

REL 工具文档

作者：2011 级硕士成立

日期：2012 年 4 月 15 日

摘要

本文档设计了一种记录抽取语言（Record Extract Language，REL），并实现了一个能够解析该语言的 Linux 实用程序 rel，用于从符合同一种格式的若干记录文件中抽取出满足指定条件的记录集合。

1 入门

1.1 引言

应用程序通常需要存储大量数据，一种可能的方案是“平面文件”。平面文件独立于任何特定应用程序格式，通常就是文本文件。平面文件的主要缺点是文本处理效率较低，不过在主流计算能力下这已经不是大问题了。

另一个不足之处是缺乏像 SQL 那样的结构化查询方式，常使对文本记录的查询陷于复杂的细节。而现成的文本处理工具又各自存在着不足，很难找到一个工具“一统天下”。比如，尽管 AWK 的记录处理功能非常强大，但处理多文件的交并集问题就代码繁复、效率低下。同时，我们也可以看到，组合 sort、uniq 等简单 shell 命令，有时却反而可以轻松解决这类问题。

实际上，每种工具都有其长处和短处，取长补短才能发挥最大威力。但这又带来一个新问题：多项工具如何组合？REL 语言正是为此而生，它允许用户将基本集合操作（如交集、并集、差集）和用户自定义操作组合成一类查询表达式，并将此表达式转换成一系列对 shell 命令（包括 sort、uniq、gawk 等）的调用，来实现复杂的组合查询操作（但其表述方式是简洁的）。因此，REL 语言既隐藏了通用操作的内在复杂性，又不失灵活性和可扩展性。

1.2 用法

(1) 编译。进入源码目录，运行：

make clean; make

若需要保留生成的 shell 脚本以及运算的中间结果：

vim rel.c

取消以下语句的注释：

#define PRESERVED

并重新编译。

(2) 一个 REL 程序是指一个集合表达式(set expression)。rel 实用程序接受一个集合表达式并进行解析，生成一个包含具体命令的 shell 脚本，执行该脚本产生最终的记录集。rel 调用格式：

./rel “<set_expression>”

注意双引号必不可少，因为表达式中部分字符在 shell 中有特殊含义，可能会导致 rel 程序收到的表达式不正确。

一个集合表达式由标识符（代表输入文件或 AWK 脚本）、数字（作为 AWK 脚本的参数）及操作符等元素组成。表达式构成规则参见 2.2 节（文法）。

下一节将讨论若干实例，基本涵盖了 REL 语言的全部用法。

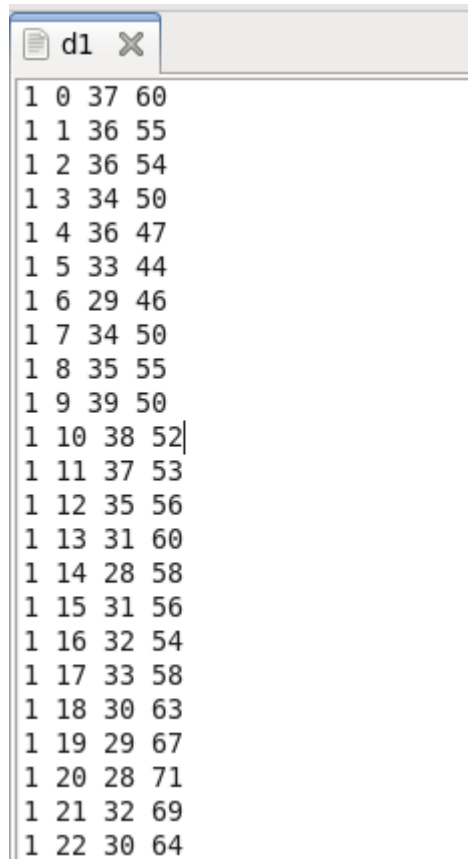
1.3 实例

背景描述：假设现在有 N 辆小车，通过轨迹跟踪系统捕获了每一辆小车随时间变化的位置信息。捕获的信息按记录(record)组织，该场景中一条记录是一个四元组：

$$r = (id, t, x, y)$$

该记录的含义是：标识为 id 的小车在 t 时刻位于 (x, y) 位置。

现假定标识为 id 的小车记录均存放在名为 $d<id>$ 的文件中，如文件 $d1$ 代表标识为 1 的小车记录文件，其片段如下。每一行即是一条记录，每条记录的 4 个字段（由空格分隔）分别代表 id 、 t 、 x 与 y 。



