正则表达式_如何写高效的正则

笔记本:	metasploit_update		
创建时间:	2022/8/26 11:22	**更新时间:**	2022/8/31 15:46
作者:	zhukeyu@qianxin.com		
URL:	https://blog.csdn.net/yeshang_lady/article/details/121756563		

目录

pcre简单介绍

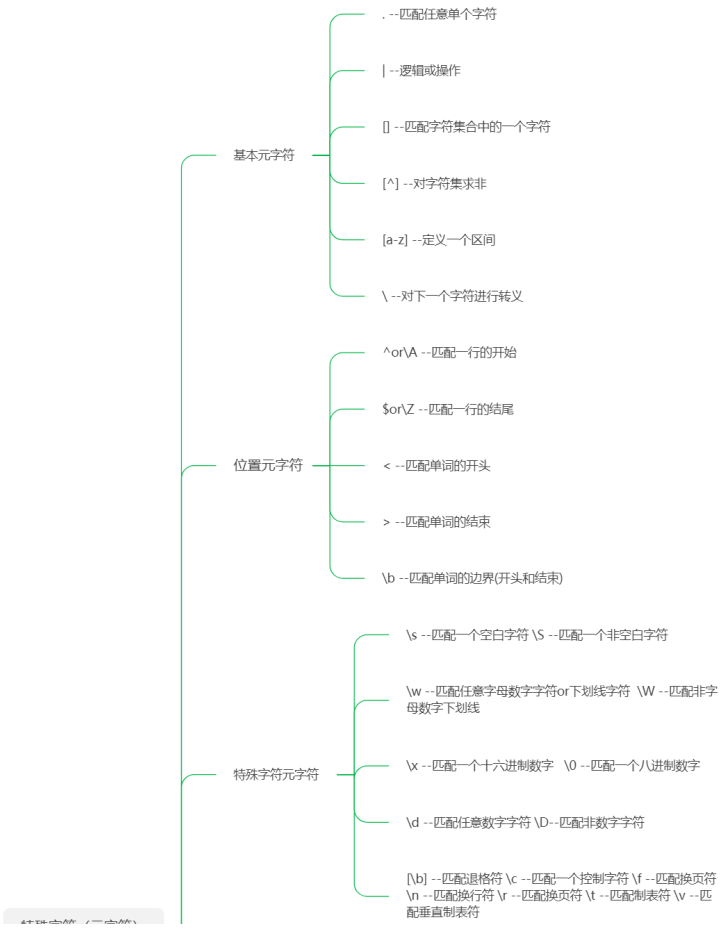
正则表达式：一种字符串匹配的模式（pattern）

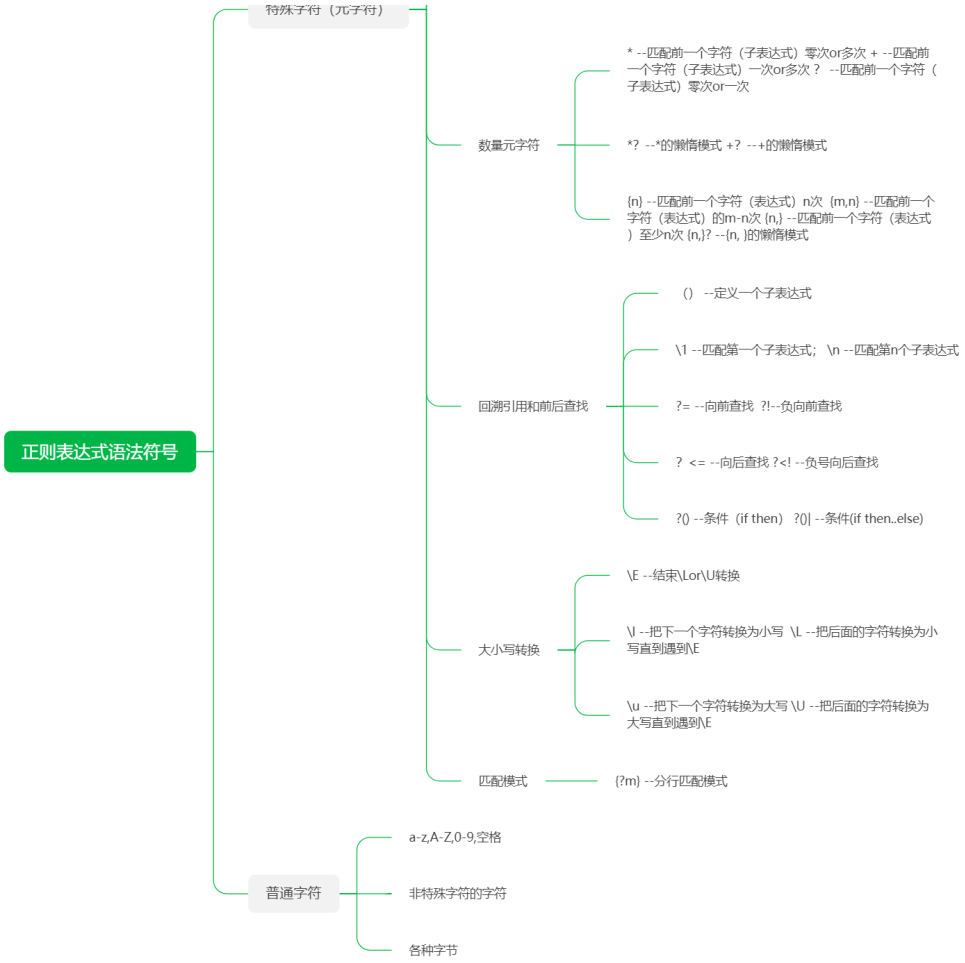
perl语言，以正则表达式为基础，开创了一个新的流派，perl流派，perl语言是一种擅长处理文本的语言。

正则表达式作为表示复杂语义的模式串的有力工具，它的定义是递归的：一个正则表达式要么是一个简单的字符串，要么是正则表达式的串联、关联或重复。

语法

正则表达式由两种字符构成：特殊字符（元字符）和普通字符。





零宽断言是一种零宽度的匹配，它匹配到的内容不会保存到匹配结果中去，最终匹配结果只是一个位置而已。
正则表达式中的锚点和零宽断点都不会匹配实际的字符而是寻找和定位字符在文本中的位置，可以将其认为是定位符。

