2009-10-30 17:18 linux shell awk 語法

shell 98707

此篇轉自於 http://blog.csdn.net/weekly123/archive/2006/12/28/1465675.aspx

Awk 是一種非常好的語言，同時有一個非常奇怪的名稱。在本系列（共三篇文章）的第一篇文章中，Daniel Robbins 將使您迅速掌握 awk 編程技巧。隨著本系列的進展，將討論更高級的主題，最後將演示一個真正的高級 awk 演示程序。

捍衛 awk
在本系列文章中，我將使您成為精通 awk 的編碼人員。我承認，awk 並沒有一個非常好聽且又非常「時髦」的名字。awk 的 GNU 版本（叫作 gawk）聽起來非常怪異。那些不熟悉這種語言的人可能聽說過 "awk"，並可能認為它是一組落伍且過時的混亂代碼。它甚至會使最博學的 UNIX 權威陷於錯亂的邊緣（使他不斷地發出 "kill -9!" 命令，就像使用咖啡機一樣）。

的確，awk 沒有一個動聽的名字。但它是一種很棒的語言。awk 適合於文本處理和報表生成，它還有許多精心設計的特性，允許進行需要特殊技巧程序設計。與某些語言不同，awk 的語法較為常見。它借鑒了某些語言的一些精華部分，如 C 語言、python 和 bash（雖然在技術上，awk 比 python 和 bash 早創建）。awk 是那種一旦學會了就會成為您戰略編碼庫的主要部分的語言。

第一個 awk
讓我們繼續，開始使用 awk，以瞭解其工作原理。在命令行中輸入以下命令：

```
$ awk '{ print }' /etc/passwd
```

您將會見到 /etc/passwd 文件的內容出現在眼前。現在，解釋 awk 做了些什麼。調用 awk 時，我們指定 /etc/passwd 作為輸入文件。執行 awk 時，它依次對 /etc/passwd 中的每一行執行 print 命令。所有輸出都發送到 stdout，所得到的結果與與執行catting /etc/passwd完全相同。

現在，解釋 { print } 代碼塊。在 awk 中，花括號用於將幾塊代碼組合到一起，這一點類似於 C 語言。在代碼塊中只有一條 print 命令。在 awk 中，如果只出現 print 命令，那麼將打印當前行的全部內容。

這裡是另一個 awk 示例，它的作用與上例完全相同：

```
$ awk '{ print $0 }' /etc/passwd
```

在 awk 中，$0 變量表示整個當前行，所以 print 和 print $0 的作用完全一樣。

如果您願意，可以創建一個 awk 程序，讓它輸出與輸入數據完全無關的數據。以下是一個示例：

```
$ awk '{ print "" }' /etc/passwd
```

只要將 "" 字符串傳遞給 print 命令，它就會打印空白行。如果測試該腳本，將會發現對於 /etc/passwd 文件中的每一行，awk 都輸出一個空白行。再次說明， awk 對輸入文件中的每一行都執行這個腳本。以下是另一個示例：

```
$ awk '{ print "hiya" }' /etc/passwd
```

運行這個腳本將在您的屏幕上寫滿 hiya。

多個字段
awk 非常善於處理分成多個邏輯字段的文本，而且讓您可以毫不費力地引用 awk 腳本中每個獨立的字段。以下腳本將打印出您的系統上所有用戶帳戶的列表：

```
$ awk -F":" '{ print $1 }' /etc/passwd
```

上例中，在調用 awk 時，使用 -F 選項來指定 ":" 作為字段分隔符。awk 處理 print $1 命令時，它會打印出在輸入文件中每一行中出現的第一個字段。以下是另一個示例：

```
$ awk -F":" '{ print $1 $3 }' /etc/passwd
```

以下是該腳本輸出的摘錄：

```
halt7
operator11
root0
shutdown6
sync5
bin1
....etc.
```

如您所見，awk 打印出 /etc/passwd 文件的第一和第三個字段，它們正好分別是用戶名和用戶標識字段。現在，當腳本運行時，它並不理想 -- 在兩個輸出字段之間沒有空格！如果習慣於使用 bash 或 python 進行編程，那麼您會指望 print $1 $3 命令在兩個字段之間插入空格。然而，當兩個字符串在 awk 程序中彼此相鄰時，awk 會連接它們但不在它們之間添加空格。以下命令會在這兩個字段中插入空格：

```
$ awk -F":" '{ print $1 " " $3 }' /etc/passwd
```

以這種方式調用 print 時，它將連接 $1、" " 和 $3，創建可讀的輸出。當然，如果需要的話，我們還可以插入一些文本標籤：

```
$ awk -F":" '{ print "username: " $1 "\t\tuid:" $3" }' /etc/passwd
```

這將產生以下輸出：

```
username: halt uid:7
username: operator uid:11
username: root uid:0
username: shutdown uid:6
username: sync uid:5
username: bin uid:1
....etc.
```

外部腳本

將腳本作為命令行自變量傳遞給 awk 對於小的單行程序來說是非常簡單的，而對於多行程序，它就比較複雜。您肯定想要在外部文件中撰寫腳本。然後可以向 awk 傳遞 -f 選項，以向它提供此腳本文件：

```
$ awk -f myscript.awk myfile.in
```

將腳本放入文本文件還可以讓您使用附加 awk 功能。例如，這個多行腳本與前面的單行腳本的作用相同，它們都打印出 /etc/passwd 中每一行的第一個字段：

```
BEGIN {
FS=":"
}

{ print $1 }
```

這兩個方法的差別在於如何設置字段分隔符。在這個腳本中，字段分隔符在代碼自身中指定（通過設置 FS 變量），而在前一個示例中，通過在命令行上向 awk 傳遞 -F":" 選項來設置 FS。通常，最好在腳本自身中設置字段分隔符，只是因為這表示您可以少輸入一個命令行自變量。我們將在本文的後面詳細討論 FS 變量。

