北京郵電大學
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

计算机应用编程实验一

大规模字符串检索

熊永平@网络技术研究院

ypxiong@bupt.edu.cn

周四15:30-17:30@3-117

2016.9.22

课程方式

- 总人数49人
  - 分为7组
  - 每组7人，讨论技术方案和调研
  - 每组分为若干实验单元，每单元2-3人，评分单元
- 讨论课
  - 每组派人做报告
  - 由组长安排
  - 组长名单：待定
- 助教
  - 分发课程材料

课程表

学期																													
年份																													
月份	八月	九月					十月					十一月				十二月							月			二月				
周次	〇	一	二	三	四	五	六	七	八	九	十	十一	十二	十三	十四	十五	十六	十七	十八	十九	二十	廿一	廿二	廿三	廿四	廿五	廿六
星期一	22	本科生新生报到	研究生上课	12	本科生上课	26	国庆节假期	10	17	24	31	7	14	21	28	5	12	19	26	2	9	16	23	30	6	13	20
星期二	23		6	13	20	27		11	18	25	1	8	15	22	29	6	13	20	27	3	10	17	24	31	7	14	21
星期三	24	本科生开学典礼	7	14	21	28		12	19	26	2	9	16	23	30	7	14	21	28	4	11	18	25	1	8	15	22
星期四	25	研究生新生报到	8	中秋节假期	22	29		13	20	27	3	10	17	24	1	8	15	22	29	5	12	19	26	2	9	16	23
星期五	26	研究生开学典礼			23	30			21	28	4		18	25	2	9	16	23	30	6		20	27	3	10	17	24
星期六	27	3			24	国庆节假期	8		22	29	5	12	19	26	3	10		24	31	7	14	21	春节	4	11	18	25
星期日	28	4		18	25		9		23	30	6	13		27	4	11	18		元旦	8	15	22	29	5	12	19	26

实验一

实验二

实验三

实验四

课程介绍

实验一讨论课

实验二讨论课

实验三讨论课

实验四讨论结束

实验一:海量字符串查找


1、实验任务
2、核心算法之Trie树
3、核心算法之BloomFilter
4、代码提交说明

实验背景

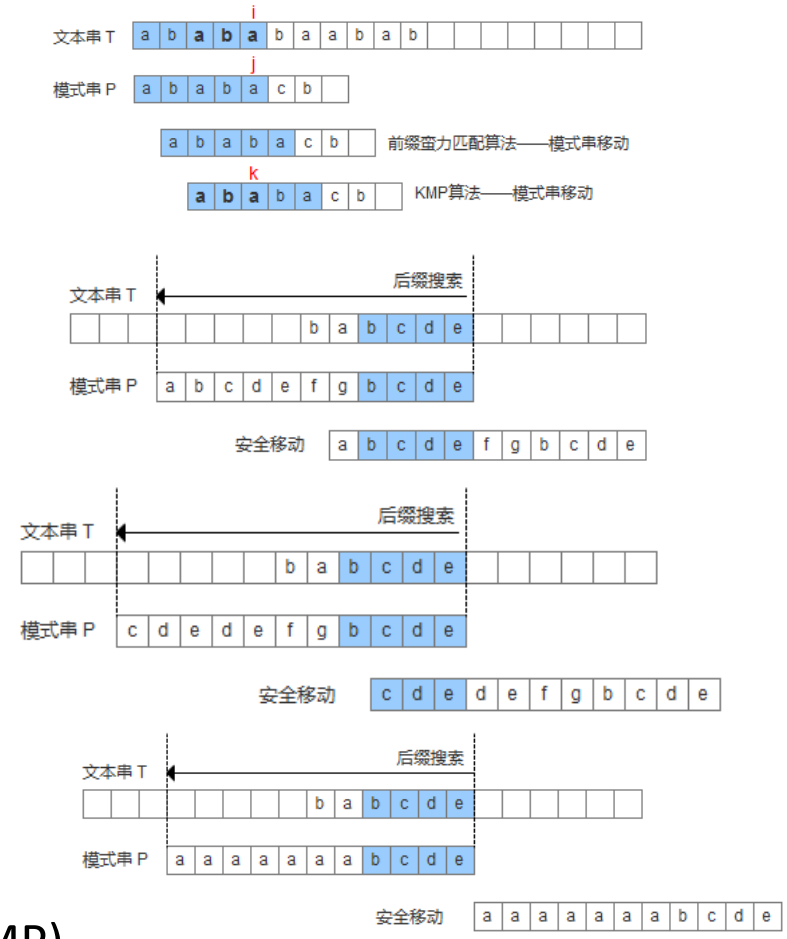
- 一些实际问题
  - 如何实现在搜索引擎输入框自动提示？
  - 一个查询串的重复度越高，说明查询它的用户越多，也就越热门。大型搜索引擎每小时有几十亿个查询请求，如何统计搜索引擎最热门前100个查询？
  - 在实现一个编辑器时，如何对输入的单词进行拼写检查？
  - GFW每次google中输入**词后，如何在**名单中查找并reset？
  - 搜索引擎的网络爬虫，每天要爬取几十亿网页，哪些URL是爬过的？
  - 收到一封邮件后，能否快速在几亿个垃圾邮件黑名单地址里快速判断发件人是否在黑名单里面？
  - 检测引擎中包含几千万条特征字符串规则，如何在10G网络流量环境下检测网络流中的恶意软件特征？

实验一：海量字符串查找

- 问题
  - 在给定的海量个数的字符串中查找特定的字符串
- 挑战
  - 实际需求
    - 几亿规模
  - 数据量大
    - 200,000,000量级
  - 外存便宜
    - 存储成本低
  - 内存不够大？
    - 200,000,000*40bytes=8000,000,000bytes=8G

根据以往的学习思路

- 字符串集合转换成一个大字符串
- 字符串匹配
  - Brute Force
    - Strstr()
  - KMP算法
    - 前缀匹配算法
  - Boyer-Moore
    - 后缀匹配算法
- 问题
  - 存储空间
- 复杂度
  - best case O(n/m)
  - worst case *still* O(nm)(BM)、O(n)(KMP)



1、实验任务

2、核心算法之Trie树

3、核心算法之BloomFilter

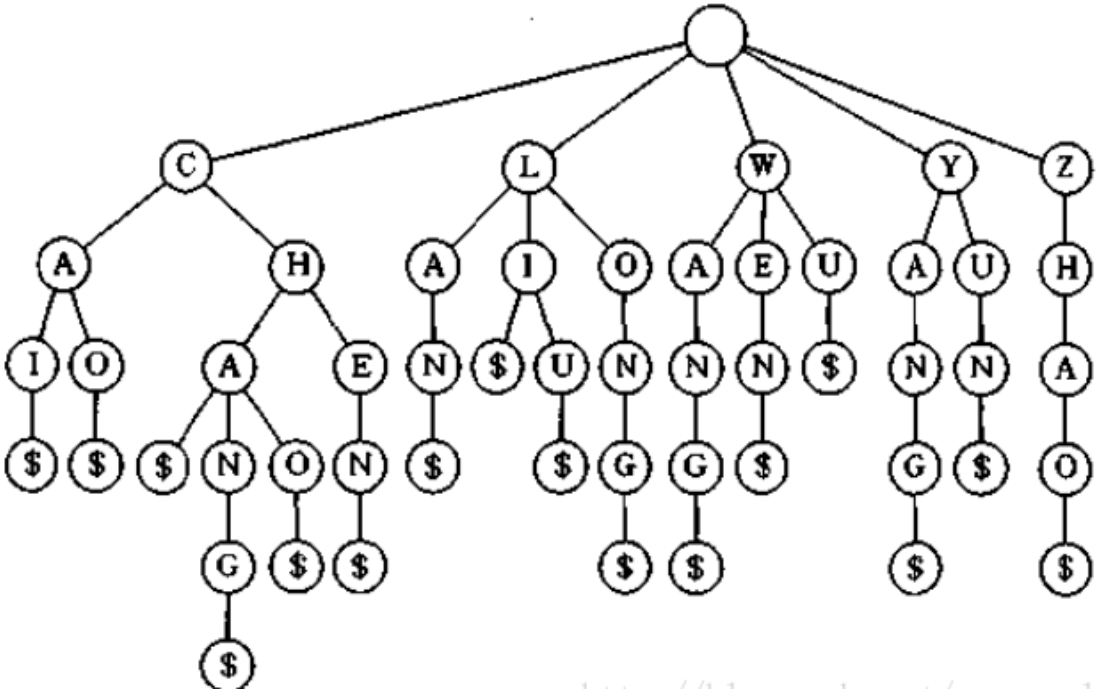
4、代码提交说明

Trie树

- 基于关键码分解的数据结构，叫作Trie结构（Trie树）
- 基于两个原则
  - 有一个固定的关键码集合
  - 对于结点的分层标记
- Trie树
  - 又称单词查找树、字典树，是一种树形结构，是一种用于快速检索的多叉树结构
- 典型应用
  - 统计和排序大量的字符串
  - 文本词频统计和文本检索
  - 优点：最大限度地减少无谓的字符串比较，查询效率比哈希表高。

