知识点整理

# 一、java HashMap工作原理及实现

## 1.概述

官方文档是这样描述的：

Hash table based **implementation of the Map interface**. This implementation provides all of the optional map operations, and permits null values and the null key. (The HashMap class is roughly equivalent to Hashtable, except that it is **unsynchronized** and **permits nulls**.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

大体意思就是：HashMap不是线程同步的，允许NULL值，不保证值的顺序，不保证在某个时间点值的顺序保持不变（因为HashMap在进行扩容的时候，需要重新移动元素，这就造成了元素的位置发生了变化。）

## 2.两个重要的参数

在HashMap中有两个重要的参数，容量（capacity）和负载因子（load factor）。

* **Initial capacity** The capacity is **the number of buckets** in the hash table, The initial capacity is simply the capacity at the time the hash table is created.
* **Load factor** The load factor is **a measure of how full the hash table is allowed to get** before its capacity is automatically increased.

简单的说，capacity就是HashMap中bucket数组的大小，loadFactor就是数据填充了bucket多满以后，容量需要扩充，默认情况下是loadFactor是0.75，扩充的时候扩充到原来的二倍。

使用上需要注意：如果在创建HashMap之前，预先知道数据的大小，假设是100，可以初始化HashMap的capacity为(10000+74)/75这样就可以保证不需要进行扩容，以免浪费性能。

## 3.put函数的实现

put函数的大致思路是：

1.根据KEY的hashcode计算hash，然后在打散，计算出index.

2.如果存在冲突，就把拼接成链表。

3.如果链表的长度超过限制，就转换成红黑树。

4.如果当前位置的key已经存在，就替换原来已经存在的oldValue。

5.如果需要进行扩容，就扩容。capacity变换为原来的二倍resize()

## 4.Get函数的实现

如果理解了put函数，get函数的实现就比较简单了。

1.根据KEY的hashcode进行计算hash，然后根据capacity计算出index。

2.如果找到bucket中相应的元素，并且进行key比较，是同一个key，则返回。

3.如果bucket含有多个元素，如果是TreeNode，则进行红黑树的遍历找到key，如果是链表，则遍历链表找到相应的key.

## 5.Hash函数的实现

从上图中可以看出，

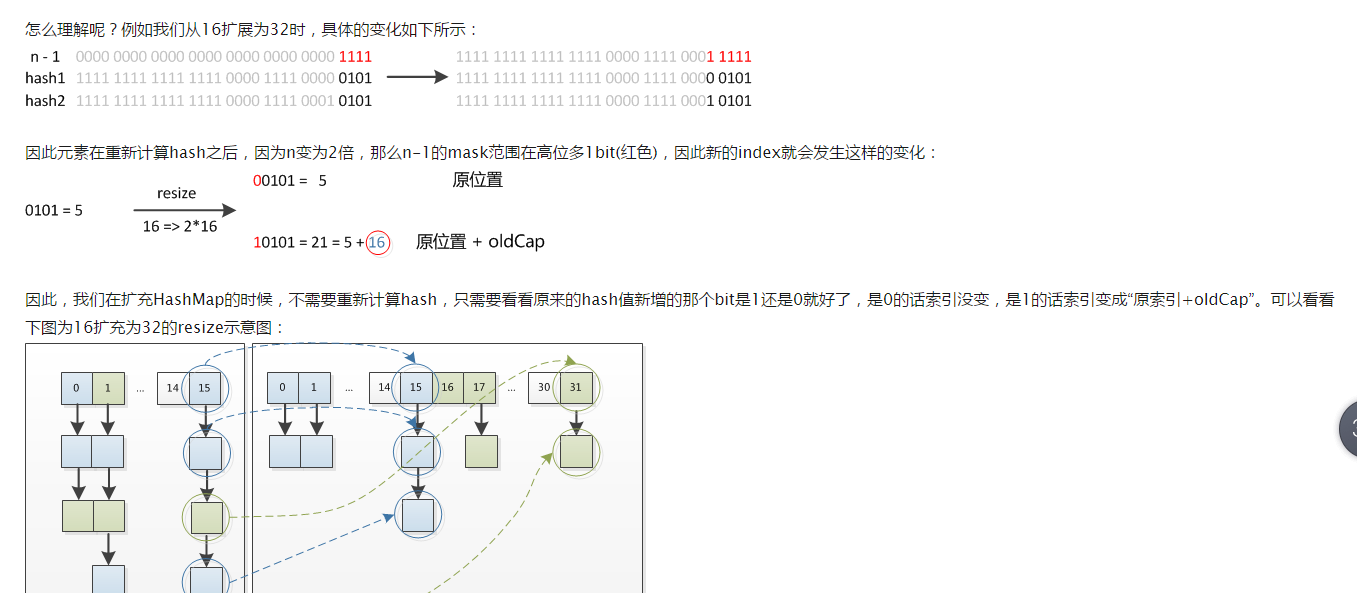
计算出key的hashCode,然后对hashCode和自身的高16bit进行异或处理，这样做的好处是保证高16bit中也参与到计算模打散的过程中。因为capacity是2的n次方，所以进行模n的时候，通过&（与）n-1即可求出模n 的余数。性能提升。

java8之前是利用链表来解决冲突的，java8之后，在链表长度达到8位的时候，转换成红黑树解决冲突，这样时间复杂度就变成了O(1)+O(log n) 相比之前的O(1)+O(n)

## 6.Resize的实现

源码对resize的描述是这样的：

Initializes or doubles table size. If null, allocates in accord with initial capacity target held in field threshold. Otherwise, because we are using power-of-two expansion, the elements from each bin must either **stay at same index**, or **move with a power of two offset** in the new table.