BEGIN 和 END 塊

通常，對於每個輸入行，awk 都會執行每個腳本代碼塊一次。然而，在許多編程情況中，可能需要在 awk 開始處理輸入文件中的文本之前執行初始化代碼。對於這種情況，awk 允許您定義一個 BEGIN 塊。我們在前一個示例中使用了 BEGIN 塊。因為 awk 在開始處理輸入文件之前會執行 BEGIN 塊，因此它是初始化 FS（字段分隔符）變量、打印頁眉或初始化其

awk 還提供了另一個特殊塊，叫作 END 塊。awk 在處理了輸入文件中的所有行之後執行這個塊。通常，END 塊用於執行最終計算或打印應該出現在輸出流結尾的摘要信息。

規則表達式和塊

awk 允許使用規則表達式，根據規則表達式是否匹配當前行來選擇執行獨立代碼塊。以下示例腳本只輸出包含字符序列 foo 的那些行：

```
/foo/ { print }
```

當然，可以使用更複雜的規則表達式。以下腳本將只打印包含浮點數的行：

```
/[0-9]+\.[0-9]*/ { print }
```

表達式和塊

還有許多其它方法可以選擇執行代碼塊。我們可以將任意一種布爾表達式放在一個代碼塊之前，以控制何時執行某特定塊。僅當對前面的布爾表達式求值為真時，awk 才執行代碼塊。以下示例腳本輸出將輸出其第一個字段等於 fred 的所有行中的第三個字段。如果當前行的第一個字段不等於 fred，awk 將繼續處理文件而不對當前行執行 print 語句：

```
$1 == "fred" { print $3 }
```

awk 提供了完整的比較運算符集合，包括 "==" 、"<" 、">" 、"<=" 、">=" 和 "!=" 。另外，awk 還提供了 "~" 和 "!~" 運算符，它們分別表示「匹配」和「不匹配」。它們的用法是在運算符左邊指定變量，在右邊指定規則表達式。如果某一行的第五個字段包含字符序列 root，那麼以下示例將只打印這一行中的第三個字段：

```
$5 ~ /root/ { print $3 }
```

條件語句

awk 還提供了非常好的類似於 C 語言的 if 語句。如果您願意，可以使用 if 語句重寫前一個腳本：

```
{
if ( $5 ~ /root/ ) {
print $3
}
}
```

這兩個腳本的功能完全一樣。第一個示例中，布爾表達式放在代碼塊外面。而在第二個示例中，將對每一個輸入行執行代碼塊，而且我們使用 if 語句來選擇執行 print 命令。這兩個方法都可以使用，可以選擇最適合腳本其它部分的一種方法。

以下是更複雜的 awk if 語句示例。可以看到，儘管使用了複雜、嵌套的條件語句，if 語句看上去仍與相應的 C 語言 if 語句一樣：

```
{
if ( $1 == "foo" ) {
if ( $2 == "foo" ) {
print "uno"
} else {
print "one"
}
} else if ($1 == "bar" ) {
print "two"
} else {
print "three"
}
}
```

使用 if 語句還可以將代碼：

```
! /matchme/ { print $1 $3 $4 }
```

轉換成：

```
{
```

```
print $1 $3 $4
}
}
```

這兩個腳本都只輸出不包含 matchme 字符序列的那些行。此外，還可以選擇最適合您的代碼的方法。它們的功能完全相同。

awk 還允許使用布爾運算符 "||"（邏輯與）和 "&&"（邏輯或），以便創建更複雜的布爾表達式：

```
( $1 == "foo" ) && ( $2 == "bar" ) { print }
```

這個示例只打印第一個字段等於 foo 且第二個字段等於 bar 的那些行。

數值變量！
至今，我們不是打印字符串、整行就是特定字段。然而，awk 還允許我們執行整數和浮點運算。通過使用數學表達式，可以很方便地編寫計算文件中空白行數量的腳本。以下就是這樣一個腳本：

```
BEGIN { x=0 }
/^$/ { x=x+1 }
END { print "I found " x " blank lines. " }
```

在 BEGIN 塊中，將整數變量 x 初始化成零。然後，awk 每次遇到空白行時，awk 將執行 x=x+1 語句，遞增 x。處理完所有行之後，執行 END 塊，awk 將打印出最終摘要，指出它找到的空白行數量。

字符串化變量
awk 的優點之一就是「簡單和字符串化」。我認為 awk 變量「字符串化」是因為所有 awk 變量在內部都是按字符串形式存儲的。同時，awk 變量是「簡單的」，因為可以對它執行數學操作，且只要變量包含有效數字字符串，awk 會自動處理字符串到數字的轉換步驟。要理解我的觀點，請研究以下這個示例：

```
x="1.01"
# We just set x to contain the *string* "1.01"
x=x+1
# We just added one to a *string*
print x
# Incidentally, these are comments
```

awk 將輸出：

```
2.01
```

有趣吧！雖然將字符串值 1.01 賦值給變量 x，我們仍然可以對它加一。但在 bash 和 python 中卻不能這樣做。首先，bash 不支持浮點運算。而且，如果 bash 有「字符串化」變量，它們並不「簡單」；要執行任何數學操作，bash 要求我們將數字放到醜陋的 $( )) 結構中。如果使用 python，則必須在對 1.01 字符串執行任何數學運算之前，將它轉換成浮點值。雖然這並不困難，但它仍是附加的步驟。如果使用 awk，它是全自動的，而那會使我們的代碼又好又整潔。如果想要對每個輸入行的第一個字段乘方並加一，可以使用以下腳本：

```
{ print ($1^2)+1 }
```

如果做一個小實驗，就可以發現如果某個特定變量不包含有效數字，awk 在對數學表達式求值時會將該變量當作數字零處理。

眾多運算符
awk 的另一個優點是它有完整的數學運算符集合。除了標準的加、減、乘、除，awk 還允許使用前面演示過的指數運算符 "^"、模（餘數）運算符 "%" 和其它許多從 C 語言中借入的易於使用的賦值操作符。

這些運算符包括前後加減（i++、--foo）、加 / 減 / 乘 / 除賦值運算符（ a+=3、b*=2、c/=2.2、d-=6.2）。不僅如此 -- 我們還有易於使用的模 / 指數賦值運算符（a^=2、b%=4）。

