哈希函数

■ 哈希表中哈希函数的实现步骤大概如下

1. 先生成 key 的哈希值（必须是整数）
2. 再让 key 的哈希值跟数组的大小进行相关运算，生成一个索引值

```
public int hash(Object key) {
    return hash_code(key) % table.length;
}
```

■ 为了提高效率，可以使用 & 位运算取代 % 运算【前提：将数组的长度设计为 2 的幂（2^n）】

```
public int hash(Object key) {
    return hash_code(key) & (table.length - 1);
}
```

与操作可以保证所以不会超过数组长度

1100 1010	1011 1100
& 1111	& 1111
1010	1100

■ 良好的哈希函数

□ 让哈希值更加均匀分布 → 减少哈希冲突次数 → 提升哈希表的性能

如何生成key的哈希值

- key 的常见种类可能有
  - 整数、浮点数、字符串、自定义对象
  - 不同种类的 key，哈希值的生成方式不一样，但目标是一致的
- ✓ 尽量让每个 key 的哈希值是唯一的
- ✓ 尽量让 key 的所有信息参与运算

- 在Java中，HashMap 的 key 必须实现 hashCode、equals 方法，也允许 key 为 `null`

- 整数
  - 整数值当做哈希值
  - 比如 10 的哈希值就是 10

```
public static int hashCode(int value) {
    return value;
}
```

- 浮点数
  - 将存储的二进制格式转为整数值

```
public static int hashCode(float value) {
    return floatToIntBits(value);
}
```

Long和Double的哈希值

```
public static int hashCode(long value) {
    return (int)(value ^ (value >>> 32));
}
```

```
public static int hashCode(double value) {
    long bits = doubleToLongBits(value);
    return (int)(bits ^ (bits >>> 32));
}
```

- >>> 和 ^ 的作用是?
  - 高32bit 和 低32bit 混合计算出 32bit 的哈希值
  - 充分利用所有信息计算出哈希值

value	1111 1111 1111 1111 1111 1111 1111 1111 1011 0110 0011 1001 0110 1111 1100 1010
value >>> 32	0000 0000 0000 0000 0000 0000 0000 0000 1111 1111 1111 1111 1111 1111 1111 1111
value ^ (value >>> 32)	1111 1111 1111 1111 1111 1111 1111 1111 0100 1001 1100 0110 1001 0000 0011 0101

字符串的哈希值

- 整数 5489 是如何计算出来的?
  - $5 * 10^3 + 4 * 10^2 + 8 * 10^1 + 9 * 10^0$

- 字符串是由若干个字符组成的
  - 比如字符串 jack，由 j、a、c、k 四个字符组成（字符的本质就是一个整数）
  - 因此，jack 的哈希值可以表示为 $j * n^3 + a * n^2 + c * n^1 + k * n^0$，等价于 $[(j * n + a) * n + c] * n + k$
  - 在JDK中，乘数 n 为 31，为什么使用 31?
    - ✓ 31 是一个奇素数，JVM会将 31 * i 优化成 (i << 5) – i

```
String string = "jack";
int hashCode = 0;
int len = string.length();
for (int i = 0; i < len; i++) {
    char c = string.charAt(i);
    hashCode = 31 * hashCode + c;
}
```

```
String string = "jack";
int hashCode = 0;
int len = string.length();
for (int i = 0; i < len; i++) {
    char c = string.charAt(i);
    hashCode = (hashCode << 5) - hashCode + c;
}
```

关于31的探讨

- 31 * i = (2^5 – 1) * i = i * 2^5 – i = (i << 5) – i

- 31不仅仅是符合2^n – 1，它是个奇素数（既是奇数，又是素数，也就是质数）
- 素数和其他数相乘的结果比其他方式更容易产成唯一性，减少哈希冲突
- 最终选择31是经过观测分布结果后的选择

小码哥教育 SEEMYGO

自定义对象的哈希值

```
public class Person {
    private int age;
    private float height;
    private String name;
    private Car car;
    @Override
    public int hashCode() {
        int hash = Integer.hashCode(age);
        hash = 31 * hash + Float.hashCode(height);
        hash = 31 * hash + name.hashCode();
        hash = 31 * hash + car.hashCode();
        return hash;
    }
    @Override
    public boolean equals(Object obj) {
        if (obj == this) return true;
        if (obj == null || obj.getClass() != getClass()) return false;
        Person person = (Person) obj;
        return person.age == age && person.height == height
                && valueEquals(person.name, name)
                && valueEquals(person.car, car);
    }
    private boolean valueEquals(Object v1, Object v2) {
        return v1 == null ? v2 == null : v1.equals(v2);
    }
}
```

- 思考几个问题
  - 哈希值太大，整型溢出怎么办？
  - ✓ 不用作任何处理

小码哥教育 SEEMYGO

自定义对象作为key

- 自定义对象作为 key，最好同时重写 hashCode 、equals 方法
  - equals ：用以判断 2 个 key 是否为同一个 key
    - ✓ 自反性：对于任何非 null 的 x，x.equals(x)必须返回true
    - ✓ 对称性：对于任何非 null 的 x、y，如果 y.equals(x) 返回 true，x.equals(y) 必须返回 true
    - ✓ 传递性：对于任何非 null 的 x、y、z，如果 x.equals(y)、y.equals(z) 返回 true，那么x.equals(z) 必须返回 true
    - ✓ 一致性：对于任何非 null 的 x、y，只要 equals 的比较操作在对象中所用的信息没有被修改，多次调用 x.equals(y) 就会一致地返回 true，或者一致地返回 false
    - ✓ 对于任何非 null 的 x，x.equals(null) 必须返回 false
  - hashCode ：必须保证 equals 为 true 的 2 个 key 的哈希值一样
  - 反过来 hashCode 相等的 key，不一定 equals 为 true

- 不重写 hashCode 方法只重写 equals 会有什么后果?
  - ✓ 可能会导致 2 个 equals 为 true 的 key 同时存在哈希表中

小码哥教育 SEEMYGO

哈希值的进一步处理：扰动计算

```
private int hash(K key) {
    if (key == null) return 0;
    int h = key.hashCode();
    return (h ^ (h >>> 16)) & (table.length - 1);
}
```