Common threads: Часть 1. Awk в примерах

Введение в замечательный язык со странным именем

Даниэль Роббинс 29.01.2009

Awk — чудесный язык с очень странным именем. В этой первой статье серии, состоящей из трёх частей, Даниэль Роббинс дает краткое введение в основы программирования на awk. В следующих статьях серии будут рассмотрены более продвинутые темы, и в завершение будет создано серьёзное демонстрационное приложение на awk из реальной практики.

В защиту awk

Ссылка на оригинал (in English)

В этой серии статей я собираюсь сделать из читателя искусного программиста на awk. Я согласен, что у awk не самое приятное и модное имя, а GNU-версия awk, названная gawk, звучит откровенно странно. Незнакомые с этим языком программисты, услышав его название, возможно, представят себе мешанину древнего и устаревшего кода, способного довести до умопомрачения даже самого знающего специалиста по UNIX (заставив его восклицать "kill -9!" и беспрестанно бегать за кофе).

Да, у awk отнюдь не замечательное имя. Но это замечательный язык. Awk создан для обработки текста и создания отчетов, но у него много хорошо проработанных функций, дающих возможность серьезного программирования. При этом, в отличие от некоторых других языков, синтаксис awk привычен и заимствует лучшее из таких языков, как C, python и bash (хотя формально awk был создан до python и bash). Awk — один из тех языков, которые, будучи один раз выучены, становятся ключевой частью стратегического арсенала программиста.

Первый шаг в awk

Давайте начнем и попробуем поэкспериментировать с awk, чтобы увидеть, как он работает. В командной строке введем следующую команду:

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

В результате должно быть показано содержимое файла /etc/passwd. Теперь — объяснение того, что делал awk. Вызывая awk, мы указали /etc/passwd в качестве входного файла. Когда мы запустили awk, он обработал команду print для каждой строки в /etc/passwd по порядку. Весь вывод отправлен в stdout, и мы получили результат, идентичный результату команды cat /etc/passwd. Теперь объясним блок `{ print }`. В awk фигурные скобки используются для группирования блоков текста, как в C. В нашем блоке текста есть лишь одна команда print. В awk команда print без дополнительных параметров печатает все содержимое текущей строки.

Вот еще один пример программы на awk, которая делает то же самое:

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

В awk переменная `$0` представляет всю текущую строку, поэтому `print` и `print $0` делают в точности одно и то же. Если угодно, можно создать программу на awk, которая будет выводить данные, совершенно не связанные с входными данными. Вот пример:

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

При передаче строки "" команде print она всегда печатает пустую строку. Если протестировать этот скрипт, обнаружится, что awk выводит одну пустую строку на каждую строку в файле /etc/passwd. Это опять-таки происходит потому, что awk исполняет скрипт для каждой строки во входном файле. Вот еще один пример:

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

Если запустить этот скрипт, он заполнит экран словами «ура». :)

Множественные поля

Awk хорошо подходит для обработки текста, разбитого на множество логических полей, и дает возможность без усилий обращаться к каждому отдельному полю из awk-скрипта. Следующий скрипт распечатает список всех учетных записей в системе:

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

В вызове awk в вышеприведенном примере параметр –F задает ":" в качестве разделителя полей. Обрабатывая команду print $1, awk выводит первое поле, встреченное в каждой строке входного файла. Вот еще один пример:

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

Вот фрагмент из вывода на экран этого скрипта:

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

Как видим, awk выводит первое и третье поля файла /etc/passwd, которые представляют собой соответственно поля имени пользователя и uid. При этом, хотя скрипт и работает, он не совершенен — нет пробелов между двумя выходными полями! Те, кто привык программировать в bash или python, возможно ожидали, что команда `print $1 $3` вставит пробел между этими двумя полями. Однако когда в программе на awk две строки оказываются рядом друг с другом, awk сцепляет их без добавления между ними пробела. Следующая команда вставит пробел между полями:

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

Когда print вызывается таким способом, он последовательно соединяет `$1`, `" "` и `$3`, создавая удобочитаемый вывод на экране. Конечно, мы можем также вставить метки полей, если нужно:

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

В результате получаем такой вывод:

```
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.
```

Внешние скрипты

Передача скриптов в awk в виде аргументов командной строки может быть удобной для небольших однострочных текстов, но когда дело доходит до сложных многострочных программ, определенно будет лучше составить скрипт в виде внешнего файла. После этого можно указать awk этот скриптовый файл с помощью параметра -f:

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

Размещение скриптов в отдельных текстовых файлах также позволяет воспользоваться дополнительными преимуществами awk. Например, следующий многострочный скрипт делает то же самое, что и один из наших предыдущих однострочных - распечатывает первое поле каждой строки из /etc/passwd:

```
BEGIN {
        FS=":"
}
{ print $1 }
```

Разница между этими двумя методами состоит в том, как мы задаем разделитель полей. В этом скрипте разделитель полей указывается внутри самой программы (установкой переменной FS), тогда как в нашем предыдущем примере FS настраивается путем передачи awk параметра -F":" в командной строке. Обычно лучше всего задавать разделитель полей внутри самого скрипта, просто потому, что тогда не потребуется запоминать ещё один аргумент командной строки. Позже в этой статье мы рассмотрим переменную FS более подробно.

Блоки BEGIN и END

Обычно awk выполняет каждый блок в тексте скрипта один раз для каждой входной строки. Однако в программировании часто встречаются ситуации, когда требуется выполнить код инициализации перед тем, как awk начнет обрабатывать текст из входного файла. Для таких случаев awk дает возможность определять блок BEGIN. Мы использовали блок BEGIN в предыдущем примере. Поскольку блок BEGIN обрабатывается до того, как awk начинает обрабатывать входной файл, это отличное место для инициализации переменной FS (разделитель полей), вывода заголовка или инициализации других глобальных переменных, которые будут позже использоваться в программе.

Awk также предоставляет еще один специальный блок, называемый блоком END. Awk выполняет этот блок после того, как все строки во входном файле были обработаны. Обычно блок END используется для выполнения заключительных вычислений или вывода итогов, которые должны появиться в конце выходного потока.

Регулярные выражения и блоки

Awk позволяет использовать регулярные выражения для избирательного выполнения отдельных блоков программы в зависимости от того, совпадает или нет регулярное выражение с текущей строкой. Вот пример скрипта, который выводит только те строки, которые содержат символьную последовательность `foo`:

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

Конечно, можно использовать более сложные регулярные выражения. Вот скрипт, который будет выводить только строки, содержащие число с плавающей точкой:

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

Выражения и блоки

Есть много других способов избирательно выполнять блок программы. Мы можем поместить перед блоком программы любое булево выражение для управления исполнением этого блока. Awk будет выполнять блок программы, только если предыдущее булево выражение равно true. Следующий пример скрипта будет выводить третье поле всех строк, в которых первое поле равно `fred`. Если первое поле текущей строки не равно `fred`, awk продолжит обработку файла и не выполнит оператор `print` для текущей строки: :

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

Awk предлагает полный набор операторов сравнения, в том числе обычные "==", "<", ">", "<=", ">=" и "!=". Кроме того, awk предоставляет операторы "~" и "!~", которые означают "совпадает" и "не совпадает". При их использовании переменная помещается слева от оператора, а регулярное выражение — справа от него. Вот пример, где выводится только третье поле строки, если пятое поле той же строки содержит символьную последовательность `root`:

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

Условные операторы

Awk предоставляет также очень приятные C-подобные операторы if. При желании можно переписать предыдущий скрипт с использованием `if`:

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

Оба скрипта работают идентично. В первом примере булево выражение находится вне блока, в то время как во втором примере блок выполняется для каждой входной строки, и мы избирательно выполняем команду печати, используя оператор `if` Оба метода работают, и выбрать можно тот, который наилучшим образом объединяется с другими частями скрипта.

