**第四页，NIO，BIO，AIO的区别没有梳理。Java的流对象没有梳理。**

**Java源码到机器码的过程：**

Java源代码---->编译器---->jvm可执行的Java字节码(即虚拟指令面向虚拟机)---->jvm---->jvm中解释器----->机器可执行的二进制机器码---->程序运行。

**什么是Java程序的主类？**

一个程序中可以有多个类，但只能有一个类是主类（包含main()方法的类），是程序执行的入口

**Java基本数据类型：**



**枚举：**

1. 用于常量
2. 用于switch

3**.**public enum MyDay {

MONDAY(1,"星期一"),THUSDAY(2,"星期二");//这个后面必须有分号

private int code;

private String name;

private MyDay(int code,String name) {

this.code = code;

this.name = name();

}

}可以添加方法，相当于public final class enumTest.MyDay extends java.lang.Enum<enumTest.MyDay>

枚举参考网站：<https://www.cnblogs.com/qlqwjy/p/9065264.html>

**访问修饰符**

Private，protected不能修饰外部类和接口，其他两种可以



**final:**

final修饰的变量必须被初始化，否则编译期错误

**Super与this:**

1.this是自身的一个对象，代表对象本身，可以理解为：指向对象本身的一个指针。super可以理解为是指向自己超（父）类对象的一个指针，而这个超类指的是离自己最近的一个父类。

2.从本质上讲，this是一个指向本对象的指针, 然而super是一个Java关键字。

3.super()和this()均需放在构造方法内第一行并且this和super不能同时出现在一个构造函里面，因为子类实例化会调用父类构造方法。

**static存在的主要意义：**

static的主要意义是在于创建独立于具体对象的域变量或者方法。以致于即使没有创建对象，也能使用属性和调用方法！

static关键字还有一个比较关键的作用就是 用来形成静态代码块以优化程序性能。static块可以置于类中的任何地方，类中可以有多个static块。在类初次被加载的时候，会按照static块的顺序来执行每个static块，并且只会执行一次。为什么说static块可以用来优化程序性能，是因为它的特性:只会在类加载的时候执行一次。因此，很多时候会将一些只需要进行一次的初始化操作都放在static代码块中进行。

**Static的加载：**

在该类被第一次加载的时候，就会去加载被static修饰的部分。static变量值在类加载的时候分配空间，以后创建类对象的时候不会重新分配。赋值的话，是可以任意赋值的

**面向对象的三大特性：**

封装：封装把一个对象的属性私有化，同时提供一些可以被外界访问的属性的方法，

可以控制对类的属性的获取

继承：继承是使用已存在的类的定义作为基础建立新类的技术，新类的定义可以增加新的数据或新的功能，也可以用父类的功能，但不能选择性地继承父类。通过使用继承我们能够非常方便地复用以前的代码。

多态：多态就是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编程时并不确定，而是在程序运行期间才确定，即一个引用变量到底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。

在Java中有两种形式可以实现多态：继承类和实现接口

方法重载（overload）实现的是编译时的多态性而方法重写（override）实现的是运行时的多态性

Java实现多态有三个必要条件：继承、重写、向上转型

**面向对象五大基本原则是什么**

单一职责原则SRP(Single Responsibility Principle)  
类的功能要单一，不能包罗万象，跟杂货铺似的。

开放封闭原则OCP(Open－Close Principle)  
一个模块对于拓展是开放的，对于修改是封闭的

里式替换原则LSP(the Liskov Substitution Principle LSP)  
子类可以替换父类出现在父类能够出现的任何地方

依赖倒置原则DIP(the Dependency Inversion Principle DIP)  
高层次的模块不应该依赖于低层次的模块，他们都应该依赖于抽象。抽象不应该依赖于具体实现，具体实现应该依赖于抽象。

接口分离原则ISP(the Interface Segregation Principle ISP)

设计时采用多个与特定客户类有关的接口比采用一个通用的接口要好。

**抽象类和接口的对比：**

相同点：

接口和抽象类都不能实例化

都位于继承的顶端，用于被其他实现或继承

都包含抽象方法，其子类都必须覆写这些抽象方法

不同点：



备注：抽象类不能用final修饰。

**成员变量与局部变量的区别有哪些**：

作用域（对象，代码块），存储位置（堆，栈），声明周期（对象，代码块），初始值（有，无）。使用原则：在使用变量时需要遵循的原则为：就近原则，首先在局部范围找，有就使用；接着在成员位置找。