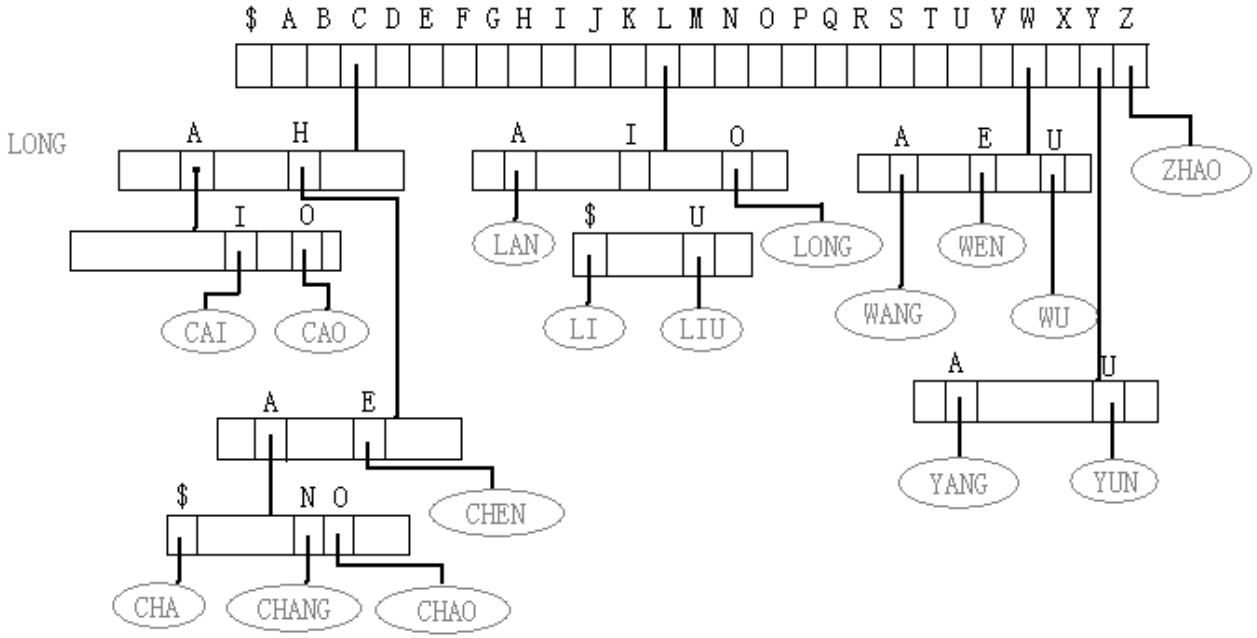
Trie结构示例

- 存储词典
  - { CAI、CAO、LI、LAN、CHA、CHANG、WEN、CHAO、YUN、YANG、LONG、WANG、ZHAO、LIU、WU、CHEN }
  - 树的高度为最长字符串长度



Trie的实现

- 数据结构实现
  - 分支结点：含有d个指针域和一个指示该结点中非空指针域的个数的整数域。
  - 分支结点所表示的字符是由其指向子树指针的索引位置决定的叶子结点：含有关键字域和指向记录的指针域。



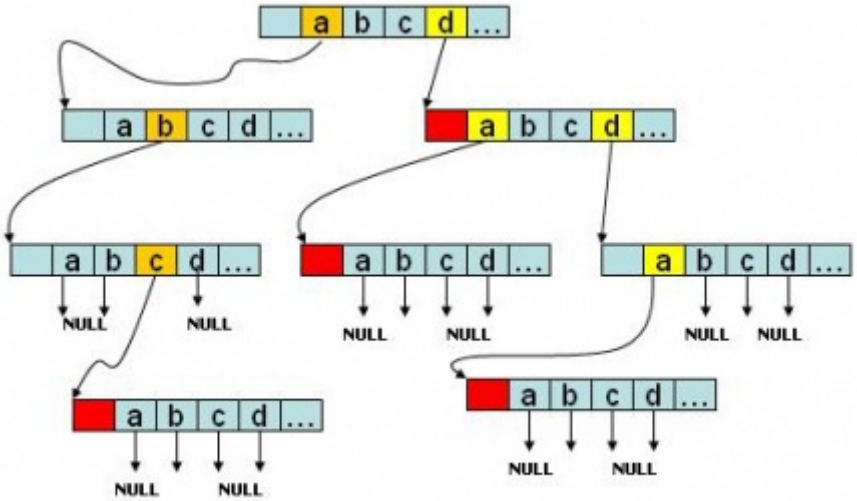

```
Typedef Struct TrieNode{
NodeKind kind ;
union {
struct {KeyType K; Record *infoptr}
lf ; //叶子结点
struct {TrieNode *ptr[27]; int num}
bh ; //分支结点} ;
} TrieNode,*TrieTree ;
```

介绍

- 不依赖于关键码的插入顺序
  - 树的深度受到关键码精度的影响
- 最坏的情况下，深度等于存储关键码所需要的位数
  - 例如，如果关键码是0到255之间的整数，关键码的精度就是8个二进制位。
  - 如果有两个关键码：10000010和10000011，它们的前面7位都是相同的
  - 所以直到第8次划分才能将这两个关键码分开
  - 这样的搜索树深度也为8，但这是最坏的情况
  - 与B+树一样，基于关键码空间分解的树结构，其内部结点仅作为占位符引导检索过程，数据纪录只存储在叶结点中

Trie查找

- 查找
  - 在Trie树上进行检索总是始于根结点。
  - 取得要查找关键词的第一个字母，并根据该字母选择对应的子树并转到该子树继续进行检索。
  - 在某个结点处相应的子树上，取得要查找关键词的第二个字母，并进一步选择对应的子树进行检索。
  - 键词的所有字母已被取出，则读取附在该结点上的信息，即完成查找
- 示例
  - trie树中保存了abc、d、da、dda四个单词



Trie树的插入

- 插入
  - 首先根据插入纪录的关键码找到需要插入的结点位置
  - 如果该结点是叶结点，那么就将为其分裂出两个子结点，分别存储这个纪录和以前的那个纪录
  - 如果是内部结点，则在那个分支上应该是空的，所以直接为该分支建立一个新的叶结点即可

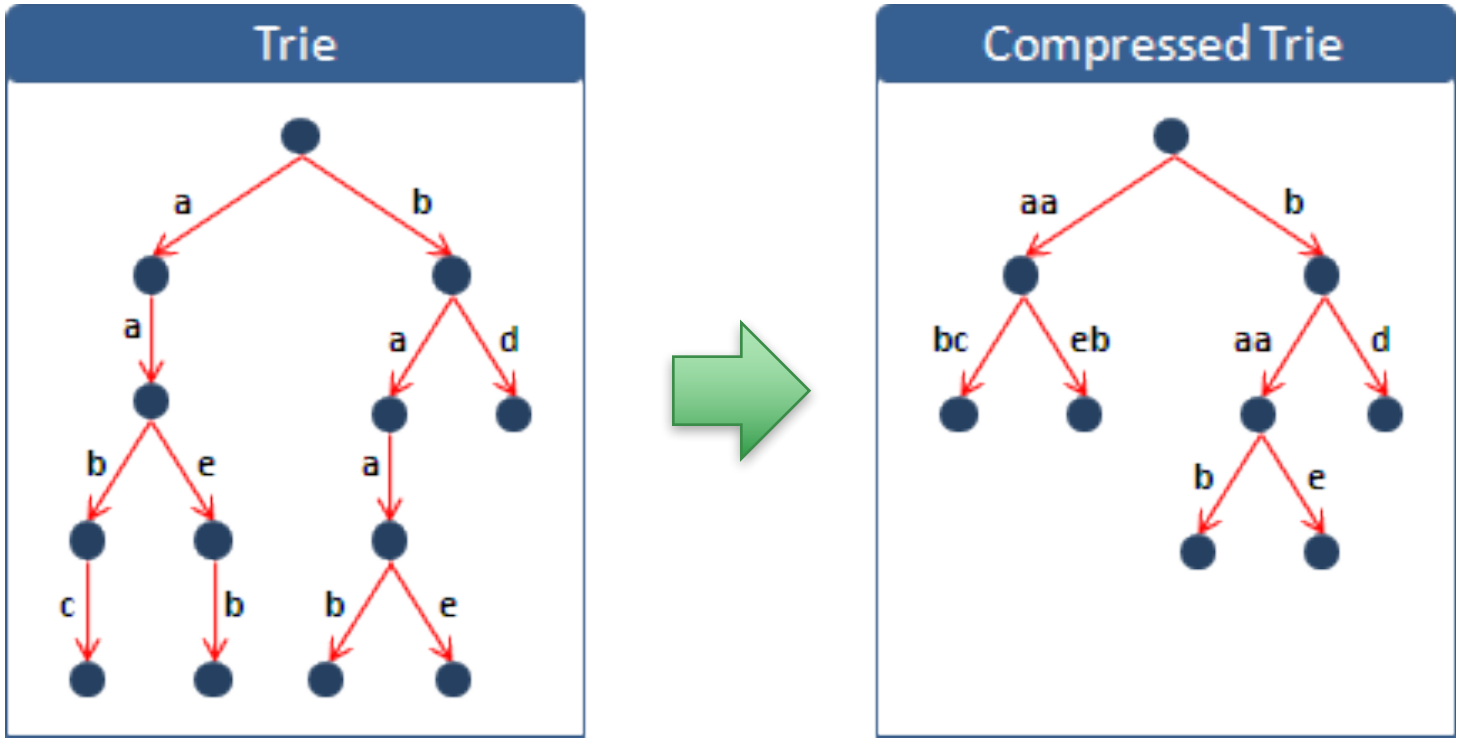
Trie查找效率分析

- 在trie树中查找一个关键字的时间和树中包含的结点数无关，而取决于组成关键字的字符数。（对比：二叉查找树的查找时间和树中的结点数有关$O(\log_2^n)$。）
- 如果要查找的关键字可以分解成字符序列且不是很长，利用trie树查找速度优于二叉查找树。
- 若关键字长度最大是5，则利用trie树，利用5次比较可以从$26^5=11881376$个可能的关键字中检索出指定的关键字。而利用二叉查找树至少要进行$\log_2 26^5=23.5$次比较。