Вот более сложный пример оператора `if` в awk. Как можно видеть, даже в случае сложных вложенных условных выражений операторы `if` выглядят идентично их аналогам в C:

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

Используя операторы if, мы можем преобразовать этот код:

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

в такой:

```
{
  if ( $0 !~ /matchme/ ) {
          print $1 $3 $4
  }
}
```

Оба скрипта распечатают только те строки, которые *не* содержат символьную последовательность `matchme`. И в этом случае тоже можно выбрать метод, который лучше работает в конкретной программе. Они оба делают одно и то же.

Awk также дает возможность использовать булевы операторы "||" ("логическое ИЛИ") и "&&"("логическое И"), что позволяет создавать более сложные булевы выражения:

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

Этот пример выведет только те строки, в которых первое поле равно `foo`*и* второе поле равно `bar`.

Числовые переменные!

До сих пор мы распечатывали либо строковые переменные, либо целые строки, либо конкретные поля. Однако awk также дает нам возможность выполнять сравнение как целых чисел, так и чисел с плавающей запятой. Используя математические выражения, очень легко написать скрипт, который считает число пустых строк в файле. Вот один такой скрипт:

```
BEGIN { x=0 }
/^$/  { x=x+1 }
END   { print "Найдено " x " пустых строк. :)"
 }
```

В блоке BEGIN мы инициализируем нашу целочисленную переменную `x` значением ноль. Затем каждый раз, когда awk встречает пустую строку, он будет выполнять оператор `x=x+1`, увеличивая `x`на 1. После того как все строки будут обработаны, будет выполнен блок END, и awk выведет конечный итог, указав число найденных пустых строк.

Строчные переменные

Одной из приятных особенностей переменных awk является то, что они "простые и строчные." Я называю переменные awk "строчными", потому что все переменные awk внутри хранятся как строки. В то же время переменные awk "простые", потому что с переменной можно производить математические операции, и если она содержит правильную числовую строку, awk автоматически позаботится о преобразовании строки в число. Чтобы понять, что я имею в виду, взглянем на этот пример:

```
x="1.01"
# Мы сделали так, что x содержит *строку* "1.01"
x=x+1
# Мы только что прибавили 1 к *строке*
print x
# Это, кстати, комментарий :)
```

Awk выведет:

```
2.01
```

Любопытно! Хотя мы присвоили переменной x строковое значение 1.01, мы все же смогли прибавить к ней единицу. Нам бы не удалось сделать это в bash или python. Прежде всего, bash не поддерживает арифметику с плавающей запятой. И, хотя в bash есть "строчные" переменные, они не являются "простыми"; для выполнения любых математических операций bash требует, чтобы мы заключили наши вычисления в уродливые конструкции `$( )` . Если бы мы использовали python, нам бы потребовалось явно преобразовать нашу строку `1.01` в значение с плавающей запятой, прежде чем выполнять какие-либо расчеты с ней. Хоть это и не трудно, но это все-таки дополнительный шаг. В случае с awk все это делается автоматически, и это делает наш код красивым и чистым. Если бы нам потребовалось возвести первое поле каждой входной строки в квадрат и прибавить к нему единицу, мы бы воспользовались таким скриптом:

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

Если немного поэкспериментировать, то можно обнаружить, что если в какой-то переменной не содержится правильного числа, awk при вычислении математического выражения будет обращаться с этой переменной как с числовым нулем.

Множество операторов

Еще одна приятная особенность awk — это полный комплект математических операторов. Кроме стандартных сложения, вычитания, умножения и деления, awk дает нам возможность использовать ранее продемонстрированный оператор показателя степени "^", оператор остатка целочисленного деления "%" и множество других удобных операторов присваивания, заимствованных из C.

К ним относятся пред- и постинкрементные/декрементные операторы присваивания (`i++, --foo`), операторы присваивания со сложением/вычитанием/умножением/делением (`a+=3, b*=2, c/=2.2, d-=6.2`). Но это ещё не все — мы имеем также удобные операторы присваивания с вычислением остатка целочисленного деления и возведением в степень (`a^=2, b%=4`).

