抛出问题

想找一个元素，无论是在数组、链表、树、图中都需要进行遍历，而有没有一种方式是不通过遍历，直接获取到元素的数据结构呢?

定义

Hash表也称散列表，也有直接译作哈希表，Hash表是一种根据关键字值（key - value）而直接进行访问的数据结构。

散列技术是在记录的存储位置和它的关键字之间建立一个确定的对应关系f，使得每个关键字key对应一个存储位置f(key)。建立了关键字

```
存储位置 = f（关键字）
```

采用散列技术将记录存在在一块连续的存储空间中，这块连续存储空间称为散列表或哈希表。那么，关键字对应的记录存储位置称为散

哈希冲突

通过哈希函数，不同的关键字，产生的存储位置是同一个。这种称为哈希冲突。

散列查找速度快的原因

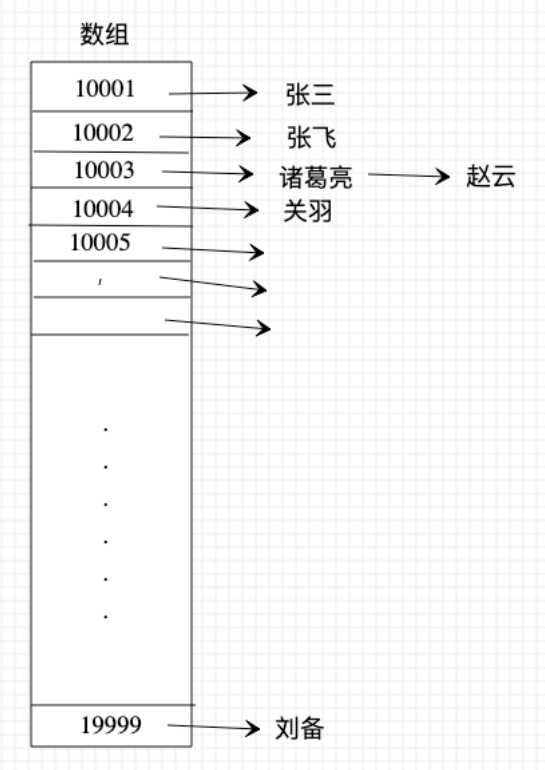
假设有一个大数组，里面存储的是人名，如果要找到 张三 。那么就需要遍历整个数组。时间复杂度为O(n)
而散列就是一种可以直接定位到 张三 的计算，时间复杂度为O(1)

所有的对象都有一个hashcode值，假定下面人名对应的 hashcode 是对应的编号

```
张三 -> 10001
张飞 -> 10002
诸葛亮 -> 10003
关羽 -> 10004
...
刘备 -> 19999
```

```
赵云 -> 10003
...
```

散列表存储的结构图



查找张飞：计算张飞的hashcode值，计算结果是10002 。 数组中位置，只有一条，那么就是他
查找赵云：计算赵云的hashcode值，计算结果是1003。数组中位置上有两个值，发现第一个是 诸葛亮，那么继续找下一个 结果就是赵云

散列基于数组，通过把关键字映射到数组的某个下标来加快查找速度，但是又和数组、链表、树等数据结构不同，在这些数据结构中查构，也就是O(N)的时间级，但是对于哈希表来说，只是O(1)的时间级。

注意：这里有个重要的问题就是如何把关键字转换为数组的下标，这个转换的函数称为哈希函数（也称散列函数），转换的过程称为哈希化

散列函数评判标准

一个好的散列函数，应该具备以下几点：

1. 给定一个key，能够很快计算出hashcode
2. 分布均匀，尽量让不同的key产生不同的hashcode，减少哈希冲突（碰撞）
3. 保证负载因子的平衡性

```
负载因子的概念：负载因子 = 元素个数（散列表中的元素个数） / 散列表大小（散列表的长度）
```

