awk编程

- 编者：陈艮龙
- 开始时间：20080310
- 还有很多没有记录，以后增加

awk编程

awk是一种优良的文本处理工具。它不仅是 Linux 中也是任何环境中现有的功能最强大的数据处理发动机之一。这种编程及数据操作语言（其名称得自于它的创始人 Alfred Aho 、Peter Weinberger 和 Brian Kernighan 姓氏的首个字母）的最大功能取决于一个人所拥有的知识。 AWK 提供了极其强大的功能：可以进行样式装入、流控制、数学运算符、进程控制语句甚至于内置的变量和函数。它具备了一个完整的语言所应具有的几乎所有精美特性。实际上 AWK 的确拥有自己的语言： AWK 程序设计语言，三位创建者已将它正式定义为“样式扫描和处理语言”。它允许您创建简短的程序，这些程序读取输入文件、为数据排序、处理数据、对输入执行计算以及生成报表，还有无数其他的功能。

awk编程

- 你可能对UNIX比较熟悉，但你可能对awk很陌生，这一点也不奇怪，的确，与其优秀的功能相比，awk还远没达到它应有的知名度。awk是什么？与其它 大多数UNIX命令不同的是，从名字上看，我们不可能知道awk的功能：它既不是具有独立意义的英文单词，也不是几个相关单词的缩写。事实上，awk是三 个人名的缩写，他们是：Aho、(Peter)Weinberg和(Brain)Kernighan。正是这三个人创造了awk---一个优秀的样式扫描与处理工具。

awk编程

- 最简单地说，AWK 是一种用于处理文本的编程语言工具。AWK 在很多方面类似 于 shell 编程语言，尽管 AWK 具有完全属于其本身的语法。它的设计思想来源于 SNOBOL4 、sed 、Marc Rochkind设计的有效性语言、语言工具 yacc 和 lex ，当然还从 C 语言中获取了一些优秀的思想。在最初创造 AWK 时，其目的是用于文本处理，并且这种语言的基础是，只要在输入数据中有模式匹配，就执行一系列指令。该实用工具扫描文件中的每一行，查找与命令行中所给定内容相匹配的模式。如果发现匹配内容，则进行下一个编程步骤。如果找不到匹配内容，则继续处理下一行。

awk编程

- 尽管操作可能会很复杂，但命令的语法始终是：
- awk '{pattern + action}' {filenames}
- 其中 pattern 表示 AWK 在数据中查找的内容，而 action 是在找到匹配内容时所执行的一系列命令。花括号 ({}) 不需要在程序中始终出现，但它们用于根据特定的模式对一系列指令进行分组。

awk编程

AWK的功能是什么？

与sed和 grep很相似，awk是一种样式扫描与处理工具。但其功能却大大强于sed和grep。awk提供了极其强大的功能：它几乎可以完成grep和sed所 能完成的全部工作，同时，它还可以可以进行样式装入、流控制、数学运算符、进程控制语句甚至于内置的变量和函数。它具备了一个完整的语言所应具有的几乎所 有精美特性。实际上，awk的确拥有自己的语言：awk程序设计语言，awk的三位创建者已将它正式定义为：样式扫描和处理语言。

awk编程

- 为什么使用awk?
- 即使如此，你也许仍然会问，我为什么要使用awk?
- 使用awk的第一个理由是基于文本的样式扫描和处理是我们经常做的工作，awk所做的工作有些象数据库，但与数据库不同的是，它处理的是文本文件，这些文件没有专门的存储格式，普通的人们就能编辑、阅读、理解和处理它们。而数据库文件往往具有特殊的存储格式，这使得它们必须用数据库处理程序来处理它们。既然这种类似于数据库的处理工作我们经常会遇到，我们就应当找到处理它们的简便易行的方法，UNIX有很多这方面的工具，例如sed、grep、sort以及find等等，awk是其中十分优秀的一种。

awk编程

- 使用awk的第二个理由是awk是一个简单的工具，当然这是相对于其强大的功能来说的。的确，UNIX有许多优秀的工具，例如UNIX天然的开发工具C语言及其延续C++就非常的优秀。但相对于它们来说，awk完成同样的功能要方便和简捷得多。这首先是因为awk提供了适应多种需要的解决方案：从解决简单 问题的awk命令行到复杂而精巧的awk程序设计语言，这样做的好处是，你可以不必用复杂的方法去解决本来很简单的问题。

awk编程

- 例如，你可以用一个命令行解决简 单的问题，而C不行，即使一个再简单的程序，C语言也必须经过编写、编译的全过程。其次，awk本身是解释执行的，这就使得awk程序不必经过编译的过 程，同时，这也使得它与shell script程序能够很好的契合。最后，awk本身较C语言简单，虽然awk吸收了C语言很多优秀的成分，熟悉C语言会对学习awk有很大的帮助，但 awk本身不须要会使用C语言——一种功能强大但需要大量时间学习才能掌握其技巧的开发工具。

