一、概述

1、概念

- **==** ： 该操作符生成的是一个boolean结果，它计算的是**操作数的值之间的关系**
- **equals** ： Object 的 **实例方法**，比较两个对象的**content**是否相同
- **hashCode** ： Object 的 **native方法**，获取对象的**哈希值**，用于确定该对象在哈希表中的索引位置，它实际上是一个int型整数

二、关系操作符 ==

1、操作数的值

- 基本数据类型变量

  在Java中有八种基本数据类型：

  浮点型：float(4 byte), double(8 byte)

  整型：byte(1 byte), short(2 byte), int(4 byte) , long(8 byte)

  字符型: char(2 byte)

  布尔型: boolean(JVM规范没有明确规定其所占的空间大小，仅规定其只能够取字面值“true”和“false”)

  对于这八种基本数据类型的变量，变量直接存储的是“值”。因此，在使用关系操作符 == 来进行比较时，比较的就是“值”本身。要注意的是，**浮点型和整型都是有符号类型的（最高位仅用于表示正负，不参与计算【以 byte 为例，其范围为 -2^7 ~ 2^7 - 1，-0 即-128】），而char是无符号类型的（所有位均参与计算，所以char类型取值范围为 0~2^16-1）**。

- 引用类型变量
  在Java中，**引用类型的变量存储的并不是“值”本身，而是与其关联的对象在内存中的地址**。比如下面这行代码，

```
1       String str1;
```

这句话声明了一个引用类型的变量，此时它并没有和任何对象关联。
而通过 new 来产生一个对象，并将这个对象和str1进行绑定：

```
str1= new String("hello");
```

那么 str1 就指向了这个对象，此时引用变量str1中存储的是它指向的对象在内存中的存储地址，并不是“值”本身，也就是说并不是直接存储的字符串”hello”。这里面的引用和 C/C++ 中的指针很类似。

2、小结

因此，对于关系操作符 == ：

- 若操作数的类型是**基本数据类型**，则该关系操作符判断的是左右两边操作数的**值**是否相等
- 若操作数的类型是**引用数据类型**，则该关系操作符判断的是左右两边操作数的**内存地址**是否相同。**也就是说，若此时返回true,则该操作符作用的一定是同一个对象。**

三、equals方法

1、来源

equals方法是基类Object中的实例方法，因此对所有继承于Object的类都会有该方法。

在 Object 中的声明：

```
    public boolean equals(Object obj) {}
```

2、equals方法的作用

初衷 ： 判断两个对象的 **content** 是否相同

为了更直观地理解equals方法的作用，我们先看Object类中equals方法的实现。

```
public boolean equals(Object obj) {
  return (this == obj);
}
```

很显然，在Object类中，equals方法是用来比较两个对象的引用是否相等，即是否指向同一个对象。

但我们都知道，下面代码输出为 true:

```
public class Main {
    public static void main(String[] args) {
        String str1 = new String("hello");
        String str2 = new String("hello");

        System.out.println(str1.equals(str2));
    }
}
```

原来是 String 类重写了 equals 方法：

```
public boolean equals(Object anObject) {    // 方法签名与 Object类 中的一致
    if (this == anObject) {     // 先判断引用是否相同(是否为同一对象),
        return true;
    }
    if (anObject instanceof String) {    // 再判断类型是否一致,
        // 最后判断内容是否一致.
        String anotherString = (String)anObject;
        int n = count;
        if (n == anotherString.count) {
        char v1[] = value;
        char v2[] = anotherString.value;
        int i = offset;
        int j = anotherString.offset;
        while (n-- != 0) {
            if (v1[i++] != v2[j++])
            return false;
        }
        return true;
        }
    }
    return false;
}
```

即对于诸如“字符串比较时用的什么方法,内部实现如何？”之类问题的回答即为：

使用equals方法，内部实现分为三个步骤：

- 先 **比较引用是否相同(是否为同一对象)**,
- 再 **判断类型是否一致（是否为同一类型）**,
- 最后 **比较内容是否一致**

Java 中所有内置的类的 equals 方法的实现步骤均是如此，特别是诸如 Integer，Double 等包装器类。

3、equals 重写原则

对象内容的比较才是设计equals()的真正目的，Java语言对equals()的要求如下，这些要求是重写该方法时必须遵循的：

- **对称性**：如果x.equals(y)返回是“true”，那么y.equals(x)也应该返回是“true”；
- **自反性**：x.equals(x)必须返回是“true”；
- **类推性**：如果x.equals(y)返回是“true”，而且y.equals(z)返回是“true”，那么z.equals(x)也应该返回是“true”；
- **一致性**：如果x.equals(y)返回是“true”，只要x和y内容一直不变，不管你重复x.equals(y)多少次，返回都是“true”；
- **对称性**：如果x.equals(y)返回是“true”，那么y.equals(x)也应该返回是“true”。
- 任何情况下，**x.equals(null)【应使用关系比较符 ==】，永远返回是“false”；x.equals(和x不同类型的对象)永远返回是“false”**