- 负载因子越大表示散列表装填程度越高，空间利用率越高，但对应的查找效率就越低。
- 负载因子越小表示散列表装填程度越低，空间利用率越低，但对应的查找效率就越高

代码实现

```
package 散列;
```

```java

public interface IHashMap {
    public void put(String key,Object object);
    public Object get(String key);
}
```

```java
package 散列;

//键值对
public class Entry {

    public Entry(Object key, Object value) {
        super();
        this.key = key;
        this.value = value;
    }
    public Object key;
    public Object value;
    @Override
    public String toString() {
        return "[key=" + key + ", value=" + value + "]";
    }

}
```

```java
package 散列;

import java.util.LinkedList;

public class MyHashMap implements IHashMap {
    LinkedList<Entry>[] values = new LinkedList[2000];

    @Override
    public void put(String key, Object object) {
        // 拿到hashcode
        int hashcode = hashcode(key);

        // 找到对应的LinkedList
        LinkedList<Entry> list = values[hashcode];
        // 如果LinkedList是null，则创建一个LinkedList
        if (null == list) {
            list = new LinkedList<>();
            values[hashcode] = list;
        }

        // 判断该key是否已经有对应的键值对
        boolean found = false;
        for (Entry entry : list) {
            // 如果已经有了，则替换掉
            if (key.equals(entry.key)) {
                entry.value = object;
                found = true;
                break;
```

```
            }
        }

        // 如果没有已经存在的键值对，则创建新的键值对
        if (!found) {
            Entry entry = new Entry(key, object);
            list.add(entry);
        }

    }

    @Override
    public Object get(String key) {
        // 获取hashcode
        int hashcode = hashcode(key);
        // 找到hashcode对应的LinkedList
        LinkedList<Entry> list = values[hashcode];
        if (null == list){
            return null;
        }

        Object result = null;

        // 挨个比较每个键值对的key，找到匹配的，返回其value
        for (Entry entry : list) {
            if (entry.key.equals(key)) {
                result = entry.value;
                break;
            }
        }

        return result;

    }

    private static int hashcode(String str) {
        // TODO Auto-generated method stub
        if (0 == str.length()) {
            return 0;
        }
        int hashcode = 0;
        char[] cs = str.toCharArray();
        for (int i = 0; i < cs.length; i++) {
            hashcode += cs[i];
        }
        hashcode *= 23;
        // 取绝对值
        hashcode = hashcode < 0 ? 0 - hashcode : hashcode;
        // 落在0-1999之间
        hashcode %= 2000;

        return hashcode;
    }

```

```
    public static void main(String[] args) {
        MyHashMap map = new MyHashMap();

         map.put("t", "坦克");
         map.put("adc", "物理");
         map.put("apc", "魔法");
         map.put("t", "坦克2");

         System.out.println(map.get("adc"));

         System.out.println(map);

        System.out.println(MyHashMap.hashcode("name=hero-2387"));
        System.out.println(MyHashMap.hashcode("name=hero-5555"));

    }

     @Override
     public String toString() {
         LinkedList<Entry> result = new LinkedList();

         for (LinkedList<Entry> linkedList : values) {
             if (null == linkedList) {
                 continue;
             }
             result.addAll(linkedList);
         }
         return result.toString();
     }

}
```

处理hash值冲突的几种方法

开放地址法

开发地址法中，若数据项不能直接存放在由哈希函数所计算出来的数组下标时，就要寻找其他的位置。分别有三种方法：线性探测、二

线性探测

在线性探测中，它会线性的查找空白单元。比如如果 5421 是要插入数据的位置，但是它已经被占用了，那么就使用5422，如果5422t推，数组下标依次递增，直到找到空白的位置。这就叫做线性探测，因为它沿着数组下标一步一步顺序的查找空白单元

二次探测

二测探测是防止聚集产生的一种方式，思想是探测相距较远的单元，而不是和原始位置相邻的单元。

线性探测中，如果哈希函数计算的原始下标是x, 线性探测就是x+1, x+2, x+3, 以此类推；而在二次探测中，探测的过程是x+1, x+4, x+9, 是步数的平方。二次探测虽然消除了原始的聚集问题，但是产生了另一种更细的聚集问题，叫二次聚集：比如讲184，302，420和544依次302需要以1为步长探测，420需要以4为步长探测， 544需要以9为步长探测。只要有一项其关键字映射到7，就需要更长步长的探测，这个严重的问题，但是二次探测不会经常使用，因为还有好的解决方法，比如再哈希法

再哈希法

为了消除原始聚集和二次聚集，我们使用另外一种方法：再哈希法。

我们知道二次聚集的原因是，二测探测的算法产生的探测序列步长总是固定的：1,4，9,16以此类推。那么我们想到的是需要产生一种依键字都一样，那么，不同的关键字即使映射到相同的数组下标，也可以使用不同的探测序列。

方法是把关键字用不同的哈希函数再做一遍哈希化，用这个结果作为步长。对于指定的关键字，步长在整个探测中是不变的，不过不同

第二个哈希函数必须具备如下特点：

一、和第一个哈希函数不同

二、不能输出0（否则，将没有步长，每次探测都是原地踏步，算法将陷入死循环）。

专家们已经发现下面形式的哈希函数工作的非常好：stepSize = constant - key % constant; 其中constant是质数，且小于数组容量。

再哈希法要求表的容量是一个质数，假如表长度为15(0-14)，非质数，有一个特定关键字映射到0，步长为5，则探测序列是0,5,10,0,5,试这三个单元，所以不可能找到某些空白单元，最终算法导致崩溃。如果数组容量为13, 质数，探测序列最终会访问所有单元。即0,5,10,2,7一个空位，就可以探测到它

链地址法

在开放地址法中，通过再哈希法寻找一个空位解决冲突问题，另一个方法是在哈希表每个单元中设置链表（即链地址法），某个数据项表的单元，而数据项本身插入到这个单元的链表中。其他同样映射到这个位置的数据项只需要加到链表中，不需要在原始的数组中寻找空位

桶

类似于链地址法，它是在每个数据项中使用子数组，而不是链表。这样的数组称为桶。

这个方法显然不如链表有效，因为桶的容量不好选择，如果容量太小，可能会溢出，如果太大，又造成性能浪费，而链表是动态分配的