Introduzione ad AWK

Davide Di Vaia, Pietro Laface

AWK è un filtro generico per file di testo che permette di trovare sequenze di caratteri in file di testo e di effettuare una serie di azioni sulle linee corrispondenti usando comandi espressi con un linguaggio molto simile al linguaggio C.

La sintassi della riga di comando è la seguente:

awk 'script' nomefile

awk –f fileprogramma nomefile

AWK elabora il file *nomefile* secondo le istruzioni contenute in *'script'* oppure nel *fileprogramma.*

Si può usare AWK anche come filtro:

comando | awk 'script'

comando | awk –f fileprogramma

AWK elabora i file di testo una riga alla volta, eseguendo azioni diverse a seconda del contenuto della riga. I due fondamenti su cui si basa AWK sono:

1) La struttura dei programmi:

pattern { azione }

pattern { azione}

……

Quando il *pattern* è soddisfatto viene eseguita l'*azione.*

2) La suddivisione dei file di testo in campi (fields) e linee (records).

Ciascun record rappresenta una linea del file e ciascun campo una "parola" (i campi sono divisi tra loro dal carattere contenuto nella variabile **FS** che di default è lo spazio).

I pattern possono essere delle semplici espressioni regolari racchiuse tra "/" (es: /pippo/) oppure un'espressione logica o ancora le espressioni **BEGIN** ed **END** (vengono ritenute vere rispettivamente prima di incominciare a leggere il file in input e dopo averlo esaminato tutto).

AWK legge una ad una le righe del file e se una riga contiene il pattern specificato viene eseguita l'azione associata.

Le azioni non sono altro che dei piccoli programmi C-like. È importante notare come le variabili

non debbano essere dichiarate (a differenza del C) e vengano automaticamente inizializzate a zero oppure alla stringa nulla.

Per indicare i campi (parole) della riga corrente si usano le variabili **$0,$1,$2**,….; la variabile **$0** contiene l'intera riga (record) mentre **$1** contiene il primo campo, **$2** il secondo e così via.

Una istruzione AWK appartiene ai seguenti tipi:

- **Assegnazione:** ***var = exp*** dove ***exp*** calcola il valore di un'espressione e lo assegna alla variabile ***var*** (es: ***doppio = pluto * 2***).

- **Statement if :** **if(exp)statement1 [else statement2]** dove se **exp** è diverso da zero viene eseguito **statement1,** altrimenti **statement2.**

- **Ciclo while:** **while(exp)statement** dove **statement** viene eseguito finchè **exp** continua ad avere un valore diverso da zero.

- **Ciclo for:** **for(exp1;exp2;exp3) statement** dove **exp1** è eseguita al momento dell'inizializzazione del ciclo, **exp3** viene eseguita all'inizio di ogni ciclo e **exp2** fa si che si esca dal ciclo quando diventa falsa.

- **Ciclo for in:** **for(var in arrayname)statement** simile al ciclo for della shell, fa sì che alla variabile **var** vengano assegnati ad uno ad uno i valori contenuti nel vettore (unidimensionale) **arrayname.**

- **Stampa: `print exp,[exp,…,exp]`** in cui ogni espressione **`exp`** viene calcolata e stampata nello standard output. I valori delle varie **`exp`** saranno distanziati dal carattere contenuto nella variabile **`OFS`** che di default è lo spazio. Se **`print`** viene usata senza **`exp`** viene eseguita la print **`$0`**.

  Notare che è possibile redirigere l'output su file nel seguente modo:

  ```
  print "Ciao","a","tutti" > ciao.txt
  ```

  scrive nel file **`ciao.txt`** la frase "**`Ciao a tutti`**" seguita dal newline (ammesso che OFS contenga uno spazio).

  ```
  print "Ciao","a","tutti" >> ciao.txt
  ```

  scrive invece in modo append (come nella shell); se il file non esiste viene creato.

- **`printf(formato,[exp,…exp])`** che è praticamente uguale alla **`printf`** del C

esistono poi le istruzioni

- **`break`** (esce dal ciclo while o for attivo)
- **`continue`** (fa partire l'iterazione seguente del ciclo **`while`** o **`for`** ignorando le istruzioni rimanenti del ciclo)
- **`next`** (salta le istruzioni rimanenti del programma AWK) ed infine **`exit`** che fa terminare immediatamente AWK.

