集合类知识点

集合类

1、有哪些集合类

Java集合框架主要包括两种类型的容器：

- 集合（Collection），存储一个元素集合
- 图（Map），存储键/值对映射。

Collection接口又有3种子类型，List、Set和Queue，再下面是一些抽象类，最后是具体实现类，常用的有ArrayList、LinkedList、HashSet、LinkedHashSet、HashMap、LinkedHashMap等等。

List接口

List接口扩展自Collection，它可以定义一个允许重复的有序集合

1、ArrayList

ArrayList从源码角度分析可以知道，它是用数组存储元素的，这个数组可以动态创建，如果元素个数超过了数组的容量，那么就创建一个更大的新数组，并将当前数组中的所有元素都复制到新数组中。

2、ArrayList的扩容方式

ArrayList 底层结构是数组，每次在add()一个元素时，ArrayList都需要对这个list的容量进行一个判断。如果容量够，直接添加，否则需要进行扩容。在1.8 ArrayList这个类中，扩容调用的是grow()方法，通过grow()方法中调用的Arrays.copyof()方法进行对原数组的复制，再通过调用System.arraycopy()方法进行复制，达到扩容的目的。

ArrayList默认容量是10，**如果初始化时一开始指定了容量，或者通过集合作为元素，则容量为指定的大小或参数集合的大小。每次扩容为原来的1.5倍，如果新增后超过这个容量，则容量为新增后所需的最小容量。**如果增加0.5倍后的新容量超过限制的容量，则用所需的最小容量与限制的容量进行判断，超过则指定为Integer的最大值，否则指定为限制容量大小。然后通过数组的复制将原数据复制到一个更大(新的容量大小)的数组。

3、ArrayList的线程安全性

ArrayList不是线程安全的：在多个线程进行add操作时可能会导致elementData数组越界；一个线程的值覆盖另一个线程添加的值。

一个 ArrayList ，在添加一个元素的时候，它可能会有两步来完成：

1. 在 elementData[size] 的位置存放此元素；
2. 增大 Size 的值。

在单线程运行的情况下，如果 Size = 0，添加一个元素后，此元素在位置 0，而且 Size=1； 而如果是在多线程情况下，比如有两个线程，线程 A 先将元素存放在位置 0。但是此时 CPU 调度线程A暂停，线程 B 得到运行的机会。线程B也向此 ArrayList 添加元素，因为此时 Size 仍然等于 0 （注意哦，我们假设的是添加一个元素是要两个步骤哦，而线程A仅仅完成了步骤1），所以线程B也将元素存放在位置0。然后线程A和线程B都继续运行，都增加 Size 的值。 那好，现在我们来看看 ArrayList 的情况，元素实际上只有一个，存放在位置 0，而 Size 却等于 2。这就是“线程不安全”了。

4、LinkedList

LinkedList的源文件分析可以知道，LinkedList是在一个链表中存储元素。所以，LinkedList的元素添加和删除其实就对应着链表节点的添加和移除。

5、Vector

Vector是特殊的集合类，用法上Vector与ArrayList基本一致，不同之处在于Vector使用了关键字synchronized将访问和修改向量的方法都变成同步的了，所以对于不需要同步的应用程序来说，类ArrayList比类Vector更高效。

6、ArrayList和LinkedList的区别以及优缺点

ArrayList和LinkedList都是实现了List接口的容器类，用于存储一系列的对象引用。他们都可以对元素的增删改查进行操作。

ArrayList和LinkedList的大致区别：

1. ArrayList是实现了基于动态数组的数据结构，LinkedList是基于链表结构。
2. 对于随机访问的get和set方法，ArrayList要优于LinkedList，因为LinkedList要移动指针。
3. 对于新增和删除操作add和remove，LinkedList比较占优势，因为ArrayList要移动数据。