在调用子类构造方法之前会先调用父类没有参数的构造方法，其目的是？帮助子类做初始化工作。

内部类：

内部类本身就是类的一个属性，与其他属性定义方法一致。

内部类可以分为四种：

成员内部类：

定义在类内部，成员位置上的非静态类。成员内部类可以访问外部类所有的变量和方法，包括静态和非静态，私有和公有。成员内部类依赖于外部类的实例

局部内部类：

定义在方法中的内部类，定义在实例方法中的局部类可以访问外部类的所有变量和方法，定义在静态方法中的局部类只能访问外部类的静态变量和方法。

匿名内部类：

没有名字的内部类，匿名内部类必须继承一个抽象类或者实现一个接口。匿名内部类不能定义任何静态成员和静态方法。当所在的方法的形参需要被匿名内部类（和局部内部类）使用时，必须声明为 final。匿名内部类不能是抽象的，它必须要实现继承的类或者实现的接口的所有抽象方法

静态内部类

定义在类内部的静态类。

内部类的优点

**我们为什么要使用内部类呢？因为它有以下优点：**

1.一个内部类对象可以访问创建它的外部类对象的内容，包括私有数据！

2.内部类不为同一包的其他类所见，具有很好的封装性；

3.内部类有效实现了“多重继承”（可以方便的使用外部类的方法），优化 java 单继承的缺陷。

4.匿名内部类可以很方便的定义回调。

使用场景：当某个类除了它的外部类，不再被其他的类使用时

**Java中只有值传递：**

一个方法不能修改一个基本数据类型的参数

一个方法可以改变一个对象参数的状态。

一个方法不能让对象参数引用一个新的对象。

**BIO，NIO和AIO 的区别和适应场景：**

NIO是通过内核线程轮询的方式来解决阻塞的问题

参考：https://thinkwon.blog.csdn.net/article/details/104390612

**Java的流对象：**

**反射：**

JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。

**反射是框架的灵魂**

**网络编程：**

参考网站：<https://blog.csdn.net/ThinkWon/article/details/104903925>

**String相关：**

字符串常量池：

字符串常量池位于堆内存中，专门用来存储字符串常量，可以提高内存的使用率，避免开辟多块空间存储相同的字符串，在创建**字符串（不通过new的方式创建）**时 JVM 会首先检查字符串常量池，如果该字符串已经存在池中，则返回它的引用，如果不存在，则实例化一个字符串放到池中，并返回其引用。

String类底层就是一个final类型的字符数组。

注：string对象有length()方法跟数组不一样。整型字面量的值在-128到127之间，那么自动装箱时不会new新的Integer对象，而是直接引用常量池中的Integer对象

