

## Lucene FST 算法流程

### FST 简介

FST (Finite State Transducer, 一种有限自动机, 或者称为 Mealy machine) 是 lucene 中的一个核心算法, 用于检索 term 信息存储的位置。在 lucene 中, term 按照其字典顺序排序 (term 在存储时称为 input), term 相关的信息按照 term 排序的次序存储在磁盘上 (其存储的位置为 outPut), <input,output>二元组将以 FST 的形式存储在内存中 (input 和 output 都是有序的)。检索时, 根据 input, 通过计算 FST 中的路径上的权值信息, 获取到 output 数据, 最终在磁盘上定位 term 的其它附加信息。同时 FST 还能够快速的判断一个 term 是否在 lucene 中。

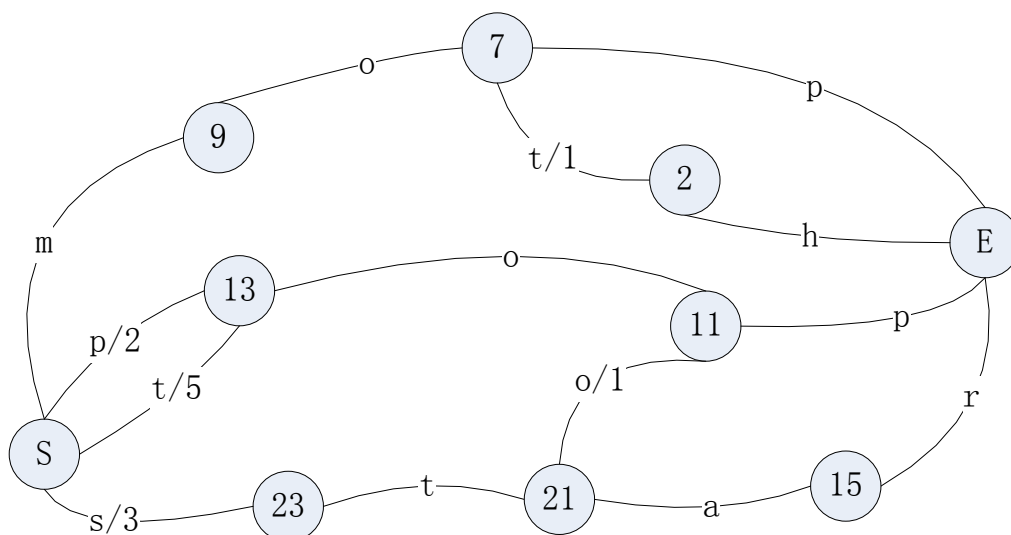
实际上 FST 相当于 term 在内存中的一个索引, lucene 使用 FST 能够快速确定系统中是否存在查询的 term, 如果存在, 能够快速定位其信息存放的具体位置。

FST 与 trie tree 结构提供相似的功能, 但是, 在内存中存储更高效。

输入到 FST 中的数据为:

```
String inputValues[] = {"mop", "moth", "pop", "star", "stop", "top"};  
long outputValues[] = {0, 1, 2, 3, 4, 5};
```

生成的 FST 图为:



从图中可以计算 stop:

s->23 的弧线 s/3,

23->21 的弧线 t,

21->11 的弧线 o/1,

11->E (E 表示终止状态) 的弧线 p

将所有弧线的权值加起来  $3+0+1+0 = 4$ ; 因此 stop 对应的 output 为 4.

## FST 的构建

### 数据结构:

1. FST bytes, FST 中的 bytes 数组, 存储 FST 数据信息。FST bytes 存储了 FST 图的完整信息 (通过 FST bytes, 可以构建一个 FST 图)。
2. FST Node, 表示 FST 图的一个节点, 有两种实现类型, UnCompiledNode (尚未存放到 FST bytes 中的节点) 和 CompiledNode (已经存放到 FST bytes 中的节点)。
3. FST arc, 用于表示 Node 间的弧线。

4. FST HashMap，一个使用探测法实现的 HashMap，key 是 FST Node 生成的 hash 值，value 是 FST node 存放在 FST bytes 数组中的下标。

（FST HashMap 不是 FST 必须的组成部分，但是，HashMap 能够加快判断某个节点是否已经在 FST bytes 中，HashMap 仅用于 FST 的构建过程）。

5. Frontier 是一个数组，用于存放未转换到 FST byte 数组中的数据信息；

### Hash 算法：

Hash 算法并不是 FST 中必不可少的部分，在构建过程中 FST 使用 HashMap 来快速判断测试节点是否已经被写入到 FST bytes 中。HashMap 的 value 是 node 在 FST bytes 中的实际存储位置（数组的下标）。

HashMap 中的 key 通过计算节点中所有的 arcs 信息生成的。其中，每个 arc 被考虑的属性包括，label（字符）、targetNode（下一个节点的地址）、outPut（节点输出）、isFinal（是否是终止弧线）、nextFinal（下一个终止弧线—这个参数在测试过程中没有发现被赋值）。

将 node1 添加到 FST bytes 时，先通过 hash 算法计算 node1 对应的 key，如果 key 在 Hashmap 中已经有对应的值 value，那么这个 value 就是与 node1 hash 值相同的节点（称为 node2）存储在 FST bytes 中的实际数组下标，FST 从 bytes 数组中转换出 node2，判断 node1 和 node2 是否相等（所有的 arc 值是否相等），如果相等，node1 就没有必要再添加到 FST bytes 中（相当于 FST 的尾部进行了归并）。