匹配原理

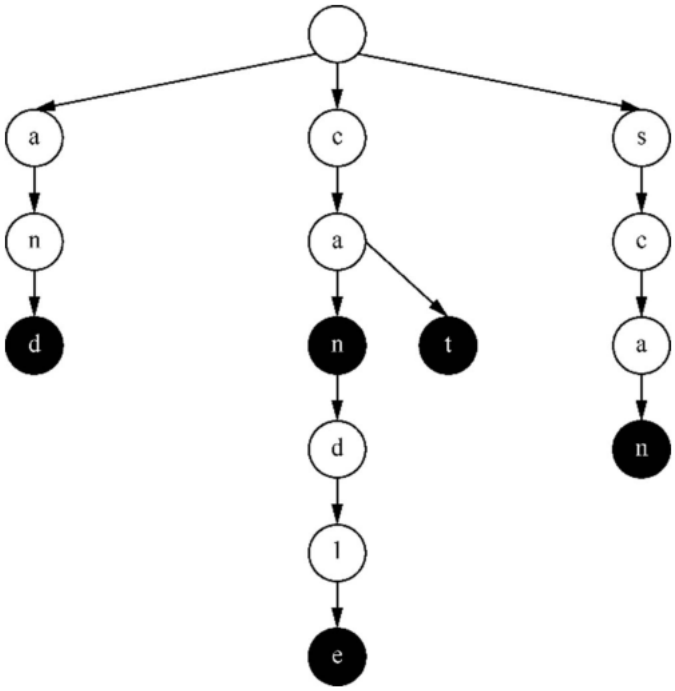
自动机理论中的匹配算法

匹配正则表达式的算法相当复杂，只有在模式串不能更简单的方法来表示的时候才会使用正则表达式来表示。

正则表达式的**匹配**通常是用**有限自动机**来解决的，具体的又可以分为非确定性有限状态自动机（Non-deterministic Finite Automata, NFA）和确定性有限状态自动机（Deterministic Finite Automata, DFA）两种。

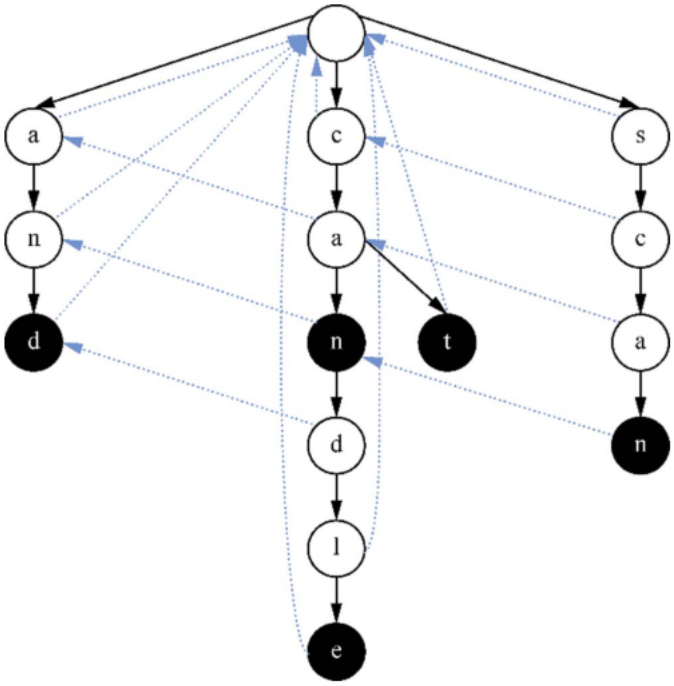
> 字符串其实是正则表达式的特殊情况，它本身不一定需要借助生成自动机来匹配，他本身的匹配算法具有更丰富的算法可选择。（单字符串：KMP\BM 多字符串：AC\SHIFT-OR算法）

非确定性有限状态自动机，其“非确定”的含义是指在NFA的某个状态上，通过某个字符串输入可能会跳转到多个后继状态，也可能没有后继状态。
NFA在任意时刻可以存在多个激活状态，在运行时需要保存所有状态的激活情况，并对所有激活状态进行处理。
可以用AC-算法（前缀树-》NFA-》DFA）简单说明NFA和DFA
前缀树orNFA：

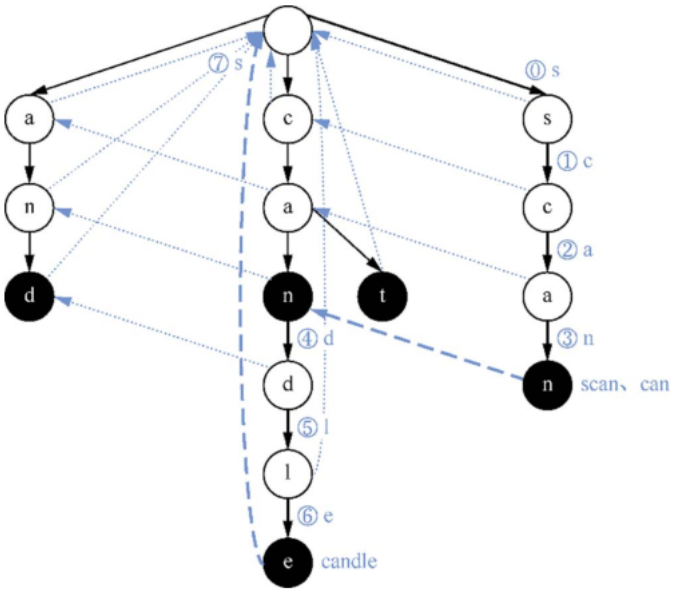


以{and,can,candle,cat,scan}为例构造一个前缀树，此前缀树也可以看作是一个NFA,此时可以推演每个位置都匹配时候的跳转

DFA:



将NFA的每个节点求最大严格后缀，就生成了DFA树，根据该图可以1.指导某位置失配时的跳转2.在某位置匹配时报出所有获得匹配的字符串。
scandles DFA的匹配过程：



