How To Synchronize Directories with Rsync

Today I'd like to show you the basic usage of **rsync** – a wonderful, old and reliable tool for incremental data transfers and synchronization of local directories or even data between different Unix systems.

rsync is quite a complicated command, so don't expect this first post to explain everything and cover every possibility. Like I said, this is only the beginning.

What is rsync?

rsync (stands for ***r****emote* ***sync****hronization*) is an open source tool for data transfers between Unix systems.

In simplest form, it's just a Unix command you run locally to synchronize two directories. But the real power of rsync is when you need to synchronize directories between remote systems. rsync relies on ssh protocol for transferring the data between Unix systems, but earlier versions used rsh. Advanced deployments imply using rsync server in addition to simply running the command – this is basically the same command but running in a stand-by daemon mode.

rsync can easily be found or installed in any modern Unix-like OS, but it's always best to check the official website for latest developments around this tool: rsync website.

What does rsync do?

rsync synchronizes directories – makes one directory look (contain the same files and subdirectories) exactly like another one. rsync works by getting a list of files in your source and destination directories, comparing them as per

specified criteria (file size, creation/modification date or checksum) and then making the destination directory reflect all the changes which happened to the source since the last synchronization session.

Basic rsync usage

Just to show you how it works, I'm going to create two directories with a few files in them. **/tmp/dir1** in my examples will be a source directory (original dataset), while **/tmp/dir2** will be a destination directory – to be made the same as /tmp/dir1 as the result of running rsync.

So that's how I set up directories and files:

ubuntu$ **mkdir /tmp/dir1 /tmp/dir2** ubuntu$ **cd /tmp** ubuntu$ **echo "original file 1" > dir1/file1** ubuntu$ **echo "original file 2" > dir1/file2** ubuntu$ **echo "original file 3" > dir1/file3** ubuntu$ **cp dir1/file1 dir2**

That's how our directories and files look now, so dir2 contains a copy of file1:

ubuntu$ **find ./dir*** ./dir1 ./dir1/file2 ./dir1/file3 ./dir1/file1 ./dir2 ./dir2/file1

Now it's time to run your first ever rsync. There's two ways of specifying options for the command, a full option name starting with – and usually having a meaningful name, or a short option name – starting with – and having short meaningless names (usually one-letter ones) for each option.

The last two parameters in an rsync command line should be the source and the destination directories.

In this example below, we're using the following options:

-avz – a for archive mode (preserve all the attributes of each file and directory – ownership, permissions, etc), v for verbose mode (report a list of files processed by rsync) and z for data compression to speed transfers up.

–stats – this option shows a summary at the end of rsync'ing process to highlight the main stats of the job

ubuntu$ **rsync -avz --stats /tmp/dir1/ /tmp/dir2** building file list ... done file2 file3 Number of files: 4 Number of files transferred: 2 Total file size: 48 bytes Total transferred file size: 32 bytes Literal data: 32 bytes Matched data: 0 bytes File list size: 87 File list generation time: 0.001 seconds File list transfer time: 0.000 seconds Total bytes sent: 221 Total bytes received: 64 sent 221 bytes received 64 bytes 570.00 bytes/sec total size is 48 speedup is 0.17

Stats are self-explanatory, and you can see that although there were 4 files found in source directory /tmp/dir1, only 2 files were transferred into /tmp/dir2 because /tmp/dir2 already had one of the files.

That's all I have for you today, in the next post on rsync I'll show you some more advanced uses of this command. For the time being, read **man rsync** or even **rsync –help** on your system to get an idea of how really powerful this tool is.

Until next time – good luck with your Unix experiments!

Related books

If you're interested in learning more, you should consider buying the following books:

Backup & Recovery

See also:

- **How to compare directories in Unix**
- **List subdirectories of a Unix directory**
- **Finding large files and directories**

List Installed Packages on Your Ubuntu Linux

If you're interested in what exactly your Ubuntu system has got installed, there's a command you can use to list the packages along with their versions and short descriptions.

How packages information is stored in Ubuntu

Essentially being a fork of the Debian Linux, Ubuntu inherited quite a lot of things from it. One of them is the way packages are installed and managed.

dpkg (Debian Package Manager) is a command found in every Ubuntu installation. While managing software packages, it stores all the files it depends upon in a **/var/lib/dpkg** directory. Most of these files you can look into using basic Unix commands, but there's really no need because **dpkg** does it for you.

