@来源 集合容器面试题（2021优化版）(qq.com)

集合容器概述

什么是集合，集合和数组的区别

集合：用于存储数据的容器。

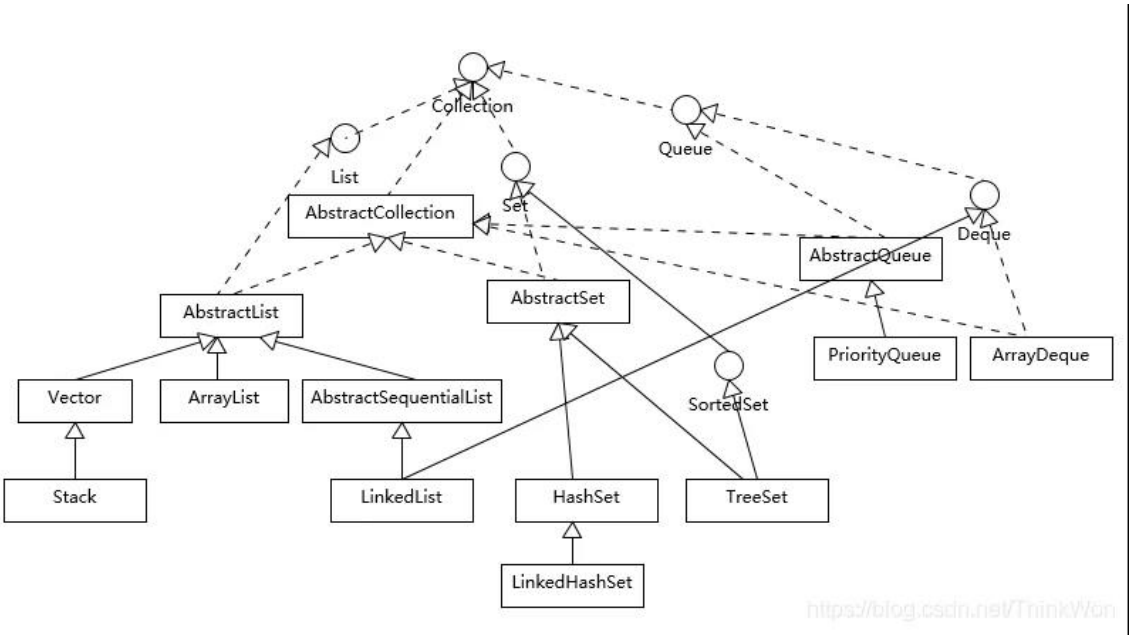
集合和数组的区别

- 数组是固定长度的；集合是可变长度的。
- 数组可以存储基本数据类型，也可以存储引用数据类型；集合只能存储引用数据类型。
- 数组是Java语言中内置的数据类型，是线性排列的，执行效率和类型检查都比集合快，集合提供了众多的属性和方法，方便操作。

联系：通过集合的toArray()方法可以将集合转换为数组，通过Arrays.asList()方法可以将数组转换为集合

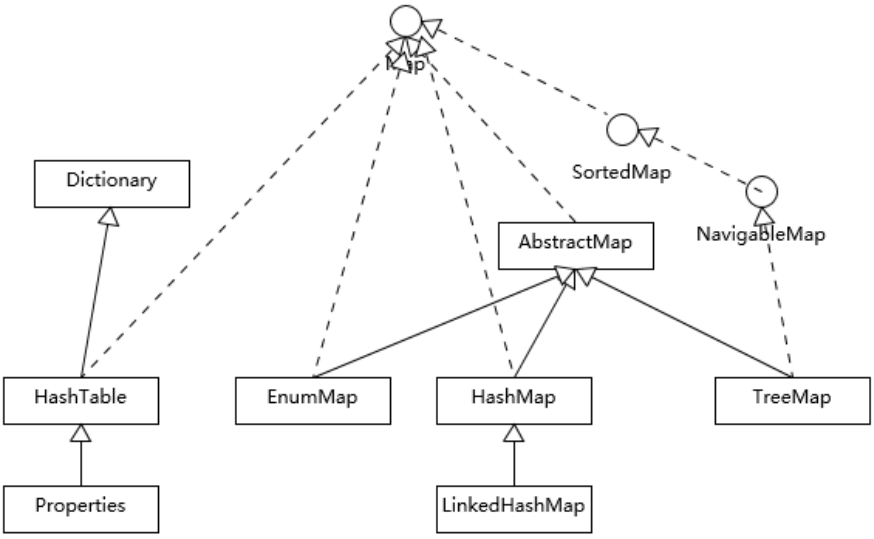
@$List，Set，Map三者的区别

Java 集合容器分为 Collection 和 Map 两大类，Collection集合的子接口有Set、List、Queue，Map接口不是collection的子接口。



Collection集合主要有List、Set和Queue接口

- List：一个有序（元素存入集合的顺序和取出的顺序一致）容器，元素可以重复，可以插入多个null元素，元素都有索引。常用的实现类有 ArrayList、LinkedList 和 Vector。
- Set：一个无序（存入和取出顺序有可能不一致）容器，不可以存储重复元素，只允许存入一个null元素，必须保证元素唯一性。常用实现类有 HashSet、LinkedHashSet 以及 TreeSet。
- Queue/Deque，则是 Java 提供的标准队列结构的实现，除了集合的基本功能，它还支持类似先入先出（FIFO， First-in-First-Out）或者后入先出（LIFO，Last-In-First-Out）等特定行为。常用实现类有ArrayDeque、ArrayBlockingQueue、LinkedBlockingDeque



Map是一个键值对集合，存储键和值之间的映射。Key无序，唯一；value 不要求有序，允许重复。Map没有继承Collection接口，从Map集合中检索元素时，只要给出键对象，就能返回对应的值对象。

常用实现类有HashMap、LinkedHashMap、ConcurrentHashMap、TreeMap、HashTable

集合框架底层数据结构

Collection

1. List
    - Arraylist：Object数组
    - LinkedList：双向循环链表
    - Vector：Object数组
2. Set
    - HashSet（无序，唯一）：基于 HashMap 实现，底层采用 HashMap 的key来保存元素
    - LinkedHashSet：LinkedHashSet 继承于 HashSet，并且其内部是通过LinkedHashMap 来实现的。
    - TreeSet（有序，唯一）：红黑树(自平衡的排序二叉树)

Map

- 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），但是数组长度小于64时会首先进行扩容，否则会将链表转化为红黑树，以减少搜索时间
- LinkedHashMap：LinkedHashMap 继承自 HashMap，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LinkedHashMap可以保持键值对的插入顺序。
- HashTable：数组+链表组成的，数组是 HashTable 的主体，链表则是主要为了解决哈希冲突而存在的
- TreeMap：红黑树（自平衡的排序二叉树）

Collection接口

List接口

@$ArrayList、LinkedList、Vector 有何区别?

这三者都是实现集合框架中的 List 接口，也就是所谓的有序集合，因此具体功能也比较近似，比如都提供搜索、添加或者删除的操作，都提供迭代器以遍历其内容等功能。

- 数据结构实现：ArrayList 和 Vector 是动态数组的数据结构实现，而 LinkedList 是双向循环链表的数据结构实现。
- 随机访问效率：ArrayList 和 Vector 比 LinkedList 在根据索引随机访问的时候效率要高，因为 LinkedList 是链表数据结构，需要移动指针从前往后依次查找。
- 增加和删除效率：在非尾部的增加和删除操作，LinkedList 要比 ArrayList 和 Vector 效率要高，因为 ArrayList 和 Vector 增删操作要影响数组内的其他数据的下标，需要进行数据搬移。因为 ArrayList 非线程安全，在增删元素时性能比 Vector 好。
- 内存空间占用：一般情况下LinkedList 比 ArrayList 和 Vector 更占内存，因为 LinkedList 的节点除了存储数据，还存储了两个引用，分别是前驱节点和后继节点
- 线程安全：ArrayList 和 LinkedList 都是不同步的，也就是不保证线程安全；Vector 使用了 synchronized 来实现线程同步，是线程安全的。
- 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 倍容量，而 ArrayList 只会增加 50%容量。
- 使用场景：在需要频繁地随机访问集合中的元素时，推荐使用 ArrayList，希望线程安全的对元素进行增删改操作时，推荐使用Vector，而需要频繁插入和删除操作时，推荐使用 LinkedList。

Java集合的快速失败机制 “fail-fast”?