对元素的增删查操作的时候，进行查操作时用ArrayList，进行增删操作的时候最好用LinkedList。

他们在性能上的有缺点：

1. 对ArrayList和LinkedList而言，在列表末尾增加一个元素所花的开销都是固定的。对 ArrayList而言，主要是在内部数组中增加一项，指向所添加的元素，偶尔可能会导致对数组重新进行分配；而对LinkedList而言，这个开销是 统一的，分配一个内部Entry对象。
2. 在ArrayList集合中添加或者删除一个元素时，当前的列表所所有的元素都会被移动。而LinkedList集合中添加或者删除一个元素的开销是固定的。
3. LinkedList集合不支持高效的随机随机访问（RandomAccess），因为可能产生二次项的行为。
4. ArrayList的空间浪费主要体现在list列表的结尾预留一定的容量空间，而LinkedList的空间花费则体现在它的每一个元素都需要消耗相当的空间

7、ArrayList、LinkedList以及Vector的区别联系

- ArrayList 本质上是一个可改变大小的数组.当元素加入时,其大小将会动态地增长.内部的元素可以直接通过get与set方法进行访问.元素顺序存储 ,随机访问很快，删除非头尾元素慢，新增元素慢而且费资源 ,较适用于无频繁增删的情况 ,比数组效率低，如果不是需要可变数组，可考虑使用数组 ,非线程安全.
- LinkedList 是一个双链表,在添加和删除元素时具有比ArrayList更好的性能.但在get与set方面弱于ArrayList. 适用于 ： 没有大规模的随机读取，有大量的增加/删除操作.随机访问很慢，增删操作很

快，不耗费多余资源 ,允许null元素,非线程安全.

- Vector （类似于ArrayList）但其是同步的，开销就比ArrayList要大。如果你的程序本身是线程安全的，那么使用ArrayList是更好的选择。 Vector和ArrayList在更多元素添加进来时会请求更大的空间。Vector每次请求其大小的双倍空间，而ArrayList每次对size增长50%。

Map接口

HashMap

1、JDK1.7和1.8 HashMap区别

在JDK1.7中：

使用一个Entry数组来存储数据，首先会用到一个Hash函数计算元素key的hash值，以此确定插入数组中的位置index，但是可能存在同一hash值的元素已经被放在数组同一位置了，这时就添加到同一hash值的元素的后面，他们在Entry数组的同一位置，这些key会形成一个链表。

在特别差的情况下，比方说所有key的hash值都相同，这个链表可能会很长，那么put/get操作都可能需要遍历这个链表。

也就是说时间复杂度在最差情况下会退化到O(n)

在JDK1.8中：

使用一个Node数组来存储数据，但这个Node可能是链表结构，也可能是红黑树结构

如果插入的key的hash值相同，那么这些key也会被定位到Node数组的同一个格子里。

如果同一个格子里的key不超过8个，使用链表结构存储。

如果超过了8个，那么会调用treeifyBin函数，将链表转换为红黑树。

那么即使hash值完全相同，由于红黑树的特点，查找某个特定元素，也只需要O(logn)的开销

也就是说put/get的操作的时间复杂度最差只有O(logn)

听起来挺不错，但是真正想要利用JDK1.8的好处，有一个限制：

key的对象，必须**正确的实现了Compare接口**

2、HashMap底层数据结构

HashMap采用数组+链表实现，HashMap是一个用于存储Key-Value键值对的集合，每一个键值对也叫做Node。这些个键值对（Node）分散存储在一个数组当中，这个数组就是HashMap的主干，实际上它的每一个元素都是链表，当添加一个元素（key-value）时，就首先计算元素key的hash值，以此确定插入数组中的位置index，但是可能存在同一hash值的元素已经被放在数组同一位置了，这时就添加到同一hash值的元素的后面，他们在数组的同一位置，但是形成了链表，所以说数组存放的是链表。新来的Entry节点插入链表时，使用的是“头插法”，因为HashMap的发明者认为，后插入Entry被查找的可能性更大。当链表长度超过8时,就转为红黑树,红黑树的增删改查较快,就提高了hashmap的性能，这样大大提高了查找的效率。

