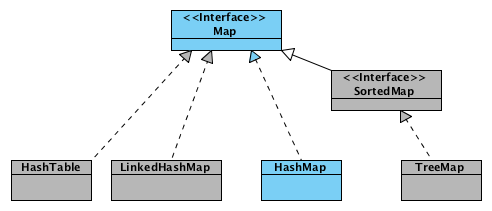
Структуры данных в картинках. HashMap

**6 min**

**1.1M**

[Java\*](https://habr.com/en/hub/java/)

Приветствую вас, хабрачитатели!  
  
Продолжаю попытки визуализировать структуры данных в Java. В предыдущих сериях мы уже ознакомились с [ArrayList](http://habrahabr.ru/blogs/java/128269/) и [LinkedList](http://habrahabr.ru/blogs/java/127864/), сегодня же рассмотрим HashMap.  
  
  
  
*HashMap — основан на хэш-таблицах, реализует интерфейс Map (что подразумевает хранение данных в виде пар ключ/значение). Ключи и значения могут быть любых типов, в том числе и null. Данная реализация не дает гарантий относительно порядка элементов с течением времени. Разрешение коллизий осуществляется с помощью*[*метода цепочек*](http://ru.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88-%D1%82%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D0%B0#.D0.9C.D0.B5.D1.82.D0.BE.D0.B4_.D1.86.D0.B5.D0.BF.D0.BE.D1.87.D0.B5.D0.BA)*.*

Создание объекта

Map<String, String> hashmap = **new** HashMap<String, String>();

Footprint{Objects=2, References=20, Primitives=[int x 3, float]}  
Object size: 120 bytes  
  
Новоявленный объект hashmap, содержит ряд свойств:

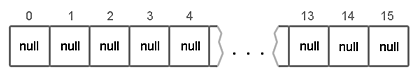
* **table** — Массив типа **Entry[]**, который является хранилищем ссылок на списки (цепочки) значений;
* **loadFactor** — Коэффициент загрузки. Значение по умолчанию 0.75 является хорошим компромиссом между временем доступа и объемом хранимых данных;
* **threshold** — Предельное количество элементов, при достижении которого, размер хэш-таблицы увеличивается вдвое. Рассчитывается по формуле **(capacity \* loadFactor)**;
* **size** — Количество элементов HashMap-а;

В конструкторе, выполняется проверка валидности переданных параметров и установка значений в соответствующие свойства класса. Словом, ничего необычного.

// Инициализация хранилища в конструкторе

// capacity - по умолчанию имеет значение 16

table = **new** Entry[capacity];

  
Вы можете указать свои емкость и коэффициент загрузки, используя конструкторы **HashMap(capacity)** и **HashMap(capacity, loadFactor)**. Максимальная емкость, которую вы сможете установить, равна половине максимального значения **int** (1073741824).

Добавление элементов

hashmap.put("0", "zero");

Footprint{Objects=7, References=25, Primitives=[int x 10, char x 5, float]}  
Object size: 232 bytes  
  
При добавлении элемента, последовательность шагов следующая:

1. Сначала ключ проверяется на равенство null. Если это проверка вернула true, будет вызван метод **putForNullKey(value)** (вариант с добавлением null-ключа рассмотрим чуть [позже](https://habr.com/en/articles/128017/#putForNullKey)).
2. Далее генерируется хэш на основе ключа. Для генерации используется метод **hash(hashCode)**, в который передается **key.hashCode()**.
3. **static** **int** **hash**(**int** h)
4. {
5. h ^= (h >>> 20) ^ (h >>> 12);
6. **return** h ^ (h >>> 7) ^ (h >>> 4);
7. }

Комментарий из исходников объясняет, каких результатов стоит ожидать — *метод****hash(key)****гарантирует что полученные хэш-коды, будут иметь только ограниченное количество коллизий (примерно 8, при дефолтном значении коэффициента загрузки).*  
  