Oltre alle variabili **`FS`** e **`OFS`**, in AWK esistono altre variabili che vengono aggiornate automaticamente durante l'elaborazione del file in input:

- **`NF`**: Numero dei campi della riga correntemente elaborata.
- **`NR`**: Numero della riga correntemente elaborata.
- **`FILENAME`**: Nome del file correntemente elaborato.Questa variabile è indefinita all'interno del blocco **`BEGIN`** e contiene "-" se non sono specificati file nella linea di comando.

Esaminiamo qualche semplice esempio:

`cat elenco.txt| awk '/Luca/ {print $3}'`

stampa il terzo campo di tutte le righe del file **`elenco.txt`** che contengono la parola **`Luca`** (stampa una riga vuota se il terzo campo della riga è vuoto).

`cat elenco.txt| awk '/Luca/ {print}'`

stampa tutte le righe del file **`elenco.txt`** che contengono la parola **`Luca`** (**`print`** equivale a **`print $0`**)

`awk 'BEGIN{FS=":"} ($2 == "OFF") {print $3,$1}'/etc/passwd`

stampa lo username e l'UID di tutti gli utenti del sistema che sono senza password

`awk '/main()/{print FILENAME}' *.c`

Stampa il nome di tutti i file con estensione **`.c`** che contengono la funzione **`main()`**.

Proviamo ora per esercizio a simulare il comportamento di alcuni semplici comandi Unix:

`cat`

`awk '{print}'` stampa l'intero file (si ricorda che **`print`** e **`print $0`** sono identiche)

`cat –n`

`awk '{print NR,$0}'` stampa l'intero file includendo i numeri di riga

`wc –l`

`awk 'END {print NR}'` stampa il numero di righe del file

Si possono anche creare comandi interessanti come:

Cercare il valore massimo contenuto di una determinata colonna in un file di input

```
awk '{if ($1>max){max = $1}} END {print max}'
```

Estrarre le righe dispari di un file

```
awk '(NR % 2) {print}'
```

(notare che il pattern **è (NR%2)**, se il risultato dell'espressione in esso contenuta è diverso da zero viene eseguita l'azione **print**).

Stampare solo i campi dispari di ciascuna riga mediante la scansione delle variabili $1,…,$NF

```
awk '{for(i = 1;i <= NF;i+=2){printf("%s ", $i)} printf("\n")}'
```

Provate per esercizio a realizzare alcuni semplici comandi che utilizzate abitualmente sotto Unix

Programmi scritti su file

Un programma awk può essere scritto su file proprio come gli shell script:

```
#!/bin/awk –f
{
  for ( i = 1; i <= 100; i++ ) {
       if ( i != 50 ) {
          print i
       }
  }
}
```

Si può notare l'uso dell'opzione `-f` che consente di leggere il programma da file.

Dopo *#!/bin/awk –f* é necessario lasciare una riga vuota.

Non bisogna dimenticare di usare il comando **`chmod`** per rendere eseguibile il file.

I vettori in AWK

La loro dichiarazione avviene nel momento stesso in cui si fa riferimento ad una variabile vettore, ad esempio:

```
saluti[2] = "ciao"
```

L'elemento 2 dell'array saluti contiene ora la stringa `ciao`.

Se l'array non esiste viene creato: i suoi elementi sono creati nel momento in cui assegnano all'array.

Un elemento non inizializzato conterrà la stringa nulla (zero se si considera come un numero).

È importante sapere che in AWK gli array sono associativi, realizzati mediante una hash table, pertanto è possibile utilizzare le seguenti espressioni:

```
dizionario["Hello"] = "Ciao"

numeri1["unoduetre"] = 123

numeri2[2345] = "Due,tre,quattro e cinque"

matrix[$1,$2,$3] = "Esempio di matrice multidimensionale, in questo caso gli
indici fanno riferimento all'elemento [$1,$2,$3] della struttura matrix"
```

Gli elementi di un array non sono ordinati sequenzialmente come in un array lineare(sono stringhe e non numeri).

