

java.util.Set 继承 Collection 接口

特点: 1. 存入元素不重复

2. 元素无序, 无索引方法, 不能用普通for遍历 (但可用迭代器或增强for)

方法与 Collection 中一致

Set 集合不允许元素重复的原理:

```
// 创建 HashSet 集合对象
HashSet<String> set = new HashSet<>();
String s1 = new String("abc");
String s2 = new String("abc");
set.add(s1);
set.add(s2);
set.add("重地");
set.add("通话");
set.add("abc");
System.out.println(set); // [重地, 通话, abc]
```

Set 集合在调用 add 方法的时候, add 方法会调用元素的 hashCode 方法和 equals 方法, 判断元素是否重复

set.add(s1);
add 方法会调用 s1 的 hashCode 方法, 计算字符串 "abc" 的哈希值, 哈希值是 96354
在集合中找有没有 96354 这个哈希值的元素, 发现没有
就会把 s1 存储到集合中

set.add(s2);
add 方法会调用 s2 的 hashCode 方法, 计算字符串 "abc" 的哈希值, 哈希值是 96354
在集合中找有没有 96354 这个哈希值的元素, 发现有 (哈希冲突)
s2 会调用 equals 方法和哈希值相同的元素进行比较 s2.equals(s1), 返回 true
两个元素的哈希值相同, equals 方法返回 true, 认定两个元素相同
就不会把 s2 存储到集合中

set 集合存储元素不重复的元素
前提: 存储的元素必须重写 hashCode 方法和 equals 方法



set.add("重地");
add 方法会调用 "重地" 的 hashCode 方法, 计算字符串 "重地" 的哈希值, 哈希值是 1179395
在集合中找有没有 1179395 这个哈希值的元素, 发现没有
就会把 "重地" 存储到集合中

set.add("通话");
add 方法会调用 "通话" 的 hashCode 方法, 计算字符串 "通话" 的哈希值, 哈希值是 1179395
在集合中找有没有 1179395 这个哈希值的元素, 发现有 (哈希冲突)
"通话" 会调用 equals 方法和哈希值相同的元素进行比较 "通话".equals("重地"), 返回 false
两个元素的哈希值相同, equals 方法返回 false, 认定两个元素不同
就会把 "通话" 存储到集合中

java.util.HashSet 哈希表结构, 实现了 Set 接口

特点: 1. 存入元素不重复

2. 元素无序, 无索引方法, 不能用普通for遍历 (但可用迭代器和增强for)

3. 有序集, 存储元素和取出元素的顺序可能不一致

4. 底层是一个哈希表结构 (查询速度非常快)

哈希值: + 进制整数, 由系统随机给出 (对象的地址值, 是模拟得出的逻辑地址, 不是数据实际存储的物理地址)

Object 类中有一个方法, 能获取对象的哈希值:

public native int hashCode(): 返回对象的哈希值.

native 代表方法调用的是本地操作系统的方法

String 类的哈希值: String 类重写 Object 类的 hashCode 方法

HashSet 集合存储数据的结构 (哈希表)

jdk1.8 版本之前: 哈希表 = 数组 + 链表

jdk1.8 版本之后:

哈希表 = 数组 + 链表;

哈希表 = 数组 + 红黑树 (提高查询的速度)