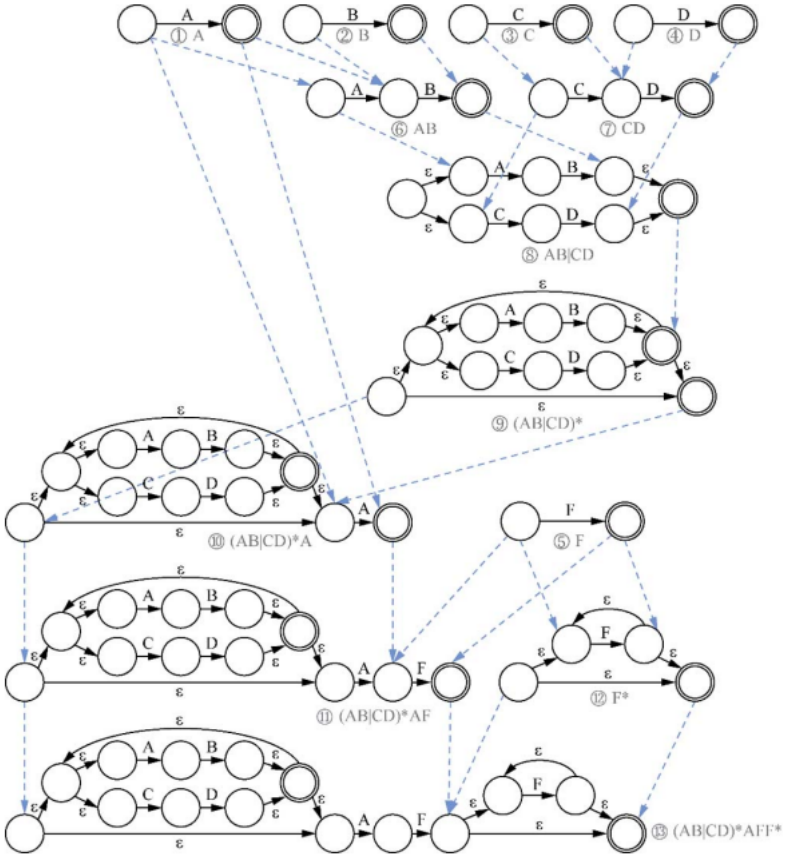
依次运行每一个字符并运行DFA

NFA

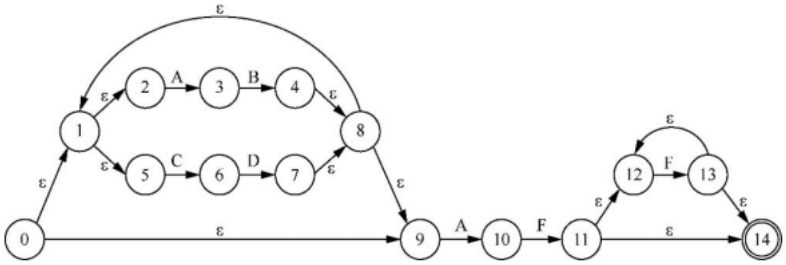
根据正则表达式构造NFA的方法：
Thompson构造法和Glushkov构造法。这两种构造法都通过递归的方式定义了3种运算的构造规则和递归出口（单个字符或空字符），不同的是Thompson构造法基于NFA子图进行构造，而Glushkov构造法基于状态和状态对集合进行构造。
当我们进行构造的时候首先需要确定正则表达式中每个运算符的计算顺序，正则表达式使用的是中缀表示法，为确定运算顺序，可以先将正则表达式转换为后缀表达式，然后按后缀表达式中运算符出现的先后顺序进行计算。首先定义运算符的优先级，3中正则运算有： “Kleene闭包” > ‘连接">"并"。算法使用一个运算符栈做辅助，以及一个字符和运算符的混合队列来保存转换的结果。从前往后处理中缀表达式时，遇到操作符直接传入结果队列，遇到左括号需增加其后运算符的优先级，遇到右括号需减小其后运算符的优先级，遇到运算符则须将其优先级与栈顶运算符的优先级做比较，若不大于栈顶运算符则令栈顶运算符出栈并传入结果队列，重复这个过程直至当前运算符的优先级大于栈顶运算符或栈空为止，再将当前运算符入栈。当处理至中缀表达式末尾时，将栈内所有运算符出栈并传入结果队列。最终结果队列中保存的便是对应的后缀表达式。
举个例子：
(A|B)*AFF* 括号增加运算符优先级>Kleene闭合>连接>并
转换成后缀表达式： AB.CD.|*A.F.F*

Thompson构造法

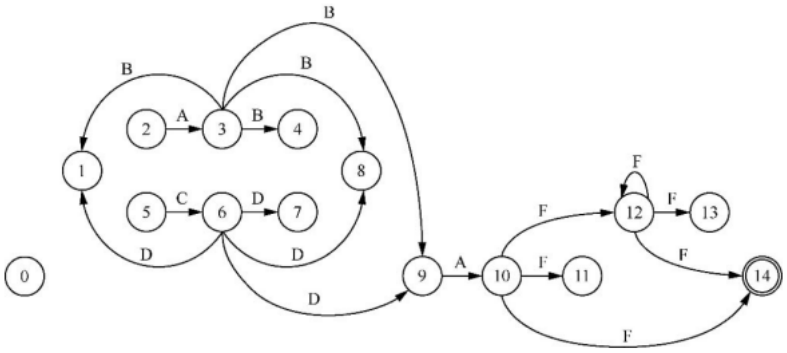
以(A|B)*AFF* 为例构造NFA的过程如图：



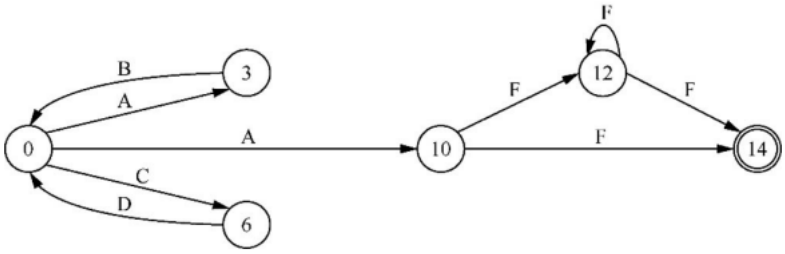
最终给构造出的Thompson NFA状态编号，如



从上面生成的NFA图中我们看到状态繁多而且含有大量空字符的跳转，这种带有空字符跳转的NFA无形中增加了运行的复杂度。为了增加效率，一般情况下需要去除空字符跳转。去除空字符跳转如下图所示