快速失败机制是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时，有可能会产生 fail-fast 机制。

例如：假设存在两个线程（线程1、线程2），线程1通过Iterator在遍历集合A中的元素，在某个时候线程2修改了集合A的结构（是结构上面的修改，而不是简单的修改集合元素的内容），那么这个时候程序就会抛出 ConcurrentModificationException 异常，从而产生fail-fast机制。

原因分析：迭代器在遍历时，ArrayList的父类AbstarctList中有一个modCount变量，每次对集合进行修改时都会modCount++，而foreach的实现原理其实就是Iterator，ArrayList的Iterator中有一个expectedModCount变量，该变量会初始化和modCount相等，每当迭代器使用hashNext()/next()遍历下一个元素之前，都会检测modCount的值和expectedmodCount的值是否相等，如果集合进行增删操作，modCount变量就会改变，就会造成expectedModCount!=modCount，此时就会抛出ConcurrentModificationException异常

解决办法：

1. 在遍历过程中，所有涉及到改变modCount值的地方全部加上synchronized。
2. 使用CopyOnWriteArrayList来替换ArrayList

迭代器Iterator是什么？如何一边遍历一边删除Collection中的元素

Iterator 接口提供遍历任何 Collection 的接口。我们可以从一个 Collection 中使用迭代器方法来获取迭代器实例。迭代器取代了 Java 集合框架中的 Enumeration，同时迭代器允许调用者在迭代过程中增删元素。

边遍历边修改 Collection 的唯一正确方式是使用 Iterator.remove() 方法，如下：

```
Iterator<Integer> it = list.iterator();
while(it.hasNext()){
  // do something
  it.remove();
}
```

一种最常见的**错误**代码如下：

```
for(Integer i : list){
  list.remove(i)
}
```

运行以上错误代码会报 **ConcurrentModificationException 异常。**

遍历一个 List 有哪些不同的方式？每种方法的实现原理是什么？Java 中 List 遍历的最佳实践是什么？

遍历方式有以下几种：

1. for 循环遍历，基于计数器。在集合外部维护一个计数器，然后依次读取每一个位置的元素，当读取到最后一个元素后停止。
2. 迭代器遍历，Iterator。Iterator 是面向对象的一个设计模式，目的是屏蔽不同数据集合的差异，提供统一遍历集合的接口。Java 在 Collections 集合都支持了 Iterator 遍历。
3. foreach 循环遍历。foreach 内部也是采用了 Iterator 的方式实现，使用时不需要显式声明 Iterator 或计数器。优点是代码简洁，不易出错；缺点是只能做简单的遍历，不能在遍历过程中不能对集合进行增删操作。

最佳实践：Java Collections 框架中提供了一个 RandomAccess 接口，用来标记 List 实现是否支持 Random Access。

- 如果一个数据集合实现了该接口，就意味着它支持 Random Access，按索引读取元素的平均时间复杂度为 O(1)，如ArrayList。
- 如果没有实现该接口，表示不支持 Random Access，如LinkedList。

推荐的做法就是，支持 Random Access 的列表可用 for 循环遍历，否则建议用 Iterator 或 foreach 遍历。

说一下 ArrayList 的优缺点

ArrayList的优点如下：

- ArrayList 底层以数组实现，ArrayList 实现了 RandomAccess 接口，根据索引进行随机访问的时候速度非常快。
- ArrayList 在尾部添加一个元素的时候非常方便。

ArrayList 的缺点如下：

- 在非尾部的增加和删除操作，影响数组内的其他数据的下标，需要进行数据搬移，比较消耗性能

ArrayList 比较适合顺序添加、随机访问的场景。

为什么 ArrayList 的 elementData 加上 transient 修饰?

ArrayList 中的数组定义如下：

```
private transient Object[] elementData;
```

再看一下 ArrayList 的定义：

```
public class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```

可以看到 ArrayList 实现了 Serializable 接口，这意味着 ArrayList 支持序列化。transient 的作用是不希望 elementData 数组被序列化，重写了 writeObject 实现：

```
private void writeObject(java.io.ObjectOutputStream s)
    throws java.io.IOException{
    // Write out element count, and any hidden stuff
    int expectedModCount = modCount;
    s.defaultWriteObject();

    // Write out size as capacity for behavioural compatibility with clone()
    s.writeInt(size);

    // Write out all elements in the proper order.
    for (int i=0; i<size; i++) {
        s.writeObject(elementData[i]);
   }

    if (modCount != expectedModCount) {
        throw new ConcurrentModificationException();
   }
}
```

writeObject 的功能：每次序列化时，先调用 defaultWriteObject() 方法序列化 ArrayList 中的非 transient 元素，然后遍历 elementData，只序列化已存入的元素，**这样既加快了序列化的速度，又减小了序列化之后的文件大小。**

源码分析add()方法，remove()方法

add()方法（有四个）

增和删是ArrayList最重要的部分，这部分代码需要我们细细研究

```
//添加一个特定的元素到list的末尾
public boolean add(E e) {
    //先确保elementData数组的长度足够，size是数组中数据的个数，因为要添加一个元素，所以size+1，先判断size+1的这个个数数组能否放得下，在这个方法中去判断数组长度是否够用
    ensureCapacityInternal(size + 1);  // Increments modCount!!
    //在数据中正确的位置上放上元素e，并且size++
    elementData[size++] = e;
    return true;
}

//在指定位置添加一个元素
public void add(int index, E element) {
    rangeCheckForAdd(index);

    //先确保elementData数组的长度足够
    ensureCapacityInternal(size + 1);  // Increments modCount!!
    //将数据整体向后移动一位，空出位置之后再插入，效率不太好
    System.arraycopy(elementData, index, elementData, index + 1,
                     size - index);
```

```
    elementData[index] = element;
    size++;
}

// 校验插入位置是否合理
private void rangeCheckForAdd(int index) {
    //插入的位置肯定不能大于size 和小于0
    if (index > size || index < 0)
        //如果是，就报越界异常
        throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
}

//添加一个集合
public boolean addAll(Collection<? extends E> c) {
    //把该集合转为对象数组
    Object[] a = c.toArray();
    int numNew = a.length;
    //增加容量
    ensureCapacityInternal(size + numNew);  // Increments modCount
    //挨个向后迁移
    System.arraycopy(a, 0, elementData, size, numNew);
    size += numNew;
    //新数组有元素，就返回 true
    return numNew != 0;
}

//在指定位置，添加一个集合
public boolean addAll(int index, Collection<? extends E> c) {
    rangeCheckForAdd(index);

    Object[] a = c.toArray();
    int numNew = a.length;
    ensureCapacityInternal(size + numNew);  // Increments modCount

    int numMoved = size - index;
    //原来的数组挨个向后迁移
    if (numMoved > 0)
        System.arraycopy(elementData, index, elementData, index + numNew,
                         numMoved);
    //把新的集合数组 添加到指定位置
    System.arraycopy(a, 0, elementData, index, numNew);
    size += numNew;
    return numNew != 0;
}
```

对数组的容量进行调整

以上两种添加数据的方式都调用到了ensureCapacityInternal这个方法，我们看看它是如何完成工作的

```
//确保内部容量够用
private void ensureCapacityInternal(int minCapacity) {
    ensureExplicitCapacity(calculateCapacity(elementData, minCapacity));
}

//计算容量。判断初始化的elementData是不是空的数组，如果是空的话，返回默认容量10与
minCapacity=size+1的较大值者。
```

```
7  private static int calculateCapacity(Object[] elementData, int minCapacity)
   {
8     if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
9         return Math.max(DEFAULT_CAPACITY, minCapacity);
10    }
11    return minCapacity;
12 }
13
14 //确认实际的容量，这个方法就是真正的判断elementData是否够用
15 private void ensureExplicitCapacity(int minCapacity) {
16    modCount++;
17
18    //minCapacity如果大于了实际elementData的长度，那么就说明elementData数组的长度不够
   用，不够用那么就要增加elementData的length。这里有的小伙伴就会模糊minCapacity到底是什么
   呢，这里解释一下
19
20 /**
21     * 当我们要 add 进第1个元素到 ArrayList 时，elementData.length 为0 （因为还是一
   个空的 list），因为执行了 `ensureCapacityInternal()` 方法 ，所以 minCapacity 此时
   为10。此时，`minCapacity - elementData.length > 0 `成立，所以会进入
   `grow(minCapacity)` 方法。
22     * 当add第2个元素时，minCapacity 为2，此时elementData.length(容量)在添加第一个元
   素后扩容成 10 了。此时，`minCapacity - elementData.length > 0 ` 不成立，所以不会进
   入 （执行）`grow(minCapacity)` 方法。
23     * 添加第3、4···到第10个元素时，依然不会执行grow方法，数组容量都为10。
24     * 直到添加第11个元素，minCapacity(为11)比elementData.length（为10）要大。进入
   grow方法进行扩容。
25     */
26    // overflow-conscious code
27    if (minCapacity - elementData.length > 0)
28        //ArrayList能自动扩展大小的关键方法就在这里了
29        grow(minCapacity);
30 }
31
32 //扩容核心方法
33 private void grow(int minCapacity) {
34    //将扩充前的elementData大小给oldCapacity
35    // overflow-conscious code
36    int oldCapacity = elementData.length;
37    //新容量newCapacity是1.5倍的旧容量oldCapacity
38    int newCapacity = oldCapacity + (oldCapacity >> 1);
39    //这句话就是适应于elementData就空数组的时候，length=0，那么oldCapacity=0，
   newCapacity=0，所以这个判断成立，在这里就是真正的初始化elementData的大小了，就是为10。
40    if (newCapacity - minCapacity < 0)
41        newCapacity = minCapacity;
42    //如果newCapacity超过了最大的容量限制，就调用hugeCapacity，也就是将能给的最大值给
   newCapacity
43    if (newCapacity - MAX_ARRAY_SIZE > 0)
44        newCapacity = hugeCapacity(minCapacity);
45    //新的容量大小已经确定好了，就copy数组，改变容量大小。
46    // minCapacity is usually close to size, so this is a win:
47    elementData = Arrays.copyOf(elementData, newCapacity);
48 }
49
50 //这个就是上面用到的方法，很简单，就是用来赋最大值。
51 private static int hugeCapacity(int minCapacity) {
52    if (minCapacity < 0) // overflow
53        throw new OutOfMemoryError();
```

