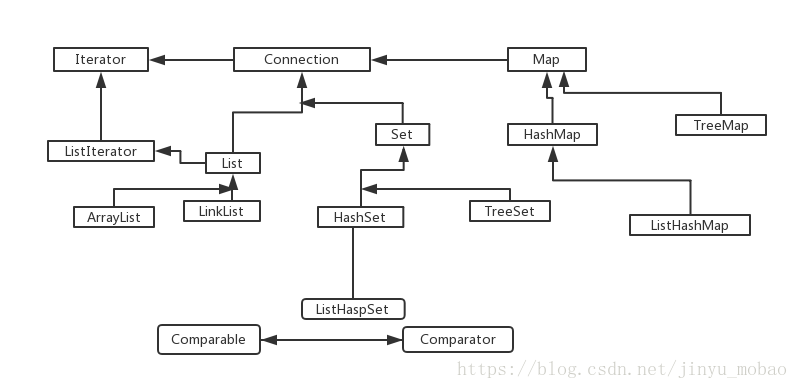
**集合有哪些？**

**常用：ArrayList、LinkedList、Vector、CopyOnWriteArrayList、HashSet、TreeSet、HashMap、TreeMap、HashTable、WeakHashMap、LinkedHashMap、ConCurrentHashMap**



**ArrayList:**

**动态数组**，可扩容，**底层为数组**，所以通过**下标访问比较快**，如果**新增或者删除需要重新排列数组，所以效率慢**

**初始容量为10**，当超出后**自动扩容**到1.5倍

//默认容量为10

private static final int **DEFAULT\_CAPACITY** = 10;

//当传入数量为0时使用该数组

private static final Object[] **EMPTY\_ELEMENTDATA** = {};

//默认的使用该数组

private static final Object[] **DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA** = {};

//实际存储数据的数组

transient Object[] **elementData**;

//数组大小

private int **size**;

知识点：

1. **transient 关键字：**忽略序列化，使用自定义序列化方法。原因：**elementData总是有空闲空间预留，序列化这些空闲空间浪费资源，所以自定义序列化数组。**
2. **数组扩容：每次扩容1.5倍，旧的容量向右位移1位**

**（**int newCapacity = oldCapacity + (oldCapacity >> 1);**）得知扩容后的容量后，进行数组拷贝，**Arrays.copyOf(elementData, newCapacity)，生成新的容量数组并且将就数组的元素拷贝过去

1. **fail-fast机制：当一个线程在修改集合，另一个线程读取会触发机制，不是线程安全的**
2. **基本方法：**
3. **int size(); //返回集合的长度**
4. **Boolean contains(Object o);//返回是否有元素**
5. **Int indexOf(Object o); //返回元素最低索引位置**
6. **Int LastIndexOf(Object o); //返回此元素的最高索引位置**
7. **Object clone(); //继承了Cloneable接口，是深拷贝，拷贝出新集合，不影响旧集合**
8. **Object[] toArray(); //返回了包含了集合所有元素的数组**
9. **E get(int index); //常用的获取某个下标的元素**
10. **E set(int index,E element); //设置index下标位置的值，有值则替换**
11. **Boolean add(E e); //添加元素到最后**
12. **Void add(int index,E element); //添加元素到index下标后**
13. **E remove(int index); //移除index下标位置的元素**
14. **Boolean remove(Object o); //移除此元素的第一个元素（符合条件的索引最低的）**
15. **Void clear(); //移除所有元素，底层是将所有元素置为null，方便CG**
16. **Boolean addAll(Collection<? extents E> c); //将此集合全部添加到集合尾部中**
17. **Boolean addAll(int index,Collection<? extents E> c);//将集合添加到集合index下标后**
18. **Boolean removeAll(Collection<? extents E> c); //移除list中指定集合包含的元素**
19. **ListIterator<E> listIterator(int index); //返回指定索引开始带有元素的迭代器**
20. **ListIterator<E> listIterator(); //返回所有带有元素的迭代器**

<https://www.cnblogs.com/gxl1995/p/7534171344218b3784f1beb90d621337.html>

**HashMap:**

HashMap是**散列表**，它存储的内容是key-value。底层是用**数组+链表**。当链表长度大于8时转换为**红黑树提高查询效率**。**允许key和value为null**。HashMap**不是线程安全**的，如果需要线程安全，可以使用**Collections.synchronizedMap返回一个线程安全的map**，或者使用**ConcurrentHashMap**。

