们将在后面看到，而为了避免上面那样的情况，好的办法通常是保证表的大小是素数。当输入的关键字是随机整数时，散列函数不仅计算起来简单而且关键字的分配也很均匀。

通常，关键字是字符串；在这种情形下，散列函数需要仔细地选择。

一种选择方法是把字符串中字符的 ASCII 码(或 Unicode 码)值加起来。图 5-2 中的例程实现这种策略。

图 5-2 中描述的散列函数实现起来简单而且能够很快地计算出答案。不过，如果表很大，函数将不会很好地分配关键字。例如，设 TableSize＝10 007(10 007 是素数)，并设所有的关键字至多 8 个字符长。由于 ASCII 字符的值最多是 127，因此散列函数只能假设值在 0 和 1 016 之间，其中 1 016 为 127 * 8。显然这不是一种均匀的分配。

```
1         public static int hash( String key, int tableSize )
2         {
3             int hashVal = 0;
4
5             for( int i = 0; i < key.length( ); i++ )
6                 hashVal += key.charAt( i );
7
8             return hashVal % tableSize;
9         }
```

图 5-2　一个简单的散列函数

另一个散列函数如图 5-3 所示。这个散列函数假设 Key 至少有 3 个字符。值 27 表示英文字母表的字母外加一个空格的个数，而 729 是 27^2。该函数只考查前三个字符，但是，假如它们是随机的，而表的大小像前面那样还是 10 007，那么我们就会得到一个合理的均衡分布。可是不巧的是，英文不是随机的。虽然 3 个字符(忽略空格)有 $26^3 = 17\,576$ 种可能的组合，但查验合理的足够大的联机词典却揭示：3 个字母的不同组合数实际只有 2 851。即使这些组合没有冲突，也不过只有表的 28% 被真正散列到。因此，虽然很容易计算，但是当散列表具有合理大小的时候这个函数还是不合适的。

```
1         public static int hash( String key, int tableSize )
2         {
3             return ( key.charAt( 0 ) + 27 * key.charAt( 1 ) +
4                      729 * key.charAt( 2 ) ) % tableSize;
5         }
```

图 5-3　另一个可能的散列函数——不是太好

图 5-4 列出了散列函数的第 3 种尝试。这个散列函数涉及关键字中的所有字符，并且一般可以分布得很好(它计算 $\sum_{i=0}^{KeySize-1} Key[KeySize - i - 1] \cdot 37^i$，并将结果限制在适当的范围内)。程序根据 Horner 法则计算一个(37 的)多项式函数。例如，计算 $h_k = k_0 + 37k_1 + 37^2k_2$ 的另一种方式是借助于公式 $h_k = ((k_2) * 37 + k_1) * 37 + k_0$ 进行。Horner 法则将其扩展到用于 n 次多项式。

这个散列函数利用到事实：允许溢出。这可能会引进负的数，因此在末尾有附加的测试。

图 5-4 所描述的散列函数就表的分布而言未必是最好的，但确实具有极其简单的优点而且速度也很快。如果关键字特别长，那么该散列函数计算起来将会花费过多的时间。在这种情况下通常的经验是不使用所有的字符。此时关键字的长度和性质将影响选择。例如，关键字可能是