Unix utilities, Part 2

Simple tools, complex problems

Peter Seebach (unixcomponents@seebs.plethora.net)
Freelance writer
June 2001

We explore how to combine the components of the Unix programming environment to solve a variety of tasks.

Contents:

Related dW content:

Unix utilities Part 1

Unix pipes are effectively a component architecture, as we discussed in Part 1 of this series. In this installment, we'll strengthen the argument for saying so. But first of all, we'll look at how Unix supports a design goal that underlies component architecture, object-oriented architecture before it, and structured programming before that. This is the principle of *decomposition*, which allows us to build full results from partial ones.

Partial results are better than none

Sometimes, the goal is simply to produce an immediate result: You don't care much about performance (within broad limits) and you don't need to worry about special cases because you know a lot about your input stream. In these cases, focus on the simplicity and convenience of the system. Unix is very good at mostly solving a problem in a couple of steps, then taking a couple more to clean up the results into the desired format.

Unix tools are built with the idea that it's fine for the first tool in a pipeline to solve only 90% of the problem. Indeed, it's fine if it only solves 10% of the problem, as long as that helps break the problem down into bite-sized chunks.

So let's say I want to know how common a word is in a file. Rather than using the "utility that counts occurrences of each word in a file" (which would be rarely used), I break the problem down into steps:

- Break the file into a series of words, one to a line
- Strip out all words that are not the one I wish to count
- Count the lines

Thus, for instance, I might use the following pipeline:

```
tr -cs a-zA-Z '\012' | grep '^foo$' | wc -l
```

What does that do? tr translates characters. In this case, it translates all characters which are *not* in a-zA-Z (-c for "complement") into newlines, "squeezing" (-s) multiple copies of a given character together. Thus, I get each word on a line by itself. Then I use grep to find lines with nothing on them but my word ("foo"), then I

count the lines.

Throughout this, you may find yourself wondering how you'll learn all these weird commands. Well, you learn them the same way you learn any other system of tools; gradual exposure and study, and heavy use of documentation. The Unix man pages, which every newbie hates, are *beautiful* references if you already know (mostly) what you want and you're trying to remember how to spell it. It's not necessarily harder to learn than anything else; it just looks hard if it's not the thing you learned first.

The man pages are especially made useful by the apropos or man -k commands, which allow you to search the summaries of man page entries. So you could enter "apropos sort" if you're looking for a command to sort data. You could think of this as an interface repository with basic search capability.

Looking only for lines that contain "foo" wouldn't have solved my problem; it wouldn't have counted "foo bar foo" on the same line as two instances. Counting lines would have been useless. However, once I broke the stream into individual records matching the kind of data I was looking for, everything fell into place.

Pipes are cheap

In Unix, tools are most often combined by stringing them together, with the output of each tool in a series being used as input to the next. The thing through which data flows is called a *pipe*. In some systems, moving data from one program to another is expensive. In Unix, pipes are cheap.

Unix favors initial solutions that are cheap in programmer time, even if they aren't maximally efficient. Obviously, a carefully tuned program customized for a given task is likely to outperform a series of generalized programs communicating through pipes. (Of course, this will require *careful* tuning -- a badly tuned program may do all of its tasks poorly and end up being slower.)

Experienced Unix users aren't afraid to add a tool to a pipeline that only does one trivial thing: It's cheaper than doing it by hand, and it's not going to slow you down measurably while you're solving the problem. If you end up repeating a task frequently, and you have a performance problem, *then* you try to improve. But you don't worry about it while you're looking at the problem for the first time.

So, for instance, with the "counting occurrences of foo" example above, I could have used grep -c '^foo$' instead of grep '^foo$' | wc -l. It didn't occur to me right away when I was typing the example, so I left it alone. The performance difference is hardly measurable on the kind of data this is likely to be used for. However, this example shows that there's often more than one way to solve a problem.