哈希表的特点: 速度快

数组结构: 把元素进行了分组, (相同哈希值的元素是一组) 链表/红黑树结构把相同哈希值的元素连接到一起

存储数据到集合中
先计算元素的哈希值



96354 在数组中的存储位置

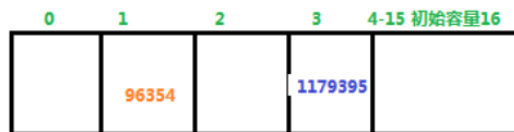


1179395



1179395

两个元素不同,
但是哈希值相同
哈希冲突

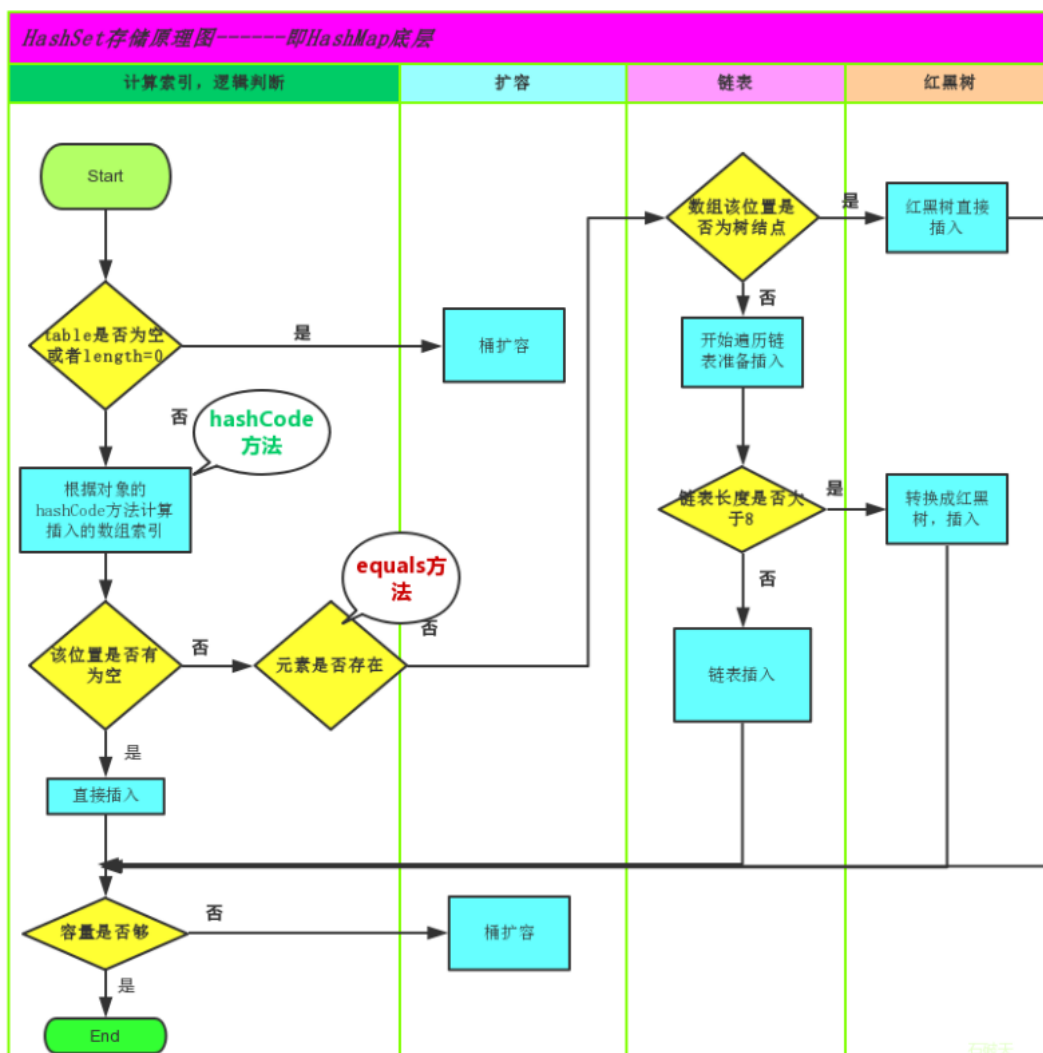
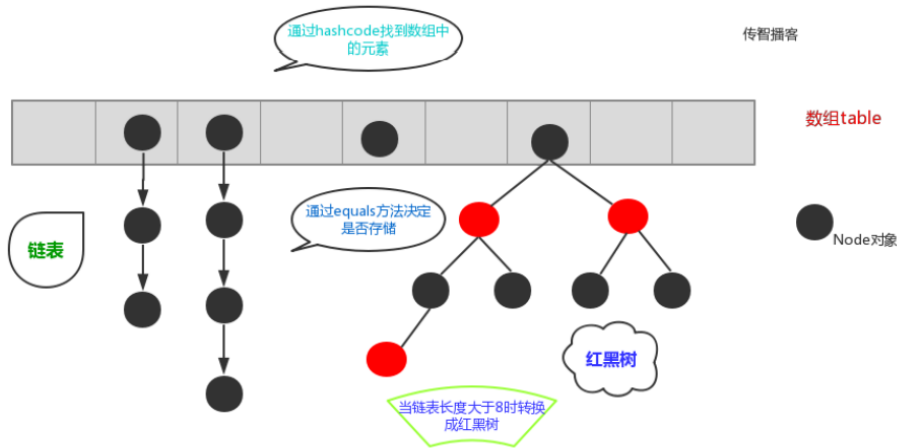


