**[lucene字典实现原理——FST](https://www.cnblogs.com/bonelee/p/6226185.html)**

## 1 lucene字典

      使用lucene进行查询不可避免都会使用到其提供的字典功能，即根据给定的term找到该term所对应的倒排文档id列表等信息。实际上lucene索引文件后缀名为tim和tip的文件实现的就是lucene的字典功能。

      怎么实现一个字典呢？我们马上想到排序数组，即term字典是一个已经按字母顺序排序好的数组，数组每一项存放着term和对应的倒排文档id列表。每次载入索引的时候只要将term数组载入内存，通过二分查找即可。这种方法查询时间复杂度为Log(N)，N指的是term数目，占用的空间大小是O(N\*str(term))。排序数组的缺点是消耗内存，即需要完整存储每一个term，当term数目多达上千万时，占用的内存将不可接受。



## 2 常用字典数据结构

很多数据结构均能完成字典功能，总结如下。

|  |  |
| --- | --- |
| 数据结构 | 优缺点 |
| 排序列表Array/List | 使用二分法查找，不平衡 |
| HashMap/TreeMap | 性能高，内存消耗大，几乎是原始数据的三倍 |
| Skip List | 跳跃表，可快速查找词语，在lucene、redis、Hbase等均有实现。相对于TreeMap等结构，特别适合高并发场景（[Skip List介绍](http://kenby.iteye.com/blog/1187303" \t "_blank)） |
| Trie | 适合英文词典，如果系统中存在大量字符串且这些字符串基本没有公共前缀，则相应的trie树将非常消耗内存（[数据结构之trie树](http://dongxicheng.org/structure/trietree/" \t "_blank)） |
| Double Array Trie | 适合做中文词典，内存占用小，很多分词工具均采用此种算法（[深入双数组Trie](http://blog.csdn.net/zhoubl668/article/details/6957830)） |
| Ternary Search Tree | 三叉树，每一个node有3个节点，兼具省空间和查询快的优点（[Ternary Search Tree](http://www.drdobbs.com/database/ternary-search-trees/184410528" \t "_blank)） |
| Finite State Transducers (FST) | 一种有限状态转移机，Lucene 4有开源实现，并大量使用 |

## 3 FST原理简析

     lucene从4开始大量使用的数据结构是FST（Finite State Transducer）。FST有两个优点：1）空间占用小。通过对词典中单词前缀和后缀的重复利用，压缩了存储空间；2）查询速度快。O(len(str))的查询时间复杂度。

     下面简单描述下FST的构造过程（工具演示：[http://examples.mikemccandless.com/fst.py?terms=&cmd=Build+it%21](http://examples.mikemccandless.com/fst.py?terms=&cmd=Build+it%21" \t "_blank)）。我们对“cat”、 “deep”、 “do”、 “dog” 、“dogs”这5个单词进行插入构建FST（注：必须已排序）。

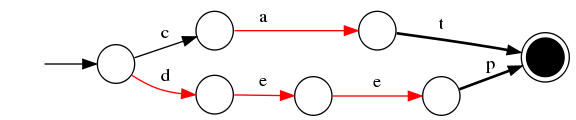
1）插入“cat”

     插入cat，每个字母形成一条边，其中t边指向终点。



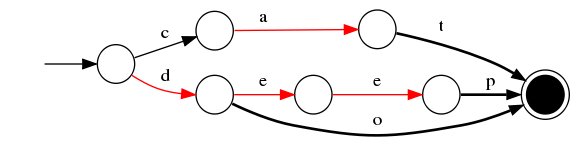
2）插入“deep”

    与前一个单词“cat”进行最大前缀匹配，发现没有匹配则直接插入，P边指向终点。



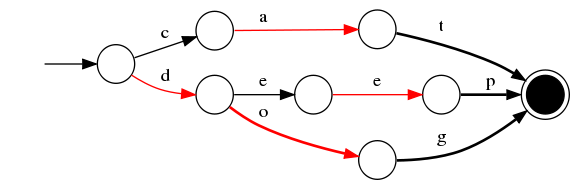
3）插入“do”

    与前一个单词“deep”进行最大前缀匹配，发现是d，则在d边后增加新边o，o边指向终点。



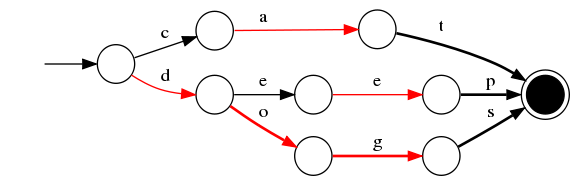
4）插入“dog”

    与前一个单词“do”进行最大前缀匹配，发现是do，则在o边后增加新边g，g边指向终点。



5）插入“dogs”

     与前一个单词“dog”进行最大前缀匹配，发现是dog，则在g后增加新边s，s边指向终点。



     最终我们得到了如上一个有向无环图。利用该结构可以很方便的进行查询，如给定一个term “dog”，我们可以通过上述结构很方便的查询存不存在，甚至我们在构建过程中可以将单词与某一数字、单词进行关联，从而实现key-value的映射。

## 4 FST使用与性能评测

      我们可以将FST当做Key-Value数据结构来进行使用，特别在对内存开销要求少的应用场景。Lucene已经为我们提供了开源的FST工具，下面的代码是使用说明。

[复制代码](javascript:void(0);)

复制代码

1 public static void main(String[] args) {

2 try {

3 String inputValues[] = {"cat", "deep", "do", "dog", "dogs"};

4 long outputValues[] = {5, 7, 17, 18, 21};

5 PositiveIntOutputs outputs = PositiveIntOutputs.getSingleton(true);

6 Builder<Long> builder = new Builder<Long>(FST.INPUT\_TYPE.BYTE1, outputs);

7 BytesRef scratchBytes = new BytesRef();

8 IntsRef scratchInts = new IntsRef();

9 for (int i = 0; i < inputValues.length; i++) {

10 scratchBytes.copyChars(inputValues[i]);

11 builder.add(Util.toIntsRef(scratchBytes, scratchInts), outputValues[i]);

12 }

13 FST<Long> fst = builder.finish();

14 Long value = Util.get(fst, new BytesRef("dog"));

15 System.out.println(value); // 18

16 } catch (Exception e) {

17 ;

18 }

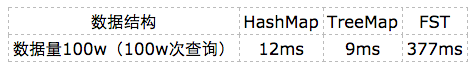
19 }

复制代码

[复制代码](javascript:void(0);)

      FST压缩率一般在3倍~20倍之间，相对于TreeMap/HashMap的膨胀3倍，内存节省就有9倍到60倍！（摘自：[把自动机用作 Key-Value 存储](http://blog.csdn.net/whinah/article/details/9980893)），那FST在性能方面真的能满足要求吗？

      下面是我在苹果笔记本（i7处理器）进行的简单测试，性能虽不如TreeMap和HashMap，但也算良好，能够满足大部分应用的需求。



## 参考文献

<http://sbp810050504.blog.51cto.com/2799422/1361551>

<http://blog.sina.com.cn/s/blog_4bec92980101hvdd.html>

<http://blog.mikemccandless.com/2013/06/build-your-own-finite-state-transducer.html>

<http://examples.mikemccandless.com/fst.py?terms=mop%2F0%0D%0Amoth%2F1%0D%0Apop%2F2%0D%0Astar%2F3%0D%0Astop%2F4%0D%0Atop%2F5%0D%0Atqqq%2F6&cmd=Build+it%21>