



关于Lucene的词典FST深入剖析

📅 Posted on 2018-12-04 | 📅 Edited on 2019-12-17 | 📁 In lucene | 👁 Views: 23434

搜索引擎为什么能查询速度那么快？

核心是在于如何快速的依据**查询词**快速的查找到所有的相关文档，这也是**倒排索引（Inverted Index）**的核心思想。那么如何设计一个快速的(常量，或者1)定位词典的数据结构就显得尤其重要。简单来说，我们可以采用HashMap， TRIE， Binary Search Tree， Tenary Search Tree等各种数据结构来实现。

那么开源的搜索引擎包Lucene是怎么来设计的呢？ Lucene采用了一种称为FST（Finite State Transducer）的结构来构建词典，这个结构保证了时间和空间复杂度的均衡，是Lucene的核心功能之一。

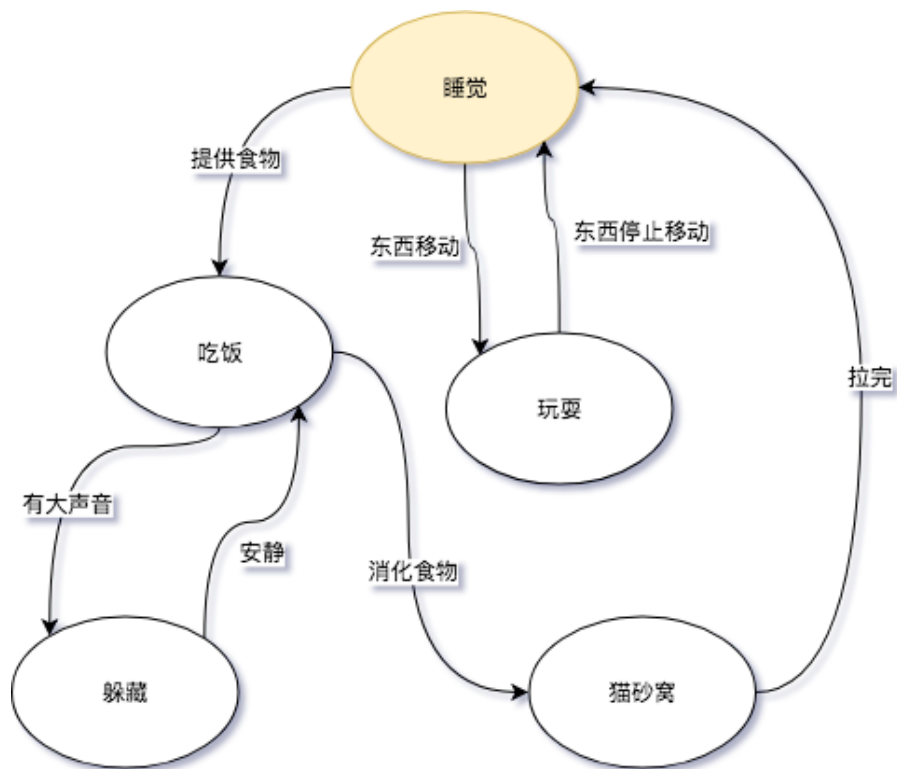
关于FST（Finite State Transducer）

FST类似一种TRIE树。

使用FSM(Finite State Machines)作为数据结构

FSM(Finite State Machines)有限状态机: 表示有限个状态（State）集合以及这些状态之间**转移**和动作的数学模型。其中一个状态被标记为**开始状态**，0个或更多的状态被标记为**final状态**。

一个FSM同一时间只处于1个状态。FSM很通用，可以用来表示多种处理过程，下面的FSM描述了《小猫咪的一天》。



其中“睡觉”或者“吃饭”代表的是**状态**,而“提供食物”或者“东西移动”则代表了**转移**。图中这个FSM是对小猫活动的一个抽象（这里并没有刻意写开始状态或者final状态），小猫咪不能同时的即处于“玩耍”又处于“睡觉”状态，并且从一个状态到下一个状态的转换只有一个输入。“睡觉”状态并不知道是从什么状态转换过来的，可能是“玩耍”，也可能是”猫砂窝”。

如果《小猫咪的一天》这个FSM接收以下的输入：

- 提供食物
- 有大声音
- 安静
- 消化食物

那么我们会明确的知道，小猫咪会这样依次变化状态： 睡觉->吃饭->躲藏->吃饭->猫砂窝.

以上只是一个现实中的例子，下面我们来看如何实现一个Ordered Sets,和Map结构。

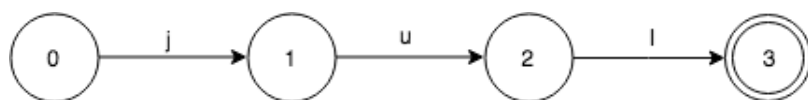
Ordered Sets

Ordered Sets是一个有序集合。通常一个有序集合可以用二叉树、B树实现。无序的集合使用hash table来实现. 这里，我们用一个**确定无环有限状态接收机**（Deterministic acyclic finite state acceptor, FSA）来实现。

FSA是一个FSM(有限状态机)的一种，特性如下：

- 确定：意味着指定任何一个状态，只可能最多有一个转移可以访问到。
- 无环：不可能重复遍历同一个状态
- 接收机：有限状态机只“接受”特定的输入序列，并终止于final状态。