字段分隔符
awk 有它自己的特殊變量集合。其中一些允許調整 awk 的運行方式，而其它變量可以被讀取以收集關於輸入的有用信息。我們已經接觸過這些特殊變量中的一個，FS。前面已經提到過，這個變量讓您可以設置 awk 要查找的字段之間的字符序列。我們使用 /etc/passwd 作為輸入時，將 FS 設置成 ":"。當這樣做有問題時，我們還可以更靈活地使用 FS。

FS 值並沒有被限制為單一字符；可以通過指定任意長度的字符模式，將它設置成規則表達式。如果正在處理由一個或多個 tab 分隔的字段，您可能希望按以下方式設置 FS：

```
FS="\t+"
```

以上示例中，我們使用特殊 "+" 規則表達式字符，它表示「一個或多個前一字符」。

如果字段由空格分隔 (一個或多個空格或 tab) ，您可能想要將 FS 設置成以下規則表達式：

```
FS="[[:space:]+]"
```

這個賦值表達式也有問題，它並非必要。為什麼？因為缺省情況下，FS 設置成單一空格字符，awk 將這解釋成表示「一個或多個空格或 tab」。在這個特殊示例中，缺省 FS 設置恰恰是您最想要的！

複雜的規則表達式也不成問題。即使您的記錄由單詞 "foo" 分隔，後面跟著三個數字，以下規則表達式仍允許對數據進行正確的分析：

```
FS="foo[0-9][0-9][0-9]"
```

字段數量
接著我們要討論的兩個變量通常並不是需要賦值的，而是用來讀取以獲取關於輸入的有用信息。第一個是 NF 變量，也叫做「字段數量」變量。awk 會自動將該變量設置成當前記錄中的字段數量。可以使用 NF 變量來只顯示某些輸入行：

```
NF == 3 { print "this particular record has three fields: " $0 }
```

當然，也可以在條件語句中使用 NF 變量，如下：

```
{
if ( NF > 2 ) {
print $1 " " $2 ":" $3
}
}
```

記錄號
記錄號 (NR) 是另一個方便的變量。它始終包含當前記錄的編號 (awk 將第一個記錄算作記錄號 1) 。迄今為止，我們已經處理了每一行包含一個記錄的輸入文件。對於這些情況，NR 還會告訴您當前行號。然而，當我們在本系列以後部分中開始處理多行記錄時，就不會再有這種情況，所以要注意！可以像使用 NF 變量一樣使用 NR 來只打印某些輸入行：

```
(NR < 10 ) || (NR > 100) { print "We are on record number 1-9 or 101+" }
```

另一個示例：

```
{
#skip header
if ( NR > 10 ) {
print "ok, now for the real information!"
}
}
AWK SHELL FOR LINUX
[ 2006-6-23 14:06:00 | By: nathena ]
```

多行記錄
awk 是一種用於讀取和處理結構化數據 (如系統的 /etc/passwd 文件) 的極佳工具。/etc/passwd 是 UNIX 用戶數據庫，並且是用冒號定界的文本文件，它包含許多重要信息，包括所有現有用戶帳戶和用戶標識，以及其它信息。在我的前一篇文章中，我演示了 awk 如何輕鬆地分析這個文件。我們只須將 FS (字段分隔符) 變量設置成 ":"。

正確設置了 FS 變量之後，就可以將 awk 配置成分析幾乎任何類型的結構化數據，只要這些數據是每行一個記錄。然而，如果要分析佔據多行的記錄，僅僅依靠設置 FS 是不夠的。在這些情況下，我們還需要修改 RS 記錄分隔符變量。RS 變量告訴 awk 當前記錄什麼時候結束，新記錄什麼時候開始。

譬如，讓我們討論一下如何完成處理「聯邦證人保護計劃」所涉及人員的地址列表的任務：


```
Jimmy the Weasel
100 Pleasant Drive
San Francisco, CA 12345
```

Big Tony
200 Incognito Ave.
Suburbia, WA 67890

理論上，我們希望 awk 將每 3 行看作是一個獨立的記錄，而不是三個獨立的記錄。如果 awk 將地址的第一行看作是第一個字段 ($1)，街道地址看作是第二個字段 ($2)，城市、州和郵政編碼看作是第三個字段 $3，那麼這個代碼就會變得很簡單。以下就是我們想要得到的代碼：

```
BEGIN {
FS="\n"
RS=""
}
```

在上面這段代碼中，將 FS 設置成 "\n" 告訴 awk 每個字段都佔據一行。通過將 RS 設置成 ""，還會告訴 awk 每個地址記錄都由空白行分隔。一旦 awk 知道是如何格式化輸入的，它就可以為我們執行所有分析工作，腳本的其餘部分很簡單。讓我們研究一個完整的腳本，它將分析這個地址列表，並將每個記錄打印在一行上，用逗號分隔每個字段。

address.awk

```
BEGIN {
FS="\n"
RS=""
}

{
print $1 ", " $2 ", " $3

}
```

如果這個腳本保存為 address.awk，地址數據存儲在文件 address.txt 中，可以通過輸入 "awk -f address.awk address.txt" 來執行這個腳本。此代碼將產生以下輸出：

```
Jimmy the Weasel, 100 Pleasant Drive, San Francisco, CA 12345
Big Tony, 200 Incognito Ave., Suburbia, WA 67890
```

OFS 和 ORS
在 address.awk 的 print 語句中，可以看到 awk 會連接 (合併) 一行中彼此相鄰的字符串。我們使用此功能在同一行上的三個字段之間插入一個逗號和空格 (", ")。這個方法雖然有用，但比較難看。與其在字段間插入 ", " 字符串，倒不如讓通過設置一個特殊 awk 變量 OFS，讓 awk 完成這件事。請參考下面這個代碼片斷。

```
print "Hello", "there", "Jim!"
```

這行代碼中的逗號並不是實際文字字符串的一部分。事實上，它們告訴 awk "Hello"、"there" 和 "Jim!" 是單獨的字段，並且應該在每個字符串之間打印 OFS 變量。缺省情況下，awk 產生以下輸出：

```
Hello there Jim!
```

