# JAVA基础随心记

## 枚举类型ENUM

枚举类型是JDK1.5以后的新特性，主要用于更加严格的约束变量的类型

### 1.1枚举的应用场景

* 基于枚举能更好的限定变量的取值
* 通常用于定义几个固定的常量值
* 如果枚举在别的类中定义，必须放在第一行

## 内部类

### 实例内部类

* 访问修饰符是任意的
* 类内部，方法外部，没有static修饰
* 依赖于外部类创建
* 内部类调用外部类对象时可以用：外部类.this
* 必须先构造外部类才能构造实例内部类

使用实例内部类时要注意内存泄漏的问题，内存泄漏是导致内存溢出的原因之一

* 当外部类对象不可访问，但是还占用着内存，这个现象称之为内存泄漏
* 当在外部类的方法中调用内部类时，就会可能导致这个内存泄漏
* 同时因为内部类依托于外部类，所以内部类会保存着外部类的引用
* 使用实例内部类时要注意手动回收内部类

### 静态内部类

* 访问修饰符是任意的
* 存在于类内部，方法外部，有static修饰
* 不依托于外部类创建
* 可以访问外部类的静态成员

### 局部类内部类

### 匿名内部类

# 面向对象

## 3.1封装

## 3.2.继承

* 当B类继承A类，如果只是访问A类中的静态成员，则B类中的静态代码块不会执行，但是A类的静态代码块会执行，因为A类属于主动加载，B类属于被动加载PS:eclipce可以通过-XX:+TraceClassLoading这个参数进行查看类的加载过程

## 3.3多态

# 类加载

* 类加载分为显式加载跟隐式加载，加载类的时候，静态代码块可以执行，但不一定会执行
* 显式加载时执行静态代码块的有：Class.forName(“类的全限名”)，但是这个方法有一个重载的方法，第二个参数为布尔类型，如果为false则不会执行静态代码块
* 显式加载不执行静态代码块的有：ClassLoad.getSystemClassLoader().loadClass(“类的全限名”)，就是使用类加载器加载类的时候不会去执行静态代码块
* getClass这个方法是final修饰的，所以不能被重写，要获得父类的类对象只能通过.getClass.getSuperClass()这个方法进行获得

## 反射

* 反射是JAVA中一种特有的技术
* 反射中有一种自省的特性，就是在运行期能发现对象本身的成员
* 是JAVA实现动态编程的一种技术

# 5.GC垃圾回收机制

## Finalize()

这个方法是垃圾回收机制，回收对象之前调用的，重写这个方法，可以用来看对象是否被回收

# 6.JVM内存

## 程序计数器

* 这个内存非常小，可以理解为用于指示当前线程的字节码执行到第几行，唯一个没有内存溢出区域
* 线程私有的

## 栈

* 虚拟机栈

一个线程的每个方法执行时，都会创建一个新的栈帧，栈中保存着有局部变量表、动态链接、方法出口等，当方法被调用时，栈帧入JVM栈，方法执行完成时，弹栈

* 本地方法栈

跟虚拟机栈差不多，唯一的区别就是虚拟机栈是用来执行java方法的，本地方法栈时用来执行native方法的，其实我们默认使用的JDK都是将这两个栈放在一起使用的

* 线程私有的

## 堆

堆是JAVA GC机制最重要的区域之一，因为堆是内存分配主要区域，也是内存最大的区域

### 堆内存的分配

1. 主要有分代分配，分代回收，这个代主要有三个，年轻代，老年代，永久代
2. 当对象被创建时，主要分配在年轻代，CG主要也是工作在这一块区域，年轻代还可以划分成Eden区(伊甸园，你懂的)连续的内存空间，所以分配内存极快,还有两个存活区(survivor0,survivor1)

当伊甸园的内存满了的时候，GC就会清理消亡的对象，将存活的对象存储到其中一个存活区，两个存活区必须有一个是空的，当下次伊甸园的孩子再满的时候，GC同样会清理消亡的对象，然后将存活的对象复制到存活区1中，这时伊甸园是一个不剩的，然后GC会去存活区0中清除已经消亡的对象，将未消亡的对象复制到老年区或者另一个存活区，反正就是要保证其中有一个存活区是空的，这个就是著名的”停止-复制”清理发

