个人资料

雅然风懿

访问： 38430次
积分： 1069
等级： BLOG 4
排名： 千里之外

原创： 64篇
转载： 5篇
译文： 2篇
评论： 0条

文章搜索

文章分类

Vi文本编辑命令大全 (1)
C++ (2)
汉罗塔 (1)
十进制转二进制 (1)
收藏 (3)
android (37)
算法 (4)
Ubuntu (3)
Google (1)
网络技术 (2)
java (15)
unity3d (1)

文章存档

2015年12月 (2)
2015年11月 (4)
2015年10月 (1)
2015年08月 (1)
2015年01月 (1)

展开

【活动】Python创意编程活动开始啦！！！ CSDN日报20170424 ——《技术方向的选择》 程序员4月书讯：Angular来了！

Java7/Java8中HashMap解析

标签： java java 7 java8

2015-10-28 09:08 821人阅读 评论

分类： java (14)

目录(?) [+]

本文从性能、内存以及各种典型问题分析Java7到Java8中HashMap的改进：

原文地址：http://coding-geek.com/how-does-a-hashmap-work-in-Java/

HashMap内部存储过程：

HashMap类实现了Map<K,V>接口，主要方法包括：

- V put(K key,V value)
- V get(Object key)
- V remove(Object key)
- Boolean containsKey(Object key)

HashMap用一个内部类来添加数据：Entry<K,V>是一个key-value键值对，包括两个元素：

- 对另一个Entry的引用可以使得HashMap像单链表一样来存储数据；
- key的hash值，该hash值先保存在key中以免后面需要的时候每次都要重新计算。

Entry的实现如下：

```
[java] view plain copy print ?
01. <span style="font-size:18px;">static class Entry<K,V> implements Map.Entry<K,V> {
02.         final K key;
03.         V value;
04.         Entry<K,V> next;
05.         int hash;
06. …
07. }</span>
```

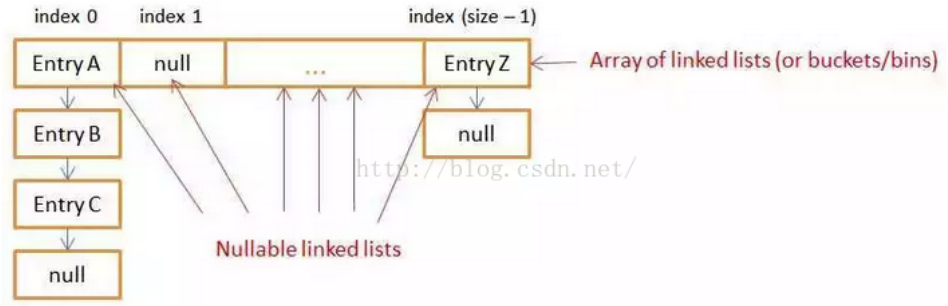
一个hashMap存储着多个单链表的数据，就像桶(buckets)或者箱子(bins)一样，所有的lists都在Entry<K,V>[] array中注册，并且内部数组(array)的 Capacity大小默认为16。


app开发报价单

15道使用频率极高的基础 (894)
2014突破性科学技术：走 (835)
Java7/Java8中HashMap (819)

评论排行
Error 1935.安装程序集 M (0)
用java代码新建布局 (0)
关于"OnClickListener ca (0)
2014突破性科学技术：走 (0)
如何快速、高效地阅读文 (0)
十进制到二进制转换 (0)
汉诺塔 (0)
n! (0)
九九乘法表 (0)
Vi相关操作 (0)

推荐文章
* 探索通用可编程数据平面
* 这是一份很有诚意的 Protocol Buffer 语法详解
* CSDN日报20170420 ——《开发和产品之间的恩怨从何来?》
* Android图片加载框架最全解析——从源码的角度理解Glide的执行流程
* 如果两个程序员差不多，选写作能力更好的那个
* 从构造函数看线程安全



上述图片展示了HashMap内部存储的实例，其中包含有空的Entry，每个Entry连接在一起形成单链表，若key的hashcode相等，则它们存在同一个链表上（bucket），若key的hashcode不相等则存储在不同的bucket中；