這是缺省情況下的輸出結果，OFS 被設置成 " "，單個空格。不過，我們可以方便地重新定義 OFS，這樣 awk 將插入我們中意的字段分隔符。以下是原始 address.awk 程序的修訂版，它使用 OFS 來輸出那些中間的 ", " 字符串：

address.awk 的修訂版

```
BEGIN {
FS="\n"
RS=""
OFS=", "
}

{
print $1, $2, $3

}
```

awk 還有一個特殊變量 ORS，全稱是「輸出記錄分隔符」。通過設置缺省為換行 ("\n") 的 OFS，我們可以控制在 print 語句結尾自動打印的字符。缺省 ORS 值會使 awk 在新行中輸出每個新的 print 語句。如果想使輸出的間隔翻倍，可以將 ORS 設置成 "\n\n"。或者，如果想要用單個空格分隔記錄 (而不換行)，將 ORS 設置成 ""。

將多行轉換成用 tab 分隔的格式
假設我們編寫了一個腳本，它將地址列表轉換成每個記錄一行，且用 tab 定界的格式，以便導入電子錶格。使用稍加修改的 address.awk 之後，就可以清楚地看到這個程序只適合於三行的地址。如果 awk 遇到以下地址，將丟掉第四行，並且不打印該行：

Cousin Vinnie
Vinnie's Auto Shop
300 City Alley
Sosueme, OR 76543

要處理這種情況，代碼最好考慮每個字段的記錄數量，並依次打印每個記錄。現在，代碼只打印地址的前三個字段。以下就是我們想要的一些代碼：

適合具有任意多字段的地址的 address.awk 版本

```
BEGIN {
FS="\n"
RS=""
ORS=""
}

{
x=1
while ( x<NF ) {
print $x "\t"
x++
}
print $NF "\n"
}
```

首先，將字段分隔符 FS 設置成 "\n"，將記錄分隔符 RS 設置成 ""，這樣 awk 可以像以前一樣正確分析多行地址。然後，將輸出記錄分隔符 ORS 設置成 ""，它將使 print 語句在每個調用結尾不輸出新行。這意味著如果希望任何文本從新的一行開始，那麼需要明確寫入 print "\n"。

在主代碼塊中，創建了一個變量 x 來存儲正在處理的當前字段的編號。起初，它被設置成 1。然後，我們使用 while 循環（一種 awk 循環結構，等同於 C 語言中的 while 循環），對於所有記錄（最後一個記錄除外）重複打印記錄和 tab 字符。最後，打印最後一個記錄和換行；此外，由於將 ORS 設置成 ""，print 將不輸出換行。程序輸出如下，這正是我們所期望的：

我們想要的輸出。不算漂亮，但用 tab 定界，以便於導入電子錶格

Jimmy the Weasel 100 Pleasant Drive San Francisco, CA 12345
Big Tony 200 Incognito Ave. Suburbia, WA 67890
Cousin Vinnie Vinnie's Auto Shop 300 City Alley Sosueme, OR 76543

循環結構
我們已經看到了 awk 的 while 循環結構，它等同於相應的 C 語言 while 循環。awk 還有 "do...while" 循環，它在代碼塊結尾處對條件求值，而不像標準 while 循環那樣在開始處求值。它類似於其它語言中的 "repeat...until" 循環。以下是一個示例：

do...while 示例

```
{
count=1
do {
print "I get printed at least once no matter what"
} while ( count != 1 )
}
```

與一般的 while 循環不同，由於在代碼塊之後對條件求值，"do...while" 循環永遠都至少執行一次。換句話說，當第一次遇到普通 while 循環時，如果條件為假，將永遠不執行該循環。

for 循環
awk 允許創建 for 循環，它就像 while 循環，也等同於 C 語言的 for 循環：

```
for ( initial assignment; comparison; increment ) {
code block
```

以下是一個簡短示例：

```
for ( x = 1; x <= 4; x++ ) {
print "iteration",x
}
```

此段代碼將打印：

```
iteration 1
iteration 2
iteration 3
iteration 4
```

break 和 continue
此外，如同 C 語言一樣，awk 提供了 break 和 continue 語句。使用這些語句可以更好地控制 awk 的循環結構。以下是迫切需要 break 語句的代碼片斷：

while 死循環

```
while (1) {
print "forever and ever..."
}
```

因為 1 永遠代表是真，這個 while 循環將永遠運行下去。以下是一個只執行十次的循環：

break 語句示例

```
x=1
while(1) {
print "iteration",x
if ( x == 10 ) {
break
}
x++
}
```

這裡，break 語句用於「逃出」最深層的循環。"break" 使循環立即終止，並繼續執行循環代碼塊後面的語句。

continue 語句補充了 break，其作用如下：

```
x=1
while (1) {
if ( x == 4 ) {
x++
continue
}
print "iteration",x
if ( x > 20 ) {
break
}
x++
}
```

這段代碼打印 "iteration 1" 到 "iteration 21"，"iteration 4" 除外。如果迭代等於 4，則增加 x 並調用 continue 語句，該語句立即使 awk 開始執行下一個循環迭代，而不執行代碼塊的其餘部分。如同 break 一樣，continue 語句適合各種 awk 迭代循環。在 for 循環主體中使用時，continue 將使循環控制變量自動增加。以下是一個等價循環：

```
for ( x=1; x<=21; x++ ) {
if ( x == 4 ) {
continue
}
print "iteration",x
}
```

在 while 循環中時，在調用 continue 之前沒有必要增加 x，因為 for 循環會自動增加 x。

數組

如果您知道 awk 可以使用數組，您一定會感到高興。然而，在 awk 中，數組下標通常從 1 開始，而不是 0：

```
myarray[1]="jim"
myarray[2]=456
```

awk 遇到第一個賦值語句時，它將創建 myarray，並將元素 myarray[1] 設置成 "jim"。執行了第二個賦值語句後，數組就有兩個元素了。

數組迭代

定義之後，awk 有一個便利的機制來迭代數組元素，如下所示：

```
for ( x in myarray ) {
print myarray[x]
}
```