Se interessa scandire tutti gli elementi inseriti nella hash table corrispondente ad un array si usa il ciclo

for (var in arrayname *)*

Esempio:

```
for (i in vettore) {
print vettore[i]
}
```

Tuttavia nella maggior parte delle applicazioni si utilizzano array indicizzati mediante interi e quindi è possibile scandire tutti gli elementi di un array `vettore` di `max` elementi in maniera "ordinata" mediante l'usuale ciclo

```
for (i=0; i<max; i++) {
     print vettore[i]
}
```

Che l'array sia associativo in questo caso è completamente trasparente per il programmatore.

Esempi di programmi AWK

1) Implementare un programma AWK che dato un file con il seguente formato:

1

2 3

4 5 6

7 8 9 10

……..

produca in uscita la somma dei valori di ciascuna colonna

Soluzione: Incominciamo con lo scrivere il programma AWK che crea il file di input. Ecco una possibile implementazione in cui sono stati utilizzati sia il costrutto **`while`** che quello **`for`**

```
#!/bin/awk -f

BEGIN {
  n=1;
  numero=1;
  while(n<=10) {
       for(i=1;i<=n;i++) printf("%d ",numero++);
       printf("\n");
       n++;
  }
}
```

notare che l'utilizzo del `;` è analogo a quello del linguaggio C.

(Provare ad eliminare `BEGIN`. Che cosa succede? Perché?)

Creato questo programma passiamo ora a risolvere il problema vero e proprio:

```
#!/bin/awk -f

{
  for(i=1;i<=NF;i++) somma[i]+=$i;
}
END {
  for(i=1;i<=NF;i++) printf("%d ",somma[i]);
  printf("\n");
}
```

Notare che il vettore `somma` non viene dichiarato, ma è creato automaticamente durante l'utilizzo (i suoi elementi sono automaticamente inizializzati a zero).

Dopo aver sommato gli elementi delle colonne si visualizzerà il risultato.

END è un pattern che risulta vero solo dopo che il file in input è stato scandito fino ad **EOF**, viene quindi spesso utilizzato per stampare i risultati della elaborazione di tutte le linee di un file.

(Esercizio: invece di stampare la somma delle colonne del file, stamparne la media).

2) Dato un file con il seguente formato:

A: 10 100 b c

B: 101 a b 200 c

C: x y z

A: 102 x 100 b c

.....

.....

scrivere un programma AWK che crei un file **doppio.txt** con lo stesso formato del precedente, ma con i numeri che compaiono nelle righe etichettate da "A:" moltiplicati per 2.

```
#!/bin/awk –f

/^A:/ {
  for(i=1;i<=NF;i++) {
     if ($i !~ /^[0-9]+/) printf("%s ", $i) >> "doppio.txt"
     else printf("%d ",$i*2) >> "doppio.txt";
  }
  printf("\n") >> "doppio.txt";
}
/^[B-Z]:/ { print >> "doppio.txt" }
```

Si sono usate le espressioni regolari per trattare in modo diverso le linee del file.

Quando AWK elabora una linea che inizia per "A:" il pattern **/^A:/** risulta vero viene quindi

eseguita l'azione.

L'operatore **!~** è usato per verificare se **$i** (i è la variabile usata per scandire **$1,$2,…,$NF**) non corrisponde all'espressione regolare **^[0-9]+** (verifica se **$i** è un numero).

Più precisamente, l'operatore **~** è usato per verificare se la stringa **$i** contiene una stringa di caratteri conforme all'espressione regolare.

N.B.

Gli operatori **~** (contiene) e !**~** (non contiene) possono essere usati solo all'interno delle azioni.

Non è consentito invertire i due operandi quindi:

/ciao$/ ~ saluti e **/ciao$/ !~ saluti**

non sono considerate sintatticamente valide.

In questo programma si nota anche l'uso della redirezione delle **print** e **printf** sul file **doppio.txt** .

Esercizio: Se questo programma viene lanciato più volte il file **doppio.txt** non viene cancellato (infatti la scrittura avviene in append). Fare il modo che il file sia riscritto ogni volta che viene lanciato il programma.

Esercizio: Questo programma non gestisce il caso di un file errato le cui righe possono anche non iniziare con la sequenza **lettera:….** Modificare il programma in modo che stampi un messaggio di errore ed esca (istruzione **exit(codice)**) in caso di formato errato del file.