```
    //如果minCapacity都大于MAX_ARRAY_SIZE，那么就Integer.MAX_VALUE返回，反之将MAX_ARRAY_SIZE返回。因为maxCapacity是三倍的minCapacity，可能扩充的太大了，就用minCapacity来判断了。
//Integer.MAX_VALUE:2147483647   MAX_ARRAY_SIZE：2147483639 也就是说最大也就能给到第一个数值。还是超过了这个限制，就要溢出了。相当于arraylist给了两层防护。
    return (minCapacity > MAX_ARRAY_SIZE) ?
        Integer.MAX_VALUE :
    MAX_ARRAY_SIZE;
}
```

至此，我们彻底明白了`ArrayList`的扩容机制了。首先创建一个空数组**elementData**，第一次插入数据时直接扩充至10，然后如果**elementData**的长度不足，就扩充至1.5倍，如果扩充完还不够，就使用需要的长度作为**elementData**的长度。

remove()方法

其实这几个删除方法都是类似的。

```
//根据索引删除指定位置的元素
public E remove(int index) {
    //检查index的合理性
    rangeCheck(index);
//这个作用很多，比如用来检测快速失败的一种标志。
    modCount++;
    //通过索引直接找到该元素
    E oldValue = elementData(index);

    //计算要移动的位数。
    int numMoved = size - index - 1;
    if (numMoved > 0)
        //移动元素，挨个往前移一位。
        System.arraycopy(elementData, index+1, elementData, index,
                         numMoved);
    //将--size上的位置赋值为null，让gc(垃圾回收机制)更快的回收它。
    elementData[--size] = null; // clear to let GC do its work
//返回删除的元素。
    return oldValue;
}

//从此列表中删除指定元素的第一个匹配项，如果存在，则删除。通过元素来删除该元素，就依次遍历，如果有这个元素，就将该元素的索引传给fastRemove(index)，使用这个方法来删除该元素，fastRemove(index)方法的内部跟remove(index)的实现几乎一样，这里最主要是知道arrayList可以存储null值
public boolean remove(Object o) {
    if (o == null) {
        //挨个遍历找到目标
        for (int index = 0; index < size; index++)
            if (elementData[index] == null) {
                //快速删除
                fastRemove(index);
                return true;
            }
  } else {
        for (int index = 0; index < size; index++)
            if (o.equals(elementData[index])) {
                fastRemove(index);
                return true;
```

```
            }
        }
        return false;
    }

    //内部方法，“快速删除”，就是把重复的代码移到一个方法里
    private void fastRemove(int index) {
        modCount++;
        int numMoved = size - index - 1;
        if (numMoved > 0)
            System.arraycopy(elementData, index+1, elementData, index,
                             numMoved);
        elementData[--size] = null; // clear to let GC do its work
    }

    //删除或者保留指定集合中的元素
    //用于两个方法，一个removeAll()：它只清除指定集合中的元素，retainAll()用来测试两个集合是否有交集。
    private boolean batchRemove(Collection<?> c, boolean complement) {
        //将原集合，记名为A
        final Object[] elementData = this.elementData;
        //r用来控制循环，w是记录有多少个交集
        int r = 0, w = 0;
        boolean modified = false;
        try {
            //遍历 ArrayList 集合
            for (; r < size; r++)
                //参数中的集合c一次检测集合A中的元素是否有
                if (c.contains(elementData[r]) == complement)
                    //有的话，就给集合A
                    elementData[w++] = elementData[r];
        } finally {
            //发生了异常，直接把 r 后面的复制到 w 后面
            if (r != size) {
                //将剩下的元素都赋值给集合A
                System.arraycopy(elementData, r,
                                 elementData, w,
                                 size - r);
                w += size - r;
            }
            if (w != size) {
                //这里有两个用途，在removeAll()时，w一直为0，就直接跟clear一样，全是为null。
                //retainAll()：没有一个交集返回true，有交集但不全交也返回true，而两个集合相等的时候，返回false，所以不能根据返回值来确认两个集合是否有交集，而是通过原集合的大小是否发生改变来判断，如果原集合中还有元素，则代表有交集，而元集合没有元素了，说明两个集合没有交集。
                // 清除多余的元素，clear to let GC do its work
                for (int i = w; i < size; i++)
                    elementData[i] = null;
                modCount += size - w;
                size = w;
                modified = true;
            }
        }
        return modified;
    }

```

```
90
91  //保留公共的
92  public boolean retainAll(Collection<?> c) {
93      Objects.requireNonNull(c);
94      return batchRemove(c, true);
95  }
96
97  //将elementData中每个元素都赋值为null，等待垃圾回收将这个给回收掉
98  public void clear() {
99      modCount++;
100     //并没有直接使数组指向 null,而是逐个把元素置为空，下次使用时就不用重新 new 了
101     for (int i = 0; i < size; i++)
102         elementData[i] = null;
103
104     size = 0;
105 }
```

总结：根据索引删除指定位置的元素，此时会把指定下标到数组末尾的元素挨个向前移动一个单位，并且会把数组最后一个元素设置为null，这样是为了方便之后将整个数组不被使用时，会被GC，可以作为小的技巧使用。

Set接口

@$HashSet如何检查重复？HashSet是如何保证数据不可重复的？

向HashSet 中add ()元素时，判断元素是否存在的依据，不仅要比较hash值，还要结合equles方法比较。HashSet 中的add()方法会使用HashMap 的put()方法。

HashMap 的 key 是唯一的，由源码可以看出 HashSet 添加进去的值就是作为HashMap 的key，并且在HashMap中如果K/V相同时，会用新的V覆盖掉旧的V，然后返回旧的V，所以不会重复（ HashMap 比较key是否相等是先比较hashcode 再比较equals ）。

以下是HashSet 部分源码：

