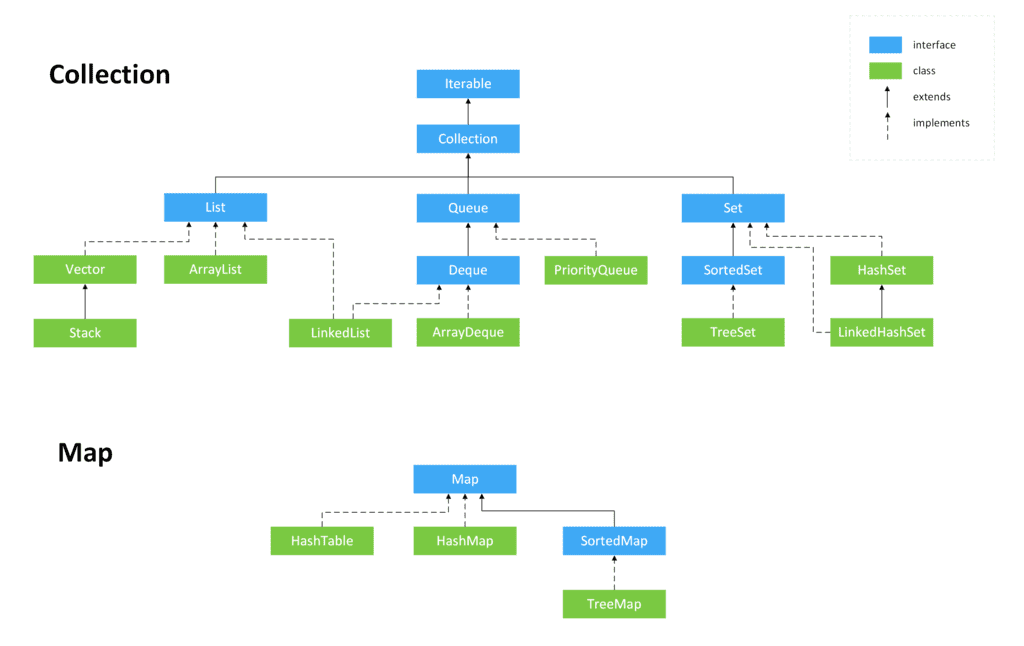
容器

容器主要包括 Collection 和 Map 两种，Collection 存储着对象的集合，而 Map 存储着键值对(两个对象)的映射表。

常见集合的关系框架



Part 1---List接口体系

List接口体系的整体特点：有序，可重复，有索引

1. ArrayList

* 特点：有序，可重复，有索引
* 非线程安全
* 底层实现：基于动态数组实现-----底层是Object数组，能容纳任何类型对象
* 默认初始容量大小：10
* 自动扩容：minCapacity＜实际元素个数时，通过ensureCapacity方法实现
* 负载因子：无，minCapacity＜实际元素个数时就扩容
* 扩容**增量**：原来容量的0.5倍左右（原为偶数，为0.5倍；若为奇数，0.5倍丢掉小数）
* 时间复杂度： size(), isEmpty(), get(), set()方法均能在常数时间内完成，add()方法的时间开销跟插入位置有关，addAll()方法的时间开销跟添加元素的个数成正比。其余方法大都是线性时间。

注意：

（1）ArrayList由于实现了RandomAccess接口，支持随机访问

（2）jdk8以无参构造创建ArrayList时，初始化时为空数组，往数组添加第一个元素时，数组容量扩为10。而jdk6以无参创建时，直接创建长为10的数组。

（3）向 ArrayList 添加大量元素之前最好先使用ensureCapacity 方法，以减少增量重新分配的次数。

（4）ArrayList采用了快速失败机制---Fail-Fast机制

（5）对于get方法，要注意的是由于底层数组是Object[]，得到元素后需要进行类型转换。

1. Vector---List的古老实现类，与ArrayList特性差不多

* 特点：与ArrayList一样，有序，可重复，有索引
* 线程安全！！！
* 底层实现：与ArrayList一样，Object数组
* 负载因子：无，空间不足就扩容
* 扩容增量：原来容量的1倍