如上图所示的解释。

保证capacity在变成2倍的时候，每个节点根据自己存储的hash，然后判断是否需要移动位置即可。

## 7.总结

综上所述，hashMap主要的亮点如下：

1.resize的时候，扩容二倍，可以重新打散冲突的元素。

2.capaticy的大小为2的n次方。这样在求模取余的时候，可以用 (capacity-1)&hashcode代替hashcode%capacity

3.冲突的解决方式：java8之前的使用链表进行冲突解决，java8后开始使用链表进行解决，当链表超过8后，改成红黑树。

# 二、红黑树原理和算法详细介绍

## 1.概述

红黑树，又叫R-BTree , Red-Black Tree，它是一种特殊的二叉查找树。跟平衡二叉树的比较：

A.平衡二叉树（AVL）,是严格意义上的平衡树，而红黑树是近视平衡的二叉查找树。

B.平衡二叉树在搜索的时候，最坏时间复杂度严格是O(log n ) ，而红黑树因为不是严格平衡的，所以会有2 O(log n )。

C.但是在插入的时候，因为平衡二叉树是严格意义上的平衡树，所以在插入新元素的时候，很容易就引起了二叉树的不平衡，需要进行旋转调整。而红黑树，则能忍受不平衡的情况，所以可以很少的触发旋转机制。在插入的时候，平衡二叉树和红黑树都可以在最多两次旋转的情况下完成调整。而在删除的时候，平衡二叉树特殊情况下，会一直递归旋转到末尾，而红黑树只需要最多三次旋转就可以完成调整。

D.基于上述特性和区别的考虑，在进行数据测试的时候，红黑树表现的性能是优于平衡二叉树的，顾红黑树是在插入，删除和查找的三者操作里面选取一个折中。

**红黑树的特性：**

（1）每个节点或是黑色，或者是红色的。

（2）根节点是黑色的。

（3）每个叶子节点（NIL）是黑色。【注意：这里叶子节点，是指为空（NIL或NULL）的叶子节点。】

（4）如果一个结点是红色的，那么它的子结点必须是黑色的。

（5）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

注意：（1）特性（3）中的叶子节点，是只为空（NIL或NULL）的节点。

（2）特性（5），确保没有一条路径会比其他路径长出两倍。因而，红黑树是相对接近平衡的二叉树。

## 2.红黑树的应用

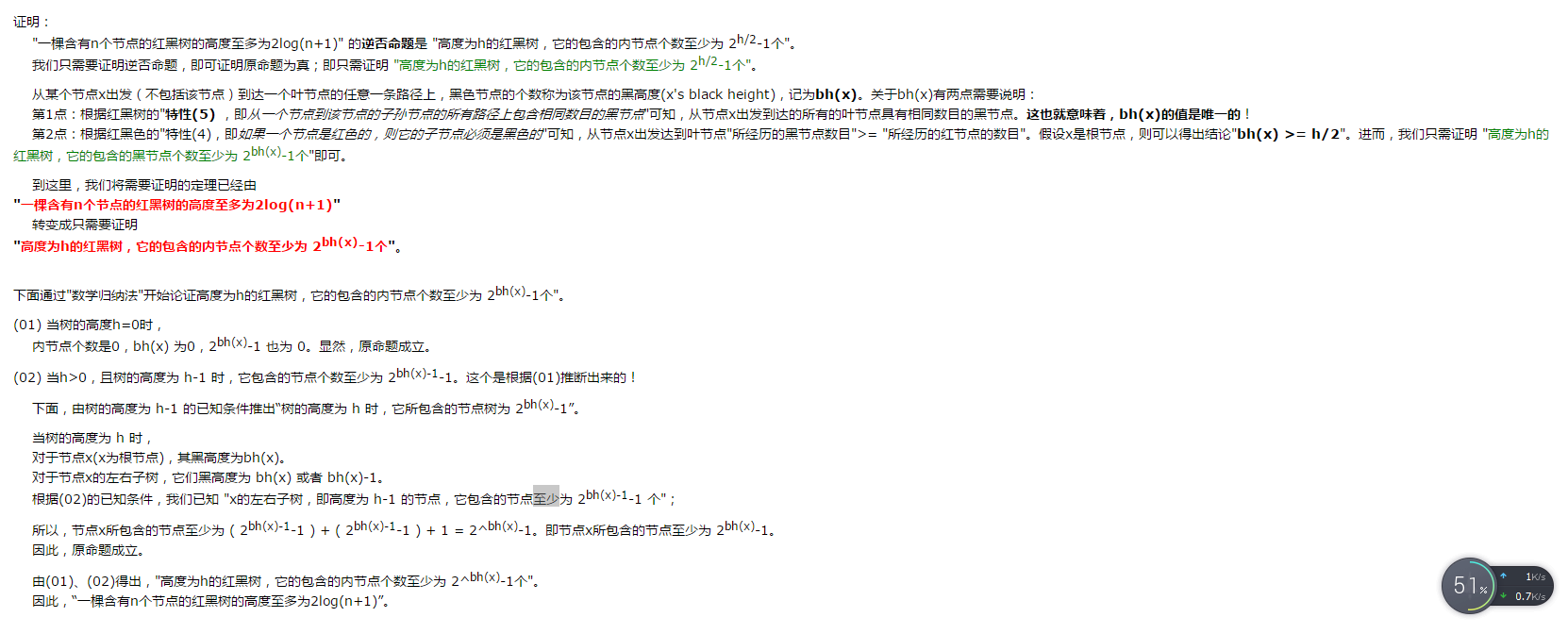
红黑树的应用比较广泛，主要用它来存储有序的数据，它的时间复杂度是O(logn),效率非常之高。例如：java中的TreeSet TreeMap,Linux中的虚拟内存管理，都是用红黑树实现的。

## 3.红黑树的时间复杂度和相关证明

红黑树的时间复杂度是O(logn)

时间复杂度可以通过数据归纳法进行证明。

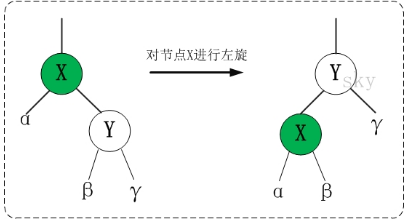
定理：一棵含有n个结点的红黑树，的高度至多为2log(n+1)。如果定理成立，则可以显然得出红黑树的时间复杂度是O(logN)

