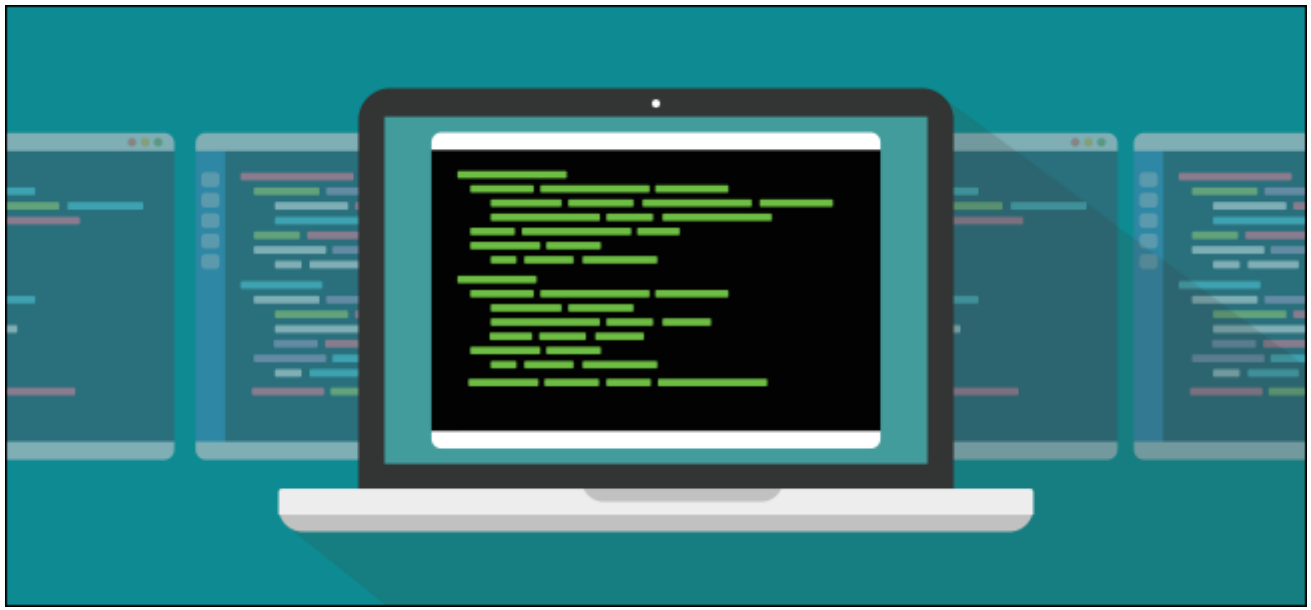
How-To Geek

Jak ustawić zmienne środowiskowe w Bash w systemie Linux

DAVE MCKAY @thegurkha
28 KWIETNIA 2020 R., GODZ. 8.00 CZASU EDT



fatmawati achmad zaenuri / Shutterstock

W systemie Linux istnieje więcej niż jeden typ zmiennej środowiskowej. Dowiedz się, jak je zobaczyć, tworzyć je dla lokalnych i zdalnych loginów i sprawić, by przetrwały restarty.

Jak działają zmienne środowiskowe

Po uruchomieniu okna terminala i powłoki w nim znajduje się zbiór zmiennych, aby upewnić się, że powłoka jest poprawnie skonfigurowana. Zmienne te zapewniają również dostępność wszelkich informacji, do których może być potrzebne okno terminala i powłoka. Łącznie zmienne te przechowują ustawienia definiujące środowisko, które można znaleźć w oknie terminala, aż do wyglądu wiersza polecenia. Oczywiście są one nazywane zmiennymi środowiskowymi.

Niektóre zmienne środowiskowe są ogólnosystemowe lub globalne. Inne dotyczą całej sesji i mogą być widoczne tylko dla Ciebie. Inni nie mogą odwoływać się do zmiennych środowiskowych sesji. Istnieje trzeci zestaw zmiennych środowiskowych zdefiniowanych w powłoce. Ustawienia regionalne, strefa czasowa i klawiatura, zestaw przeszukiwanych katalogów, gdy powłoka próbuje znaleźć polecenie, oraz domyślny edytor są przechowywane w zmiennych środowiskowych powłoki.

Pokażemy ci, jak zobaczyć zmienne środowiskowe, które istnieją w twoim systemie, i opiszemy, jak stworzyć własne. Pokażemy Ci również, jak udostępnić je procesom potomnym i jak zachować je podczas ponownego uruchamiania.

Środowiska i dziedziczenie

Po uruchomieniu powłoka przechodzi przez fazę inicjalizacji. W tym momencie odczytuje zmienne środowiskowe, które definiują środowisko powłoki.

Kiedy program lub polecenie jest uruchamiane z tej powłoki - znanej jako proces potomny - dziedziczy środowisko procesu macierzystego - ale uważaj! Jak zobaczymy, możesz tworzyć

zmienne, które nie zostaną dodane do twojego środowiska, aby nie zostały odziedziczone przez proces potomny.

Jeśli proces potomny jest powłoką, powłoka ta zainicjuje się z własnego, świeżego zestawu zmiennych. Jeśli więc zmienisz wiersz polecenia w bieżącej powłoce, a następnie uruchomisz powłokę podrzędną, powłoka podrzędna nie odziedziczy zmodyfikowanego wiersza polecenia nadrzędnego.