下面来看，我们如何来表示只有一个key：“jul”的集合。FSA是这样的：



当查询这个FSA是否包含“jul”的时候，按字符依序输入。

- 输入j, FSA从0->1
- 输入u, FSA从1->2
- 输入l, FSA从2->3

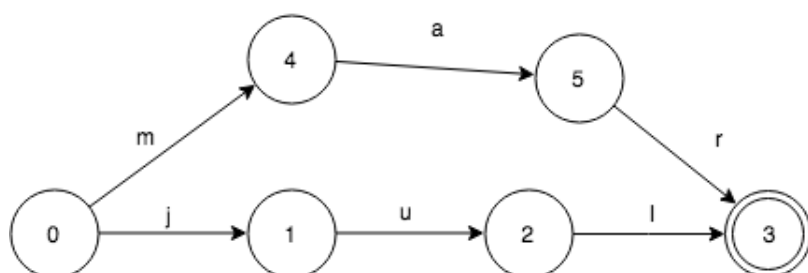
这个时候，FSA处于final状态3，所以“jul”是在这个集合的。

设想一下如果输入“jun”，在状态2的时候**无法移动**了，就知道不在这个集合里了。

设想如何输入“ju”，在状态2的时候，已经没有输入了。而状态2并不是**final状态**，所以也不在这个集合里。

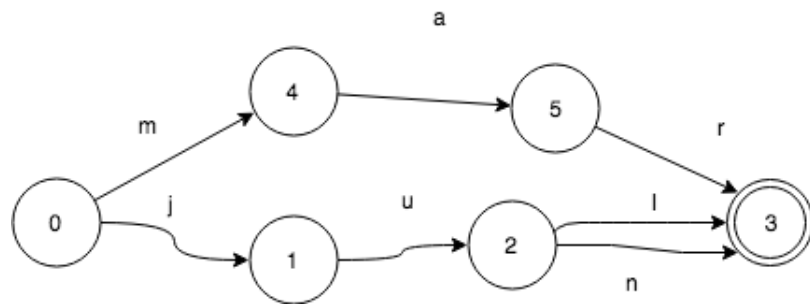
值得指出的是，查找这个key是否在集合内的时间复杂度，取决于key的长度，而不是集合的大小。

现在往FSA里再加一个key. FSA此时包含keys:”jul”和“mar”。



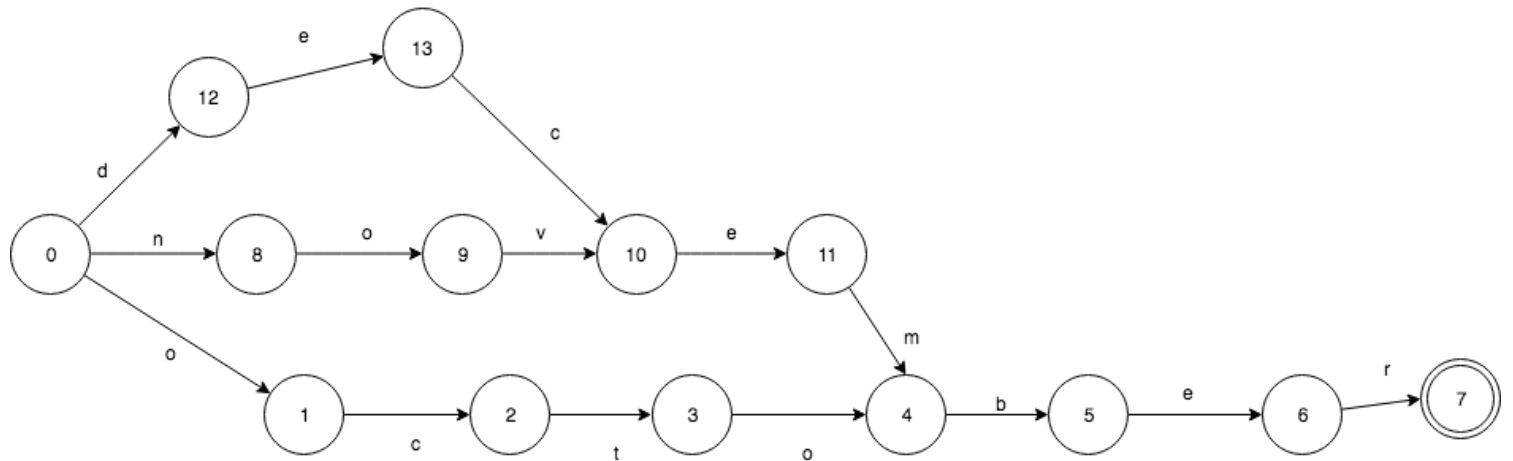
start状态0此时有了2个转移：j和m。因此，输入key:”mar”,首先会跟随m来转移。 final状态是“jul”和“mar”共享的。这使得我们能**用更少的空间来表示更多的信息**。

当我们在这个FSA里加入“jun”，那么它和“jul”有共同的前缀“ju”：



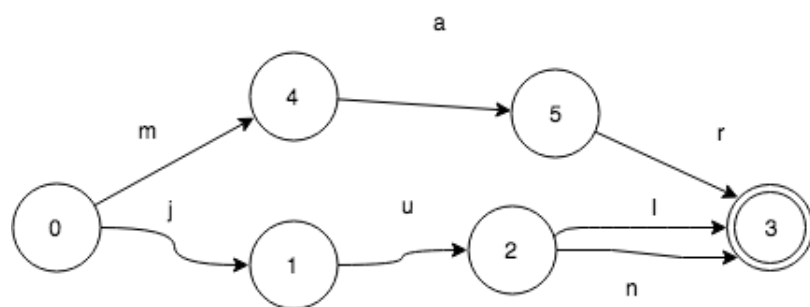
这里变化很小，没有增加新的状态，只是多了一个转移而已。

下面来看一下由“october”，“november”，”december”构成的FSA.