3、链表与红黑树的转换

在`JDK1.8`中，`HashMap`底层是用数组`Node`数组存储，数组中每个元素用链表存储元素，当元素超过`8`个时，将链表转化成红黑树存储。

红黑树

红黑树本质上是平衡查找二叉树，用于存储有序数据，相对于链表数据查找来说，链表的时间复杂度`O(n)`，红黑树的时间复杂度`O(lgn)`。

红黑树需要满足一下五种特性：

- 每个节点是红色或者黑色
- 根节点是黑色
- 每一个空叶子节点必须是黑色
- 红色节点的子节点必须是黑色
- 节点中到达任意子节点包含相同数组的黑节点

当新增节点或者减少节点，红黑树会通过左旋或者右旋操作来调整树结构，使其满足以上特性。

TreeNode 类

`HashMap`中红黑树是通过`TreeNode`类构造的。`TreeNode`是`HashMap`的静态内部类，继承于`LinkedHashMap.Entry`类

treeifyBin 方法

在`HashMap`中`put`方法时候，但数组中某个位置的链表长度大于`8`时，会调用`treeifyBin`方法将链表转化为红黑树。

构成红黑树--treeify 方法

`treeify`方法是`TreeNode`类的方法，作用是将`Treenode`链转化成红黑树。

新增元素后平衡红黑树--balanceInsertion方法

当红黑树中新增节点的时候需要调用`balanceInsertion`方法来保证红黑树的特性。

4、put()方法实现原理

比如调用 hashMap.put("apple", 0) ，插入一个Key为“apple"的元素。这时候我们需要利用一个哈希函数来确定Node的插入位置（index）：

但是，因为HashMap的长度是有限的，当插入的键值对越来越多时，再完美的Hash函数也难免会出现index冲突的情况。比如下面这样：

HashMap数组的每一个元素不止是一个Entry对象，也是一个链表的头节点。每一个Entry对象通过Next指针指向它的下一个Entry节点。当新来的Entry映射到冲突的数组位置时，只需要插入到对应的链表即可：

新来的节点插入链表时，使用的是“头插法”。因为HashMap的发明者认为，后插入Entry被查找的可能性更大。

5、get()方法实现原理

使用Get方法根据Key来查找Value

首先会把输入的Key做一次Hash映射，得到对应的index，由于刚才所说的Hash冲突，同一个位置有可能匹配到多个Node，这时候就需要顺着对应链表的头节点，一个一个向下来查找。

第一步，我们查看的是头节点，如果头结点的Key是不是我们要找的结果。

第二步，我们就查看的是Next节点，直到找到我们要的结果。

6、HashMap扩容方式

HashMap的容量是有限的。当经过多次元素插入，使得HashMap达到一定饱和度时，Key映射位置发生冲突的几率会逐渐提高。这时候，HashMap需要扩展它的长度，也就是进行Resize。HashMap的默认初始长度是16，并且每次自动扩展或是手动初始化时，长度必须是2的幂。

影响发生Resize的因素有两个：

1.Capacity

HashMap的当前长度。HashMap的长度是2的幂。

2.LoadFactor

HashMap负载因子，默认值为0.75f。

衡量HashMap是否进行Resize的条件如下：

HashMap.Size >= Capacity * LoadFactor

HashMap的Resize不是简单的把长度扩大，而是经过下面两个步骤：

1.扩容

创建一个新的Entry空数组，长度是原数组的2倍。

2.ReHash

遍历原Entry数组，把所有的Entry重新Hash到新数组。为什么要重新Hash呢？因为长度扩大以后，Hash的规则也随之改变。

7、HashMap扩容是否重新计算Hashcode

是的，因为长度扩大以后，Hash的规则也随之改变。让我们回顾一下Hash公式：

index = HashCode（Key） & （Length - 1）

当原数组长度为8时，Hash运算是和111B做与运算；新数组长度为16，Hash运算是和1111B做与运算。Hash结果显然不同。

8、HashMap扩容发生死锁，不是线程安全的

HashMap 的死锁问题就出在这个`transfer()`函数上。

Hashmap不是线程安全的

Hashmap的Resize包含扩容和ReHash两个步骤，ReHash在并发的情况下可能会形成链表环。

假设一个HashMap已经到了Resize的临界点。此时有两个线程A和B，在同一时刻对HashMap进行Put操作，链表会出现了环形的现象，当调用Get查找一个不存在的Key，而这个Key的Hash结果恰好等于这个index的时候，由于位置带有环形链表，所以程序将会进入死循环！