В моем случае, для ключа со значением **''0''** метод **hashCode()** вернул значение 48, в итоге:

h ^ (h >>> 20) ^ (h >>> 12) = 48

h ^ (h >>> 7) ^ (h >>> 4) = 51

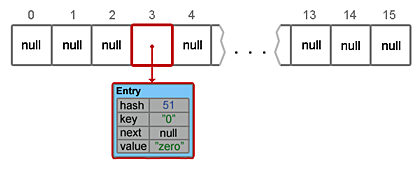
1. С помощью метода **indexFor(hash, tableLength)**, определяется позиция в массиве, куда будет помещен элемент.
2. **static** **int** **indexFor**(**int** h, **int** length)
3. {
4. **return** h & (length - 1);
5. }

При значении хэша **51** и размере таблице **16**, мы получаем индекс в массиве:

h & (length - 1) = 3

1. Теперь, зная индекс в массиве, мы получаем список (цепочку) элементов, привязанных к этой ячейке. Хэш и ключ нового элемента поочередно сравниваются с хэшами и ключами элементов из списка и, при совпадении этих параметров, значение элемента перезаписывается.
2. **if** (e.hash == hash && (e.key == key || key.equals(e.key)))
3. {
4. V oldValue = e.value;
5. e.value = value;
7. **return** oldValue;
8. }

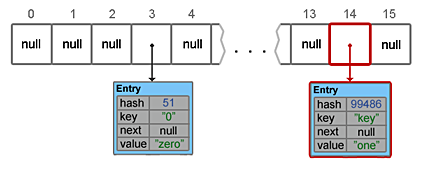
1. Если же предыдущий шаг не выявил совпадений, будет вызван метод **addEntry(hash, key, value, index)** для добавления нового элемента.
2. **void** **addEntry**(**int** hash, K key, V value, **int** index)
3. {
4. Entry<K, V> e = table[index];
5. table[index] = **new** Entry<K, V>(hash, key, value, e);
6. ...
7. }



Для того чтобы продемонстрировать, как заполняется HashMap, добавим еще несколько элементов.

hashmap.put("key", "one");

Footprint{Objects=12, References=30, Primitives=[int x 17, char x 11, float]}  
Object size: 352 bytes

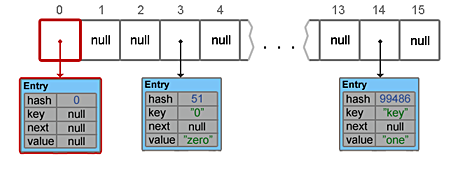
1. Пропускается, ключ не равен null
2. **''key''.hashCode()** = 106079
3. h ^ (h >>> 20) ^ (h >>> 12) = 106054
4. h ^ (h >>> 7) ^ (h >>> 4) = 99486
5. Определение позиции в массиве
6. h & (length - 1) = 14
7. Подобные элементы не найдены
8. Добавление элемента  
     
   

hashmap.put(**null**, **null**);

Footprint{Objects=13, References=33, Primitives=[int x 18, char x 11, float]}  
Object size: 376 bytes  
  
Как было сказано выше, если при добавлении элемента в качестве ключа был передан null, действия будут отличаться. Будет вызван метод **putForNullKey(value)**, внутри которого нет вызова методов **hash()** и **indexFor()** (потому как все элементы с null-ключами всегда помещаются в **table[0]**), но есть такие действия:

1. Все элементы цепочки, привязанные к **table[0]**, поочередно просматриваются в поисках элемента с ключом null. Если такой элемент в цепочке существует, его значение перезаписывается.
2. Если элемент с ключом null не был найден, будет вызван уже знакомый метод **addEntry()**.

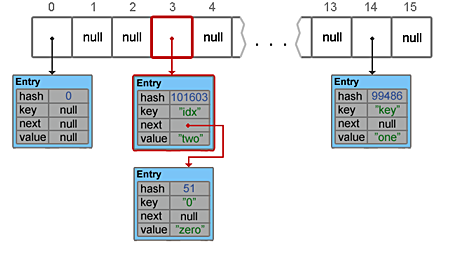
addEntry(0, **null**, value, 0);



hashmap.put("idx", "two");

Footprint{Objects=18, References=38, Primitives=[int x 25, char x 17, float]}  
Object size: 496 bytes  
  
Теперь рассмотрим случай, когда при добавлении элемента возникает коллизия.