For example, status of all the installed packages is stored in **/var/lib/dpkg/status** file.
Here's how it looks, just to give you an idea:

Package: bash Essential: yes Status: install ok installed Priority: required Section: shells Installed-Size: 2012

Maintainer: Ubuntu Core developers <ubuntu-devel-discuss@lists.ubuntu.com> Architecture: amd64 Version: 3.2-0ubuntu7 Replaces: bash-doc (<= 2.05-1), bash-completion Depends: base-files (>= 2.1.12), debianutils (>= 2.15) Pre-Depends: libc6 (>= 2.5-0ubuntu1), libncurses5 (>= 5.4-5) Suggests: bash-doc Conflicts: bash-completion Conffiles: /etc/skel/.bashrc 52acca91b52f797661c89b181809b9f3 /etc/skel/.profile 7d97942254c076a2ea5ea72180266420 /etc/skel/.bash_logout 22bfb8c1dd94b5f3813a2b25da67463f /etc/bash.bashrc 860d464fca66fff1af4993962a253611 /etc/bash_completion c8bce25ea68fb0312579a421df99955c /etc/skel/.bash_profile d1a8c44e7dd1bed2f3e75d1343b6e4e1 obsolete Description: The GNU Bourne Again SHell Bash is an sh-compatible command language interpreter that executes commands read from the standard input or from a file. Bash also incorporates useful features from the Korn and C shells (ksh and csh). .

As you can see, there’s all the possible information about bash package (the Bourne Again Shell), but you usually don’t need to know this much, so instead we’ll use dpkg command to confirm what packages are installed and which ones are not.

List installed packages with dpkg

The easiest way to confirm the list of packages installed on your Ubuntu OS is to run dpkg -l command. The output is quite long, so I’ll only show you a fragment of it:

```
ubuntu# dpkg -l | more
Desired=Unknown/Install/Remove/Purge/Hold |
Status=Not/Installed/Config-files/Unpacked/Failed-config/Half-installed |/ Err?=(none)/Hold/Reinst-required/X=both-problems (Status,Err: uppercase=bad) ||/ Name
Version Description +++-
==========================-
=========================================-
========================================  ii
adduser
3.100 Add and remove users and groups ii  alsa-base
```

```
1.0.13-3ubuntu1                                 ALSA driver
configuration files ii  alsa-utils
1.0.13-1ubuntu5                                 ALSA utilities
ii    apache2
2.2.3-3.2ubuntu2.1                              Next generation,
scalable, extendable web se rc  apache2-common
2.0.55-4ubuntu4                                 next generation,
scalable, extendable web se ii  apache2-doc
2.2.3-3.2ubuntu2.1                              documentation for
apache2   ii     apache2-mpm-prefork
2.2.3-3.2ubuntu2.1                              Traditional model
for Apache HTTPD 2.1 ii  apache2-utils
2.2.3-3.2ubuntu2.1                              utility programs
for  webservers  ii   apache2.2-common
2.2.3-3.2ubuntu2.1                              Next generation,
scalable, extendable web se ii  apt
0.6.46.4ubuntu10                                Advanced front-
end  for  dpkg  ii   apt-utils
0.6.46.4ubuntu10                                APT utility
programs  ii   aptitude
0.4.4-1ubuntu3                                  terminal-based
apt  frontend  ii   at
3.1.10ubuntu4                                   Delayed job
execution and batch processing ii  autoconf
2.61-3                                          automatic
configure script builder ii  automake1.4                1.4-
p6-12                                   A tool for generating
GNU Standards-complian ii  automake1.9
1.9.6+nogfdl-3ubuntu1                           A tool for
generating  GNU  Standards-complian  ii    autotools-
dev             20060920.1
Update  infrastructure  for  config.{guess,sub}  ii
awstats
6.5+dfsg-1ubuntu3                               powerful and
featureful web server log analy ii  base-files
4ubuntu2                                        Debian base
system miscellaneous files
```

The legend at the very top of the output explains the first 3 characters of each line in the dpkg output, the symbols there confirm whether each package is expected to be installed, and

whether it’s actually installed or partially installed:

```
Desired=Unknown/Install/Remove/Purge/Hold          |
Status=Not/Installed/Config-files/Unpacked/Failed-config/Half-
installed |/ Err?=(none)/Hold/Reinst-required/X=both-problems
(Status,Err: uppercase=b ||/ Name
Version                                      Description +++-
==========================-
=============================================-
=============================================    ii
adduser
3.100                                         Add and remove
users and groups
```

A first letter of each option is used, so ii for the adduser package in this example means that the desired state for this package is “Installed” (first i) and the actual status is “Installed” as well. That’s the normal condition for most of your packages.