证明贴图如上。

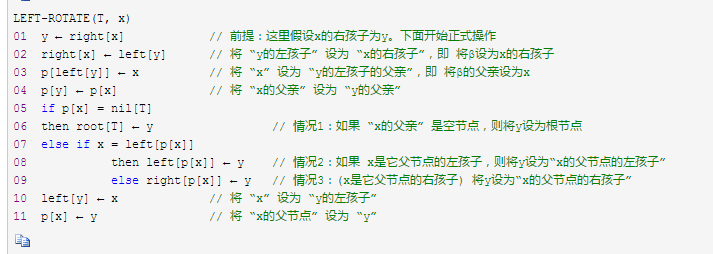
## 4红黑树的基本操作（一）：左旋和右旋

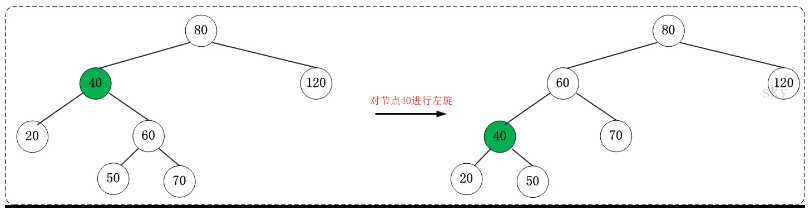
红黑树的基本操作是：添加和删除。而进行添加和删除，会有一定概率打破红黑树的基本特性，这时候就需要进行旋转，来保证适应红黑树的特点。

左旋：

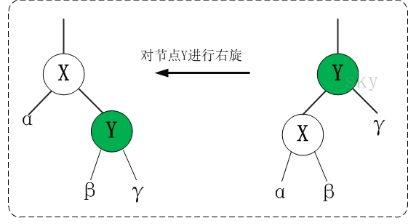


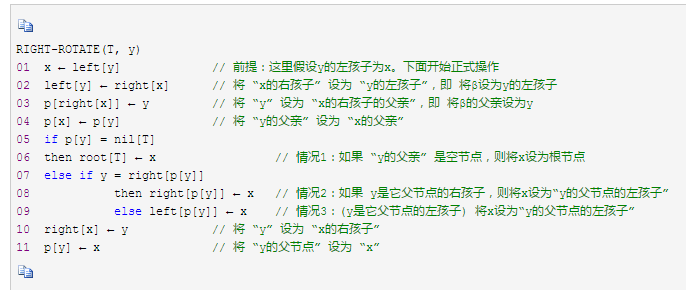
伪代码如下：

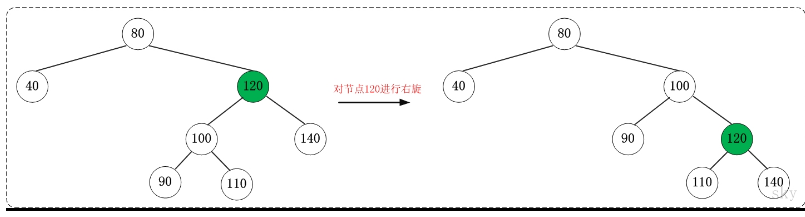




右旋







## 5红黑树的基本操作（二）：插入

将一个结点插入红黑树粗略经过如下步骤：

首先：把红黑树当作一个二叉查找树，进行查找，找到相应的位置。

然后：将结点着色为红色，插入到树中。

最后:如果该结点打破了红黑树的平衡，则通过旋转和修改着色，使之成为一个棵新的红黑树。

**1）将红黑树当做一个二叉查找树，将结点插入。**

因为红黑树本身就是一个二叉查找树，所以可以进行这种操作来确定需要插入的结点再书中的位置。

**2）将插入的结点着色为红色。**

为什么将结点着色成红色，而不着色成黑色呢。

五个特性：

（1）每个节点，或者是黑色的，或者是红色的。

（2）根节点是黑色的。

（3）每个叶子节点是黑色的。这里的叶子节点是只为空的叶子节点。

（4）如果一个结点是红色的，那么它的子结点一定是黑色的。

（5）从一个结点到包含子孙的所有节点的路径上，含有相同数目的黑色的结点。

按照红黑树的五个特性来进行考虑。将结点插入到树中，并不违背特性五，而其他四个特性会在一定条件下不满足，这时候就需要调整。而如果着色成黑色的，那一定不满足条件五，所以一定需要进行调整。