1. 老年代，老年代的内存一般比年轻代大，因为有些大对象，而且GC清理的次数也没这么频繁

第二个区别就是清理的算法不一样，如果老年代也用”停止-复制”清理法，相当的低效，所以一般都是”标记-整理”清理法，标记的时还存在引用的对象，将这里对象统一向内存的一端移动，保证内存的联系性，如果加入老年代的对象内存大于剩余的内存就会调用GC进行清理

## 方法区

* 方法区也叫永久代
* 回收的信息也很简单，只有两种：一个是常量池中的常量，一个是无用的类信息。常量的回收很简单，没了引用就回收，但是类信息的回收必须要保证以下三点

1. 类的实例必须全部被回收
2. 类加载器ClassLoader被回收
3. Class对象(类对象没有被引用

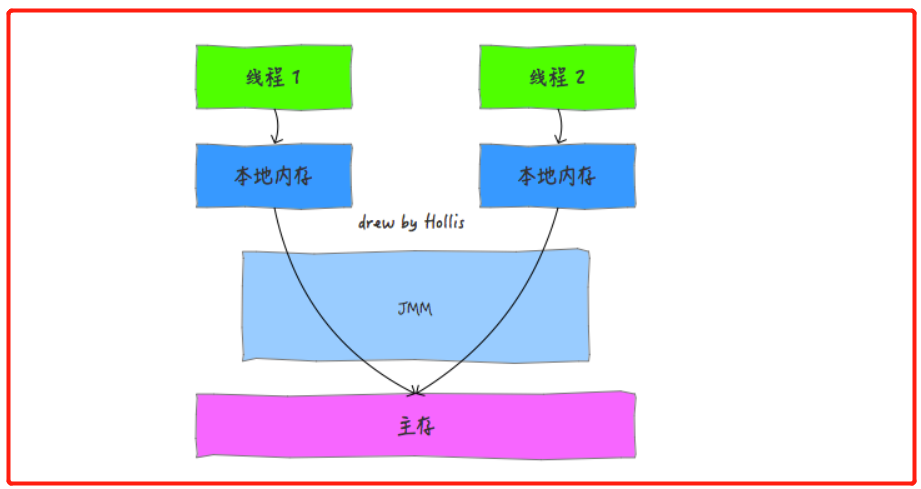
## 备注

在介绍垃圾收集器之前，需要明确一点，就是在新生代采用的停止复制算法中，“停 止（Stop-the-world）”的意义是在回收内存时，需要暂停其他所 有线程的执行。这个是很低效的，现在的各种新生代收集器越来越优化这一点，但仍然只是将停止的时间变短，并未彻底取消停止。

# 7.JMM java内存模型

Java程序是需要运行在Java虚拟机上面的，Java内存模型（JMM）就是一种符合内存模型规范的，屏蔽了各种硬件和操作系统的访问差异的，保证了Java程序在平台下对内存的访问都能保证效果一致的机制及规范

有下图可以得知：Java内存模型规定了所有变量都存在主内存中，每个线程都有自己的工作内存，线程的工作内存中保存了该线程中用到的变量的副本拷贝，线程对变量的操作都必须在工作内存中进行，而不能直接读写主内存。不同线程之间也无法直接访问对方工作内存中的变量，线程间变量的传递均需要自己的工作内存和主存之间进行数据同步进行。



Java 提供了一些关键字Synchronized,volatile就是java内存模型封装了底层的实现后，提供给我们使用的

**Java内存模型，除了定义了一套规范，还提供了一系列原语，封装了底层实现后，供开发者直接使用。**

## 原子性

通过Synchronized 或者ReentrantLock 实现,而Synchronized 实现 原子性操作是通过对象锁，Synchronized 可以作用于方法或者代码块，两者实现的原子性操作不应，通过反编译知道Synchronized 作用于方法时，会 ACC\_SYNCHRONIZED 标记，线程要开始被标记的方法，就需要获取对象的锁，没获得锁的线程就会被阻塞，而作用于代码块的时候，是通过两个命令（monitorentor,monitorexit）监视器实现，相当于计数器的形式，当对象没有被获得锁的时候，计数器为0，有线程获得该对象的锁的时候，计数器加1，可重复获得锁，计数器累加，当线程释放锁时，计数器减1，直至为0