1. LinkedList

特点：有序，可重复，有索引

非线程安全

底层实现：基于双向链表

实现接口：List接口、Queue接口、Deque接口

时间复杂度：首尾增删为常数时间，所有跟下标相关的操作都是线性时间

注意：

1. 为追求效率LinkedList没有实现同步(synchronized)，如果需要多个线程并发访问，可以先采用**Collections.synchronizedList()**方法对其进行**包装**。
2. Node是其私有的内部类
3. 删除remove 具体实现是“unlink”操作，就是修改引用，使得要删节点前后指空
4. 只能顺序访问
5. 可以快速地在链表中间插入和删除元素
6. 可以用作栈、队列和双向队列。但实际应用上，关于栈或队列，现在的首选是**ArrayDeque**，它有着比LinkedList(当作栈或队列使用时)有着更好的性能。

ArrayDeque 与 LinkedList 的区别

ArrayDeque 和 LinkedList 都实现了 Deque 接口，两者都具有队列的功能，但两者有什么区别呢？

* ArrayDeque 是基于**可变长的数组**和**双指针**来实现，而 LinkedList 则通过**链表**来实现。
* ArrayDeque 不支持存储 NULL 数据，但 LinkedList 支持。
* ArrayDeque 是在 **JDK1.6** 才被引入的，而LinkedList 早在 JDK1.2 时就已经存在。
* ArrayDeque 插入时可能存在扩容过程, 不过均摊后的插入操作依然为 O(1)。虽然 LinkedList 不需要扩容，但是每次插入数据时均需要申请新的堆空间，均摊性能相比更慢。

从性能的角度上，选用 ArrayDeque 来实现队列要比 LinkedList 更好。此外，ArrayDeque 也可以用于实现栈

==========================================================

Queue接口与Deque接口（继承Queue）体系

Queue和Deque的区别：

Queue 是单端队列，只能从一端插入元素，另一端删除元素，实现上一般遵循 先进先出（FIFO） 规则。



Deque 是双端队列，在队列的两端均可以插入或删除元素。Deque继承Queue接口，除了支持Queue的方法之外，还支持insert, remove和examine操作，由于Deque是双向的，所以可以对队列的头和尾都进行操作



如前面所说：Deque 还提供有 push() 和 pop() 等其他方法，可用于模拟栈。

1. LinkedList---实现List接口、Queue接口、Deque接口
2. ArrayDeque---实现Deque接口

特点：

* 底层实现：基于**可变长的数组**和**双指针**来实现。
* 与LinkedList一样，都是非线程安全
* 可以实现**栈**、队列、双向队列。
* 不支持存储Null

ArrayDeque 与 LinkedList 的区别

ArrayDeque 和 LinkedList 都实现了 Deque 接口，两者都具有队列的功能，但两者有什么区别呢？

* ArrayDeque 是基于可变长的数组和双指针来实现，而 LinkedList 则通过链表来实现。
* ArrayDeque 不支持存储 NULL 数据，但 LinkedList 支持。
* ArrayDeque 是在 JDK1.6 才被引入的，而LinkedList 早在 JDK1.2 时就已经存在。
* ArrayDeque 插入时可能存在扩容过程, 不过均摊后的插入操作依然为 O(1)。虽然 LinkedList 不需要扩容，但是每次插入数据时均需要申请新的堆空间，均摊性能相比更慢。