参考: [org.apache.lucene.util.fst.NodeHash<T>](https://lucene.apache.org/core/3.9.0/org/apache/lucene/util/fst/NodeHash.html)

### 算法基本步骤:

1. 对于新的输入 NInput (new input), 首先要计算其与上次输出的 LInput (Lastest input, LInput 数据存放在 Frontier 中) 之间的公共前缀 prefix, 之后调用 freezeTail, 将 LInput 的后缀部分转换到 FST bytes 数组中 (注意, LInput 的 prefix 的直接后缀那个元素没有存放到 FST bytes 中)。
2. 在 freezeTail 过程中, 会将 LInput 的后缀, 从后向前的顺序逐个存储到 FST bytes 中, 在存储前, node 通过 Hash 算法判断其是否已经在 bytes 中, 如果不存在就保存, 并更新 HashMap; 如果已经存在, 通过 HashMap 会返回节点在 bytes 数组中的实际位置 pos, 通过这个 pos, 可以在 bytes 中转换出一个先前存入的 preNode, 比较 preNode 和 node, 如果相同, 直接返回 preNode 的存储位置, 如果不同, 需要生成一个新的 HashKey, 重复上面的判断 (探测法的 hashmap) 直到
  - a) HashMap 中找不到 key 对应的 value, 将新节点添加到 FST bytes 中 (添加节点, 实际是将节点包含的所有 arc 都写入到 FST bytes 中, 并不是将节点本身写入 FST bytes 中);
  - b) HashMap 中找到 key 对应的 value, 并且转换出的 preNode 与当前 node 的值相等, 直接返回 preNode 的存储位置 pos, 相当于尾部节点的合并。

存储的 Flag 字段的含义：

每个 arc 存储在 FST bytes 中都会有一个 flag 字段，FST 通过 flag 字段可以判断出 FST bytes 中有几个 bytes 与当前 Node 相关，并且能够计算出当前节点的出度和每个出度存储的位置等信息。

Final arc: 1 表示这是一个 Final arc，arc 指向一个终止状态 E。

Last arc: 2 表示 arc 是当前 node 节点的最后一个 arc；

Target next: 4 当前 arc 的下一个 arc 是当前节点最后一个 arc 的向前的一个 arc（数组位置，向前邻接的那个 arc）。

Stop node: 8 终止状态 E。

Has output: 16 当前 arc 有 output 值（表示 output 不为 0）

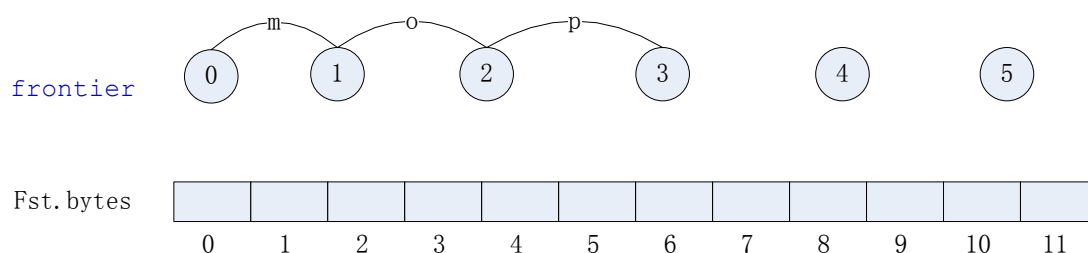
### 构建过程模拟：

Input 和 output 的数据：

```
String inputValues[] = {"mop", "moth", "pop", "star", "stop", "top"};  
long outputValues[] = {0, 1, 2, 3, 4, 5};
```

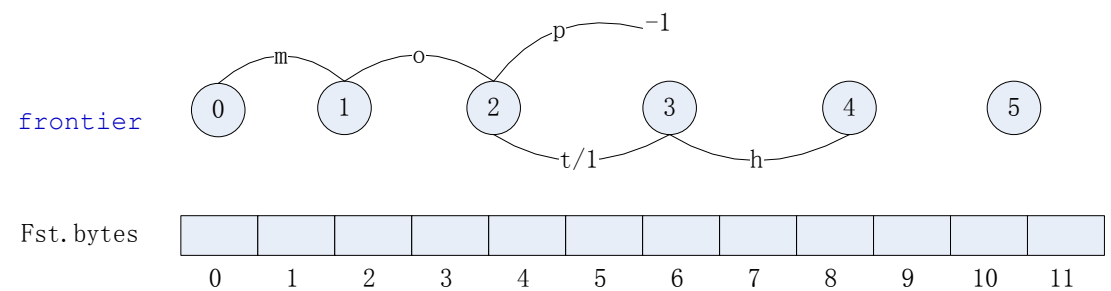
1. 输入 mop 后：

输入 mop 后，m/o/p 三个弧线的信息都保存在 frontier 中（frontier 表示尚未存放到 FST bytes 中的数据）。FSTbytes 数据位空。



