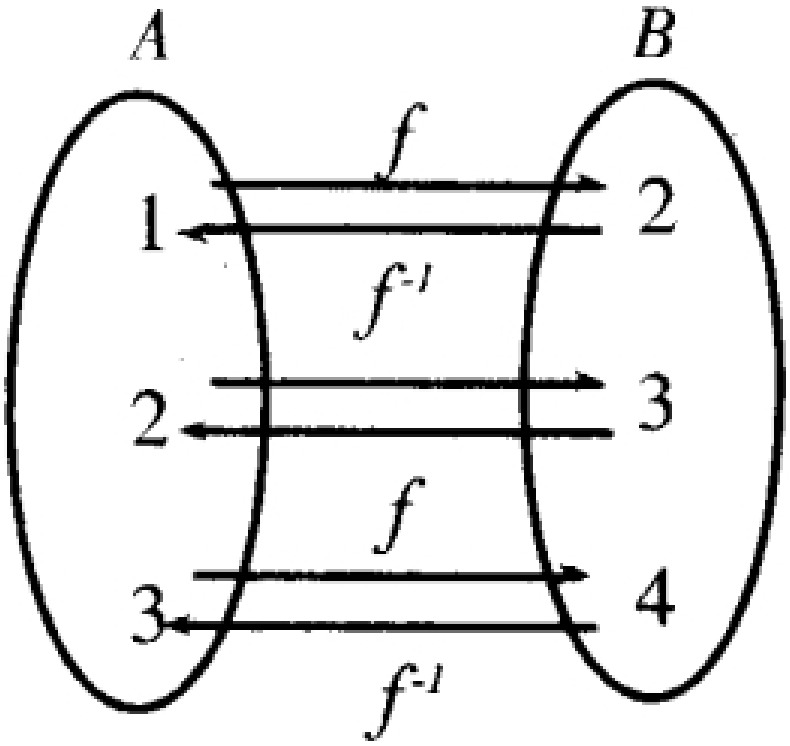
📖 青空的霞光B站课程

📅2024年4月18日

> ps: 要好好看代码内部的注释，很多接受都在里面

Map映射

⭐ 映射是指两个元素之间相互对应关系，也就是说元素之间它们是两两相对的，是以键值对的形式存在



⭐ 在`Java`中`Map`是为了实现这种数据结构而存在的，`Map`类似于`Python`的字典，是以键值对的形式来存储关系，通过健就可以找到它对应的值了，比如现在我们要保存很多学生的信息，而这些学生都有自己的ID，我们可以将其以映射的形式保存，将ID作为键，学生详细信息作为值，这样我们就可以通过学生的ID快速找到对应学生的信息了

⭐ 查看`Map`接口的源码，他需要接受两个参数一个作为`Key`一个作为`value`

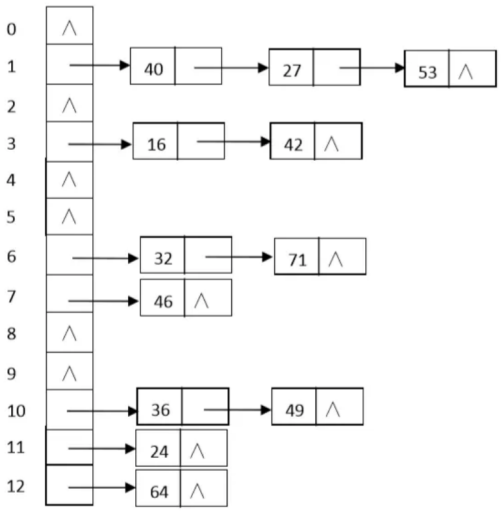
```java
//Map并不是Collection体系下的接口，而是单独的一个体系，因为操作特殊
//这里需要填写两个泛型参数，其中K就是键的类型，V就是值的类型，比如上面的学生信息，ID一般是int，那么键就是Integer类型的，而值就是学生信息，所以说值是学生对象类型的
public interface Map<K,V> {
    //-------- 查询相关操作 --------

    //获取当前存储的键值对数量
    int size();

    //是否为空
    boolean isEmpty();

    //查看Map中是否包含指定的键
    boolean containsKey(Object key);