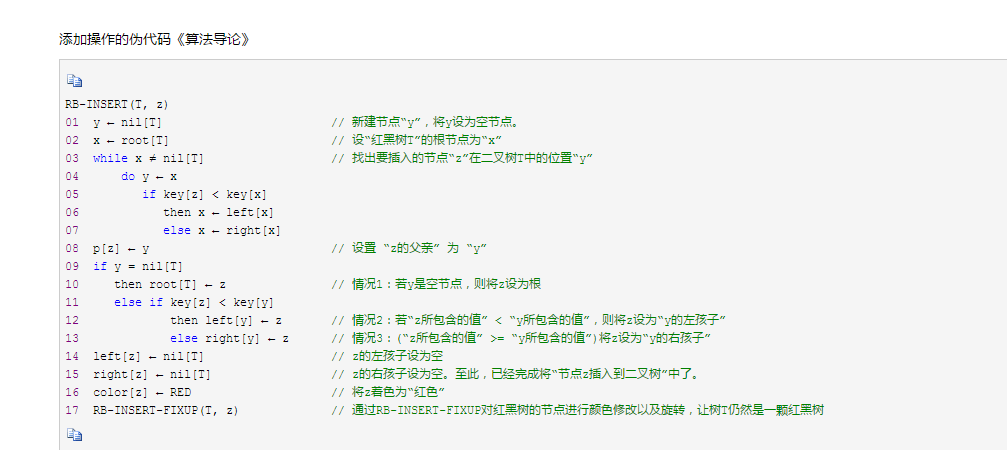
**3）通过一系列的旋转或着色等操作，使之重新成为一颗红黑树。**

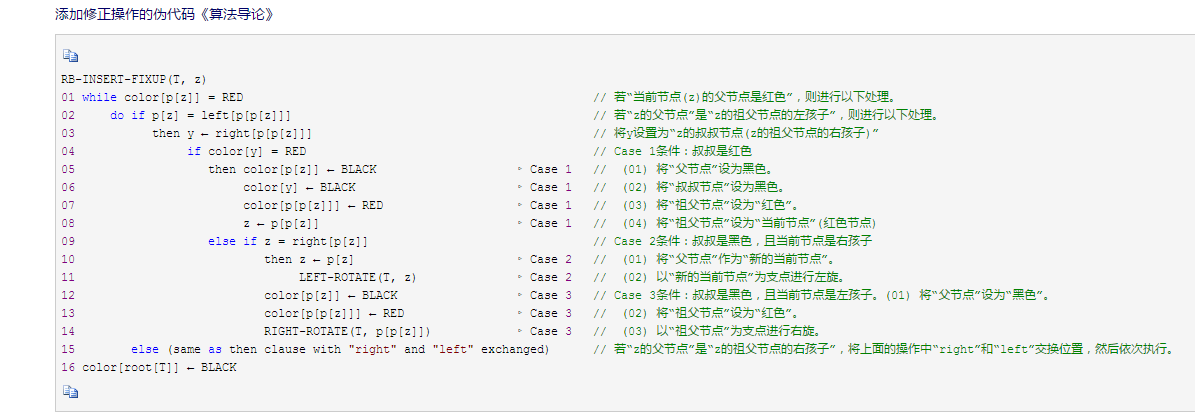
增加一个结点可能违背的特性：

对于特性1，显然不会违背了。对于特性2，显然也不会违背，因为根节点不会修改。

对于特性3：因为叶子节点是空节点，插入非空节点，并不会影响叶子节点。

对于特性4，是有可能违背的。





分情况讨论特性4违背后如何处理：

① 情况说明：被插入的结点是根节点

处理：直接把跟节点涂为黑色的。

② 情况说明：被插入的结点的父亲节点是黑色的。

处理：什么都不需要做，结点插入后形成的树依然是红黑树。

③ 情况说明：被插入的结点的父节点是红色的。

处理：该情况与红黑树的特点5相冲突。该情况下需要进行旋转和着色进行处理。

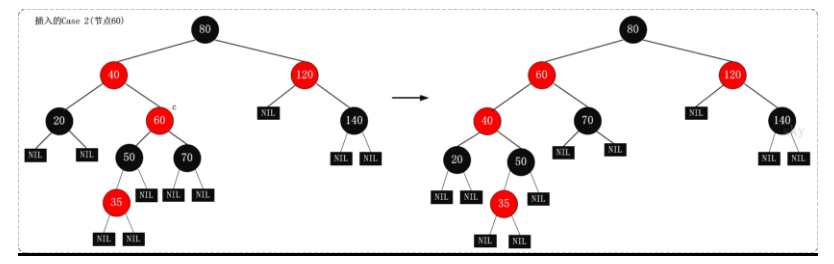
|  |  |  |
| --- | --- | --- |
|  | 现象说明 | 处理策略 |
| Case 1 | 当前结点的父结点是红色的，且当前结点的祖父结点的另一个子结点（即叔叔结点）也是红色的。 | （01）将“父节点”设为黑色的。  （02）将“叔叔结点”设置为黑色的。  （03）将祖父结点设置为红色的。（因为父亲节点为红色的，所以祖父结点一定是黑色的）  （04）因为修改了祖父结点为红色的，可能会影响祖父结点父辈之间的关系。所以将祖父几点当做当前结点， 继续进行旋转和着色操作，来完成红黑树的调整。 |
| Case 2 | 当前结点的父节点是红色的，叔叔结点是黑色的，并且当前结点是其父亲节点的右孩子。 | （01）将“父节点”作为新的当前结点。  （02）以“新的当前结点”为支点进行左旋。 |
| Case 3 | 当前结点的父节点是红色的，叔叔结点是黑色的，并且当前结点极其父亲节点的左孩子。 | （01）将“父节点”设置为黑色。  （02）将“祖父结点设置为”红色  （03）以祖父为支点进行右旋转。 |

### 1）Case 1 叔叔是红色的。

这里进行说明修改步骤：首先插入结点是红色的，而父亲节点是红色的，叔叔结点也是红色的。这样推论，祖父结点就一定是黑色的。此时，需要将父亲节点修改为黑色的。这样父亲节点下的两个分支，因为没有加入黑色的结点，所以黑色数量是不变的。而叔叔结点这边，因为父亲节点是黑色的，黑色结点的数量+1，所以叔叔结点这边也需要+1，来保持平衡，所以叔叔结点修改为黑色的。而此时，祖父结点原来是黑色的，这就导致了祖父结点下面的这个子树黑色结点数量增加了1，需要将祖父结点修改为红色的。这样就保证了祖父结点这个子树的黑色结点数量保持不变。再把祖父结点当成当前结点继续向上一级进行考虑，如果祖父结点是根节点，那直接将祖父结点修改为黑色的，整棵树的黑色定点的高度增加1，也满足红黑树的特点。

### 2) Case 2叔叔是黑色的，且当前结点是右孩子。

解释：当前结点是红色的，而父亲节点是红色的，叔叔结点是黑色的，这个时候，以祖父节点为起点，那叔叔的分支和父亲的分支黑色的数量是一致的。所以，进行父亲内的子节点调整，只要不影响父亲节点这边的总的黑色结点的高度就可以了。因为父亲节点是黑色的，所以新结点的兄弟结点（即父亲节点的左孩子一定是一个黑色的结点），可以推断出，兄弟结点的黑色树的高度和当前结点的左右子孩子的黑色结点的高度是一样的。所以，可以将树以父亲节点为基准进行左旋转。即可保证当前子树的黑色结点数量不变。



### 3）Case 3叔叔是黑色的，且当前结点是左孩子。

处理策略：

（1）将“父节点”设为“黑色”

（2）将“祖父结点”设为“红色”