内部使用一个**Entry表示键值对**，数组就是Entry数组（**Bucket**），Entry通过链表的方式连接后续节点（1.8之后根据链表长度决定是否转换为TreeMap）

***HashMap常量定义：***

**//默认初始容量为 16，必须为 2 的 n 次方 (一定是合数)**

**static final int DEFAULT\_INITIAL\_CAPACITY = 1 << 4;**

**//最大容量为 2 的 30 次幂**

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

**//HashMap 的默认负载因子**

**static final float DEFAULT\_LOAD\_FACTOR = 0.75f;**

**//链表转成红黑树的阈值。即在哈希表扩容时，当链表的长度(桶中元素个数)超过这个值的时候，进行链表到红黑树的转变**

**static final int TREEIFY\_THRESHOLD = 8;**

**// 红黑树转为链表的阈值。即在哈希表扩容时，如果发现链表长度(桶中元素个数)小于 6，则会由红黑树重新退化为链表**

**static final int UNTREEIFY\_THRESHOLD = 6;**

**//HashMap 的最小树形化容量。这个值的意义是：位桶（bin）处的数据要采用红黑树结构进行存储时，整个Table的最小容量（存储方式由链表转成红黑树的容量的最小阈值）当哈希表中的容量大于这个值时，表中的桶才能进行树形化，否则桶内元素太多时会扩容，而不是树形化 为了避免进行扩容、树形化选择的冲突，这个值不能小于 4 \* TREEIFY\_THRESHOLD**

**static final int MIN\_TREEIFY\_CAPACITY = 64;**

**// Node 是 HashMap 的一个内部类，实现了 Map.Entry 接口，本质是就是一个映射 (键值对)**

static class Node<K,V> implements Map.Entry<K,V> {

final int hash; // 用来定位数组索引位置

final K key;

V value;

Node<K,V> next; // 链表的下一个node

Node(int hash, K key, V value, Node<K,V> next) { ... }

public final K getKey() { ... }

public final V getValue() { ... }

public final String toString() { ... }

public final int hashCode() { ... }

public final V setValue(V newValue) { ... }

public final boolean equals(Object o) { ... }

}

**//哈希桶数组，分配的时候，table的长度总是2的幂**

**transient Node<K,V>[] table;**

//

**transient Set<Map.Entry<K,V>> entrySet;**

**// HashMap 中实际存储的 key-value 键值对数量**

**transient int size;**

**//用来记录 HashMap 内部结构发生变化的次数，主要用于迭代的快速失败机制**

**transient int modCount;**

**// HashMap 的门限阀值/扩容阈值，所能容纳的 key-value 键值对极限，当 size>=threshold时，就会扩容。计算方法：容量capacity \* 负载因子load factor**

**int threshold;**

**// HashMap 的负载因子**

**final float loadFactor;**

}

**主要点：**

**解决Hash冲突：**

什么是Hash冲突，HashMap通过HashCode计算得出的哈希桶数组索引位置（也就是Entry数组），总会有冲突，这时候需要处理。

**处理方法：**

1. 开放地址法
2. 再哈希法
3. 链地址法
4. 建立一个公共溢出区

**计算Hash值：**

通过key的hashCode来计算。（ key.hashCode() ^ key.hashCode >>> 16）

将取得的哈希值无符号右移16位，高位补0。并与前面第一步获得的hash码进行按位异或^ 运算。这样做有什么用呢？这其实也是**扰动函数**，**为了降低哈希码的冲突**。右位移16位，正好是32bit的一半，高半区和低半区做异或，就是为了混合原始哈希码的高位和低位，以此来加大低位的随机性。而且混合后的低位掺杂了高位的部分特征，这样高位的信息也被变相保留下来。也就是保**证考虑到高低Bit位都参与到Hash的计算中**

**Hash找哈希桶索引位置：**

Int i = hash & (table.length-1)

也就是计算出的**hash值与(数组长度-1)进行&运算。数组长度必须为2的幂数，**这样设计非常巧妙

.