它们有共同的后缀“ber”，所以在FSA只出现了1次。 其中2个有共同的后缀”ember”，也只出现了1次。

那么我们如何来遍历一个FSA表示的所有key呢，我们以前面的”jul”，“jun”，”mar”为例：



遍历算法是这样的：

- 初始状态0， key=""
- ->1, key="j"
- ->2, key="ju"
- ->3, key="jul", 找到jul

- 2<-, key="ju"
- ->3, key="jun", 找到jun
- 2<-, key="ju"
- 1<-, key="j"
- 0<-, key=""
- ->4, key="m"
- ->5, key="ma",
- ->3, key="mar",找到mar

这个算法时间复杂度 $O(n)$, n 是集合里所有的key的大小, 空间复杂度 $O(k)$, k 是结合内最长的key字段length。

Ordered maps

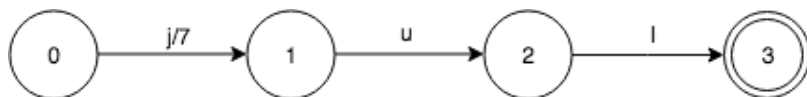
Ordered maps就像一个普通的map, 只不过它的key是有序的。我们来看一下如何使用**确定无环状态转换器 (Deterministic acyclic finite state transducer, FST)** 来实现它。

FST是也一个有限状态机 (FSM) ,具有这样的特性:

- 确定: 意味着指定任何一个状态, 只可能最多有一个转移可以遍历到。
- 无环: 不可能重复遍历同一个状态
- transducer: 接收特定的序列, 终止于final状态, 同时会**输出一个值**。

FST和FSA很像, 给定一个key除了能回答是否存在, 还能输出一个**关联的值**。

下面来看这样的输入: "jul:7", 7是jul关联的值, 就像是一个map的entry.



这和对应的有序集合基本一样, 除了第一个0->1的转换j关联了一个值7. 其他的转换u和l,默认关联的值是0,这里不予展现。

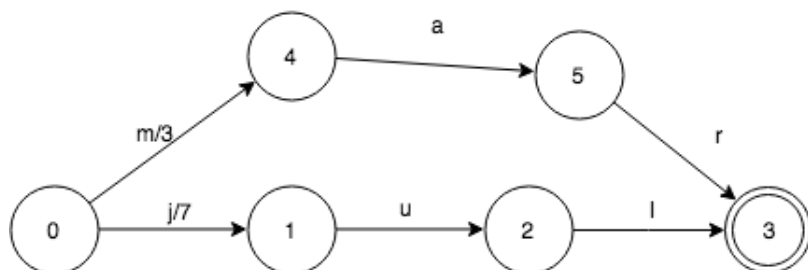
那么当我们查找key:"jul"的时候, 大概流程如下:

- 初始状态0

- 输入j, FST从0->1, value=7
- 输入u, FST从1->2, value=7+0
- 输入l, FST从2->3, value=7+0+0

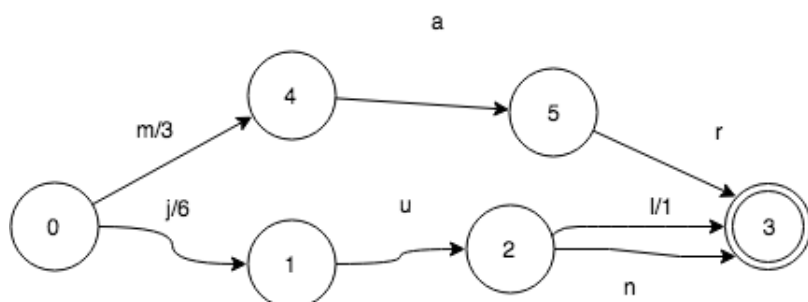
此时, FST处于final状态3, 所以存在jul, 并且给出output是7.