（3）以祖父结点为支点，进行旋转。

## 6 红黑树的基本操作（三）：删除

将红黑树的一个结点进行删除，就是先通过查找的方式，找到红黑树，然后删除，如果树不满足红黑树的条件，则进行调整，使之满足红黑树的基本条件。

步骤：

**第一步：将红黑树当成一个查找树，进行结点删除。**

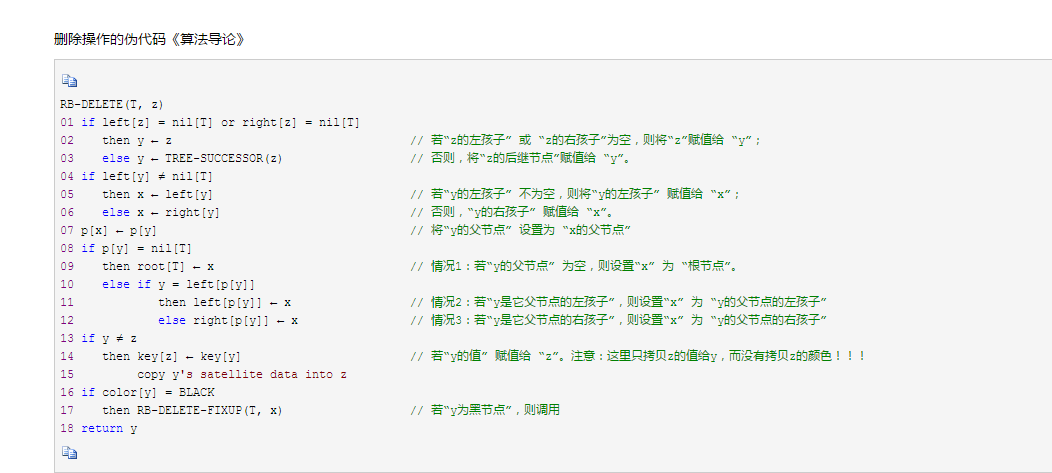
当作二叉查找树进行结点删除，分3中情况：

①：被删除的结点没有儿子，即为叶子节点。那么，直接将该节点删除就OK了。

②：被删除的结点只有一个儿子，那么直接删除该结点，并用该结点唯一的儿子替代该结点。

③：如果被删除的结点有两个儿子，那么找出该结点的后继结点，然后把该结点的值用后继结点代替，然后递归删除后继结点。

**第二步：通过旋转或者重新着色，来进行修正，使之成为一棵新的红黑树。**





**http://www.cnblogs.com/skywang12345/p/3245399.html**

**未完待续。**

# 三、读写锁的理解

## 1.概述

访问临界资源的时候，需要进行线程同步，而对临界资源进行操作的时候，可能进行读操作，也可能进行写操作。为了保持尽可能的并发性，需要区分读锁和写锁。这样可以保持读操作共享临界资源，而写操作当前只有一个线程进行操作。这样对读写锁一步一步升级，就可以形成并发性很好的读写锁。

## 2.简单的读写锁

读要求：多个线程可以同时访问临界资源。

写要求：同一时刻只能有一个线程进行写操作。并且，写操作的优先级高于读操作的优先级。因为如果写操作的优先级不高于读操作的优先级，就会造成“饥饿“现象，即临界资源一直被读线程占用，写请求等待读线程释放掉该资源，而此时会有源源不断的读线程占用资源，这样就导致写线程一直获取不到临界资源的操作权，即无法进行写操作。

实现代码：C3\_2\_1ReadWriteLock

**package** c3;  
  
*/\*\*  
 \* Created by bjduanweidong on 2017/10/31.  
 \*/***public class** C3\_2\_1ReadWriteLock {  
 *//当前读锁的线程数* **private int readers** = 0;  
 *//当前写锁的线程数* **private int writers** = 0;  
 *//当前请求写锁但是还未获取到写锁的线程数* **private int writeRequests**=0;  
  
 */\*\*  
 \* 获取读锁  
 \** ***@throws*** *InterruptedException  
 \*/* **public synchronized void** lockRead() **throws** InterruptedException {  
 *//循环获取读锁，因为写锁的优先级比较高，这里使用了  
 //writers 和 writeRequests两个来进行比较,而且是while循环，这样唤醒了就一直循环下去。* **while**(**writers**>0||**writeRequests**>0){  
 wait();  
 }  
 **readers**++;  
 }  
  
 */\*\*  
 \* 释放读锁  
 \*/* **public synchronized void** unlockRead(){  
 **readers**--;  
 notifyAll();  
 }  
  
 */\*\*  
 \* 获取写锁  
 \*/* **public synchronized void** lockWrite() **throws** InterruptedException {  
 *//首先增加writeRequests* **writeRequests**++;  
 *//循环等待 读锁或者写锁释放* **while**(**readers**>0||**writers**>0){  
 wait();  
 }  
 *//获取到了写锁* **writers**++;  
 *//释放writeRequests 这两条的顺序没有没有区别，因为一旦代码执行到了这里，就是一个原子操作，用synchronized修饰。* **writeRequests**--;  
 }  
  
 **public synchronized void** unlockWrite(){  
 **writers**--;  
 notifyAll();  
 }  
}

说明：这里说明几点，释放读锁或者写锁的时候，使用notifyAll()，这个命令是为了保证不会出现“信号量丢失”现象，比如：线程1获取写锁，但是这个时候被其他线程占用，线程1等待，这时候线程2来获取读锁，因为线程1获取写锁的存在，线程2必须等待，这时候如果线程3来了，获取写锁，也会进行等待。此时writeRequest 为2 ， 此时线程完成了操作，轮到线程1进行写操作，线程1完成以后，唤醒线程2，因为线程3的存在，线程2必须等待，这样就造成了“信号量丢失”的现象。所以需要使用notifyAll来进行唤醒线程。再者，如果

于是，这种读写锁的实现存在弊端。

## 3.可重入锁。

现象：当线程1获取资源的读锁，此时线程2来获取线程的写锁，而此时线程1再想获取资源的读锁的时候，就会造成阻塞，这就产生了死锁现象。

为了避免这种现象的产生，引入了可重入锁。此时线程1获取该资源的读锁的时候，可以继续获得锁，不会出现死锁现象。

读锁和写锁都加入可重入。

实现代码：C3\_3\_1ReadWriteLock

**package** c3;  
  
**import** java.util.HashMap;  
**import** java.util.Map;  
  
*/\*\*  
 \* 可重入锁  
 \* Created by bjduanweidong on 2017/10/31.  
 \*/***public class** C3\_3\_1ReadWriteLock {  
  
