[浅谈HashMap以及重写hashCode()和equals()方法](http://www.cnblogs.com/chn58/p/6544599.html)  
**因为，equals()方法只比较两个对象是否相同，相当于==，而不同的对象hashCode()肯定是不同，所以如果我们不是看对象，而只看对象的属性，则要重写这两个方法，如Integer和String他们的equals（）方法都是重写过了，都只是比较对象里的内容。**  
使用HashMap，如果key是自定义的类，就必须重写hashcode()和equals()。

一般先写hashCode再写equals。因为它返回的是对象的哈希值，那么不同的new出不同的对象，他们虽然名字一样但是哈希码可能会不一样。

为了阐明其作用，我们先来假设有如下一个Person类。

**[java]** [view plain](https://blog.csdn.net/qq_27093465/article/details/51556311) [copy](https://blog.csdn.net/qq_27093465/article/details/51556311)

1. **class** Person {
2. **public** Person(String name, **int** age) {
3. **this**.name = name;
4. **this**.age = age;
5. }
6. **private** String name;
7. **private** **int** age;
9. **public** String getName() {
10. **return** name;
11. }
12. **public** **void** setName(String name) {
13. **this**.name = name;
14. }
15. **public** **int** getAge() {
16. **return** age;
17. }
18. **public** **void** setAge(**int** age) {
19. **this**.age = age;
20. }
21. **public** String toString() {
22. **return** "{" + name + ", " + age + "}";
23. }
24. }

现在有很多Person类的对象需要存储，很自然联想到用HashSet来存储，于是乎，写了下面的程序来测试一下：

**[java]** [view plain](https://blog.csdn.net/qq_27093465/article/details/51556311) [copy](https://blog.csdn.net/qq_27093465/article/details/51556311)

1. **import** java.util.\*;
3. **public** **class** HashSetDemo {
4. **public** **static** **void** main(String[] args) {
5. Collection set = **new** HashSet();
6. set.add(**new** Person("张三", 21));
7. set.add(**new** Person("李四", 19));
8. set.add(**new** Person("王五", 22));
9. set.add(**new** Person("张三", 21));
10. sop(set);
11. }
12. **private** **static** **void** sop(Collection set) {
13. Iterator it = set.iterator();
14. **while** (it.hasNext()) {
15. Person p = it.next();
16. System.out.println(p.toString());
17. }
18. }
19. }

在存储的时候，我故意存了两个“21岁的张三”，我的本意是这是同一个人，也就是说set集合里面只需要出现一个“21岁的张三”，可事实是：



出现了两个一样的张三，为什么会这样呢？

其实，在往HashSet集合放置元素时，会根据其hashCode来判断两个元素是否一样，如果是一样，这后者覆盖前者。而hashCode默认是比较其地址值。于是，对于两个new 出来的“21岁的张三”，其地址值不一样，所以HashSet才将两个均加入其中。

1. **class** Person {
3. //都一样，变化的就是下面的
4. **public** **int** hashCode() {
5. **return** name.hashCode() + age \* 10;
6. }
7. **public** **boolean** equals(Object obj) {
8. **if** (!(obj **instanceof** Person))
9. **throw** **new** ClassCastException("类型不匹配");
10. Person p = (Person) obj;
11. **return** **this**.name.equals(p.getName()) && **this**.age == p.getAge();
12. }
13. }

此时，再运行重写，结果如下：



**总结：一般对于存放到Set集合或者Map中键值对的元素，需要按需要重写hashCode与equals方法，以保证唯一性！**

如果你重载了equals，比如说是基于对象的内容实现的，而保留hashCode的实现不变，那么很可能某两个对象明明是“相等”，而hashCode却不一样。  
这样，当你用其中的一个作为键保存到hashMap、hasoTable或hashSet中，再以“相等的”找另一个作为键值去查找他们的时候，则根本找不到(因为找的过程中是通过key值对应的hash值去寻找的)。(则在get可能找不到。)

对于每一个对象，通过其hashCode()方法可为其生成一个整形值（散列码），该整型值被处理后，将会作为数组下标，存放该对象所对应的Entry（存放该对象及其对应值）。   
equals()方法则是在HashMap中**插入值或查询时**会使用到。当HashMap中插入值或查询值对应的散列码与数组中的散列码相等时，则会通过equals方法比较key值是否相等，  
所以想以自建对象作为HashMap的key，必须重写该对象继承object的hashCode和equals方法。

put的源码的关键部分

[复制代码](javascript:void(0);)

for (Entry<K,V> e = table[i]; e != null; e = e.next) {

Object k;

if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {

V oldValue = e.value;

e.value = value;

e.recordAccess(this);

return oldValue;

}

}

modCount++;

addEntry(hash, key, value, i);

return null;