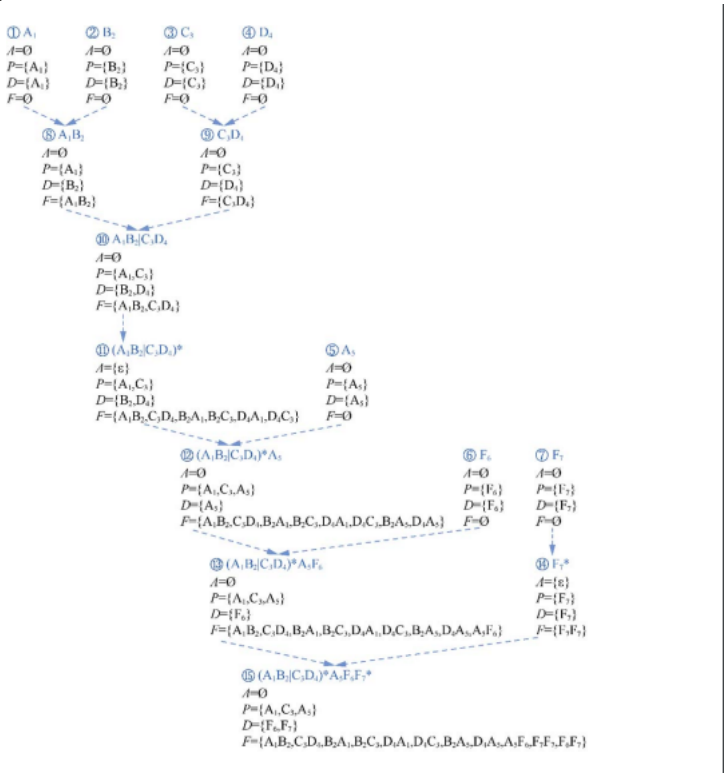


进一步对此图进行简化，首先移除无用的状态以及合并初始状态，最后的状态图如下

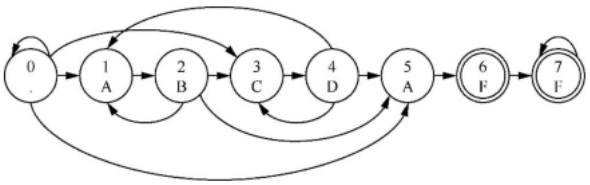


Glushkov构造法

Glushkov构造法是一种基于位置的构造法，而并非像Thompson构造法那样直接进行图的构造。Glushkov比Thompson要简洁一些，不包含空字符跳转，而且对任意一个状体而言其所有的转入条件都相同。这样的好处在于跳转条件不需要保存在NFA图的每条边上，只需要保存在每个节点中作为节点的激活条件即可。据此可以实现高效的NFA模型。
在Glushkov构造法中，出现在表达式中的每个字符都占据不同的位置，每个位置对应一个独立的状态。对正则表达式中每个位置的字符从1开始编号，对应NFA图中的状态id.编号0预留给唯一的初始状态。
Glushkov构造法依赖正则表达式的几个集合的信息：初始状态集(函数P)、结束状态集(函数D)、连接状态对集合(函数F)，以及每个子表达式是否可为空(函数^)
同样以(A|B)*AFF*为例构造的过程如下图：



最后构造完成的图如下

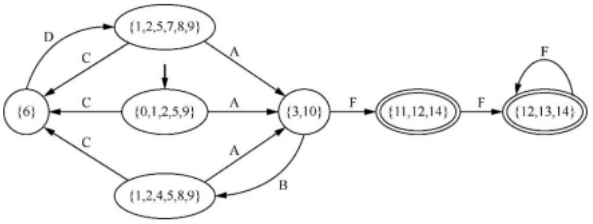


显而易见，Glushkov NFA比Thompson NFA要简洁的多，且由于跳转条件被转移到节点内变成了节点的激活条件，运行时的处理也变得比较方便，在运行时读入一个字符c时，同时也知道了字符c的可激活状态集，那么只需要在Glushkov NFA当前的激活状态集s的基础上计算其后继状态集。两个状态集的交集就可以作为下一刻的激活状态。而对于Thompson NFA，节点的激活条件并不一定唯一，则运行时不能采用这种方法。

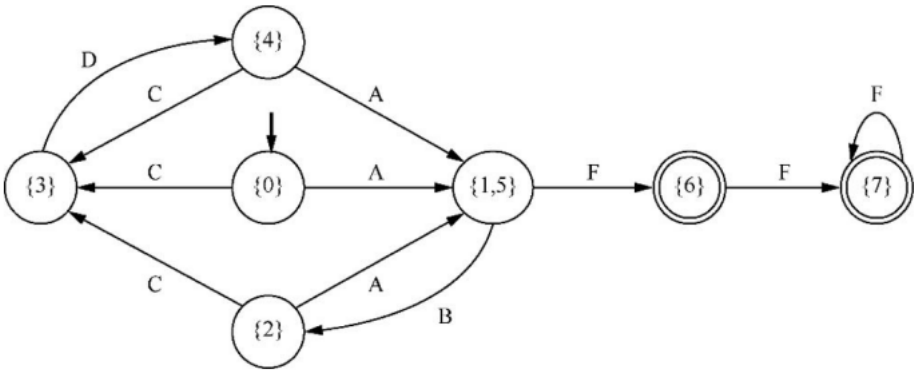
DFA

确定性有限状态机也可以识别正则表达式，它是词法分析和模式识别领域中的常见工具。
“确定性”指的是自动机运行过程中下一状态转移的唯一性；在DFA中，每读入一个输入符号，当前状态都会确定地转入由状态转移规则所指定的下一