3) Scrivere un programma che dato l'output generato dal comando **ls –l** sia in grado di calcolare la distribuzione delle lunghezze dei vari file in intervalli di 1 Kb e di tracciare l'istogramma corrispondente.

```
#!/bin/awk –f


{
 num=int($5/1024);
 distrib[num]++;
 if (num>max) max=num; # max e'automaticamente inizializzata a 0
}
```

```
END {
# Disegna l'istogramma
 print "Istogramma\n" # Aggiunge un newline a quello di default di print
 for(i=0;i<=max;i++) {
    printf("[%6d -%6d] K = %6d |",i,i+1,distrib[i]);
    for(j=0;j<distrib[i];j++) printf("*");
    printf("\n")
 }
}
```

Si è usata la **`funzione int()`** che converte un numero reale in uno intero tramite troncamento; AWK dispone di molte funzioni utili che possono essere esaminate con il comando di Unix *man awk (o* man gawk se non dovesse funzionare). In questo programma si nota inoltre la somiglianza della funzione **`printf`** di AWK con la corrispondente funzione del C.

Questo è un esempio di output del programma:

```
Istogramma
[ 0 - 1] K = 15|***************
[ 1 - 2] K = 0 |
[ 2 - 3] K = 0 |
[ 3 - 4] K = 0 |
[ 4 - 5] K = 6 |******
[ 5 - 6] K = 0 |
[ 6 - 7] K = 0 |
[ 7 - 8] K = 1 |*
```

Si sarebbe potuto utilizzare il costrutto **`for (i in`** `distrib`**),** come è riportato nel secondo esempio di soluzione**,** ma occorre ricordare che questo uso è appropriato :

- quando si vuole fare riferimento solo agli elementi del vettore che sono stati modificati
- l'ordine degli elementi nel vettore non ha importanza.

```
#!/bin/awk –f

{
 distrib[int($5/1024)]++; # $5 è il campo che contiene la lunghezza del file
}
END {
# Disegna l’istogramma
 print "Istogramma\n" # Meglio avere due newline
 for(i in distrib) {
    printf("[%6d -%6d] K = %6d |",i,i+1,distrib[i]);
    for(j=0;j<distrib[i];j++) printf("*");
    printf("\n")
 }
}
```

Questo è un esempio di output di questa soluzione:

```
 Istogramma
[  4 -  5] K = 6 |******
[ 26 - 27] K = 1 |*
[  7 -  8] K = 1 |*
[ 47 - 48] K = 1 |*
[  9 - 10] K = 1 |*
[ 49 - 50] K = 1 |*
[ 20 - 21] K = 1 |*
[ 22 - 23] K = 1 |*
[  0 -  1] K = 14|**************
[ 50 - 51] K = 1 |*
```

Come si può notare l'output non è ordinato (il costrutto **`for (i in distrib)`** non scandisce ordinatamente il vettore) e inoltre mancano le righe [1 – 2] K = 0,[2 – 3] K = 0,ecc.

È possibile ordinare questo output mandandolo in pipe al comando **`sort`** ?

Esercizio: Modificare il programma in modo che gestisca il caso in cui il comando **`ls`** restituisca degli errori quando cerca di visualizzare i dati di un file o di una directory.

Esercizio: Fare il modo che le directory non vengano considerate (suggerimento: la prima lettera del primo campo dell'output generato da **`ls -l`** è **`d`** per le directory si può quindi risolvere il problema con il pattern appropriato).

4) Realizzare una semplice calcolatrice con AWK.

```
#!/bin/awk -f

# In questa calcolatrice si assume che l'utente non commetta errori di
digitazione
($2=="*") { print $1*$3 }
($2=="/") { print $1/$3 }
($2=="+") { print $1+$3 }
($2=="-") { print $1-$3 }
($1=="q") { exit(0) }
```

In questa soluzione non sono state usate le espressioni regolari.

Vediamo ora un esempio che sfrutta le espressioni regolari:

```
#!/bin/awk -f
```

```
/[0-9\.]+ [\+\-\*\/] [0-9\.]+/ {
if($2=="*") { print $1*$3; next }
if($2=="/") { print $1/$3; next }
if($2=="+") { print $1+$3; next }
if($2=="-") { print $1-$3; next }
}
($1=="q") { exit(0) }

# Molte vecchie versioni di awk accettano anche continue o break al posto di
# next (nella versione gawk continue e break devono essere usate solo
# all'interno dei cicli a meno che non si usi l'opzione --traditional)
```

Domanda: L'espressione **[\+\-*\/]** poteva essere scritta anche come **[+-*/]** oppure **[\+\-*/]**?