In some systems, pipes are implemented using hidden temporary files, and the first program in a pipe must be complete before the second can start running. Unix systems don't do this -- all of the programs in a pipe run simultaneously, and process output as it comes through.

Distributed components using network sockets depend on the same feature -- both sides of a socket can be working at once. Indeed, some of the Unix tools end up providing a pipe that runs to another machine over a network.

Build your own tools

If you find that you need a specific partial result frequently when you're working on something, you can take the part of the command line that does that job and save it in a file. In doing so, you're writing a program. Because the shell itself is a programming language, command lines are themselves programs; you can save one of these programs and then use it just like any other. You don't need to know C to program a Unix system.

For instance, let's say I frequently want a list of words in a file, sorted by frequency. I might create a shell script that looks something like:

```
#!/bin/sh
cat $* |
tr -cs a-zA-Z '\012' |
sort |
uniq -c |
sort -n |
awk '{print $2}
```

Once again, no one of these utilities is doing anything very hard, but the series ends up being quite powerful.

However, the resulting utility itself can be used quite effectively in another pipeline. Note to readers who aren't used to Unix: That is the *whole program* -- no IDL wrappers, no frameworks, no templates, no makefiles, no nothing. If you type it in and flag it as executable, it's a program, and it will do exactly what it's supposed to do. You don't need a special tool to make this program work as a component. It already is a component.

Some of you may not think this is a useful application, but it's really the same as, say, an application sorting a customer base by the number of times they've missed payments. You hand the application the list of people who missed payments, and the output is the list of people, sorted in order of how often they missed payments.

Let's say I want to check on the least and most used words in a given file. I can now use:

```
freq | tail
```

to find the most common words, or

```
freq | head
```

to find the least common. Having found a useful combination, I now treat it as a stepping stone, not just as a goal.

Tools as plug-ins

When I was sending the first draft of this article to my editor, I noticed that there were no linebreaks in the file I'd just read into my mailer. Luckily, there's a tool called fmt which breaks lines. So I told vi to run all of the file -- from the line I was on to the end -- through an external program and replace the contents of the file with the output of the program.

This is a great model. Instead of providing every feature you could ever want in an editor, Unix tends to give you a way to run the contents of a file *through* a tool, producing corrected output. Want arithmetic evaluation? Shove the current line through a calculator. This is one of the things Unix does very well. In most systems, merely knowing that a component exists wouldn't make it practical for a user to plug it into a word processor and apply it in place to a block of text.

Challenge your expectations

In the first article of this series, I talked about how other tools that need editing "services" may simply pass the task on to the user's preferred editor. This is found in an environment variable called $EDITOR.

The user can set $EDITOR to point to any program. What if it's *not* an interactive editor?

It turns out that everything works just fine. Imagine that we keep the human resources database as a tab-delimited flat file with several sections: the employee listings, the department listings, and the performance

evaluation section. We want to remove all the various records associated with the employee who left the company. (Note that, as we'll see later, it doesn't matter if we *actually* keep the data tab delimited; we can export and import to make it *temporarily* tab delimited.)

Or perhaps we maintain a customer contact database, and now we have to comply with a federal regulation to remove all information for customers who have signed on to an opt-out list.

I can do this by hand. It will take a while, but I can do it.

Or, I can create a shell script:

```
#!/bin/sh
grep -v XYZ $i > $i.new
mv $i.new $i
```

If I set $EDITOR to point to this script, and run vipw, vipw handles all of the "work" of altering the password file. But the actual editing, rather than being a painstaking process that involves trying to read every line of a few-hundred-line file, takes a few thousandths of a second. (Actually, in older versions of vipw, this could be a problem, because the program assumed that a time stamp would always change if the file had really been changed.)

Essentially, to vipw, $EDITOR is just another plugin. It doesn't matter what it is, as long as it edits a file in place.