从性能的角度上，选用 ArrayDeque 来实现队列要比 LinkedList 更好。此外，ArrayDeque 也可以用于实现栈。

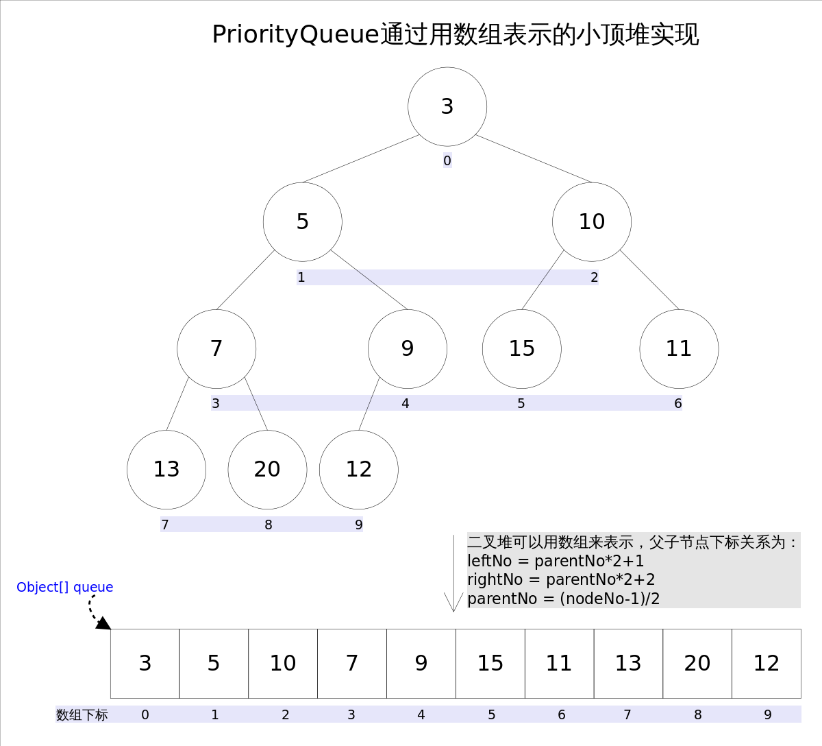
Java有Stack类，但已不推荐使用，现在推荐使用更高效的ArrayDeque实现。Deque中相关实现stack的方法：



1. PriorityQueue（优先队列）---实现Queue接口

特点：

* 优先队列的作用是能保证每次取出的元素都是队列中权值最小的(Java的优先队列每次取最小元素，C++的优先队列每次取最大元素)。
* 时间复杂度：PriorityQueue的**peek()和element操作是常数时间**，**add(), offer(), 无参数的remove()以及poll()方法的时间复杂度都是log(N)**。
* 非线程安全
* 元素大小的评判可**以通过元素本身的自然顺序**(natural ordering)，**也可以通过构造时传入的比较器(Comparator)**实现
* 不允许存入null
* 通过**堆**实现（具体来说是**通过完全二叉树实现的小顶堆**）
* 底层通过**数组**实现---为什么？看下去



父子节点的编号之间有如下关系:

leftNo = parentNo\*2+1

rightNo = parentNo\*2+2

parentNo = (nodeNo-1)/2

通过上述三个公式，可以轻易计算出某个节点的父节点以及子节点的下标。所以能用数组存储堆

方法分析：

（1）add（）和offer（）---插入元素可能会破坏小顶堆的性质，要进行调整：

调整的过程为: 从k指定的位置开始，将x逐层与当前点的parent进行比较并交换，直到满足x >= queue[parent]为止\*\*。注意这里的比较可以是元素的自然顺序，也可以是依靠比较器的顺序。

（2）element和peek都是获取但不删除队首元素，按照小顶堆性质，队首元素就应该是数组下标为0处的元素。

（3）remove和poll，获取并删除队首元素，会破坏小顶堆的性质，要进行调整：

将叶子节点最小元素放到空出来的堆顶处，然后从上往下依次找左右子节点中更小的节点进行替换，直到x小于或等于左右孩子中的任何一个为止。

（4）remove（Object o），用于删除队列中跟o相等的某一个元素（若有多个相等，则删除第一个）。会改变队列结构，要维护小顶堆。remove(Object o)可以分为2种情况: 1. 删除的是最后一个元素。直接删除即可，不需要调整。2. 删除的不是最后一个元素，**从删除点开始（前面的删除是从堆顶开始）**以最后一个元素为参照调用一次siftDown()即可。

Part 2 Set接口体系---无序、不重复、无索引

因为Set体系底层实现与Map体系很相似，所以一起比较

Part 3 Map体系—无序、不重复、无索引

=====================HashSet与HashMap===================

HashSet和HashMap在Java里有着相同的实现，前者仅仅是对后者做了一层包装，也就是说HashSet里面有一个HashMap(适配器模式)。着重分析HashMap

