GetHashCode函数一般是在操作HashTable或者Dictionary之类的数据集的时候被调用，目的是产生一个Key，为了方便在 HashTable或者 Dictionary中的检索。 每个类型，不管是值类型还是引用类型，都提供这个基本函数，同样也可以像重写ToString或者Equals函数一样去重写它。但是不建议重写此函数，而且在使用这个函数也需要加倍小心。

对于GetHashCode来说，要满足如下三点，这也是判断一个GetHashCode函数是否有效的标准。

第一，两个相等的对象，通过GetHashCode函数产生的结果要相 等，此外两个不相等的对象，通过GetHashCode函数的返回值要不相等；否则，通过其产生HashCode而存入HashTable中的数据就无法 取出来了。

第二，对于一个类型的对象来说，其GetHashCode函数的返 回值要自始至终要保持一致。否则，和第一点一样。

第三，在 GetHashCode函数中需要提供一个比较好的哈希函数，也就是在最小的范围内来实现数据分散，换句话说它的离散度决定HashTable存取效率。

对于引用类型自带的GetHashCode函数来说，基本上是正确的，但是效率不高；而对于值类型自带的GetHashCode函数而言，基本上是不正确的，即使正确也是效率不高。

首先说说引用类型自带的GetHashCode函数实现。一个.Net程序在运行的时候会对引用类型的对象进行标记，大致操作类似如下：标记起始为0，当创建一个引用类型对象的时候，标记会自动加一，对象释放后标记并不做减一操作，这有点儿像**[数据库](http://lib.csdn.net/base/mysql" \o "MySQL知识库" \t "http://blog.csdn.net/cnhk1225/article/details/_blank)**中的自增字段。

这也就是为什么说引用类型的GetHashCode函数基本是正确的原因。对于第一条来说，如果类型没有重载Equals或者Operator==函数的话，类型自带的Equals函数只是在对象引用层进行验证，也就是说，一个对象等于另外一个对象就说明这个对象要么是另外一个对象的引用，要么另外一个对象是这个对象的引用。这说明没有新的引用对象产生，那么当引用标记也不会发生变化，所以对于第一条来说满足（如果要是重载了Equals或者Operator==函数的话，那么相应要提供此版本的GetHashCode函数，这一点在后面进行叙说）。至于第二条来说，由于对象数据成员发生改变不会影响到引用标记的改变，所以对于第二条来说也是满足的。对于第三点来说，由于引用标记是相对于整个程序而言的，并不是类型所特有的，那么它的效率不高是不言而喻的。

那么对于值类型自带的GetHashCode函数呢，就更有趣了，为了更形象地说明它的有趣，请先参看如下的代码，猜猜Debug的输出是什么。

**C#代码  [IMG_256](http://ajwang.iteye.com/blog/683170" \o "收藏这段代码" \t "http://blog.csdn.net/cnhk1225/article/details/_blank)**

1. **public** **struct** ErrorMessage
2. {
3. **private** **string** strMsg;
4. **private** **int** nErrorCode;
5. **private** DateTime dtInvoked;
6. **public** **bool** TestHashCode()
7. {
8. **return** **this**.GetHashCode() == strMsg.GetHashCode();
9. }
11. }
12. // Test "GetHashCode" function in value type
13. ErrorMessage err = **new** ErrorMessage( "Test", 0 );
14. **if**( err.TestHashCode() )
15. Debug.WriteLine( "Both hash code equal!" );
16. **else**
17. Debug.WriteLine( "Not equal!" );

可能谁都没有想到，Debug中的输出是“Both hash code equal!”。为什么呢？原因很简单，值类型自带的GetHashCode是以其第一个成员的GetHashCode值作为其的返回值。对于第一条来说，两个相等值类型对象，其的GetHashCode函数返回值是相等的，这没什么问题；但是对于不相等的两个对象来说，它们的GetHashCode返回值则有可能相等。显然违反了第一条。对于第二条来说，由于值类型的GetHashCode返回值等于其第一个成员的GetHashCode函数值，那么修改了第一个成员的值，也就间接的修改了对象的GetHashCode值，从而对于一致性来说也是不满足的。至于第三条，就函数本身来说，效率也和引用类型基本一样，没有采用特殊的**[算法](http://lib.csdn.net/base/datastructure" \o "算法与数据结构知识库" \t "http://blog.csdn.net/cnhk1225/article/details/_blank)**，所以想得到比较好的效率也是不可能的。

下面说说如何实现一个比较正确的GetHashCode函数 如何避免如上的错误 （这里所说的实现主要满足前两条即可，最后一条牵扯到Hash函数算法，这里不做讨论）。

