HashMap的性能常常优于TreeMap的性能，不过不按这两种方式编写代码很难有把握肯定。因此，在HashMap或TreeMap可以接受的情况下，更可取的方法是：使用接口类型Map进行变量的声明，然后，将TreeMap的实例变成HashMap的实例并进行记时测试。

在Java中，能够被合理地插入到一个HashSet中去或是所谓关键字被插入到HashMap中去的那些库类型已经被定义了equals和hashCode方法。特别是String类中有一个hashCode方法，它基本上就是图5-4中除掉第14行到第16行并将第37行用第31行代替后的程序。因为散列表操作中费时多的部分就是计算hashCode方法，所以在String类中的hashCode方法包含一个重要的优化：每个String对象内部都存储它的hashCode值。该值初始为0，但若hashCode被调用，那么这个值就被记住。因此，如果hashCode对同一个String对象被第2次计算，我们则可以避免昂贵的重新计算。这个技巧叫做**闪存散列代码**(caching the hash code)，并且表示另一种经典的时空交换。图5-23显示闪存散列代码的String类的一种实现。

```
1     public final class String
2     {
3         public int hashCode( )
4         {
5             if( hash != 0 )
6                 return hash;
7
8             for( int i = 0; i < length( ); i++ )
9                 hash = hash * 31 + (int) charAt( i );
10            return hash;
11        }
12
13        private int hash = 0;
14    }
```

图5-23 String类hashCode摘录

闪存散列代码之所以有效，只是因为Sting类是不可改变的：要是String允许变化，那么它就会使hashCode无效，而hashCode就只能重置回0。虽然两个具有相同状态的String对象的hashCode必须独立计算，但是，存在许多情况使同一个String对象的散列代码总是被查询。闪存散列代码有用的一种情况是在再散列期间发生，因为在再散列中所涉及到的所有String对象的散列代码都已经闪存过。另一方面，闪存散列代码对于单词变换例子中的代表映射(representative map)是无用的。每个代表都是通过从一个更大的String中删除一个字母所计算出的一个不同的String，因此每一个String只能让它的散列代码单独计算。然而，在第3个映射中，闪存散列代码没有什么用处，因为那些关键字都只是些String，它们被存放在String的原始数组中。

5.7 可扩散列

本章最后的论题处理数据量太大以至于装不进主存的情况。正如我们在第4章看到的，此时主要的考虑是检索数据所需的磁盘存取次数。

与前面一样，我们假设在任一时刻都有 N 个记录要存储；N 的值随时间而变化。此外，最多可把 M 个记录放入一个磁盘区块。本节将设 $M=4$。

如果使用探测散列或分离链接散列，那么主要的问题在于，在一次查找操作期间冲突可能引起多个区块被检察，甚至对于理想分布的散列表也在所难免。不仅如此，当散列表变得过满的时