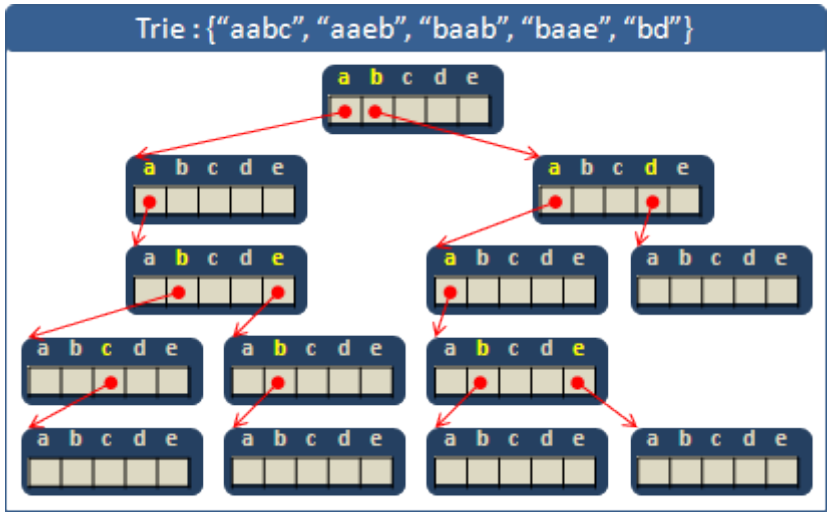
Trie树特性

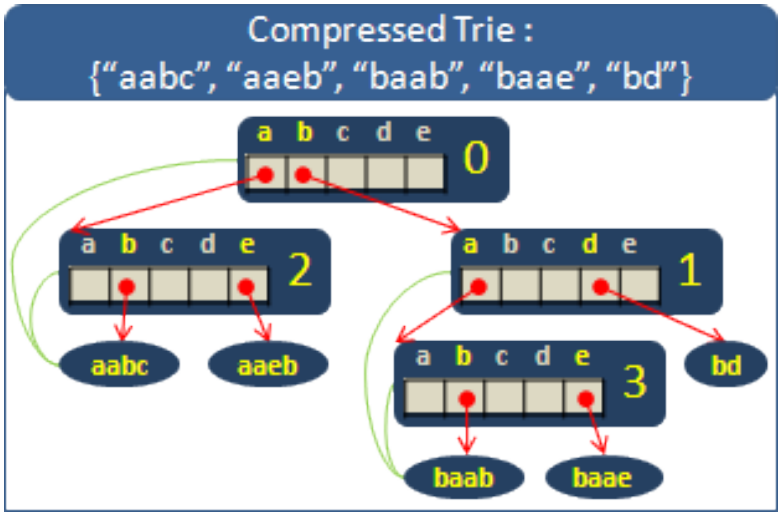
- 核心思想
  - 空间换时间
  - 利用字符串的公共前缀来降低查询时间的开销以达到提高效率的目的
- 优点
  - 查找效率高，与词表长度无关
    - Trie树的查找效率只与关键词长度有关
  - 索引的插入，合并速度快
    - 注意，直接遍历Trie树需要搜索大量的无效节点
    - 可以把数据存在一个数组中，Trie只保存指针
    - 这样合并时，只需要对数组进行遍历即可
- 缺点
  - 内存空间消耗大
    - 如果是完全m叉树，节点数指数级增长
    - 不可达上限： 词数 ×字符序列长度 ×字符集大小 ×指针长度
    - 例如：20000 × 6 × 256 ×4 ＝120M
  - 实现较复杂

Trie优化（1）--Compressed Trie



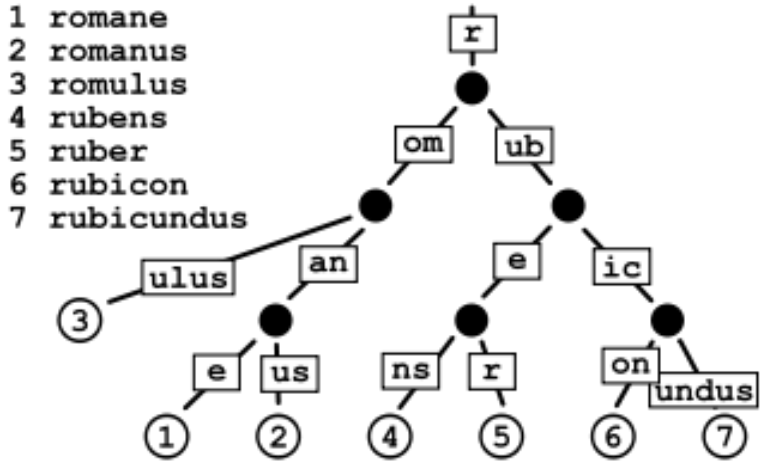
Radix Tree

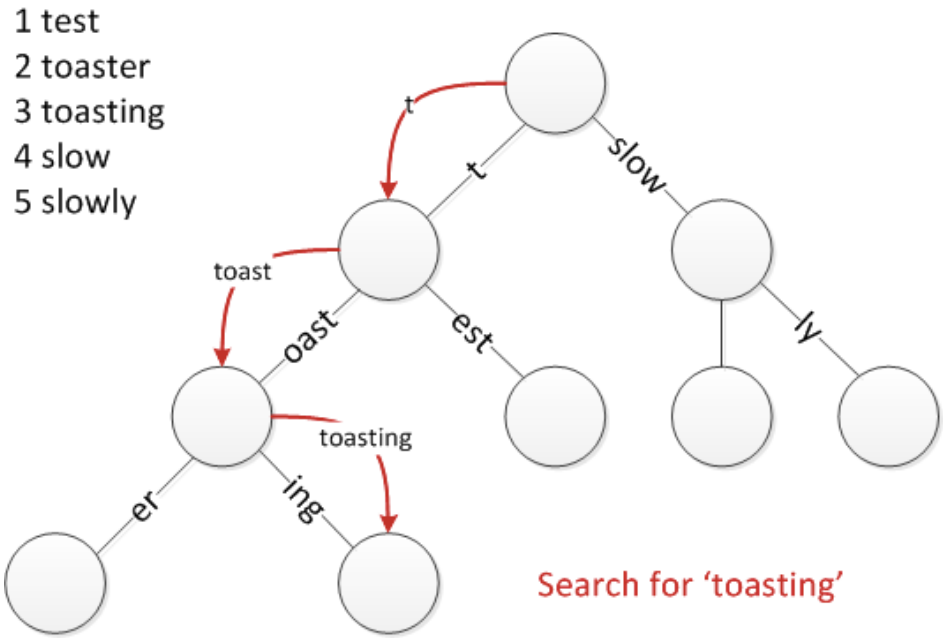




- 去掉没有分岔、连成直线的节点。
- 每个节点增加一个数字，纪录是第几个字符开始分岔。
- 去掉节点之后，字符串信息不完整，在树叶里储存完整字符串。
- 每个节点增加一个指针，纪录要参考哪一个叶子的字符串开头。