我们再看一下, 加入mar:3之后, FST变成什么样:



同样的很简单, **需要注意的是**mar自带的值3放在了第1个转移上。这只是为了算法更简单而已, 事实上, 可以放在其他转移上。

如果共享前缀, FST会发生什么呢? 这里我们继续加入jun:6。



和sets一样, jun和jul共享状态3, 但是有一些变化。

- 0->1转移, 输出从7变成了6
- 2->3转移, 输入l, 输出值变成了1。

这个输出变化是很重要的, 因为他改变了查找jul输出值的过程。

- 初始状态0
- 输入j, FST从0->1, value=6
- 输入u, FST从1->2, value=6+0
- 输入l, FST从2->3, value=6+0+1

最终的值仍旧是7，但是走的路径却是不一样的。

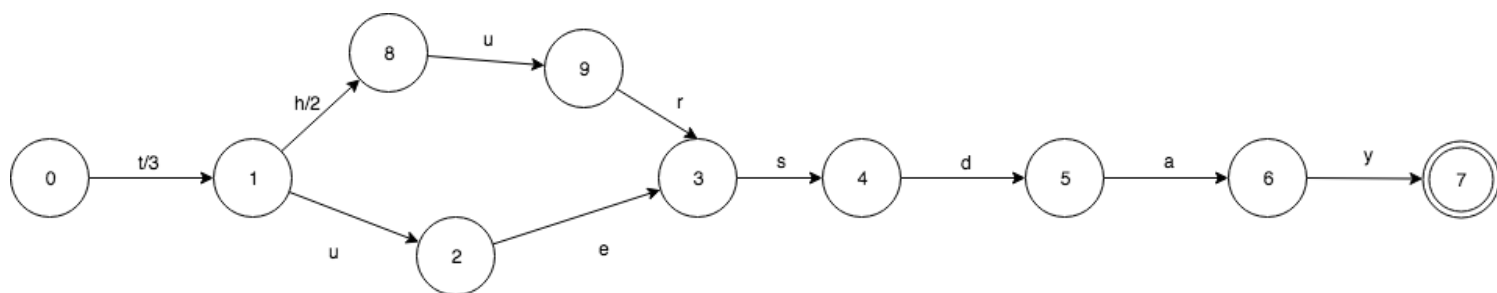
那查找jun是不是也是正确的呢？

- 初始状态0
- 输入j, FST从0 \rightarrow 1, value=6
- 输入u, FST从1 \rightarrow 2, value=6+0
- 输入n, FST从2 \rightarrow 3, value=6+0+0

从上可知，jun的查询也是正确的。FST保证了不同的转移有**唯一**的值，但同时也复用了大部分的数据结构。

实现共享状态的**关键点**是：每一个key,都在FST中对应一个唯一的路径。因此，对于任何一个特定的key，总会有一些value的转移组合使得路径是唯一的。我们需要做的就是如何来在转移中**分配**这些组合。

key输出的共享机制同样适用于共同前缀和共同后缀。比如我们有tuesday:3和thursday:5这样的FST：



2个key有共同的前缀**t**，共同后缀**sday**。关联的2个value同样有共同的前缀。3可以写做**3+0**，而5可以写作：**3+2**。这样很好的让实现了关联value的共享。

上面的这个例子，其实有点简单化，并且局限。假如这些关联的value并不是int呢？实际上，FST对于关联value(outputs)的类型是要求必须有以下操作（method）的。

- 加（Addition）
- 减（Subtraction）
- 取前缀（对于整数来说，就是min）

FST的构建

前面，一直没有提到如何构建FST。构建相对于遍历来说，还是有些复杂的。

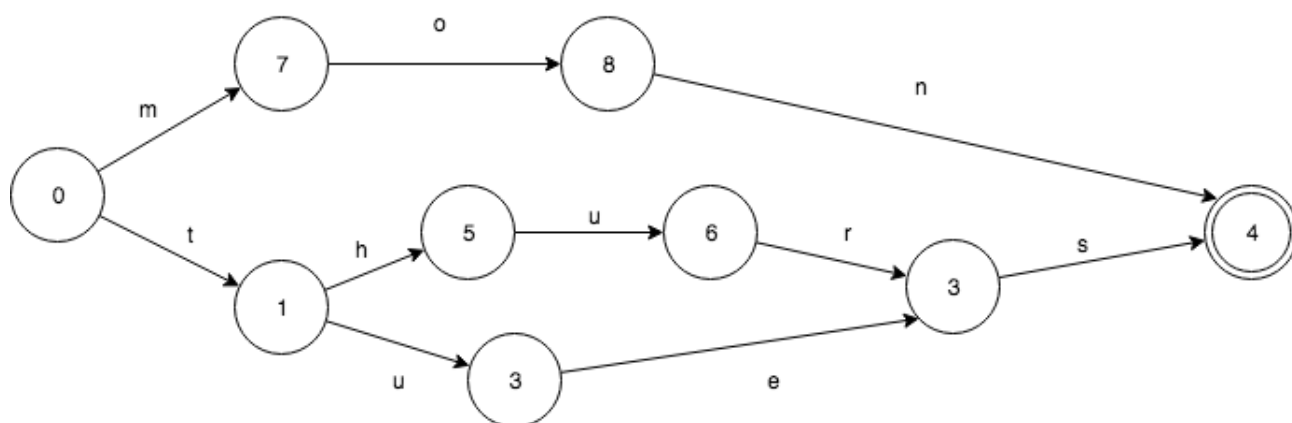
为了简单化，我们假设set或者map里的数据是按字典序加入的。这个假设是很沉重的限制，不过我们会讲如何来缓解它。