## 可见性

Java内存模型是通过在变量修改后将新值同步回主内存，在变量读取前从主内存刷新变量值的这种依赖主内存作为传递媒介的方式来实现的。

Volatile 这个关键字提供了一个功能，就是工作线程更新被其修饰的变量后，会马上同步到主内存

除了volatile，Java中的synchronized和final两个关键字也可以实现可见性。只不过实现方式不同，这里不再展开了。

## 有序性

volatile关键字会禁止指令重排。因为java编译之后，被优化器优化，代码执行的顺序并不是按我们想象那样执行的，但是被volatile标记之后就会按照我们想要的顺序执行，Synchronized 并不会禁止指令重排

# 8.锁

## 8.1乐观锁

乐观锁在操作数据时非常乐观，认为别人不会同时修改数据。因此乐观锁不会上锁，只是在执行更新的时候判断一下在此期间别人是否修改了数据：如果别人修改了数据则放弃操作，否则执行操作。最常用的就是CAS算法(Compare And Swap(比较并交换),乐观锁因为没有上锁，所以可以提高系统读的性能

### CAS的缺点：

* ABA问题:就是当A线程去操作数据的时候，需要预期值是A，但是在某一刻，线程B去把数据的预期值改为了B，然后再改会A，对于线程A来说，数据时没有变更过的，但实际这样是不安全的，要解决这个问题，可以在变量加上版本号，每次修改，变量加1，变成:1A-2B-3A
* 循环时间长,开销大：如果CAS长时间不成功，会导致一直自旋，给CPU带来非常大的开销
* 只能保证一个共享变量的原子操作：如果要多个，请使用JKD1.5的AtomicReference类

## 8.2 悲观锁

悲观锁在操作数据时，总人会别人会同时修改数据，所以会对资源上锁，在没释放锁的时候，其它线程想要操作都需要等待，实现悲观锁的有Synchronized关键字和Lock的实现类

优点：能够保证每次操作都是原子性的

缺点：降低了系统的吞吐量

## 8.3 自旋锁和适应性自旋锁

线程阻塞会让出CPU的时间片，阻塞或唤醒一个Java线程需要操作系统切换CPU状态来完成，这种状态转换需要耗费处理器时间。如果同步代码块中的内容过于简单，状态转换消耗的时间有可能比用户代码执行的时间还要长。

在许多场景中，同步资源锁定的时间很短，为了这一小段时间去切换线程，线程挂起和恢复线程的花费可能会让系统的不尝试，如果物理机器有多个处理器，能够让两个或以上的线程同时并行执行，我们就可以让后面那个请求锁的线程不放弃CPU的执行时间，看持有锁的线程是否很快就会释放锁

### 缺点：

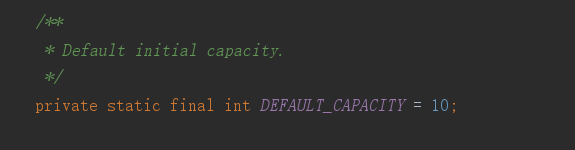
自旋锁本身是有缺点的，它不能代替阻塞。自旋锁虽然避免了线程切换的开销，但它要占用处理器的时间。如果锁被占用的时间很短，自旋锁的效果就会很好。反之，如果锁被占用的时间很长，那么自旋锁就是白白浪费处理器的资源。所以自旋锁等待的时间必须要有一定的限度，如果自旋超过了限定次数（默认是10次，可以使用-XX:PreBlockSpin来更改）没有成功获得锁，就应当挂起线程。自旋锁的实现原理同样也是CAS，现在JDK1.8了默认是适应性自旋锁

