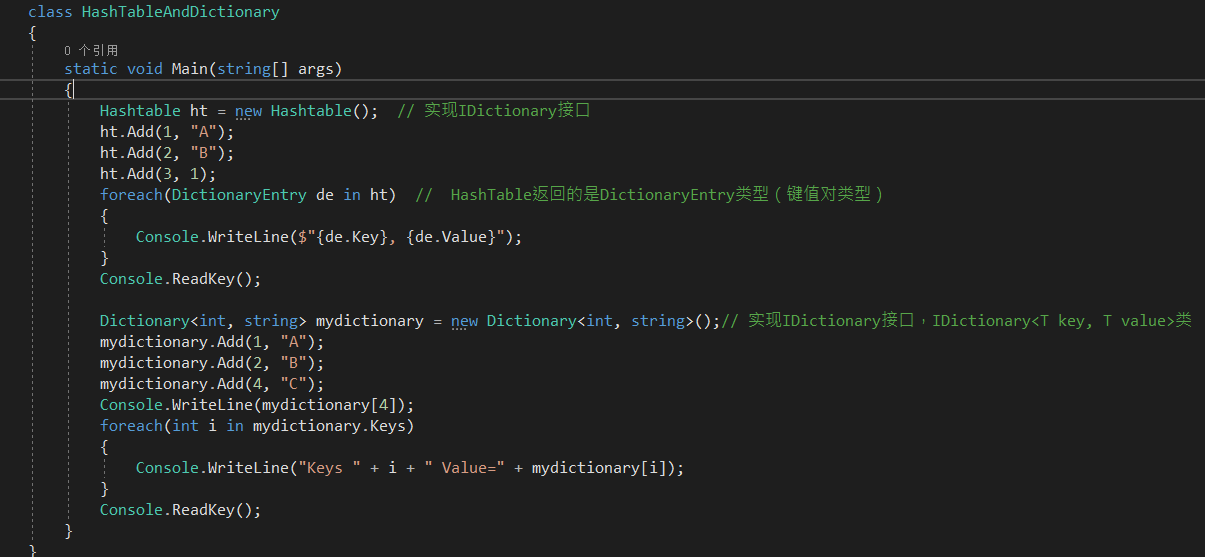
HashTable（非泛型集合）对应Dictionary<K,V>（泛型集合）

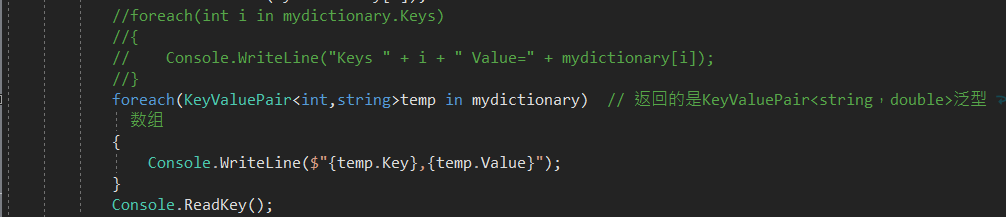
1. 单线程程序中推荐使用Dictionary，有泛型优势，且读取速度较快，容量利用更充分。
2. 多线程程序中推荐使用HashTable，默认的HashTable允许单线程写入，多线程读取，对HashTable进一步调用Synchronized()方法可以获得完全线程安全的类型，而Dictionary非线程安全，必须人为使用lock语句进行保护，效率大减。
3. Dictionary有按插入顺序排列数据的特性（当调用Remove()删除节点后顺序被打乱）,因此在需要体现顺序的情境中使用Dictionary能获得一定方便.

HashTable类和Dictionary泛型类实现的IDictionary接口,Dictionary泛型类还实现IDictionary泛型接口.因此这些集合中的每个元素都是一个键值对.

Dictionary类与HashTable类的功能相同.