- Hashmap在插入元素过多的时候需要进行Resize，Resize的条件是

  HashMap.Size >= Capacity * LoadFactor。
- Hashmap的Resize包含扩容和ReHash两个步骤，ReHash在并发的情况下可能会形成链表环。

9、hash值的计算

hash ： 该方法主要是将Object转换成一个整型。

在JDK1.8的实现中，优化了高位运算的算法，通过hashCode()的高16位异或低16位实现的：(h = k.hashCode()) ^ (h >>> 16)，主要是从速度、功效、质量来考虑的。以上方法得到的int的hash值，然后再通过`h & (table.length -1)`来得到该对象在数据中保存的位置。

要实现一个尽量均匀分布的Hash函数，需要通过利用Key的Hash值来做某种运算。为了实现高效的Hash算法，HashMap的发明者采用了位运算的方式。

index = Hash（Key） & （Length - 1）

这样做不但效果上等同于取模，而且还大大提高了性能。长度16或者其他2的幂，Length-1的值是所有二进制位全为1，这种情况下，index的结果等同于HashCode后几位的值。只要输入的HashCode本身分布均匀，Hash算法的结果就是均匀的。

10、HashMap初始容量和负载因子

HashMap的默认初始长度是16，HashMap负载因子，默认值为0.75f。

11、为什么因子是0.75

默认负载因子（0.75）在时间和空间成本上提供了很好的折衷。提高空间利用率和减少查询成本的折中，主要是泊松分布，0.75的话碰撞最小，

加载因子过高，例如为1，虽然减少了空间开销，提高了空间利用率，但同时也增加了查询时间成本；

加载因子过低，例如0.5，虽然可以减少查询时间成本，但是空间利用率很低，同时提高了rehash操作的次数。

LinkedHashMap

HashMap是无序的，当我们希望有顺序地去存储key-value时，就需要使用LinkedHashMap了。当put元素时，不但要把它加入到HashMap中去，还要加入到双向链表中，所以可以看出LinkedHashMap就是HashMap+双向链表。LinkedHashMap不会去改变HashMap节点的性质，LinkedHashMap所做的，只是建立节点之间的联系，LinkedHashMap的增删改查操作，只是在HashMap的增删改查的基础上，进行head和tail指针的移动。

HashMap与LinkedHashMap的结构对比

LinkedHashMap其实就是可以看成HashMap的基础上，多了一个双向链表来维持顺序。

- LinkedHashMap是继承于HashMap，是基于HashMap和双向链表来实现的。

- HashMap无序；LinkedHashMap有序，可分为插入顺序和访问顺序两种。如果是访问顺序，那put和get操作已存在的Entry时，都会把Entry移动到双向链表的表尾(其实是先删除再插入)。
- LinkedHashMap存取数据，还是跟HashMap一样使用的Entry[]的方式，双向链表只是为了保证顺序。
- LinkedHashMap是线程不安全的。

ConcurrentHashMap

1、JDK1.7和JDK1.8区别

在JDK1.7中ConcurrentHashMap采用了ReentrantLock+Segment+HashEntry的方式实现。

JDK1.8版本中synchronized+CAS+HashEntry+红黑树实现。

- 数据结构：取消了Segment分段锁的数据结构，取而代之的是数组+链表+红黑树的结构。
- 保证线程安全机制：JDK1.7采用segment的分段锁机制实现线程安全，其中segment继承自ReentrantLock。JDK1.8采用CAS+Synchronized保证线程安全。
- 锁的粒度：原来是对需要进行数据操作的Segment加锁，现调整为对每个数组元素加锁（Node）。
- 链表转化为红黑树:定位结点的hash算法简化会带来弊端,Hash冲突加剧,因此在链表节点数量大于8时，会将链表转化为红黑树进行存储。
- 查询时间复杂度：从原来的遍历链表O(n)，变成遍历红黑树O(logN)。