```
1  private static final Object PRESENT = new Object();
2  private transient HashMap<E,Object> map;
3
4  public HashSet() {
5      map = new HashMap<>();
6  }
7
8  public boolean add(E e) {
9      // 调用HashMap的put方法,PRESENT是一个至始至终都相同的虚值
10     return map.put(e, PRESENT)==null;
11 }
```

HashSet与HashMap的区别

	HashMap	HashSet
父接口	实现了Map接口	实现Set接口
存储数据	存储键值对	仅存储对象
添加元素	调用put()向map中添加元素	调用add()方法向Set中添加元素
计算哈希值	HashMap使用键（Key）计算hashcode	HashSet使用对象来计算hashcode值，对于两个对象来说hashcode可能相同，需要用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
获取元素的速度	HashMap相对于HashSet较快，因为它是使用唯一的键获取对象	HashSet较HashMap来说比较慢

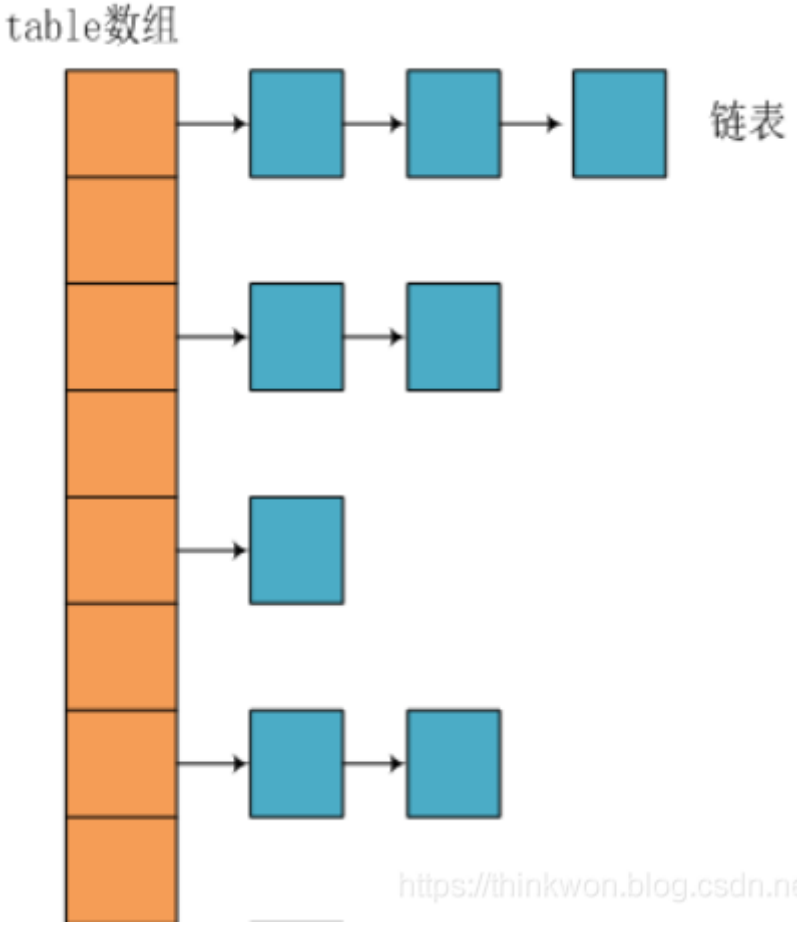
Map接口

@$说一下 HashMap 的实现原理？HashMap在JDK1.7和JDK1.8中有哪些不同？HashMap的底层实现

在Java中，保存数据有两种比较简单的数据结构：数组和链表。**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，使用一种叫做**拉链法**的方式可以解决哈希冲突。

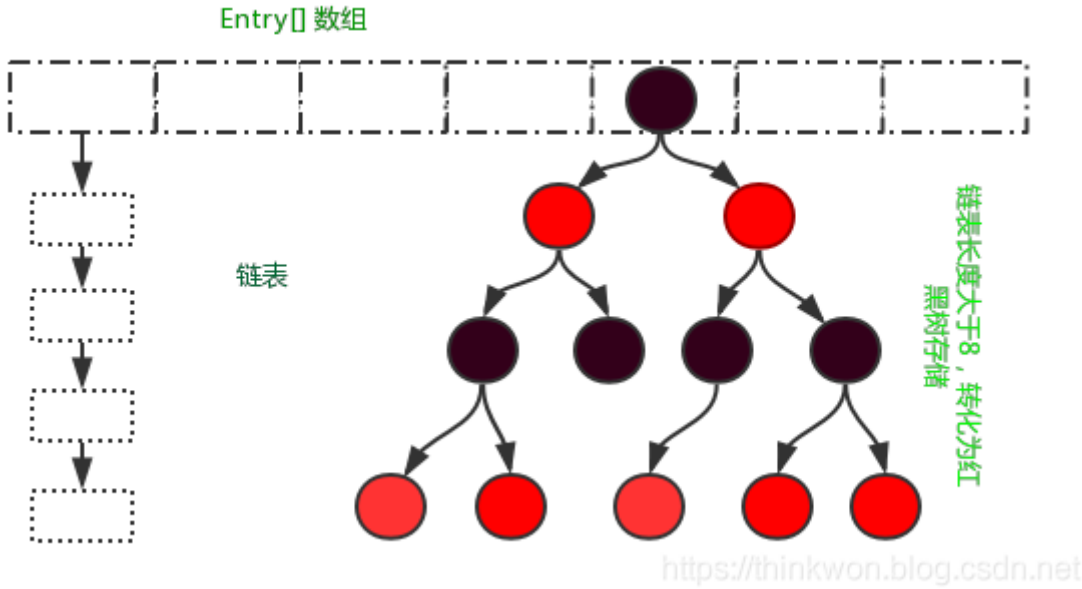
JDK1.8之前

JDK1.8之前采用的是拉链法。**拉链法**：将链表和数组相结合。也就是说创建一个链表数组，数组中每一格就是一个链表。若遇到哈希冲突，则将冲突的值加到链表中即可。



JDK1.8之后

相比于之前的版本，jdk1.8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），但是数组长度小于64时会首先进行扩容，否则会将链表转化为红黑树，以减少搜索时间。



JDK1.7 VS JDK1.8 比较

JDK1.8主要解决或优化了一下问题：

resize 扩容优化

引入了红黑树，目的是避免单条链表过长而影响查询效率

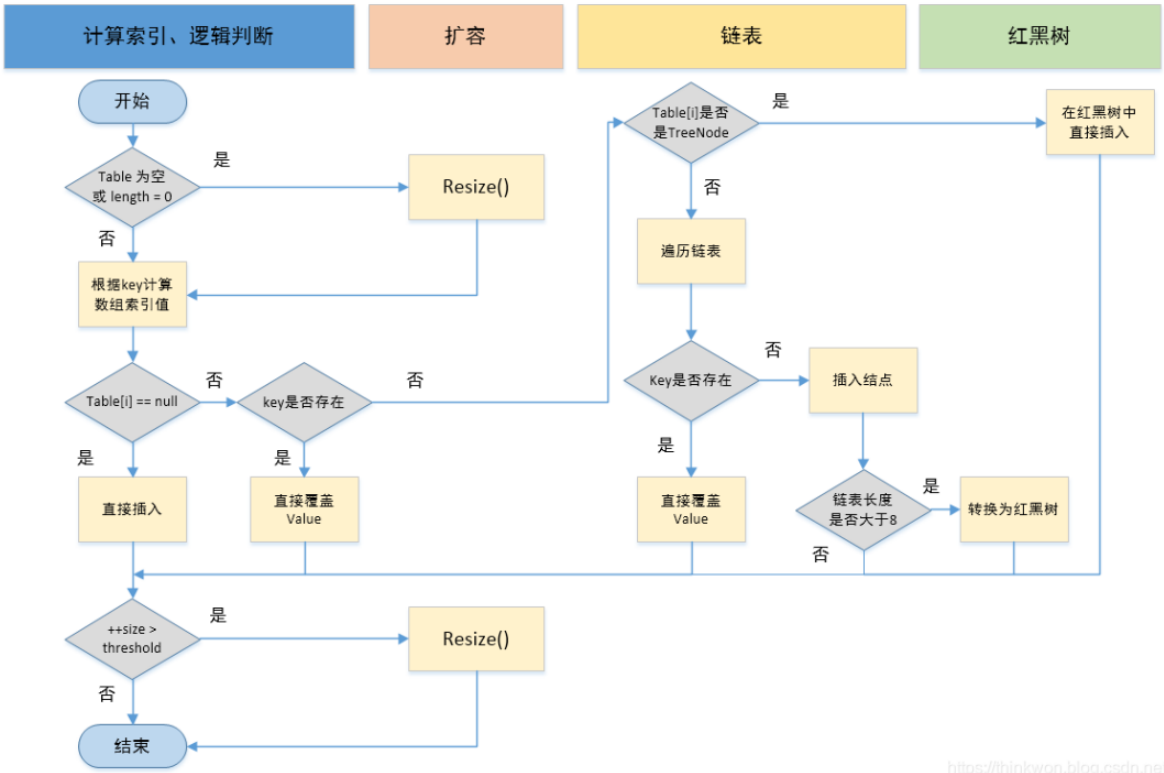
解决了多线程死循环问题，但仍是非线程安全的，多线程时可能会造成数据丢失问题。

不同	JDK 1.7	JDK 1.8
存储结构	数组 + 链表	数组 + 链表 + 红黑树
初始化方式	单独函数：`inflateTable()`	直接集成到了扩容函数`resize()`中
hash值计算方式	扰动处理 = 9次扰动 = 4次位运算 + 5次异或运算	扰动处理 = 2次扰动 = 1次位运算 + 1次异或运算
存放数据的规则	无冲突时，存放数组；冲突时，存放链表	无冲突时，存放数组；冲突 & 链表长度 < 8：存放单链表；冲突 & 链表长度 > 8 & 数组长度 < 64，扩容；冲突 & 数组长度 > 64：链表树化并存放红黑树
插入数据方式	头插法（先将原位置的数据都向后移动1位，再插入数据到头位置）	尾插法（直接插入到链表尾部/红黑树）
扩容后存储位置的计算方式	全部按照原来方法进行计算（即hashCode ->> 扰动函数 ->> (h&length-1)）	按照扩容后的规律计算（即扩容后的位置=原位置 or 原位置 + 旧容量）

@$HashMap的put方法的具体流程?

当我们put的时候，首先计算 key的hash值，这里调用了 hash方法，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，为了减少散列的碰撞，设计者综合考虑了速度、作用、质量之后，使用高16bit和低16bit异或来简单处理减少碰撞，而且JDK8中用了复杂度 O(logN) 的树结构来提升碰撞下的性能。

putVal方法执行流程图




```java
public V put(K key, V value) {
    return putVal(hash(key), key, value, false, true);
}

static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}

//实现Map.put和相关方法
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
                   boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    // 步骤①：tab为空则创建
    // table未初始化或者长度为0，进行扩容
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    // 步骤②：计算index，并对null做处理
    // (n - 1) & hash 确定元素存放在哪个桶中，桶为空，新生成结点放入桶中(此时，这个结点是放在数组中)
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    // 桶中已经存在元素
    else {
        Node<K,V> e; K k;
        // 步骤③：节点key存在，直接覆盖value
        // 比较桶中第一个元素(数组中的结点)的hash值相等，key相等
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
                // 将第一个元素赋值给e，用e来记录
                e = p;
        // 步骤④：判断该链为红黑树
        // hash值不相等，即key不相等；为红黑树结点
```

```
        // 如果当前元素类型为TreeNode，表示为红黑树，putTreeVal返回待存放的node，e可能为null
        else if (p instanceof TreeNode)
            // 放入树中
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        // 步骤⑤：该链为链表
        // 为链表结点
        else {
            // 在链表最末插入结点
            for (int binCount = 0; ; ++binCount) {
                // 到达链表的尾部

                //判断该链表尾部指针是不是空的
                if ((e = p.next) == null) {
                    // 在尾部插入新结点
                    p.next = newNode(hash, key, value, null);
                    //判断链表的长度是否达到转化红黑树的临界值，临界值为8
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                        //链表结构转树形结构
                        treeifyBin(tab, hash);
                    // 跳出循环
                    break;
                }
                // 判断链表中结点的key值与插入的元素的key值是否相等
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    // 相等，跳出循环
                    break;
                // 用于遍历桶中的链表，与前面的e = p.next组合，可以遍历链表
                p = e;
            }
        }
        //判断当前的key已经存在的情况下，再来一个相同的hash值、key值时，返回新来的value这个值
        if (e != null) {
            // 记录e的value
            V oldValue = e.value;
            // onlyIfAbsent为false或者旧值为null
            if (!onlyIfAbsent || oldValue == null)
                //用新值替换旧值
                e.value = value;
            // 访问后回调
            afterNodeAccess(e);
            // 返回旧值
            return oldValue;
        }
    }
    // 结构性修改
    ++modCount;
    // 步骤⑥：超过最大容量就扩容
    // 实际大小大于阈值则扩容
    if (++size > threshold)
        resize();
    // 插入后回调
    afterNodeInsertion(evict);
    return null;
}
```

①.判断键值对数组table[i]是否为空或为null，是的话执行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。

HashMap的扩容操作是怎么实现的?

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，新容量为旧容量的2倍；

③.扩展后元素的位置要么在原位置，要么移动到原位置 + 旧容量的位置。

在putVal()中使用到了2次resize()方法，resize()方法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(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+旧容量的位置

