Advanced Hashing Concept

<http://java-performance.info/changes-to-string-java-1-7-0_06/>

An original String implementation has 4 non static field: char[] value with string characters, int offset and int count with an index of the first character to use from value and a number of characters to use and int hash with a cached value of a String hash code. As you can see, in a very large number of cases a String will have offset = 0 and count = value.length. The only exception to this rule were the strings created via String.substring calls and all API calls using this method internally (like Pattern.split).

String.substring created a String, which shared an internal char[] value with an original String, which allowed you:

1. To save some memory by sharing character data
2. To run String.substring in a constant time ( O(1) )

At the same time such feature was a source of a possible memory leak: if you extract a tiny substring from an original huge string and discard that original string, you will still have a live reference to the underlying huge char[] value taken from an original String. The only way to avoid it was to call a new String( String ) constructor on such string – it made a copy of a required section of underlying char[], thus unlinking your shorter string from its longer “parent”.

From Java 1.7.0\_06 (as well as in current versions of Java 8 – Nov 13) offset and count fields were removed from a String. This means that you can’t share a part of an underlying char[] valueanymore. Now you can forget about a memory leak described above and never ever use new String(String) constructor anymore. As a drawback, you now have to remember thatString.substring has now a linear complexity instead of a constant one.

**Changes to hashing logic**

There is another change introduced to String class in the same update: a new hashing algorithm. Oracle suggests that a new algorithm gives a better distribution of hash codes, which should improve performance of several hash-based collections: HashMap, Hashtable, HashSet,LinkedHashMap, LinkedHashSet, WeakHashMap and ConcurrentHashMap. Unlike changes from the first part of this article, these changes are experimental and turned off by default.

As you may guess, these changes are only for String keys. If you want to turn them on, you’ll have to set a jdk.map.althashing.threshold system property to a non-negative value (it is equal to-1 by default). This value will be a collection size threshold, after which a new hashing method will be used. A small remark here: hashing method will be changed on rehashing only (when there is no more free space). So, if a collection was rehashed last time at size = 160 andjdk.map.althashing.threshold = 200, then a method will only be changed when your collection will grow to size of 320 (approximately).

String now has a hash32() method, which result is cached in int hash32 field. The biggest difference of this method is that the result of hash32() on the same string may be different on various JVM runs (actually, it will be different in most cases, because it uses a singleSystem.currentTimeMillis() and two System.nanoTime calls for seed initialization). As a result, iteration order on some of your collections will be different each time you run your program.

<http://java-performance.info/hashcode-method-performance-tuning/>

# **hashCode method performance tuning**

In this chapter we will discuss various implications of hashCode method implementation on application performance.

The main purpose of hashCode method is to allow an object to be a key in the hash map or a member of a hash set. In this case an object should also implement equals(Object) method, which is consistent with hashCode implementation:

* If a.equals(b) then a.hashCode() == b.hashCode()
* If hashCode() was called twice on the same object, it should return the same result provided that the object was not changed

#### **hashCode from performance point of view**

From the performance point of view, the main objective for your hashCode method implementation is to minimize the number of objects sharing the same hash code. All JDK hash based collections store their values in an array. Hash code is used to calculate an initial lookup position in this array. After that equals is used to compare given value with values stored in the internal array. So, if all values have distinct hash codes, this will minimize the possibility of hash collisions. On the other hand, if all values will have the same hash code, hash map (or set) will degrade into a list with operations on it having O(n2) complexity.

 JDK is using a method called[open addressing](http://en.wikipedia.org/wiki/Open_addressing), but there is another method called “chaining” – all key-value pairs with the same hash code are stored in a linked list.

**Open addressing**, or **closed hashing**, is a method of [collision resolution in hash tables](https://en.wikipedia.org/wiki/Hash_table#Collision_resolution). With this method a hash collision is resolved by **probing**, or searching through alternate locations in the array (the *probe sequence*) until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table.[[1]](https://en.wikipedia.org/wiki/Open_addressing#cite_note-tenenbaum90-1) Well known probe sequences include:

Let’s see the difference in hash code quality. We will compare a normal String with a string wrapper, which overrides hashCode method in order to return the same hash code for all objects.

