Внутренняя работа HashMap в Java

*[примечание от автора перевода] Перевод был выполнен для собственных нужд, но если кому -то это окажется полезным, значит мир стал хоть немного, но лучше! Оригинальная статья — [Internal Working of HashMap in Java](https://www.geeksforgeeks.org/internal-working-of-hashmap-java/)*

В этой статье мы увидим, как изнутри работают методы get и put в коллекции HashMap. Какие операции выполняются. Как происходит хеширование. Как значение извлекается по ключу. Как хранятся пары ключ-значение.

Как и в [предыдущей статье](https://www.geeksforgeeks.org/java-util-hashmap-in-java), HashMap содержит массив Node и Node может представлять класс, содержащий следующие объекты:

1. int — хэш
2. K — ключ
3. V — значение
4. Node — ссылка на следующий элемент, если индексы (вычисленные по хешам ключей) совпадают

Теперь мы увидим, как все это работает. Для начала мы рассмотрим процесс хеширования.

Хэширование

Хэширование -это процесс преобразования объекта в целочисленную форму, выполняется с помощью метода hashCode(). Очень важно правильно реализовать метод hashCode() для обеспечения лучшей производительности класса HashMap.

Здесь я использую свой собственный класс Key и таким образом могу переопределить метод hashCode() для демонстрации различных сценариев. Мой класс Key:

*// специальный класс Key для переопределени методов hashCode()*

*// и equals()*

class Key {

String key;

Key(String key) {

this.key = key;

}

@Override public int hashCode() {

return (int)key.charAt(0);

}

@Override public boolean equals(Object obj) {

return key.equals((String)obj);

}

}

Здесь переопределенный метод hashCode() возвращает ASCII код первого символа строки. Таким образом, если первые символы строки одинаковые, то и хэш коды будут одинаковыми. Не стоит использовать подобную логику в своих программах.

Этот код создан исключительно для демонстрации. Поскольку HashCode допускает ключ типа null, хэш код null всегда будет равен 0.

Метод hashCode()

Метод hashCode() используется для получения хэш кода объекта. Метод hashCode() класса Object возвращает ссылку памяти объекта в целочисленной форме ([идентификационный хеш (identity hash code)](https://habr.com/company/mailru/blog/321306/)). Сигнатура метода public native hashCode(). Это говорит о том, что метод реализован как нативный, поскольку в java нет какого -то метода позволяющего получить ссылку на объект. Допускается определять собственную реализацию метода hashCode(). В классе HashMap метод hashCode() используется для вычисления корзины (bucket) и следовательно вычисления индекса.

Метод equals()

Метод equals используется для проверки двух объектов на равенство. Метод реализованн в классе Object. Вы можете переопределить его в своем собственном классе. В классе HashMap метод equals() используется для проверки равенства ключей. В случае, если ключи равны, метод equals() возвращает true, иначе false.

Корзины (Buckets)

Bucket -это единственный элемент массива HashMap. Он используется для хранения узлов (Nodes). Два или более узла могут иметь один и тот -же bucket. В этом случае для связи узлов используется структура данных [связанный список](https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D1%8F%D0%B7%D0%BD%D1%8B%D0%B9_%D1%81%D0%BF%D0%B8%D1%81%D0%BE%D0%BA). Bucket -ы различаются по ёмкости (свойство capacity). Отношение между bucket и capacity выглядит следующим образом:

capacity = number of buckets \* load factor

Один bucket может иметь более, чем один узел, это зависит от реализации метода hashCode(). Чем лучше реализованн ваш метод hashCode(), тем лучше будут использоваться ваши bucket -ы.

Вычисление индекса в HashMap

Хэш код ключа может быть достаточно большим для создания массива. Сгенерированный хэш код может быть в диапазоне целочисленного типа и если мы создадим массив такого размера, то легко получим исключение outOfMemoryException. Потому мы генерируем индекс для минимизации размера массива. По сути для вычисления индекса выполняется следующая операция:

index = hashCode(key) & (n-1).