## 8.4 无锁、偏向锁、轻量级锁、重量级锁

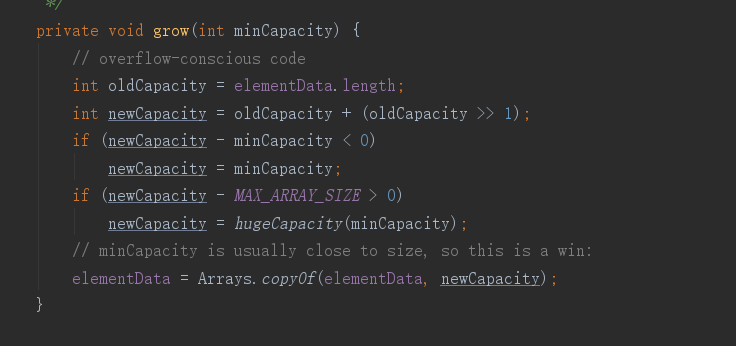
这四种锁是指锁的状态，专门针对synchronized的。在介绍这四种锁状态之前还需要介绍一些额外的知识。

# 9.集合

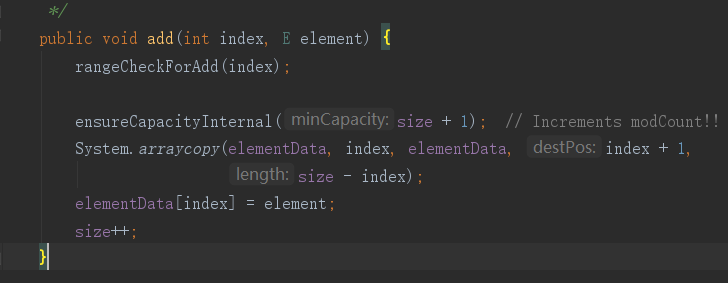
## 8.1ArrayList

底层实现是用Object数组实现的，如果不指定长度，则开始长度为0，添加第一个元素的时候扩展到默认的长度10

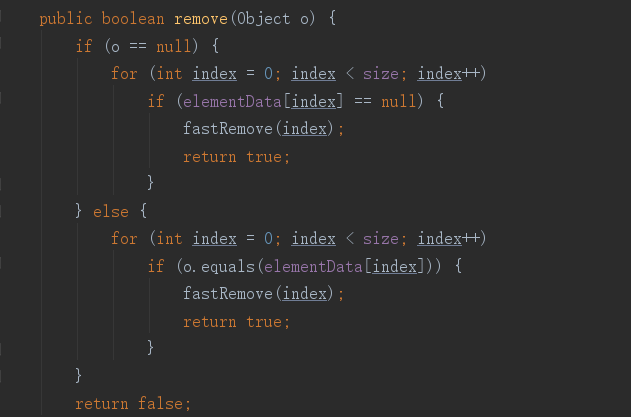
当容量满了的时候会扩展1.5倍，调用的是Arrays.copyOf(elementdata,newSize)方法，所以新建ArrayList集合的时候最好指定长度。因为扩容会降低效率，特别是数据量大的时候

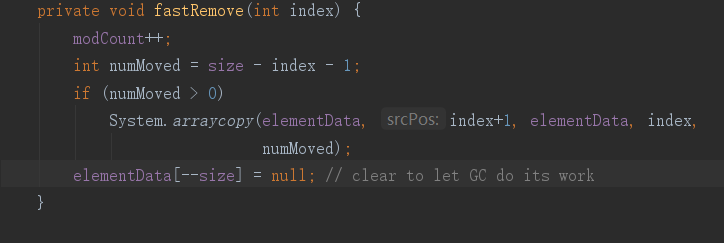


指定位置添加数据的时候add(index,data)，这个方法也是要移动数据调用System.arrayCopt(date,index,date,index+1,size-index)，所以ArrayList添加是比较慢的，而查询快，根据数组下标进行查询



删除元素的时候，也是通过复制移动元素调用的是System.arrayCopy(data,index+1,date,index,size-index-1)，所以删除也是比较慢的