Passare variabili esterne a AWK

Talvolta è utile passare delle variabili esterne ad una procedura AWK.

Per fare questo si usa l'opzione **–v**, ad esempio:

```
awk –v inizio=3 –v fine=6 '{for(i=inizio;i<=fine;i++) print $i}'
```

oppure, nel caso in cui si voglia passare gli argomenti all'interno di uno script di shell:

```
awk –v inizio=$1 –v fine=$2 '{for(i=inizio;i<=fine;i++) print
$i}'
```

Questa opzione permette spesso di evitare l'uso del pattern **BEGIN** che generalmente viene utilizzato solamente per l'inizializzazione delle variabili a valori diversi da zero.

La fonte delle informazioni seguenti è la pagina man di **gawk**, la versione GNU di AWK che si trova nelle distribuzioni di Linux.

Operatori

Gli operatori di AWK, in ordine decrescente di precedenza, sono

(...**)** Raggruppamento

$ Riferimento a campi.

++ -- Incremento e decremento, sia prefisso che postfisso.

^ Elevamento a potenza (si può anche usare ****** al suo posto, e ****=** nell'assegnamento con operatore).

+ - ! Più e meno unari, e negazione logica.

*** / %** Moltiplicazione, divisione e resto.

+ - Addizione e sottrazione.

spazio Concatenazione di stringhe.

< >

<= >=

!= == I ben noti operatori di relazione.

~ !~ Controllo di conformità ("match") tra regular expression, e controllo di

non conformità. NOTA: Non usare una regular expression costante **(/foo/)** come operando di sinistra di **~** o **!~**. Si indichi sempre la regular expression come operando di destra. L'espressione **/foo/ ~ exp** ha lo stesso significato di **($0 ~ /foo/)~ exp)**. Non è questo, in genere, che ci si aspetta.

in Controllo di appartenenza ad un vettore.

&& AND logico.

|| OR logico.

?: L'espressione condizionale in stile C.

Ha la forma **expr1? expr2 : expr3**. Se **expr1** è vera, il valore dell'espressione è **expr2**, altrimenti è **expr3**. È valutata solo una delle due espressioni **expr2** ed **expr3**.

= += -=

***= /= %= ^=** Assegnamento. Sono riconosciuti sia l'assegnamento assoluto

(**var = value**) sia quello con operatore (le altre forme).

Funzioni Numeriche

AWK ha le seguenti funzioni aritmetiche predefinite:

atan2(y, x) l'arcotangente di **y/x** in radianti.

cos(expr) coseno di **expr** (si aspetta radianti).

exp(expr) esponenziale.

int(expr) troncamento ad intero.

log(expr) logaritmo naturale.

rand() fornisce un numero casuale tra 0 ed 1.

sin(expr) seno di **expr** (si aspetta radianti).

sqrt(expr) radice quadrata.

srand([expr]) usa **expr** come nuovo seme per il generatore di numeri casuali. Se **expr** non è indicata, sarà utilizzata la data corrente. Il valore di ritorno è il valore precedente del seme.

Funzioni su Stringhe

Gawk offre le seguenti funzioni di stringa predefinite:

getline Setta $0 leggendo la linea successiva; setta anche NF, NR, FNR.

getline <file Come sopra, ma legge da file

getline var Setta var leggendo la linea successiva; setta NR, FNR.

getline var <file Come sopra, ma legge da file

gensub(r, s, h [, t]) cerca nella stringa obiettivo **t** corrispondenze con la regular expression **r**. Se **h** è una stringa che inizia con **g** o **G**, tutte le corrispondenze con **r** sono sostituite con **s**; altrimenti, **h** è un numero che

indica la particolare corrispondenza con **r** che si vuole sostituire. Se **t** non è specificata, al suo posto è usato **$0**.

La sequenza **\0** rappresenta il testo ricoperto dall'intera espressione, e così pure il carattere **&**.