awk编程

- 使用awk的第三个理由是awk是一个容易获得的工具。与C和C++语言不同，awk只有一个文件(/bin/awk)，而且几乎每个版本的UNIX都提供各自版本的awk，你完全不必费心去想如何获得awk。但C语言却不是这样，虽然C语言是UNIX天然的开发工具，但这个开发工具却是单独发行的，换言之，你必须为你的UNIX版本的C语言开发工具单独付费（当然使用D版者除外），获得并安装它，然后你才可以使用它。

awk编程

- 基于以上理由，再加上awk强大的功能，我们有理由说，如果你要处理与文本样式扫描相关的工作，awk应该是你的第一选择。在这里有一个可遵循的一般原 则：如果你用普通的shell工具或shell script有困难的话，试试awk，如果awk仍不能解决问题，则便用C语言，如果C语言仍然失败，则移至C++。

awk编程

- awk的调用方式：
- 1、 awk命令行，你可以象使用普通UNIX命令一样使用awk，在命令行中你也可以使用awk程序设计语言，虽然awk支持多行的录入，但是录入长长的命令行并保证其正确无误却是一件令人头疼的事，因此，这种方法一般只用于解决简单的问题。当然，你也可以在shell script程序中引用awk命令行甚至awk程序脚本。

awk编程

- 2、使用-f选项调用awk程序。awk允许将一段awk程序写入一个文本文件，然后在awk命令行中用-f选项调用并执行这段程序。具体的方法我们将在后面的awk语法中讲到。
- 3、利用命令解释器调用awk程序：利用UNIX支持的命令解释器功能，我们可以将一段awk程序写入文本文件，然后在它的第一行加上:
#!/bin/awk -f

awk编程

- 并赋予这个文本文件以执行的权限。这样做之后，你就可以在命令行中用类似于下面这样的方式调用并执行这段awk程序了。
-
- awk脚本文本名 待处理文件

awk编程

- awk的语法：
- 与其它UNIX命令一样，awk拥有自己的语法：
- awk [-F re] [parameter...] ['prog'] [-f progfile][in_file...]
- 参数说明：
- -F re:允许awk更改其字段分隔符。
- parameter: 该参数帮助为不同的变量赋值。
- 'prog': awk的程序语句段。这个语句段必须用单拓号：'和'括起，以防被shell解释。这个程序语句段的标准形式为：
- 'pattern {action}'

awk编程

- 其中pattern参数可以是egrep正则表达式中的任何一个，它可以使用语法/re/再加上一些样式匹配技巧构成。与sed类似，你也可以使用"," 分开两样式以选择某个范围。关于匹配的细节，你可以参考附录，如果仍不懂的话，找本UNIX书学学grep和sed（本人是在学习sed时掌握匹配技术 的）。 action参数总是被大括号包围，它由一系统awk语句组成，各语句之间用";"分隔。awk解释它们，并在pattern给定的样式匹配的记录上执行 其操作。

awk编程

- 与shell类似，你也可以使用“#”作为注释符，它使“#”到行尾的内容成为注释，在解释执行时，它们将被忽略。你可以省略pattern和action之一，但不能两者同时省略，当省略pattern时没有样式匹配，表示对所有行（记录）均执行操作，省略action时执行缺省的操作——在 标准输出上显示。

awk编程

- -f progfile:允许awk调用并执行progfile指定有程序文件。progfile是一个文本文件，他必须符合awk的语法 .
- in_file:awk的输入文件，awk允许对多个输入文件进行处理。值得注意的是awk不修改输入文件。如果未指定输入文件，awk将接受标准输入，并将结果显示在标准输出上。awk支持输入输出重定向。

awk编程

- awk的记录、字段与内置变量:
- 前面说过，awk处理的工作与数据库的处理方式有相同之处，其相同处之一就是awk支持对记录和字段的处理，其中对字段的处理是grep和sed不能实现 的，这也是awk优于二者的原因之一。在awk中，缺省的情况下总是将文本文件中的一行视为一个记录，而将一行中的某一部分作为记录中的一个字段。为了操 作这些不同的字段，awk借用shell的方法，用$1,$2,$3...这样的方式来顺序地表示行（记录）中的不同字段。

awk编程

- 特殊地，awk用$0表示整个行（记录）。不同的字段之间是用称作分隔符的字符分隔开的。系统默认的分隔符是空格。awk允许在命令行中用-F re的形式来改变这个分隔符。事实上，awk用一个内置的变量FS来记忆这个分隔符。 awk中有好几个这样的内置变量，例如，记录分隔符变量RS、当前工作的记录数NR等等，本文后面的附表列出了全部的内置变量。这些内置的变量可以在awk程序中引用或修改，例如，你可以利用NR变量在模式匹配中指定工作范围，也可以通过修改记录分隔符RS让一个特殊字符而不是换行符作为记录的分隔符。