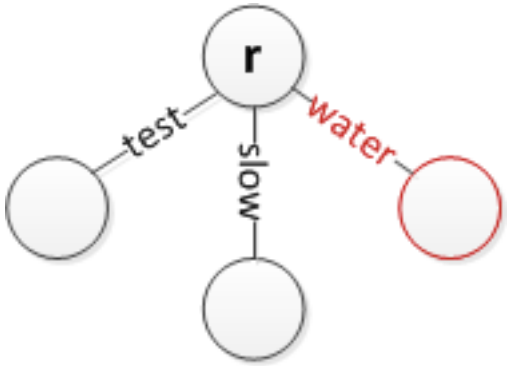
Radix Tree



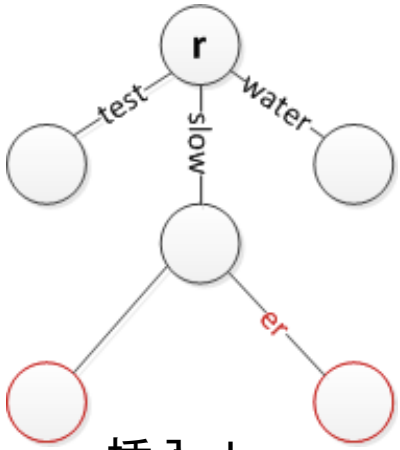


Radix Trie 查找

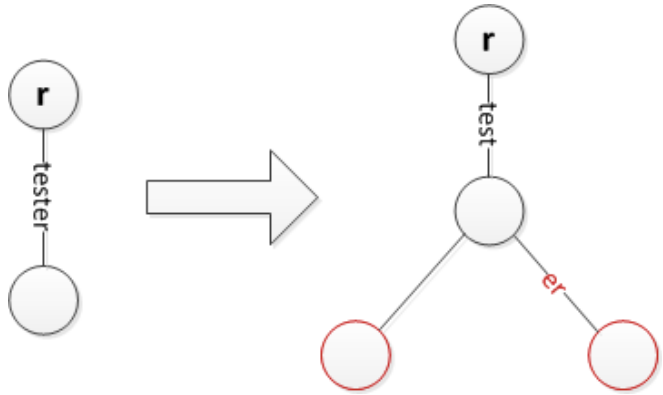
Radix Tree插入



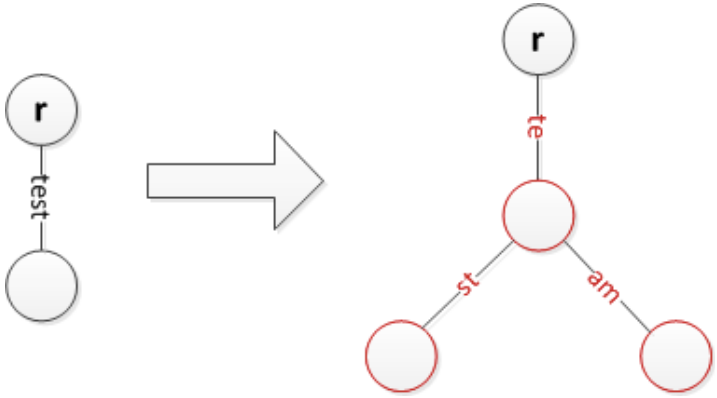
插入water



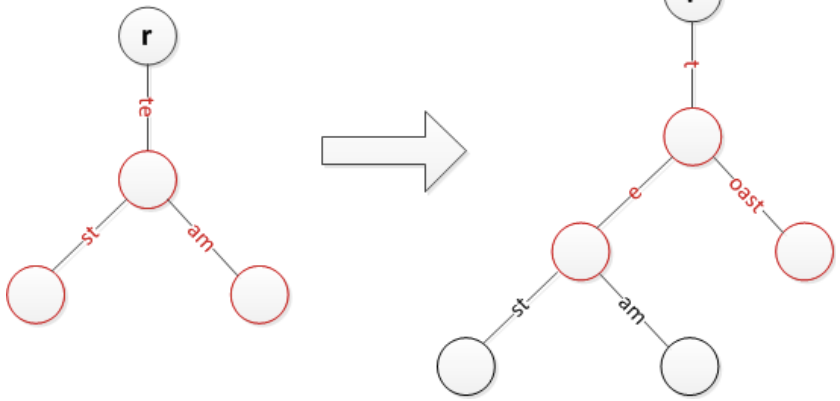
插入slower



插入tester



插入team

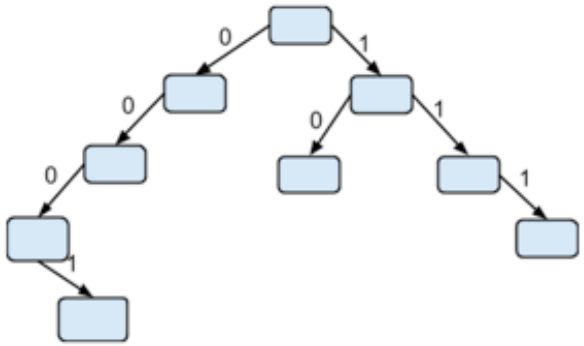


插入toast

Trie优化（2）– PAT Trie

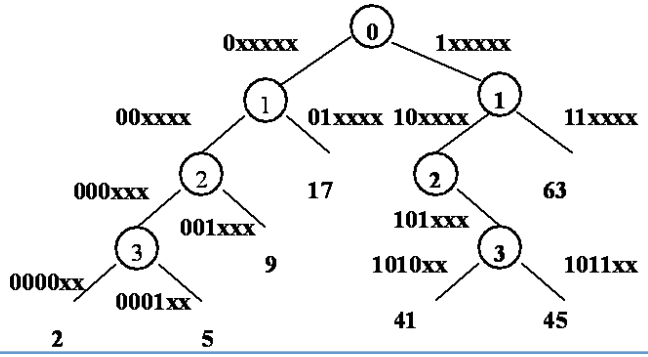
- Trie结构缺点
  - Trie结构显然也不是平衡的
  - 存取英文单词时，显然t子树下的分支比z子树下的分支多很多
  - 26个分支因子使得树的结构过于庞大，检索不便
- PATRICIA Trie（“Practical Algorithm To Retrieve Information Coded In Alphanumeric”）
  - 关键码二进制形式存储
  - 根据关键码每个二进制位的编码来划分
  - 是对整个关键码大小范围的划分

每个内部结点都代表一个位的比较，必然产生两个子结点，所以它是一个满二叉树，进行一次检索，最多只需要关键码位数次的比较即可。



PATRICIA应用举例

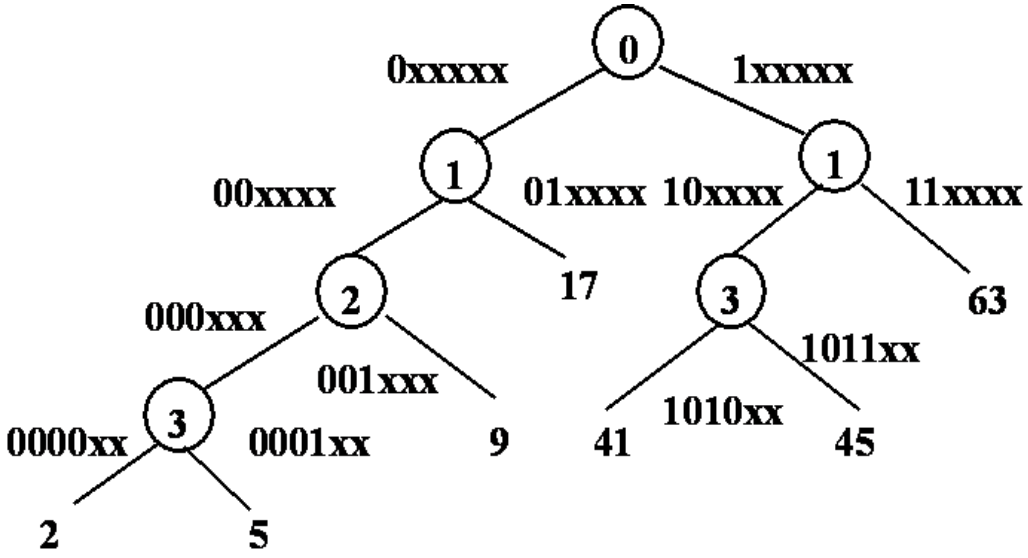
- 举例（2、5、9、17、41、45、63）
  - 因为最大的数是63，用6位二进制表示即可
  - 每个结点都有一个标号，表示它是比较第几位，然后根据那一位是0还是1来划分左右两个子树
  - 标号为2的结点的右子树一定是编码形式为xx1xxx，（x表示该位或0或1，标号为2说明比较第2位）
  - 在图中检索5的话，5的编码为000101
  - 首先我们比较第0位，从而进入左子树，然后在第1位仍然是0，还是进入左子树，在第2位还是0，仍进入左子树，第3位变成了1，从而进入右子树，就找到了位于叶结点的数字5



编码： 2：000010 5：000101 9：001001
17：010001 41：101001 45：101101 63：111111

PATRICIA压缩优化

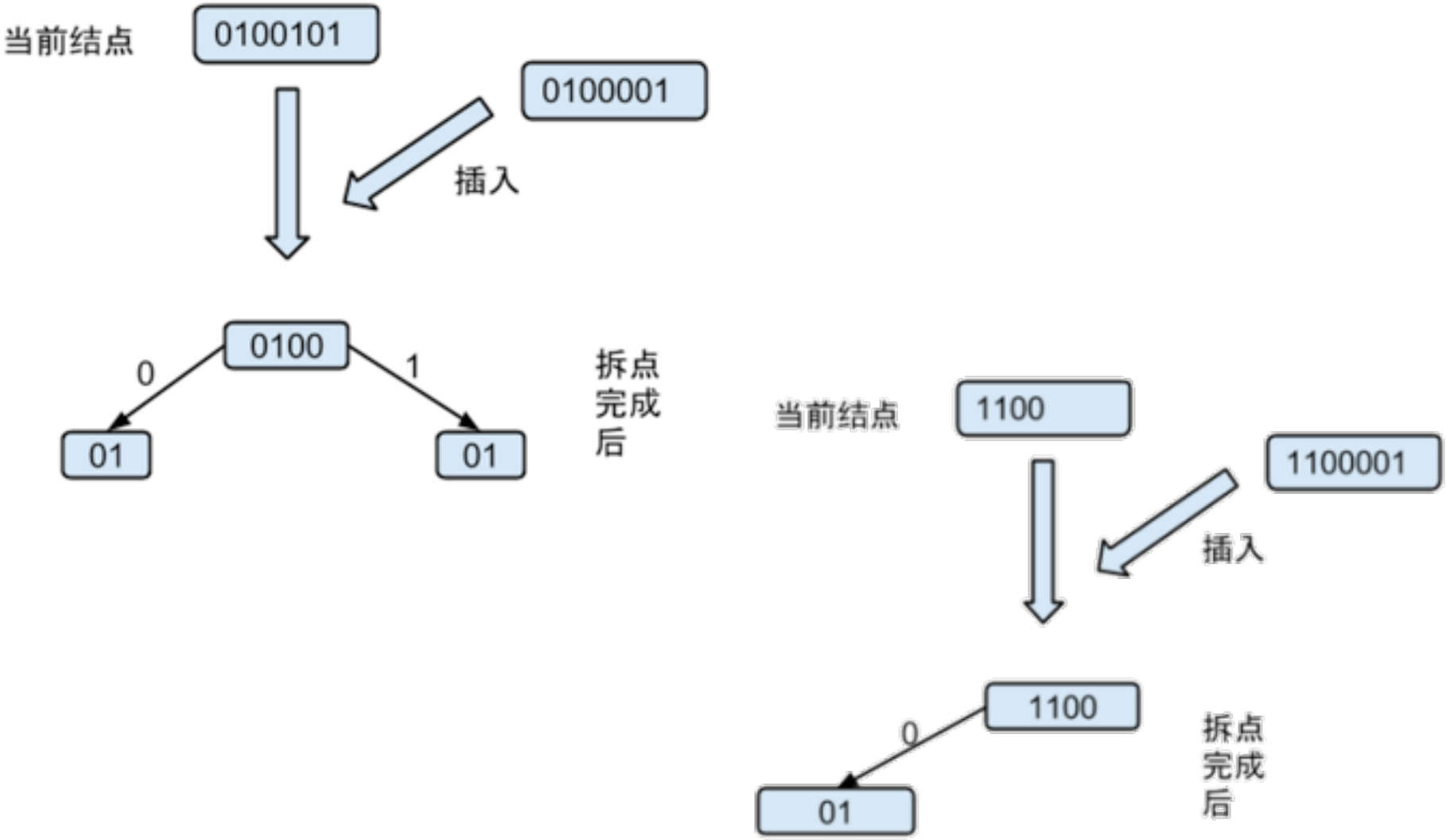
- 优化
  - 在区分2和5、41和45时，第3个二进制位的比较不能区别它们，可以将它省略，得到一棵更为简洁的树 。