4、小结

因此，对于 equals 方法：

- 其**本意** 是 **比较两个对象的 content 是否相同**
- 必要的时候，我们需要重写该方法，避免违背本意，且要遵循上述原则

四、hashCode 方法

1、hashCode 的来源

hashCode 方法是基类Object中的 **实例native方法**，因此对所有继承于Object的类都会有该方法。

在 Object类 中的声明（native方法暗示这些方法是有实现体的，但并不提供实现体，因为其实现体是由非java语言在外面实现的）：

```
1        public native int hashCode();
```

2、哈希相关概念

我们首先来了解一下哈希表：

- **概念 ：** Hash 就是把任意长度的输入(又叫做预映射， pre-image)，通过散列算法，变换成固定长度的输出(**int**)，该输出就是散列值。这种转换是一种 **压缩映射**，也就是说，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，从而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。**

- **应用–数据结构 ：** 数组的特点是：寻址容易，插入和删除困难; 而链表的特点是：寻址困难，插入和删除容易。那么我们能不能综合两者的特性，做出一种寻址容易，插入和删除也容易的数据结构？答案是肯定的，这就是我们要提起的哈希表，哈希表有多种不同的实现方法，我接下来解释的是最常用的一种方法——**拉链法**，我们可以理解为 **“链表的数组”** ，如图：

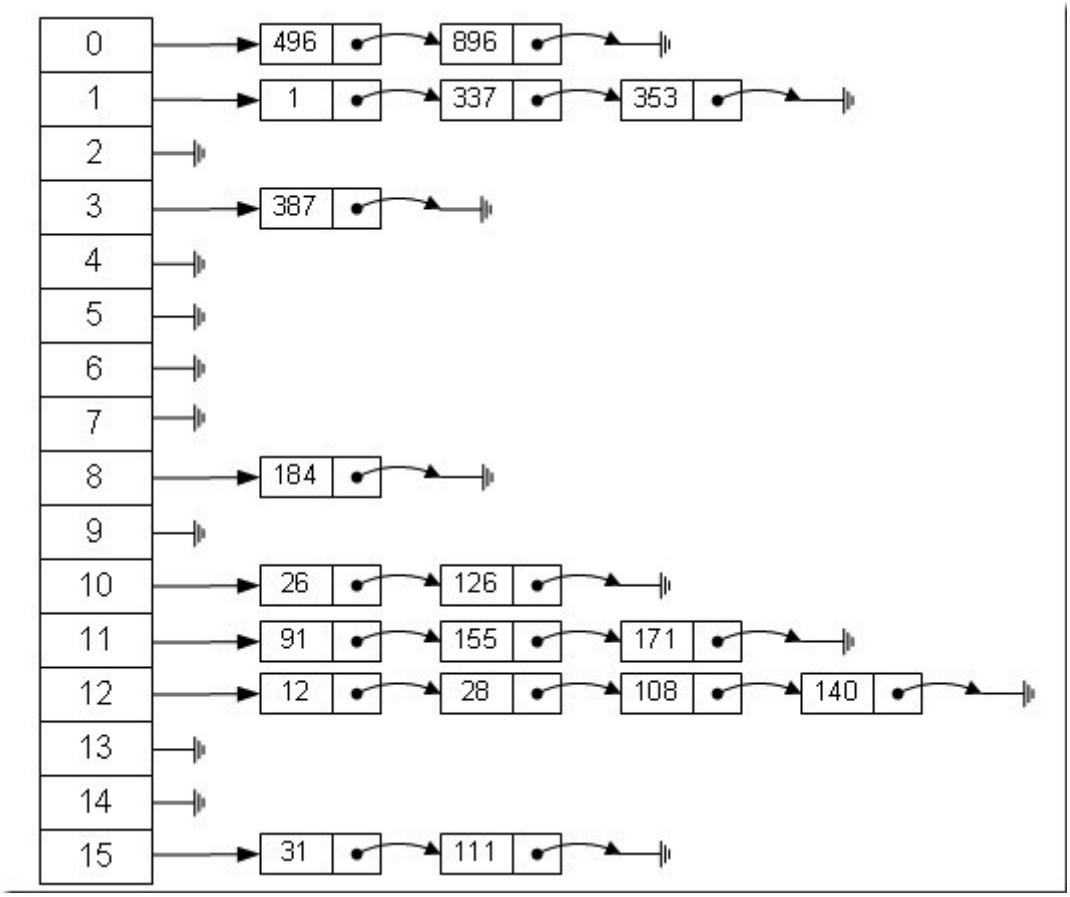

图1 哈希表示例

左边很明显是个数组，数组的每个成员是一个链表。该数据结构所容纳的所有元素均包含一个指针，用于元素间的链接。我们根据元素的自身特征把元素分配到不同的链表中去，也是根据这些特征，找到正确的链表，再从链表中找出这个元素。**其中，将根据元素特征计算元素数组下标的方法就是散列法。**