awk编程

- awk的内置变量:
- ARGC　　命令行参数个数
- ARGV　　命令行参数排列
- ENVIRON　支持队列中系统环境变量的使用
- FILENAME　浏览的文件名
- FNR　　浏览文件的记录数
- FS　　输入文件的分隔符，等价于命令行-F选项
- NF　　浏览记录的域个数
- NR　　已读的记录数
- OFS　　输出的域分隔符
- ORS　　输出的记录分隔符
- RS　　控制记录分隔符

awk的内置变量

- ENVIRON　　UNIX环境变量
  ERRNO　　UNIX系统错误消息

awk编程

- ARGC支持传入awk脚本的参数，ARGV是它的参数数组，与main(int argc,char *argv[ ])这种一样的，其中一个元素表示为：ARGV[n],表示第n个元素。
- 例:显示文本文件myfile中第七行到第十五行中以字符%分隔的第一字段，第三字段和第七字段:
- awk -F % 'if(NR==7||NR==15) {printf $1 $3 $7}' myfile

awk编程

- awk的内置函数：
- awk 之所以成为一种优秀的程序设计语言的原因之一是它吸收了某些优秀的程序设计语言（例如C）语言的许多优点。这些优点之一就是内置函数的使用，awk定义并 支持了一系列的内置函数，由于这些函数的使用，使得awk提供的功能更为完善和强大。

awk编程

- 例如，awk使用了一系列的字符串处理内置函数(这些函数看起来与C 语言的字符串处理函数相似，其使用方式与C语言中的函数也相差无几)，正是由于这些内置函数的使用，使awk处理字符串的功能更加强大。本文后面的附录中 列有一般的awk所提供的内置函数，这些内置函数也许与你的awk版本有些出入，因此，在使用之前，最好参考一下你的系统中的联机帮助。

awk编程

- 作为内置函数的一个例子，我们将在这里介绍awk的printf函数，这个函数使得awk与c语言的输出相一致。实际上，awk中有许多引用形式都是从C 语言借用过来的。如果你熟悉C语言，你也许会记得其中的printf函数，它提供的强大格式输出功能曾经带我们许多的方便。幸运的是，我们在awk中又和 它重逢了。awk中printf几乎与C语言中一模一样，如果你熟悉C语言的话，你完全可以照C语言的模式使用awk中的printf。因此在这里，我们 只给出一个例子，如果你不熟悉的话，请随便找一本C语言的入门书翻翻。

awk编程

- 例:显示文件myfile中的行号和第3字段:
- awk '{printf("%03d,%s\n",NR,$1)} ' myfile

awk的内置函数

- awk的常用的串的一些函数：
- 1、gsub（r,s）在整个$0中用s代替r,gsub(r,s,t)在在整个t中用s代替r，用法：/目标模式/,替换模式
- awk ‘gsub(/3100/,0000){print $0}’ shell_awk.unl 将文件中的3100用0代替，如果记录有：3100|20|30||||||则在屏幕上会有0|20|30||||||
- awk 'gsub(/3100/,"0000"){print $0}' shell_awk.unl
- 将文件中的3100用0000代替，如果记录有：3100|20|30||||||则在屏幕上会有0000|20|30||||||

awk的内置函数

- 2、index(s,t) 返回s在t中的第一位置，字符串必须用双引号，用法：
- awk -F "|" '{print index($0,"20")}' shell_awk.unl
- 如果有记录3100|20|30||||||在该记录中20出现的位置为：6
- 3、length（s）返回串s的长度，用法：
- awk -F "|" '{print length($1)}' shell_awk.unl
- 有记录3100|20|30||||||则第1个字段的长度为4
- 4、match(s,r)测试s是否含有匹配r的字符串，用法：awk -F “|” ‘{print match($1,“20”)}’ shell_awk.unl与index命令类似返回值，

awk的内置函数

- 5、substr(s,p)返回s中p开始的以后部分：
- awk -F “|” ‘{print substr($1,2)}’ shell_awk.unl显示第一个字段从第二位开始的后续部分，有记录3100|20|30||||||结果为：100
- substr(s,p,n)返回s中从p开始以后长度为n的部分：
- awk -F "|" '{print substr($1,2,3)}' shell_awk.unl有记录31001|20|30||||||结果为： 100
- awk -F “|” ‘{print substr($1,“|”)}’ shell_awk.unl显示第一个‘|’前面的所有部分。
- awk -F “|” ‘{print substr($1,2)}’ shell_awk.unl显示第一个‘|’前面的从第2个字符开始的部分，有记录3100|20|30||||||，则结果为： 100