As you can also see, each line shows you the version of each package you have and provides a brief description of what a package is used for.

That’s it, this should be a good start for your Ubuntu exploration, I’ll post a few more things about **dpkg** in the future.

Related books

If you want to learn more, here’s a great book:

Ubuntu Kung

Fu

See also:

- **Debian @ Wikipedia**
- **Finding out the release version of your Unix**
- **apt-get behind proxy**

How To Use This Website

For all the new visitors of this blog, I've just created a Using Unix Tutorial page which is linked from the sidebar.

Please feel free to have a look and let me know if there's any more introductory information I can add to make it even easier for you to find and get exactly what you came looking.

Updating Values of Your Shell Variables in Unix

If you're just getting started with your **Unix scripting** or new to Unix shell, today's little tip will make your life much easier: I'm going to show you how to update your shell variables.

Updating shell variable in Unix

If you have a text variable in your script and would like to append some text to it or somehow process this value, it's perfectly normal to reuse the current variable value using a syntax like this:

ubuntu$ **echo $VAR1** hello ubuntu$ **VAR1=$VAR1" world!"** ubuntu$ **echo $VAR1** hello world!

You see? It's this easy!

Common cases of reusing variable values in Unix shells

Most frequently, I use the technique described above to update some of my environment variables. I'm sure you'll find them useful too.

Updating the PATH variable

PATH is the user environment variable responsible for the directories where Unix shell looks for executable commands every time you type something. Quite often you get a "file not found" error not because there isn't such a command installed in your OS, but simply because your PATH variable has not a correpct path to a directory with that command.

Here's an example from a standard user on one of my Red Hat Enterprise Linux systems.

I like the runlevel command, it's quite simple and can be useful in scripts. When I attempt to run it, here's what happens:

redhat$ **runlevel** bash: runlevel: command not found

Like I said earlier, it's most likely because my PATH variable doesn't have the /sbin directory which is where this command

resides. Let’s confirm this:

redhat$ **echo $PATH** /usr/lib/qt-3.3/bin:/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin

Now it’s time to update the PATH and append the /sbin path to it:

redhat$ **PATH=$PATH:/sbin**

runlevel command will be found now and here’s a proof:

redhat$ **runlevel** N 5

Updating the MANPATH variable

Another good example is the **MANPATH** variable, which contains the list of directories with manpages which **man command** uses.

If some command doesn’t have its man page in neither of the directories specified by MANPATH, you’ll get an error.

Here’s an example from one of the Solaris systems, I’m looking for a Veritas vxprint command man page:

solaris$ **man vxprint** No manual entry for vxprint. solaris$ **echo $MANPATH** /usr/man:/opt/sfw/man:/usr/perl5/man

Let’s add the /opt/VRTS/man to the MANPATH variable:

bash-2.03# **MANPATH=$MANPATH:/opt/VRTS/man** solaris$ **echo $MANPATH** /usr/man:/opt/sfw/man:/usr/perl5/man:/opt/VRTS/man

It’s bound to work now:

solaris$ **man vxprint** Reformatting page. Please Wait... done ...

That’s all I wanted to share with you today. Hope you liked the tip, and as always – feel free to ask any questions!

See also

- **Environment variables in Unix**
- **Using variables in shell scripts**
- **Unix scripts: basic arithmetic operations**
- Unix commands
- Unix Basic Commands

System Resources Usage When Running Unix commands

In my system administration job, I come across this particular requirement quite often: run a command, then confirm how long it took to complete the job. You guessed it right – there's a very easy way to do it.

Using the time command

What you need to do is to use the time command, which takes as a parameter the full command line you need to run. Upon completion of the command, time will report the system resources usage during the execution. All the reported numbers are presented in a simple table.

Here's an example: running du command to confirm the size of a directory while timing the effort and reporting the stats:

```
ubuntu# time du -sk /var 4228720 /var real    0m17.747s
user    0m0.010s sys     0m0.080s
```

As you can see, first the du command output is shown, and then the table with real, user and system times reported.

See also:

- **Time and date in Unix scripts**
- **Finding large files and directories**
- **Unix commands section**

How To Parse Text Files Line by Line in Unix scripts

I'm finally back from my holidays and thrilled to be sharing next of my Unix tips with you!

Today I'd like to talk about parsing text files in Unix shell scripts. This is one of the really popular areas of scripting, and there's a few quite typical limitations which everyone comes across.

