toolman

Generating Web Pages with `sh` and `make`, Part 2

by Daniel E. Singer

Dan has been doing a mix of programming and system administration since 1983. He is currently a system administrator in the Duke University Department of Computer Science in Durham, North Carolina, USA.

<des@cs.duke.edu>

In the previous issue of *;login:*, we began to explore the topic of how the Bourne shell and `make` can be used to maintain and generate collections of Web pages. In Part 1, we covered how the shell – or practically any scripting or macro language – can satisfy the objectives of *consistency* and *simplification:* variables and functions can be defined in a master script and in individual "source" files to hold values and consolidate HTML coding. In fact, the full power of the shell can be exploited to simplify many tasks.

In this second installment, we'll discuss how `make` fits into this picture. This will not be a tutorial on writing makefiles, though relevant makefile features will be presented.

Using `make` for Automation and Mutability

We can exploit the UNIX `make` utility to great advantage, both to automate the generation of the Web pages from their source files and the master script, and to help in the production of alternative builds of the Web pages. For those unfamiliar with `make`, let me just say that it will update designated "target" files (for example, recreate a compiled binary executable) based on any changes that have been made to any source files on which they depend, and utilizing implicit or explicit production rules. See `man make` for the details.

Just as various approaches might be taken in the shell-scripting end of this process, various approaches can be taken when constructing a makefile (a specialized sort of script for `make`, usually in a file named "makefile" or "Makefile"). I've chosen to construct my makefiles in a way that makes adding a new Web page easy: I only have to add the basename of the new source file to one line in the makefile!

Here are the components that are necessary, or that at least make things a lot easier. And though there are many platform-specific capabilities and features of `make`, the ones used here should be fairly portable. Please note that I'm no `make` expert, and so there might be more elegant ways to implement some of this.

The first section of code, below, sets up macros to identify our (fictitious) Web pages. The first line is just a list of the basenames of the files, which should be the same for both those that contain the HTML source and those that contain the final HTML code. The second and third lines use a substring replacement macro feature to make lists of both the HTML source files (with a `.hsrc` suffix), and the HTML files (with a `.html` suffix). If we need to add a new Web page, we just add its basename to that first line. That's it, the exception being if a page has any special processing needs (see below).

```
NAMES = index part1 part2 part3 notes appendix
HSRCS = $(NAMES:=.hsrc)
HTMLS = $(NAMES:=.html)
```

The next section defines an implicit (default) build rule. The first two lines establish an implicit dependency of any `.html` file on an associated `.hsrc` file. The second two lines define the default command to execute to generate a `.html` file from a `.hsrc` file. This is all in cryptic *makefile-ese*, which we won't delve into here.

```
.SUFFIXES:
.SUFFIXES: .html .hsrc $(SUFFIXES)
.hsrc.html:
	./gen_html.sh $*
```

The next section defines some targets. The first target `docs` just says to make sure that all of the `.html` files are up to date. This might be the first (default) target in the makefile, and can be invoked with `make docs` or just `make`. The second one says that all of the `.html` files are dependent on the master script in addition to their default dependencies (their corresponding `.hsrc` files), and that they will all be rebuilt if it changes. The third line establishes a dependency of the file basenames on the `.html` files, and is the one that lets us specify a basename as a target; for example, `make index` instead of `make index.html`. The build rule for `index` is actually empty, but this still results in its dependencies and their dependencies (and so on) to be checked and rebuilt if necessary.

```
docs: $(HTMLS)
$(HTMLS): gen_html.sh
$(NAMES): $$@.html
```

With these definitions, rules, and dependencies in the makefile, we can now type `make` in the directory where these files reside, and any of the HTML pages that are out of date will be expeditiously updated from their respective source files, the master script, and any supplemental scripts and files. Please note that this is by no means the complete makefile. I've only included the parts here that warranted discussion. See the `man` page or other reference materials for more details on `make` and makefile fabrication. See the end of this article for URLs for a couple of sample makefiles.