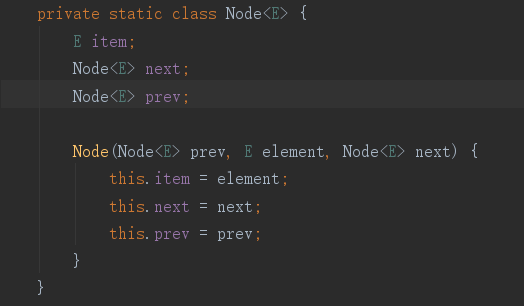




## 8.2LinkedList

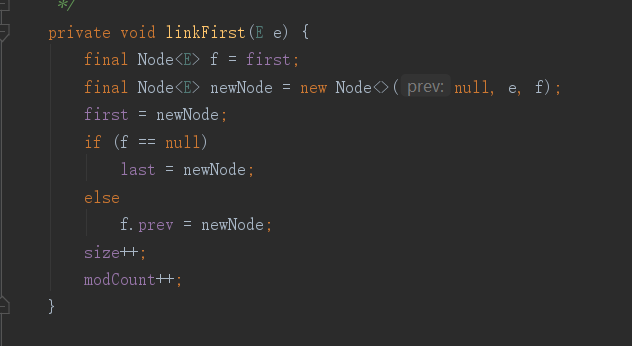
### 8.2.1Node结点

其实就保存了三个值，一个是本身的Node，一个是前驱的Node,一个的后续的Node,双向链表

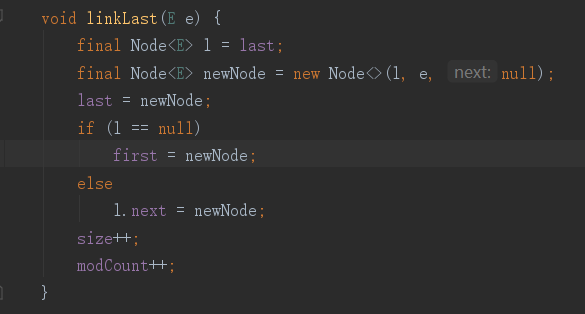


### 8.2.2添加操作：

头部插入，将first保存成为一个临时常量，调用构造方法，为e构造一个Node（构造方法如上图），将e变为first,如果f为空，则是第一个元素，first的前驱指向e。PS:插入元素为头元素时，只需改原头元素的前驱(原头元素的前驱为Null，改为指向插入元素，后续元素原本已指向，所以不用改变)

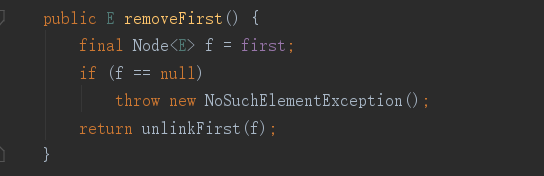


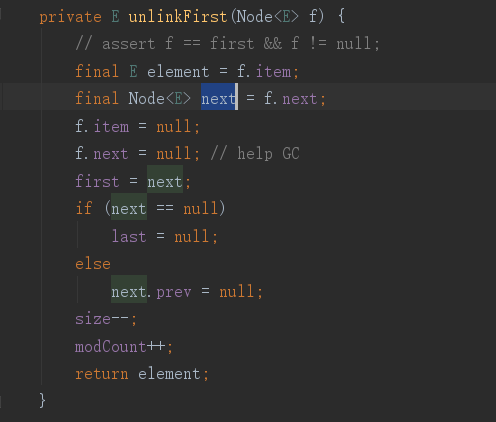
尾部插入：跟头部相反，只需关心原尾部的后续指向



### 8.2.3删除操作：

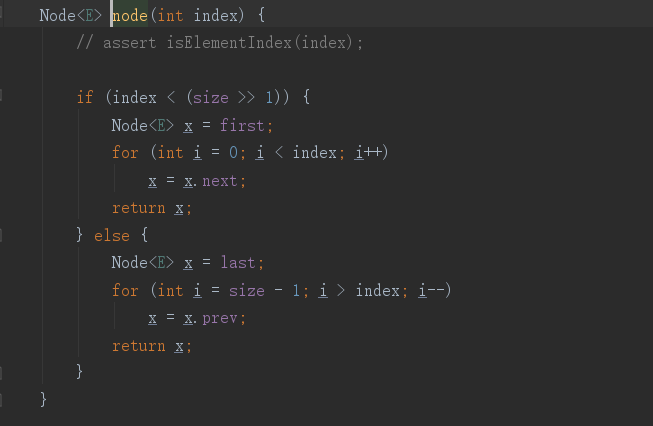
删除头部，首先传进去的是First元素，获得头元素的后续Node,将头元素的本身，后续都赋值为null（GC回收快点）,如果后续元素Node==null,则表示为尾节点，所以Last==null,否则Node前驱赋值为空(因为头节点前驱为null,删除尾部的元素操作相似