這段代碼將打印數組 myarray 中的每一個元素。當對於 for 使用這種特殊的 "in" 形式時，awk 將 myarray 的每個現有下標依次賦值給 x (循環控制變量)，每次賦值以後都循環一次循環代碼。雖然這是一個非常方便的 awk 功能，但它有一個缺點 -- 當 awk 在數組下標之間輪轉時，它不會依照任何特定的順序。那就意味著我們不能知道以上代碼的輸出是：

```
jim
456
```

還是

```
456
jim
```

套用 Forrest Gump 的話來說，迭代數組內容就像一盒巧克力 -- 您永遠不知道將會得到什麼。因此有必要使 awk 數組「字符串化」，我們現在就來研究這個問題。

數組下標字符串化

在我的前一篇文章中，我演示了 awk 實際上以字符串格式來存儲數字值。雖然 awk 要執行必要的轉換來完成這項工作，但它卻可以使用某些看起來很奇怪的代碼：

```
a="1"
b="2"
c=a+b+3
```

執行了這段代碼後，c 等於 6。由於 awk 是「字符串化」的，添加字符串 "1" 和 "2" 在功能上並不比添加數字 1 和 2 難。這兩種情況下，awk 都可以成功執行運算。awk 的「字符串化」性質非常可愛 -- 您可能想要知道如果使用數組的字符串下標會發生什麼情況。例如，使用以下代碼：

```
myarr["1"]="Mr. Whipple"
print myarr["1"]
```

可以預料，這段代碼將打印 "Mr. Whipple"。但如果去掉第二個 "1" 下標中的引號，情況又會怎樣呢？

```
myarr["1"]="Mr. Whipple"
print myarr[1]
```

猜想這個代碼片斷的結果比較難。awk 將 myarr["1"] 和 myarr[1] 看作數組的兩個獨立元素，還是它們是指同一個元素？答案是它們指的是同一個元素，awk 將打印 "Mr. Whipple"，如同第一個代碼片斷一樣。雖然看上去可能有點怪，但 awk 在幕後卻一直使用數組的字符串下標！

瞭解了這個奇怪的真相之後，我們中的一些人可能想要執行類似於以下的古怪代碼：

```
myarr["name"]="Mr. Whipple"
print myarr["name"]
```

這段代碼不僅不會產生錯誤，而且它的功能與前面的示例完全相同，也將打印 "Mr. Whipple"！可以看到，awk 並沒有限制我們使用純整數下標；如果我們願意，可以使用字符串下標，而且不會產生任何問題。只要我們使用非整數數組下標，如 myarr["name"]，那麼我們就在使用關聯數組。從技術上講，如果我們使用字符串下標，awk 的後台操作並沒有什麼不同

（因為即便使用「整數」下標，awk 還是會將它看作是字符串）。但是，應該將它們稱作關聯數組 -- 它聽起來很酷，而且會給您的上司留下印象。字符串化下標是我們的小秘密。

數組工具

談到數組時，awk 給予我們許多靈活性。可以使用字符串下標，而且不需要連續的數字序列下標（例如，可以定義 myarr[1] 和 myarr[1000]，但不定義其它所有元素）。雖然這些都很有用，但在某些情況下，會產生混淆。幸好，awk 提供了一些實用功能有助於使數組變得更易於管理。

首先，可以刪除數組元素。如果想要刪除數組 fooarray 的元素 1，輸入：

```
delete fooarray[1]
```

而且，如果想要查看是否存在某個特定數組元素，可以使用特殊的 "in" 布爾運算符，如下所示：

```
if ( 1 in fooarray ) {
print "Ayep! It's there."
} else {
print "Nope! Can't find it."
}
```

AWK SHELL FOR LINUX
[2006-6-23 14:07:00 | By: nathena]

格式化輸出

雖然大多數情況下 awk 的 print 語句可以完成任務，但有時我們還需要更多。在那些情況下，awk 提供了兩個我們熟知的老朋友 printf() 和 sprintf()。是的，如同其它許多 awk 部件一樣，這些函數等同於相應的 C 語言函數。printf() 會將格式化字符串打印到 stdout，而 sprintf() 則返回可以賦值給變量的格式化字符串。如果不熟悉 printf() 和 sprintf()，介紹 C 語言的文章可以讓您迅速瞭解這兩個基本打印函數。在 Linux 系統上，可以輸入 "man 3 printf" 來查看 printf() 幫助頁面。

以下是一些 awk sprintf() 和 printf() 的樣本代碼。可以看到，它們幾乎與 C 語言完全相同。

```
x=1
b="foo"
printf("%s got a %d on the last test\n","Jim",83)
myout=("%s-%d",b,x)
print myout
```

此代碼將打印：

```
Jim got a 83 on the last test
foo-1
```

字符串函數

awk 有許多字符串函數，這是件好事。在 awk 中，確實需要字符串函數，因為不能像在其它語言（如 C、C++ 和 Python）中那樣將字符串看作是字符數組。例如，如果執行以下代碼：

```
mystring="How are you doing today?"
print mystring[3]
```

將會接收到一個錯誤，如下所示：

```
awk: string.gawk:59: fatal: attempt to use scalar as array
```

噢，好吧。雖然不像 Python 的序列類型那樣方便，但 awk 的字符串函數還是可以完成任務。讓我們來看一下。

首先，有一個基本 length() 函數，它返回字符串的長度。以下是它的使用方法：

```
print length(mystring)
```

此代碼將打印值：

24

好，繼續。下一個字符串函數叫作 index，它將返回子字符串在另一個字符串中出現的位置，如果沒有找到該字符串則返回 0。使用 mystring，可以按以下方法調用它：

```
print index(mystring,"you")
```

awk 會打印：

9

讓我們繼續討論另外兩個簡單的函數，tolower() 和 toupper()。與您猜想的一樣，這兩個函數將返回字符串並且將所有字符分別轉換成小寫或大寫。請注意，tolower() 和 toupper() 返回新的字符串，不會修改原來的字符串。這段代碼：

```
print tolower(mystring)
print toupper(mystring)
print mystring
```