每当调用put(K key,V value)或者get(Object key)时首先得计算bucket的index(索引值)，判断是否在同一个Entry中，然后使用迭代器（iterates）遍历链表，然后再调用equals()方法在同一个bucket中寻找对应的value值；

在get()中，如果value存在的话，则返回Entry对应的value;

在put(K key ,V value)方法中如果entry存在，则利用新的value值置换原来的value，并且在链表 个新的entry；

桶（链表）的Index值产生需要以下3步：

1. 计算Key的hashcode;
2. 为了避免性能较差的hash函数计算key的hashcode都一样，进而导致所有数据都放在同一个bucket中,需要重新计算hashcode;
3. rehash采用当前key的hashcode与(数组长度-1)相&(该操作假设所有的index都在数组的长度范围之内,可以把它看作是一个模计算的优化函数)

Java7和Java8处理Index的源码：

[java] view plain copy print ?

```
01. <pre name="code" class="java"><span style="font-
    size:18px;">// the "rehash" function in JAVA 7 that takes the hashcode of the key
02. static int hash(int h) {
03.     h ^= (h >>> 20) ^ (h >>> 12);
04.     return h ^ (h >>> 7) ^ (h >>> 4);
05. }
06. // the "rehash" function in JAVA 8 that directly takes the key
07. static final int hash(Object key) {
08.     int h;
09.     return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  //>>>无符号右移
10.     }
11. // the function that returns the index from the rehashed hash
12. static int indexFor(int h, int length) {
13.     return h & (length-1);
14. }</span>
```

为了提升效率，内部数组的大小必须是2的幂次方，原因如下：

假设数组长度为17，那么mask值就是17-1=16。二进制表示形式：0...010000，对于任何hash值H通过位操作(H & 16)结果要么是16要么是0，这表明数组长度为17的时候key只能存放在2个bucket之中的任意一个(index=0或者index=16),效率很低；

但是当数组的长度是16的时候(2^4)，位操作(H & 15),二进制表示形式0...001111,因此结果可以是0到15之间，数组中的每个bucket得到充分利用，举例来说：

- 若H=952，二进制表示形式：0...01110111000,其index就是0...01000=8;
- 若H=1576，二进制表示形式：0...0111000101000,其index就是0...01000=8;
- 若H=12356，二进制表示形式：0..01011110010001010000110010,其index就是0...00010=2;
- 若H=12356，二进制表示形式：0..01110100111000011,其index就是0...00011=3;

以上就是为什么数组的长度都是2的幂次方，该机制对程序开发者是透明的：如果选择hashmap的大小为37，那么Map会自动选择扩容到37以后的2次幂(64)

(Auto resizing)自动调整大小：

每次计算index之后，get()、put()、remove()方法访问或者遍历链表来查看对于给定的key是否存在对应的Entry.在没有调整的情况下，由于函数需要迭代整个Entry链表来查看给定的entry是否存在，性能很低；假设内部数组的长度初始为16，但是你需要存放200万个元素，在最好的情况下，每个链表都会存放125000(2/16*1000000) 个元素

，因此每次get(),remove()和put()至少需要125000(2/16*1000000)次迭代操作，为了避免这种情况，HashMap可以自动调整内部数组长度来保证链表长度最短；

当你创建一个HashMap时，你可以用以下构造器来指定初始化的大小和LoadFactor:

```
[java] view plain copy print ?
01. <pre name="code" class="java"><span style="font-size:18px;">public HashMap(int initialCapacity, floatloadFactor)</span>
```