为了构建FSM，我们先来看看TRIE树是如何构建的。

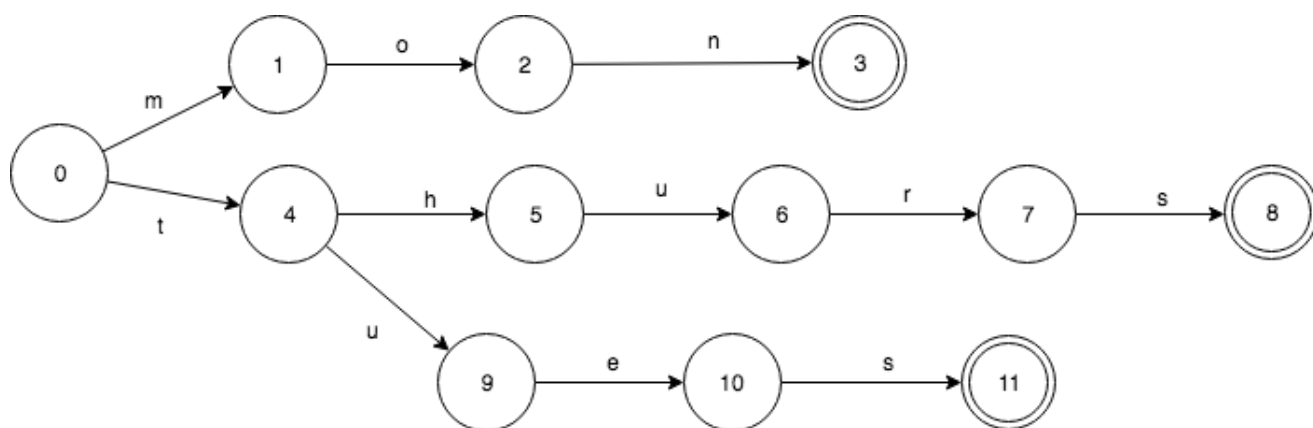
TRIE树的构建

TRIE可以看做是一个FSA,唯一的一个不同是TRIE只共享前缀，而FSA不仅共享前缀还共享后缀。

假设我们有一个这样的Set: mon,tues,thurs。FSA是这样的：



相应的TRIE则是这样的，只共享了前缀。



TRIE有重复的3个final状态3，8，11. 而8，11都是s转移，是可以合并的。

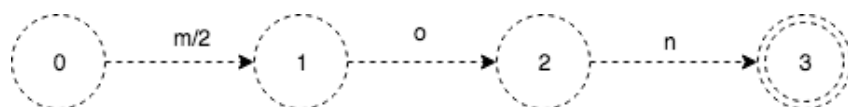
构建一个TRIE树是相当简单的。插入1个key，只需要做简单的查找就可以了。如果输入先结束，那么当前状态设置为final；如果无法转移了，那么就直接创建新的转移和状态。不要忘了最后一个创建的状态设置为final就可以了。

FST的构建

构建FST在很大程度上和构建FSA是一样的，主要的不同点是，怎么样在转移上**放置和共享outputs**。

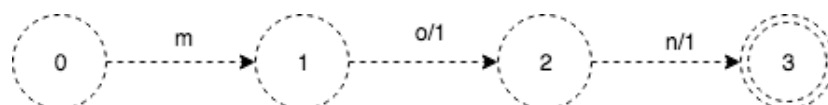
仍旧使用前面提到的例子，mon,tues和thurs，并给他们关联相应的星期数值2，3和5.