Switching modes

What if the editor *doesn't* edit in place? In traditional component architectures, every component has to provide for itself every interface you are likely to need. In Unix, changing the interface is a matter of having a tool that changes the interface of another tool.

Of course, for a large component in a traditional component architecture, this is hardly a detectable load. However, it makes small components prohibitively expensive. Adding 1% to the code base of a large and powerful program is under the radar. Tripling the cost of writing a small component means you don't write it, or that you end up writing components that try to do it all instead of writing components that follow the excellent example set by uniq and sort.

Take our example above in which we used grep on the password file. If grep had a mode for editing in place, you could use it as your editor. But it doesn't, so you need a wrapper.

Unix comes with a variety of wrappers that perform key translations; others can be written easily, as we have seen in this article (and will see more in the next).

The most powerful, and mind-bending, of all the Unix wrappers is probably xargs. The xargs program takes, on standard input, a list of files then runs these files through a named command. It works particularly well in conjunction with a utility called find, which produces lists of files. (In fact, find can also do this, but it's less efficient in most cases.)

For instance, let's say I wish to look for the string "foo" in every C source file in or under the current directory:

```
find . -name "*.c" -print | xargs grep foo
```

Perhaps I just wish to know which files contain the string?

```
find . -name "*.c" -print | xargs grep -l foo
```

Note that, in the Unix environment, the find command just lists files; it doesn't try to find files by content. If you want to look for content, you pass the list of files you want to search into a tool that knows how to find content. In some environments, a single program has to try to provide an interface that can search either file characteristics or content. In Unix, the tool that knows how to find files does the thing it's good at and leaves other tasks to the tools that are good at those other things.

Let's say you want to run a given program on a series of files, but have the output be in place rather than in a stream.

A simple version of this utility might take names on standard input, and the command to run as an argument. It might look like this:

```
#!/bin/sh
while read i
do $* < $i > $i.new && mv $i.new $i
done
```

Note that this is not safe or reliable. For a "real" application, you'd want to do a lot more testing, and protect against some common mistakes. However, if you know the inputs and the data set, you can do this and it'll work fine.

Not sure you got it right? Change the middle line to

```
do echo "$* < $i > $i.new && mv $i.new $i"
```

and see whether the list of commands it generates looks right. Better yet, if you want to run a test case, just run the utility into head -1 | sh, and see if it does what you wanted to the first file.

The shell itself is quite willing to be used as a tool. It takes commands on standard input, and puts the output of those commands on standard output.

In other words, every tool is designed to be used to build other tools, and the resulting tools are themselves still designed to play well with others. If a tool doesn't work the way you want it to, there's probably a tool to perform the "translation" of interfaces. If not, you can write one easily.

In Part 3, I'll discuss building new tools.

Resources

- This article is the second in a four-part series. The other parts are:
  - Unix utilities, Part 1: Yet another new component architecture

- *The Mythical Man Month*, Frederick P. Brooks, Addison-Wesley 1995.
- *A Quarter Century of Unix*, Peter H. Salus, Addison-Wesley 1994.
- In *Let's Make Unix Not Suck*, Miguel De Icaza does not agree that Unix utilities are components.
- Chris Browne on The Unix Philosophy.
- Thomas Scoville's well-known *UNIX as Literature* essay.
- IBM Tools and Toys for Unix page, where you can also download and study the code for these IBM open-source Unix utilities.

- An collection of links to excellent Unix resources from the IBM eServer group
- Application development tools for AIX and Unix. For all of those who think Unix is "old," we just had to link to this page, which states "Behind every cutting-edge software development, there's a Unix developer." How true that is ...

About the author

As a small child, Peter Seebach thought that "rm" was a perfectly intuitive name for the command to "ReMove" a file. He has been a Unix bigot ever since. You can reach him at unixcomponents@seebs.plethora.net.

What do you think of this article?

Killer! (5) Good stuff (4) So-so; not bad (3) Needs work (2) Lame! (1)

Comments?

Submit feedback