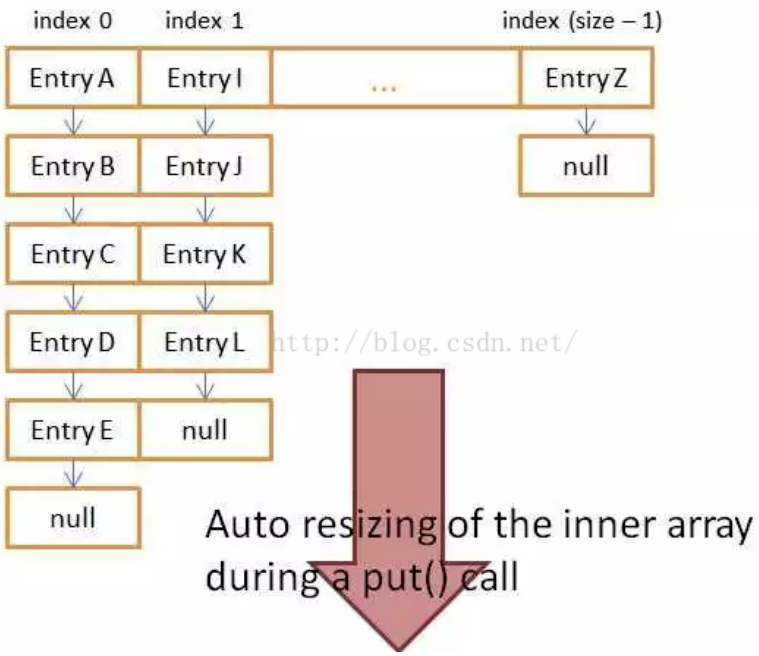
如果你没有指定参数的大小，默认的initialCapacity的大小是16，LoadFactor是0.75，initialCa
部数组的长度；

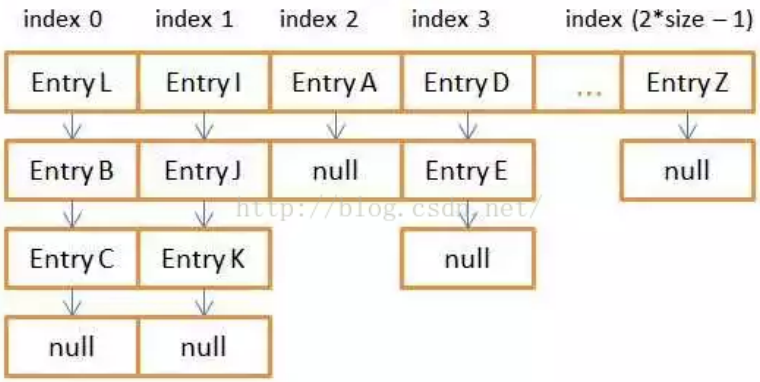
每次使用put()添加key/value到Map时首先会检查其是否需要增加内部数组的capacity，那么该怎
可以在Map中存放2个数据：

- Map的size：表示HashMap中entry的数量，value在每次添加或者删除之后都需要更新；
- threshold(阈值Capacity*loadFactor):作为每次内部数组调整大小的依据。

在添加新的Entry之前，put(...)会检查size是否大于threshold，如果是，则数组长度变为原来的两倍(capacity*2),由于数组的大小改变了，index函数(hash(key) AND (sizeOfArray-1))改变了；因此，数组大小会变为原来的2倍并且重新存放所有的entry到bucket中(创建一个新的数组，把原来的entry存放到新的数组中)；

resize的目的就是为了降低链表长度改变时put()/remove()或者get()方法执行的代价，在所有的entry中，若hashcode的key的hashcode相同，则这些 entry会存放到同一个bucket中；但是不同hash值的2个entry可能经过调整之后会处在同一个bucket中。





上图展示了内部数组在resize之前和之后的状态，在增加size之前，为了获得Entry E，map必须迭代整个链表5次，而resize之后，相同的get()操作仅仅需要迭代整个链表2次，速度是原来的2倍；

PS:HashMap仅仅增加内部数组的大小，它并不需要减小其size。

Thread Safety(线程安全性)：

大家都知道HashMap是线程非安全的，但这是什么原因呢？举例来说，假设你有一个Writer线程往数据到Map中，同时一个Reader线程需要从Map中读取数据，会发生什么现象呢？

- 由于HashMap中的auto-resizing机制，如果一个线程试图执行put()或者get()方法时，map也许会返回old index值而找不到bucket中对应Entry更新后的值；
- 最坏的情况就是当2个线程同时调用put()方法时需要同时执行Map中的resize()方法，因为两个线程同时修改了链表，Map可能会在链表中执行内部循环，如果你内部循环去获取链表中的数据时，get()也会不断循环。

HashTable采用了线程安全的策略来防止上述情况的发生，但是所有的CRUD方法都同步会导致执行效率很低。

举例来说，如果线程1调用了get(key1),线程2调用了get(key2)以及线程3调用了get(key3),但同一个时刻只有一个线程可以执行get()方法得到相应的value。