编码：2：000010　5：000101　9：001001

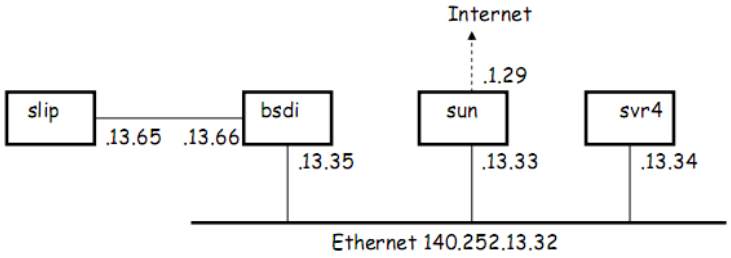
17：010001　41：101001　45：101101 63：111111

压缩Patricia Trie



PATRICIA 应用：路由表查找

- Example Net



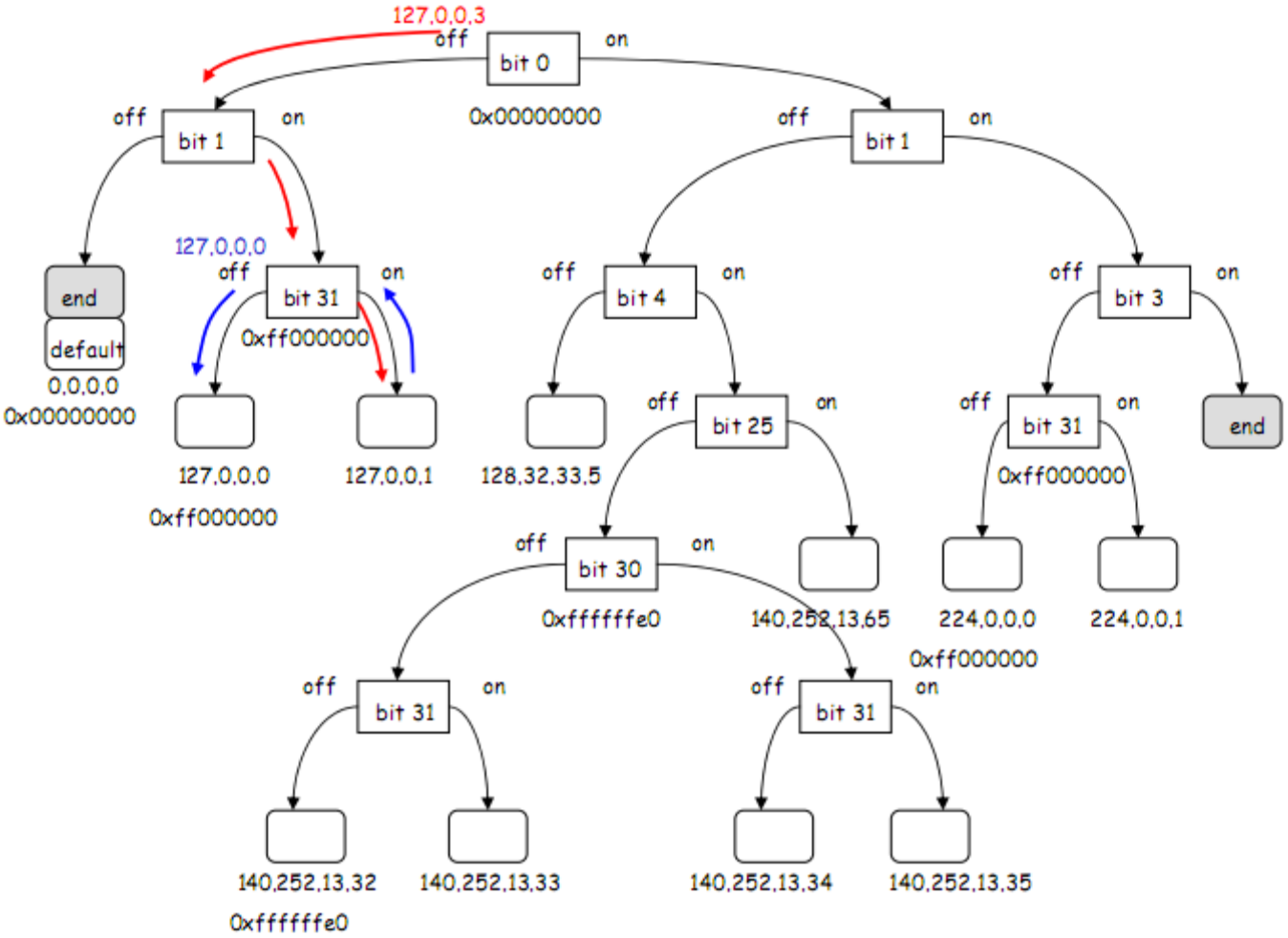
- Routing Table

Destination	Gateway	Flags	Ref	Use	Interface
default	140.252.13.33	UGS	0	3	le0
127	127.0.0.1	UGSR	0	2	lo0
127.0.0.1	127.0.0.1	UH	1	55	lo0
128.32.33.5	140.252.13.33	UGHS	2	16	le0
140.252.13.32	link#1	UC	0	0	le0
140.252.13.33	8:0:20:3:f6:42	UHL	11	55146	le0
140.252.13.34	0:0:c0:c2:9b:26	UHL	0	3	le0
140.252.13.35	0:0:c0:6f:2d:40	UHL	1	12	lo0
140.252.13.65	140.252.13.66	UH	0	41	sl0
224	link#1	UC	0	0	le0
224.0.0.1	link#1	UHL	0	5	le0

	32-bit IP address (bits 32-63)	
Bit:	0123 4567 8911 1111 1111 2222 2222 2233 01 2345 6789 0123 4 567 8901	
	0000 0000 0000 0000 0000 0000 0000 0000	0.0.0.0
	0111 1111 0000 0000 0000 0000 0000 0000	127.0.0.0
	0111 1111 0000 0000 0000 0000 0000 0001	127.0.0.1
	1000 0000 0010 0000 0010 0001 0000 0101	128.32.33.5
	1000 1100 1111 1100 0000 1101 0010 0000	140.252.13.32
	1000 1100 1111 1100 0000 1101 0010 0001	140.252.13.33
	1000 1100 1111 1100 0000 1101 0010 0010	140.252.13.34
	1000 1100 1111 1100 0000 1101 0010 0011	140.252.13.35
	1000 1100 1111 1100 0000 1101 0100 0001	140.252.13.65
	1110 0000 0000 0000 0000 0000 0000 0000	224.0.0.0
	1110 0000 0000 0000 0000 0000 0000 0001	224.0.0.1