awk的内置函数

- match(string,reg) 返回常规表达式reg匹配的string中的位置
- printf(format,variable) 格式化输出，按format提供的格式输出变量variable。
- split(string,store,delim) 根据分界符delim,分解string为store的数组元素
- sprintf(format,variable) 返回一个包含基于format的格式化数据，variables是要放到串中的数据

awk的内置函数

- strftime(format,timestamp) 返回一个基于format的日期或者时间串，timestmp是systime()函数返回的时间
- sub(reg,string,target) 第一次当常规表达式reg匹配，替换target串中的字符串
- totower(string) 返回string中对应的小写字符
- toupper(string) 返回string中对应的大写字符
- atan(x,y) x的余切(弧度)

awk的内置函数

- cos(x) x的余弦(弧度)
  exp(x) e的x幂
  int(x) x的整数部分
  log(x) x的自然对数值
  rand() 0-1之间的随机数
  sin(x) x的正弦(弧度)
  sqrt(x) x的平方根
  srand(x) 初始化随机数发生器。如果忽略x，则使用srand()

awk编程

- 在命令行使用awk
- 按照顺序，我们应当讲解awk程序设计的内容了，但在讲解之前，我们将用一些例子来对前面的知识进行回顾，这些例子都是在命令行中使用的，由此我们可以知 道在命令行中使用awk是多么的方便。这样做的原因一方面是为下面的内容作铺垫，另一方面是介绍一些解决简单问题的方法，我们完全没有必要用复杂的方法来 解决简单的问题----既然awk提供了较为简单的方法的话。

awk编程

- 例：显示文本文件mydoc匹配（含有）字符串"sun"的所有行。
- awk '/sun/{print}' mydoc
- 由于显示整个记录（全行）是awk的缺省动作，因此可以省略action项。
- $awk '/sun/' mydoc

awk编程

- 例：下面是一个较为复杂的匹配的示例：
- awk '/[Ss]un/,/[Mm]oon/ {print}' myfile
- 它将显示第一个匹配Sun或sun的行与第一个匹配Moon或moon的行之间的行，并显示到标准输出上。

awk编程

- 例：下面的示例显示了内置变量和内置函数length（）的使用：
- awk 'length($0)>80 {print NR}' myfile
- 这个命令是显示长度大于80的行的行号
- 在这里，用$0表示整个记录（行），同时，内置变量NR不使用标志符'$'

awk编程

- 作为一个较为实际的例子，我们假设要对UNIX中的用户进行安全性检查，方法是考察/etc下的passwd文件，检查其中的 passwd字段（第二字段）是否为"*"，如不为"*"，则表示该用户没有设置密码，显示出这些用户名（第一字段）。我们可以用如下语句实现：
- awk -F":" '{if($2=="")printf("%s no assword",$1)}' /etc/passwd

awk编程

- 在这个示例中，passwd文件的字段分隔符是“：”，因此，必须用-F “:”来更改默认的字段分隔符，这个示例中也涉及到了内置函数printf的使用。
- awk -F":" '{if($2!="")printf("%s no assword",$1)}' /etc/passwd
-

awk的变量

- awk的变量
- 如同其它程序设计语言一样，awk允许在程序语言中设置变量，
- awk 提供两种变量，一种是awk内置的变量，这前面我们已经讲过，需要着重指出的是，与后面提到的其它变量不同的是，在awk程序中引用内置变量不需要使用标 志符"$"（回忆一下前面讲过的NR的使用）。awk提供的另一种变量是自定义变量。awk允许用户在awk程序语句中定义并调用自已的变量。

awk的变量

- 当然这种变量不能与内置变量及其它awk保留字相同，在awk中引用自定义变量必须在它前面加上标志符"$"。与C语言不同的是，awk中不需要对变量进行初始化， awk根据其在awk中第一次出现的形式和上下文确定其具体的数据类型。当变量类型不确定时，awk默认其为字符串类型。这里有一个技巧：如果你要让你的 awk程序知道你所使用的变量的明确类型，你应当在在程序中给它赋初值。在后面的实例中，我们将用到这一技巧。

运算与判断

- 作为一种程序设计语言所应具有的特点之一，awk支持多种运算，这些运算与C语言提供的几本相同：如+、-、*、/、%等等，同时，awk也支持C语言中 类似++、--、+=、-=、=+、=-之类的功能，这给熟悉C语言的使用者编写awk程序带来了极大的方便。作为对运算功能的一种扩展，awk还提供了 一系列内置的运算函数（如log、sqr、cos、sin等等）和一些用于对字符串进行操作（运算）的函数（如length、substr等等）。这些函 数的引用大大的提高了awk的运算功能。