2. 然后输入 moth，freezeTail 之后得到：

输入 moth 时，moth 与先前的 mop 有 mo 两个公共前缀，在 freezeTail 过程中，将先前弧线 p 后面的节点（节点 3）存储到 FST bytes 中，由于节点 3 没有任何出度，因此 FSTbytes 中的数据仍然是空，而 p 的下一个节点是 FSTbytes 中的-1 的位置（-1 表示终止）。之后将 th 添加到 frontier 中，增加两个弧线 t/1 和 h。

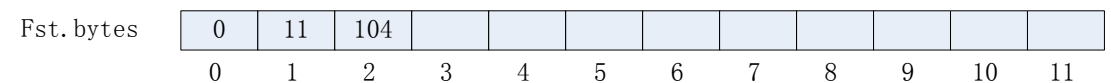


3. 输入 pop，并且 freezeTail 如下：

输入 pop 后，pop 与 moth 没有公共的前缀，因此在 freezeTail 过程中，需要将 frontier 中的 4,3,2,1 都存储到 FST bytes 中。

(1) 由于 4 没有出度，因此 3->4 的 h 的弧度的下一节点为-1。

(2) 存储节点 3，有一个出度 3->4，值为 h，因此如下图：11 表示  $8+2+1=stopNode(-1)+lastArc+FinalArc$ 。stopNode 说明当前弧是一个指向终点的弧（stopNode，FinalArc），并且当前弧是当前节点的最后一条弧线（FinalArc）。104 是 h 的 ascii 值。



注：在每个节点被写入后，会将这个节点的所有数据倒置，写入结束后，FST bytes 变为（0,104,11）。数组 0 的位置永远存储的

是 0.

(3) 存储节点 2，节点 2 有两个弧度：

a) 先写入 p, 9 和 112。9= stop\_node+ final\_arc，表示弧线 p 是终止节点。

b) 写入 t/1, 22、116 和 1。22 表示  $16 + 4 + 2 = \text{has\_output} + \text{targetNext} + \text{last\_arc}$ ，说明，t/1 这个弧线有 output，弧线的后继弧线是紧邻着的那条（就是 FST bytes 中左侧（小端）的那条）并且是当前节点的最后一个 arc。如下图

Fst.bytes	0	104	11	9	112	22	116	1				
	0	1	2	3	4	5	6	7	8	9	10	11

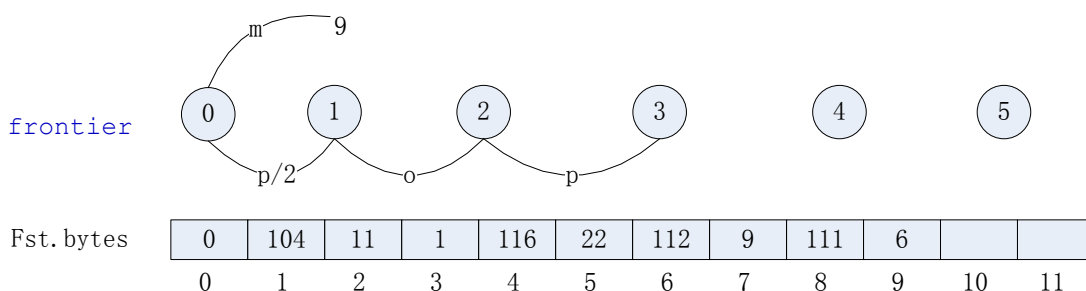
（由于存储结束后 fst 会将节点数据倒置，因此与 t/1 邻接的弧线就是 h 那条弧线）反转后，如下图：可以看出 t/1 的向前邻接的节点是 h。

Fst.bytes	0	104	11	1	116	22	112	9				
	0	1	2	3	4	5	6	7	8	9	10	11

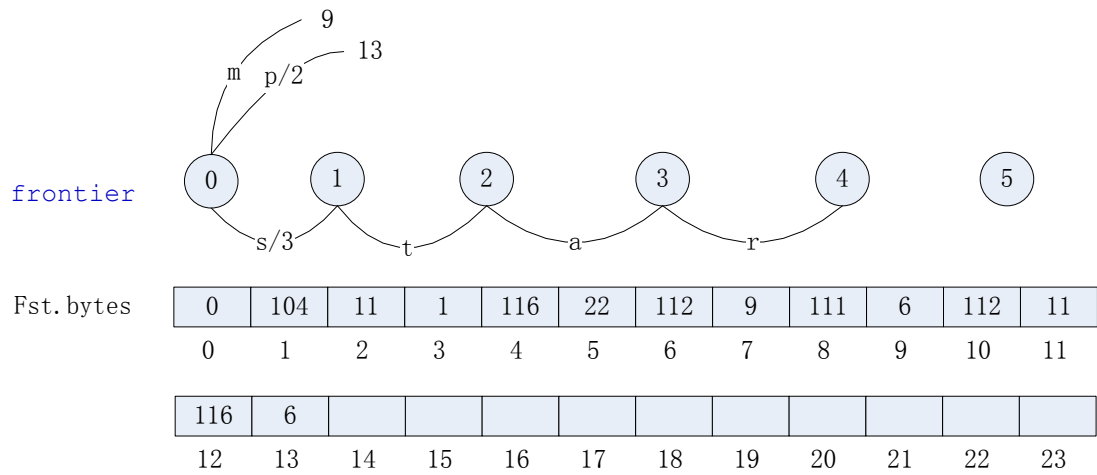
4) 存储节点 1 并反转后，6 表示 target\_next 和 last\_arc.