如果链表的长度超过了8位
那么就会把链表转换
为红黑树 (提高查询的速度)

HashSet集合存储数据的结构(哈希表HashMap):速度快

在JDK1.8之前，哈希表底层采用数组+链表实现，即使用链表处理冲突，同一hash值的链表都存储在一个链表里。但是当位于一个桶中的元素较多，即hash值相等的元素较多时，通过key值依次查找的效率较低。而JDK1.8中，哈希表存储采用数组+链表+红黑树实现，当链表长度超过阈值（8）时，将链表转换为红黑树，这样大大减少了查找时间。

简单的来说，哈希表是由数组+链表+红黑树（JDK1.8增加了红黑树部分）实现的，如下图所示。



总而言之，JDK1.8引入红黑树大程度优化了HashMap的性能，那么对于我们来讲保证HashSet集合元素的唯一，其实就是根据对象的hashCode和equals方法决定的。如果我们往集合中存放自定义的对象，那么保证其唯一，就必须复写hashCode和equals方法建立属于当前对象的比较方式。

用HashSet存放自定义类型元素时,需要重写对象的hashCode和equals方法,建立比较方式,才能保证HashSet集合中的对象唯一。

创建自定义Student类

```
public class Student {
    private String name;
    private int age;

    public Student() {
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
        if (o == null || getClass() != o.getClass())
            return false;
        Student student = (Student) o;
        return age == student.age &&
            Objects.equals(name, student.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age);
    }
}
```

```
public class HashSetDemo2 {
    public static void main(String[] args) {
        //创建集合对象 该集合中存储 Student类型对象
        HashSet<Student> stuSet = new HashSet<Student>();
        //存储
        Student stu = new Student("于谦", 43);
        stuSet.add(stu);
        stuSet.add(new Student("郭德纲", 44));
        stuSet.add(new Student("于谦", 43));
        stuSet.add(new Student("郭麒麟", 23));
        stuSet.add(stu);

        for (Student stu2 : stuSet) {
            System.out.println(stu2);
        }
    }
}

执行结果:
Student [name=郭德纲, age=44]
Student [name=于谦, age=43]
Student [name=郭麒麟, age=23]
```

java.util.LinkedHashSet 链表和哈希表组合的一个数据存储结构,继承了HashSet集合
特点: 底层是一个哈希表+链表(记录元素存储顺序,保证元素有序)

```
/*
    java.util.LinkedHashSet集合 extends HashSet集合
    LinkedHashSet集合特点:
        底层是一个哈希表(数组+链表/红黑树)+链表:多了一条链表(记录元素的存储顺序),保证元素有序
*/
public class Demo04LinkedHashSet {
    public static void main(String[] args) {
        HashSet<String> set = new HashSet<>();
        set.add("www");
        set.add("abc");
        set.add("abc");
        set.add("itcast");
        System.out.println(set);//[abc, www, itcast] 无序,不允许重复

        LinkedHashSet<String> linked = new LinkedHashSet<>();
        linked.add("www");
        linked.add("abc");
        linked.add("abc");
        linked.add("itcast");
        System.out.println(linked);//[www, abc, itcast] 有序,不允许重复
    }
}
```


可变参数:

使用前提: 当方法的参数列表数据类型已经确定, 但参数的个数不确定, 可在定义方法时使用

修饰符 返回值类型 方法名(参数类型... 形参名){

//...

}

实际上与 修饰符 返回值类型 方法名(参数类型[] 形参名){

//...

}

完全等价, 但后者用时必须传递数组, 前者传数据即可)

原理: 底层是一个数组, 根据传参个数不同, 会创建不同长度的数组来储存
传参个数可以是 0, 1, 2, ... 多个

注意: 1. 一个方法的参数列表只能有一个可变参数

2. 若方法的参数有多个, 则可变参数必须在参数列表的末尾