HashMap---Map接口，即允许放入key为null的元素，也允许插入value为null的元素。

* 特点：无序、不重复、无索引
* 非线程安全---因为没有实现同步
* 底层实现： java7---数组+链表；

java8---数组+链表+红黑树（链表元素>=8时链表转为红黑树）

* 自动扩容： 当元素个数超过capacity\*load\_factor时，容器将自动扩容并重新哈希。

java7是先扩容后插入新值；java8是先插值再扩容

* 初始容量：16
* 负载系数：0.75
* 数组扩容：resize() 方法用于初始化数组或数组扩容，**每次扩容后**，**容量为原来的 2 倍**，并进行数据迁移。
* 时间复杂度：

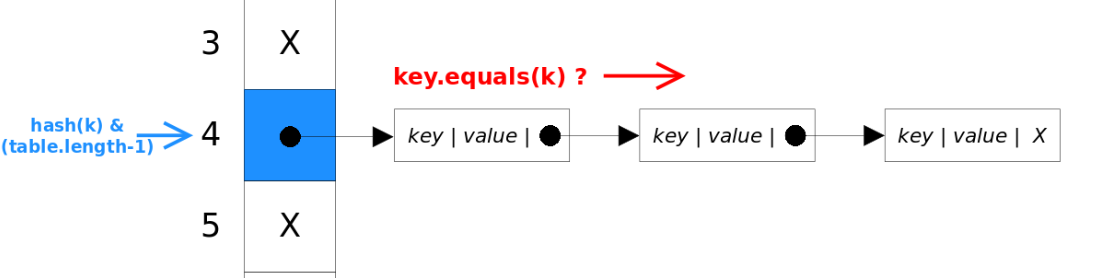
java7---根据hash值能快速定位到bucket，但要顺着链表一个个找，所以查询时间复杂度取决于链表长度，为O(n)

java8---在 Java8 中，当链表中的元素达到了 8 个时，会将链表转换为红黑树，在这些位置进行查找的时候可以降低时间复杂度为 O(logN)。

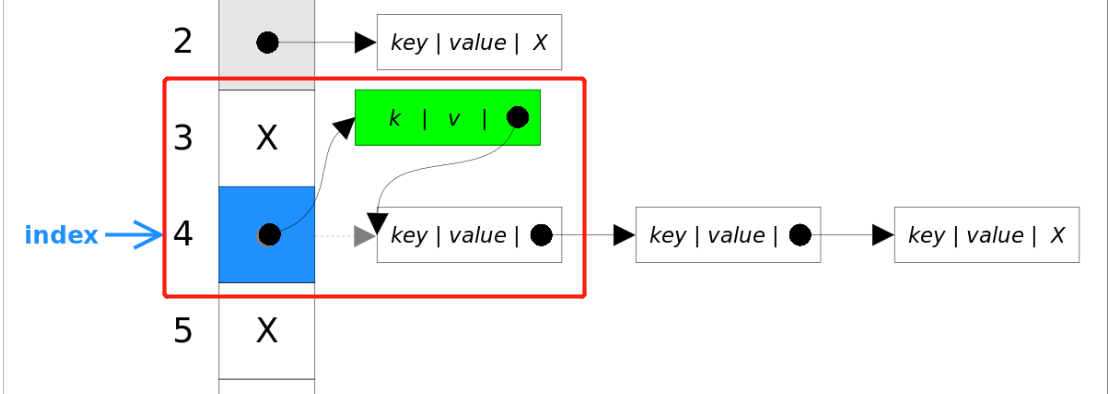
注意：

----------------------------------java7--------------------------

1. 根据对冲突的处理方式不同，哈希表有两种实现方式，一种**开放地址方式**(Open addressing)，另一种是**冲突链表方式**(Separate chaining with linked lists)。**Java7 HashMap采用的是冲突链表方式**。
2. 如果选择合适的哈希函数，put()和get()方法可以在常数时间内完成。但在对HashMap进行迭代时，需要遍历整个table以及后面跟的冲突链表。因此**对于迭代比较频繁的场景，不宜将HashMap的初始大小设的过大。**
3. 有两个参数可以影响HashMap的性能: 初始容量(inital capacity)和负载系数(load factor)。初始容量指定了初始table的大小，负载系数用来指定自动扩容的临界值。当entry的数量超过capacity\*load\_factor时，容器将自动扩容并重新哈希。**对于插入元素较多的场景，将初始容量设大可以减少重新哈希的次数。每次扩容就要重新哈希一次。**
4. 将对象放入到HashMap或HashSet中时，有两个方法需要特别关心: hashCode()和equals()。**hashCode()方法决定了对象会被放到哪个bucket里，当多个对象的哈希值冲突时，equals()方法决定了这些对象是否是“同一个对象”。**所以，如果要将自定义的对象放入到HashMap或HashSet中，需要\*@Override\*hashCode()和equals()方法。
5. get(Object key)方法根据指定的key值返回对应的value，该方法调用了getEntry(Object key)得到相应的entry，然后返回entry.getValue()。因此getEntry()是算法的核心。算法思想是首先通过hash()函数得到对应bucket的下标，然后依次遍历冲突链表，通过key.equals(k)方法来判断是否是要找的那个entry。