从第1个key, mon:2开始：



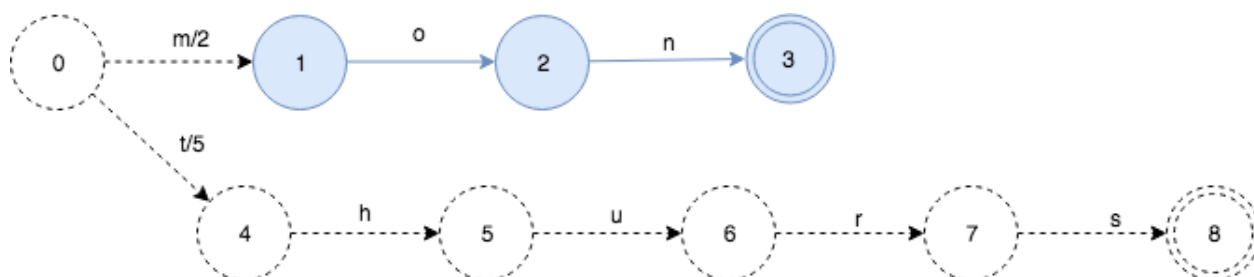
这里虚线代表，在后续的insert过程中，FST可能有变化。

需要关注的是，这里只是把2放在了第1个转移上。技术上说，下面这样分配也是正确的。



只不过，把output放在靠近start状态的算法更容易写而已。

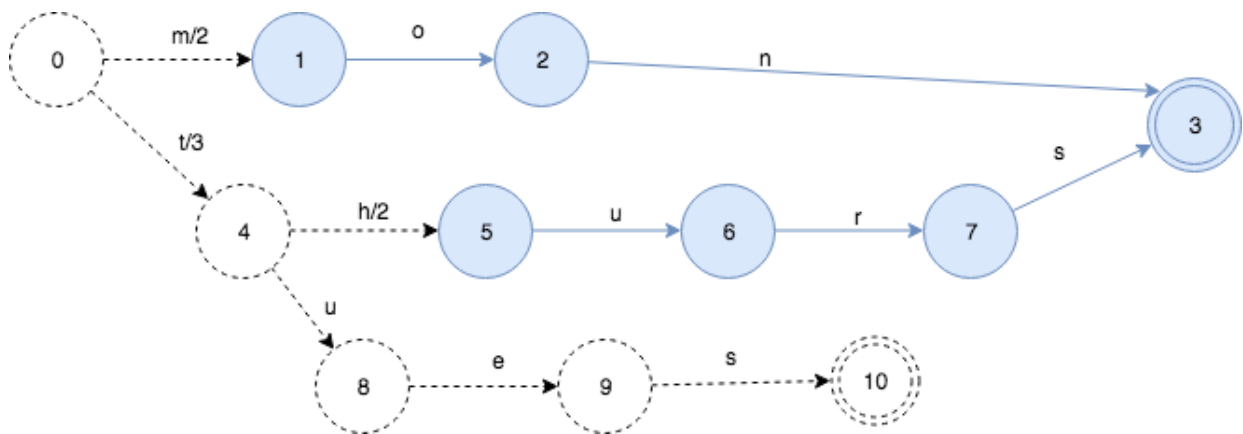
下面继续把thurs:5插入：



就像FSA的insert一样，插入thurs之后，我们可以知道FST的mon部分（蓝色）就不会再变了。

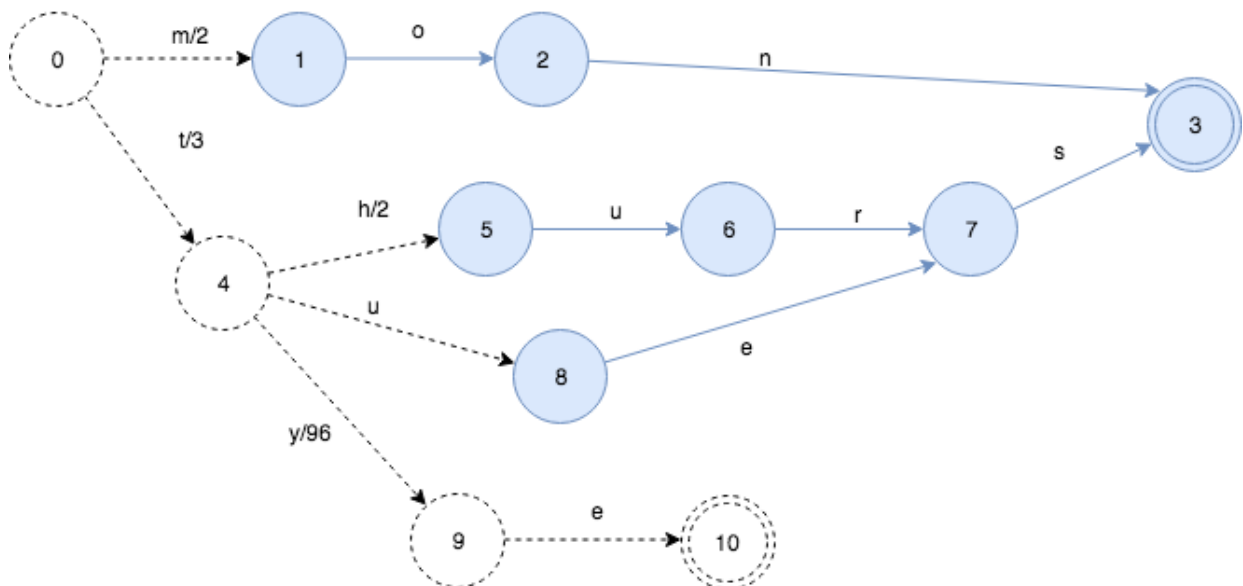
由于mon和thurs没有共同的前缀，只是简单的2个map中的key. 所以他们的output value可以直接放置在start状态的第1个转移上。

下面，继续插入tues:3,



这引起了新的变化。有一部分被冻住了，并且知道以后不会再修改了。output value也出现了重新的分配。因为tues的output是3，并且tues和thurs有共同的前缀t，所以5和3的prefix操作得出的结果就是3. 状态0->状态4的value被分配为3，状态4->状态5设置为2。