 *//当前写锁的线程,重入的次数* **private int writers** = 0;  
 *//当前请求写锁但是还未获取到写锁的线程数* **private int writeRequests**=0;  
 *//存储当前进入锁的线程以及重入数量* **private** Map<Thread,Integer> **readingThreads** = **new** HashMap<>();  
  
 */\*\*  
 \* 记录当前的写线程  
 \*/* **private** Thread **writingThread**;  
  
  
 */\*\*  
 \* 获取读锁  
 \** ***@throws*** *InterruptedException  
 \*/* **public synchronized void** lockRead() **throws** InterruptedException {  
 Thread callingThread = Thread.*currentThread*();  
 **while**(!canGrantReadAccess(callingThread)) {  
 wait();  
 }  
 *//获取到锁以后，进行+1* **readingThreads**.put(callingThread,getAccessCount(callingThread)+1);  
 }  
  
 */\*\*  
 \* 获取读权限  
 \** ***@param callingThread*** *\** ***@return*** *\*/* **private boolean** canGrantReadAccess(Thread callingThread) {  
 *//如果有进行写操作的线程，获取不了* **if**(**writers**>0)**return false**;  
 *//当前读线程，可以读* **if**(isReader(callingThread))**return true**;  
 *//如果有写等待，不可以* **if**(**writeRequests**>0)**return false**;  
 **return true**;  
 }  
  
 **private boolean** isReader(Thread callingThread) {  
 **return readingThreads**.get(callingThread)!=**null**;  
 }  
  
 **private int** getAccessCount(Thread callingThread) {  
 Integer count = **readingThreads**.get(callingThread);  
 **if**(count==**null**){  
 **return** 0;  
 }**else**{  
 **return** count.intValue();  
 }  
 }  
  
 */\*\*  
 \* 释放读锁  
 \*/* **public synchronized void** unlockRead(){  
 Thread callingThread = Thread.*currentThread*();  
 **int** accessCount = getAccessCount(callingThread);  
 **if**(accessCount==1){  
 **readingThreads**.remove(callingThread);  
 }**else**{  
 **readingThreads**.put(callingThread,accessCount-1);  
 }  
 notifyAll();  
 }  
  
 */\*\*  
 \* 获取写锁,写重入锁  
 \*/* **public synchronized void** lockWrite() **throws** InterruptedException {  
 Thread callingThread = Thread.*currentThread*();  
 *//首先增加writeRequests* **writeRequests**++;  
 *//循环等待 读锁或者写锁释放* **while**(!canGrantWriteAccess(callingThread)){  
 wait();  
 }  
 *//获取到了写锁* **writers**++;  
 *//释放writeRequests 这两条的顺序没有没有区别，因为一旦代码执行到了这里，就是一个原子操作，用synchronized修饰。* **writeRequests**--;  
 *//记录当前的写线程* **writingThread** = callingThread;  
 }  
  
 **private boolean** canGrantWriteAccess(Thread callingThread) {  
 *//当前写线程，可重入* **if**(isWriter(callingThread)){  
 **return true**;  
 }  
 *//有其他的写线程，或者存在读线程* **if**(**writers**>0||**readingThreads**.size()>0){  
 **return false**;  
 }  
 **return true**;  
  
 }  
  
 **private boolean** isWriter(Thread callingThread) {  
 **return writingThread**==callingThread;  
 }  
  
 **public synchronized void** unlockWrite(){  
 *//减少writers的数量* **writers**--;  
 **if**(**writers**==0){  
 **writingThread**=**null**;  
 }  
 notifyAll();  
 }  
}

## 4.锁升级，锁降级（读锁升级到写锁，写锁降级到读锁）

有的时候，我们希望拥有读锁的线程获取写锁，而这时候，我们要求该线程为唯一拥有读锁的线程。例如：此时线程1和线程2均获得了读锁，而此时，线程1想获取写锁，则线程1获取不到，因为线程2的存在，此时线程1就会堵塞，而线程2访问结束后，会释放读锁，而此时线程1被唤醒，获取资源的写锁。但是如果存在这种情况，线程1和线程2都获取了读锁，而此时二者又都要获取写锁，就会彼此等待，造成死锁的发生。此时代码中应该反映出，如果当前线程获取不到写锁，就抛出异常。详情见代码表示：

代码示例：C3\_4\_1ReadWriteLock

**package** c3;  
  
**import** java.util.HashMap;  
**import** java.util.Map;  
  
*/\*\*  
 \* 可重入锁  
 \* Created by bjduanweidong on 2017/10/31.  
 \*/***public class** C3\_4\_1ReadWriteLock {  
  
 *//当前写锁的线程,重入的次数* **private int writers** = 0;  
 *//当前请求写锁但是还未获取到写锁的线程数* **private int writeRequests**=0;  
 *//存储当前进入锁的线程以及重入数量* **private** Map<Thread,Integer> **readingThreads** = **new** HashMap<>();  
  
 */\*\*  
 \* 记录当前的写线程  
 \*/* **private** Thread **writingThread**;  
  
  
 */\*\*  
 \* 获取读锁  
 \** ***@throws*** *InterruptedException  
 \*/* **public synchronized void** lockRead() **throws** InterruptedException {  
 Thread callingThread = Thread.*currentThread*();  
 **while**(!canGrantReadAccess(callingThread)) {  
 wait();  
 }  
 *//获取到锁以后，进行+1* **readingThreads**.put(callingThread,getAccessCount(callingThread)+1);  
 }  
  
 */\*\*  
 \* 获取读权限  
 \** ***@param callingThread*** *\** ***@return*** *\*/* **private boolean** canGrantReadAccess(Thread callingThread) {  
 *//当前线程已经获取到了写锁，可以直接获取读锁* **if**(isWriter(callingThread)){  
 **return true**;  
 }  
 *//如果有进行写操作的线程，获取不了* **if**(**writers**>0)**return false**;  
 *//当前读线程，可以读* **if**(isReader(callingThread))**return true**;  
 *//如果有写等待，不可以* **if**(**writeRequests**>0)**return false**;  
 **return true**;  
 }  
  