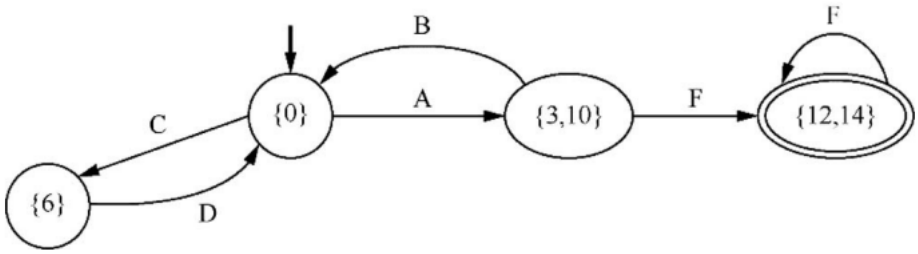
个唯一的状态。DFA仍然有初始状态和结束状态，但在自动机运行的任何时刻，都有且仅有一个激活状态。
(A|B)*AFF*的DFA状态转移图



基于Glushkov NFA的DFA状态转移



基于去空字符跳转的Thompson NFA的DFA状态转移



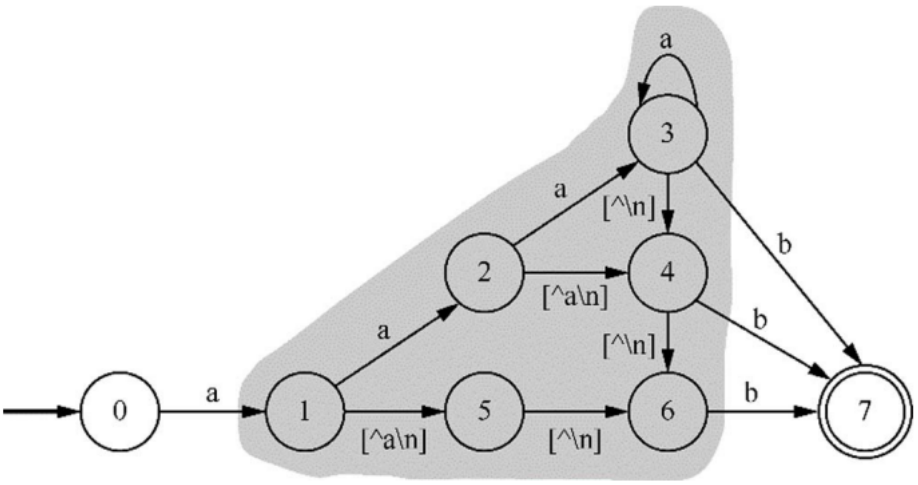
从上面的图中我们看到转换后的DFA的状态数比原来的NFA的状态数要少。

DFA的实际规模：

在之前举得例子中，我们看到转换后的DFA的状态数比原NFA的状态数要少，但在实际应用中的DFA状态数和原NFA状态数通常差不多，甚至可以说DFA状态数应该远多于原NFA状态数。
（1）等量规模
当正则表达式语法十分简单时，例如没有任何“重复”子表达式，除了一些通配符，或者字符类剩下都是普通字符，其对应的DFA和NFA在本质上和纯字符串匹配类似，每个非结束状态在输入成功时只能转入唯一的下一状态。在这种情况下，两类有限状态自动机的数目几乎一致。

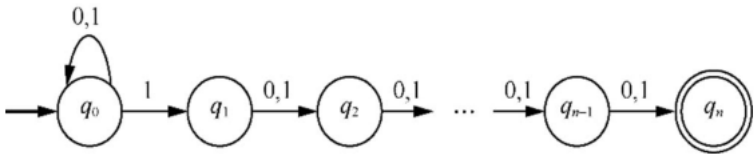
在引入若干简单的重复语法元素，如.*,此时NFA和DFA的状态数仍然不会有显著的区别，因为它们的存在只会影响不同自动机上的转移行为，而不会影响状态数。
（2）线性增长规模
虽然.*不能使单条正则表达式形成DFA的状态爆炸，但是当多条表达式一起生成有限状态自动机时，复杂的变化就会****发生。
多条正则表达式同时转化为DFA很常见，.*语法元素也很常见。这类实例带来的DFA状态数相对与NFA状态数来说是线性增长的，因为每出现一处.*,其内部就会将其他简单形式的子表达式结构复制一份。
（3）平方增长规模
.*经常出现在正则表达式的开始位置，它意味者后续的子表达式可以在输入的任意位置开始匹配，这是正则表达式编写人员非常熟悉的书写习惯。另一种通常出现在正则规则开始位置的语法元素是^，表示必须从输入的开始位置进行匹配。和.*不意味着表达式的复杂类似，^本身也不意味着表达式的简单，一切取决于紧跟其后的子表达式。
当字符类的无限重复和有限重复产生关联时，DFA状态数又是另一种情形。