……將產生以下輸出：

```
how are you doing today?
HOW ARE YOU DOING TODAY?
How are you doing today?
```

到現在為止一切不錯，但我們究竟如何從字符串中選擇子串，甚至單個字符？那就是使用 substr() 的原因。以下是 substr() 的調用方法：

```
mysub=substr(mystring,startpos,maxlen)
```

mystring 應該是要從中抽取子串的字符串變量或文字字符串。startpos 應該設置成起始字符位置，maxlen 應該包含要抽取的字符串的最大長度。請注意，我說的是最大長度；如果 length(mystring) 比 startpos+maxlen 短，那麼得到的結果就會被截斷。substr() 不會修改原始字符串，而是返回子串。以下是一個示例：

```
print substr(mystring,9,3)
```

awk 將打印：

you

如果您通常用於編程的語言使用數組下標訪問部分字符串（以及不使用這種語言的人），請記住 substr() 是 awk 代替方法。需要使用它來抽取單個字符和子串；因為 awk 是基於字符串的語言，所以會經常用到它。

現在，我們討論一些更耐人尋味的函數，首先是 match()。match() 與 index() 非常相似，它與 index() 的區別在於它並不搜索子串，它搜索的是規則表達式。match() 函數將返回匹配的起始位置，如果沒有找到匹配，則返回 0。此外，match() 還將設置兩個變量，叫作 RSTART 和 RLENGTH。RSTART 包含返回值（第一個匹配的位置），RLENGTH 指定它佔據的字符跨度（如果沒有找到匹配，則返回 -1）。通過使用 RSTART、RLENGTH、substr() 和一個小循環，可以輕鬆地迭代字符串中的每個匹配。以下是一個 match() 調用示例：

```
print match(mystring,/you/), RSTART, RLENGTH
```

awk 將打印：

9 9 3

字符串替換

現在，我們將研究兩個字符串替換函數，sub() 和 gsub()。這些函數與目前已經討論過的函數略有不同，因為它們確實修改原始字符串。以下是一個模板，顯示了如何調用 sub()：

```
sub(regexp,replstring,mystring)
```

調用 sub() 時，它將在 mystring 中匹配 regexp 的第一個字符序列，並且用 replstring 替換該序列。sub() 和 gsub() 用相同的自變量；唯一的區別是 sub() 將替換第一個 regexp 匹配（如果有的話），gsub() 將執行全局替換，換出字符串

```
sub(/o/,"O",mystring)
print mystring
mystring="How are you doing today?"
gsub(/o/,"O",mystring)
print mystring
```

必須將 mystring 復位成其初始值，因為第一個 sub() 調用直接修改了 mystring。在執行時，此代碼將使 awk 輸出：

```
HOw are you doing today?
HOw are yOu dOing tOday?
```

當然，也可以是更複雜的規則表達式。我把測試一些複雜規則表達式的任務留給您來完成。

通過介紹函數 split()，我們來匯總一下已討論過的函數。split() 的任務是「切開」字符串，並將各部分放到使用整數下標的數組中。以下是一個 split() 調用示例：

```
numelements=split("Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec",mymonths,",")
```

調用 split() 時，第一個自變量包含要切開文字字符串或字符串變量。在第二個自變量中，應該指定 split() 將填入片段部分的數組名稱。在第三個元素中，指定用於切開字符串的分隔符。split() 返回時，它將返回分割的字符串元素的數量。split() 將每一個片段賦值給下標從 1 開始的數組，因此以下代碼：

```
print mymonths[1],mymonths[numelements]
```

……將打印：

```
Jan Dec
```

特殊字符串形式
簡短註釋 -- 調用 length()、sub() 或 gsub() 時，可以去掉最後一個自變量，這樣 awk 將對 $0（整個當前行）應用函數調用。要打印文件中每一行的長度，使用以下 awk 腳本：

```
{
print length()
}
```

財務上的趣事
幾星期前，我決定用 awk 編寫自己的支票簿結算程序。我決定使用簡單的 tab 定界文本文件，以便於輸入最近的存款和提款記錄。其思路是將這個數據交給 awk 腳本，該腳本會自動合計所有金額，並告訴我餘額。以下是我決定如何將所有交易記錄到 "ASCII checkbook" 中：

```
23 Aug 2000 food - - Y Jimmy's Buffet 30.25
```

此文件中的每個字段都由一個或多個 tab 分隔。在日期（字段 1，$1）之後，有兩個字段叫做「費用分類帳」和「收入分類帳」。以上面這行為例，輸入費用時，我在費用字段中放入四個字母的別名，在收入字段中放入 "-"（空白項）。這表示這一特定項是「食品費用」。 以下是存款的示例：

```
23 Aug 2000 - inco - Y Boss Man 2001.00
```

在這個實例中，我在費用分類帳中放入 "-"（空白），在收入分類帳中放入 "inco"。"inco" 是一般（薪水之類）收入的別名。使用分類帳別名讓我可以按類別生成收入和費用的明細分類帳。至於記錄的其餘部分，其它所有字段都是不需加以說明的。「是否付清？」字段（"Y" 或 "N"）記錄了交易是否已過帳到我的帳戶；除此之外，還有一個交易描述，和一個正的美元金額。

用於計算當前餘額的算法不太難。awk 只需要依次讀取每一行。如果列出了費用分類帳，但沒有收入分類帳（為 "-"），那麼這一項就是借方。如果列出了收入分類帳，但沒有費用分類帳（為 "-"），那麼這一項就是貸方。而且，如果同時列出了費用和收入分類帳，那麼這個金額就是「分類帳轉帳」；即，從費用分類帳減去美元金額，並將此金額添加到收入分類帳。此外，所有這些分類帳都是虛擬的，但對於跟蹤收入和支出以及預算卻非常有用。

代碼
現在該研究代碼了。我們將從第一行（BEGIN 塊和函數定義）開始：

```
balance：第 1 部分
```

```
#!/usr/bin/env awk -f
BEGIN {
FS="\t+"
months="Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec"
}

function monthdigit(mymonth) {
return (index(months,mymonth)+3)/4
}
```