где n равна числу bucket или значению длины массива. В нашем примере я рассматриваю n, как значение по умолчанию равное 16.

* ***изначально пустой hashMap:*** здесь размер hashmap равен 16:

HashMap map = new HashMap();

HashMap:



* ***вставка пар Ключ — Значение:*** добавить одну пару ключ — значение в конец HashMap

map.put(new Key("vishal"), 20);

Шаги:

1. Вычисляем значение ключа {"vishal"} (т.е. хэш для этого значения). Оно будет сгенерированно, как 118.
2. По Хешу вычисляем индекс с помощью метода index, который будет равен 6.
3. Создаем объект node.

{

int hash = 118

*// {"vishal"} не строка, а*

*// объект класса Key*

Key key = {"vishal"}

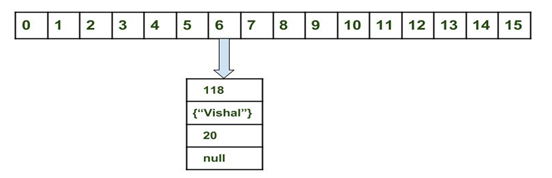
Integer value = 20

Node next = null

}

1. Поместить объект в позицию с индексом 6, если место свободно.

Теперь HashMap выглядит примерно так:



* ***добавление другой пары ключ — значение:*** теперь добавим другую пару

map.put(new Key("sachin"), 30);

Шаги:

1. Вычислить значение ключа {"sachin"} (т.е. хэш для этого значения). Оно будет сгенерированно, как 115.
2. Вычислить индекс с помощью метода index, который будет равен 3.
3. Создать объект node.

{

int hash = 115

Key key = {"sachin"}

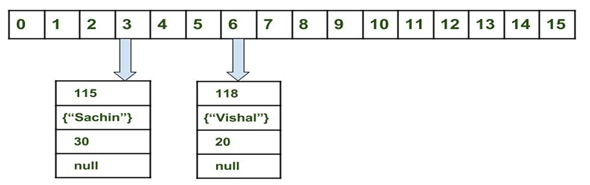
Integer value = 30

Node next = null

}

1. Поместить объект в позицию с индексом 3, если место свободно.

Теперь HashMap выглядит примерно так:



* ***в случае возникновения коллизий:*** теперь добавим другую пару

map.put(new Key("vaibhav"), 40);

Шаги:

1. Вычислить значение ключа {"vaibhav"}. Оно будет сгенерированно, как 118.
2. Вычислить индекс с помощью метода index, который будет равен 6.
3. Создать объект node.

{

int hash = 118

Key key = {"vaibhav"}

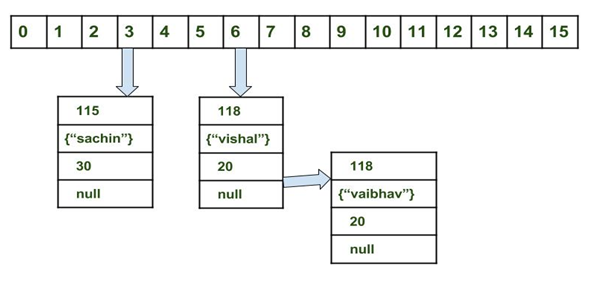
Integer value = 20

Node next = null

}

1. Поместить объект в позицию с индексом 6, если место свободно.
2. В данном случае в позиции с индексом 6 уже существует другой объект, этот случай называется коллизией.
3. В таком случае проверяем с помощью методов hashCode() и equals(), что оба ключа одинаковы.
4. Если ключи*(не хеши ключей, а именно значения самих ключей!)* одинаковы, заменить текущее значение новым.
5. Иначе связать новый и старый объекты с помощью структуры данных "связанный список", указав ссылку на следующий объект в текущем и сохранить оба под индексом 6.