Fst.bytes	0	104	11	1	116	22	112	9	111	6		
	0	1	2	3	4	5	6	7	8	9	10	11

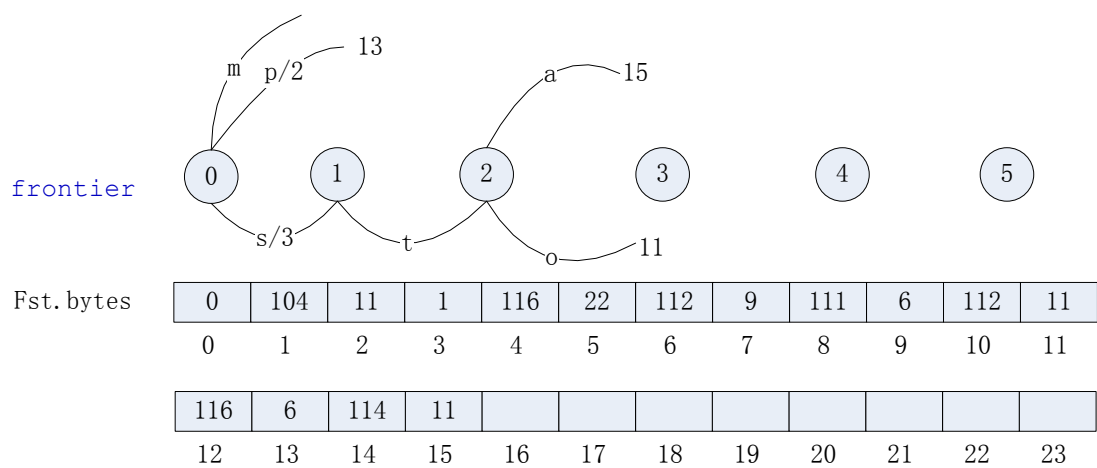
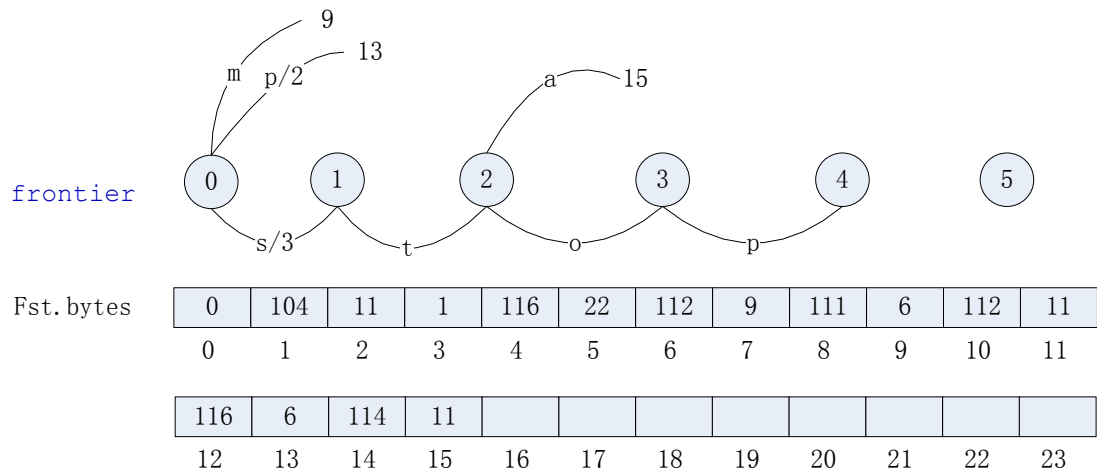
5) freezeTail 结束后，将 pop 放入 frontier 中。如下图。