1	0	37	60
1	1	36	55
1	2	36	54
1	3	34	50
1	4	36	47
1	5	33	44
1	6	29	46
1	7	34	50
1	8	35	55
1	9	39	50
1	10	38	52
1	11	37	53
1	12	35	56
1	13	31	60
1	14	28	58
1	15	31	56
1	16	32	54
1	17	33	58
1	18	30	63
1	19	29	67
1	20	28	71
1	21	32	69
1	22	30	64

本例中，小车记录文件与 `rel` 实用程序位于同一目录下。在该目录中还有预先编写好的两个 AWK 脚本：`after` 与 `in`，可作为原语供 `rel` 调用。

(1) 提取标识为 1 的小车在时刻 10 以后的所有记录。

`./rel "d1.after(10)"`

该表达式的含义为：把参数 10 传递给 AWK 脚本 `after`，并将它应用于文件 `d1`。点操作符是为了给人以面向对象编程的错觉，实际上这里的含义的确与之类似，更符合人的直觉。

结果如图所示：可以看到 `t` 分量是从 11 开始的。注意，`rel` 实用程序假定记录文件 `d1` 和 AWK 脚本 `after` 位于程序启动目录下。如果实际文件并不存在，那么 `rel` 实用程序不会报错，因为它只是生成一个 `shell` 脚本，而不关心文件是否存在，但在执行该脚本时会报错。

```
root@localhost:~/rel
文件(E) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[root@localhost rel]# ./rel "d1.after(10)"
1 11 37 53
1 12 35 56
1 13 31 60
1 14 28 58
1 15 31 56
1 16 32 54
1 17 33 58
1 18 30 63
1 19 29 67
1 20 28 71
1 21 32 69
1 22 30 64
1 23 27 63
1 24 29 58
1 25 25 58
1 26 28 56
1 27 26 61
1 28 27 60
1 29 26 59
1 30 27 54
1 31 27 54
1 32 29 58
1 33 25 54
```

(2) 提取标识为 1 的小车位于圆心为(30, 60)、半径为 5 的圆内的所有记录。

`./rel "d1.in(30,60,5)"`

结果如图所示：可以看到返回的每一条记录均符合要求。

```
[root@localhost rel]# ./rel "d1.in(30,60,5)"
1 13 31 60
1 14 28 58
1 15 31 56
1 17 33 58
1 18 30 63
1 22 30 64
1 23 27 63
1 24 29 58
1 26 28 56
1 27 26 61
1 28 27 60
1 29 26 59
1 32 29 58
1 36 29 58
1 38 33 59
1 56 26 58
[root@localhost rel]#
```

- (3) 提取标识为 1 的小车，在时刻 20 以后，且位于圆心为(30, 60)、半径为 5 的圆内的所有记录。初步的想法是：先求出时刻 20 以后的记录集合，再求出位于该圆内的记录集合，最后求两个集合的交集（用乘号表示）。

*`./rel "d1.after(20)*d1.in(30,60,5)"`*

结果如图所示。



```
root@localhost:~/rel
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[root@localhost rel]# ./rel "d1.after(20)*d1.in(30,60,5)"
1 22 30 64
1 23 27 63
1 24 29 58
1 26 28 56
1 27 26 61
1 28 27 60
1 29 26 59
1 32 29 58
1 36 29 58
1 38 33 59
1 56 26 58
[root@localhost rel]#
```

但这种方法效率较低，需要遍历 d1 文件两次。设计 REL 语言时，提供了一种更高效的方式，这种链式方式在面向对象编程中也很常见。但在后面的例子中会看到，这种方式并不总是最好的。

`./rel "d1.after(20).in(30,60,5)"`

可见结果是一样的，但只需遍历 d1 文件一次即可。




```
root@localhost:~/rel
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[root@localhost rel]# ./rel "d1.after(20).in(30,60,5)"
1 22 30 64
1 23 27 63
1 24 29 58
1 26 28 56
1 27 26 61
1 28 27 60
1 29 26 59
1 32 29 58
1 36 29 58
1 38 33 59
1 56 26 58
[root@localhost rel]#
```

- (4) 提取标识为 1 的小车在时刻 10 至时刻 20 之间的所有记录。我们知道时刻 20 以后的记录集合是包含于时刻 10 以后的记录集合的，因此需要求一次差集（用减号表示）。