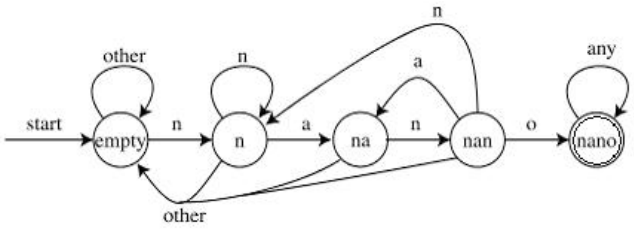
PATRICIA应用：路由表查找

- Example:127.0.0.3



Trie优化（3）--Trie 图

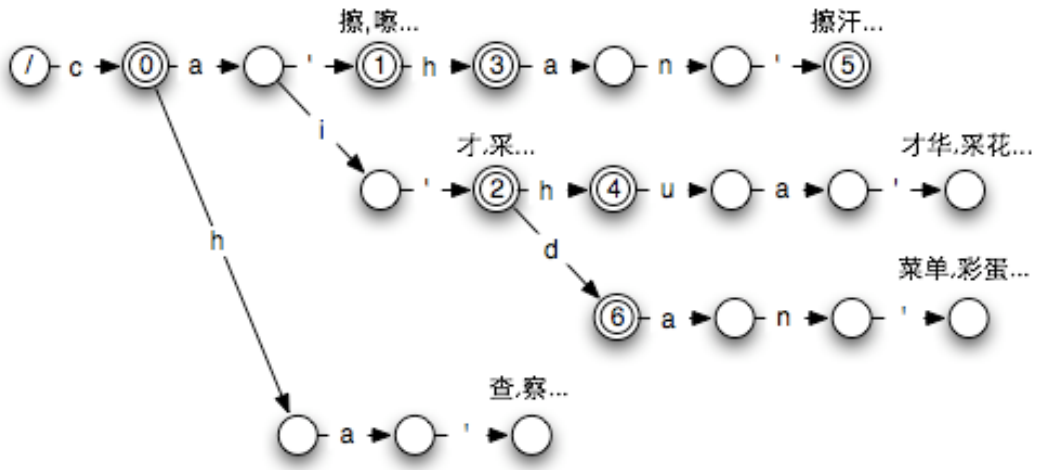
- 背景
  - 反病毒引擎一般都用字符串作为恶意软件特征，在大规模网络流量中如何判断是否出现恶意软件？
- 问题
  - 给一个很长很长的母串长度为n，然后给m个小模式串。
  - 求这m个模式串里边有多少个是母串的字串。
- Trie图
  - 一个确定性有限自动机DFA，由Trie树为基础构造
  - 每点都是一个状态，状态之间的转换用有向边来表示
  - 用母串作为DFA的输入，在DFA 上行走，走到终止节点，就意味着匹配了相应的模式串



Trie图实例—输入法

- 举例：
  - 音节表

string[] _spellMusicCode = new string[]{ “a”,“ai”, “an”, “ang”, “ao”, “ba”, “bai”, “ban”, “bang”, “bao”, “bei”, “ben”, “beng”, “bi”, “bian”, “biao”, “bie”, “bin”, “bing”, “bo”,“bu”,“ca”, “cai”, “can”, “cang”, “cao”, “ce”, “ceng”, “cha”, “chai”, “chan”, “chang”, “chao”, “che”, “chen”, “cheng”, “chi”, “chong”, “chou”, “chu”,“chuai”, “chuan”, “chuang”, “chui”, “chun”, “chuo”, “ci”, “cong”, “cou”, “cu”,……



本实验的Trie树应用

Ctrl	Dec	Hex	Char	Code
^@	0	00		NUL
^A	1	01	☺	SOH
^B	2	02	☻	STX
^C	3	03	♥	ETX
^D	4	04	♦	EOT
^E	5	05	♣	ENQ
^F	6	06	♠	ACK
^G	7	07	•	BEL
^H	8	08	◘	BS
^I	9	09	○	HT
^J	10	0A	◙	LF
^K	11	0B	♂	VT
^L	12	0C	♀	FF
^M	13	0D	♪	CR
^N	14	0E	♫	SO
^O	15	0F	☼	SI
^P	16	10	►	DLE
^Q	17	11	◄	DC1
^R	18	12	↕	DC2
^S	19	13	‼	DC3
^T	20	14	¶	DC4
^U	21	15	§	NAK
^V	22	16	▬	SYN
^W	23	17	↨	ETB
^X	24	18	↑	CAN
^Y	25	19	↓	EM
^Z	26	1A	→	SUB
^[	27	1B	←	ESC
^\	28	1C	∟	FS
^]	29	1D	↔	GS
^^	30	1E	▲	RS
^_	31	1F	▼	US

Dec	Hex	Char
32	20	SP
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Dec	Hex	Char
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

Dec	Hex	Char
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	⌂

Email格式：

- 开头为数字或字母
- 中间可含“_”、“-”、“.”3类字符
- 不区分大小写
- local part 为“@”前面的部分，最多64个字符
- domain part 为“@”后面的部分，最多255个字符
- 总长度不超过255+64+“@”=320字符

参考实现(Trie)

- 著名实现代码

- Linux Kernel Implementation, used for the page cache, among other things.
- GNU C++ Standard library has a trie implementation
- Java implementation of Concurrent Radix Tree, by Niall Gallagher
- C# implementation of a Radix Tree
- Practical Algorithm Template Library, a C++ library on PATRICIA tries (VC++ >=2003, GCC G++ 3.x), by Roman S. Klyujkov
- Patricia Trie C++ template class implementation, by Radu Gruian
- Haskell standard library implementation "based on big-endian patricia trees". Web-browsable source code.
- Patricia Trie implementation in Java, by Roger Kapsi and Sam Berlin
- Crit-bit trees forked from C code by Daniel J. Bernstein
- Patricia Trie implementation in C, in libcprops
- Patricia Trees : efficient sets and maps over integers in OCaml, by Jean-Christophe Filliâtre
- Radix DB (Patricia trie) implementation in C, by G. B. Versiani


1、实验任务
2、核心算法之Trie树
3、核心算法之BloomFilter
4、代码提交说明

Hash思想回顾

- 思想
  - 把任意长度的输入通过Hash算法，变换成固定长度的输出
    - 把一些不同长度的信息转化成杂乱的128/256位的编码
    - 代表算法：MD5, SHA
  - 特点
    - 一般是压缩映射，不可逆映射
    - 把一个大范围映射到一个小范围
- 使用领域
  - 信息安全基础设施实现加密、消息摘要等
  - 数字指纹：文档去重、内容检索
  - 内容安全：数据防泄漏

数据结构之Hash Table

- 思想
  - 利用线性表存储集合元素
  - 利用Hash函数计算元素对应的地址entry，并在对应的区域存储该元素
  - 实际查找通过先hash计算entry，再匹配元素是否相同（字符串匹配）
- 问题
  - 解决冲突
  - 加大表空间，
  - 查找复杂（拉
  - 准确率100%

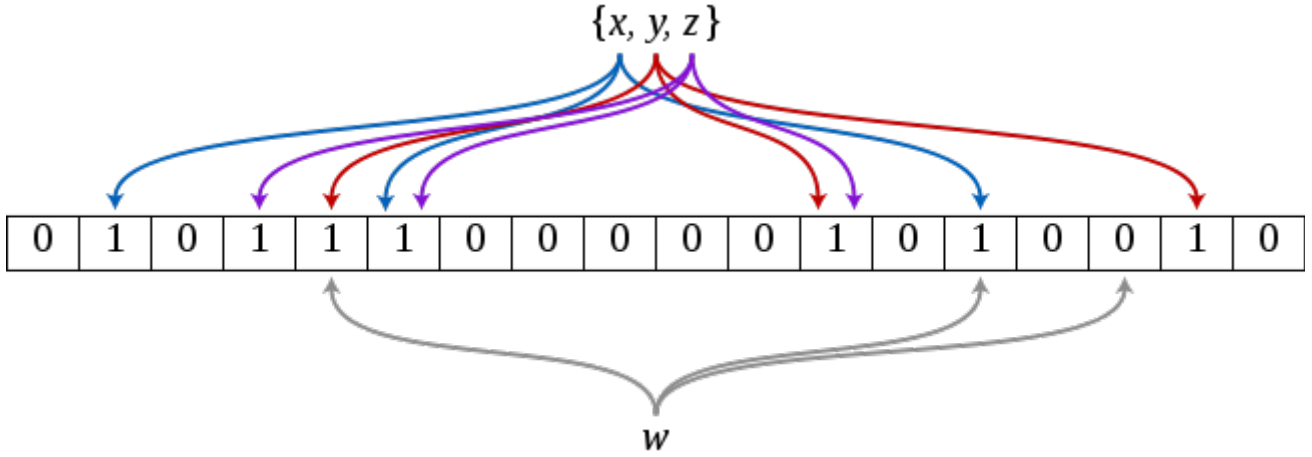
如何改进？

Bloom Filter概念

- 背景
  - 1970年Burton Bloom论文《Space/time trade-offs in hash coding with errors》中提出
  - 吴军《数学之美》中称之为布隆过滤器
- 概念
  - **一个很长的二进制向量和一系列随机映射函数**
  - 用于检索一个元素是否在一个集合中
  - 准确率换空间思想延伸
- 优点
  - 空间效率和查询时间都远超过一般的算法
- 缺点
  - 有一定的误识别率和删除困难

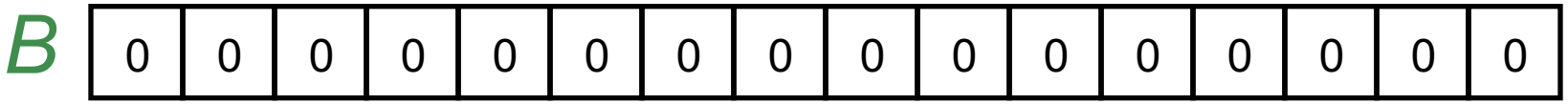
Bloom Filter构建

- 定义
  - 将n个元素集合S={x_1, x_2,…,x_n}
  - 一个包含m位的二进制位数组存储
  - K个**相互独立**的哈希函数映射到{1,…,m}的范围
  - S集合中的每个元素用k个hash函数映射到，{1,…,m}范围内，将相应的位置为1