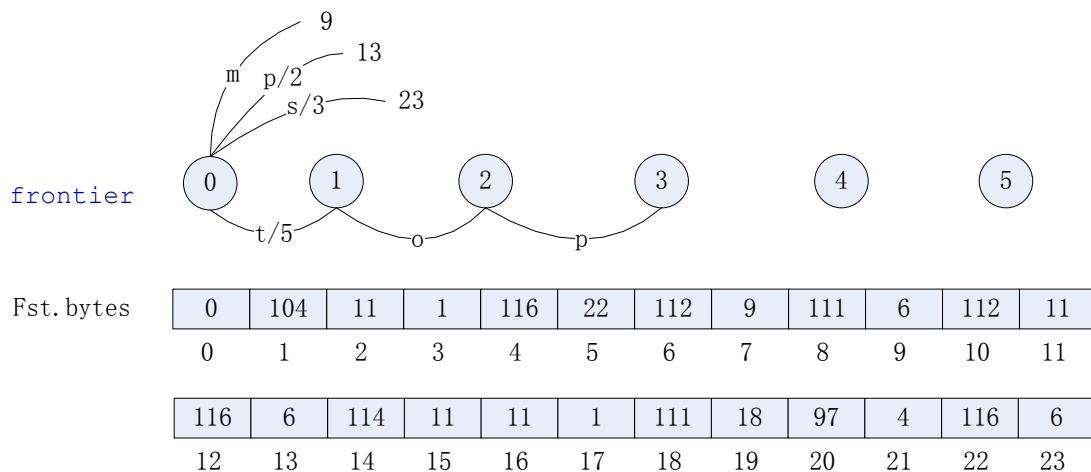
4. 输入 star



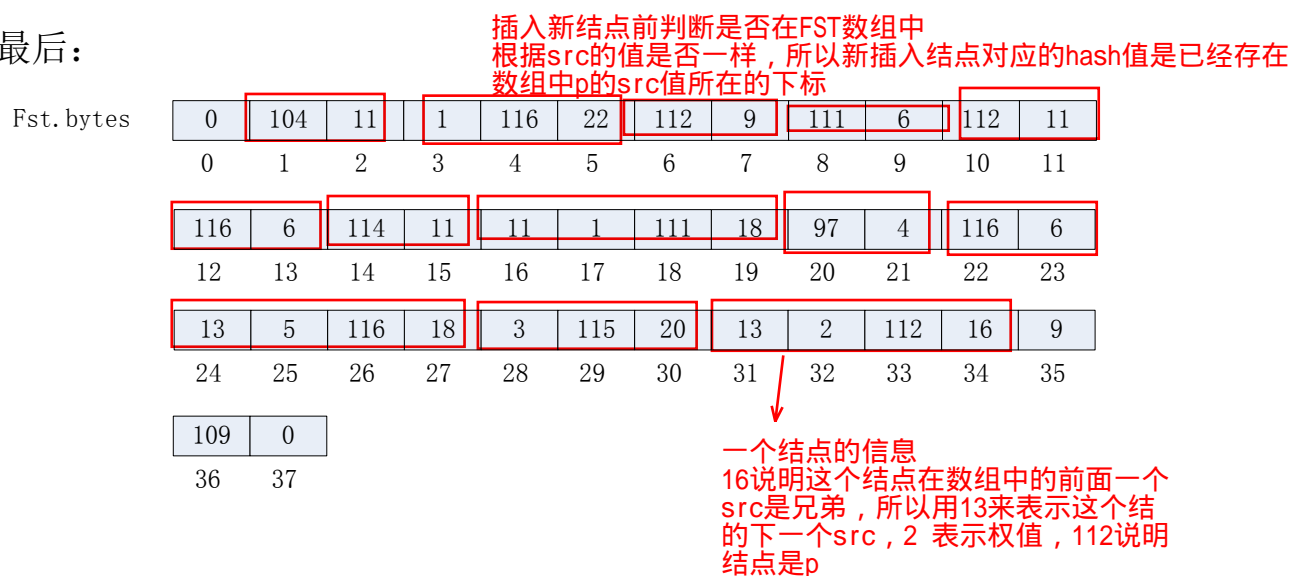
## 5. 输入 stop



## 6. 输入 top



最后:



## 还原 FST 图

根据 byte 数组, 还原 FST 图, 由 24-37 可以获得第一个节点的出度:

1. 第一个节点 37 号开始 (如下图)

1) 37 flag 为 0, 表示不是 last arc, 不是 Target\_next, 不是 has\_output.

不是 last\_arc 表示邻接的是兄弟 arc, 不是 has\_output 表示没有输出,

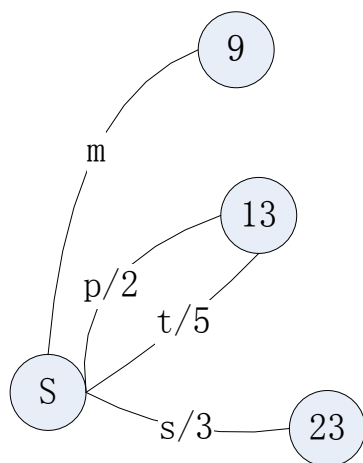
不是 target\_next 表示存储了 arc 的下一个 arc 的地址, 因此 9 位置表示下一个 arc 的起始地址。

2) 34 flag 为 16 表示有 output, 没有 last arc, 邻接的是兄弟 arc,

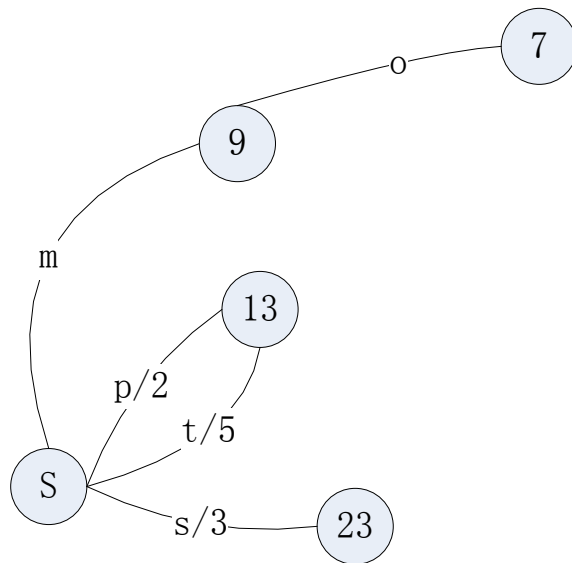
没有 target\_next, 因此, 2 是 output, 13 是下一个 arc 的位置。

3) 30 flag 为 20 表示有 output, 没有 last arc, 有 target\_next, 因此 3 表示 output, 邻接的是兄弟 arc, 节点所有的 arc 读出后, 邻接的 arc 是当前 arc 的下一个节点。