1. Пропускается, ключ не равен null
2. **''idx''.hashCode()** = 104125
3. h ^ (h >>> 20) ^ (h >>> 12) = 104100
4. h ^ (h >>> 7) ^ (h >>> 4) = 101603
5. Определение позиции в массиве
6. h & (length - 1) = 3
7. Подобные элементы не найдены
8. Добавление элемента
9. // В table[3] уже хранится цепочка состоящая из элемента ["0", "zero"]
10. Entry<K, V> e = table[index];
11. // Новый элемент добавляется в начало цепочки
12. table[index] = **new** Entry<K, V>(hash, key, value, e);



Resize и Transfer

Когда массив **table[]** заполняется до предельного значения, его размер увеличивается вдвое и происходит перераспределение элементов. Как вы сами можете убедиться, ничего сложного в методах **resize(capacity)** и **transfer(newTable)** нет.

**void** **resize**(**int** newCapacity)

{

**if** (table.length == MAXIMUM\_CAPACITY)

{

threshold = Integer.MAX\_VALUE;

**return**;

}

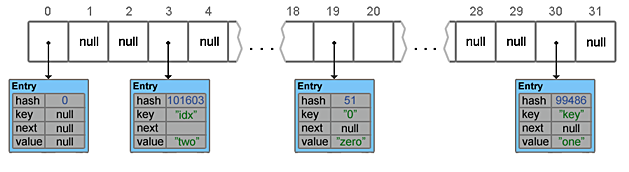
Entry[] newTable = **new** Entry[newCapacity];

transfer(newTable);

table = newTable;

threshold = (**int**)(newCapacity \* loadFactor);

}

Метод **transfer()** перебирает все элементы текущего хранилища, пересчитывает их индексы (с учетом нового размера) и перераспределяет элементы по новому массиву.  
  
Если в исходный **hashmap** добавить, скажем, еще 15 элементов, то в результате размер будет увеличен и распределение элементов изменится.  
  


Удаление элементов

У HashMap есть такая же «проблема» как и у ArrayList — при удалении элементов размер массива **table[]** не уменьшается. И если в ArrayList предусмотрен метод **trimToSize()**, то в HashMap таких методов нет (хотя, как сказал один мой коллега — "*А может оно и не надо?*").  
  
Небольшой тест, для демонстрации того что написано выше. Исходный объект занимает 496 байт. Добавим, например, 150 элементов.  
  
Footprint{Objects=768, References=1028, Primitives=[int x 1075, char x 2201, float]}  
Object size: 21064 bytes  
  
Теперь удалим те же 150 элементов, и снова замерим.  
  
Footprint{Objects=18, References=278, Primitives=[int x 25, char x 17, float]}  
Object size: 1456 bytes  
  
Как видно, размер даже близко не вернулся к исходному. Если есть желание/потребность исправить ситуацию, можно, например, воспользоваться конструктором **HashMap(Map)**.

hashmap = **new** HashMap<String, String>(hashmap);

Footprint{Objects=18, References=38, Primitives=[int x 25, char x 17, float]}  
Object size: 496 bytes

Итераторы

HashMap имеет встроенные итераторы, такие, что вы можете получить список всех ключей **keySet()**, всех значений **values()** или же все пары ключ/значение **entrySet()**. Ниже представлены некоторые варианты для перебора элементов:

// 1.

**for** (Map.Entry<String, String> entry: hashmap.entrySet())

System.out.println(entry.getKey() + " = " + entry.getValue());

// 2.

**for** (String key: hashmap.keySet())

System.out.println(hashmap.get(key));

// 3.

Iterator<Map.Entry<String, String>> itr = hashmap.entrySet().iterator();

**while** (itr.hasNext())

System.out.println(itr.next());

Стоит помнить, что если в ходе работы итератора HashMap был изменен (без использования собственным методов итератора), то результат перебора элементов будет непредсказуемым.

Итоги

— Добавление элемента выполняется за время O(1), потому как новые элементы вставляются в начало цепочки;  
— Операции получения и удаления элемента могут выполняться за время O(1), если хэш-функция равномерно распределяет элементы и отсутствуют коллизии. Среднее же время работы будет Θ(1 + α), где α — коэффициент загрузки. В самом худшем случае, время выполнения может составить Θ(n) (все элементы в одной цепочке);  
— Ключи и значения могут быть любых типов, в том числе и null. Для хранения примитивных типов используются соответствующие классы-оберки;  
— Не синхронизирован.