更加高效的线程安全方法在Java5中已经实现了：ConcurrentHashMap.仅仅只有在同一bucket中才需要同步，而没有访问相同的bucket或者不需要resize内部数组大小时允许多个线程在同一时刻get(),remove()或者put()数据，因此最好是在多线程中使用此种方法。

Key immutability(key的不变性)：

对于HashMap来说，为什么String和Integers是一个好的方法？大多数情况下是因为其不可改变的，如果你新建了一个key类而没有使其immutable,在HashMap中也许会改变key中的数据。

看看以下的示例：

- 一个key的值为1；
- 利用put()将key添加到Map中；
- HashMap生成key对应的HashCode；
- Map在新建的Entry中存放了该hash值；
- 修改了key值为2
- key的hash值被修改了，但是HashMap并没有修改(因为old hash被保存下来了)
- 用get()获取修改后的key
- map计算key(2)的hash寻找其是否在链表中的entry:
  - 由于修改了key，map在错误的bucket中寻找entry并没有找到
  - 修改了key之后产生的bucket与之前old产生的bucket相同，map迭代整个链表来找相同key的entry，找到了key，map首先计算hash值，然后调用equals()方法来比较，由于修改后的key并没有产生相同的hash，所以map找不到链表的entry

下面有一个例子，I put 2 个key-value键值对到map中，I修改了第一个key然后去get 修改后的2个值，结果仅仅返回了第二个value，第1个值在HashMap中丢失了：

> **[java]** view plain copy print ?


```
<pre name="code" class="java"><span style="font-size:18px;">public class MutableKeyTest {

    public static void main(String[] args) {

        class MyKey {
            Integer i;

            public void setI(Integer i) {
                this.i = i;
            }

            public MyKey(Integer i) {
                this.i = i;
            }

            @Override
            public int hashCode() {
                return i;
            }

            @Override
            public boolean equals(Object obj) {
                if (obj instanceof MyKey) {
                    return i.equals(((MyKey) obj).i);
                } else
                    return false;
            }

        }

        Map<MyKey, String> myMap = new HashMap<>();
        MyKey key1 = new MyKey(1);
        MyKey key2 = new MyKey(2);

        myMap.put(key1, "test " + 1);
        myMap.put(key2, "test " + 2);

        // modifying key1
        key1.setI(3);

        String test1 = myMap.get(key1);
        String test2 = myMap.get(key2);

        System.out.println("test1= " + test1 + " test2=" + test2);

    }

}</span>
```

输出结果："test1= null test2=test 2". 正如所预期的那样，Map并没有找到修改后第一个key1的字符串

下面看看Java8在这方面是如何改进的：

HashMap的内部表示在Java8中得到很大的改进，比如说在Java7中实现HashMap需要1k行代码，而在Java8中需要2K行代码。在Java8中内部仍然以数组实现，但是以节点(Node)来作为Entry存储信息，并且同样也包括链表：

以下就是Java8中Node部分实现：

```
[java] view plain  copy  print  ?
<pre name="code" class="java"><span style="font-size:18px;">static class Node<K,V> implements Map.Entry<K,V> {
        final int hash;
        final K key;
        V value;
        Node<K,V> next;
</span>
}
```

因此这和Java7有很大的区别吗？当然了，节点(Nodes)可以扩展为树节点(TreeNode)。一个树节点(TreeNode)可以扩展成一棵红黑树(red-black tree)结构来存储更多的信息，还可以高效(时间复杂度为Olog(n))的来执行add,delete或者get等操作。

下面是树节点存储的详细链表：

```
[java] view plain  copy  print  ?
<pre name="code" class="java"><span style="font-size:18px;">static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V> {
    final int hash; // inherited from Node<K,V>
    final K key; // inherited from Node<K,V>
    V value; // inherited from Node<K,V>
    Node<K,V> next; // inherited from Node<K,V>
    Entry<K,V> before, after;// inherited from LinkedHashMap.Entry<K,V>
    TreeNode<K,V> parent;
    TreeNode<K,V> left;
    TreeNode<K,V> right;
    TreeNode<K,V> prev;
    boolean red;
}</span>
```