对于dictionary集合的键值对输出格式，还可以如下



DictionaryEntry类型是一个可设置和检索的键值对类型（仅使用与System.Collections,Dictionary在System.Collections.Generic）

Value = mydictionary[i]，i为对应key，适用于这两个集合

1. HashTable
2. HashTable是一种散列表，它内部维护很多对Key-Value键值对，还有一些类似索引的值叫做散列值（HashCode），它是根据GetHashCode方法对Key通过一定算法得到的，所有的查找操作定位操作都是基于散列值来实现找到对应的Key和Value值的。
3. 我们需要使用一个算法让散列值（生成的独一无二的值）对应HashTable的空间地址尽量不重复，这就是散列函数（GetHashCode）需要做的事。
4. 当一个HashTable被占用一大半的时候我们通过计算散列值取得的地址值可能会重复指向同一地址，这就是哈希冲突。
5. 在.NET中键值对在HashTable中的位置

**Position = (HashCode&0X7FFFFFFF)%HashTable.Length**，.NET中是通过探测法解决哈希冲突的,当通过散列值取得的位置position被占用的时候,就会增加一个位移X值判断下一个position+x是否被占用,如果仍然被占用就继续往下位移x判断position+2\*x是否被占用，如果没有则将值放入其中。当HashTable中的可用空间越来越小时，则获取得到可用空间的难度越大，消耗的时间越多。

1. 当前HashTable中的被占用空间达到一个百分比的时候就会将该空间自动扩容，在.net中这个百分比是72%，也叫.Net中HashTable的填充因子为0.72.例如有一个HashTable的空间大小是100，当它需要添加第73个值的时候将会扩容此HashTable。
2. 这个自动扩容的大小是当前空间的两倍最接近的素数，例如当前HashTable所占空间为素数71，如果扩容，则大小为素数131.
3. Dictionary
4. Dictionary是一种变种的HashTable，它采用一种分离链接散列表的数据结构来解决哈希冲突的问题。
5. 分离链接散列表是将散列到同一个地址的值存为一个链表中。
6. 这个变种HashTable的填充因子是1.

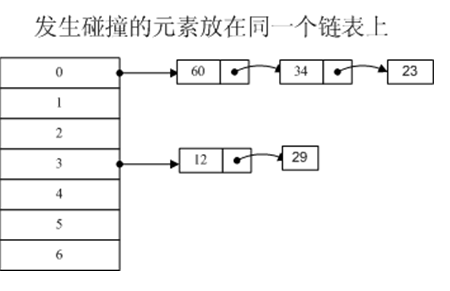
<https://blog.csdn.net/exiaojiu/article/details/51252515> C#Dictionary源码解析

对于不同的关键字可能得到同一哈希地址，即key1!=key2 =>F(key1)=F(key2),这种现象叫做冲突，在一般情况下，冲突只能尽可能减少，而不能完全避免。因为，哈希函数是从关键字集合到地址集合的映像。通常，关键字集合比较大，它的元素包括多可有可能的关键字。那么如何处理冲突就是十分重要的事情了。

在哈希表上进行查找的过程和哈希造表的过程基本一致。给定K值，根据造表时设定好的哈希函数求得哈希地址（position = HashCode%tableLength），若表中此位置没有记录，则查找不成功；否则比较关键字，若给定值相等，则查找成功；否则根据处理冲突的方法寻找“下一地址”，知道哈希表中某个位置为空或者表中所填记录的关键字等于给定值时为止。

Dictionary使用解决冲突的方法是拉链法，又称链地址法。

原理：将所有关键字为同义词的结点链接到同一个单链表中。若选定的散列表长度为m，则可将散列表定义为一个由m个头指针组成的指针数组T[0…m-1]。凡是散列地址为i的结点，均插入到以T[i]为头指针的单链表中。T中各分量的初值均应为空指针。



以下皆为Dictionary源码

基本成员

Dictionary没有把集合处理为一个链表。

而是使用数组entries存放数据，entries数组中存放的是Entry结构



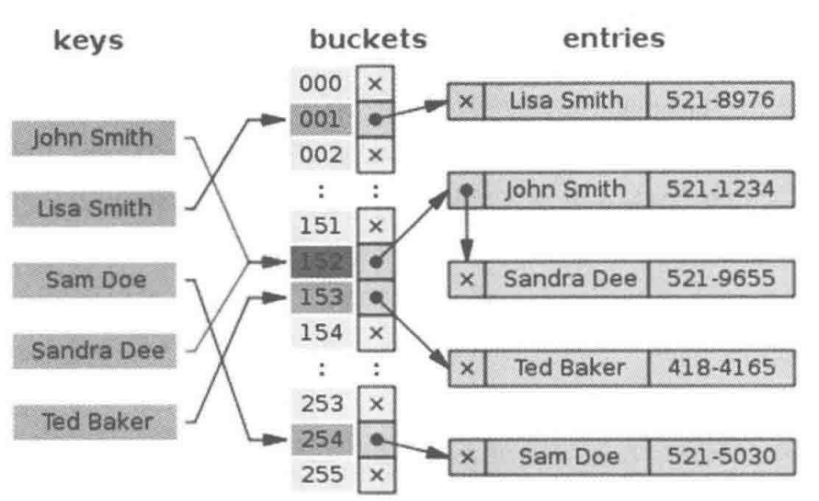
通过next作为索引，在数组entries中构成了“链式结构”。

将哈希函数与数据数组解耦

同时，，dictionary的冲突处理策略通过一个数据通buckets将哈希函数与数据数组进行了解耦。

即根据key得到的哈希值（通过GetHashCode方法）是buckets的索引，而不是数据数组entries的索引。通过buckets的索引得到第一个哈希值相等的数据。