```
final Node<K,V>[] resize() {
    Node<K,V>[] oldTab = table;//oldTab指向hash桶数组
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    int oldThr = threshold;
    int newCap, newThr = 0;
    if (oldCap > 0) {//如果oldCap不为空的话，就是hash桶数组不为空
        if (oldCap >= MAXIMUM_CAPACITY) {//如果大于最大容量了，就赋值为整数最大的阀值
            threshold = Integer.MAX_VALUE;
            return oldTab;//返回
        }//如果当前hash桶数组的长度在扩容后仍然小于最大容量 并且oldCap大于默认值16
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                 oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; // double threshold 双倍扩容阀值threshold
    }
    // 旧的容量为0，但threshold大于零，代表有参构造有cap传入，threshold已经被初始化成最小2的n次幂
    // 直接将该值赋给新的容量
    else if (oldThr > 0) // initial capacity was placed in threshold
        newCap = oldThr;
    // 无参构造创建的map，给出默认容量和threshold 16, 16*0.75
    else {               // zero initial threshold signifies using defaults
        newCap = DEFAULT_INITIAL_CAPACITY;
        newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
    }
    // 新的threshold = 新的cap * 0.75
    if (newThr == 0) {
        float ft = (float)newCap * loadFactor;
```

```
27          newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY
?
28                    (int)ft : Integer.MAX_VALUE);
29      }
30      threshold = newThr;
31      // 计算出新的数组长度后赋给当前成员变量table
32      @SuppressWarnings({"rawtypes","unchecked"})
33          Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];//新建hash桶数组
34      table = newTab;//将新数组的值复制给旧的hash桶数组
35      // 如果原先的数组没有初始化，那么resize的初始化工作到此结束，否则进入扩容元素重排逻
辑，使其均匀的分散
36      if (oldTab != null) {
37          // 遍历新数组的所有桶下标
38          for (int j = 0; j < oldCap; ++j) {
39              Node<K,V> e;
40              if ((e = oldTab[j]) != null) {
41                  // 旧数组的桶下标赋给临时变量e，并且解除旧数组中的引用，否则就数组无法
被GC回收
42                  oldTab[j] = null;
43                  // 如果e.next==null，代表桶中就一个元素，不存在链表或者红黑树
44                  if (e.next == null)
45                      // 用同样的hash映射算法把该元素加入新的数组
46                      newTab[e.hash & (newCap - 1)] = e;
47                  // 如果e是TreeNode并且e.next!=null，那么处理树中元素的重排
48                  else if (e instanceof TreeNode)
49                      ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
50                  // e是链表的头并且e.next!=null，那么处理链表中元素重排
51                  else { // preserve order
52                      // loHead,loTail 代表扩容后不用变换下标，见注1
53                      Node<K,V> loHead = null, loTail = null;
54                      // hiHead,hiTail 代表扩容后变换下标，见注1
55                      Node<K,V> hiHead = null, hiTail = null;
56                      Node<K,V> next;
57                      // 遍历链表
58                      do {
59                          next = e.next;
60                          if ((e.hash & oldCap) == 0) {
61                              if (loTail == null)
62                                  // 初始化head指向链表当前元素e，e不一定是链表的第一
个元素，初始化后loHead
63                                  // 代表下标保持不变的链表的头元素
64                                  loHead = e;
65                              else
66                                  // loTail.next指向当前e
67                                  loTail.next = e;
68                              // loTail指向当前的元素e
69                              // 初始化后，loTail和loHead指向相同的内存，所以当
loTail.next指向下一个元素时，
70                              // 底层数组中的元素的next引用也相应发生变化，造成
lowHead.next.next.....
71                              // 跟随loTail同步，使得lowHead可以链接到所有属于该链表
的元素。
72                              loTail = e;
73                          }
74                          else {
75                              if (hiTail == null)
76                                  // 初始化head指向链表当前元素e，初始化后hiHead代表
下标更改的链表头元素
```

```
                                    hiHead = e;
                                else
                                    hiTail.next = e;
                                hiTail = e;
                            }
                        } while ((e = next) != null);
                        // 遍历结束，将tail指向null，并把链表头放入新数组的相应下标，形成新的映射。
                        if (loTail != null) {
                            loTail.next = null;
                            newTab[j] = loHead;
                        }
                        if (hiTail != null) {
                            hiTail.next = null;
                            newTab[j + oldCap] = hiHead;
                        }
                    }
                }
            }
        }
        return newTab;
    }
```

@$HashMap是怎么解决哈希冲突的?

在解决这个问题之前，我们首先需要知道**什么是哈希冲突**，而在了解哈希冲突之前我们还要知道**什么是哈希**才行；

什么是哈希?

Hash，一般翻译为“散列”，也有直接音译为“哈希”的，这就是把任意长度的输入通过散列算法，变换成固定长度的输出，该输出就是散列值（哈希值）；这种转换是一种压缩映射，也就是，散列值的空间通常远小于输入的空间，不同的输入可能会散列成相同的输出，所以不可能从散列值来唯一的确定输入值。**简单的说就是一种将任意长度的消息压缩到某一固定长度的消息摘要的函数。**

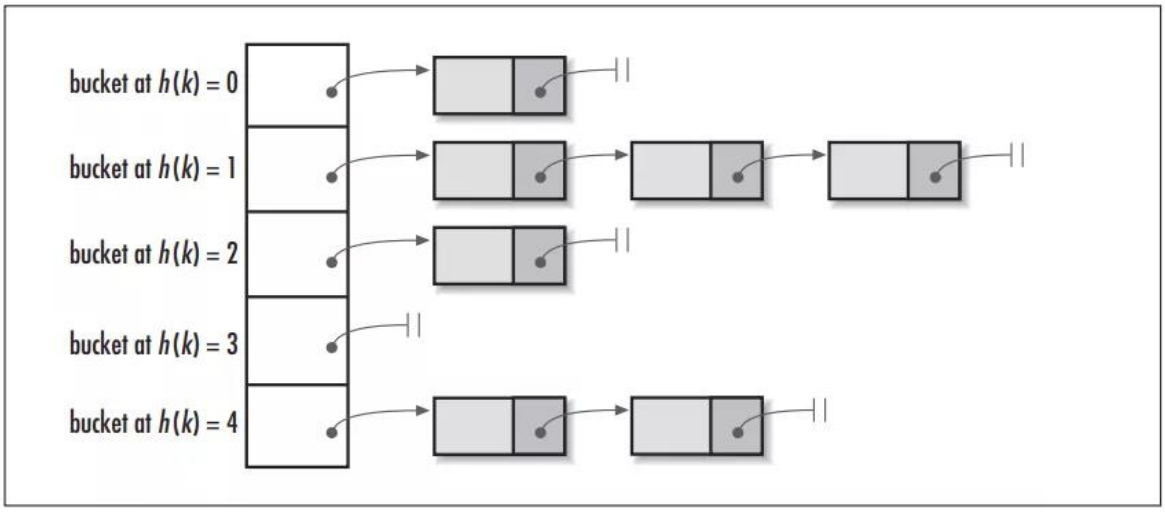
所有散列函数都有如下一个基本特性：**根据同一散列函数计算出的散列值如果不同，那么输入值肯定也不同。但是，根据同一散列函数计算出的散列值如果相同，输入值不一定相同。**

什么是哈希冲突?

当两个不同的输入值，根据同一散列函数计算出相同的散列值的现象，我们就把它叫做碰撞（哈希碰撞）。

HashMap的数据结构

在Java中，保存数据有两种比较简单的数据结构：**数组和链表。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，使用一种叫做**拉链法**的方式可以解决哈希冲突。



这样我们就可以将拥有相同哈希值的对象组织成一个链表放在hash值所对应的bucket下，**但相比于hashCode返回的int类型，我们HashMap初始的容量大小DEFAULT_INITIAL_CAPACITY = 1 << 4（即2的四次方16）要远小于int类型的范围，所以我们如果只是单纯的用hashCode取余来获取对应的bucket这将会大大增加哈希碰撞的概率，并且最坏情况下还会将HashMap变成一个单链表**，所以我们还需要对hashCode作一定的优化

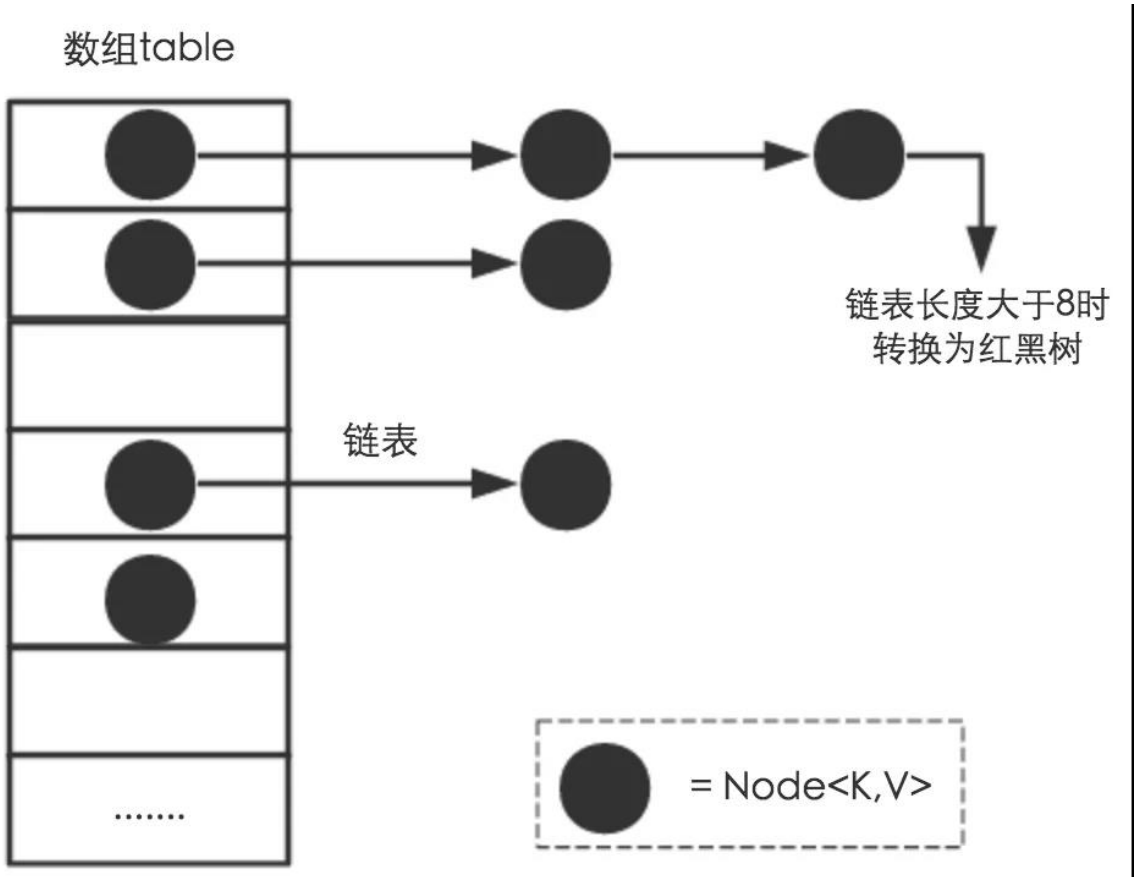
hash()函数

上面提到的问题，主要是因为如果使用hashCode取余，那么相当于**参与运算的只有hashCode的低位**，高位是没有起到任何作用的，所以我们的思路就是让hashCode取值出的高位也参与运算，进一步降低hash碰撞的概率，使得数据分布更平均，我们把这样的操作称为**扰动**，在**JDK 1.8**中的hash()函数如下：

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);// 与自己右移16位进行异或运算（高低位异或）
}
```