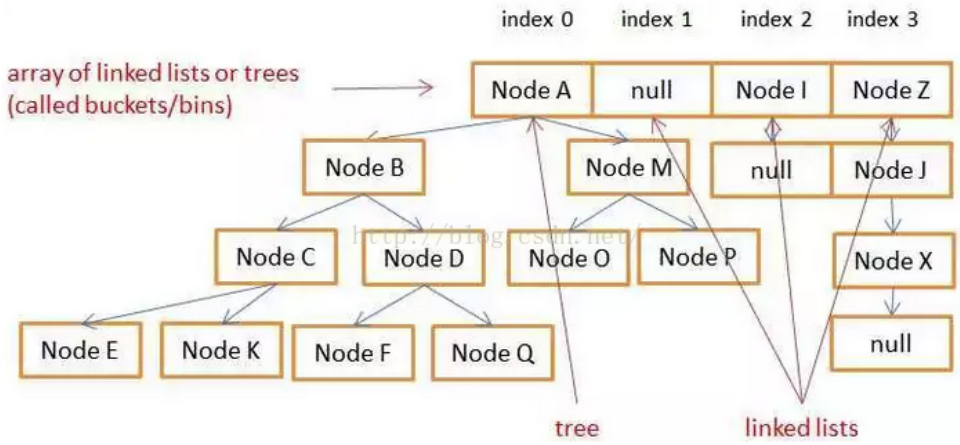
红黑树本身就是二叉平衡搜索树，其内部机制可以保证无论添加或者删除节点，其时间复杂度总是为log(n).主要优点就是当很多数据在数组中的同一个index(bucket)中的时候使用红黑树，此时的搜索代价都是 链表的开销是O(n)。

树其实占用比链表更多的空间

通过继承，内部表可以包括节点(链表)和树节点(红黑树)；**Oracle**按照以下规则来决定使用哪种**数据结构**：

——对于内部表中给定的index(bucket)，若超过8个节点，链表可以转换为红黑树；

——对于内部表中给定的index(bucket)，若少于6个节点，树可以转换为链表；



上图展示了Java8中HashMap以树和链表来表示的数组，其中(bucket 0超过8个节点)以树来实现，链表(bucket 1,2,3少于6个节点)用链表来表示。

Memory overhead (内存开销):

Java7

HashMap使用树带来表示带来了一定内存的开销，在Java7中，HashMap中的Entry包括key-value键值对。

- 一个Entry包括以下内容：
- 下一个entry的引用；
- 预先计算的hash(integer)；
- key的引用；
- value的引用。

还有Java7使用内部数组，若Java7 HashMap包括N个元素，数组容量(CAPACITY),额外的内存开销大概如下：

sizeOf(integer)*N+sizeOf(reference)*(3*N+ CAPACITY);

其中：integer为4个字节;引用的大小取决于JVM/OS/Processor,通常为4个字节;

内存开销大致为：16*N+4*CAPACITY字节.

PS:Map在resize之后，内部数组的CAPACITY等于N之后的2次幂(如N=30，N=x^5=32)

自从Java7开始，HashMap执行了延迟初始化，这表示即使你给hashMap分配了大小，在内存中Entry内部数组也不会被分配空间，直到第一次使用put()方法才会分配内存。

Java8

在Java8中，内存分配稍微复杂点儿，由于一个节点可以包含:相同值的Entry或者相同数据超过6个引用，还有一个Boolean来判断是否为TreeNode。

如果所有的节点都是节点(Nodes)，Java8中HashMap内存分配和Java7中HashMap是相同的。

如果所有的节点是树节点(TreeNodes)，Java8中HashMap内存开销大致如下：

N * sizeOf(integer) + N * sizeOf(boolean) +sizeOf(reference)* (9*N+CAPACITY)