[复制代码](javascript:void(0);)

还有一点，**位置0上存放的一定是null**。

然后在遍历这个位置上的链表的过程中，如果发现在已经存在由equal函数确定的相等的Key，那么用新的Value替换掉老的Value，并返回老的Value。不然就在链表最后添加结点，并返回null。

看一下get的源码

[复制代码](javascript:void(0);)

public V get(Object key) {

if (key == null)

return getForNullKey();

Entry<K,V> entry = getEntry(key);

return null == entry ? null : entry.getValue();

}

[复制代码](javascript:void(0);)

再看getEntry的源码

[复制代码](javascript:void(0);)

final Entry<K,V> getEntry(Object key) {

if (size == 0) {

return null;

}

int hash = (key == null) ? 0 : hash(key);

for (Entry<K,V> e = table[indexFor(hash, table.length)];

e != null;

e = e.next) {

Object k;

if (e.hash == hash &&

((k = e.key) == key || (key != null && key.equals(k))))

return e;

}

return null;

}

[复制代码](javascript:void(0);)

意思很明确，就是先用hash函数确定在哪个位置，然后遍历这个位置上对应的链表，直到找到这个Key，然后返回value   
这里有个地方很关键，那就是**如何判判断相等**我们看到这里是靠equal函数来判断的，equal函数是所有类都会从Object类处继承的函数，

当我们在HashMap中存储我们自己定义的类的时候，默认的equal函数的行为可能不能符合我们的要求，所以需要重写。

总结：

1、如果两个对象相同（即用equals比较返回true），那么它们的hashCode值一定要相同；  
2、如果两个对象的hashCode相同，它们并不一定相同(即用equals比较返回false)

先写hashCode再写equals

再来一次：

# [重写equals就必须重写hashCode的原理分析](http://www.cnblogs.com/wang-meng/p/7501378.html)

因为最近在整理Java集合的源码， 所以今天再来谈谈这个古老的话题，因为后面讲HashMap会用到这个知识点， 所以重新梳理下。  
  
如果不被重写（原生Object）的hashCode和equals是什么样的？

1. 不被重写（原生）的hashCode值是根据内存地址换算出来的一个值。
2. 不被重写（原生）的equals方法是严格判断一个对象是否相等的方法（object1 == object2）。

为什么需要重写equals和hashCode方法？  
      在我们的业务系统中判断对象时有时候需要的不是一种严格意义上的相等，而是一种业务上的对象相等。在这种情况下，原生的equals方法就不能满足我们的需求了  
      所以这个时候我们需要重写equals方法，来满足我们的业务系统上的需求。那么为什么在重写equals方法的时候需要重写hashCode方法呢？

      我们先来看一下Object.hashCode的通用约定（摘自《Effective Java》第45页）

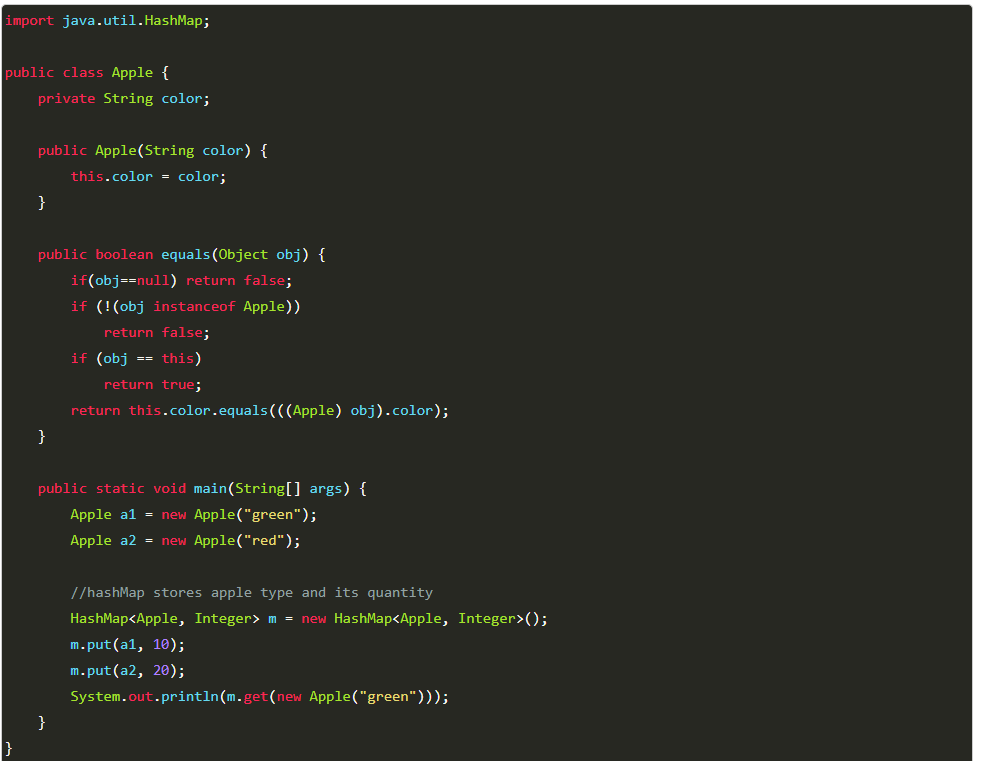
1. 在一个应用程序执行期间，如果一个对象的equals方法做比较所用到的信息没有被修改的话，那么，对该对象调用hashCode方法多次，它必须始终如一地返回 同一个整数。在同一个应用程序的多次执行过程中，这个整数可以不同，即这个应用程序这次执行返回的整数与下一次执行返回的整数可以不一致。
2. 如果两个对象根据equals(Object)方法是相等的，那么调用这两个对象中任一个对象的hashCode方法必须产生同样的整数结果。
3. 如果两个对象根据equals(Object)方法是不相等的，那么调用这两个对象中任一个对象的hashCode方法，不要求必须产生不同的整数结果。然而，程序员应该意识到这样的事实，对于不相等的对象产生截然不同的整数结果，有可能提高散列表（hash table）的性能。