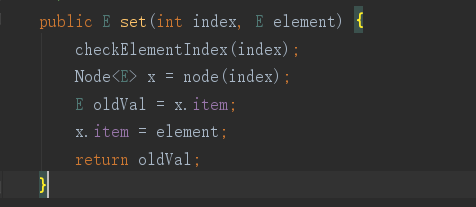
### 8.2.4查找：

判断index的值是否大于size>>2，如果大于则从尾部开始移动，否则从头部开始移动指针，这样可以提高效率



### 8.2.5修改：

也是通过上方的查询操作找到某个节点，然后修改改节点的element的值



## 8.3 HashMap

底层实现是Node数组加链表实现的，初始容量是空的数组，第一次添加元素的时候才会默认新建容量为16的Node数组，加载因子是0.75,阈值为12，每次扩容都会翻倍，当链表长度为8时会转为红黑树**且**容量为64时也会变为红黑树，红黑树长度为6时则转换为链表

### 8.3.1添加元素：

确定数组下标：根据key.hash & table.length-1 确定下标

**流程：**

1. 根据key计算得到key.hash = (h = k.hashCode()) ^ (h >>> 16)；

2. 根据key.hash计算得到桶数组的索引index = key.hash & (table.length - 1)，这样就找到该key的存放位置了：

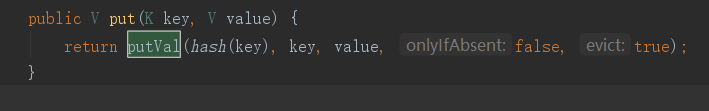
① 如果该位置没有数据，用该数据新生成一个节点保存新数据，返回null；

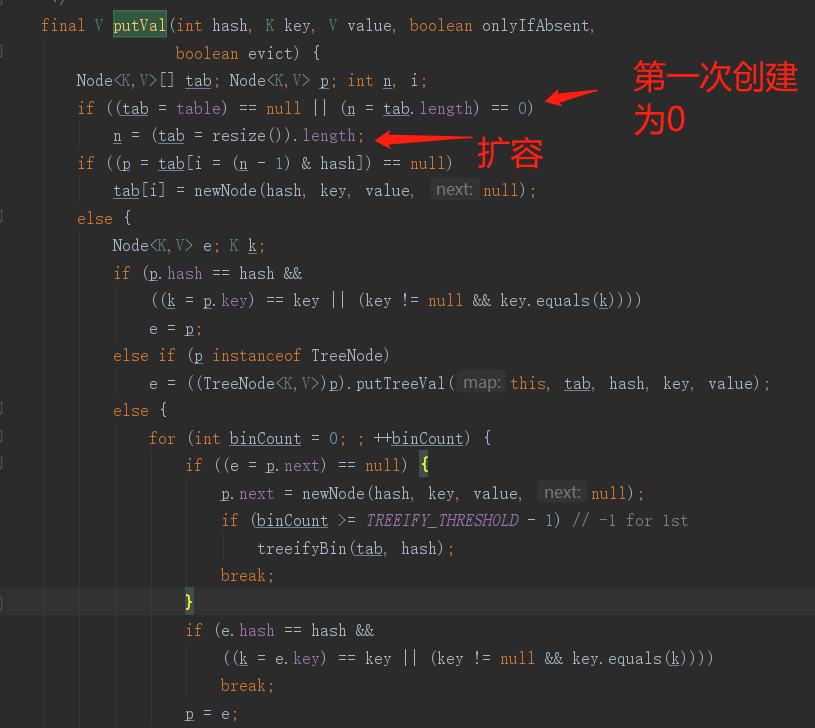
② 如果该位置有数据是一个红黑树，那么执行相应的插入 / 更新操作；

③ 如果该位置有数据是一个链表，分两种情况一是该链表没有这个节点，另一个是该链表上有这个节点，注意这里判断的依据是key.hash是否一样：

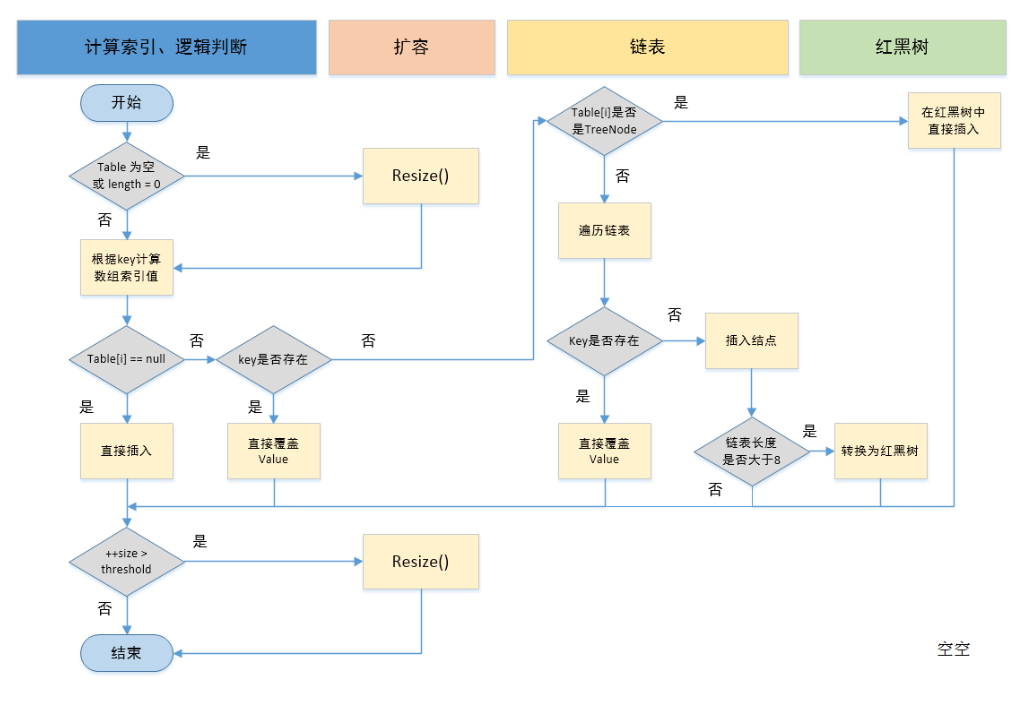