4) 27 flag 为 18 表示有 output, 是 last arc, 不是 target\_next, 不是 final。当前 arc 的下一个 arc 为 13, output 是 5. 因此 3) 的下一个 arc 是 23. (因为 27 表示的 arc 已经是当前节点的最后一个 arc (即 last\_arc), 因此邻接的节点 23 是标有 target\_next (就是 30 那个) 的下一个 arc)。

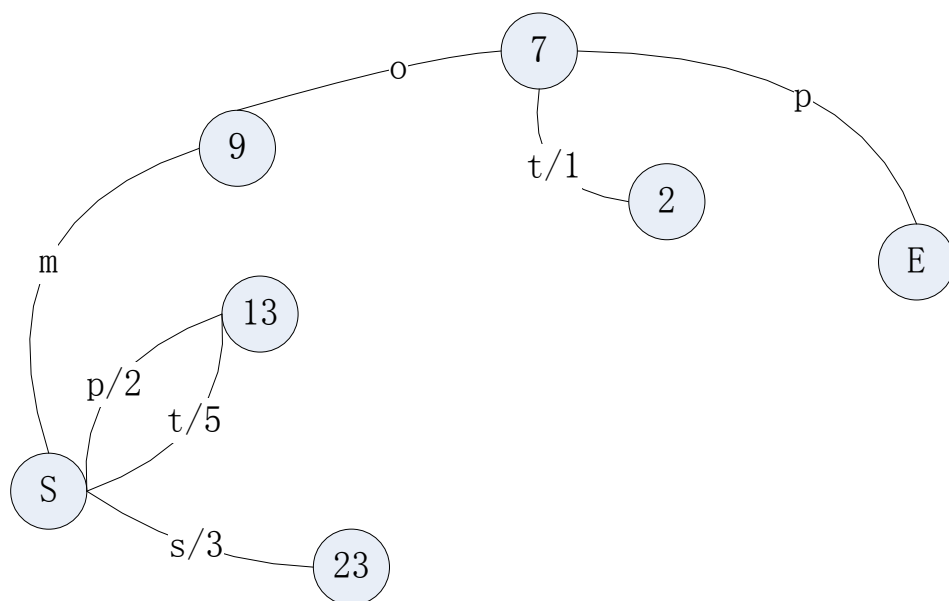


3. 由 9 的出度得到如下: flag 是 6, 是 last\_arc 和 target\_next, 因此 7 是当前 arc 的下一个 arc, 而且当前节点仅有一个出度。



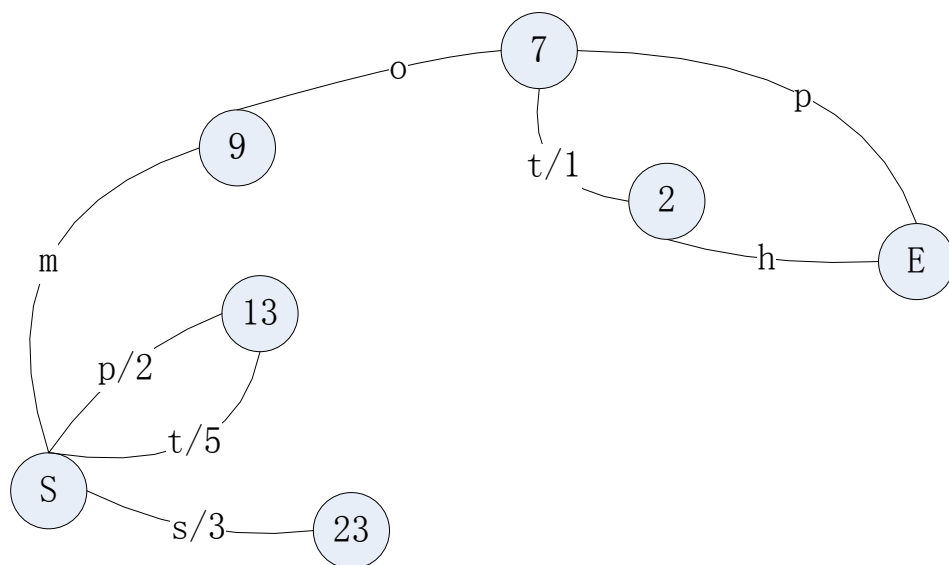
4. 有 7 的出度得到如下：flag 是 9，表示是一个终点 arc，arc 的指向结束状态 E。并且不是 last\_arc，因此，5 就是其兄弟节点。

5 flag 是 22，has\_output,target\_next,last\_arc，因此 1 是 output，并且是本节点最后的 arc，当前 arc 的下一个 arc，是邻接的那一个(就是 2 那个)。