Reading text files in Unix shell

If we agree that by "reading a text file" we assume a procedure of going through all the lines found in a clear text file with a view to somehow process the data, then **cat command** would be the simplest demonstration of such procedure:

redhat$ **cat /etc/redhat-release** Red Hat Enterprise Linux Client release 5 (Tikanga)

As you can see, there's only one line in the **/etc/redhat-release** file, and we see what this line is.

But if you for whatever reason wanted to read this file from a script and assign the whole release information line to a Unix variable, using cat output would not work as expected:

bash-3.1$ **for i in `cat /etc/redhat-release`; do echo $i; done;** RedHat Enterprise Linux Client release 5 (Tikanga)

Instead of reading a line of text from the file, our one-liner splits the line and outputs every word on a separate line of the output. This happens because of the shell syntax parsing – Unix shells assume space to be a delimiter of various elements in a list, so when you do a for loop, Unix shell interpreter treats each line with spaces as a list of elements, splits it and returns elements one by one.

How to read text files line by line

Here's what I decided: if I can't make Unix shell ignore the spaces between words of each line of text, I'll disguise these spaces. Since my solution was getting pretty bulky for a one-liner, I've made it into a script. Here it is:

bash-3.1$ **cat /tmp/cat.sh** #!/bin/sh FILE=$1 UNIQUE='-={GR}=-' # if [-z "$FILE"]; then exit; fi; # for LINE in `sed "s/ /$UNIQUE/g" $FILE`; do LINE=`echo $LINE | sed "s/$UNIQUE/ /g"`; echo $LINE; done;

As you can see, I've introduced an idea of a UNIQUE variable, something containing a unique combination of characters which I can use to replace spaces in the original string. This variable needs to be a unique combination in a context of your text files, because later we turn the string back into its original version, replacing all the instances of $UNIQUE text with plain spaces.

Since most of the needs of mine required such functionality for a relatively small text files, this rather expensive (in terms of CPU cycles) approach proved to be quite usable and pretty fast.

Update: please see comments to this post for a much better approach to the same problem. Thanks again, Nails!

Here's how my script would work on the already known

/etc/redhat-release file:

bash-3.1$ **/tmp/cat.sh /etc/redhat-release** Red Hat Enterprise Linux Client release 5 (Tikanga)

Exactly what I wanted! Hopefully this little trick will save some of your time as well. Let me know if you like it or know an even better one yourself!

Related books

If you want to learn more, here's a great book:

Classic Shell Scripting

See also:

- **Using variables in scripts**
- **Unix scripting: time and date**

Environment Variables in Unix

Each process in Unix has its own set of environment variables. They're called environment variables because the default set of such variables consists mostly of session-wide variables used for configuration purposes.

From the point of a Unix shell though, environment variables

can be accessed the same way as any other variable.

Common environment variables in Unix

Most well known environment variables are the following:

- **USER** – username of a Unix user
- **HOME** – full path to a user's home directory
- **TERM** – terminal or terminal emulator used by a current user
- **PATH** – list of directories searched for executable files when you type a command in Unix shell
- **PWD** – current directory

Example of using environment variables

Using Unix username to control the flow of a script

Sometimes it's quite useful to double-check the username of whoever called your script – maybe you'll want to provide different functionality for different users. A common use of such scenario is many commands which are not meant to be run by anyone except superuser (root). If you try running them as a normal user, you'll be told right away that you have to be root in order to use them.

In Unix scripts, the opposite functionality is more useful: making sure you don't run a script as root. Here's one way of doing it:

```
#!/bin/bash # echo "- Verifying the current user..." if [ "$USER" = "root" ]; then         echo "You are ROOT, please run as a normal user";         exit else         echo "User $USER, script is ready to continue" fi         echo "Work in progress..."
```

And that's how it would work:

ubuntu$ **/tmp/script.sh** - Verifying the current user... User

greys, script is ready to continue Work in progress...

If I use **sudo** in Ubuntu to run the script as root, I'll get a warning and the script will end:

$ **sudo /tmp/script.sh** - Verifying the current user... You are ROOT, please run as a normal user

Getting full list of environment variables

In case you feel like exploring, you can use the **env command** to get a full list of currently set environment variables (the output in this example is abridged):

ubuntu$ **env** TERM=xterm SHELL=/bin/bash USER=greys MAIL=/var/mail/greys PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/bin/X11:/usr/games PWD=/home/greys EDITOR=vim ..

That's all I wanted to share with you today. Let me know how exactly you'd like me to further expand and cover this topic – more posts will definitely follow!
h3>Recommended books:


See also:

- **Using variables in Unix**
- Environment variables – article @ Wikipedia
- Unix scripting example: time and date

List Groups A User Belongs To In Unix

Still seeing this request quite frequently in my website logs, I thought I'd expand the topic a bit more, given the fact that I've somehow left out the most obvious way to confirm the group membership for a Unix user.