```

```
    //查看Map中是否包含指定的值
    boolean containsValue(Object value);

    //通过给定的键，返回其映射的值,没有则返回null
    V get(Object key);

    //-------- 修改相关操作 --------

    //向Map中添加新的映射关系，也就是新的键值对
    V put(K key, V value);

    //根据给定的键，移除其映射关系，也就是移除对应的键值对
    V remove(Object key);


    //-------- 批量操作 --------

    //将另一个Map中的所有键值对添加到当前Map中
    void putAll(Map<? extends K, ? extends V> m);

    //清空整个Map
    void clear();


    //-------- 其他视图操作 --------

    //返回Map中存放的所有键，以Set形式返回，因为是唯一
    Set<K> keySet();

    //返回Map中存放的所有值，值可以不唯一可以使用集合类
    Collection<V> values();

    //返回所有的键值对，这里用的是内部类Entry在表示
    Set<Map.Entry<K, V>> entrySet();

    //这个是内部接口Entry，表示一个键值对
    interface Entry<K,V> {
        //获取键值对的键
        K getKey();

        //获取键值对的值
        V getValue();

        //修改键值对的值
        V setValue(V value);

        //判断两个键值对是否相等
        boolean equals(Object o);

        //返回当前键值对的哈希值
        int hashCode();

```

```
        ...
    }

    ...
}
```

⭐ `HashMap`是`Map`接口的一个实现类，它底层示也是采用哈希表的实现，它的底层是一个`Node`(`Node`是Map中Entry的实现类，不过他是`HashMap`的内部类)数组，但是它的哈希表是可以自动扩容的，而且它的每个哈希表内并不是单纯的采用链表方式，当达到一定限制的时候就会转换为红黑树结构，下面时源代码如何实现`Hash`表的

> ps：看下面代码的时候建议去编译器结合源代码一起看更好




```
public class HashMap<K,V> extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable {

    ...

    static class Node<K,V> implements Map.Entry<K,V> {    //内部使用结点，实际上就是存放的映射关系
        final int hash;
        final K key;    //跟我们之前不一样，我们之前一个结点只有键，而这里的结点既存放键也存放值，当然计算哈希还是使用键
        V value;
        Node<K,V> next;
                ...
    }

    ...

    transient Node<K,V>[] table;    //这个就是哈希表本体了，可以看到跟我们之前的写法是一样的，也是头结点数组，只不过HashMap中没有设计头结点（相当于没有头结点的链表）

    final float loadFactor;    //负载因子，这个东西决定了HashMap的扩容效果

    public HashMap() {
```

```
        this.loadFactor = DEFAULT_LOAD_FACTOR; //当我们创建对象时，会使用默认的负载因子，值为0.75
    }

    ...
}
```

1 先演示一下`Map`的使用，直接使用`put`方法传入一个键值对，但是在Map中不能存在同一个`Key`，如果后面闯入同一个`Key`，那么后面的就会将前面的覆盖，一下一个整体的功能使用

```
    public static void main(String[] args) {
        Map<Integer,String> map = new HashMap<>();
        map.put(1,"小明"); //put加入Map传入键值对
        map.put(2,"小明");
        map.put(1,"小红"); //相同的key
        map.putIfAbsent(2,"小强"); //这个是只有当前key不存在时才会插入
        System.out.println(map.get(2)); //获取key为2的值
        System.out.println(map.get(3));
        System.out.println(map); //重写了toString方法
        System.out.println(map.keySet()); //返回key的Set集合
        System.out.println(map.values()); //放回values的Collection集合
        map.entrySet().forEach(System.out::println); //entry会返回一个打包所有键值对的Set集合，可以直接使用forEach方法来遍历
    }

//输出:
小明
null
{1=小红, 2=小明}
[1, 2]
[小红, 小明]
1=小红
2=小明
```