Additional Dependencies

In the previous article, I mentioned the example of the alumni address databases. An additional consideration in this and similar situations is that when we update one of these databases, we also need to rebuild the `.html` files that contain the data, that is, that depend on them. It's also a good idea to update these pages if the database conversion script is modified. We can accomplish all of this by providing some additional explicit dependency lines, such as:

```
email-addrs.hsrc: email-addrs.db
     touch $@
email-addrs.html: cvt_addrs.sh
```

The first two lines say that when the database changes, the `.hsrc` file should be marked as modified. The underlying reason why we want to do this is because we want the "Last update:" line in the Web page to reflect this change, and that date happens to be based on the modification time of the `.hsrc` file. This also indirectly establishes a dependency for the `.html` file. The third line says that to the current dependencies of the `.html` file, the conversion script should be added. In this arrangement, a change to the conversion script will not alter the "Last update:" date. That's just a design decision. If we want it otherwise, we could just add `cvt_addrs.sh` to the first line, and omit the third line.

Alternative Builds

Now let's say that the versions of the Web pages in our current directory are intended as prototypes, and that the *real* pages will contain different hypertext links and will live in a different location – possibly on a totally different network and HTTP server. We can adjust our HTML source and makefile somewhat to accommodate both versions, such that we can generate either on demand.

The way that I've done this (and, again, there may be a better way) is to set up the makefile so that we can have it hard-link all of the relevant files (including itself) to a subdirectory, and then have it call itself with an argument (a macro definition) that causes an additional argument to be passed to `gen_html.sh`. This, then, tells

I've chosen to construct my makefiles in a way that makes adding a new Web page easy: I only have to add the basename of the new source file to one line in the makefile!

Got a tool that's useful, unique, way cool? Please send a description to <Toolman@usenix.org>.

gen_html.sh to use a different BASE HREF for all of the Web pages it generates, and that's about all it takes.

The additional lines in the makefile might look like this:

```
AUX_FILES = Makefile gen_html.sh cvt_addrs.sh
DB_FILES  = email-addrs.db web-addrs.db
DIST_FILES = $(AUX_FILES) $(DB_FILES) $(TMPLS)
DIST_DIR  = production-dir
...
## clear the distribution and install all links
install-dist:
     @ echo "Erasing..." ; \
          cd $(DIST_DIR) && rm -f $(DIST_FILES) $(HTMLS)
     @ echo "Linking..." ; \
          cd $(DIST_DIR) && \
          for F in $(DIST_FILES) ; do ln ../$$F ; done
## update dist html pages; assumes that links are up to date;
make-dist:
     cd $(DIST_DIR) && $(MAKE) docs DIST="-dist"
## completely wipe, install, and make the dist html pages;
all-dist: install-dist make-dist
```

And in this makefile, the rule for building the .html file will instead look like this:

```
.hsrc.html:
     ./gen_html.sh $(DIST) $*
```

When we make the distribution version with "make all-dist", make calls itself with 'DIST="-dist",' which causes gen_html.sh to be called with "-dist". All gen_html.sh has to do is catch this option, and take some appropriate actions such as setting some variables, particularly the one that sets the BASE HREF. Now all that *we* need to do is copy the newly generated set of .html files from the subdirectory to their new home!

So, maybe I've gone a bit more in the tutorial direction with makefiles than I had intended, but I feel that these examples are beneficial in building the case on the fundamental role that the make utility can play in the design and upkeep of a collection of Web pages. I hope I've convinced you.

Wrappin' Up

There're a lot of free programs and packages out there on the Web (speak of the devil) to help you to develop Web pages, not to mention the plethora of commercial packages. Some are specialized macro packages, some are WYSIWYG editors, others are server-side dynamic page generators. (I haven't used any of them; let me know if you have any recommendations.) If you care to explore, you might try searching on "HTML editors" or "HTML authoring" (or substitute "Web" for "HTML"). A good starting point for surfing might be <http://www.w3.org/Tools/>, though unfortunately this otherwise excellent page is no longer being updated.

Of course, for us do-it-yourselfers, a shell and make have a certain appeal. I'll provide some sample scripts and makefiles on the Toolman home page and FTP site; you can download them for use as starting points, if you're inclined to experiment with these ideas.

Happy scripting!

URLs:

<http://www.cs.duke.edu/~des/toolman.html>

<ftp://ftp.cs.duke.edu/pub/des/scripts/gen_html/INDEX.html>