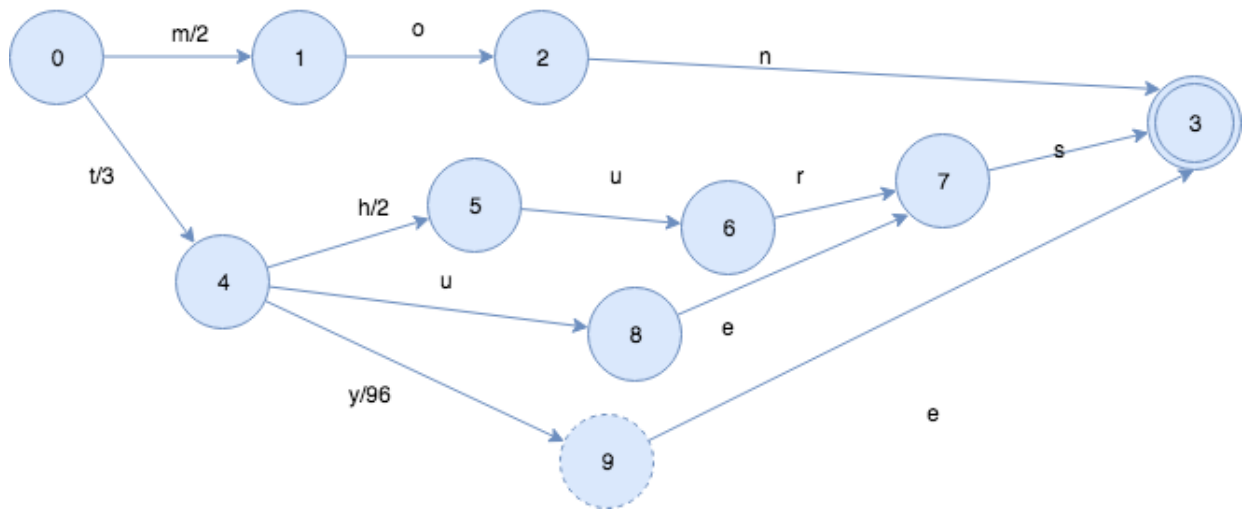
我们再插入更多的key，这次插入tye:99看发生什么情况：



插入tye，导致”es”部分被冻住，同时由于共享前缀t，状态4->状态9的输出是99-3=96。

最后一步，结束了，再执行一次冻住操作。

最终的FST长这样：
























































Lucene FST

上一部分，对于FST的概念以及构建进行了详细的介绍。本部分将对Lucene FST的实现以及具体进行详细的分析。

Lucene关于FST相关的代码在package: `org.apache.lucene.util.fst`。

从 `org.apache.lucene.util.fst.Builder` 看起，这个是构建FST的Builder：

	Builder	
	 dedupHash	NodeHash<T>
	 fst	FST<T>
	 NO_OUTPUT	T
	 minSuffixCount1	int
	 minSuffixCount2	int
	 doShareNonSingletonNodes	boolean
	 shareMaxTailLength	int
	 lastInput	IntsRefBuilder
	 frontier	UnCompiledNode<T>[]
	 lastFrozenNode	long
	 reusedBytesPerArc	int[]
	 arcCount	long
	 nodeCount	long
	 allowArrayArcs	boolean
	 bytes	BytesStore
	 getTermCount()	long
	 getNodeCount()	long
	 getArcCount()	long
	 getMappedStateCount()	long
	 compileNode(UnCompiledNode<T>, int)	CompiledNode
	 freezeTail(int)	void
	 add(IntsRef, T)	void
	 validOutput(T)	boolean
	 finish()	FST<T>
	 compileAllTargets(UnCompiledNode<T>, int)	void
	 fstRamBytesUsed()	long

Builder通过泛型T，从而可以构建包含不同类型的FST。我们重点关注属性。

从其中插入数据 add() 方法看起：

```

1  /** Add the next input/output pair. The provided input
2    * must be sorted after the previous one according to
3    * {@link IntsRef#compareTo}. It's also OK to add the same
4    * input twice in a row with different outputs, as long

```

```

5      * as {@link Outputs} implements the {@link Outputs#merge}
6      * method. Note that input is fully consumed after this
7      * method is returned (so caller is free to reuse), but
8      * output is not. So if your outputs are changeable (eg
9      * {@link ByteSequenceOutputs} or {@link
10     * IntSequenceOutputs}) then you cannot reuse across
11     * calls. */
12     public void add(IntsRef input, T output) throws IOException {
13
14         ...
15         // prefixLenPlus1是计算出input和lastInput具有公共前缀的位置
16         final int prefixLenPlus1 = pos1+1;
17
18         // 1.新插入的节点放到frontier数组, UnCompileNode表明是新插入的, 以后还可能会变化, 以后
19         if (frontier.length < input.length+1) {
20             final UnCompiledNode<T>[] next = ArrayUtil.grow(frontier, input.length+1);
21             for(int idx=frontier.length;idx<next.length;idx++) {
22                 next[idx] = new UnCompiledNode<>(this, idx);
23             }
24             frontier = next;
25         }
26
27         // minimize/compile states from previous input's
28         // orphan'd suffix
29
30         // 2.从prefixLenPlus1, 进行freeze冰冻操作, 添加并构建最小FST
31         freezeTail(prefixLenPlus1);
32
33         // init tail states for current input
34         // 3.将当前input剩下的部分插入, 构建arc转移 (前缀是共用的, 不用添加新的状态)。
35         for(int idx=prefixLenPlus1;idx<=input.length;idx++) {
36             frontier[idx-1].addArc(input.ints[input.offset + idx - 1],
37                                   frontier[idx]);
38             frontier[idx].inputCount++;
39         }
40
41         final UnCompiledNode<T> lastNode = frontier[input.length];
42         if (lastInput.length() != input.length || prefixLenPlus1 != input.length +
43             lastNode.isFinal = true;
44             lastNode.output = NO_OUTPUT;
45         }
46
47         // push conflicting outputs forward, only as far as
48         // needed