 **private boolean** isReader(Thread callingThread) {  
 **return readingThreads**.get(callingThread)!=**null**;  
 }  
  
 **private int** getAccessCount(Thread callingThread) {  
 Integer count = **readingThreads**.get(callingThread);  
 **if**(count==**null**){  
 **return** 0;  
 }**else**{  
 **return** count.intValue();  
 }  
 }  
  
 */\*\*  
 \* 释放读锁  
 \*/* **public synchronized void** unlockRead(){  
 Thread callingThread = Thread.*currentThread*();  
 **int** accessCount = getAccessCount(callingThread);  
 **if**(accessCount==1){  
 **readingThreads**.remove(callingThread);  
 }**else**{  
 **readingThreads**.put(callingThread,accessCount-1);  
 }  
 notifyAll();  
 }  
  
 */\*\*  
 \* 获取写锁,写重入锁，可能存在读锁升级为写锁的情况  
 \*/* **public synchronized void** lockWrite() **throws** InterruptedException, IllegalAccessException {  
 Thread callingThread = Thread.*currentThread*();  
 *//首先增加writeRequests* **writeRequests**++;  
 *//循环等待 读锁或者写锁释放* **while**(!canGrantWriteAccess(callingThread)){  
 wait();  
 }  
 *//获取到了写锁* **writers**++;  
 *//释放writeRequests 这两条的顺序没有没有区别，因为一旦代码执行到了这里，就是一个原子操作，用synchronized修饰。* **writeRequests**--;  
 *//记录当前的写线程* **writingThread** = callingThread;  
 }  
  
 **private boolean** canGrantWriteAccess(Thread callingThread) **throws** IllegalAccessException {  
 *//当前写线程，可重入* **if**(isWriter(callingThread)){  
 **return true**;  
 }  
 **if**(isOnlyReader(callingThread)){  
 **return true**;  
 }  
 *//有其他的写线程，或者存在读线程* **if**(**writers**>0){  
 **return false**;  
 }  
 **return true**;  
  
 }  
  
 **private boolean** isOnlyReader(Thread callingThread) **throws** IllegalAccessException {  
 **if**(**readingThreads**.size()==1&&**readingThreads**.get(callingThread)!=**null**){  
 *//只有当前线程获得资源，可以升级为写锁。* **return true**;  
 }**else**{  
 *//不仅仅当前线程获得资源，可能有其他线程获取到了读锁，此时不能进行升级，并且不能阻塞，否则会发生死锁现象。* **if**(**readingThreads**.size()>1&&**readingThreads**.get(callingThread)!=**null**){  
 **throw new** IllegalAccessException(**"有其他线程已经获取到了该资源的读锁，请重试"**);  
 }  
 }  
 **return false**;  
 }  
  
 **private boolean** isWriter(Thread callingThread) {  
 **return writingThread**==callingThread;  
 }  
  
 **public synchronized void** unlockWrite(){  
 *//减少writers的数量* **writers**--;  
 **if**(**writers**==0){  
 **writingThread**=**null**;  
 }  
 notifyAll();  
 }  
}

## 5.小结

上面实现是基于synchronized实现的读写锁，性能并不是很优秀，只是通过这种方式，加深对读写锁特点的理解。

# 四、ReentrantReadWriteLock 的源码解读

1.概述：

# 五、java锁的理解

锁作为并发共享数据，保证数据一致性的工具。锁主要分为以下几类：

## 1.自旋锁。

### A.概述

自旋锁的意思就是：竞争某一临界资源的进程先获得控制该临界区的锁。获取锁的方式就是不停的check锁的状态，是不是可以进入临界区。通常的实现就是CAS操作，如果某个state（volatile）的值，0代表是没有线程进入临界区，1代表已经有线程获取了锁，进入了临界区。通过CAS(0,1).

操作，如果不成功，就继续执行，如果成功了，就代表进入了临界区。

### B.示例代码

见：C5\_1\_1SpinLock

**package** c5;  
  
**import** java.util.concurrent.atomic.AtomicIntegerFieldUpdater;  
  
*/\*\*  
 \* 自旋锁的实现代码  
 \* Created by bjduanweidong on 2017/11/23.  
 \*/***public class** C5\_1\_1SpinLock {  
 *//获取锁的状态。0代码没有任何线程获取锁， 1 代表该锁已经被其他线程拥有。* **private volatile int state** = 0;  
  
 */\*\*  
 \* 拥有该锁的线程。  
 \*/* **private volatile** Thread **thread**;  
  
 **private static final** AtomicIntegerFieldUpdater ***UPDATER*** = AtomicIntegerFieldUpdater.*newUpdater*(C5\_1\_1SpinLock.**class**,**"state"**);  
  
 **public void** lock(){  
 *//* **while**(!***UPDATER***.compareAndSet(**this**,0,1));  
 **thread** = Thread.*currentThread*();  
 }  
  
 **public void** unlock(){  
 **if**(**thread**==Thread.*currentThread*()){  
 **state**=0;  
 **thread**=**null**;  
 }  
 }  
}

### C.优缺点比较

优点：与传统的阻塞式锁比较，能得到效率上的提高。

缺点：

①但是因为线程不阻塞，所以在获取锁的过程中，频繁的消耗CPU资源，以及在进行state状态检查的时候，会引起内存的竞态，通过频繁的冲刷cache来解决，从而降低cache的命中率。

②因为线程都是竞争获取锁，所以后来的线程也可能先获得到锁，这对于对于锁的获取顺序要求比较严格的应用来说，不适用。即非公平锁。

## 2.自旋锁的其他种类。

## 3.阻塞锁

## 4.可重入锁。

## 5.读写锁。

## 6.互斥锁。

## 7.悲观锁。

## 8.乐观锁。

## 9.公平锁。

## 10.非公平锁。

## 11.偏向锁。

## 12.对象锁。

## 13.线程锁。

## 14.锁粗化。

## 15.轻量级锁。

## 16.锁消除。

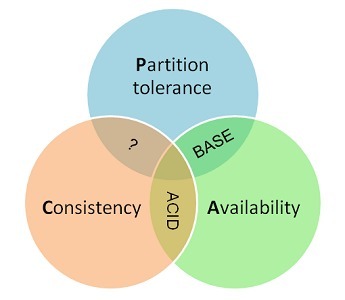
## 17.锁膨胀。

## 18.信号量。

# 六、CAP定理

## 1.概述

任何一个分布式系统中，最多只能同时满足，一致性，可用性，分区容错性中的两项。



## 2.定义

Consistency(一致性)：“all nodes see the same data at the same time”,即所有的结点在同一时刻，数据是一致的。