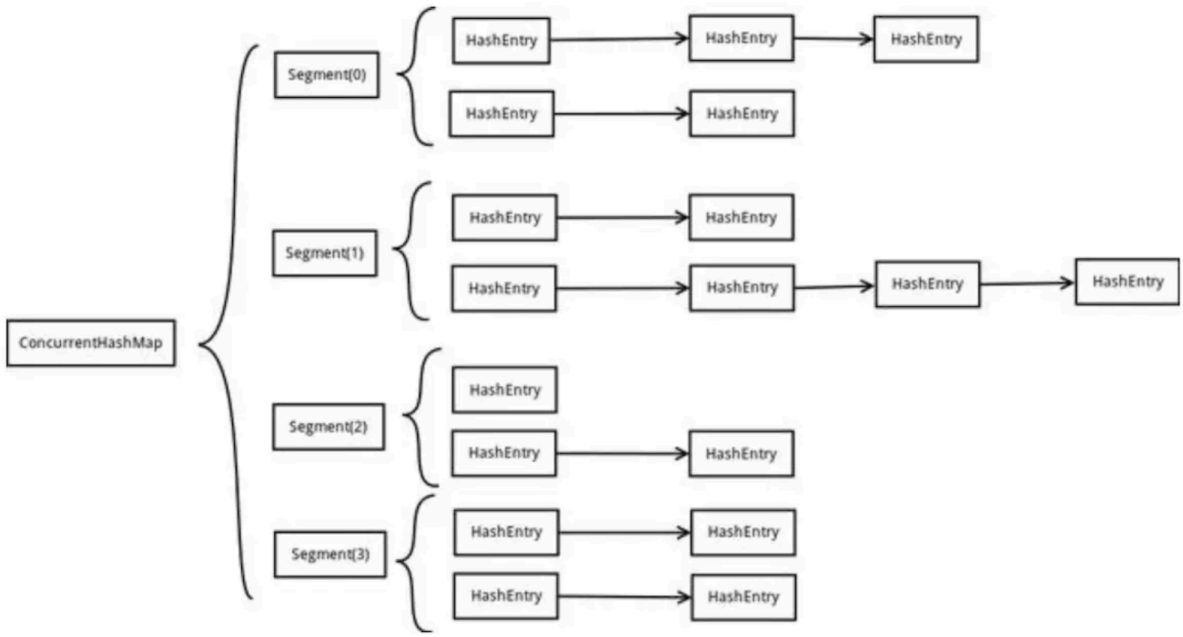
2、JDK1.7版本的CurrentHashMap的实现原理

1.Segment(分段锁)

ConcurrentHashMap中的分段锁称为Segment，它即类似于HashMap的结构，即内部拥有一个Entry数组，数组中的每个元素又是一个链表,同时又是一个ReentrantLock（Segment继承了ReentrantLock）。

2.内部结构

ConcurrentHashMap使用分段锁技术，将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问，能够实现真正的并发访问。如下图是ConcurrentHashMap的内部结构图：



从上面的结构我们可以了解到，ConcurrentHashMap定位一个元素的过程需要进行两次Hash操作。

第一次Hash定位到Segment，第二次Hash定位到元素所在的链表的头部。

3.该结构的优劣势

坏处 这一种结构的带来的副作用是Hash的过程要比普通的HashMap要长

好处 写操作的时候可以只对元素所在的Segment进行加锁即可，不会影响到其他的Segment，这样，在最理想的情况下，ConcurrentHashMap可以最高同时支持Segment数量大小的写操作（刚好这些写操作都非常平均地分布在所有的Segment上）。

所以，通过这一种结构，ConcurrentHashMap的并发能力可以大大的提高。

总结：

ConcurrentHashMap使用分段锁技术Segment，它即类似于HashMap的结构，即内部拥有一个Entry数组，数组中的每个元素又是一个链表,同时又是一个ReentrantLock（Segment继承了ReentrantLock）。分段锁技术将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问，能够实现真正的并发访问。

ConcurrentHashMap定位一个元素的过程需要进行两次Hash操作。第一次Hash定位到Segment，第二次Hash定位到元素所在的链表的头部。这一种结构的带来的副作用是Hash的过程要比普通的HashMap要长，但是写操作的时候可以只对元素所在的Segment进行加锁即可，不会影响到其他的Segment，并发能力可以大大的提高。