如果该链表没有这个节点，那么采用尾插法新增节点保存新数据，返回null；如果该链表已经有这个节点了，那么找到该节点并更新新数据，返回老数据。

注意：





putVal方法执行过程可以通过下图来理解：



①.判断键值对数组table[i]是否为空或为null，否则执行resize()进行扩容；

②.根据键值key计算hash值得到插入的数组索引i，如果table[i]==null，直接新建节点添加，转向⑥，如果table[i]不为空，转向③；

③.判断table[i]的首个元素是否和key一样，如果相同直接覆盖value，否则转向④，这里的相同指的是hashCode以及equals；

④.判断table[i] 是否为treeNode，即table[i] 是否是红黑树，如果是红黑树，则直接在树中插入键值对，否则转向⑤；

⑤.遍历table[i]，判断链表长度是否大于8，大于8的话把链表转换为红黑树，在红黑树中执行插入操作，否则进行链表的插入操作；遍历过程中若发现key已经存在直接覆盖value即可；

⑥.插入成功后，判断实际存在的键值对数量size是否超多了最大容量threshold，如果超过，进行扩容。

### 8.3.2 查找元素：

**getNode方法**

说明：HashMap同样并没有直接提供getNode接口给用户调用，而是提供的get方法，而get方法就是通过getNode来取得元素的。

首先传入两个参数key.hash,key ，根据这两个值，确定下标，如果该下标有值，则判断Key跟hash是否相同，相同则返回，不相同则找红黑树或者链表

final Node<K,V> getNode(int hash, Object key) {

Node<K,V>[] tab; Node<K,V> first, e; int n; K k;