1. java7的put(K key, V value)方法是将指定的key, value对添加到map里。该方法首先会对map做一次查找，看是否包含该元组，如果已经包含则直接返回，查找过程类似于getEntry()方法；如果没有找到，则会通过addEntry(int hash, K key, V value, int bucketIndex)方法插入新的entry，插入方式为**头插法**。



-----------------------------------java8-----------------------

1. 相比于之前的版本， JDK1.8 之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）（将链表转换成红黑树前会判断，如果当前数组的长度小于 64，那么会选择先进行数组扩容，而不是转换为红黑树）时，将链表转化为红黑树，以减少搜索时间。
2. Java7 中使用 Entry 来代表每个 HashMap 中的数据节点，Java8 中使用 Node，基本没有区别，都是 key，value，hash 和 next 这四个属性，不过，Node 只能用于链表的情况，红黑树的情况需要使用 TreeNode。**根据Node和TreeNode类型的不同，当根据hash值找到一个bucket的时候，读取bucket第一个节点元素的类型，就知道这个bucket是链表结构还是红黑树结构了。**
3. Java8的get方法： 相对于 put 来说，get 真的太简单了。

* 计算 key 的 hash 值，根据 hash 值找到对应数组下标: hash & (length-1)
* 判断数组该位置处的元素是否刚好就是我们要找的，如果不是，走第三步
* **判断该元素类型**是否是 TreeNode，如果是，用红黑树的方法取数据，如果不是，走第四步
* 遍历链表，直到找到相等(==或equals)的 key

对比1---HashMap和HashSet的区别



对比2---HashMap和Hashtable的区别

|  |  |  |
| --- | --- | --- |
|  | HashMap | Hashtable（基本淘汰） |
| 继承的父类 | 继承AbstractMap | 继承Dictionary类 |
| 实现接口 | 实现Map接口 | 实现Map接口 |
| 是否线程安全？ | 非线程安全 | 线程安全（因为底层每个方法都使用synchronized） |
| 效率 | 更高 | 相比较低 |
| 支持null key，null value？ | HashMap 可以存储 null 的 key 和 value，但 null 作为键只能有一个，null 作为值可以有多个 | Hashtable 不允许有 null 键和 null 值，否则会抛出 NullPointerException |
| 初始容量 | 16 | 11 |
| 扩容后 | 容量变为原来2倍 | 容量变为原来2倍+1 |
| 底层数据结构 | Jdk8，数组+列表+红黑树 | 没有链表转红黑树机制 |
| contains方法 | 改成containsKey方法和containsValue方法 | 保留contains方法，有containsKey方法和containsValue方法。  其中，contains方法和containsValue功能相同 |