这样解耦带来的好处是，当哈希函数变化时，entries不需要变化（只需要改变buckets的对应关系即可）。



初始化函数：

该函数用于初始化的数据构造



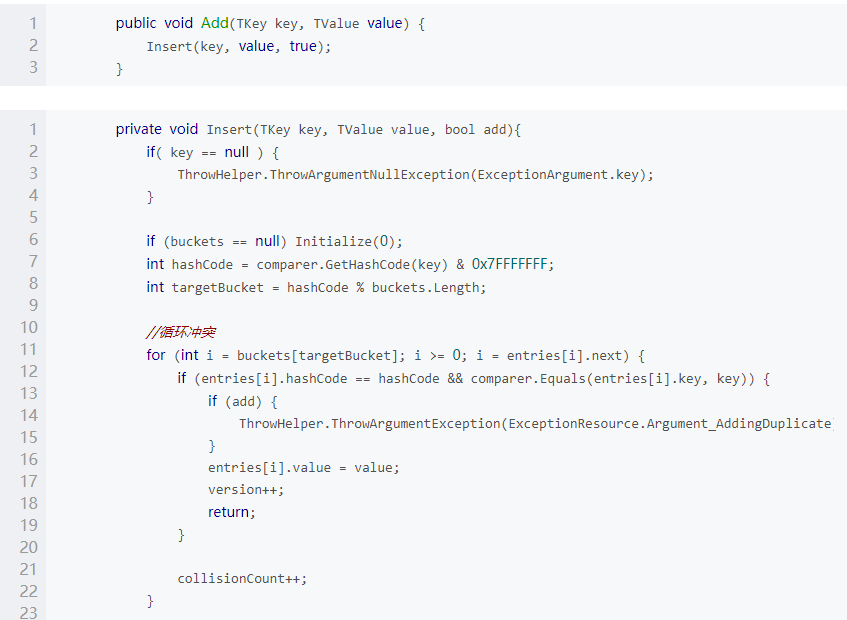
Size 哈希表的长度是素数，可以使元素更均匀地分布在每个节点上。

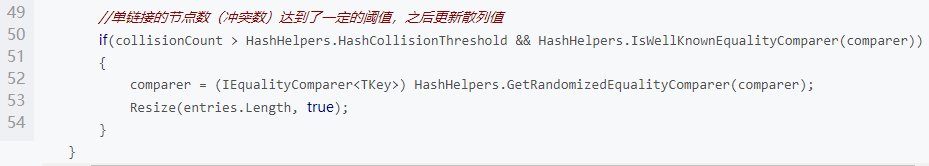
Buckets中的节点值，-1表示空值。

Freelist为-1表示没有空链表。

Buckets和freelist所指向的数据其实全是存储于一块连续的内存空间中（entries）。

插入元素



 思路分析：

1. 通过哈希函数寻址，计算出哈希地址（因为中间有一个解耦关系buckets，所以不再直接指向entries的索引值，而是buckets的索引。）
2. 判断buckets中映射到的值是否为-1（即空位）。若不为- 1，表示有冲突，遍历冲突链，不允许重复的键。
3. 判断是否有空链表，有则插入空链表的当前位置，将freelist指针后移，freecount减一，否则将元素插入当前空位。当容量不足的时候，要进行扩容。

**Dictionary中存储元素的结构非常有趣，通过一个数据通buckets将哈希函数与数据数组进行了解耦，使得每一个buckets的值对应的都是一个单链表，在内存上确实连续的存储块。同时Dictionary在空间与性能之间做了一些取舍，消耗了空间，提升了性能**。

从上面的结果可以看出：

1. HashTable大数据量插入数据时需要花费比Dictionary大得多的时间（因为HashTable为非泛型集合，写入时需要进行大量的拆箱操作）
2. For方式遍历HashTable和Dictionary速度最快
3. 在foreach方式遍历时，Dictionary遍历速度更快。