***private******static******class*** *SlowString*

*{*

***public******final*** *String m\_str;*

***public*** *SlowString(* ***final*** *String str ) {*

***this****.m\_str = str;*

*}*

*@Override*

***public******int*** *hashCode() {*

***return*** *37;*

*}*

*@Override*

***public******boolean*** *equals(Object o) {*

***if*** *(****this*** *== o)* ***return******true****;*

***if*** *(o ==* ***null*** *|| getClass() != o.getClass())* ***return******false****;*

***final*** *SlowString that = ( SlowString ) o;*

***return*** *!(m\_str !=* ***null*** *? !m\_str.equals(that.m\_str) : that.m\_str !=* ***null****);*

*}*

*}*

Here is a testing method. It worth quoting because we will use it again later. It accepts a prepared list of objects (in order not to include these objects creation time in our test) and callsMap.put followed by Map.containsKey on each value in the list.

***private******static******void*** *testMapSpeed(* ***final*** *List lst,* ***final*** *String name )*

*{*

***final*** *Map<Object, Object> map =* ***new*** *HashMap<Object, Object>( lst.size() );*

***int*** *cnt = 0;*

***final******long*** *start = System.currentTimeMillis();*

***for*** *(* ***final*** *Object obj : lst )*

*{*

*map.put( obj, obj );*

***if*** *( map.containsKey( obj ) )*

*++cnt;*

*}*

***final******long*** *time = System.currentTimeMillis() - start;*

*System.out.println( "Time for "  + name + " is " + time / 1000.0 + " sec, cnt = " + cnt );*

*}*

Both String and SlowString objects are created in a loop as "ABCD" + i. It took 0.041 sec to process 100,000 String objects. As for the same number of SlowString objects, it took 82.5 seconds to process them.

As it turned out, String class has exceptional quality hashCode method. Let’s write another test. We will create a list of Strings. First half of them will be equal to "ABCdef\*&" + i, second half –"ABCdef\*&" + i + "ghi" (to ensure that changes in the middle of the string with a constant tail will not decrease hash code quality). We will create 1M, 5M, 10M and 20M strings and see how many of them will share hash codes and how many strings will share the same hash code. This is test output:

*Number of duplicate hashCodes for 1000000 strings = 0*

*Number of duplicate hashCodes for 5000000 strings = 196*

*Number of hashCode duplicates = 2 count = 196*

*Number of duplicate hashCodes for 10000000 strings = 1914*

*Number of hashCode duplicates = 2 count = 1914*

*Number of duplicate hashCodes for 20000000 strings = 17103*

*Number of hashCode duplicates = 2 count = 17103*

So, as you can see, only a very small number of strings is sharing the same hash code and it is very unlikely that one hash code will be shared by more than two strings (unless they are specially crafted, of course). Of course, your data may be different – just run a similar test on your typical keys.

#### **Autogenerated** hashCode **for** long **fields**

It is worth mentioning how hashCode method is generated for long datatype by most of IDEs. Here is a generated hashCode method for a class with 2 long fields:

***public******int*** *hashCode() {*

***int*** *result = (****int****) (val1 ^ (val1 >>> 32));*

*result = 31 \* result + (****int****) (val2 ^ (val2 >>> 32));*

***return*** *result;*

*}*

And here is the similar method generated for a class with 2 int fields:

***public******int*** *hashCode() {*

***int*** *result = val1;*

*result = 31 \* result + val2;*

***return*** *result;*

*}*

As you see, long is treated differently. Similar code is used in java.util.Arrays.hashCode(long a[]). Actually, you will get better hash code distribution if you will extract high and low 32 bits oflong and treat them as int while calculating a hash code. Here is an improved hashCode method for a class with 2 long fields (note that this method runs slower than an original method, but quality of new hash codes will allow hash collections to run faster even at the expense ofhashCode slowdown).

***public******int*** *hashCode() {*

***int*** *result = (****int****) val1;*

*result = 31 \* result + (****int****) (val1 >>> 32);*

*result = 31 \* result + (****int****) val2;*

***return*** *31 \* result + (****int****) (val2 >>> 32);*

*}*

Here are results of testMapSpeed method for processing 10M objects of all three kinds. They were initialized with the same values (all longs actually fit into int values).

| Two longs with original hashCode | Two longs with modified hashCode | Two ints |
| --- | --- | --- |
| 2.596 sec | 1.435 sec | 0.737 sec |

As you can see, hashCode update makes a difference. Not so big, but worth noticing for performance critical code.