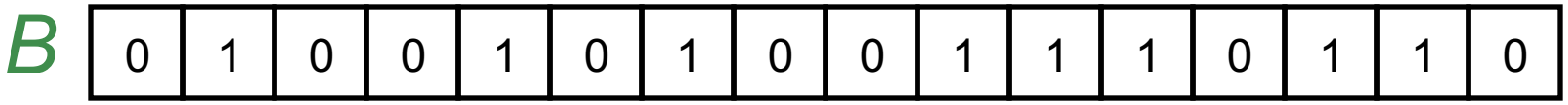


Bloom Filter原理

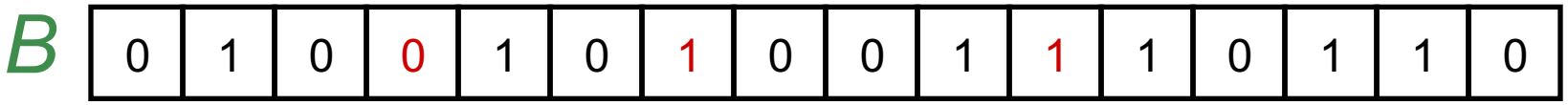
初始化时m 位数组置零



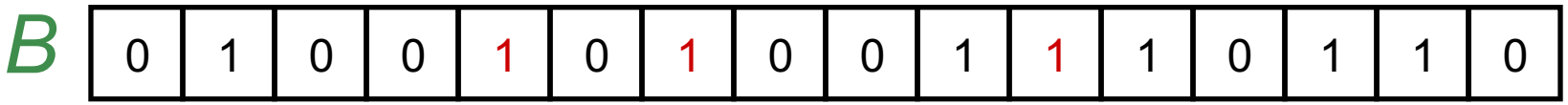
对集合中的每个元素x_j 分别进行k 次hash，If $H_i(x_j) = a$, set $B[a] = 1$.



要检测y是否在集合S，测试所有k 个B $[H_i(y)]$是否都为1.



可能出现false positive：即所有k个值都是1，但y不在集合S中



n items　　$m = cn$ bits　　k hash functions

错误率估计

- 初始位向量为0
- 插入一个元素后，被k个哈希函数（完全独立）映射到位向量后，某一位还是为0的概率是：$\left(1-\frac{1}{m}\right)^k$
- 集合$S=\{x_1, x_2,\ldots,x_n\}$的所有元素都插入到位向量后，某一位还是为0的概率就是

$$p' = \left(1-\frac{1}{m}\right)^{kn} \approx e^{-kn/m}. \qquad \lim_{x\to\infty}\left(1-\frac{1}{x}\right)^{-x} = e$$

- 令ρ为位向量中0的比例，则ρ的数学期望E(ρ)= p'。令$p = e^{-kn/m}$ 在ρ已知时要求的错误率（false positive rate）为：

$$(1-\rho)^k \approx (1-p')^k \approx (1-p)^k. \qquad f = \left(1-e^{-kn/m}\right)^k = (1-p)^k.$$

(1-ρ)为位数组中1的比例，$(1-\rho)^k$就表示k次哈希都刚好选中1的区域

M. Mitzenmacher已经证明位向量中0的比例非常集中地分布在它的数学期望值的附近

最优的哈希函数个数k

- 问题
  - Bloom Filter要用多个哈希函数将集合映射到位向量中，应该选择几个哈希函数才能使元素查询时的错误率降到最低？
  - 对 $f = \left(1 - e^{-kn/m}\right)^k = (1-p)^k.$
  - 令$g = k \ln(1 - e^{-kn/m})$，让g取到最小，f自然也取到最小
  - 将g写成

$$g = -\frac{m}{n}\ln(p)\ln(1-p),$$

- 结果
  - 很容易看出当$p = 1/2$，也就是$k = \ln 2 \cdot (m/n)$时，g取得最小值
  - 在这种情况下，最小错误率f等于$(1/2)^k \approx (0.6185)^{m/n}$
  - 另外，注意到p是位数组中某一位仍是0的概率，所以$p = 1/2$对应着位数组中0和1各一半
  - 换句话说，要想保持错误率低，最好让位数组有一半还空着

位向量大小m

- 问题
  - 在不超过一定错误率的情况下，Bloom Filter至少需要多少位才能表示全集中任意n个元素的集合。假设全集中共有u个元素，允许的最大错误率为ϵ，求位数组的位数m。
  - 一个确定的位向量可以表示的集合个数：$\binom{n+\epsilon(u-n)}{n}$
  - m位的位数组可以表示的集合个数：$2^m\binom{n+\epsilon(u-n)}{n}$
  - 全集中n各元素的集合个数：$\binom{u}{n}$
  - 要让m位的位数组能够表示所有n个元素的集合，必须有

$$2^m\binom{n+\epsilon(u-n)}{n} \geq \binom{u}{n}$$

  - 即：

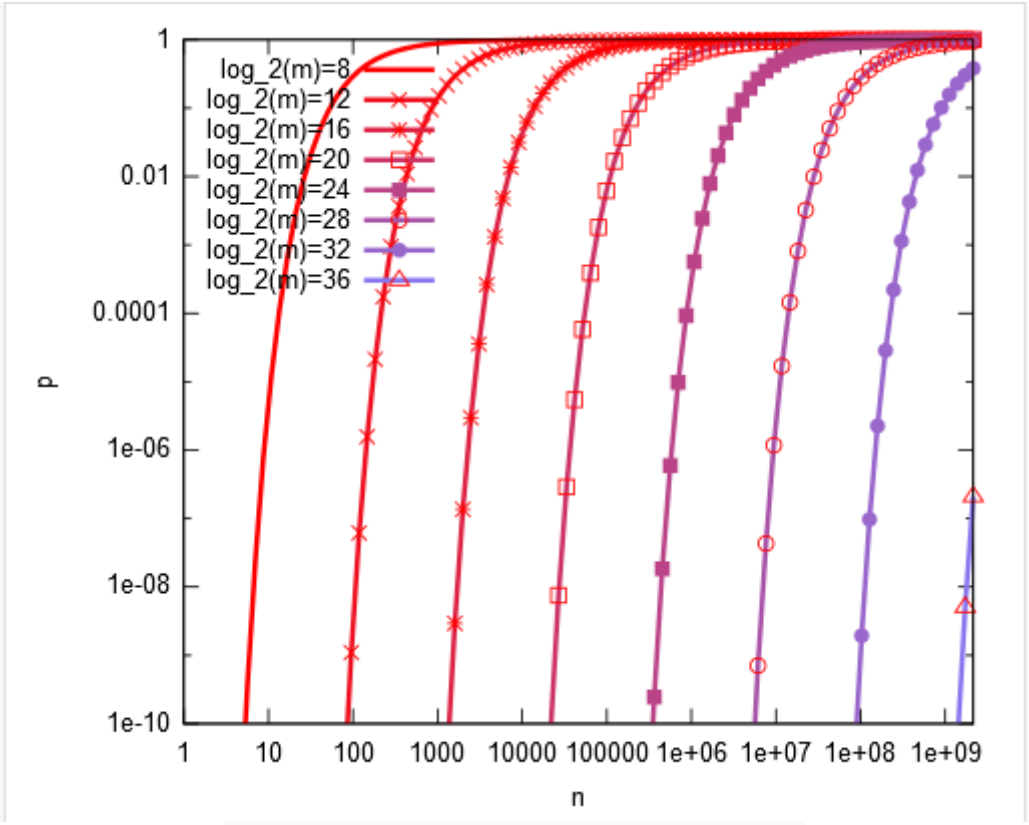
$$m \geq \log_2 \frac{\binom{u}{n}}{\binom{n+\epsilon(u-n)}{n}} \approx \log_2 \frac{\binom{u}{n}}{\binom{\epsilon u}{n}} \geq \log_2 \epsilon^{-n} = n\log_2(1/\epsilon).$$

  - 上页曾算出当$k = \ln 2 \cdot (m/n)$时错误率f最小，这时$f = (1/2)^k = (1/2)^{m\ln 2 / n}$。现在令$f \leq \epsilon$，可以推出，约大了1.44倍

$$m \geq n\frac{\log_2(1/\epsilon)}{\ln 2} = n\log_2 e \cdot \log_2(1/\epsilon).$$

函数分析

False positive rate随Hash函数个数和存储空间增加成指数级降低



$$k = (m/n)\ln 2$$

测试与参数选择

- 进行3组实验，每组取5个N
  - 取FP1 =0.01%，N=[50W, 100W, 300W, 500W,1000W]
  - **取FP 2=0.001%**，N=[50W, 100W, 300W, 500W,**1000W**]
  - 取FP3 =0.00001%，N=[50W, 100W, 300W, 500W,1000W]

N	(Vector size)m			内存			(Hash num) k			X		
	FP1	FP2	FP3	FP1	FP2	FP3	FP1	FP2	FP3	FP1	FP2	FP3
50W	958W	1198 W	1677W	1M	1M	1M	13	17	23	37091	3691	38
100W	1917 W	2396 W	3355W	2M	2M	3M	13	17	23	36958	3770	47
300W	5751 W	7188 W	10064 W	6M	8M	11M	13	17	23	36585	3689	38
500W	9585 W	11981 W	16773 W	11 M	14M	19M	13	17	23	36569	3701	45
1000W	19170W	23962W	33547 W	22 M	28M	39M	13	17	23	36533	3552	41