**线程安全性**：String每次改变都会创建一个新的对象，所以其线程安全。StringBuffer加了同步锁线程安全，StringBuilder线程不安全。

**三者使用场景：**

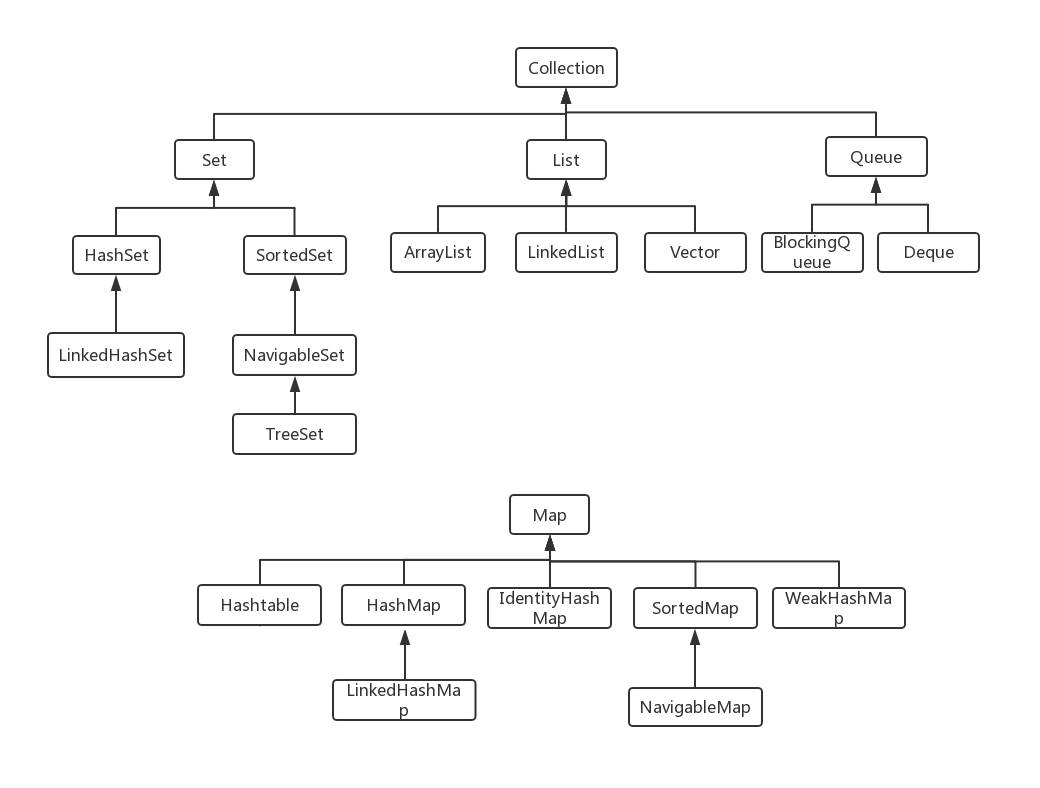
如果要操作少量的数据用 = String

单线程操作字符串缓冲区 下操作大量数据 = StringBuilder

多线程操作字符串缓冲区 下操作大量数据 = StringBuffer

**集合面试题：**

**集合的继承图：**



**Map接口和Collection接口是所有集合框架的父接口：**

Collection接口的子接口包括：Set接口和List接口

Map接口的实现类主要有：HashMap、TreeMap、Hashtable、ConcurrentHashMap以及Properties等

Set接口的实现类主要有：HashSet、TreeSet、LinkedHashSet等

List接口的实现类主要有：ArrayList、LinkedList、Stack以及Vector（比arraylist多了sychronizes，线程安全）等

HashMap：

JDK1.8之前HashMap由数组+链表组成的，数组是HashMap的主体，链表则是主要为了解决哈希冲突而存在的（“拉链法”解决冲突）.JDK1.8以后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为8）时，将链表转化为红黑树，以减少搜索时间

LinkedHashMap：LinkedHashMap 继承自 HashMap，所以它的底层仍然是基于拉链式散列结构即由数组和链表或红黑树组成。另外，LinkedHashMap 在上面结构的基础上，增加了一条双向链表，使得上面的结构可以保持键值对的插入顺序。同时通过对链表进行相应的操作，实现了访问顺序相关逻辑。

HashTable： 数组+链表组成的，数组是HashMap的主体，链表则是主要为了解决哈希冲突而存在的（相比hashmap，Hashtable 中的方法是Synchronize的）

Jdk1.7头插法形成环形链表的原理：

以a---b---c---d为例，因为是扩容时也是头插法，假设元素的下标位置没有变化，则变成了d---c---b---a。假设另一个扩容线程在之前已经拿到了a作为链表的表头元素，就会将a指向table[i],而这时table[i]中放置的是b,c,d都有可能。这时候就会形成循环链表。

Jdk采用尾插法put元素和扩容（关键点在于线程扩容时，会将table[i]直接置为空值）：不会形成循环链表。

Jdk1.7扩容的同时put数据时造成数据丢失。

Jdk1.8扩容的同时put数据时造成数据插入位置出错。

TreeMap： 红黑树（自平衡的排序二叉树）

**Java集合的快速失败机制 “fail-fast”？：**

是java集合的一种错误检测机制，当多个线程对集合进行结构上的改变的操作时，有可能会产生 fail-fast 机制。List类中维持了一个modcount变量，涉及到对list修改的操作会改变该值。而迭代器遍历元素得到元素时检查该值是否改变，如果改变则抛出异常。

确保一个集合不会被修改：Collections. unmodifiableCollection(Collection c) 方法可以创建一个只读集合，任何改变集合的操作都会抛出异常。

**小知识**：Java Collections 框架中提供了一个 RandomAccess 接口，用来标记 List 实现是否支持随机访问，即通过索引来访问。

**ArrayList在添加元素和删除元素**时因为某个index空缺会调用元素复制的方法，重新调整元素位置，这个方法调用的本地native方法。

**ArrayList 和 LinkedList 的区别是什么？**

1.数据结构实现：ArrayList 是动态数组的数据结构实现，而 LinkedList 是双向链表的数据结构实现。

2.随机访问效率：ArrayList 比 LinkedList 在随机访问的时候效率要高，因为 LinkedList 是线性的数据存储方式，所以需要移动指针从前往后依次查找。

