字典的底层实现

```
字典：
{key:value,key:value}
1. Python3.6
        的字典底层保留了字典输入时的顺序
        底层还是无序的
        内存占用量减少了(20%~25%)，执行效率提高了
2. Python3.5
        字典底层是无序的
```

- 认可知识
- 放自己一马

Python3.5中的字典

1. 第一次创建字典：
    1. 先创建一个长度为8的，每一个元素都是三个长度的二维数组
    2. 取出键，计算hash值
    3. 将计算出的hash值和数组长度取模
        目的：算出一个下标
    4. 存储数据：
        [哈希值，键的首地址，值的首地址]
2. 访问字典：
    1. 取出键，计算hash值
    2. 逐个比对哈希值，如果存在则返回值，不存在则报错
3. 再次插入键值对：
    同 创建字典的2，3，4步骤

- 优缺点：

1. 如果执行遍历操作
    1. 底层数组，不是所有的元素都有字典的内容，有很多空白内容
    2. 空白内容也会进行比对或遍历的操作
    3. 有部分的资源浪费到了空白的内容上
2. 哈希冲突问题：
    1. 开发寻址法
        Python3.5中使用的线性探测
            如果发生了哈希冲突，下标+1
    2. 链地址法
            使用链表链接所有相同下标的元素
    3. 公共溢出区
    4. 再哈希
            换一个算法接着算，直到算出来计算后的下标有位置为止

Python3.6 字典的实现

保留了创建时的顺序，但是底层是无序的

1. 创建字典：
    1. 创建两个数组：
        index=[None,None,None,None,None,None,None,None]
        这个数组用于存储键值对存储在第二个数组中的下标
        entry=[] 是一个二维数组，

每个元素都是[哈希值，键的首地址，值的首地址]
这个数组用于存储键值对，并记录创建的顺序
注意：entry开始时，没有预先创建数据
2. 取出键，计算hash值
3. 将计算出的hash值和数组长度取模
目的：算出一个下标
4. 存储数据：
index对应的下标的位置将None值修改为entry对应的下标值
entry中添加元素值
2. 访问字典：
1. 取出键，计算hash值
2. 在entry中逐个比对哈希值，如果存在则返回值，不存在则报错
3. 再次插入键值对：
同 创建字典的2，3，4步骤

- 优势：

相比较Python3.5
1. 可以记录创建顺序
2. entry中的数据，按需添加的，遍历时，没有空白收据，减少资源消耗
3. 执行效率高，内存占用更小