`./rel "d1.after(10)-d1.after(20)"`

结果如图所示。



```
root@localhost:~/rel
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[root@localhost rel]# ./rel "d1.after(10)-d1.after(20)"
1 11 37 53
1 12 35 56
1 13 31 60
1 14 28 58
1 15 31 56
1 16 32 54
1 17 33 58
1 18 30 63
1 19 29 67
1 20 28 71
[root@localhost rel]#
```

在实际应用中，需要注意时间区间端点的包含性！

- (5) 提取标识为 1 或 2 的小车在时刻 10 至时刻 20 之间的所有记录。注意此时加入了第二辆小车，需要使用并集操作（用加号表示）。

`./rel "d1.after(10)-d1.after(20)+d2.after(10)-d2.after(20)"`

结果如图所示。为后续处理方便，已按时间排序，因此看起来两辆小车的数据是交错的。

```
root@localhost:~/rel
文件(E) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[root@localhost rel]# ./rel "d1.after(10)-d1.after(20)+d2.after(10)-d2.after(20)"
"
1 11 37 53
2 11 76 12
1 12 35 56
2 12 75 15
1 13 31 60
2 13 78 19
1 14 28 58
2 14 82 20
1 15 31 56
2 15 84 17
1 16 32 54
2 16 82 22
1 17 33 58
2 17 87 18
1 18 30 63
2 18 87 14
1 19 29 67
2 19 82 13
1 20 28 71
2 20 80 17
```

- (6) 提取标识为 1 的小车，在时刻 10 至时刻 30 之间，位于圆心为(30, 60)、半径为 5 的圆内，或者位于圆心为(25,55)、半径为 5 的圆内的所有记录。这是最后一个入门实例，也是最复杂的一个，但万变不离其宗。

`./rel "(d1.after(10)-d1.after(30))*(d1.in(30,60,5)+d1.in(25,55,5))"`

结果如图所示。

```
root@localhost:~/rel
文件(E) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[root@localhost rel]# ./rel "(d1.after(10)-d1.after(30))*(d1.in(30,60,5)+d1.in(25,55,5))"
1 13 31 60
1 14 28 58
1 15 31 56
1 17 33 58
1 18 30 63
1 22 30 64
1 23 27 63
1 24 29 58
1 25 25 58
1 26 28 56
1 27 26 61
1 28 27 60
1 29 26 59
1 30 27 54
```

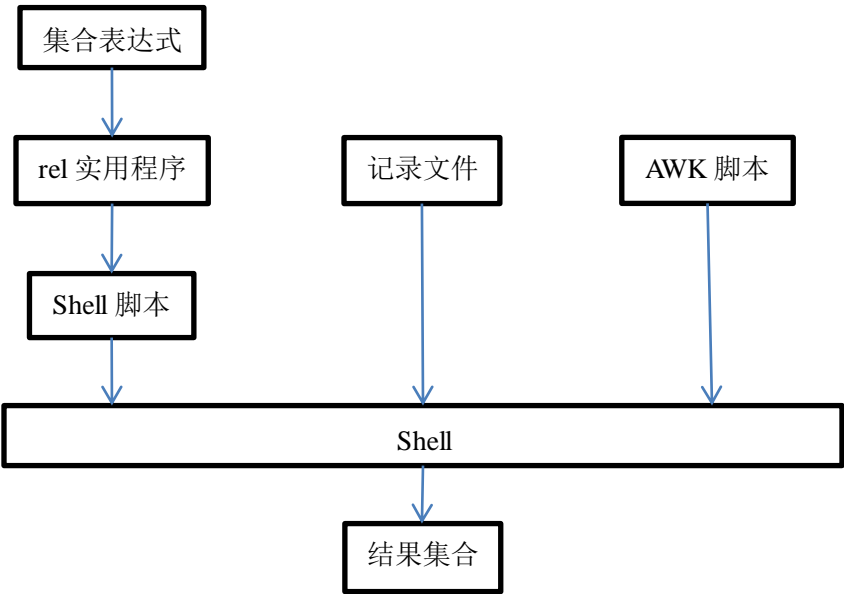
注意到，这里对差集和并集操作分别加了括号，其含义不言而喻。令乘号优先级高于加号和减号，同样是为了符合人的直觉（但实际数学里三种集合操作的优先级是相同的，注意区分！）。

另外这个表达式当然可以写成如下形式，但明显复杂了不少。可见交集操作并不总是适合用链式的点操作代替，使用哪种方式要具体情况具体分析。

“(d1.after(10)-d1.after(30)).in(30,60,5)+(d1.after(10)-d1.after(30)).in(25,55,5)”