- **拉链法的适用范围 ：** 快速查找，删除的基本数据结构，通常需要总数据量可以放入内存。

- **要点 ：**
  hash函数选择，针对字符串，整数，排列，具体相应的hash方法；

碰撞处理，一种是open hashing，也称为拉链法，另一种就是closed hashing，也称开地址法，opened addressing。

3、hashCode 简述

在 Java 中，**由 Object 类定义的 hashCode 方法会针对不同的对象返回不同的整数。（这是通过将该对象的内部地址转换成一个整数来实现的，但是 JavaTM 编程语言不需要这种实现技巧）。**

hashCode 的常规协定是：

- 在 Java 应用程序执行期间，在对同一对象多次调用 hashCode 方法时，必须一致地返回相同的整数，前提是将对象进行 equals 比较时所用的信息没有被修改。从某一应用程序的一次执行到同一应用程序的另一次执行，该整数无需保持一致。
- 如果根据 equals(Object) 方法，两个对象是相等的，那么对这两个对象中的每个对象调用 hashCode 方法都必须生成相同的整数结果。
- 如果根据 equals(java.lang.Object) 方法，两个对象不相等，那么对这两个对象中的任一对象上调用 hashCode 方法 **不要求** 一定生成不同的整数结果。但是，程序员应该意识到，为不相等的对象生成不同整数结果可以提高哈希表的性能。

要想进一步了解 hashCode 的作用，我们必须先要了解Java中的容器，**因为 HashCode 只是在需要用到哈希算法的数据结构中才有用，比如 HashSet, HashMap 和 Hashtable。**

Java中的集合（Collection）有三类，一类是List，一类是Queue，再有一类就是Set。 前两个集合内的元素是有序的，元素可以重复；最后一个集合内的元素无序，但元素不可重复。

那么, 这里就有一个比较严重的问题：要想保证元素不重复，可两个元素是否重复应该依据什么来判断呢？ 这就是 **Object.equals** 方法了。但是，如果每增加一个元素就检查一次，那么当元素很多时，后添加到集合中的元素比较的次数就非常多了。 也就是说，如果集合中现在已经有1000个元素，那么第1001个元素加入集合时，它就要调用1000次equals方法。这显然会大大降低效率。于是，Java采用了**哈希表的原理**。 这样，我们对每个要存入集合的元素使用哈希算法算出一个值，然后根据该值计算出元素应该在数组的位置。所以，当集合要添加新的元素时，可分为两个步骤：

- **先调用这个元素的 hashCode 方法，然后根据所得到的值计算出元素应该在数组的位置。如果这个位置上没有元素，那么直接将它存储在这个位置上；**
- **如果这个位置上已经有元素了，那么调用它的equals方法与新元素进行比较：相同的话就不存了，否则，将其存在这个位置对应的链表中（Java 中 HashSet, HashMap 和 Hashtable的实现总将元素放到链表的表头）。**

4、equals 与 hashCode

前提： 谈到hashCode就不得不说equals方法，二者均是Object类里的方法。由于Object类是所有类的基类，所以一切类里都可以重写这两个方法。

- **原则 1 ：** 如果 x.equals(y) 返回 “true”，那么 x 和 y 的 hashCode() 必须相等 ；
- **原则 2 ：** 如果 x.equals(y) 返回 “false”，那么 x 和 y 的 hashCode() 有可能相等，也有可能不等 ；
- **原则 3 ：** 如果 x 和 y 的 hashCode() 不相等，那么 x.equals(y) 一定返回 “false” ；
- **原则 4 ：** **一般来讲，equals 这个方法是给用户调用的，而 hashcode 方法一般用户不会去调用 ；**
- **原则 5 ：** **当一个对象类型作为集合对象的元素时，那么这个对象应该拥有自己的equals()和hashCode()设计，而且要遵守前面所说的几个原则。**

5、实现例证

hashCode()在object类中定义如下：

```
1  public native int hashCode();
```

说明是一个本地方法，它的实现是根据本地机器相关的。

String 类是这样重写它的：

```
1  public final class String
2      implements java.io.Serializable, Comparable<String>, CharSequence
3  {
4      /** The value is used for character storage. */
5      private final char value[];     //成员变量1
6
7      /** The offset is the first index of the storage that is used. */
8      private final int offset;      //成员变量2
9
10     /** The count is the number of characters in the String. */
11     private final int count;       //成员变量3
12
13    /** Cache the hash code for the string */
14     private int hash; // Default to 0    //非成员变量
15
16     public int hashCode() {
17     int h = hash;
18         int len = count;          //用到成员变量3
19     if (h == 0 && len > 0) {
20         int off = offset;          //用到成员变量2
21         char val[] = value;        //用到成员变量1
22
```

```
23                 for (int i = 0; i < len; i++) {
24                     h = 31*h + val[off++];       //递推公式
25                 }
26                 hash = h;
27             }
28             return h;
29         }
    }
```

对程序的解释：`h = s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]`，由此可以看出，对象的hash地址不一定是实际的内存地址。

五、小结

- **hashcode是系统用来快速检索对象而使用**
- **equals方法本意是用来判断引用的对象是否一致**
- **重写equals方法和hashcode方法时，equals方法中用到的成员变量也必定会在hashcode方法中用到,只不过前者作为比较项，后者作为生成摘要的信息项，本质上所用到的数据是一样的，从而保证二者的一致性**