```

```

49 // 4.如果有冲突的话, 重新分配output值
50 for(int idx=1;idx<prefixLenPlus1;idx++) {
51     final UnCompiledNode<T> node = frontier[idx];
52     final UnCompiledNode<T> parentNode = frontier[idx-1];
53
54     final T lastOutput = parentNode.getLastOutput(input.ints[input.offset +
55     assert validOutput(lastOutput);
56
57     final T commonOutputPrefix;
58     final T wordSuffix;
59
60     if (lastOutput != NO_OUTPUT) {
61         // 使用common方法, 计算output的共同前缀
62         commonOutputPrefix = fst.outputs.common(output, lastOutput);
63         assert validOutput(commonOutputPrefix);
64         // 使用subtract方法, 计算重新分配的output
65         wordSuffix = fst.outputs.subtract(lastOutput, commonOutputPrefix);
66         assert validOutput(wordSuffix);
67         parentNode.setLastOutput(input.ints[input.offset + idx - 1], commonOut
68         node.prependOutput(wordSuffix);
69     } else {
70         commonOutputPrefix = wordSuffix = NO_OUTPUT;
71     }
72     output = fst.outputs.subtract(output, commonOutputPrefix);
73     assert validOutput(output);
74 }
75
76 ...
77 }

```

通过注释, 我们看到input是经过排序的, 也就是ordered。否则生成的就不是最小的FST。另外如果NO_OUTPUT就退化为FSA了, 不用执行第4步重新分配output了。

其中 freezeTail 方法就是将不再变化的部分进行冰冻, 又叫compile, 把UnCompileNode, 给构建进FST里。进入到FST是先进进行compileNode, 然后addNode进去的。

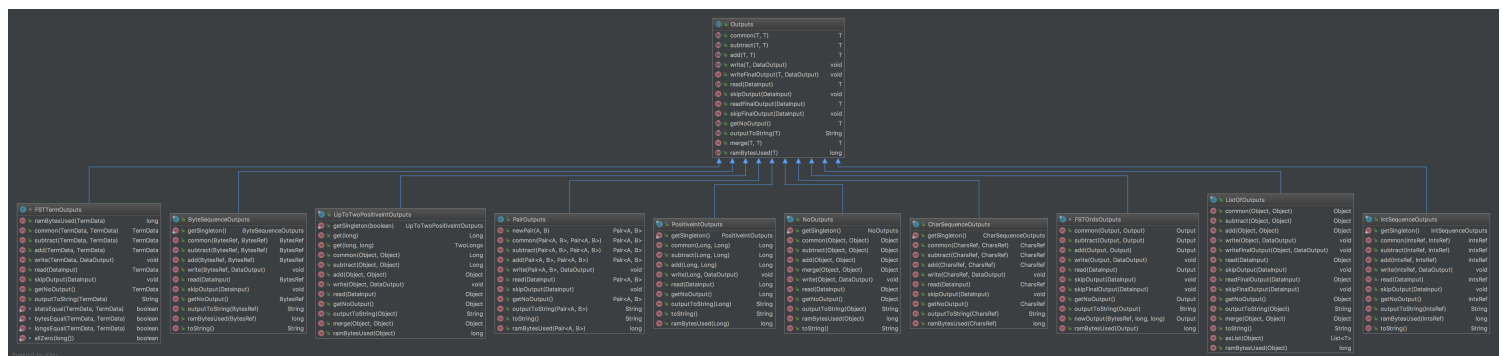
总结以下, 加入节点过程:

- 1)新插入input放入frontier, 这里还没有加入FST
- 2)依据当前input, 对上次插入数据进行freezeTail操作, 放入FST内
- 3)构建input的转移 (Arc) 关系

- 4)解决Output冲突，重新分配output，保证路径统一(NO_OUTPUT,不执行)

最后在 finish 方法里，执行 freezeTail(0)，把所有的input构建进FST内。

另外，值得注意的是Lucene里定义的Outputs类型：



其中3个method是Outputs接口定义的，有11个不同类型的实现：

- T add(T prefix, T output); 加
- T subtract(T output, T inc); 减
- T common(T output1, T output2) 前缀

完全满足我们上个部分的限制，可见就是基于之前算法的一个完整的实现。

除了在Term词典这块有应用，FST在整个lucene内部使用的也是很广泛的，基本把hashmap进行了替换。

场景大概有以下：

- 自动联想：suggester
- charFilter: mappingcharFilter
- 同义词过滤器
- hunspell拼写检查词典

总结

FST，不但能共享前缀还能共享后缀。不但能判断查找的key是否存在，还能给出响应的输入output。它在时间复杂度和空间复杂度上都做了最大程度的优化，使得Lucene能够将Term Dictionary完全加载到内存，快速的定位Term找到响应的output（posting倒排列表）。

参考文档：

[Burst Tries](#)

[Direct Construction of Minimal Acyclic Subsequential Transducers](#)

[Index 1,600,000,000 Keys with Automata and Rust](#)

[DFA minimization Wikipedia](#)

[Smaller Representation of Finite State Automata](#)

[Using Finite State Transducers in Lucene](#)

您的支持是我原创的动力！

[Donate](#)

Post author: 申艳超

Post link: <https://www.shenyanchao.cn/blog/2018/12/04/lucene-fst/>

Copyright Notice: All articles in this blog are licensed under [CC BY-NC-SA](#) unless stating additionally.



RSS

[# lucene](#) [# FST](#) [# FSM](#) [# FSA](#)

[< 利用MAT来分析JAVA内存泄露](#)

[Lucene数字类型处理 >](#)

© 2012 — 2022  申艳超

 70482 |  120229

Powered by [Hexo](#) & [NexT.Mist](#)