运算与判断

- 作为对条件转移指令的一部分，关系判断是每种程序设计语言都具备的功能，awk也不例外。awk中允许进行多种测试，如常用的==（等于）、！=（不等于）、>（大于）、<（小于）、>=（大于等于）、>=（小于等 于）等等，同时，作为样式匹配，还提供了~（匹配于）和！~（不匹配于）判断。

运算与判断

- 1）awk -F “|” ‘{if($1~/100/)print $0}’ shell_awk.unl 将文件shell_awk.unl 中第1个字段含有100的输出在屏幕上。
- 2）awk -F “|” ‘$1~/100/’ shell_awk.unl可以实现上面相同的功能。
- 可以用if($1！ ~ /100/)表示第1个字段不是100的。
- 2、精确匹配：两种方式
- 1）awk -F “|” ‘{if($1==“410”)print $0}’ shell_awk.unl 会将第1个字段为410的输出在屏幕上。
- 2）awk -F "|" '$1=="410"' shell_awk.unl
- 非的方式与上面同理

运算与判断

- 作为对测试的一种扩充，awk也支持用逻辑运算符:!(非)、&&（与）、||（或）和括号（）进行多重判断，这大大增强了awk的功能。本文的附录中列出了awk所允许的运算、判断以及操作符的优先级。

awk的流程控制

- 流程控制语句是任何程序设计语言都不能缺少的部分。任何好的语言都有一些执行流程控制的语句。awk提供的完备的流程控制语句类似于C语言，这给我们编程带来了极大的方便。
- 1、BEGIN和END:
- 在awk 中两个特别的表达式，BEGIN和END，这两者都可用于pattern中（参考前面的awk语法），提供BEGIN和END的作用是给程序赋予初始状态 和在程序结束之后执行一些扫尾的工作。

awk的流程控制

- 任何在BEGIN之后列出的操作（在{}内）将在awk开始扫描输入之前执行，而END之后列出的操作将在扫描完全部的输入之后执行。因此，通常使用BEGIN来显示变量和预置（初始化）变量，使用END来输出最终结果。
- 例：累计销售文件xs中的销售金额（假设销售金额在记录的第三字段）：
- $awk
  >'BEGIN { FS=":";print "统计销售金额";total=0}
  >{print $3;total=total+$3;}
  >END {printf "销售金额总计：%.2f",total}' sx
  （注：>是shell提供的第二提示符，如要在shell程序awk语句和awk语言中换行，则需在行尾加反斜杠）

awk的流程控制

- 在这里，BEGIN预置了内部变量FS（字段分隔符）和自定义变量total,同时在扫描之前显示出输出行头。而END则在扫描完成后打印出总合计。

awk的流程控制

- 2、流程控制语句
- awk提供了完备的流程控制语句，其用法与C语言类似。下面我们一一加以说明：
- 2.1、if...else语句：
- 格式：
  if(表达式)
  语句1
  else
  语句2

awk的流程控制

- 格式中“语句1”可以是多个语句，如果你为了方便awk判断也方便你自己阅读，你最好将多个语句用{}括起来。awk分枝结构允许嵌套，其格式为：
- if(表达式1)
  {
- 　　if(表达式2)
  　　语句1
  　　else
  　　语句2
  }

awk的流程控制

else

- {
- if(表达式3)
  语句4
  else
  语句5
  }
- 语句6

awk的流程控制

- 2.2、while语句
- 格式为:
- while(表达式)
  {
- 　语句
  }

awk的流程控制

- 2.3、do-while语句
- 格式为:
- do
  {
  语句
  }while(条件判断语句)

awk的流程控制

- 2.4、for语句
- 格式为:
- for(初始表达式;终止条件;步长表达式)
  {
- 　语句
- }
- 在awk 的 while、do-while和for语句中允许使用break,continue语句来控制流程走向，也允许使用exit这样的语句来退出。break 中断当前正在执行的循环并跳到循环外执行下一条语句。

awk的流程控制

- continue从当前位置跳到循环开始处执行。对于exit的执行有两种情况： 当exit语句不在 END 中时，任何操作中的exit命令表现得如同到了文件尾，所有模式或操作执行将停止，END模式中的操作被执行。而出现在END中的exit将导致程 序终止。

awk的流程控制

- awk中的自定义函数
- 定义和调用用户自己的函数是几乎每个高级语言都具有的功能，awk也不例外，但原始的awk并不提供函数功能，只有在nawk或较新的awk版本中才可以增加函数。
- 函数的使用包含两部分：函数的定义与函数调用。其中函数定义又包括要执行的代码（函数本身）和从主程序代码传递到该函数的临时调用。