Теперь HashMap выглядит примерно так:



*[примечание от автора перевода] Изображение взято из оригинальной статьи и изначально содержит ошибку. Ссылка на следующий объект в объекте vishal с индексом 6 не равна null, в ней содержится указатель на объект vaibhav.*

* ***получаем значение по ключу sachin:***

map.get(new Key("sachin"));

Шаги:

1. Вычислить хэш код объекта {“sachin”}. Он был сгенерирован, как 115.
2. Вычислить индекс с помощью метода index, который будет равен 3.
3. Перейти по индексу 3 и сравнить ключ первого элемента с имеющемся значением. Если они равны -вернуть значение, иначе выполнить проверку для следующего элемента, если он существует.
4. В нашем случае элемент найден и возвращаемое значение равно 30.

* ***получаем значение по ключу vaibahv:***

map.get(new Key("vaibhav"));

Шаги:

1. Вычислить хэш код объекта {"vaibhav"}. Он был сгенерирован, как 118.
2. Вычислить индекс с помощью метода index, который будет равен 6.
3. Перейти по индексу 6 и сравнить ключ первого элемента с имеющемся значением. Если они равны -вернуть значение, иначе выполнить проверку для следующего элемента, если он существует.
4. В данном случае он не найден и следующий объект node не равен null.
5. Если следующий объект node равен null, возвращаем null.
6. Если следующий объект node не равен null, переходим к нему и повторяем первые три шага до тех пор, пока элемент не будет найден или следующий объект node не будет равен null.

// Java программа для иллюстрации

// внутренней работы HashMap

import java.util.HashMap;

class Key {

String key;

Key(String key)

{

this.key = key;

}

@Override

public int hashCode()

{

int hash = (int)key.charAt(0);

System.out.println("hashCode for key: "

+ key + " = " + hash);

return hash;

}

@Override

public boolean equals(Object obj)

{

return key.equals(((Key)obj).key);

}

}

// Driver class

public class GFG {

public static void main(String[] args)

{

HashMap map = new HashMap();

map.put(new Key("vishal"), 20);

map.put(new Key("sachin"), 30);

map.put(new Key("vaibhav"), 40);

System.out.println();

System.out.println("Value for key sachin: " + map.get(new Key("sachin")));

System.out.println("Value for key vaibhav: " + map.get(new Key("vaibhav")));

}

}

Вывод:

hashCode for key: vishal = 118

hashCode for key: sachin = 115

hashCode for key: vaibhav = 118

hashCode for key: sachin = 115 Value for key sachin: 30

hashCode for key: vaibhav = 118 Value for key vaibhav: 40

Изменения в Java 8

Как мы уже знаем в случае возникновения коллизий объект node сохраняется в структуре данных "связанный список" и метод equals() используется для сравнения ключей. Это сравнения для поиска верного ключа в связанном списке -линейная операция и в худшем случае сложность равнa *O(n)*.

Для исправления этой проблемы в Java 8 после достижения определенного порога вместо связанных списков используются сбалансированные деревья. Это означает, что HashMap в начале сохраняет объекты в связанном списке, но после того, как колличество элементов в хэше достигает определенного порога происходит переход к сбалансированным деревьям. Что улучшает производительность в худшем случае с O(n) до O(log n).

Важный момент

1. Сложность операций get() и put() практически константна до тех пор, пока не будет проведенно повторное хэширование.
2. В случае коллизий, если индексы двух и более объектов node одинаковые, объекты node соединяются с помощью связанного списка, т.е. ссылка на второй объект node хранится в первом, на третий во втором и т.д.
3. Если данный ключ уже существует в HashMap, значение перезаписывается.
4. Хэш код null равен 0.
5. Когда объект получается по ключу происходят переходы по связанному списку до тех пор, пока объект не будет найден или ссылка на следующий объект не будет равна null.