2 先看看它的`put`方法

```
public V put(K key, V value) {
    //这里计算完键的哈希值之后，调用的另一个方法进行映射关系存放
    return putVal(hash(key), key, value, false, true);
}

final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    if ((tab = table) == null || (n = tab.length) == 0)  //如果底层哈希表没初始化，先初始化
        n = (tab = resize()).length;   //通过resize方法初始化底层哈希表，初始容量为16，后续会根据情况扩容，底层哈希表的长度永远是2的n次方
    //因为传入的哈希值可能会很大，这里同样是进行取余操作
    //(n - 1) & hash 等价于 hash % n 这里的i就是最终得到的下标位置了
    if ((p = tab[i = (n - 1) & hash]) == null)
```

```
        tab[i] = newNode(hash, key, value, null);   //如果这个位置上什么都没有，那就直接放一个新的结点
    else {   //这种情况就是哈希冲突了
        Node<K,V> e; K k;
        if (p.hash == hash &&   //如果上来第一个结点的键的哈希值跟当前插入的键的哈希值相同，键也相同，说明已经存放了相同键的键值对了，那就执行覆盖操作
            ((k = p.key) == key || (key != null && key.equals(k))))
            e = p;   //这里直接将待插入结点等于原本冲突的结点，一会直接覆盖
        else if (p instanceof TreeNode)   //如果第一个结点是TreeNode类型的，说明这个链表已经升级为红黑树了
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);  //在红黑树中插入新的结点
        else {
            for (int binCount = 0; ; ++binCount) {  //普通链表就直接在链表尾部插入
                if ((e = p.next) == null) {
                    p.next = newNode(hash, key, value, null);  //找到尾部，直接创建新的结点连在后面
                    if (binCount >= TREEIFY_THRESHOLD - 1) //如果当前链表的长度已经很长了，达到了阈值
                        treeifyBin(tab, hash);          //那么就转换为红黑树来存放
                    break;   //直接结束
                }
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))  //同样的，如果在向下找的过程中发现已经存在相同键的键值对了，直接结束，让p等于e一会覆盖就行了
                    break;
                p = e;
            }
        }
        if (e != null) { // 如果e不为空，只有可能是前面出现了相同键的情况，其他情况e都是null，所有直接覆盖就行
            V oldValue = e.value;
            if (!onlyIfAbsent || oldValue == null)
                e.value = value;
            afterNodeAccess(e);
            return oldValue;   //覆盖之后，会返回原本的被覆盖值
        }
    }
    ++modCount;
    if (++size > threshold)   //键值对size计数自增，如果超过阈值，会对底层哈希表数组进行扩容
        resize();   //调用resize进行扩容
    afterNodeInsertion(evict);
    return null;  //正常插入键值对返回值为null
}
```