awk中的自定义函数

- awk函数的定义方法如下:
- function 函数名(参数表)
- {
  函数体
  }

cfg_file.sh

awk中的自定义函数

- 在gawk中允许将function省略为func，但其它版本的awk不允许。函数名必须是一个合法的标志符，参数表中可以不提供参数（但在调用函数时 函数名后的一对括号仍然是不可缺少的），也可以提供一个或多个参数。与C语言相似，awk的参数也是通过值来传递的。

awk中的自定义函数

- 在awk 中调用函数比较简单，其方法与C语言相似，但awk比C语言更为灵活，它不执行参数有效性检查。换句话说，在你调用函数时，可以列出比函数预计（函数定义 中规定）的多或少的参数，多余的参数会被awk所忽略，而不足的参数，awk将它们置为缺省值0或空字符串，具体置为何值，将取决于参数的使用方式。

awk中的自定义函数

- awk函数有两种返回方式：隐式返回和显式返回。当awk执行到函数的结尾时，它自动地返回到调用程序，这是函数是隐式返回的。如果需要在结束之前退出函数，可以明确地使用返回语句提前退出。方法是在函数中使用形如：return 返回值 格式的语句。

awk中的自定义函数

- 例：下面的例子演示了函数的使用。在这个示例中，定义了一个名为print_header的函数，该函数调用了两个参数FileName和 PageNum，FileName参数传给函数当前使用的文件名，PageNum参数是当前页的页号。这个函数的功能是打印（显示）出当前文件的文件名，和当前页的页号。完成这个功能后，这个函数将返回下一页的页号。

awk中的自定义函数

```
'BEGIN{
    pageno=1;file=FILENAME
   #调用函数  print_header
    pageno=print_header(file，pageno);
    printf("当前页页号是：%dn",pageno);
}
    #定义函数print_header
    function print_header(FileName,PageNum)
    {
        printf(“%s %d\n",FileName,PageNum);
        PageNum++;
        return        PageNUm;
    }
}' myfile
```

awk高级输入输出

- 1.读取下一条记录:
- awk的next语句导致awk读取下一个记录并完成模式匹配，然后立即执行相应的操作。通常它用匹配的模式执行操作中的代码。next导致这个记录的任何额外匹配模式被忽略。
- 2.简单地读取一条记录
- awk 的 getline语句用于简单地读取一条记录。如果用户有一个数据记录类似两个物理记录，那么getline将尤其有用。它完成一般字段的分离(设置字段变 量$0 FNR NF NR)。如果成功则返回1，失败则返回0（到达文件尾）。

awk高级输入输出

- 如果需简单地读取一个文件，则可以编写以下代码：
- 例： 示例getline的使用
- {
- while(getline==1)
  {
  #process the inputted fields
  }
}

awk高级输入输出

- 也可以使getline保存输入数据在一个字段中，而不是通过使用getline variable的形式处理一般字段。当使用这种方式时，NF被置成0，FNR和NR被增值。
- 用户也可以使用getline<"filename"方式从一个给定的文件中输入数据，而不是从命令行所列内容输入数据。此时，getline将完 成一般字段分离（设置字段变量$0和NF)。如果文件不存在，返回-1,成功，返回1,返回0表示失败。用户可以从给定文件中读取数据到一个变量中，也可 以用stdin(标准输入设备）或一个包含这个文件名的变量代替filename。值得注意的是当使用这种方式时不修改FNR和NR。

Awk中读文件

```
awk -v pidchidresult=${pidchidresult} -v cfgfile=${cfgfile} 'BEGIN{
FS="=";
#首先读出配置文件数据
while(getline<cfgfile)
{
    if(NF < 2)
    {
        continue;
    }
     m_pid=$1;
    m_chid=$2;
    gsub(" ","",m_pid);
    gsub(" ","",m_chid);
    key=sprintf("%s",m_pid);
    arry[key]=m_chid;
}
FS="|";
}
{
   #处理部分
}
END{#处理结束
}’ 文件名
```