以规则^a+[^\n]{2}b为例，它的DFA中某些结构的状态数想对NFA来说是平方规模的。

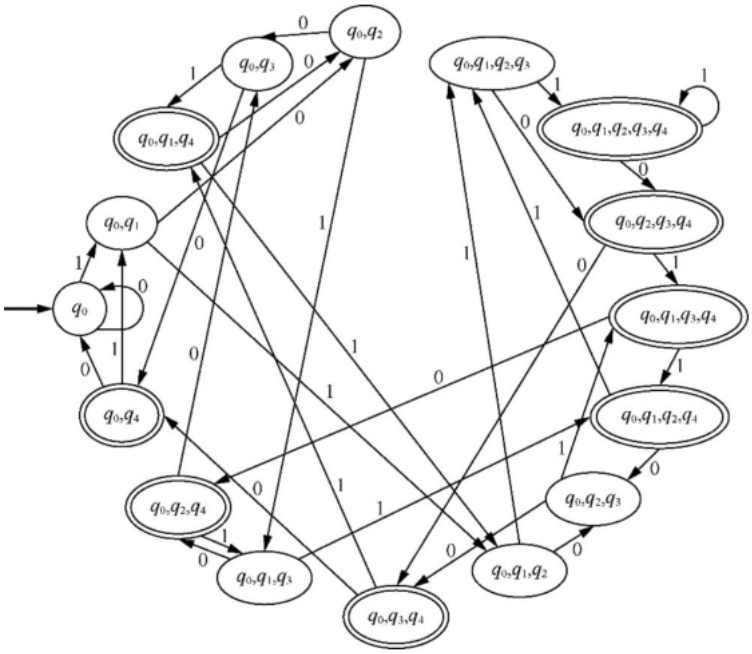


平方的高复杂度源自字符集的有限重复[^\n]{2}与规则前缀中的无限重复a+产生了重叠，它需要记住已经出现的字符a的个数和位置，以便为下一个字符找到正确的转移目标。
符合平方增长的例子，^SEARCH\s+[^\n]{1024}就是符合平方增长的例子。空白字符类\s被字符类[^\n]所包含，输入连续的空白字符会提高转移行为的模糊性。特别的当输入中紧跟字符串SEARCH后出现1024个空白字符和1024个字符a时，规则有1024中方式可以成功匹配。这就需要在DFA中有1024平方个状态来容纳这些不同长度的匹配路径。
（4）指数增长规模
被无限重复的内容在状态转移结构上的子结构并不需要有多个副本，因为它们可以通过转移行为回到这一子结构的初始状态；而被有限重复的内容，则需要确实的在状态转移图上出现特定数量的副本，且每一个副本的复杂度是等同的。
值得一提的是，DFA中指数级别的状态增长，常常和有限重复语法元素有关。

示例：.*1.{n}
假设n为4，其NFA状态转义图为：



其DFA图为：2的四次方，16个状态 呈现指数增长



DFA的状态最小化：
存在DFA状态数关于NFA状态数呈指数关系的理论值，这一现象被称为DFA的指数状态爆炸。

> 指数状态爆炸-这篇文章描述的非常清楚
> https://mp.weixin.qq.com/s?__biz=MzI1Nzk4NzExMg==&mid=2247484018&idx=1&sn=2208309a5ce675b7b7c916e8fc3ced23&chksm=ea0e4535dd79cc238fbeb16b0ec094b8dff7

有一个结论：**识别任何正则表达式的状态数最少的DFA是唯一确定的。** -----这里有疑问？？？？

> 确定型与非确定型： 在写正则表达式之前可以确定字符匹配顺序的是确定型，不确定字符匹配顺序的是非确定型。

> 有穷： 有限，在有限次数内得到结果

> 自动机： 自动机就是自动完成，在我们设置好匹配规则后由引擎自动完成，不需要人为干预

NFA执行原理：

NFA引擎的特点：
1.表达式主导： 按照表达式的一部分执行，如果不匹配换其他部分继续匹配，直到表达式匹配完成
2.会记录位置： 当执行到（d|b）时NFA引擎会记录字符的位置，然后选择其中一个先匹配。
3.单个字符可能检查多次： 当执行（d|b）时比较发现d不匹配，于是NFA引擎换表达式的另一个分支b,同时文本位置回退，重新匹配字符‘b’.
4.可实现反向引用等功能： 因为具有回退这一步，所以可以很容易的实现反向引用、环视等一些功能。

DFA执行原理：

DFA引擎的特点：
1.文本主导： 按照文本的顺序执行。
2.记录当前有效的可能： 当执行到（d|b）同时比较表达式中的d和b,所以会需要更多的内存。
3.每个字符只检查一次：这就提高了执行效率，而且速度与正则表达式无关。
4.不能使用反向引用等功能： 因为每个字符只检查一次，文本位置只记录当前比较值，所以不能使用反向引用、环视等一些功能。

pcre&hyperscan中实际的匹配算法

PCRE和PCRE2

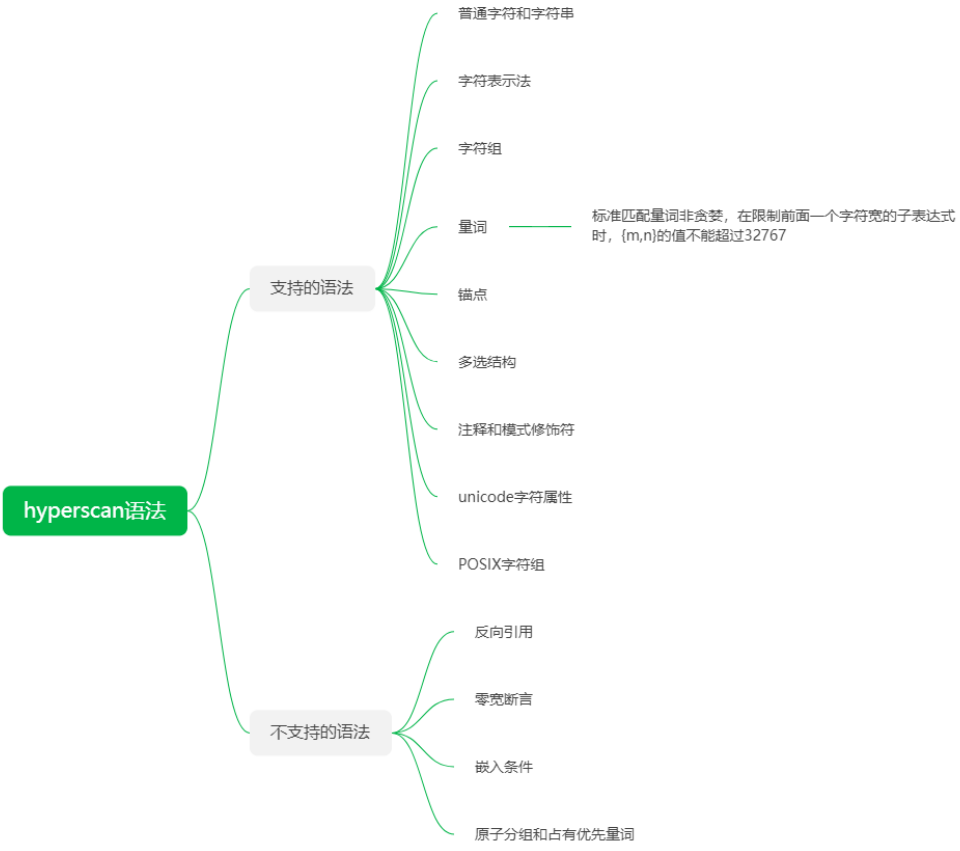
PCRE中的NFA在执行功能操作时：
1.在单个路径上执行深度优先搜索，直到找到NFA图上的不匹配节点
2.根据当前最后匹配节点搜索其他可选路径，如果搜索失败，回到前一节点再寻找其他可能的替代方案。
匹配过程中同时包含了NFA图和输入字符序列的反向遍历。规则中定义的贪婪特性会决定不同路径的遍历顺序，一旦匹配到一个叶子节点，就表示有一个匹配并直接停止运行。
PCRE中的DFA,允许存在多个活跃状态。它执行广度优先搜索算法，该算法在图上搜索针对输入字符序列所有可能的匹配。它保留所有活跃状态，并基于当前字符对每个活跃状态进行状态遍历。因此，此算法返回所有可能的匹配，而不是像基于NFA算法一样，只返回单一的匹配。它的确定在于让跟踪所有匹配路径边的十分困难，所以不支持捕获和反向引用。