Int hash = 9; hash值

Int tab = 4; 数组长度

所以:9 & (4-1) = 9 & 3 = 1001 & 0011 = 0001 = 1合理

如果tab 不为2的幂数 ,tab = 5

所以:9 & (5-1) = 9 & 4 = 1001 & 0100 = 0000 = 0不合理

**Put()方法：**

1. 先对key的HashCode()做hash算法，再计算出index
2. 如果没冲突就直接放入Entry，冲突则放入到Entry后的链表
3. 如果链表长度大于8则把链表转换为红黑树
4. 如果节点存在就替换旧的值
5. 如果size超过了扩容阈值就进行扩容（resize）

**Get()方法：**

1. 通过key的hashcode()计算hash，计算出index
2. 先判断first的Entry的key是否与参数key相等，相等取value，不相等遍历链表进行取值

Resize()方法：

1. 首先计算resize()后的新的容量和扩容阈值
2. 创建新的数组Entry，容量为新的容量
3. 将旧数组的元素放入新数组中

注意：

Jdk1.7前扩容会重新计算旧元素的Hash和下标

1.8后进行优化：

在扩容中只用判断原来的 hash 值与左移动的一位（newtable 的值）按位与操作是 0 或 1 就行，0 的话索引就不变，1 的话索引变成原索引加上扩容前数组，

节省了重新计算Hash的时间，而且把之前冲突的节点均匀分散到其他桶

## **平时在使用 HashMap 时一般使用什么类型的元素作为 Key？**

使用**String或者Integer**类型的。

**因为这些类都是不可变的类**，这些类已经很规范的覆写了equal和hashCode()方法，如果**使用可变对象，hash值可能就发生变化**，就有可能再也查询不到这个key了

<https://www.jianshu.com/p/b40fd341711e?utm_campaign=maleskine&utm_content=note&utm_medium=seo_notes&utm_source=recommendation>

**LinkedHashMap：**

LinkedHashMap**是HashMap的子类**，但是内部还有一个**双向链表维护键值对的顺序**，每个键值对既位于哈希表中，也位于双向链表中。LinkedHashMap支持两种顺序**插入顺序** 、 **访问顺序**

插入顺序：按照插入元素的顺序排序

访问顺序：将最新访问过的元素扔到链表末尾

构造方法中：accessOrder参数为true是访问顺序，默认为false

可以用作LRU算法：最不经常被使用的被抛弃

需要重写removeEldestEntry()方法，目的是什么时候移除最少使用的

**ConCurrentHashMap：**

是线程安全的。Jdk1.7前用分段锁实现线程安全，继承ReentrantLock，是重入锁

1.8后用CAS和****synchronized**来实现。在**put时候如果头节点值为null，则通过CAS**来设置当前值，**如果不为null，则用synchronized来申请锁**进行对链表进行设置值的操作。**

**ArrayList和LinkedList的区别：**

1. 数据结构：ArrayList是**数组**，LinkedList是**双向链表**
2. 效率：
   * + 1. 数组有下标，所以查询某个位置的值比较快，链表需要遍历整个链表，效率慢
       2. 数组删除或者插入都需要重新排序，效率慢，链表只需要改变上一个节点和下一个节点，效率快

总结;ArrayList**查询修改快**，**增删慢**，LinkedList**查询修改慢，增删快**

**ArrayList和Vector的区别：**

ArrayList**线程不安全，性能较好**，Vector**线程安全，因为同步所以性能较差**

ArrayList**扩容是1.5倍**，**Vector扩容是2倍**

**CopyOrWriteArrayList：**

内部持有一个**ReentrantLock lock = new ReentrantLock();**

底层是用**volatile** transient声明的数组 array

**读写分离，写时复制出一个新的数组，完成插入、修改或者移除操作后将新数组 赋值给array，整个过程不会影响读的线程，当修改完成后读的线程会立马察觉到 这个修改，因为array被volatile修饰了**

Vector是增删改查方法都加了synchronized，保证同步，但是每个方法执行的时候都要去获得锁，性能就会大大下降，而CopyOnWriteArrayList 只是在增删改上加锁，但是**读不加锁**，在读方面的性能就好于Vector，CopyOnWriteArrayList支持读多写少的并发情况。

**volatile 关键字：被它修饰的变量具有可见性，也就是其他线程修改后，读线程立马可见，禁止重排序**

**HashSet：**

作为Set集合的一种。 首先是无序的，**不可重复的**；允许存放null值；底层封装HashMap；实现是不同步的，线程不安全

**TreeMap：**

基于红黑树实现，TreeMap没有调优选项，因为该树总处于平衡状态。适用于按自然顺序或自定义顺序遍历键(key)。在需要排序的Map时候才用TreeMap。

**强引用：只要某个对象有强引用与之关联，就是报内存溢出都不会回收**

**软引用：内存不足时候回收（**SoftReference**）**

**弱引用：每次GC时候回收（**WekReference**）**

**虚引用：任何时候都可能被垃圾回收器回收。（**PhantomReference**）**