Awk中读文件

- 在BEGIN{}中可以先读取配置文件中的类型，便于读取数据文件的时候处理

awk高级输入输出

- 另一种使用getline语句的方法是从UNIX命令接受输入，例如下面的例子:
- 例： 示例从UNIX命令接受输入
- {
- ```
  while("who -u"|getline)
  {
      #process each line from the who command
  }
}
```
- 当然，也可以使用如下形式:
- "command" | getline variable
```

awk高级输入输出

- 3.关闭文件:
- 　　awk中允许在程序中关闭一个输入或输出文件，方法是使用awk的close语句。
- close("filename"t
- 　　filename可以是getline打开的文件（也可以是stdin,包含文件名的变量或者getline使用的确切命令）。或一个输出文件（可以是stdout,包含文件名的变量或使用管道的确切命令）。

awk高级输入输出

- 4.输出到一个文件:
- awk中允许用如下方式将结果输出到一个文件:
- printf("hello word!n"t>"datafile"
  或
  printf("hello word!n"t>>"datafile"

awk高级输入输出

- 5.输出到一个命令
- awk中允许用如下方式将结果输出到一个命令:
- printf("hello word!n"t|"sort-t',"

awk与shell script混合编程

awk与shell script混合编程

- 因为awk可以作为一个shell命令使用，因此awk能与shell批处理程序很好的融合在一起，这给实现awk与shell程序的混合编程提供了可能。实现混合编程的关键是awk与shell script之间的对话，换言之，就是awk与shell script之间的信息交流:awk从shell script中获取所需的信息（通常是变量的值）、在awk中执行shell命令行、shell script将命令执行的结果送给awk处理以及shell script读取awk的执行结果等等。

awk与shell script混合编程实例讲解

- 1、将shell中的变量串给awk
- 有两个数，d1=100，d2=200
- echo “”|awk –v value1=${d1} –v value2=${d2} ‘BEGIN{
- diff=0
- }
- {
- diff=value1-value2
- print diff
- }‘|read ret_result
- echo “ret_sult=${ret_sult}”
- 结果：为-100

shell_and_awk.sh

awk与shell script混合编程

- 从上面的例子可以看出，要将shell中的两个变量d1,d2传到awk中，用-v value1=${d1} -v value2=${d2} 这种形式就可以将shell中的值传给awk，也可以用其他的方式传递给awk，下面的这种方法就是另一种传递。

awk与shell script混合编程

- 2、将shell命令的执行结果送给awk处理
- who -u | awk '{printf("%s正在执行%sn",$2,$1)}'
- 该命令将打印出注册终端正在执行的程序名

```
find . -name "*.unl" | xargs awk -F "|"
  '{if(substr($1,1,2)=="10"){print $0}}‘
```

这个会将以unl结尾的文件中的第一个字段的前两个字符为10的记录打印在屏幕上

awk与shell script混合编程

- 处理文件传递参数:
- cfgareaFLAG=0
- cfgareaFLAG=`awk -F"=" 'BEGIN{
- 处理文件的开始准备工作
- }
- function throwblank(outString)
- {
- 处理
- }
- {
- 调用函数循环处理文件中的每一行。
- }
- END{
- 处理完文件的最后的处理，print 返回值
- }' config.cfg`
- 根据变量cfgareaFLAG的值判断处理情况,例子shell_awk_readfile.sh

shell_awk_readfile.sh

awk中函数的用法

- Input1=10
- Input2=20
- awk –v parameter1=${input1} –v parameter2=${input2} ‘BEGIN{
- }

- function fun(参数1，参数2，...)
- {
- 处理
- }

- {
- ret=0
- ret= fun(参数1,参数2)
- }’ |read ret_result

shell_and_awk_fun.sh

- 函数可以有多个函数，方式是一样的，
- -v是一种传递参数形式， parameter1是awk内部用的参数，input是从shell中传递过来的值,实际的例子：shell_and_awk_fun.sh

读文件的方式

- #! /bin/ksh
- . ${HOME}/smp_run/bin/win_smpser_common.rc
- #读一般文件
- awk -v Date1=$date -v filename=$tempfile 'BEGIN{FS="|"}
- {
- n=0
- n=split($0,arr,"|")
- print "n="n
- print "arr[1]="arr[1]" arr[2]="arr[2]" arr[3]="arr[3]" arr[4]="arr[4]
- print $0
- }
- END{
- }' shell_awk.unl
- 这里的split是将字段以“|”为分隔符放到数组arr中，如果有记录10|20|30|则n=4,arr[1]=10,arr[2]=20,arr[3]=30,arr[4]=空
- 实际例子：shell_and_awkreadfile.sh

shell_and_awkreadfile.sh

读文件的方式

- 先用shell读一个文件的内容，然后在另一个文件中查找与之相同的记录。实际例子：shell_awk_fileZ.sh
- #! /bin/sh
- oldIFS=$IFS
- #字段分隔符号为'|'
- IFS="|"
- while read filed1 other
- do
- echo "filed1=${filed1}"
- line2=`uncompress -cf ${accinputfile} | awk -F'|' -v value=${filed1} 'BEGIN{}{if(value == $1){print $0;print "第"NR"行的第一个字段与之相同！";exit}else{print NR"行的第一个字段不同"}}END{}'`
- echo "line2=${line2}"
- done < shell_awk3.unl
- IFS=${oldIFS}

读文件的方式

- 在一个文件中存放有文件名，先用shell读出文件名，再用awk处理文件，
  oldIFS=${IFS}
- IFS="|“
- while read file
- do
- awk -v File=${outfile} 'BEGIN{
- FS="|"
- }
- {
- 处理
- }
- END{
- 处理
- }’ ${file}
- done < ${filename}
- IFS=${oldIFS}
- 实际例子：shell_awk_whilereadfile.sh

shell_awk_whilereadfile.sh

用awk读多个文件处理