**如果只重写了equals方法而没有重写hashCode方法的话，则会违反约定的第二条：相等的对象必须具有相等的散列码（hashCode）。**

**同时对于HashSet和HashMap这些基于散列值（hash）实现的类。**HashMap的底层处理机制是以数组的方法保存放入的数据的(Node<K,V>[] table)，其中的关键是数组下标的处理。**数组的下标是根据传入的元素hashCode方法的返回值再和特定的值异或决定的。**如果该数组位置上已经有放入的值了，且传入的键值相等则不处理，若不相等则覆盖原来的值，如果数组位置没有条目，则插入，并加入到相应的链表中。检查键是否存在也是根据hashCode值来确定的。所以如果不重写hashCode的话，**可能导致HashSet、HashMap不能正常的运作、**

    如果我们将某个自定义对象存到HashMap或者HashSet及其类似实现类中的时候，**如果该对象的属性参与了hashCode的计算，那么就不能修改该对象参数hashCode计算的属性了。**有可能会移除不了元素，导致内存泄漏。

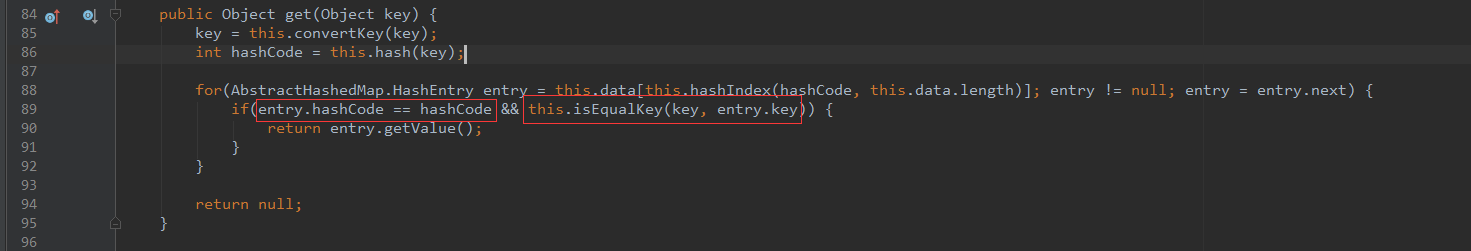
接着来看一个代码片段：



运行这段代码发现结果返回的是null。

再来看一下HashMap中的get源码：

get的时候会**先比较hashCode然后再去比较equals**， 返回结果为null其实都是hashCode惹的祸。



以Java.lang.Object来理解， JVM每次new一个Object， 都会将Object丢到一个哈希表中去， 这样的话，下次做Object的比较或者取这个对象的时候， 它会根据对象的hashcode再从Hash表中取这个对象。这样做的目的是提高取对象的效率。

**1.new Object(),JVM根据这个对象的Hashcode值,放入到对应的Hash表对应的Key上,如果不同的对象确产生了相同的hash值,也就是发生了Hash key相同导致冲突的情况,那么就在这个Hash key的地方产生一个链表,将所有产生相同hashcode的对象放到这个单链表上去,串在一起。**

**2.比较两个对象的时候,首先根据他们的hashcode去hash表中找他的对象,当两个对象的hashcode相同,那么就是说他们这两个对象放在Hash表中的同一个key上,那么他们一定在这个key上的链表上。那么此时就只能根据Object的equal方法来比较这个对象是否equal。当两个对象的hashcode不同的话，肯定他们不能equals.**