A differenza di **sub()** e **gsub(),** la stringa modificata è data come risultato della funzione; l'originale resta inalterata.

gsub(r, s [, t]) per ogni sottostringa conforme alla regular expression **r** nella stringa **t**, sostituisce la stringa **s**, e restituisce il numero di sostituzioni. Se **t** non è specificata, usa **$0**. Una **&** nel testo di rimpiazzo è sostituita dal testo trovato conforme con la regular expression.

Si usi **\&** per indicare il carattere **&** inteso letteralmente. Si veda AWK Language Programming per una più ampia discussione circa le regole che riguardano **&** e i "backslash" nel testo di rimpiazzo di **sub(), gsub(),** e **gensub().**

index(s, t) trova l'indice posizionale della stringa **t** nella stringa **s**, o restituisce **0** se **t** non è presente.

length([s]) la lunghezza della stringa **s**, oppure la lunghezza di **$0** se **s** non è specificata.

match(s, r) trova la posizione in **s** del tratto che si conforma alla regular expression **r**, oppure **0** se non ci sono conformtà.

I valori di **RSTART** ed **RLENGTH** sono modificati di conseguenza.

split(s, a [, r]) spezza la stringa **s** nel vettore a utilizzando il metodo di separazione descritto dalla regular expression **r**, e restituisce il numero di campi. Se **r** è omessa, il separatore utilizzato è **FS**.

Il precedente contenuto del vettore a è cancellato. La divisione di una stringa in parti e l'isolamento dei campi nei record, descritto prima, sono compiuti esattamente allo stesso modo.

sprintf(fmt, expr-list) stampa (in modo fittizio) **expr-list** secondo il formato **fmt**, e restituisce la stringa risultante.

sub(r, s [, t]) come **gsub(),** ma è sostituita solo la prima sottostringa trovata.

substr(s, i [, n]) restituisce la sottostringa di **s** di **n** caratteri al più che inizia nella posizione **i.** Se **n** è omesso, è usato il resto di **s**.

tolower(str) restituisce una copia della stringa **str**, con tutti i caratteri maiuscoli tradotti nei minuscoli corrispondenti. I caratteri non alfabetici restano invariati.

toupper(str) restituisce una copia della stringa **str**, con tutti i caratteri minuscoli tradotti nei maiuscoli corrispondenti. I caratteri non alfabetici restano invariati.

Funzioni di Tempo

Poiché uno dei principali campi di applicazione dei programmi AWK è l'elaborazione di file di traccia ("log files") contenenti informazioni di tipo marca temporale ("time stamp"), gawk mette a disposizione le seguenti due funzioni per ottenere marche temporale e per manipolarle.

systime() restituisce la data e l'ora correnti, espresse come numero di secondi trascorsi da una certa data convenzionale (la mezzanotte del 1/1/1970 sui sistemi POSIX).

strftime([format [, timestamp]])

Applica il formato format a **timestamp**. **timestamp** dovrebbe essere nella forma utilizzata da **systime()**. Se **timestamp** è omesso, sono utilizzate la data e l'ora correnti. Se format è omesso, è assunto un formato equivalente a quello utilizzato dal comando **date(1**). Per sapere quali formati di conversione siano disponibili, si faccia riferimento alle specifiche della funzione **strftime()** del C ANSI.

Una versione di pubblico dominio di **strftime(3)** e le relative pagine di manuale sono distribuite con gawk; se gawk è stato compilato utilizzando tale versione di **strftime**, potrà eseguire tutte le conversioni descritte nel manuale associato.

Riferimenti:

- A.D. Robbins, "Effective AWK Programming"

è il manuale completo che si trova in /usr/doc/gawk…. in formato Postscript nella distribuzione RedHat di Linux.

- Al sito http://www.ictp.trieste.it/texi/gawk/gawk_toc.html è presente il manuale completo in formato html.
- John J. Valley "La grande guida UNIX", Jackson libri
- Massimo Dal Zotto "AWK questo sconosciuto" , PLUTO JOURNAL in http://www.pluto.linux.it/journal/pj9809/index.html
- Pagine man di gawk
- Appunti Linux di Daniele Giacomini (un manuale di Linux completo e scaricabile gratuitamente da Internet (viene aggiornato molto spesso).

Per gli appassionati di WINDOWS

Collegandosi al sito

http://sourceware.cygnus.com/cygwin

è possibile scaricare gratuitamente i classici programmi GNU (tra cui **gawk**) ed un ambiente operativo (shell) UNIX.