Hash算法选择

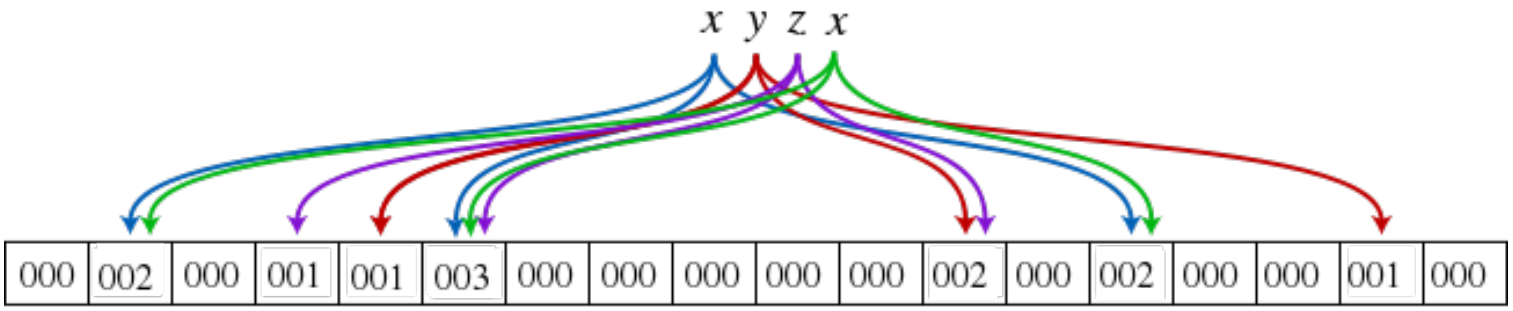
- K个hash越相互独立效果越好
- 常用的HASH算法
  - http://www.partow.net/programming/hashfunctions/
  - unsigned int RSHash (char* str, unsigned int len);
  - unsigned int JSHash (char* str, unsigned int len);
  - unsigned int PJWHash (char* str, unsigned int len);
  - unsigned int ELFHash (char* str, unsigned int len);
  - unsigned int BKDRHash(char* str, unsigned int len);
  - unsigned int SDBMHash(char* str, unsigned int len);
  - unsigned int DJBHash (char* str, unsigned int len);
  - unsigned int DEKHash (char* str, unsigned int len);
  - unsigned int BPHash (char* str, unsigned int len);
  - unsigned int FNVHash (char* str, unsigned int len);
  - unsigned int APHash (char* str, unsigned int len);

Hash算法不够

- Hash算法特性
  - 抗冲突性(collision-resistant)，即在统计上无法产生2个散列值相同的预映射。
  - 映射分布均匀性和差分分布均匀性
    - 散列结果中，为 0 的 bit 和为 1 的 bit ，其总数应该大致相等；
- 超级HASH算法
  - The requirement of designing k different independent hash functions can be prohibitive for large k. For a good hash function with a wide output, there should be little if any correlation between different bit-fields of such a hash, so this type of hash can be used to generate multiple "different" hash functions by slicing its output into multiple bit fields.
  - MurmurHash
  - 官网：https://sites.google.com/site/murmurhash/

扩展：计数型Bloom Filter

- 标准BF
  - 没法删除元素
  - m长数组，每unit用1位表示，只能表示[0, 1]
- 如果每个unit用多位表示，如3位，则可以表示[0,1,...,7]



扩展：压缩Bloom Filter

- 由于标准Bloom Filter在f最小时，任意位为0的概率为1/2，所以，为了更好的在网络上传输Bloom Filter，可以对Bloom Filter进行压缩，能得到约69.3%

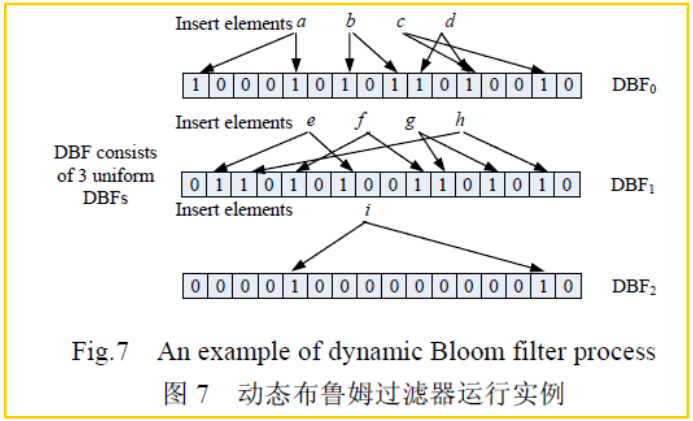


Fig.7 An example of dynamic Bloom filter process

图 7 动态布鲁姆过滤器运行实例


1、实验任务
2、核心算法之Trie树
3、核心算法之BloomFilter
4、代码提交说明

实验输入和输出

- 输入数据文件
  - 总共2000万个字符串，每个字符串一行

```
113247  timothy154473y4002@yahoo.com
113248  timothy154473y5@hotpop.com
113249  timothy154473y5@yahoo.com
113250  timothy154473y741@yahoo.com
113251  timothy15447fds@imailbox.com
113252  timothy15447fqds@flashmail.com
113253  timothy15447sdf4796@yahoo.com
113254  timothy15447sdf@imailbox.com
113255  timothy15447sdfq@imailbox.com
113256  timothy15447sdfqq5017@yahoo.com
113257  timothy15447sq3413@yahoo.com
113258  timothy15447y365y@hotpop.com
```

  - 待检测数据1000个字符串，每个字符串一行
- 输出要求
  - 每行输出一个1000行结果的文件，每行一个yes/no，表示1000个email是否在2000万字符串中

实验说明

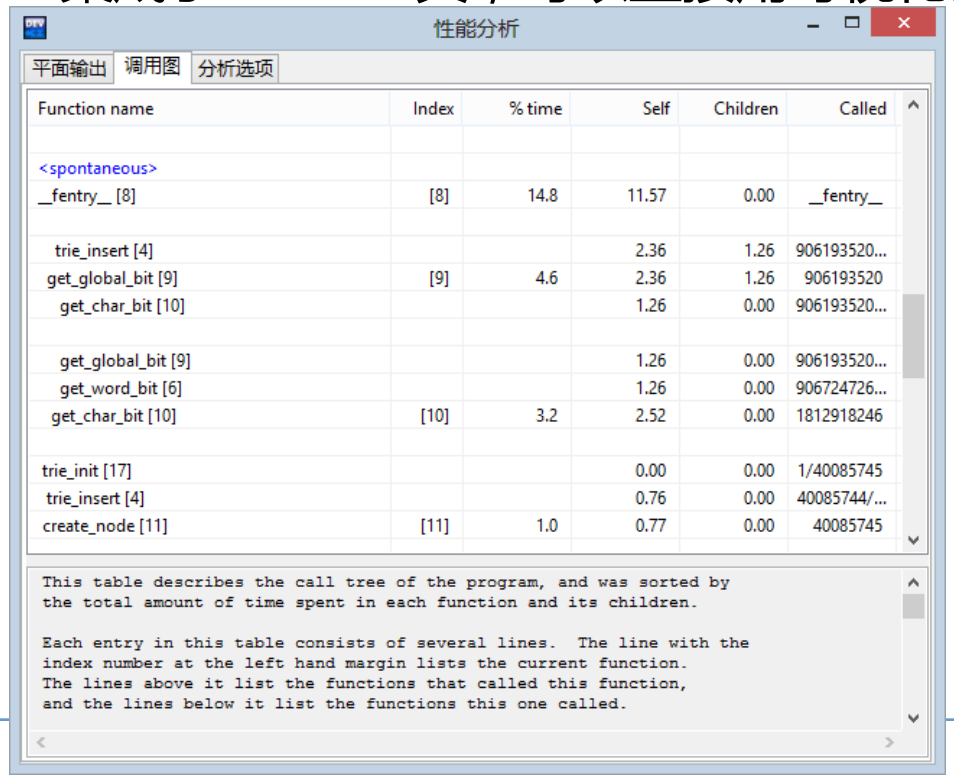
- 代码文件
  - 代码文件
    - main.c bloom.c bloom.h trie.c trie.h hash.c hash.h
    - Makefile编译规则文件
    - Readme说明程序的编译和使用方法
  - 命令行格式
    - ./strsearch strlist.txt checklist.dat checkedresult.dat
  - 输入文件
    - strlist.dat，2000万个源字符串
    - checklist.dat，1000个待检测的字符串
  - 输出文件（要求程序生成）
    - checkresult.dat
- 文档
  - 实验XXXXX设计文档.doc

编程实验环境

- 代码和数据服务器
  - 见邮件
- 操作系统
  - 推荐Ubuntu Linux操作系统（虚拟机或cygwin）
- 编辑器
  - Source Insight或Dev C++
- 编译和调试
  - GCC/G++
  - gdb
  - Makefile

性能优化工具

- 性能瓶颈分析
  - 可以采用gcc编译工具包**gprofile**工具来动态分析程序的性能瓶颈
  - 可以看到程序的每个函数的调用次数以及占用CPU的时间比例
  - Dev-CPP集成了Profile工具，可以直接用可视化界面查看



几个性能优化的方法

- 函数内联
  - 把一些大量调用的方法内联，添加inline关键字

```
inline
ull get_word_bit(char * word, ull index){
    return get_char_bit(word[index / BITS_USED], index % BITS_USED);
}
```

- 减少malloc次数
  - 内存预分配
  - tips：tcmalloc (多线程) / Jemalloc
    - 比glibc的malloc速度快2倍以上
    - Thread-Caching Malloc by Google


THE END