#### **Use case: How to benefit from** String.hashCode **quality**

Let’s assume we have a map from string identifiers to some values. Map keys (string identifiers) are not stored anywhere else in memory (at most only some of them may be stored somewhere else at a time). We have already collected all map entries, for example, on the first pass of some two phase algorithm. On the second phase we will need to query map values by keys. We will query our map only using existing map keys.

How can we improve the map? As you have seen before, String.hashCode returns mostly distinct values. We can scan all keys, calculate hash codes of all keys and find not unique hash codes:

*Map<Integer, Integer> cnt =* ***new*** *HashMap<Integer, Integer>( max );*

***for*** *(* ***final*** *String s : dict.keySet() )*

*{*

***final******int*** *hash = s.hashCode();*

***final*** *Integer count = cnt.get( hash );*

***if*** *( count !=* ***null*** *)*

*cnt.put( hash, count + 1 );*

***else***

*cnt.put( hash, 1 );*

*}*

*//keep only not unique hash codes*

***final*** *Map<Integer, Integer> mult =* ***new*** *HashMap<Integer, Integer>( 100 );*

***for*** *(* ***final*** *Map.Entry<Integer, Integer> entry : cnt.entrySet() )*

*{*

***if*** *( entry.getValue() > 1 )*

*mult.put( entry.getKey(), entry.getValue() );*

*}*

Now we can create 2 maps out of the old one. Let’s assume for simplicity that old map values were just Objects. In this case, we will end up with Map<Integer, Object> and Map<String, Object>(for production code Trove TIntObjectHashMap is recommended instead of Map<Integer, Object>). First map will contain mapping from unique hash codes to values, second map – mapping from strings with not unique hash codes to values.

***final*** *Map<Integer, Object> unique =* ***new*** *HashMap<Integer, Object>( 1000 );*

***final*** *Map<String, Object> not\_unique =* ***new*** *HashMap<String, Object>( 1000 );*

*//dict - original map*

***for*** *(* ***final*** *Map.Entry<String, Object> entry : dict.entrySet() )*

*{*

***final******int*** *hashCode = entry.getKey().hashCode();*

***if*** *( mult.containsKey( hashCode ) )*

*not\_unique.put( entry.getKey(), entry.getValue() );*

***else***

*unique.put( hashCode, entry.getValue() );*

*}*

Now, in order to get a value, we need to query unique map first and not\_unique map if first query has not returned a valid result:

***public*** *Object get(* ***final*** *String key )*

*{*

***final******int*** *hashCode = key.hashCode();*

*Object value = m\_unique.get( hashCode );*

***if*** *( value ==* ***null*** *)*

*value = m\_not\_unique.get( key );*

***return*** *value;*

*}*

**Hashing function in Map in Java**

//**As of JDK 6**  
static int hash(int h) {

// This function ensures that hashCodes that differ only by  
// constant multiples at each bit position have a bounded  
// number of collisions (approximately 8 at default load factor).

h ^= (h >>> 20) ^ (h >>> 12);  
 return h ^ (h >>> 7) ^ (h >>> 4);  
}

//**As of JDK 7**  
final int hash(Object k) {

int h = hashSeed;  
if (0 != h && k instanceof String) {  
 return sun.misc.Hashing.stringHash32((String) k);  
}  
  
h ^= k.hashCode();  
// This function ensures that hashCodes that differ only by  
// constant multiples at each bit position have a bounded  
// number of collisions (approximately 8 at default load factor).  
  
h ^= (h >>> 20) ^ (h >>> 12);  
 return h ^ (h >>> 7) ^ (h >>> 4);  
}

<https://vaskoz.wordpress.com/2013/04/06/java-7-hashing-drastically-better-than-java-6/>

**Java 7 hashing drastically better than Java 6**

### How does Java 6 hashing work?

java.lang.String#hashCode() is calculated by iterating over each character in the string, and executing this function:

**h = 31\*h + val[i]**

Now let’s examine how the hash code is used by a popular data structure: HashMap. Looking at java.util.HashMap#put(K,V) runs HashMap#hash(int) on the key’s hashCode. That function is a supplemental hash that provides some protection against bad hash function.

**h ^= (h >>> 20) ^ (h >>> 12);**  
**return h ^ (h >>> 7) ^ (h >>> 4);**

This provides no randomization or protection from [**collision attacks**](http://en.wikipedia.org/wiki/Collision_attack). In Java 6, every JVM running everywhere follows these 2 mathematical functions. Some might think, Java hash code should always be the same whenever you call hashCode on the same string, but that’s only true within a single invocation of the JVM. There is no requirement that if 2 different JVM instances running should resolve to the same hashCode for the same string.