3、JDK1.8版本的CurrentHashMap的实现原理

JDK1.8的实现已经摒弃了Segment的概念，而是直接用Node数组+链表+红黑树的数据结构来实现，并发控制使用Synchronized和CAS来操作，整个看起来就像是优化过且线程安全的HashMap

CAS是compare and swap的缩写，即我们所说的比较交换。cas是一种基于锁的操作，而且是乐观锁。

CAS算法实现无锁化的修改值的操作，他可以大大降低锁代理的性能消耗。这个算法的基本思想就是不断地去比较当前内存中的变量值与你指定的一个变量值是否相等，如果相等，则接受你指定的修改的值，否则拒绝你的操作。因为当前线程中的值已经不是最新的值，你的修改很可能会覆盖掉其他线程修改的结果。这一点与乐观锁，SVN的思想是比较类似的。

> 在java中锁分为乐观锁和悲观锁。悲观锁是将资源锁住，等一个之前获得锁的线程释放锁之后，下一个线程才可以访问。而乐观锁采取了一种宽泛的态度，通过某种方式不加锁来处理资源，比如通过给记录加version来获取数据，性能较悲观锁有很大的提高。

JDK8中彻底放弃了Segment转而采用的是Node，其设计思想也不再是JDK1.7中的分段锁思想。

Node：保存key，value及key的hash值的数据结构。其中value和next都用volatile修饰，保证并发的可见性。

Java8 ConcurrentHashMap结构基本上和Java8的HashMap一样，不过保证线程安全性。JDK8中ConcurrentHashMap在链表的长度大于某个阈值的时候会将链表转换成红黑树进一步提高其查找性能。JDK1.8版本的ConcurrentHashMap的数据结构已经接近HashMap，相对而言，ConcurrentHashMap只是增加了同步的操作来控制并发.

总结：

Java8 ConcurrentHashMap结构基本上和Java8的HashMap一样，不过保证线程安全性。JDK1.8的实现已经摒弃了Segment的概念，而是直接用Node数组+链表+红黑树的数据结构来实现，并发控制使用Synchronized和CAS来操作。JDK8中采用的是Node，它的value和next都用volatile修饰，保证并发的可见性。

CAS是compare and swap的缩写，即我们所说的比较交换。实现无锁化的修改值的操作，他可以大大降低锁代理的性能消耗。这个算法的基本思想就是不断地去比较当前内存中的变量值与你指定的一个变量值是否相等，如果相等，则接受你指定的修改的值，否则拒绝你的操作。CAS是一种基于锁的操作，而且是乐观锁。

4、ConcurrentHashMap扩容机制

jdk8中，采用多线程无锁扩容。多线程无锁扩容的关键就是通过CAS设置sizeCtl与transferIndex变量，协调多个线程对table数组中的node进行迁移。通过CAS设置sizeCtl属性，多线程之间，以volatile的方式读取sizeCtl属性，来判断ConcurrentHashMap当前所处的状态。

当往hashMap中成功插入一个key/value节点时，有可能触发扩容动作

1、如果新增节点之后，所在链表的元素个数达到了阈值 **8**，则会调用`treeifyBin`方法把链表转换成红黑树，不过在结构转换之前，会对数组长度进行判断，如果数组长度n小于阈值`MIN_TREEIFY_CAPACITY`，默认是64，则会调用`tryPresize`方法把数组长度扩大到原来的两倍，并触发`transfer`方法，重新调整节点的位置。

2、新增节点之后，会调用`addCount`方法记录元素个数，并检查是否需要进行扩容，当数组元素个数达到阈值时，会触发`transfer`方法，重新调整节点的位置。

5、ConcurrentHashMap线程安全的原理

JDK1.7采用segment的分段锁机制实现线程安全，其中segment继承自ReentrantLock。JDK1.8采用CAS+Synchronized保证线程安全。

具体如上。

HashTable

1、HashTable和HashMap的联系与区别

联系：

1. 都实现了Map接口，保存了Key-Value（键值对）
2. 两者的数据结构类似。HashMap和HashTable都是由数组元素为链表头节点的数组组成。

区别：

1. 两者继承的父类不同；

> HashMap继承AbstractMap类，而HashTable继承Dictionary类。

2. HashMap是线程不安全的，而HashTable是线程安全的；

> HashTable中的方法是Synchronize的；可以在多并发环境下直接使用HashTable；

3. 提供contains：

> 1、HashMap没有contians，只有containsKey和containsValue，认为contians会容易引起误解；
>
> 2、HashTable保留了contains，containsValue和containsKey方法。（但contains和containsValue的功能是相同的）