这比在**JDK 1.7**中，更为简洁，**相比在1.7中的4次位运算，5次异或运算（9次扰动），在1.8中，只进行了1次位运算和1次异或运算（2次扰动）**；

JDK1.8新增红黑树



通过上面的**链地址法（使用散列表）**和**扰动函数**我们成功让我们的数据分布更平均，哈希碰撞减少，但是当我们的HashMap中存在大量数据时，加入我们某个bucket下对应的链表有n个元素，那么遍历时间复杂度就为O(n)，为了针对这个问题，JDK1.8在HashMap中新增了红黑树的数据结构，进一步使得遍历复杂度降低至O(logn)；

总结

简单总结一下HashMap是使用了哪些方法来有效解决哈希冲突的：

1. **使用拉链法（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key?

String、Integer等包装类的特性能够保证Hash值的不可更改性和计算准确性，能够有效的减少Hash碰撞的几率

1. 都是final类型，即不可变性，保证key的不可更改性，不会存在同一对象获取hash值不同的情况
2. 内部已重写了equals()、hashCode()等方法，遵守了HashMap内部的规范，不容易出现Hash值计算错误的情况；

如果使用Object作为HashMap的Key，应该怎么办呢?

重写`hashCode()`和`equals()`方法

1. **重写hashCode()是因为需要计算数据的存储位置**，需要注意不要试图从散列码计算中排除掉一个对象的关键部分来提高性能，这样虽然能更快，但可能会导致更多的Hash碰撞；
2. **重写`equals()`方法**，需要遵守自反性、对称性、传递性、一致性以及对于任何非null的引用值x，x.equals(null)必须返回false的这几个特性，**目的是为了保证key在哈希表中的唯一性**；

HashMap 的长度为什么是2的幂次方

为了能让 HashMap 存取高效，尽量较少碰撞，也就是要尽量把数据分配均匀，每个链表/红黑树长度大致相同。这个实现就是把数据存到哪个链表/红黑树中的算法。

这个算法应该如何设计呢?

我们首先可能会想到采用%取余的操作来实现。但是，重点来了：“取余(%)操作中如果除数是2的幂次则等价于与其除数减一的与(&)操作（也就是说 **hash%length==hash&(length-1)的前提是 length 是2的n 次方**；）。” 并且 采用二进制与操作 &，相对于%能够提高运算效率，这就解释了 HashMap 的长度为什么是2的幂次方。

那为什么是两次扰动呢?

这样就是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；

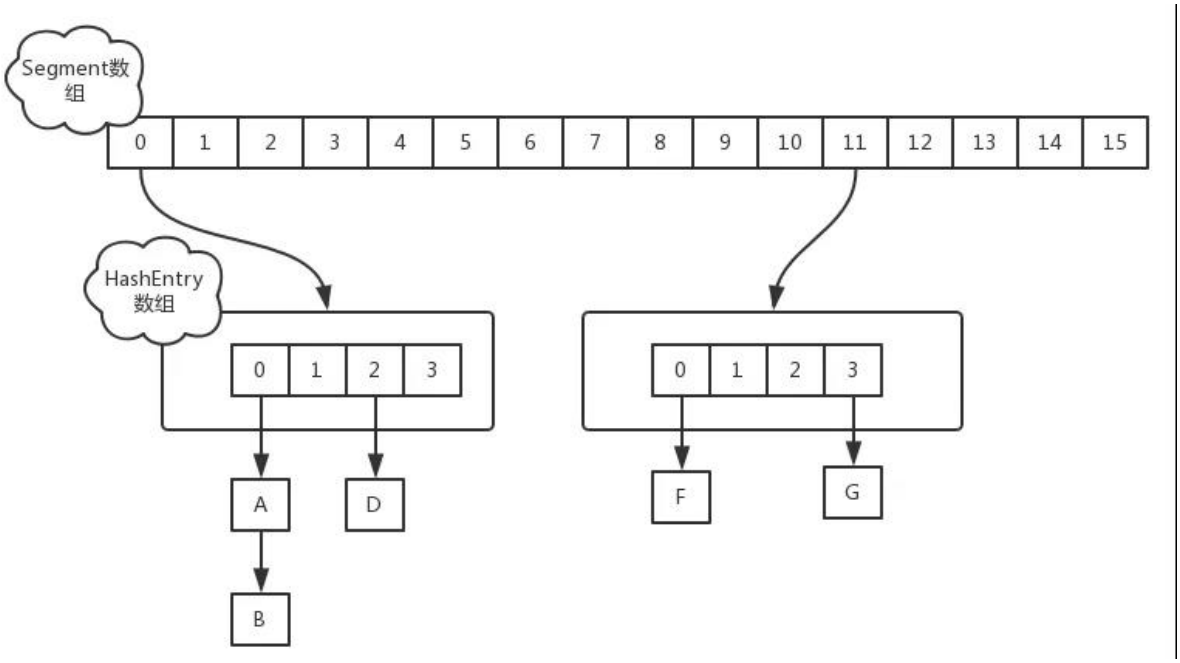
@$ConcurrentHashMap 底层具体实现知道吗？实现原理是什么?

JDK1.7

首先将数据分为一段一段的存储，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时，其他段的数据也能被其他线程访问。

在JDK1.7中，ConcurrentHashMap采用Segment + HashEntry的数据结构，结构如下：

一个 ConcurrentHashMap 里包含一个 Segment 数组。Segment 的结构和HashMap类似，是一种数组和链表结构，segment继承了ReentrantLock，一个 Segment 包含一个 HashEntry 数组，每个 HashEntry 是一个链表结构的元素。当对 HashEntry 数组的数据进行修改时，必须首先获得对应的 Segment 的锁。

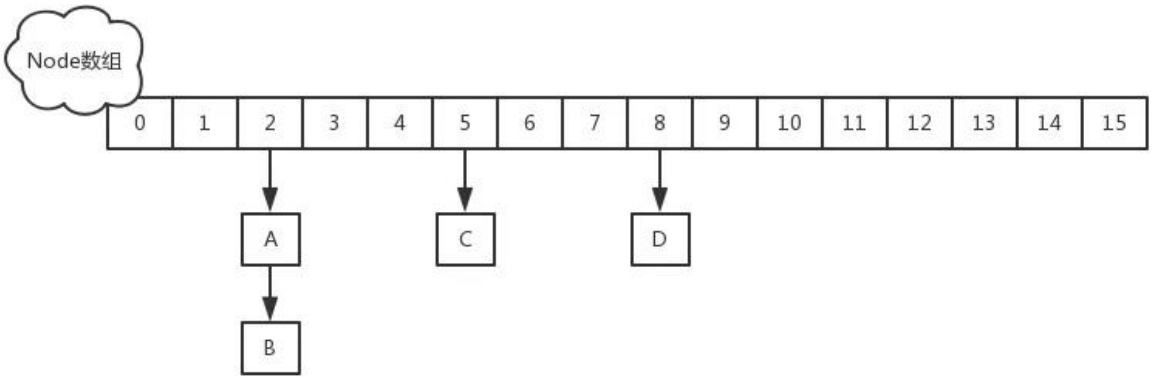


1. 该类包含两个静态内部类 HashEntry 和 Segment ； 前者用来封装映射表的键值对，后者用来充当锁的角色；
2. Segment 是一种可重入的锁 ReentrantLock，每个 Segment 守护一个HashEntry 数组里的元素，当对 HashEntry 数组的数据进行修改时，必须首先获得对应的 Segment 锁。

JDK1.8

在**JDK1.8中，放弃了Segment臃肿的设计，取而代之的是采用Node + CAS + Synchronized来保证并发安全**，synchronized只锁定当前链表或红黑二叉树的首节点，这样只要hash不冲突，就不会产生并发，效率又提升N倍。

结构如下：



对比 Hashtable、HashMap、TreeMap 有什么不同?