3.增加和删除效率：在非首尾的增加和删除操作，LinkedList 要比 ArrayList 效率要高，因为 ArrayList 增删操作要影响数组内的其他数据的下标。

4.内存空间占用：LinkedList 比 ArrayList 更占内存，因为 LinkedList 的节点除了存储数据，还存储了两个引用，一个指向前一个元素，一个指向后一个元素。

5.线程安全：ArrayList 和 LinkedList 都是不同步的，也就是不保证线程安全

**ArrayList 和 Vector 的区别是什么？**

这两个类都实现了 List 接口（List 接口继承了 Collection 接口），他们都是有序集合

1.线程安全：Vector 使用了 Synchronized 来实现线程同步，是线程安全的，而 ArrayList 是非线程安全的。

2.性能：ArrayList 在性能方面要优于 Vector。

3.扩容：ArrayList 和 Vector 都会根据实际的需要动态的调整容量，只不过在 Vector 扩容每次会增加 1 倍，而 ArrayList 只会增加 50%

**多线程下怎么保证ArrayList的线程安全？**

**为什么 ArrayList 的 elementData 加上 transient 修饰？**

**HashMap 的实现原理：存值，取值的过程（先比较hash再比较对象是否相等；怎么根据hash获取数组下标）；底层数据结构（数组加链表）**

**Hashmap在1.7和1.8有什么不同？：**

数组的特点是：寻址容易，插入和删除困难；链表的特点是：寻址困难，但插入和删除容易；所以我们将数组和链表结合在一起，发挥两者各自的优势。使用链表来解决hash冲突。

**JDK1.8主要解决或优化了一下问题：**

1.resize 扩容优化

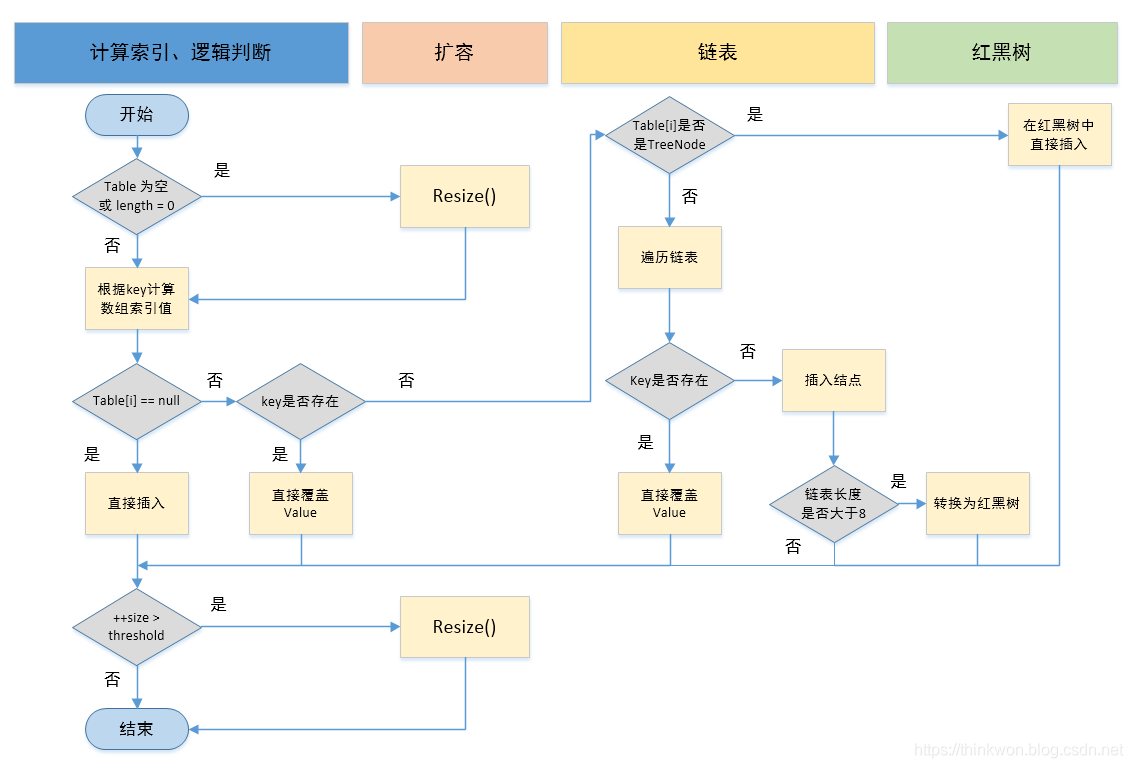
2.引入了红黑树，目的是避免单条链表过长而影响查询效率，红黑树算法请参考（在大于8的时候从红黑树转化成链表）