4. 对key和value的空值允许情况：

> HashTable中key和value都不允许为null；
>
> HashMap中空值可以作为Key，也可以有一个/多个Key的值为空值；

5. HashMap和HashTable内部的数组初始化和扩容方式也不相同

> 1、HashMap的hash数组默认长度大小为16，扩容方式为2的指数：
>
> length_HashMap = 16 * 2^n（n为扩容次数）
>
> 2、HashTable的hash数组默认长度大小为11，扩容方式为两倍加一：
>
> length_HashTable = 上一次HashTable数组长度 * 2 + 1

6. Hash值计算不一样，HashMap采用的是位运算，而HashTable采用的是直接取模。

2、HashTable底层数据结构

HashTable类继承自`Dictionary`类，实现了三个接口，分别是`Map`，`Cloneable`和`java.io.Serializable`， HashTable的主要方法的源码实现逻辑，与HashMap中非常相似，有一点重大区别就是所有的操作都是通过`synchronized`锁保护的。只有获得了对应的锁，才能进行后续的读写等操作。

put方法

1. 先获取`synchronized`锁。
2. put方法不允许`null`值，如果发现是`null`，则直接抛出异常。
3. 计算`key`的哈希值和index
4. 遍历对应位置的链表，如果发现已经存在相同的hash和key，则更新value，并返回旧值。
5. 如果不存在相同的key的Entry节点，则调用`addEntry`方法增加节点。
6. `addEntry`方法中，如果需要则进行扩容，之后添加新节点到链表头部。

get方法

1. 先获取`synchronized`锁。
2. 计算key的哈希值和index。
3. 在对应位置的链表中寻找具有相同hash和key的节点，返回节点的value。
4. 如果遍历结束都没有找到节点，则返回`null`。

reHash扩容

1. 数组长度增加一倍（如果超过上限，则设置成上限值）。
2. 更新哈希表的扩容门限值。
3. 遍历旧表中的节点，计算在新表中的index，插入到对应位置链表的头部。

3、HashTable并发性能

Hashtable的方法都是声明为synchronized的，这样就能够保证Hashtable是线程安全的。

Hashtable的所有操作都会锁住整个对象，虽然能够保证线程安全，但是性能较差；

TreeMap

1、TreeMap和HashMap联系

TreeMap基于红黑树数据结构的实现，键值可以使用Comparable或Comparator接口来排序。TreeMap继承自AbstractMap，同时实现了接口NavigableMap，而接口NavigableMap则继承自SortedMap。SortedMap是Map的子接口，使用它可以确保图中的条目是排好序的。

TreeMap的优点是可以顺序遍历元素。只有需要顺序遍历元素时才使用TreeMap否则使用HasnMap。

1. HashMap 非线程安全 TreeMap 非线程安全，都继承了AbstractMap
2. HashMap：基于哈希表实现，TreeMap：基于红黑树实现
3. HashMap：适用于在Map中插入、删除和定位元素，Treemap：适用于按自然顺序或自定义顺序遍历键(key)
4. HashMap通常比TreeMap快一点(树和哈希表的数据结构使然)，建议多使用HashMap，在需要排序的Map时候才用TreeMap
5. HashMap的结果是没有排序的。TreeMap实现**SortMap**接口，能够把它保存的记录根据键排序,默认是按键值的升序排序，也可以指定排序的比较器，当用Iterator遍历**TreeMap**时，得到的记录是排过序的。

2、TreeMap底层数据结构

TreeMap实现了SotredMap接口，它是有序的集合。而且是一个红黑树结构，每个key-value都作为一个红黑树的节点。存入TreeMap的元素应当实现Comparable接口或者实现Comparator接口，会按照排序后的顺序迭代元素，如果在调用TreeMap的构造函数时没有指定比较器，则根据key执行自然排序。