3 上面`Put`代码中出现了`resize`方法，它是用来初始化和增加哈希表的长度的，这样是为了解决哈希冲突的，虽然当`HashMap`的一个链表长度过大时，会自动转换为红黑树但是，还是治标不治本，还不如直接增加哈希表的长度

```
final Node<K,V>[] resize() {
```

```
    Node<K,V>[] oldTab = table;   //先把下面这几个旧的东西保存一下
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    int oldThr = threshold; //threshlod它取决于何时来觉得扩容操作默认值为16
    int newCap, newThr = 0;  //这些是新的容量和扩容阈值
    if (oldCap > 0) {  //如果旧容量大于0，那么就开始扩容
        if (oldCap >= MAXIMUM_CAPACITY) {  //如果旧的容量已经大于最大限制了，那么直接给到 Integer.MAX_VALUE
            threshold = Integer.MAX_VALUE;
            return oldTab;  //这种情况不用扩了
        }
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                 oldCap >= DEFAULT_INITIAL_CAPACITY)   //新的容量等于旧容量的2倍，同样不能超过最大值
            newThr = oldThr << 1; //新的阈值也提升到原来的两倍
    }
    else if (oldThr > 0) // 旧容量不大于0只可能是还没初始化，这个时候如果阈值大于0，直接将新的容量变成旧的阈值
        newCap = oldThr;
    else {               // 默认情况下阈值也是0，也就是我们刚刚无参new出来的时候
        newCap = DEFAULT_INITIAL_CAPACITY;   //新的容量直接等于默认容量16
        newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY); //阈值为负载因子乘以默认容量，负载因子默认为0.75，也就是说只要整个哈希表用了75%的容量，那么就进行扩容，至于为什么默认是0.75，原因很多，这里就不解释了，反正作为新手，这些都是大佬写出来的，我们用就完事。
    }
    ...
    threshold = newThr;
    @SuppressWarnings({"rawtypes","unchecked"})
    Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
    table = newTab;   //将底层数组变成新的扩容之后的数组
    if (oldTab != null) {  //如果旧的数组不为空，那么还需要将旧的数组中所有元素全部搬到新的里面去
        ...   //详细过程就不介绍了
    }
}
```

4 可以在看一下`entrySet`的源码，我对这个比较感兴趣

> ps: 集合AI和源代码去看会比较好懂

```
public Set<Map.Entry<K,V>> entrySet() {
    Set<Map.Entry<K,V>> es;
    return (es = entrySet) == null ? (entrySet = new EntrySet()) : es; //这是一个三元运算符，entrySet如果为空就创建一个新的 EntrySet,
}
//返回的就是这个内部类，它继承自AbstractSet
 final class EntrySet extends AbstractSet<Map.Entry<K,V>> {
    public final int size()                 { return size; } //调用entrySet可以直接使用size方法
    public final void clear()               { HashMap.this.clear(); } //同上
    public final Iterator<Map.Entry<K,V>> iterator() { //这是一个迭代器，它会遍历Map的Entry中的每个元素并返回，这也就是entrySet的核心功能
```

```
        return new EntryIterator();
    }
     //==========================下面就是一下基本功能了
================================//
    public final boolean contains(Object o) { //判断是否包含
        if (!(o instanceof Map.Entry))
            return false;
        Map.Entry<?,?> e = (Map.Entry<?,?>) o;
        Object key = e.getKey();
        Node<K,V> candidate = getNode(hash(key), key);
        return candidate != null && candidate.equals(e);
    }
    public final boolean remove(Object o) { //删除
        if (o instanceof Map.Entry) {
            Map.Entry<?,?> e = (Map.Entry<?,?>) o;
            Object key = e.getKey();
            Object value = e.getValue();
            return removeNode(hash(key), key, value, true, true) != null;
        }
        return false;
    }
    public final Spliterator<Map.Entry<K,V>> spliterator() {
        return new EntrySpliterator<>(HashMap.this, 0, -1, 0, 0);
    }
    public final void forEach(Consumer<? super Map.Entry<K,V>> action) {
//forEach方法
        Node<K,V>[] tab;
        if (action == null)
            throw new NullPointerException();
        if (size > 0 && (tab = table) != null) {
            int mc = modCount;
            for (int i = 0; i < tab.length; ++i) {
                for (Node<K,V> e = tab[i]; e != null; e = e.next)
                    action.accept(e);
            }
            if (modCount != mc)
                throw new ConcurrentModificationException();
        }
    }
}
```

