще не видели;
- **next()** переходит на следующий элемент коллекции и возвращает ссылку на него;
- **remove()** удаляет элемент, на который указывает итератор в настоящий момент.

Поставим следующую задачу: есть список строк, из которого необходимо удалить строки “A”. Ниже представлено решение этой задачи.

```
public  static  void main(String[] args) {
   List<String> list =  new  ArrayList<>(Arrays.asList( "A" ,  "B" ,  "C" ,  "C"
,  "A" ,  "A" , "B" ,  "C" ,  "B" ));
   Iterator<String> iter = list.iterator();
    while  (iter.hasNext()) {
String str = iter.next();
if (str.equals( "A" )) {
iter.remove();
       }
   }
   System.out.println(list);
}
// Результат: [B, C, C, B, C, B]
```

При работе с List мы можем использовать “расширенный” вариант итератора - ListIterator.

```
public  static  void main(String[] args) {
   List<String> list =  new  ArrayList<>(Arrays.asList( "A" ,  "B" ,  "C" ,  "C"
,  "A" ,  "A" , "B" ,  "C" ,  "B" ));
   ListIterator<String> iter = list.listIterator();
}
```

Этот интерфейс добавляет больше гибкости при работе с List.

- **hasPrevious()** - проверка есть ли элемент слева;
- **previous()** - переход на левый элемент и возврат ссылки на него;
- **nextIndex()** - получение индекса следующего элемента;
- **previousIndex()** - получение индекса предыдущего элемента;
- **add()** - добавить новый элемент на то место, на которое указывает итератор;
- **set()** - изменить элемент, на который указывает итератор.

Как видно из приведенного выше списка методов, при работе с листами мы можем не только обходить элементы и удалять их, но и: работать с индексами элементов, добавлять/изменять объекты в коллекции, двигаться не только вправо, но и влево по списку.

Интерфейс Comparable

При необходимости отсортировать коллекцию, или использовать упорядоченную, возникает вопрос - каким образом Java понимает как сортировать объекты? Рассмотрим этот вопрос на примере класса Cat.

```
public  class Cat {
private  String name;
private  int age;
     public  Cat(String name,  int
age) {          this .name = name;
this .age = age;
   }

    @Override
    public  String toString() {
return  "Cat [" + name +  "]" ;
   }
}
```

Давайте создадим список объектов типа Cat, и попробуем отсортировать его с помощью статического метода Collections.sort().

```
public  static  void main(String[] args) {
   List<Cat> cats =  new ArrayList<>(Arrays.asList(
            new Cat( "A" ,  5) ,  new Cat( "B" ,  2) ,  new
Cat( "C" ,  4)     ));
   System.out.println(cats);
   Collections.sort(cats);
}
```

При таком варианте написания кода мы получим ошибку на этапе компиляции. Java требует, чтобы класс Cat реализовал интерфейс Comparable.

```
public  class Cat  implements
Comparable {     private String name;
private  int age;

    public Cat(String name,  int
age) {         this .name = name;
this .age = age;
   }

    @Override
    public  int compareTo(Object o)
{        Cat another = (Cat)o;
if ( this .age > another.age) {
return  1;
       }         if ( this .age <
another.age) {             return -
1;
       }         return  0;
   }

    @Override     public String
toString() {         return  "Cat
[" + name +  "]" ;
   } }
```

В интерфейсе Comparable описан метод compareTo(Object o), который отвечает за сравнение объектов нашего класса. Если метод compareTo() вернет положительное число, значит текущий объект (this) больше o, если отрицательное - this меньше o, если вернул 0, значит объекты равны между собой. Указанную выше реализацию метода compareTo можно сократить до:

```
public  class Cat  implements
Comparable {     // ...     @Override
public  int compareTo(Object o) {
return  this .age - ((Cat)o).age;    }
    // ...
}
```

Объяснив Java что хотим сравнивать котов именно по возрасту, мы можем отсортировать список в порядке возрастания, и вывести его в консоль.

```
public  static  void main(String[] args) {
   List<Cat> cats =  new ArrayList<>(Arrays.asList(
           new Cat( "A" ,  5) ,  new Cat( "B" ,  2) ,  new
Cat( "C" ,  4)     ));
   System.out.println(cats);
Collections.sort(cats);
   System.out.println(cats);
}

// Результат:
// [Cat [A], Cat [B], Cat [C]]
// [Cat [B], Cat [C], Cat [A]]
```

Итак, интерфейс Comparable служит для описания способа сравнения объектов для их дальнейшего упорядочивания. Данный интерфейс указывает что объекты этого типа могут быть упорядочены.

Практическое задание

1. Создать массив с набором слов (10-20 слов, должны встречаться повторяющиеся). Найти и вывести список уникальных слов, из которых состоит массив (дубликаты не считаем). Посчитать, сколько раз встречается каждое слово.
2. Написать простой класс Телефонный Справочник, который хранит в себе список фамилий и телефонных номеров. В этот телефонный справочник с помощью метода add() можно добавлять записи, а с помощью метода get() искать номер телефона по фамилии. Следует учесть, что под одной фамилией может быть несколько телефонов (в случае однофамильцев), тогда при запросе такой фамилии должны выводиться все телефоны. *Желательно не добавлять лишний функционал (дополнительные поля (имя, отчество, адрес), взаимодействие с пользователем через консоль и т.д). Консоль использовать только для вывода результатов проверки телефонного справочника.*

Дополнительные материалы

1. Кей С. Хорстманн, Гари Корнелл Java. Библиотека профессионала. Том 1. Основы // Пер. с англ. — М.: Вильямс, 2014. — 864 с.
2. Брюс Эккель. Философия Java // 4-е изд.: Пер. с англ. — СПб.: Питер, 2016. — 1 168 с.
3. Г. Шилдт. Java 8. Полное руководство // 9-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 1 376 с.
4. Г. Шилдт. Java 8: Руководство для начинающих. // 6-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 720 с.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Г. Шилдт. Java 8. Полное руководство // 9-е изд.: Пер. с англ. — М.: Вильям с , 2015. — 1 376 с.