Hashtable、HashMap、TreeMap 都是最常见的 Map 实现，是以**键值对**的形式存储和操作数据的容器。

Hashtable 是早期 Java 类库提供的一个哈希表实现，本身是同步的即线程安全，不支持 null 键和值，由于同步导致的性能开销，所以已经很少被推荐使用。

HashMap 是应用更加广泛的哈希表实现，行为上大致上与 HashTable 一致，主要区别在于 HashMap 不是同步的，支持 null 键和值等。通常情况下，HashMap 进行 put 或者 get 操作，可以达到常数时间的性能，所以**它是绝大部分利用键值对存取场景的首选。**

TreeMap 则是基于红黑树的一种提供顺序访问的 Map，和 HashMap 不同，它的 get、put、remove 之类操作都是 O(logn) 的时间复杂度，具体顺序可以由指定的 Comparator 来决定，或者根据键的自然顺序来判断。

HashMap 和 ConcurrentHashMap 的区别

1. ConcurrentHashMap对整个桶数组进行了分割分段(Segment)，然后在每一个分段上都用lock锁进行保护，相对于HashTable的synchronized锁的粒度更精细了一些，并发性能更好，而HashMap没有锁机制，不是线程安全的。（JDK1.8之后ConcurrentHashMap启用了一种全新的方式实现，利用CAS算法。）
2. HashMap的键值对允许有null，但是ConCurrentHashMap都不允许。

ConcurrentHashMap 和 Hashtable 的区别?

jdk1.8采用的数据结构跟hashmap1.8的结构一样，数组+链表/红黑树，hashtable和jdk1.8之前的hashmap的底层数据机构类似都是采用数组+链表的形式，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

ConcurrentHashMap 和 Hashtable 的区别主要体现在底层数据结构和实现线程安全的方式上不同。

- **底层数据结构**：JDK1.7的 ConcurrentHashMap 底层采用 **分段的数组+链表** 实现，JDK1.8 采用的数据结构跟HashMap1.8的结构一样，数组+链表/红黑树。Hashtable 和 JDK1.8 之前的 HashMap 的底层数据结构类似都是采用 **数组+链表** 的形式，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；
- 实现线程安全的方式（重要）：

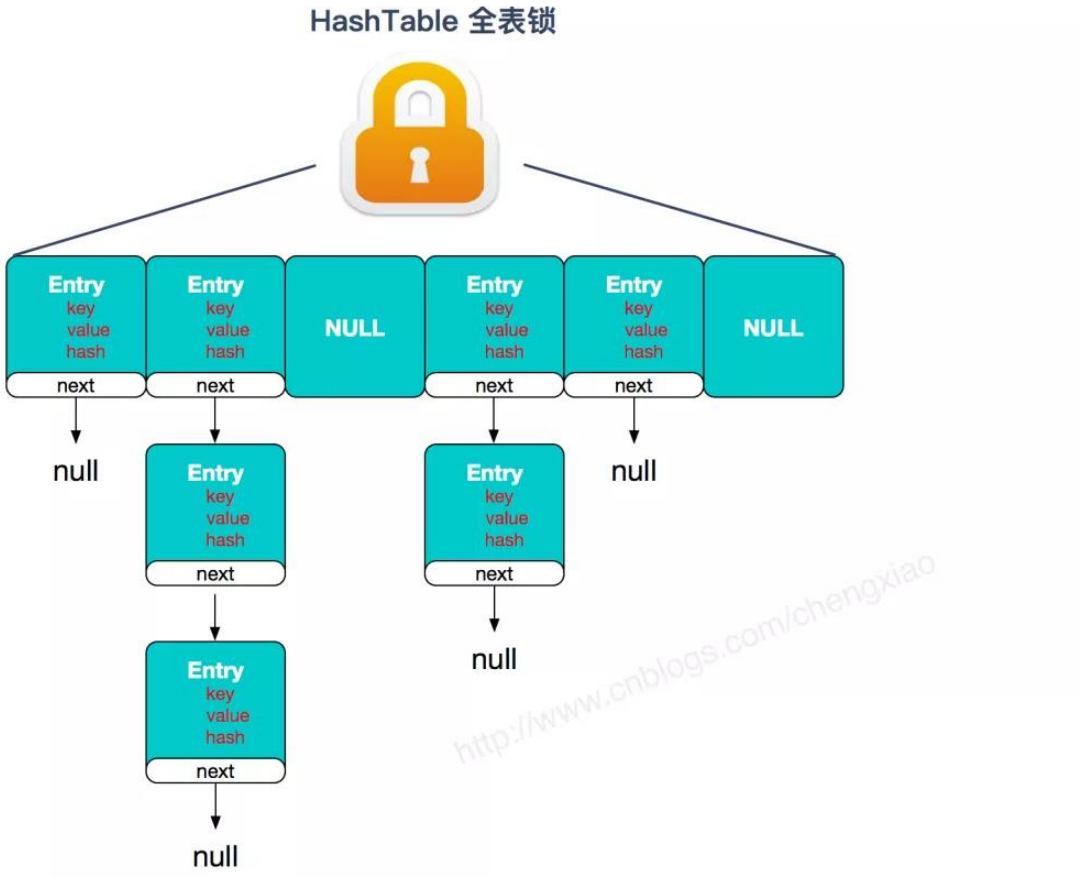
  ① **在JDK1.7的时候，ConcurrentHashMap（分段锁）** 对整个桶数组进行了分割分段(Segment)，每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。（默认分配16个Segment，比Hashtable效率提高16倍。） **到了JDK1.8 的时候已经摒弃了Segment的概念，而是直接用 Node 数组+链表+红黑树的数据结构来实现，并发控制使用 synchronized 和 CAS 来操作。（JDK1.6以后 对 synchronized锁做了很多优**

化） 整个看起来就像是优化过且线程安全的 HashMap，虽然在JDK1.8中还能看到 Segment 的数据结构，但是已经简化了属性，只是为了兼容旧版本；

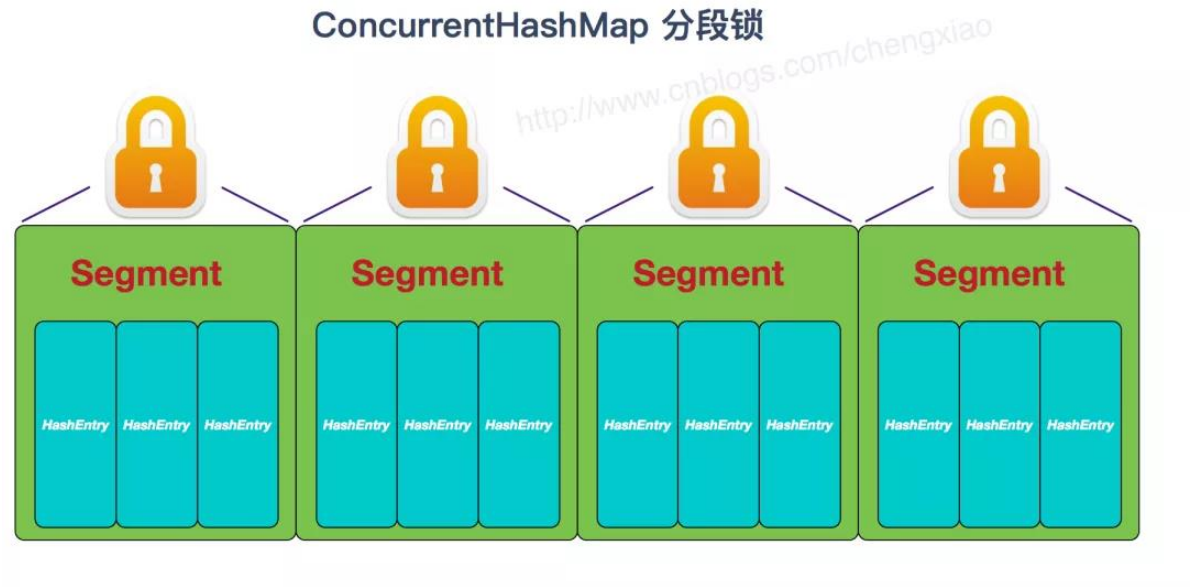
② **Hashtable(同一把锁)** :使用 synchronized 来保证线程安全，效率非常低下。当一个线程访问同步方法时，其他线程也访问同步方法，可能会进入阻塞或轮询状态，如使用 put 添加元素，另一个线程不能使用 put 添加元素，也不能使用 get，竞争会越来越激烈，效率越低。

两者的对比图:

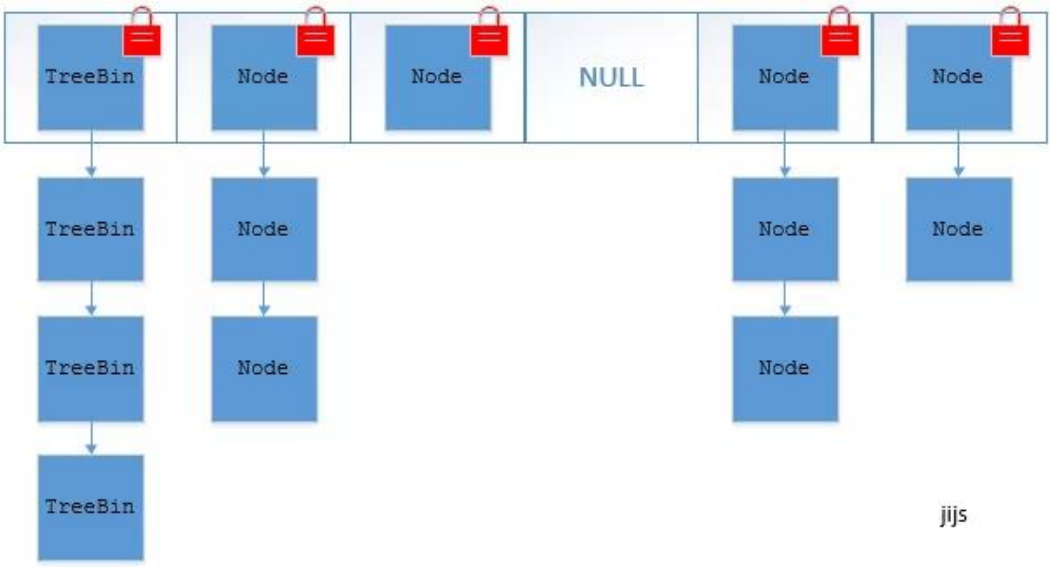
HashTable:



JDK1.7的ConcurrentHashMap：



JDK1.8的ConcurrentHashMap（TreeBin: 红黑树节点 Node: 链表节点）：



ConcurrentHashMap 结合了 HashMap 和 HashTable 二者的优势。HashMap 没有考虑同步，HashTable 考虑了同步的问题。但是 HashTable 在每次同步执行时都要锁住整个结构。ConcurrentHashMap 锁的方式是稍微细粒度的。

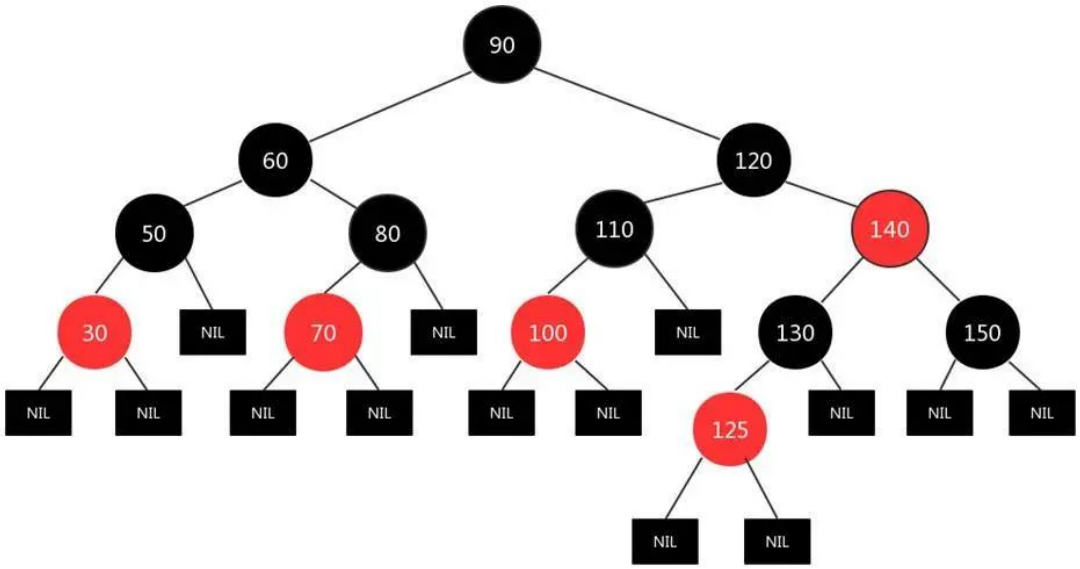
什么是红黑树

什么是二叉树

二叉树简单来说就是 每一个节上可以关联两个子节点

```
                    a
                  /   \
                b       c
              / \       / \
             d   e     f   g
            / \ / \   / \ / \
           h  i j k  l  m n o
```

红黑树定义和性质



红黑树是一种含有红黑结点并能自平衡的二叉查找树。它必须满足下面性质：

- 性质1：每个节点要么是黑色，要么是红色。
- 性质2：根节点是黑色。
- 性质3：每个叶子节点（NIL）是黑色。**[注意：这里叶子节点，是指为空(NIL或NULL)的叶子节点！]**
- 性质4：每个红结点的两个子结点一定都是黑色。
- 性质5：任意一结点到每个叶子结点的路径都包含相同数量的黑结点。

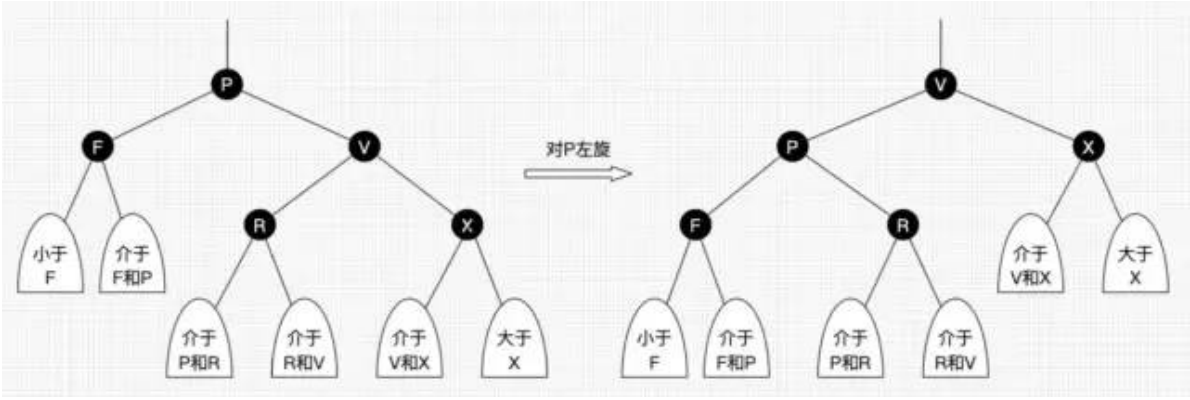
从性质5又可以推出：

- 性质5.1：如果一个结点存在黑子结点，那么该结点肯定有两个子结点

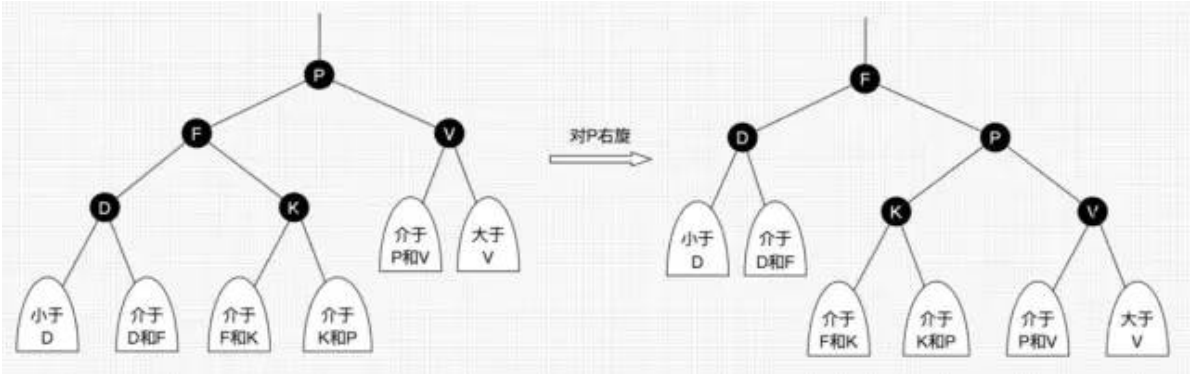
红黑树能自平衡，它靠的是什么？三种操作：左旋、右旋和变色。简单点说，旋转和变色的目的是让树保持红黑树的特性。

- **左旋**：以某个结点作为支点(旋转结点)，其右子结点变为旋转结点的父结点，右子结点的左子结点变为旋转结点的右子结点，左子结点保持不变。
- **右旋**：以某个结点作为支点(旋转结点)，其左子结点变为旋转结点的父结点，左子结点的右子结点变为旋转结点的左子结点，右子结点保持不变。
- **变色**：结点的颜色由红变黑或由黑变红。

左旋



右旋



辅助工具类

comparable 和 comparator的区别?

- Comparable 接口在 `java.lang` 包下，它有一个 compareTo(Object obj)方法用来排序；而 `Comparator` 接口在 `java.util` 包下，它有一个compare(Object obj1, Object obj2)方法用来排序
- 一个类实现了 Comparable 接口，意味着该类的对象可以直接进行比较（排序），但比较（排序）的方式只有一种，很单一；一个类如果想要保持原样，又需要进行不同方式的比较（排序），就可以定制比较器（实现 Comparator 接口）。
- Comparable 更多的像一个内部比较器，而 Comparator 更多的像一个外部比较器（体现了一种策略模式，耦合性较低），如果对象的排序需要基于自然顺序，请选择 `Comparable`，如果需要按照对象的不同属性进行排序，请选择 `Comparator`。

Collection 和 Collections 有什么区别

java.util.Collection 是一个集合接口（集合类的一个顶级接口）。它提供了对集合对象进行基本操作的通用接口方法。Collection接口在Java 类库中有很多具体的实现。Collection接口的意义是为各种具体的集合提供了最大化的统一操作方式，其直接继承接口有List、Set，Queue。

java.util.Collections 则是集合类的一个工具类/帮助类，其中提供了一系列静态方法，用于对集合中元素进行排序、搜索以及线程安全等各种操作。