首先執行 "chmod +x myscript" 命令，那麼將第一行 "#!..." 添加到任何 awk 腳本將使它可以直接從 shell 中執行。其餘行定義了 BEGIN 塊，在 awk 開始處理支票簿文件之前將執行這個代碼塊。我們將 FS（字段分隔符）設置成 "\t+"，它會告訴 awk 字段由一個或多個 tab 分隔。另外，我們定義了字符串 months，下面將出現的 monthdigit() 函數將使用它。

最後三行顯示了如何定義自己的 awk 。格式很簡單 -- 輸入 "function"，再輸入名稱，然後在括號中輸入由逗號分隔的參數。在此之後，"{ }" 代碼塊包含了您希望這個函數執行的代碼。所有函數都可以訪問全局變量（如 months 變量）。另外，awk 提供了 "return" 語句，它允許函數返回一個值，並執行類似於 C 和其它語言中 "return" 的操作。這個特定函數將以 3 個字母字符串格式表示的月份名稱轉換成等價的數值。例如，以下代碼：

```
print monthdigit("Mar")
```

……將打印：

```
3
```

現在，讓我們討論其它一些函數。

財務函數
以下是其它三個執行簿記的函數。我們即將見到的主代碼塊將調用這些函數之一，按順序處理支票簿文件的每一行，從而將相應交易記錄到 awk 數組中。有三種基本交易，貸方 (doincome)、借方 (doexpense) 和轉帳 (dotransfer)。您會發現這三個函數全都接受一個自變量，叫作 mybalance。mybalance 是二維數組的一個佔位符，我們將它作為自變量進行傳遞。目前，我們還沒有處理過二維數組；但是，在下面可以看到，語法非常簡單。只須用逗號分隔每一維就行了。

我們將按以下方式將信息記錄到 "mybalance" 中。數組的第一維從 0 到 12，用於指定月份，0 代表全年。第二維是四個字母的分類帳，如 "food" 或 "inco"；這是我們處理的真實分類帳。因此，要查找全年食品分類帳的餘額，應查看 mybalance[0,"food"]。要查找 6 月的收入，應查看 mybalance[6,"inco"]。

balance，第 2 部分

```
function doincome(mybalance) {
mybalance[curmonth,$3] += amount
mybalance[0,$3] += amount
}

function doexpense(mybalance) {
mybalance[curmonth,$2] -= amount
mybalance[0,$2] -= amount
}

function dotransfer(mybalance) {
mybalance[0,$2] -= amount
mybalance[curmonth,$2] -= amount
mybalance[0,$3] += amount
mybalance[curmonth,$3] += amount
}
```

調用 doincome() 或任何其它函數時，我們將交易記錄到兩個位置 -- mybalance[0,category] 和 mybalance[curmonth, category]，它們分別表示全年的分類帳餘額和當月的分類帳餘額。這讓我們稍後可以輕鬆地生成年度或月度收入 / 支出明細分類帳。

如果研究這些函數，將發現在我的引用中傳遞了 mybalance 引用的數組。另外，我們還引用了幾個全局變量：curmonth，它保存了當前記錄所屬的月份的數值，$2（費用分類帳），$3（收入分類帳）和金額（$7，美元金額）。調

主塊

以下是主代碼塊，它包含了分析每一行輸入數據的代碼。請記住，由於正確設置了 FS，可以用 $ 1 引用第一個字段，用 $2 引用第二個字段，依次類推。調用 doincome() 和其它函數時，這些函數可以從函數內部訪問 curmonth、$2、$3 和金額的當前值。請先研究代碼，在代碼之後可以見到我的說明。

balance，第 3 部分

```
{
curmonth=monthdigit(substr($1,4,3))
amount=$7

#record all the categories encountered
if ( $2 != "-" )
globcat[$2]="yes"
if ( $3 != "-" )
globcat[$3]="yes"

#tally up the transaction properly
if ( $2 == "-" ) {
if ( $3 == "-" ) {
print "Error: inc and exp fields are both blank!"
exit 1
} else {
#this is income
doincome(balance)
if ( $5 == "Y" )
doincome(balance2)
}
} else if ( $3 == "-" ) {
#this is an expense
doexpense(balance)
if ( $5 == "Y" )
doexpense(balance2)
} else {
#this is a transfer
dotransfer(balance)
if ( $5 == "Y" )
dotransfer(balance2)
}
}
```

在主塊中，前兩行將 curmonth 設置成 1 到 12 之間的整數，並將金額設置成字段 7（使代碼易於理解）。然後，是四行有趣的代碼，它們將值寫到數組 globcat 中。globcat，或稱作全局分類帳數組，用於記錄在文件中遇到的所有分類帳 -- "inco"、"misc"、"food"、"util" 等。例如，如果 $2 == "inco"，則將 globcat["inco"] 設置成 "yes"。稍後，我們可以使用簡單的 "for (x in globcat)" 循環來迭代分類帳列表。

在接著的大約二十行中，我們分析字段 $2 和 $3，並適當記錄交易。如果 $2=="-" 且 $3!="-"，表示我們有收入，因此調用 doincome()。如果是相反的情況，則調用 doexpense()；如果 $2 和 $3 都包含分類帳，則調用 dotransfer()。每次我們都將 "balance" 數組傳遞給這些函數，從而在這些函數中記錄適當的數據。

您還會發現幾行代碼說「if ($5 == "Y")，那麼將同一個交易記錄到 balance2 中」。我們在這裡究竟做了些什麼？您將回憶起 $5 包含 "Y" 或 "N"，並記錄交易是否已經過帳到帳戶。由於僅當過帳了交易時我們才將交易記錄到 balance2，因此 balance2 包含了真實的帳戶餘額，而 "balance" 包含了所有交易，不管是否已經過帳。可以使用 balance2 來驗證數據項（因為它應該與當前銀行帳戶餘額匹配），可以使用 "balance" 來確保沒有透支帳戶（因為它會考慮您開出的尚未兌現的所有支票）。