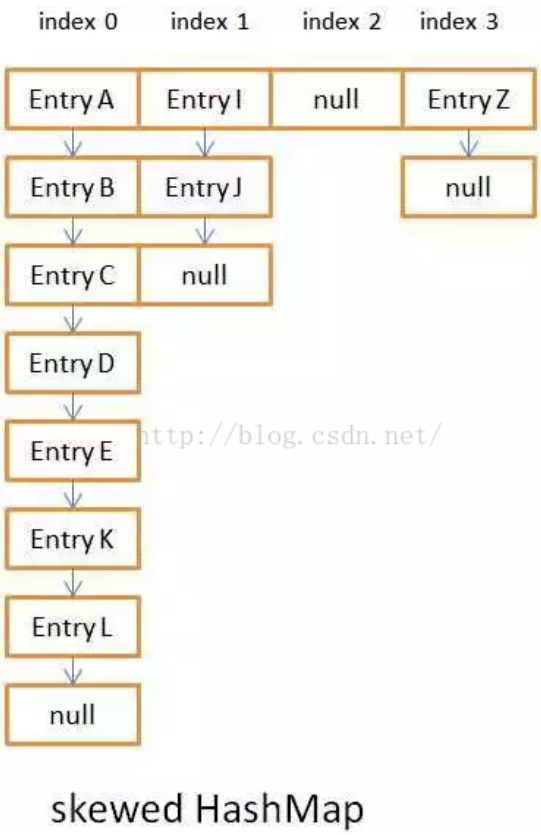
在大多数标准的JVM, 内存等于44 * N + 4 * CAPACITY字节

Performance issues(性能问题)：

非平衡的HashMap VS平衡的HashMap

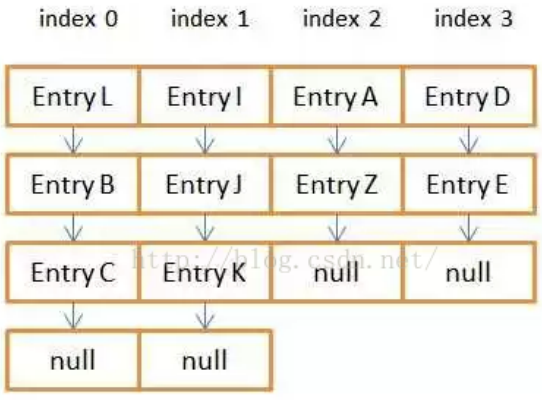
在最好的方案中，get()和put()方法时间复杂度为O(1)。但是，如果你不关心key的hash函数，put()和get()效率可能会很低.put()性能的好坏取决于内部数组不同buckets索引中的数据的重新分配情况；如果[illegible]设计得很好，那么数据将会重新分配(无论内部数组的容量多大)，所有的put()和get()方法都会遍历整个链表导致性能很低。在最坏的情况下(如果所有的数据存在同一个bucket中),时间复杂度为O(n)。

下面一个例子来说明非平衡HashMap和平衡HashMap的区别：



skewed HashMap

在非平衡HashMap中，bucket0中get()和put()方法代价很大，Getting Entry需要6次迭代。



well balanced HashMap

在平衡HashMap中，getting Entry只需迭代3次，所有的HashMaps存储等量的数据并且仅仅是桶中Entry分布的hash函数值不同。

下面有个例子，创建了一个hash函数，put所有的数据到同一个bucket中，然后添加200万个元素

```
[java] view plain copy print ?
<pre name="code" class="java"><span style="font-size:18px;">public class Test {

    public static void main(String[] args) {

        class MyKey {
            Integer i;
            public MyKey(Integer i){
                this.i =i;
            }

            @Override
            public int hashCode() {
                return 1;
            }

            @Override
            public boolean equals(Object obj) {
            …
            }

        }
        Date begin = new Date();
        Map <MyKey,String> myMap= new HashMap<>(2_500_000,1);
        for (int i=0;i<2_000_000;i++){
            myMap.put( new MyKey(i), "test "+i);
        }

        Date end = new Date();
        System.out.println("Duration (ms) "+ (end.getTime()-begin.getTime()));
    }
}</span>
```

在我电脑上，处理器为i5-2500K@3.6GHz上，已经超过了50分钟(我在50分钟杀死了该进程)

现在，如果相同的代码，我使用以下的hash函数：

```
[java] view plain copy print ?
<pre name="code" class="java"><span style="font-size:18px;">@Override
    public int hashCode() {
        int key = 2097152-1;
        return key+2097152*i;
}</span>
```

它需要46S，效果很棒！Hash函数相对于第一个来说对数据会更好地重新分配，因此put()方法更快。而我利用以下的hash函数可以更好地再分配：

```
[java] view plain copy print ?
<pre name="code" class="java"><span style="font-size:18px;">@Override
 public int hashCode() {
```


```
03.     return i;
04. }</span>
```