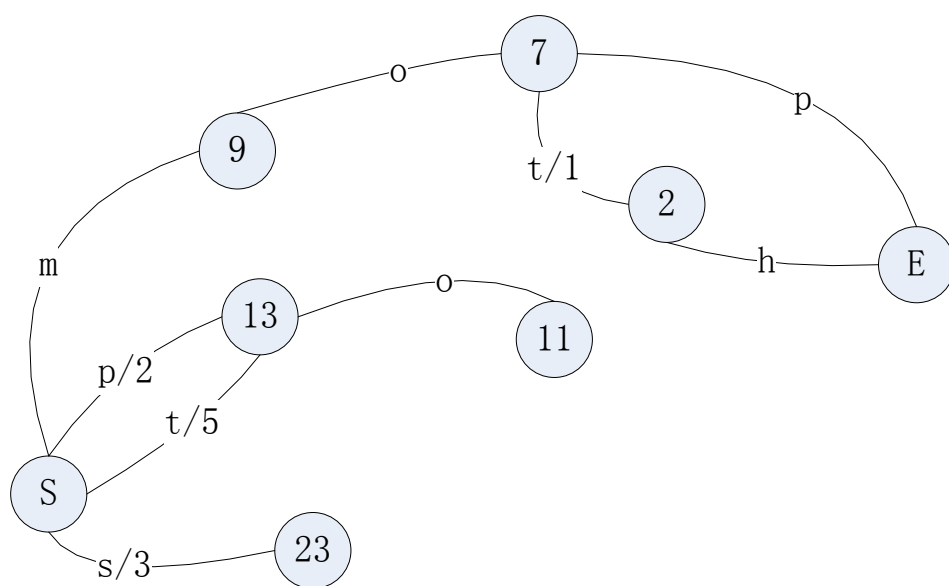


5. 由 2 的出度如下，flag 是 11，表示 stop\_node, final\_arc,last\_arc，

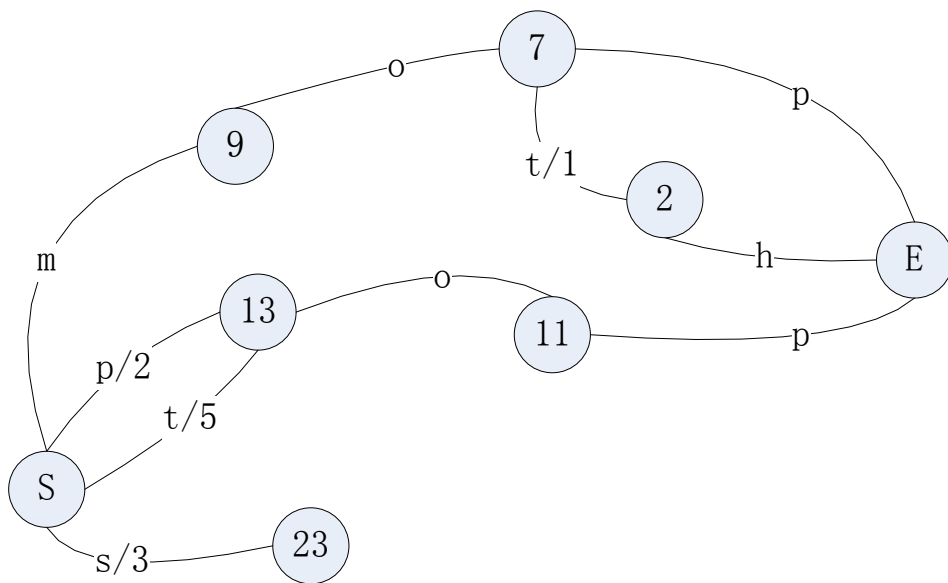
因此当前 arc 是当前节点的最后一条 arc，并且是一个终止 arc。



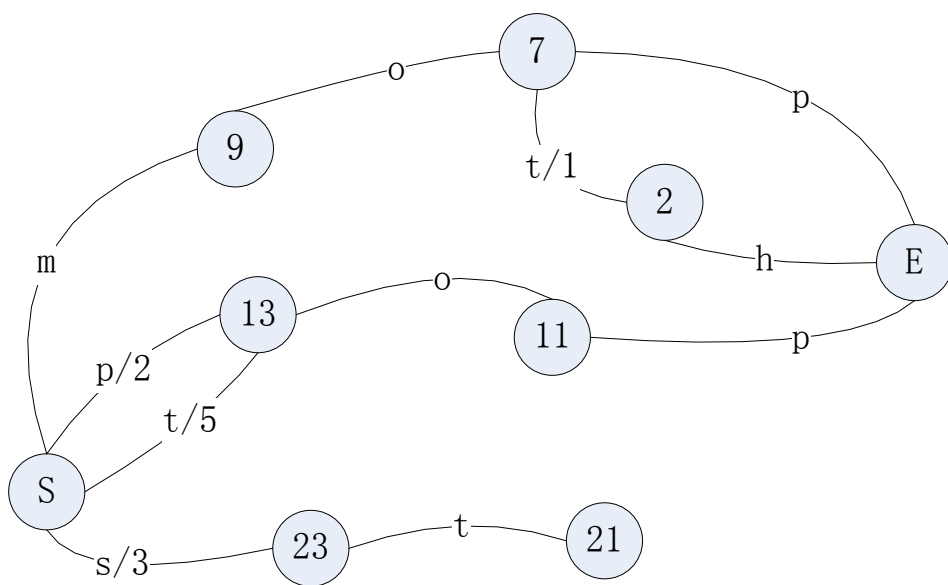
6. 由 13 的出度如下：



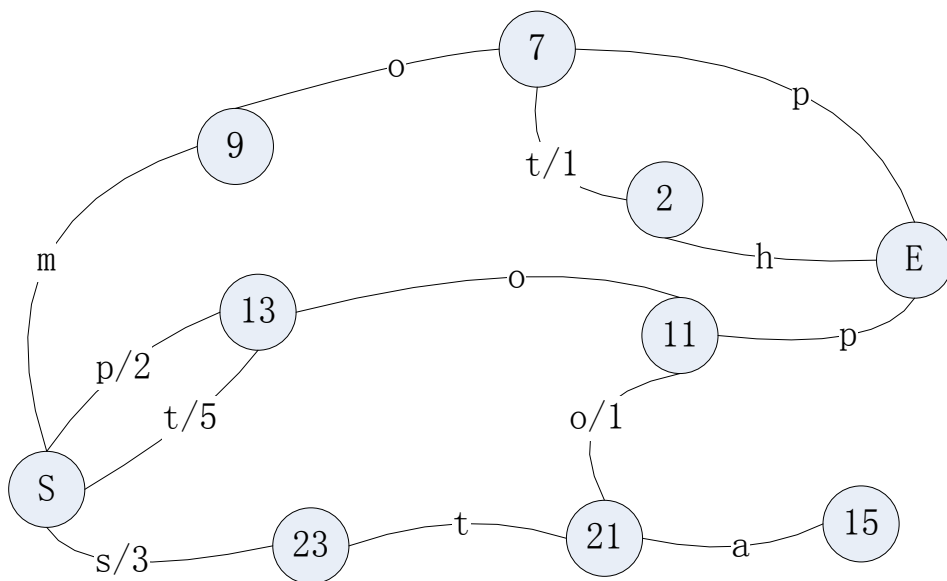
7. 由 11 的出度如下图：



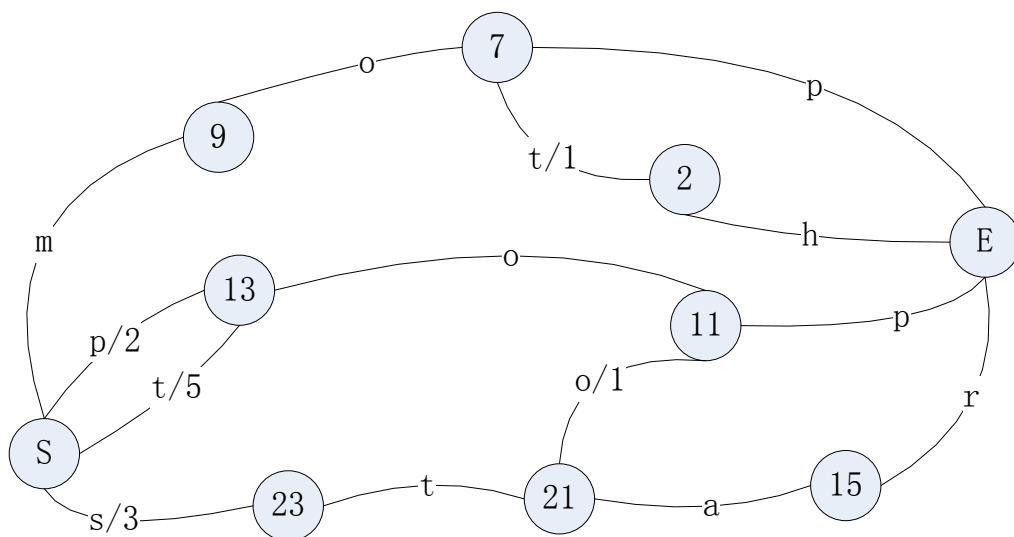
8. 由节点 23 的出度得到:



9. 由 21 的出度获得:



10.最后 15 的出度:



## 检索过程

基本数据结构:

1. `cachedRootArcs` 一个 128 个元素的 `arc` 数组, 存储每个 `ascii` 字符对应的起始弧线。例如: 要检索以 `a` 开始的 `term`, `cachedRootArcs[97]` 就是起始弧线。

注意, 系统内部仅存储以上图 `S` 开始的那些弧线, 其它弧线都是通过

对 FST 的 bytes 数组计算获得。

因此，在 term 的查询过程中，首先通过 cachedRootArcs 获取 term 首字母对应的弧线（即其在数组中对应的下标位置），之后按照上面提到过的方式转换其它弧度，按照图的查找算法。如果要查找以某个 term 开始的所有 terms，则按照上述方法，先查找到 term，然后遍历这个 term 下的所有 terms 即可。

从分析可以看出，FST 的这个索引在内存中仅用一个 bytes 数组，额外加上少量的 cachedRootArcs 数组空间，即可。是十分节省内存空间的，但是查询时可能比较消耗 cpu 的计算。

### 小结

FST 的算法十分复杂，描述和理解起来都比较困难，比较容易学习的方法就是看 lucene 的源码，通过 debug 的方式，查看关键数据结构中的值，来理解算法原理。笔者从看源码到理解花费了一周左右的时间。有兴趣的同学也可以自己去看看相应的源码。本文是以 lucene4.8.0 为基础分析的。

Lucene 通过使用 FST 在内存中存储了 term 的一份索引，紧凑的数据格式(FST bytes 数组)，不但减少了内存使用量，还有效的提高了 term 的检索效率，是一个非常优秀的算法。

### 参考资料

<http://blog.mikemccandless.com/2010/12/using-finite-state-transducers-in.html>

Direct Construction of Minimal Acyclic Subsequential Transducers.

Applications of Deterministic Finite Automata.

下面的博客虽然有地方有些错误，但对初学者理解比较容易：

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

关新全

转载请标明出处