// table已经初始化，长度大于0，根据hash寻找table中的项也不为空

if ((tab = table) != null && (n = tab.length) > 0 &&

(first = tab[(n - 1) & hash]) != null) {

// 桶中第一项(数组元素)相等

if (first.hash == hash && // always check first node

((k = first.key) == key || (key != null && key.equals(k))))

return first;

// 桶中不止一个结点

if ((e = first.next) != null) {

// 为红黑树结点

if (first instanceof TreeNode)

// 在红黑树中查找

return ((TreeNode<K,V>)first).getTreeNode(hash, key);

// 否则，在链表中查找

do {

if (e.hash == hash &&

((k = e.key) == key || (key != null && key.equals(k))))

return e;

} while ((e = e.next) != null);

}

}

return null;

}

## 8.4ConcurrentHashMap

线程安全的Map集合，相比HashTab性能更加高效，因为HashTab是通过锁表的方式来实现线程安全问题的，只要有人在写，就不能读，而ConcurrentHashMap是通过Unsafe和Synchronized两种方式实现的，在去的sizeCtl、某个位置的Node的时候，使用的都是Unsafe的方法，来达到并发安全的目的。

* 当需要在某个位置设置节点的时候，则会通过Synchronized的同步机制来锁定改位置的的节点
* 在数组扩容的时候，则通过处理的步长和fwd节点来达到并发安全的目的，通过设置hash只为MOVED
* 当把某个位置的节点复制到扩张后的table的时候，也通过Synchronied的同步机制来保证线程安全

**下面是几个重要的属性：**

private static final int MAXIMUM\_CAPACITY = 1 << 30;

private static final int DEFAULT\_CAPACITY = 16;

static final int TREEIFY\_THRESHOLD = 8;

static final int UNTREEIFY\_THRESHOLD = 6;

static final int MIN\_TREEIFY\_CAPACITY = 64;

static final int MOVED = -1; // 表示正在转移

static final int TREEBIN = -2; // 表示已经转换成树

static final int RESERVED = -3; // hash for transient reservations

static final int HASH\_BITS = 0x7fffffff; // usable bits of normal node hash

transient volatile Node<K,V>[] table;//默认没初始化的数组，用来保存元素

private transient volatile Node<K,V>[] nextTable;//转移的时候用的数组

/\*\*

\* 用来控制表初始化和扩容的，默认值为0，当在初始化的时候指定了大小，这会将这个大小保存在sizeCtl中，大小为数组的0.75

\* 当为负的时候，说明表正在初始化或扩张，

\* -1表示初始化

\* -(1+n) n:表示活动的扩张线程

\*/

private transient volatile int sizeCtl;

### 三个原子操作的方法

// 获取tab数组的第i个node

static final <K,V> Node<K,V> tabAt(Node<K,V>[] tab, int i) {  
 return (Node<K,V>)U.getObjectVolatile(tab, ((long)i << ASHIFT) + ABASE);  
}  
// 利用CAS算法设置i位置上的node节点。在CAS中，会比较内存中的值与你指定的这个值是否相等，如果相等才接受  
static final <K,V> boolean casTabAt(Node<K,V>[] tab, int i,  
 Node<K,V> c, Node<K,V> v) {  
 return U.compareAndSwapObject(tab, ((long)i << ASHIFT) + ABASE, c, v);  
}  
// 利用volatile方法设置第i个节点的值，这个操作一定是成功的。  
static final <K,V> void setTabAt(Node<K,V>[] tab, int i, Node<K,V> v) {  
 U.putObjectVolatile(tab, ((long)i << ASHIFT) + ABASE, v);  
}

### 8.4.1添加元素：

首先判断是否为数组是否为空，为空则初始化数组，利用unsafe类判断该数组位置是否为空，为空则直接插入也是利用Unsafe类，不为空，则用Synchronied（f）,抢添加元素的锁，