它仅仅需要2S！

因此设计Hash函数是极其重要的，如果相同的**测试**在Java7上，时间复杂度会更高(在Java7中put()时间为O(n),Java8中为Olog(n))；

当使用HashMap的时候，你需要找到合适的Hash函数尽可能的将key分配到更多的bucket里面(减少碰撞次数).如此你便可以避免hash碰撞。String对象是非常好的key由于其有很好的hash函数，Integers的hashcode是自己的value值也是很不错的。

Resizing overhead(重新调整大小的开销):

如果你需要存储大量的数据，你应该制定HashMap的Capacity，若不这样做的话，Map默认的size为16，LoadFactor为0.75，第11 次之前的put()是非常快的，但是第12次(16*0.75)将会重新创建一个新的内部数组(大小为32).13到23次会很快，而第23(32*0.75)次又会重新创建一个新的内部数组。内部重新调
第48个，96个，192个...调用put()。在容量很小时，重新创建内部数组是很快的但是当数据很多时就会花很多秒甚至几分钟来完成。通过初始化的时候设置所需容量的大小，可以避免该操作的性能开销。

但是这有一个弊端：如果你设置的数组的大小为2^28,但是你只能使用2^26个桶；会浪费太多的空间(3/4节)

总结：

举个简单的例子，你不需要知道HashMap如何工作的，因为你看不到O(1) O(n) O(logn)的区别，但是可以更好的理解数据结构的潜在机制，同时这对于Java开发者来说是一道很好的面试题。

在数据量很大的情况下，了解HashMap如何运行以及hash函数的重要性是极其重要的。

希望该文章可以帮助你更好地理解HashMap的实现机制！由于翻译水平有限，有误请指出！谢谢！

顶 0 踩 0

上一篇 Unity3D EasyTouch使用教程包括实例资源
下一篇 Facebook Android客户端热更新

我的同类文章

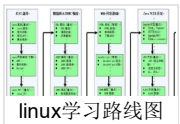
java (14)

Java并发 (多线程)	2015-11-14 阅读 351	java逆序英文句子中的单词顺..	2015-11-09 阅读 572
Java网络编程	2014-12-08 阅读 243	Java多线程	2014-12-05 阅读 247
线程同步：银行帐户存、取...	2014-11-30 阅读 792	Thread详解	2014-11-30 阅读 253
Java IO	2014-11-27 阅读 296	Java的异常处理机制	2014-11-12 阅读 289
Java泛型	2014-11-10 阅读 351	操作集合的工具类：Collecti...	2014-11-08 阅读 357

更多文章


web前端开发 薪资

夜大

linux红帽认证

linux学习路线图

电商托管

参考知识库

Java 知识库
25945 关注 | 1457 收录

Oracle知识库
4937 关注 | 252 收录

软件测试知识库
4532 关注 | 318 收录

Java SE知识库
25792 关注 | 479 收录

Java EE知识库
17775 关注 | 1316 收录

算法与数据结构知识库
15802 关注 | 2320 收录

猜你在找

数据结构基础系列(5)：数组与广义表
2016年11月软考信息安全工程师上午真题解析
2016年11月软考信息安全工程师下午真题解析
Android核心技术——Android数据存储
数据结构基础系列(7)：图
Mac OSX 中java7 java8环境的配置
java7新功能解析
java7 AbstractQueuedSynchronizer
java7语法新特性
Java7的新特性尝试

查看评论

暂无评论

发表评论

用户名： jinxin70

评论内容：

提交

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack
VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery
BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity
Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack
FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo
Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr
Angular Cloud Foundry Redis Scala Django Bootstrap

公司简介 | 招贤纳士 | 广告服务 | 联系方式 | 版权声明 | 法律顾问 | 问题报告 | 合作伙伴 | 论坛反馈

网站客服 杂志客服 微博客服 webmaster@csdn.net 400-600-2320 | 北京创新乐知信息技术有限公司 版权所有 | 江苏知之为计算机有限公司 | 江苏乐知网络技术有限公司

京 ICP 证 09002463 号 | Copyright © 1999-2016, CSDN.NET, All Rights Reserved