Разделители полей

В awk есть свой собственный комплект специальных переменных. Некоторые из них дают возможность тонкой настройки работы awk, а другие содержат ценную информацию о вводе. Мы уже затронули одну из этих специальных переменных, FS. Как упоминалось ранее, эта переменная позволяет задать последовательность символов, которую awk будет считать разделителем полей. Когда мы использовали в качестве ввода /etc/passwd, FS была установлена в ":". Это оказалось достаточно, но FS предоставляет нам еще большую гибкость.

Значение переменной FS не обязано быть одним символом; ей может быть присвоено регулярное выражение, задающее символьный шаблон любой длины. Если производится обработка полей, разделенных одним или несколькими символами табуляции, то FS нужно настроить таким образом:

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

Выше мы использовали специальный символ регулярных выражений "+", который означает "одно или несколько вхождений предыдущего символа".

Если поля разделены пустой областью (один или несколько пробелов или символов табуляции), возможно, вам захочется установить для FS следующее регулярное выражение:

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

Хотя такая настройка сработает, в ней нет необходимости. Почему? Потому что по умолчанию значение FS равно одному символу пробела, который awk интерпретирует как "один или несколько пробелов или символов табуляции". В нашем конкретном примере значение FS по умолчанию именно такое, как нам было нужно!

Со сложными регулярными выражениями также не возникает проблем. Даже если записи разделены словом "foo", за которым следуют три цифры, следующее регулярное выражение позволит правильно разобрать данные:

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

Число полей

Следующие две переменные, которые мы собираемся рассмотреть, обычно не предназначены для записи в них, а используются для чтения и получения полезной информации о вводе. Первая из них — переменная NF, называемая также "число полей". Awk автоматически устанавливает значение этой переменной равным числу полей в текущей записи. Можно использовать переменную NF для отображения только определенных входных строк:

```
NF == 3 { print "в этой записи три поля: " $0 }
```

Конечно, переменную NF можно использовать и в условных операторах, например:

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

Номер записи

Еще одна удобная переменная - номер записи (NR). Она всегда содержит номер текущей записи (awk считает первую запись записью номер 1). До сих пор мы имели дело с входными файлами, которые содержали одну запись на строку. В таких ситуациях NR также сообщит номер текущей строки. Однако когда мы начнем обрабатывать многострочные записи в следующих статьях этой серии, это уже будет не так, поэтому нужно проявлять осторожность! NR можно использовать подобно переменной NF для вывода только определенных строк ввода:

```
(NR < 10 ) || (NR > 100) { print "Мы на записи номер 1-9 или 101 и более" }
```

Еще один пример:

```
{
  #skip header
  if ( NR > 10 ) {
          print "вот теперь пошла настоящая информация!"
  }
}
```

Awk предоставляет дополнительные переменные, которые могут быть использованы для различных целей. Мы рассмотрим эти переменные в следующих статьях. Мы подошли к концу нашего начального исследования awk. В следующих статьях серии я покажу более сложную функциональность awk, и мы закончим эту серию awk-приложением из реальной практики. А пока, если хочется узнать больше, можно просмотреть ресурсы, перечисленные ниже.

Похожие темы

- Оригинал статьи Common threads: Awk by example, Part 1. (EN)
- Если вы предпочитаете добротные старомодные книги, то отличным выбором будет sed & awk, 2nd Edition издательства O'Reilly.(EN)
- comp.lang.awk FAQ: много дополнительных ссылок по awk. (EN)
- Учебник по awk Патрика Хартигана (Patrick Hartigan) содержит множество удобных скриптов на awk. (EN)
- TAWK Compiler Томпсона компилирует скрипты на awk в быстрые исполняемые файлы. Имеются версии для Windows и DOS. (EN)
- The GNU Awk User's Guide—сетевой справочник. (EN)