3、TreeMap有序性保持

TreeMap基于红黑树数据结构的实现，键值可以使用Comparable或Comparator接口来排序。TreeMap继承自AbstractMap，同时实现了接口NavigableMap，而接口NavigableMap则继承自SortedMap。SortedMap是Map的子接口，使用它可以确保图中的条目是排好序的。

LinkedHashMap会存储数据的插入顺序，是进入时有序；TreeMap则是默认key升序，是进入后有序

Set接口

HashSet

1、HashSet与HashMap的联系

HashSet是实现Set接口的一个实体类，里面的不能包含重复数据。

HashMap实现了Map接口，Map接口对键值对进行映射。Map中不允许重复的键。Map接口有两个基本的实现，

区别

HashMap	***HashSet***
HashMap实现了Map接口	HashSet实现了Set接口
HashMap储存键值对	HashSet仅仅存储对象
使用put()方法将元素放入map中	使用add()方法将元素放入set中
HashMap中使用键对象来计算hashcode值	HashSet使用成员对象来计算hashcode值，对于两个对象来说hashcode可能相同，所以equals()方法用来判断对象的相等性，如果两个对象不同的话，那么返回false
HashMap比较快，因为是使用唯一的键来获取对象	HashSet较HashMap来说比较慢

2、HashSet底层数据结构和去重原理

HashSet是一个用于实现Set接口的具体类，可以使用它的无参构造方法来创建空的散列集，也可以由一个现有的集合创建散列集。

通过HashSet的源码实现可以看到它内部是使用一个HashMap来存放元素的，因为HashSet的元素就是其内部HashMap的**键集合**，所以HashSet可以做到元素不重复。

TreeSet

1、TreeSet与HashSet联系

TreeSet可以用于元素进行排序及保证元素唯一，可以用过对象实现Comparable接口，或者new TreeSet的时候传入自定义的比较方法，实现自定义的排序。

区别：

1. TreeSet背后的结构是TreeMap,也就是红黑树，能够实现自动排序。通过equals和compareTo方法进行内容的比较。
2. HashSet背后是HashMap，key是无序的，只能做外部排序。既然是Hash,那么就要重写其对象的hashCode和equals方法。

- HashSet可以接受null值，有且只有一个
- TreeSet默认不可以接受null值，会直接抛出空指针异常

2、底层数据结构和有序性的保持

TreeSet底层则采用一个NavigableMap来保存TreeSet集合的元素。但实际上，由于NavigableMap只是一个接口，因底层依然是使用TreeMap来包含Set集合中的所有元素。

TreeSet底层数据结构是红黑树，是一种自平衡的二叉树。该自平衡二叉树保证了元素的有序性（存储逻辑顺序），因为按照前、中、后三种顺序都可以有序的读取到集合中的元素。

通过观察TreeSet的底层源码发现，TreeSet的add(E e)方法，底层是根据实现Comparable的方式来实现的唯一性，通过compare(Object o)的返回值是否为0来判断是否为同一元素。 compare() == 0，元素不入集合。 compare() > 0 ,元素入右子树。 compare() < 0,元素入左子树。 而对其数据结构：自平衡二叉树做前（常用）、中、后序遍历即可保证TreeSet的有序性。

queue

1、有什么队列

Java中对于队列的实现分为**非阻塞**和**阻塞**两种。阻塞队列与普通队列的区别在于，当队列是空的时，从队列中获取元素的操作将会被阻塞，或者当队列是满时，往队列里添加元素的操作会被阻塞。

1）**ArrayDeque**，（数组双端队列）

2）**PriorityQueue**，（优先级队列）

3）**ConcurrentLinkedQueue**，（基于链表的并发队列）

4）**DelayQueue**，（延期阻塞队列）（阻塞队列实现了**BlockingQueue**接口）

5）**ArrayBlockingQueue**，（基于数组的并发阻塞队列）

6）**LinkedBlockingQueue**，（基于链表的**FIFO**阻塞队列）

7）**LinkedBlockingDeque**，（基于链表的**FIFO**双端阻塞队列）

8）**PriorityBlockingQueue**，（带优先级的无界阻塞队列）

9）**SynchronousQueue**，（并发同步阻塞队列）