在看完源代码后，发现它里面也有很多方法可以试一下，可以使用`entrySet`调用`EntrySet`内部类的方法

```
    public static void main(String[] args) {
        Map<Integer,String> map = new HashMap<>();
        map.put(1,"小明");
        map.put(2,"小强");
        map.put(3,"小红");
        System.out.println(map.entrySet().contains(new AbstractMap.SimpleEntry<>(1,"小明"))); //可以直接调用EntrySet类中的方法,AbstractMap.SimpleEntry表示创建一个简单的map
    }

//输出:
true
```

使用`forEach`方法还可以调用到`Entry`中的方法，也就是`HashMap`中的`Node`内部类

```
    public static void main(String[] args) {
        Map<Integer,String> map = new HashMap<>();
        map.put(1,"小明");
        map.put(2,"小强");
        map.put(3,"小红");
        map.entrySet().forEach(entry -> {
            System.out.println(entry.getKey());
            System.out.println(entry.getValue());
        });
    }

//输出:
1
2
3
```

⭐ `LinkedHashMap`是直接继承至`HashMap`的，具有`HashMap`的全部性质，同时得益于每一个节点都是一个双向链表，在插入键值对时，同时保存了插入顺序

```
static class Entry<K,V> extends HashMap.Node<K,V> {   //LinkedHashMap中的结点实现
    Entry<K,V> before, after;   //这里多了一个指向前一个结点和后一个结点的引用
    Entry(int hash, K key, V value, Node<K,V> next) {
        super(hash, key, value, next);
    }
}
```

1️⃣ 它的遍历`forEach`是依靠它的after遍历

```
    public void forEach(BiConsumer<? super K, ? super V> action) {
        if (action == null)
            throw new NullPointerException();
        int mc = modCount;
        for (LinkedHashMap.Entry<K,V> e = head; e != null; e = e.after)
            action.accept(e.key, e.value);
        if (modCount != mc)
            throw new ConcurrentModificationException();
    }
```

⭐ 在回头看看`HashSet`的源代码，`HashSet`的底层就是`HashMap`

```
public class HashSet<E>
    extends AbstractSet<E>
    implements Set<E>, Cloneable, java.io.Serializable
{

    private transient HashMap<E,Object> map;    //对，你没看错，底层直接用map来做事

    // 因为Set只需要存储Key就行了，所以说这个对象当做每一个键值对的共享Value，也就是虽然存放的只有一个数据，由于HashMap需要存放键值对，所以说它这里会创建一个空对象来顶替那个值
    private static final Object PRESENT = new Object();

    //直接构造一个默认大小为16负载因子0.75的HashMap
    public HashSet() {
        map = new HashMap<>();
    }

    ...

    //你会发现所有的方法全是替身攻击
    public Iterator<E> iterator() {
        return map.keySet().iterator();
    }

    public int size() {
        return map.size();
    }

    public boolean isEmpty() {
        return map.isEmpty();
    }
}
```

1️⃣ 主要看它的`add`方法，会发现它是直接调用put方法的，而且还加上了那个创建的空对象，编程替身攻击

> 通过观察`HashSet`的源码发现，`HashSet`几乎都在操作内部维护的一个`HashMap`，也就是说，`HashSet`只是一个表壳，而内部维护的`HashMap`才是灵魂！就像你进了公司，在外面花钱请别人帮你写公司的业务，你只需要坐着等别人写好然后你自己拿去交差就行了。所以说，`HashSet`利用了`HashMap`内部的数据结构，轻松地就实现了Set定义的全部功能！

```
    public boolean add(E e) {
        return map.put(e, PRESENT)==null;
    }
```

⭐ `TreeSet` 底层使用的就是`TreeMap`,`TreeMap`时根据`Key`来自动排序，`TreeMap`，是一个二叉查找树，在底层就只需要一个比较器就好了(可以看看数据结构的二叉树的定义)

```
public class TreeMap<K,V>
    extends AbstractMap<K,V>
    implements NavigableMap<K,V>, Cloneable, java.io.Serializable
{
    /**
     * The comparator used to maintain order in this tree map, or
     * null if it uses the natural ordering of its keys.
     *
     * @serial
     */
    private final Comparator<? super K> comparator; //比较器

    private transient Entry<K,V> root; //根节点
}
```