========================================================

=========================================================

=============LinkedHashSet和LinkedHashMap=============

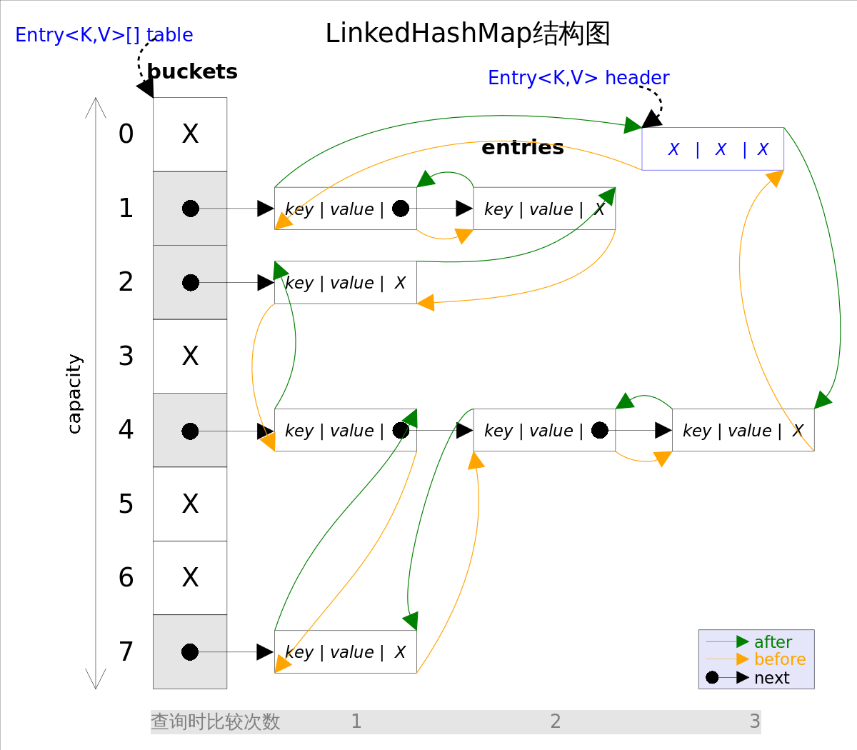
同理，LinkedHashSet和LinkedHashMap在Java里也有着相同的实现，前者仅仅是对后者做了一层包装，也就是说**LinkedHashSet**里面有一个**LinkedHashMap**(**适配器模式**)。着重介绍LinkedHashMap。

LinkedHashMap实现了Map接口，即允许放入key为null的元素，也允许插入value为null的元素。它同时满足HashMap和linked list的某些特性。**可将LinkedHashMap看作采用linked list增强的HashMap。**

LinkedHashMap---继承HashMap，实现map接口

* 特点：有序，无重复，无索引
* 非线程安全---因为非同步，若要在多线程情况下使用，要程序员手动同步

结构图：原来hashmap的链表改为双链表，且多了个header



注意：

1. 事实上LinkedHashMap是HashMap的直接子类，**二者唯一的区别是LinkedHashMap在HashMap的基础上，采用双向链表(doubly-linked list)的形式将所有entry连接起来，这样是为保证元素的迭代顺序跟插入顺序相同。**LinkedHashMap的结构图，主体部分跟HashMap完全一样，**多了header指向双向链表的头部(是一个哑元)**，该**双向链表的迭代顺序就是entry的插入顺序**。
2. 除了可以保迭代历顺序，这种结构还有一个好处 : **迭代LinkedHashMap时不需要像HashMap那样遍历整个table，而只需要直接遍历header指向的双向链表即可**，也就是说LinkedHashMap的迭代时间就只跟entry的个数相关，而跟table的大小无关。
3. 插入-put方法

插入有两重含义:

* 从**table的角度**看，新的entry需要插入到对应的bucket里，当有哈希冲突时，采用头插法将新的entry插入到**冲突链表的头部**。
* 从h**eader的角度**看，新的entry需要插入到**双向链表的尾部**。

1. 删除-remove方法

* 从table的角度看，需要将该entry从对应的bucket里删除，如果对应的冲突链表不空，需要修改冲突链表的相应引用。
* 从header的角度来看，需要将该entry从双向链表中删除，同时修改链表中前面以及后面元素的相应引用。

=======================================================

====================================================

===================TreeSet和TreeMap==================

同理，TreeSet只是对TreeMap做了一层包装，TreeSet里面有一个TreeMap(适配器模式)

TreeMap---实现了SortedMap接口，所以会按key的大小对Map中元素排序

* 特点：可排序，无重复，无索引
* 非线程安全
* 底层实现：红黑树
* 时间复杂度：containsKey(), get(), put(), remove()都有着log(n)的时间复杂度
* Key的大小评判：可以通过其本身的自然顺序(natural ordering)，也可以通过构造时传入的比较器(Comparator)。

红黑树是一种近似平衡的二叉查找树，它能够确保任何一个节点的左右子树的高度差不会超过二者中较低那个的一倍。