> PCRE和PCRE2包含一个即时JIT编译选项，可以用来在编译期进行大量优化，并可显著提高运行期的匹配性能。但JIT编译优化需要额外的编译流程，而且只能应用于NFA中。

hyperscan

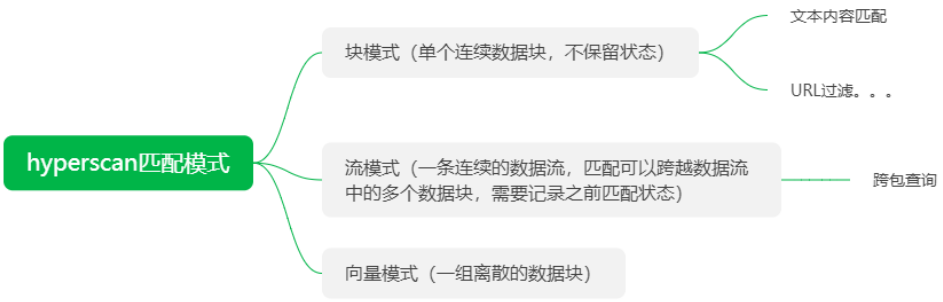
hyperscan是来自intel的正则表达式匹配库。它支持大规模多正则匹配，并利用intel单指令多数据技术加速匹配性能。

> hyperscan使用了经典自动机原理的NFA和DFA,其设计主要针对1：正则表达式编译相对不频繁，生成的数据库可用于匹配大量数据。

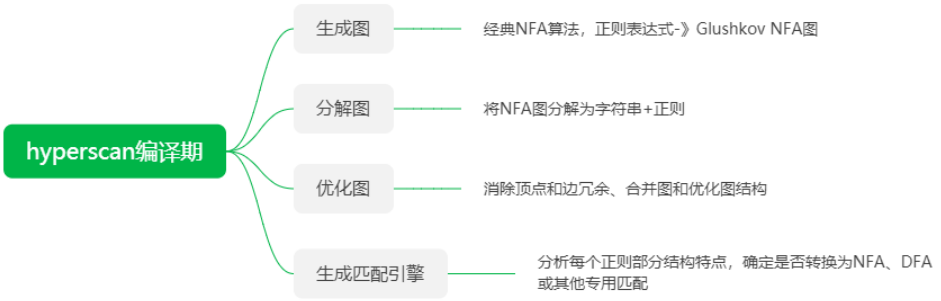


> 在流模式下当m和n的值比较大时，有界重复量词会占用相当多的内存来保存匹配状态。

hyperscan匹配模式



hyperscan编译期的设计



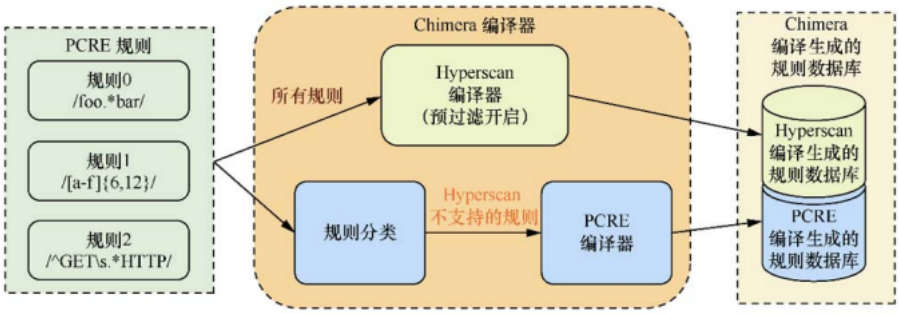
正则表达式建议

hyperscan的调优

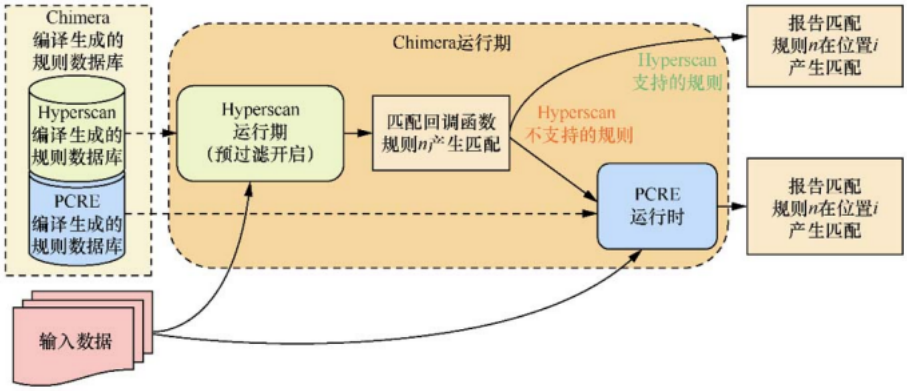
怎么在大量网络流量的情况下写出一个高效的正则表达式除了需要了解了正则表达式的匹配，还需要了解实际使用的引擎中正则表达式的匹配，引擎中可能会根据实际情况做调整。
正则表达式执行流程图