1️⃣ 在来看看`TreeSet`的大小

```
public class TreeSet<E> extends AbstractSet<E>
    implements NavigableSet<E>, Cloneable, java.io.Serializable
{
    //底层需要一个NavigableMap，就是自动排序的Map
    private transient NavigableMap<E,Object> m;

    //不用我说了吧
    private static final Object PRESENT = new Object();

    ...

    //直接使用TreeMap解决问题
    public TreeSet() {
        this(new TreeMap<E,Object>());
    }

    ...
}
```

⭐ Map中的其他方法

1️⃣ `compute`方法

```
    public static void main(String[] args) {
        Map<Integer, String> map = new HashMap<>();
        map.put(1, "A");
        map.put(2, "B");
        map.compute(1, (k, v) -> {   //compute会将指定Key的值进行重新计算，若Key不存在，v会返回null
            return v+"M";     //这里返回原来的value+M
        });
        map.computeIfPresent(1, (k, v) -> {   //当Key存在时存在则计算并赋予新的值
            return v+"M";     //这里返回原来的value+M
        });
        System.out.println(map);
    }
```

值得注意的时`compute`的传参是一个`BiFunction`带`Bi`一般都表示要传入两个对象

也可以使用`computeIfAbsent`，当不存在Key时，计算并将键值对放入Map中

```
public static void main(String[] args) {
    Map<Integer, String> map = new HashMap<>();
    map.put(1, "A");
    map.put(2, "B");
    map.computeIfAbsent(0, (k) -> {   //若不存在则计算并插入新的值
        return "M";     //这里返回M
    });
    System.out.println(map);
}
```

2️⃣ `mergae` 处理数据

```
public static void main(String[] args) {
    List<Student> students = Arrays.asList(
            new Student("yoni", "English", 80),
            new Student("yoni", "Chiness", 98),
            new Student("yoni", "Math", 95),
            new Student("taohai.wang", "English", 50),
            new Student("taohai.wang", "Chiness", 72),
            new Student("taohai.wang", "Math", 41),
            new Student("Seely", "English", 88),
            new Student("Seely", "Chiness", 89),
            new Student("Seely", "Math", 92)
    );
    Map<String, Integer> scoreMap = new HashMap<>();
    //merge方法可以对重复键的值进行特殊操作，比如我们想计算某个学生的所有科目分数之后，那么就可以像这样：
    students.forEach(student -> scoreMap.merge(student.getName(), student.getScore(), Integer::sum));
    scoreMap.forEach((k, v) -> System.out.println("key:" + k + "总分" + "value:" + v));
}

```

```
static class Student {
    private final String name;
    private final String type;
    private final int score;

    public Student(String name, String type, int score) {
        this.name = name;
        this.type = type;
        this.score = score;
    }

    public String getName() {
        return name;
    }

    public int getScore() {
        return score;
    }

    public String getType() {
        return type;
    }
}
```

3 `replace`可以替换值

```
public static void main(String[] args) {
    Map<Integer , String> map = new HashMap<>();
    map.put(0, "单走");
    map.replace(0, ">>>");   //直接替换为新的
    map.replace(1,"A","C"); //只有key和value都满足才替换
    System.out.println(map);
}
```

```
    public static void main(String[] args) {
        Map<Integer, String> map = new HashMap<>();
        map.put(1, "A");
        map.put(2, "B");
        map.remove(1,"A"); //只有key和value都满足才替换
        System.out.println(map);
    }
```

4 `getOrDefault`方法

```
    public static void main(String[] args) {
        Map<Integer, String> map = new HashMap<>();
        map.put(1, "A");
        map.put(2, "B");
        map.getOrDefault(2,"C"); //如果存在2就返回2的值，如果不存在就返回C
    }
```