生成報表

主塊重複處理了每一行記錄之後，現在我們有了關於比較全面的、按分類帳和按月份劃分的借方和貸方記錄。現在，在這種情況下最合適的做法是只須定義生成報表的 END 塊：

balance，第 4 部分

```
END {
bal=0
```

```
for (x in globcat) {
bal=bal+balance[0,x]
bal2=bal2+balance2[0,x]
}
printf("Your available funds: %10.2f\n", bal)
printf("Your account balance: %10.2f\n", bal2)
}
```

這個報表將打印出匯總，如下所示：

```
Your available funds:1174.22
Your account balance:2399.33
```

在 END 塊中，我們使用 "for (x in globcat)" 結構來迭代每一個分類帳，根據記錄在案的交易結算主要餘額。實際上，我們結算兩個餘額，一個是可用資金，另一個是帳戶餘額。要執行程序並處理您在文件 "mycheckbook.txt" 中輸入的財務數據，將以上所有代碼放入文本文件 "balance"，執行 "chmod +x balance"，然後輸入 "./balance mycheckbook.txt"。然後 balance 腳本將合計所有交易，打印出兩行餘額匯總。

升級

我使用這個程序的更高級版本來管理我的個人和企業財務。我的版本（由於篇幅限制不能在此涵蓋）會打印出收入和費用的月度明細分類帳，包括年度總合、淨收入和其它許多內容。它甚至以 HTML 格式輸出數據，因此我可以在 Web 瀏覽器中查看它。 如果您認為這個程序有用，我建議您將這些特性添加到這個腳本中。不必將它配置成要 記錄任何附加信息；所需的全部信息已經在 balance 和 balance2 裡面了。只要升級 END 塊就萬事具備了！

本文来自CSDN博客，转载请标明出处：http://blog.csdn.net/weekly123/archive/2006/12/28/1465675.aspx

編 / Xuite日誌 / 回應(0) / 引用(22)
case 在shell的用法|日誌首頁|linux cut 指令

引用

本文的引用網址 https://blog.xuite.net/mb1016.flying/linux/28111008/track 複製 我要引用

2016-09-11 17:54:59
引用文章：brazilian body wav
引用日誌：brazilian body wav
文章摘要：This girl was so very interested to spread out her brazilian body wav relating to X-mas evening. We ...

2014-11-14 23:39:18
引用文章：188金宝博现场滚球,188bet金宝博滚球投注官方网站备用网址_228238.com
引用日誌：188金宝博现场滚球,188bet金宝博滚球投注官方网站备用网址_228238.com
文章摘要：要睡觉了呵呵...

2014-11-11 15:28:14
引用文章：调情香水
引用日誌：调情香水
文章摘要：我纵横网络多年,自以为再也不会有任何能打动我,没想到今天看到了如此精妙绝伦的这样一篇帖子。...

2014-11-11 08:49:18
引用文章：cheap ugg boots
引用日誌：cheap ugg boots
文章摘要：http://earlyeducationforum.orgsuperior singing method review...

2014-11-06 09:30:29
引用文章：cheap ugg boots
引用日誌：cheap ugg boots
文章摘要：http://www.freemasons-freemasonry.com/video/ugg0.aspcheap ugg boots...

2014-10-23 06:27:17
引用文章：venus factor reviews
引用日誌：venus factor reviews
文章摘要：http://donorcents.orgthe venus factor reviews...

2014-09-28 16:22:16
引用文章：太阳城百家乐路单
引用日誌：太阳城百家乐路单
文章摘要：1、这场爱恋，如烟花般在心的荒芜中绚烂，尽管短暂，但那瞬间的美丽，足...

2014-09-18 11:48:52
引用文章：澳门娱乐城
引用日誌：澳门娱乐城
文章摘要：自己就如同那鸡肋一般，嚼之无味，弃之可惜。...

2014-07-27 14:51:54
引用文章：皇冠备用足球投注
引用日誌：皇冠备用足球投注

2014-06-26 00:25:40
引用文章：皇冠现金
引用日誌：皇冠现金
文章摘要：娱乐圈从来不缺俊男美女，...

2014-06-20 18:28:57
引用文章：娱乐城开户送钱求指点
引用日誌：娱乐城开户送钱求指点
文章摘要：虽然他们当中一些人看上去极其"不着调"以致闹出不少笑话，...

2014-06-19 22:11:52
引用文章：六合神童
引用日誌：六合神童
文章摘要：但是在9月14日第12轮的补赛中，面对实力绝对弱于自己的青岛中能队，山东?...

2014-06-19 03:58:37
引用文章：fuck google
引用日誌：fuck google
文章摘要：Drop dead google, fuck Googleï¼ŒSFSsafiji2142...

2014-06-18 18:45:42
引用文章：送金娱乐城
引用日誌：送金娱乐城
文章摘要：但好在他们都表示还凑合...

2014-06-18 14:31:52
引用文章：博彩平台
引用日誌：博彩平台
文章摘要：而事情却并非如此简单。在《人祸摧花》一文中，基本上看不到确切的事实?...

2014-06-18 02:21:05
引用文章：fuck google
引用日誌：fuck google
文章摘要：the son of bitch google, fuck Google...

2014-06-17 10:34:50
引用文章：fuck google
引用日誌：fuck google
文章摘要：mind you own business google, fuck Google...

2014-06-16 18:50:54
引用文章：fuck google
引用日誌：fuck google
文章摘要：You''re an asshole google, fuck Google...

2014-06-14 14:39:40
引用文章：toms Norge
引用日誌：toms Norge
文章摘要：Does your site have a contact page? I''m having problems locating it but, I''d like to send you an e...

2014-06-06 20:38:31
引用文章：jordan shop italia
引用日誌：jordan shop italia
文章摘要：Hiya. Sorry to trouble you but I ran across your website and noticed you are using the exact same th...

2014-05-23 08:56:30
引用文章：jordan carmine 6s
引用日誌：jordan carmine 6s
文章摘要：Below this you have links to fun things like Daily Ticket...

2014-05-06 17:26:57
引用文章：乐天堂官网
引用日誌：乐天堂官网

回應