Group membership in Unix

Every Unix-like OS is bound to have a group command, which is aimed to help you confirm Unix group membership for any user known to your system.

The easiest it to run it and specify a username in command line. The result is a list of Unix groups:

ubuntu$ **groups greys** greys : admin www-data

If you're looking at confirming the membership for a few users, you can specify usernames in the same command line:

ubuntu$ **groups greys root** greys : admin www-data root : root

There you go – hopefully this satisfies your interest. Good luck with Unix experiments!

See also:

- **Finding the primary Uinx group a user belongs to**
- **Find files belonging to a user or Unix group**
- **How to find who owns a file in Unix**

Unix Tutorial Digest: Interesting Links #1

Every week there's a few announcements or articles which I find particularly interesting, and so I've decided to share them with you. I'm not a Unix guru (yet), but if any of the listed materials require further explanation – do feel free to ask and I'll be glad to help.

Ubuntu 8.04.1 release

About a week ago, the first update to Ubuntu 8.04 was announced – Ubuntu 8.04.1 TLS. I have completed my experiment of using Ubuntu Hardy as my desktop OS a few weeks ago, and so haven't upgraded yet – but I think this release is not so useful for anyone who's been automatically updating their system – it's just another milestone and a way to download a complete Ubuntu 8.04.1 as one image.

The highlights for me would be Firefox upgraded to the final 3.0 release and Gnome upgrade (it's 2.22.2 in this release).

Gentoo Linux 2008.0 release

For some of you, it's probably been a long-awaited release. Move to 2.6.24 kernel provided support for much more hardware, and this is bound to look good with the updated and much improved Gentoo installer.

Read more in the official Gentoo Linux 2008.0 announcement.

Cache poisoning vulnerability in DNS

Dan Kaminsky has found quite a nasty weakness in DNS implementations: deficiencies in the DNS protocol and common DNS implementations
facilitate DNS cache poisoning attacks.

Thanks to the seriousness of the problem and a great coordination, most of the vendors were given the time to publish a fix, so the Vulnerability Note VU#800113 contains a comprehensive list of vulnerable implementations of DNS (both server and client sides are affected, by the way!) and links to fixes provided by various vendors.

Whether you're managing a server farm or just a Linux desktop – be sure to update!

Wine 1.1.1 release

Things are going much faster with Wine development after the 1.0 release – it didn't take long for the 1.1 to appear, and now almost every other week brings another great update with tons of bugs fixed.

Wine 1.1.1 release includes more than 50 bugfixes and hundreds of changes since Wine 1.1.0, notably the fixes for Adobe Photoshop CS3 and Microsoft Office 2007 installers, as well as improved video playback and many other improvements.

How To Determine Physical Memory Size in Linux

If you're logged in at some remote Linux system and need to quickly confirm the amount of available memory, there's a few commands you will find quite useful.

free – free and used memory stats

free command is the most obvious choice for a first command when it comes to your RAM.

Simply run it without any parameters, and it will show you something like this:

```
ubuntu# free              total       used       free     shared    buffers      cached Mem:        4051792    4024960      26832          0      63768    3131532 -/+ buffers/cache:     829660    3222132 Swap:      4096492      43212    4053280
```

For this exercise, you're only interested in the "total" column of the first line. 4051792 confirms that my home PC seems to have around 4Gb of memory available for Ubuntu to use.

Using dmesg to check memory size as recognized by Linux kernel

dmesg command shows you the last status messages reported by your OS kernel, and since every boot procedure includes scanning the hardware and confirming the devices and resources recognized by the kernel, you can see some basic information by using **dmesg**.

For our purpose, we need to filter out the memory stats:

```
ubuntu# dmesg | grep Memory [    18.617904] Memory: 4043492k/5242880k available (2489k kernel code, 150360k reserved, 1318k data, 320k init)
```

Once again, the overall amount of memory confirms that 4Gb of RAM were still found during the last time my PC booted up.

Using /proc/meminfo to confirm the RAM size

/proc/meminfo is one of the special files managed by Linux kernel. It's a clear text presentation of the most vital memory stats of your system (this means you can do something like **cat /proc/meminfo** to see all the parameters)

This is what you need to do to get the total size of your physical memory:

ubuntu# **grep MemTotal /proc/meminfo** MemTotal: 4051792 kB

That’s it for today, enjoy!