Avaliablity(可用性)：“Reads and writes always succeed.” 即，服务一直可用，而且是正常相应时间。

Partition Tolerance(分区容错性)：“the system continues to operate despite arbitrary message loss or failure of part of the system.” 即分布式系统遇到信息丢失或者系统某结点故障的时候，仍然能够对外提供满足一致性和可用性的服务。

## 3.深入理解

这三者的关系在分布式系统中，是不停的利用策略来进行权衡的，某一个时刻，可能为了注重数据一致性，而降低了可用性，而另一个时刻，就为了保证可用，而降低了数据一致性，这种权衡，根据不同的系统的侧重点，进行取舍。

**一致性的理解**：

A.关系型数据库，每一次更新操作后，要求后续的操作都必须能看到这次更新的结果，这就是强一致性。

B.如果能容忍后面的访问部分或者全部都看不到上一次的更新，就是弱一致性。

C.如果经过一段时间后，能访问到更新后的数据，则是最终一致性。

D.最终一致性根据更新数据后各进程访问到数据的时间和方式不同，又可以区分为：

①因果一致性（causal consistency）:如果进程A通知进程B更新了一个数据项，那么进程B的后续访问将返回更新后的值，且一次写入将取代保证前一次写入。与进程A无关的进程C执行一般的最终一致性。

②读己之所写（READ-YOUR-WIRTES）：如果一个进行更新了一个数据项后，他总是访问更新后的值，不会看到旧值。

③会话一致性（SESSION）：这是上一个模型的实用版本。它把访问系统的进程放到回话上下文中，只要会话还存在，系统就保证“读己之所写”一致性，如果会话终止，则下一次访问就是一次新的会话，两次会话不会受影响。

④单调读一致性（monotonic）：如果程序看到了某个数据对象的某个值，那么之后的访问，不会出现旧值。

⑤单调写一致性：系统来自同一个进行的写操作按照顺序执行，如果系统保证不了这个，就很难变成了。

**可用性的理解：**

基本可用是指在系统出现不可预知的故障的时候，允许损失部分可用性，这不等同于系统不可用。

基本可用的例子：

A.响应时间上的损失。正常情况下，一个在线搜索引擎可以再0.5秒的时间内给用户返回结果，但是出现异常后，可能会增加到1~2秒。

B.功能上的损失：比如正常的时候可以访问服务的所有页面，但是出现故障的情况下，只能访问到系统的降级页面。

**BASE理论**

BASE理论是Basically Available (基本可用)、Soft State(软状态) 和Eventually consistency（最终一致性）的缩写。Base是对cap中可用性和一致性权衡的结果，是基于CAP定理演化而来的，其核心思想是，即使无法做到强一致性，但每个应用都可以根据自身的业务特点，采用适当的方式，来达到最终一致性。

## 4.总结

CAP在分布式系统中，如何选择，可以根据业务的情况。如果业务要求数据准确一致，就类似抢票系统，那只能损失扩展性，和响应时间。如果想淘宝这种秒杀某类商品，可以容忍超扣的情况，那可以追求可用性和扩展性，一定程度上损失数据一致性。

# 七、JAVA内存模型

## 1.概述

想要JAVA内存模型，必须要理解以下几点：内存屏障、指令重排序、happens-before、as-if-serial语义。具体阐述如下：

## 2.内存屏障。

内存屏障：包括一下几类

LOADLOAD:前面的读对后面的读可见。指令重排序方面的体现：即该指令前面所有的读和后面所有的读不能发生指令重排序。内存处理方面的体现：执行当前线程的CPU，弃掉当前所有的读缓存，重新从主存中加载一次数据。

LOADSTORE：前面的读对后面的写可见。即该指令前面所有的读与后面所有的写，不可以发生指令重排序。内存处理方面的体现：执行当前线程的CPU，弃掉当前所有的读缓存，从主存中重新加载一次数据。

STORESTORE：前面的写对后面的写可见。即该指令前面所有的写操作不可以跟后面所有的写操作发生指令重排序。内存处理方面的体现：执行当前线程的CPU，flush掉当前的写缓存，写到主存中，以保证对后面的写可见。

STORELOAD：前面的写对后面的读可见。即该指令前面所有的写操作不可以跟后面所有的读操作发生指令重排序。内存处理方面的体现：执行当前线程的CPU，flush掉当前的写缓存，写到驻村中，以保证对后面的读可见。

## 3.指令重排序

执行程序是为了提高性能，编译器和处理器常常会对指令做重排序。重排序分为三种类型：

1.编译器优化的重排序。编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序。

2.指令集并行的重排序。现代处理器采用了指令集并行技术（Instruction-Level Parallelism,ILP）,来对多条指令重叠执行。如果不存在数据依赖性，处理器改变语句对应机器指令的执行顺序。

3.内存系统重排序.由于处理器采用读写缓存，这使得加载和写操作看上去是乱序的。

## 4.Happens-Before

从JDK1.5开始，java使用心得JSR-133内存模型。JSR-133使用happens-before来阐述操作之间的内存可见性。在JMM中，如果一个操作的结果对另一个操作可见，那么这两个操作之间必须存在happens-before关系。这里提到的两个操作，既可以在一个线程，也可以再两个线程之间。

规则如下：

1.程序顺序规则。一个线程中的每个操作，都happens-before中的任意后续操作。

2.监视器锁规则。对一个监视器的释放锁要早与随后对该监视器的加锁。

3.volatile变量规则：对一个volatile域的写早于后续对该volatile的读。

4.传递性。A happens-before B, B happens-before C。 则 A happens-before C。

## 5.as-if-serial 语义

在一个程序单线程执行的过程中，无论怎么进行指令重排序，都要保证每一次的执行结果是一致的。