这次先说说值类型，因为值类型本身提供的基本不正确。如果不想做过多处理，毕竟提供一个好的哈希函数不容易，那么从值类型的GetHashCode规律出发，即从类型自身元素出发。对于一个值类型，如果其本身存在某个数据可以唯一标明此类型对象，有点儿像数据库中的Key字段，那么用它作为类型的第一个元素。例如就前面所说的ErrorMessage来说，dtInvoked成员可以唯一表示这个类型数据，那么就可以如下修改。这样就满足了验证第一条，对于第二条，就是要保证这个类型的对象通过GetHashCode能自始至终一样，就要防止第一个成员被修改，比较好的做法就是给它加上readonly标示，那么比较完整的样式应该如下。这样对于ErrorMessage类型的GetHashCode至少是正确的。有人说了，如果定义的类型没有一个单独成员能作为唯一标示，那我就建议你不要把这种类型的数据来产生Key。

**C#代码  [IMG_257](http://ajwang.iteye.com/blog/683170" \o "收藏这段代码" \t "http://blog.csdn.net/cnhk1225/article/details/_blank)**

1. **public** **struct** ErrorMessage
2. {
3. **private** **readonly** DateTime dtInvoked;
5. **private** **string** strMsg;
7. **private** **int** nErrorCode;
8. }

接下来说说对于引用类型的GetHashCode函数改写。对于第一条来说，在引用类型中有可能重新编写Equals函数，那么类型自带的GetHashCode函数将不能适应这个要求，需要进行重写来适应这种改变。如何简便的改写GetHashCode函数而达到效果呢，这里可以延用前面值类型的做法，即选择一个能唯一标示这个对象的成员来生成HashCode，同时要避免这个成员被修改。

例如一个比较合理的引用类型的GetHashCode函数大致如下（此例引用于原书）：

    public class Customer

    {

        private readonly string \_name;

        private decimal \_revenue;

        public Customer( string name ): this( name, 0 )

        {

        }

        public Customer( string name, decimal revenue )

        {

            \_name = name;

            \_revenue = revenue;

        }

        /// <summary>

        /// Name property which only can be accessed in reading mode

        /// </summary>

        public string Name

        {

            get{ return \_name;}

        }

        /// <summary>

        /// Create a new object with new name

        /// </summary>

        /// <param name="newName"></param>

        /// <returns></returns>

        public Customer ChangeName( string newName )

        {

            return new Customer( newName, \_revenue );

        }

        /// <summary>

        /// Customer hash code generated by name

        /// </summary>

        /// <returns></returns>

        public override int GetHashCode()

        {

            return \_name.GetHashCode ();

        }

    }

对于Customer类型对象来说，它的HashCode是由其\_name成员所决定的，所以不能轻易改变，如果通过调用ChangeName方法来替换原先的对象的时候，要首先操作HashTable，先把原先的删除，创建新的之后再保存。具体如下：

    Customer c1 = new Customer( "test1" );

    object orders = new object();

    myHashTable.Add( c1, orders );

    //Change name

    Customer c2 = c1.ChangeName( "test2" );

    object o = myHashTable[ c1 ];

    myHashTable.Remove( c1 );

    myHashTable.Add( c2, o );

对于如上中Custemer对象来说，只是为了产生在HashTable中所存对象的HashCode，当然在实际应用中，两者需要关联，否则使用HashTable存这些数据就没有任何意义了。

这样对于值类型和引用类型的GetHashCode改写到此基本已经结束了，显然如上的改写，只是为了保证类型的GetHashCode正确，但是对于其的效率并没有得到长足的进步，或者换句话来说，改写后的GetHashCode函数仍然保留HashTable使用效率不高。如何在GetHashCode函数使用比较好的哈希函数，使产生的HashCode具有比较好的分布，我在此不对它进行讨论，因为光这个问题就足够写好几本书的。

对于GetHashCode函数，大致就说到这儿，最后为了加深记忆，总结一下。

首先，在不重写此函数的情况下，这里主要说说使用当中应该注意的。

1． 不建议使用值类型对象的GetHashCode函数返回值来作为HashTable对象的Key；

2． 引用类型是可以使用的，但是要注意如果重写了Equals函数，一定要重写GetHashCode函数来达到一致；

再说说重写此函数时需要注意的。

1． 不管是值类型还是引用类型，要保证产生HashCode的成员不能被修改；

2． 对于产生HashCode的成员修改，要以产生新对象进行处理，同时要在使用端作相应的修改，即先删除旧的在添加新的。

使用类型的GetHashCode有微妙的危险，所以使用的时候要特别注意。