开始-》编译 -》执行
编译分为： 预编译-》编译

正则表达式由引擎执行。
Hyperscan基于混合有限自动机的设计，会使用NFA和DFA进行匹配。
（1）图的生成：应用经典的NFA生成算法，将原始正则表达式转换为Glushkov NFA图。
（2）图的分解：因为字符串匹配通常比正则表达式匹配快几个数量级，Hyperscan会将生成的NFA图分解为字符串部分和正则部分，并依赖字符串匹配的与过滤来尽可能避免运行性能更慢的正则部分。
（3）图的优化：可以对原始的NFA图和正则的图进行进一步优化，包括消除顶点和边的冗余、合并图和优化图的结构等。
（4）匹配引擎的生成：为所有字符串部分生成一个统一的字符串匹配引擎。
编译器流程图



运行期流程图



规则建议结论

在我们IPS实际书写规则时
1.规则串越特殊越好，因为网络场景中存在大量流量，ips引擎中一旦存在规则发生匹配，hyperscan就将调用用户提供的回调函数返回给上层应用程序。

> 回调函数和扫描函数在同一个上下文中，回调函数中过于复杂的处理会阻塞后续数据的扫描，所以用户需要尽量优化回调函数。此外，回调函数返回零时将继续扫描。一旦回调函数返回一个非零值，整个扫描过程会停止。

2.减少跨多个目标数据块的规则，因为hyperscan流模式为了维护跨多个目标数据块的匹配状态，需要为每条流分配一定的内存。

3.避免复杂的逻辑组合正则表达式，实际漏洞规则书写时，我们的匹配依赖于多个规则能否分别满足特定条件。

4.减少模糊匹配，.* 能拆分成多个字符串就拆分成多个字符串，因为字符串匹配速度是正则的几十倍。

5.减少数量限定匹配

6.减少分支嵌套

7.量词等量转换，正则表达式/a{3}/要比表达式/aaa/快一些

使用hyperscan的工具hsbench进行实际正则表达式举例：
30条正则表达式，其中前15条是实际匹配IPS漏洞的正则表达式

	Scantime	Throughput	Matches	Performance		Scantime	Throughput	Matches	Performance		Scantime	Throughput	Matches	Performance
0	0.029	55553.976	214	54110.428	11	0.029	55626.402	2	54588.406	22	0.050	31770.403	109	31612.505
1	0.030	53066.050	0	44734.252	12	0.076	21094.195	0	25032.094	23	0.053	30491.952	2117	30150.410
2	0.052	30676.388	9	41251.582	13	0.036	44515.023	0	45393.594	24	0.038	41597.297	0	32602.443
3	0.027	59843.195	0	59984.731	14	0.044	36414.895	0	45490.676	25	0.038	42231.460	0	43116.029
4	0.040	40152.072	0	52750.734	15	0.002	877412.642	3	991014.962	26	0.053	30386.352	0	29308.877
5	0.036	44954.924	0	43761.193	16	0.028	56620.887	0	58370.164	27	0.040	40276.439	19056	39109.416
6	0.031	51199.796	14	49999.237	17	0.030	52825.847	2	53221.050	28	0.035	45756.800	0	45252.860
7	0.034	47188.977	1	45943.794	18	0.035	45692.302	19	46311.531	29	0.053	30149.768	1	31171.424
8	0.041	39297.243	4	42418.269	19	0.052	30678.212	114119	30159.887	30	0.027	58706.069	1088	57907.581
9	0.052	30735.487	0	25190.563	20	0.032	49566.640	0	47947.712					
10	0.029	56149.047	1	54545.099	21	0.466	3434.213	129970	3332.145					

进行规则优化&测试结果：

0:/0M8R4KGxGuEAAAAAAAAAAAAAAAAAAAAPgADAP7\/CQAG/
Throughput 55553.976 Matches **214** performance **54110.428**
优化：0:/0M8R4KGxGuEA{21}PgADAP7\/CQAG/
Throughput 49151.823 Matches **214** performance **38563.857**

12：/Attachment_Times[\S\s]*?A{200,}/
Throughput 21094.195 Matches **0** performance **25032.094**
优化：12：/Attachment_Times[\S\s]{0,20}A{200}/
Throughput 3087.819 Matches **0**performance **3479.799**

15：/^(a*)b/
Throughput 877412.642 Matches **3** performance **991014.962**
优化：15：/^a{3,200}b/
Throughput 700716.125 Matches **3** performance **508815.162**

19：/Huck[a-zA-Z]+|Saw[a-zA-Z]+/ 21：/[a-q][^u-z]{13}x/ 27：/[a-zA-Z]+ing/
扫描时间以及匹配次数都比较高
优化措施：去掉不够特殊的特征对不确定性特征尽可能添加范围

参考链接&工具安装

安装hyperscan

```
1.下载源码hyperscan-5.0.0.zip
2.安装依赖boost （安装这个依赖时间稍微长一些）
  tar vxf boost_1_57_0.tar.gz
  cd boost_1_57_0
  ./bootstrap.sh
  ./b2 install
3.安装依赖ragel
  tar zvxf ragel-6.10.tar.gz
  cd ragel-6.10
  ./configure
  make && make install
4.编译hyperscan5.0
  unzip hyperscan-master.zip
  cd hyperscan-master
  mkdir build
  cd build
  cmake  -G  "Unix Makefiles" ..
  cmake  -DBUILD_STATIC_AND_SHARED=on -DCMAKE_BUILD_TYPE=Debug ../
  make && make install

5.运行 cd到bin目录下
./patbench pattern/content+pcre pcap/bps_8000_1204_s1port5_1.pcap
  建立pattern目录
  vim pattern文件：添加pcre 格式是 1：/pcre/ #数字：/pcre/
  建立pcap目录：放置测试包
```

参考:

```
《深入浅出hyperscan》

https://mp.weixin.qq.com/s/WoKOFqxCu2ktQByo_1SvTQ

https://mp.weixin.qq.com/s/tX4xbD_XOMkhP-SceObpPA

https://mp.weixin.qq.com/s/-6Dn95NKb4k1aqdC1iApBQ

https://mp.weixin.qq.com/s/H1EuT53ntbUte2Kj07i4Tw

https://mp.weixin.qq.com/s/ezu16Dg4f6_WLHXAkEgd6g
```