2 原理

2.1 架构



2.2 文法

(1) Flex 词法

记号	正则表达式
<i>D</i>	<i>[0-9]</i>
<i>L</i>	<i>[_a-zA-Z]</i>
<i>IDENTIFIER</i>	<i>{L}({L} {D})*</i>
<i>NUMBER</i>	<i>[+-]?{D}+</i>
<i>NUMBER</i>	<i>[+-]?{D}+"."{D}+</i>

(2) Bison 文法

set_expression

: intersect_expression

/ set_expression '+' intersect_expression

/ set_expression '-' intersect_expression

intersect_expression

: postfix_expression

/ intersect_expression '' postfix_expression*

postfix_expression

: primary_expression

/ postfix_expression '.' IDENTIFIER '(' ')'

/ postfix_expression '.' IDENTIFIER '(' argument_list ')'

primary_expression

: IDENTIFIER

/ '(' set_expression ')'

argument_list

: NUMBER

/ argument_list ',' NUMBER

2.3 脚本生成

rel 实用程序输出的结果是包含具体命令的脚本，比如针对如下集合表达式生成的 shell 脚本如图所示（要查看生成的脚本请定义 PRESERVED 宏）。

```
./rel "(d1.after(10)-d1.after(30))*(d1.in(30,60,5)+d1.in(25,55,5))"
```

```
#!/bin/bash
# script extracting records from:
# "(d1.after(10)-d1.after(30))*(d1.in(30,60,5)+d1.in(25,55,5))"

# T0 = d1.after
gawk -f after -v v1=10.00 d1 > T0

# T1 = d1.after
gawk -f after -v v1=30.00 d1 > T1

# T2 = T0 minus T1
sort T0 T1 T1 | uniq -u > T2

# T3 = d1.in
gawk -f in -v v3=5.00 -v v2=60.00 -v v1=30.00 d1 > T3

# T4 = d1.in
gawk -f in -v v3=5.00 -v v2=55.00 -v v1=25.00 d1 > T4

# T5 = T3 union T4
sort T3 T4 | uniq > T5

# T6 = T2 intersect T5
sort T2 T5 | uniq -d > T6

# print result!|
sort -n -k2 T6
```

生成的脚本名为 `extract.sh`，默认为 Bash 生成。该脚本包含了若干条逻辑相关的命令，正如传统编译器为高级语言代码所生成的汇编指令，两者有一定的相似性。

相似性主要表现在它们都将一个复杂的表达式分解成若干个简单的操作按序执行，中间的操作会不断生成若干中间结果（图中的 T_n 都是中间结果），直到产生最终的结果（图中是 T_6 ）。

集合表达式中的每一个操作都需要被转换为相应的 shell 命令，对应关系如下：

操作	shell 命令
$A \text{ 交 } B$	<code>sort A B uniq -d</code>
$A \text{ 并 } B$	<code>sort A B uniq</code>
$A \text{ 减 } B$	<code>sort A B B uniq -u</code>
$ID1.ID2(\dots)$	<code>gawk -f ID2 -v ... ID1</code>

何时生成 shell 命令？答案是当从分析栈中归约出一个非终结符时，但并非每次归约都要生成，比如把 *primary_expression* 归约为 *postfix_expression* 时，只需要传递前者产生的中间结果文件名就可以了，不需要产生新的 shell 命令，否则会导致重复。

3 可扩展性

本语言设计时就充分考虑到了可扩展性，主要表现在如下两方面：

(1) 与文本格式无关。求集合的交并差与记录格式无关，只需将记录当作完整的字符串

来处理即可。而针对具体记录、字段的处理，则只依赖于 AWK 脚本，因此文本格式的改变只会导致 AWK 脚本的改变，甚至很少会涉及集合表达式的改变。而 rel 实用程序则完全不需作任何改动。

- (2) 功能可扩展。现在只提供了 *after* 和 *in* 两种操作，可处理的问题有限。但增加新的操作非常容易，只需增加一个相应的 AWK 脚本即可。这使得 rel 实用程序的应用范围变得十分广泛。

4 总结

简而言之，利用 REL 语言以及 rel 实用程序，一个文本查询问题等价于设计若干 AWK 脚本以及一个集合表达式。设计 AWK 脚本不宜过于复杂，应尽可能简洁、通用（AWK 语言的教程参见：<http://www.ibm.com/developerworks/cn/education/aix/au-gawk/index.html>）。而设计集合表达式的时候，不仅要考虑表达式的正确性，更要考虑效率问题，选择合适的操作。