```
#! /bin/ksh
awk -v Date1=$date -v filename=$tempfile 'BEGIN{
FS="|"
total=0
}
{
        file1=sprintf("shell_awk.unl")
        file2=sprintf("shell_awk2.unl")
        file3=sprintf("shell_awk3.unl")
        if(FILENAME == file1)
        {
                处理
        }
```

用awk读多个文件处理

- if(FILENAME == file2)
- {
- 处理
- }
- if(FILENAME == file3)
- {
- 处理
- }
- }
- END{
- print "结果：total="total
- }' *.unl
- 实际例子：shell_and_awk_morfile.sh

shell_and_awk_morfile.sh

其他的shell命令与awk的套用

- 1）cat shell_awk.unl | awk -F "|" '{print $1}‘
- cat文件用管道形式传给awk 将每行的第一个字段打印在屏幕上。
- 2） line2=`uncompress -cf ${accinputfile} | awk -F‘|’ -v value=${filed1} ‘BEGIN{}{if(value == $1){print $0;print “第” NR“行的第一个字段与之相同！ ” ;exit}else{print NR“行的第一个字段不同” }}END{}’`
- 将解压后的文件管道传给awk处理判断文件的第一个字段与传进来的值value是否相同，将print的结果传给line2
- 也可以：uncompress -cf 20071106_pps_managelog_101.list.Z | awk -F "|" '{print $0}‘
- 在屏幕上将解压后的文件的记录输出在屏幕上。

其他的shell命令与awk的套用

- 3) line=`uncompress -cf ${accinputfile}|wc -l|awk '{print $1}'`
- 将解压文件的记录行数传递给line

- 4) file_size=`ls -l ${accinputfile} |awk 'BEGIN { FS = " " ;} { printf "%s\t%s",$9, $5;}'`
- 将文件名和文件的大小传递给file_size

awk与shell script混合编程

- 一些自己写的实际简单例子，可以参考看看

混合编程

shell与awk混合编程例子

转义 – 左对齐，右对齐

- echo "afa|2" | awk -F"|" '{m_line=sprintf("%\%\-%ds",$2); print m_line}‘
- 得到结果:
- %-2s
- Printf(“%-20s”,s);左对齐，右边补空格
- Printf(“%+20s”,s);右对齐，左边补空格
- 边长的数据根据指定的长度进行左补齐或者右补齐:
- echo "|20|44|" | awk -F"|" '{m_line=sprintf("%\%\-%ds%\%\s",$2);
- print m_line;
- m_re= sprintf(m_line,$1,$3);
- print "m_re=["m_re"]";}‘
- 结果为:
- %-20s%s
- m_re=[44]

转义 – 左对齐，右对齐

- 右对齐方式:
- echo "11|20|44|" | awk -F"|" '{m_line=sprintf("%\%\+%ds%\%\s",$2);print m_line;m_re= sprintf(m_line,$1,$3);print "m_re=["m_re"]";}‘
- 结果为:
- %+20s%s
- m_re=[1144]
- 右对齐，左边补0的小数位
- echo "23" | awk '{printf("%020.2f",1);}‘
- 左补0对齐
- echo "20" |awk '{printf("%04s",$1);}'

数组

- echo ""|awk '{msum["11"]=10;print msum["11"];}‘
- 有数组sum[key]循环数组方式
- for(i in sum)
- {
- print "key["i"]="sum[i] >> resultfile;
- }

函数去左右空格

```
function throwblank(outString)
{
  #去掉字符串左右空格
                Len=length(outString);
                for(i=1; i <= Len; i++)
                {
                        start=substr(outString, 1, 1);
                        if(start == " ")
                        {
                                outString=substr(outString, 2);
                        }
                        else
                        {
                                break
                        }
                }
                Len=length(outString)
                if(Len == 0)
                {
                        return 0
                }
```

函数去左右空格

```
    Len=length(outString)
    for(i=Len; i >0; i--)
    {
            start=substr(outString, i, 1)
            if(start == " ")
            {
                    outString=substr(outString, 1, i-1)
            }
            else
            {
                    break
            }
    }
    Len=length(outString)
    if(Len == 0)
    {
            return 0
    }
    else
    {
            return outString
    }
}
```

单，双引号输出

- 单引号输出
- echo "111" | awk '{print "'\"";}‘
- Print后面是一个双引号，一个单引号，一个反斜杠，两个单引号，一个双引号

- 双引号输出：
- echo "111" | awk '{print "\"";}‘
- Print后面是一个双引号，一个反斜杠，一个双引号，一个双引号