3.解决了多线程死循环问题，但仍是非线程安全的，多线程时可能会造成数据丢失问题



hash方法实际是让key.hashCode()与key.hashCode()>>>16进行异或操作，高16bit补0，一个数和0异或不变，所以 hash 函数大概的作用就是：**高16bit不变，低16bit和高16bit做了一个异或，目的是减少碰撞**。按照函数注释，因为bucket数组大小是2的幂，计算下标index = (table.length - 1) & hash，如果不做 hash 处理，相当于散列生效的只有几个低 bit 位。（**扩容的阈值为0.75**）

（底层数组初始化，是否转化成红黑树，是否扩容，是否覆盖，下标计算）



**HashMap的扩容操作：**

1.在jdk1.8中，resize方法是在hashmap中的键值对大于阀值时或者初始化时，就调用resize方法进行扩容；

2.每次扩展的时候，都是扩展2倍；

3.扩展后Node对象的位置要么在原位置，要么移动到原偏移量两倍的位置。

在putVal()中，我们看到在这个函数里面使用到了2次resize()方法，resize()方法表示的在进行第一次初始化时会对其进行扩容，或者当该数组的实际大小大于其临界值值(第一次为12),这个时候在扩容的同时也会伴随的桶上面的元素进行重新分发，这也是JDK1.8版本的一个优化的地方，在1.7中，扩容之后需要重新去计算其Hash值，根据Hash值对其进行分发，但在1.8版本中，则是根据在同一个桶的位置中进行判断(e.hash & oldCap)是否为0，重新进行hash分配后，该元素的位置要么停留在原始位置，要么移动到原始位置+增加的数组大小这个位置上。**这里需要看源码！！！！**

**HashMap：**

1. 使用链地址法（使用散列表）来链接拥有相同hash值的数据；  
   2. 使用2次扰动函数（hash函数）来降低哈希冲突的概率，使得数据分布更平均；  
   3. 引入红黑树进一步降低遍历的时间复杂度，使得遍历更快。

**为什么HashMap中String、Integer这样的包装类适合作为K**？

1. 都是final类型，即不可变性，保证key的不可更改性，不会存在获取hash值不同的情况String。这些值不改变的的类一般都会有一个hash成员属性来保存自己的hashcode.

**HashMap 的长度为什么是2的幂次方：**

为了能让 HashMap 存取高效（利用hash值计算数组下标时计算效率高）。与length-1进行&运算和取模操作一样，但是效率更高，另外在resize之后元素下标会变化，计算resize之后元素的新的下标更方便。

**为什么是两次扰动呢？**

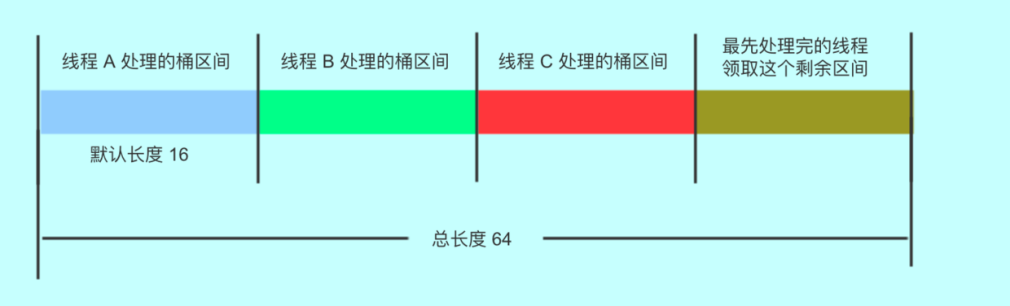
加大哈希值低位的随机性，使得分布更均匀，从而提高对应数组存储下标位置的随机性&均匀性，最终减少Hash冲突，两次就够了，已经达到了高位低位同时参与运算的目的。

ConcurrentHashMap:

ConcurrentHashMap有几个核心：（在HashMap的基础上加上了并发控制，在put之前会进行是否在扩容和初始化的判断）

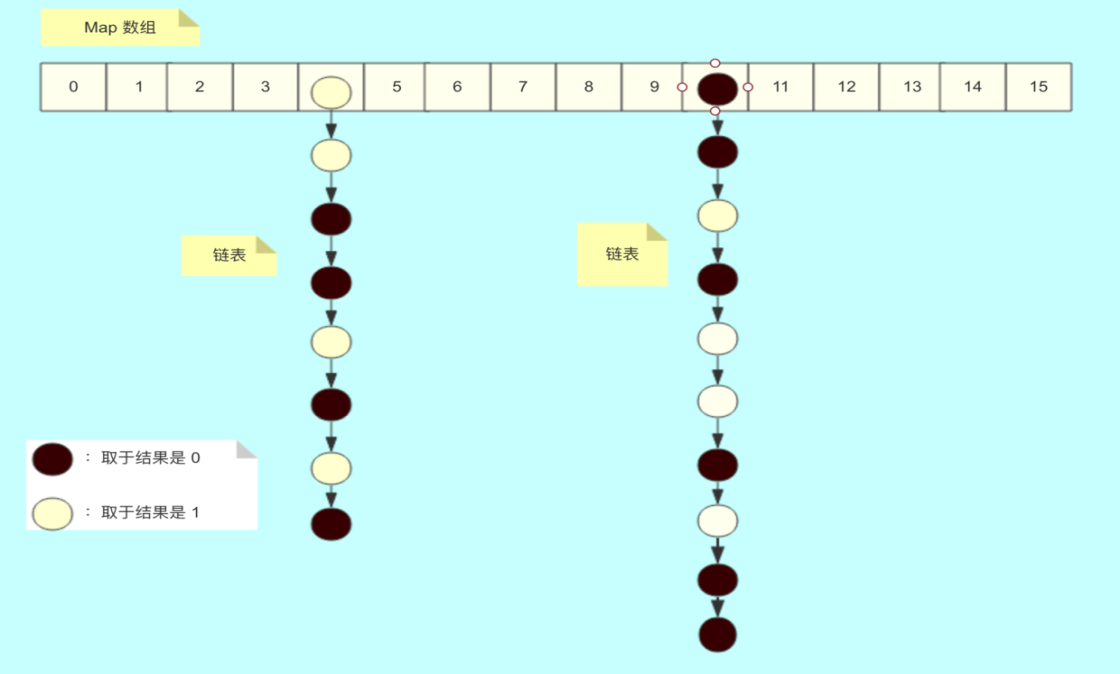
1. 添加元素之前会进行table是否为空的判断，为空则初始化数组长度为16.在初始化前通过sizeCtl用来表示是否正有其他线程在进行初始化或者扩容操作
2. 添加元素时一定会用到cas操作或者对下标为i的元素加锁。（在下标i处为null时cas,不为null时加锁）
3. 不为null又有是否正在进行元素扩容和转移的判断（hash为-1，其在**该下标转移完毕后放了一个forwardNode节点**，其hash为-1），如果正在进行元素转移操作，如果是，就帮助进行元素转移，然后返回新的table数组。不为null且没有进行元素转移，则进行判断下标i所对应的是链表是红黑树。**红黑树用TreeBin表示，其hash为-2**，**并且其first属性放置的是红黑树的第一个节点。**
4. 在put成功后判断是否应该将链表转化成红黑树，在该链表大于等于8并且数组长度不小于64的情况下转化成红黑树。否则进行扩容处理。
5. 之后进入**addcount()方法**,**这里面会计算元素个数时候达到阈值，如果达到阈值则进行扩容并转移元素**。**其中每次进行put或者remove操作都会对basecount或者countCell数组进行cas操作以便进行元素的统计**。计算元素的总个数会对baseCount和countCell数组进行加法操作。

扩容的核心代码流程图：



**备注**：这里这个图有一点问题，其先处理下标最大的元素。

而每个线程在处理自己桶中的数据的时候，是下图这样的：



**每个元素要么在原位，要么在原位下标+旧数组长度。**

**备注：元素在transfer元素时只是对该下标元素加了锁，也就是说在transfer元素时，线程可以在其他下标处添加元素**

**具体流程见该文件夹的concurrenthashMap.java文件中的注释进一步理解细节。**