### Java 7 finally fixed hashing.

java.lang.String has add a static initializer for the single purpose of creating a**HASHING\_SEED**. This is brand new for Java 7. The “seed materials” used to randomize hash codes so that different JVMs don’t have predictable hash codes are: System.currentTimeMillis and System.nanoTime and others. But wait, if you look at java.lang.String#hashCode() in Java 7 source ***it looks identical to Java 6’s implementation!***So let’s checkout java.util.HashMap in Java 7, and we find that the hash() function there has changed. It takes an Object instead of an int, and if the Object is an instanceof String, then through **sun.misc.Hashing.stringHash32()**it actually called a new method insidejava.lang.String#hash32(). The hash32() function passes the character data plus the HASHING\_SEED computed above to **sun.misc.Hashing.murmur3\_32(HASHING\_SEED, value, 0, value.length)**

The [**Murmur3 hashing function**](http://en.wikipedia.org/wiki/MurmurHash) with the hashing seed data which is based on the time that java.lang.String’s static initializer is executed provides excellent protection against collision attacks.

public class Hash {

public static void main(String[] args) {

String str = "Java Programming Language";

System.out.println(str + " hashCode(): "

+ str.hashCode()

+ " new Java 7 hashCode used by collections: "

+ sun.misc.Hashing.stringHash32(str));

}

}

The above provide two different hashcode, str.hashCode() always gives constant value where as sun.misc.Hashing.stringHash32(str) gives random value.

# **MurmurHash**

<https://en.wikipedia.org/wiki/MurmurHash>

**MurmurHash** is a non-[cryptographic](https://en.wikipedia.org/wiki/Cryptographic_hash_function) [hash function](https://en.wikipedia.org/wiki/Hash_function) suitable for general hash-based lookup. The name comes from two basic operations, multiply (MU) and rotate (R), though the algorithm actually uses shift and xor instead of rotate.

# **Collision attack**

<https://en.wikipedia.org/wiki/Collision_attack>

In cryptography, a **collision attack** on a [cryptographic hash](https://en.wikipedia.org/wiki/Cryptographic_hash) tries to find two inputs producing the same hash value, i.e. a [hash collision](https://en.wikipedia.org/wiki/Hash_collision).

**Collision attack**

Find two different messages *m1* and *m2* such that *hash(m1)* = *hash(m2)*.

## Classical collision attack

Mathematically stated, a collision attack finds two different messages *m1* and *m2*, such that *hash(m1)* = *hash(m2)*. In a classical collision attack, the attacker has no control over the content of either message, but they are arbitrarily chosen by the algorithm.

## Attack scenarios[[edit](https://en.wikipedia.org/w/index.php?title=Collision_attack&action=edit&section=3)]

Many applications of crytographic hash functions do not rely on [collision resistance](https://en.wikipedia.org/wiki/Collision_resistance), thus collision attacks do not affect their security. For example, [HMACs](https://en.wikipedia.org/wiki/HMAC) are not vulnerable.[[9]](https://en.wikipedia.org/wiki/Collision_attack#cite_note-collision-qna-9) For the attack to be useful, the attacker must be in control of the input to the hash function.

### Digital signatures

Because [digital signature](https://en.wikipedia.org/wiki/Digital_signature) algorithms cannot sign a large amount of data efficiently, most implementations use a hash function to reduce ("compress") the amount of data that needs to be signed down to a constant size. Digital signature schemes are often vulnerable to hash collisions, unless using techniques like randomized hashing.[[10]](https://en.wikipedia.org/wiki/Collision_attack#cite_note-10)

Note that all [public key certificates](https://en.wikipedia.org/wiki/Public_key_certificate), like [SSL](https://en.wikipedia.org/wiki/Transport_Layer_Security) certificates, also rely on the security of digital signatures and are compromised by hash collisions.

The usual attack scenario goes like this:

1. Mallory creates two different documents A and B, that have an identical hash value (collision).
2. Mallory then **sends document A to Alice**, who agrees to what the document says, signs its hash and sends it back to Mallory.
3. Mallory copies the signature sent by Alice from document A to document B.
4. Then she **sends document B to Bob**, claiming that Alice signed the other document (document B). Because the digital signature matches the document hash, Bob's software is unable to detect the modification.