Globalne zmienne środowiskowe

Zgodnie z konwencją zmienne środowiskowe otrzymują nazwy pisane wielkimi literami. Oto niektóre globalne zmienne środowiskowe i ich wartości:

- **SHELL:** Nazwa powłoki, która uruchomi się po otwarciu okna terminala. W większości dystrybucji Linuksa będzie to bash, chyba że zmienisz go na domyślny.
- **TERM:** Okna terminala są w rzeczywistości emulacjami terminala sprzętowego. Zawiera typ terminala sprzętowego, który będzie emulowany.
- **USER:** nazwa użytkownika bieżącej osoby korzystającej z systemu.
- **PWD:** Ścieżka do bieżącego katalogu roboczego.
- **OLDPWD:** katalog, w którym byłeś przed przejściem do bieżącego katalogu roboczego.
- **LS_COLORS:** Lista kodów kolorów używanych przez `ls` podświetlenie różnych typów plików .
- **MAIL:** If the `mail`system has been set up on your Linux computer (by default, it isn't), this will hold the path to the current user's mailbox.
- **PATH:** A list of directories that the shell will search through to find command executables.

- **LANG:** The language, localization, and character encoding settings.
- **HOME:** The home directory of the current user.
- **_:** The underscore (_) environment variable holds the last command that was typed.

RELATED: *How to Use pushd and popd on Linux*

We can see what some of these are set to using nothing more sophisticated than `echo`, which will write the values to the terminal window. To see the *value* held by an environment variable, you need to add a dollar sign (`$`) to the start of its name.

A nice touch is that you can use tab completion to fill in the environment variable name for you. Type a few letters of the name and hit Tab. The name of the variable is completed by the shell. If that doesn't happen, you'll need to type a few more letters to distinguish the environment variable from other commands with names that start with those same letters:

```
echo $SHELL
```

```
echo $LANG
```

```
echo $HOME
```

```
echo $PWD
```

To create your own *global* environment variables, add them to the `/etc/environment` file. You'll need to use `sudo` to edit this file:

```
sudo gedit /etc/environment
```

To add an environment variable, type its name, an equal sign (=), and the value you want the environment variable to hold. Don't space before or after the equal sign (=). The name of the environment variable can contain letters, an underscore (_), or numbers. However, the first character of a name cannot be a number.

If there are spaces in the value, be sure you enclose the entire value in quotation marks (").

Save the file, and then log out and back in again. Use `echo` to test that a new variable exists and holds the value you set:

```
echo $WEBSITE
```

Because it's a global environmental variable, and available to everyone, user `mary` can reference the environment variable when she next logs in:

```
echo $WEBSITE
```

To see all the environment variables at once, type `printenv`. There's a lot of output, so it makes sense to pipe it through `sort`, and then into `less`:

```
printenv | sort | less
```

The sorted list of environment variables is displayed for us in `less`.

We can pipe the output through `grep` to look for environment variables related to a particular topic.

```
printenv | grep GNOME
```

RELATED: *How to Edit Text Files Graphically on Linux With gedit*

Shell Environment Variables

These are some of the shell environment variables used in `bash` to dictate or record its behavior and functionality. Some of the values are updated as you use the terminal. For example, the `COLUMNS` environment variable will be updated to reflect changes you might make to the width of the terminal window:

- **BASHOPTS:** The command-line options that were used when `bash` was launched.
- **BASH_VERSION:** The `bash` version number as a string of words and numbers.
- **BASH_VERSINFO:** The `bash` version as a digit.
- **COLUMNS:** The current width of the terminal window.
- **DIRSTACK:** The directories that have been added to the directory stack by the `pushd` command.
- **HISTFILESIZE:** Maximum number of lines permitted in the `history` file.
- **HISTSIZE:** Number of lines of `history` allowed in memory.
- **HOSTNAME:** The hostname of the computer.
- **IFS:** The Internal Field Separator used to separate input on the command line. By default, this is a space.
- **PS1:** The `PS1` environment variable holds the definition for the primary, default, and command prompt. A set of tokens called escape sequences can be included in the definition of your command prompt. They represent such things as the host- and username, the current working directory, and the time.
- **PS2:** When a command spans more than one line and more input is expected, the secondary command prompt is shown. The `PS2` environment variable holds the definition of this secondary prompt, which, by default, is the greater than sign (`>`).

- **SHELLOPTS:** Shell options you can set using the `set` option.
- **UID:** The User Identifier of the current user.

RELATED: *How to Use pushd and popd on Linux*

Let's check a few of these shell variables:

```
echo $BASH_VERSION
```

```
echo $HOSTNAME
```

```
echo $COLUMNS
```

```
echo $HISTFILESIZE
```

```
echo $UID
```

For the sake of completeness, here are the tokens you can use in the command prompt definitions:

- **\t:** The current time, formatted as HH:MM:SS.
- **\d:** The current date, expressed as weekday, month, date.
- **\n:** A new-line character.
- **\s:** The name of your shell.
- **\W:** The name of your current working directory.
- **\w:** The path to your current working directory.
- **\u:** The username of the person who's logged in.
- **\h:** The hostname of the computer.
- **\#:** Each command within a shell is numbered. This allows you to see the number of the command in your command prompt. This is not the same as the number the command will have in the `history` list.
- **\$:** Sets the final character of the prompt to a dollar sign (`$`) for a regular user, and a hash symbol (`#`) for the root user. This works by checking the UID of the user. If it's zero, the user is root.

You'll find the definition of your `PS1` environment variable in your `.bashrc` file.

Creating Session Environment Variables

To create environment variables for your own use, add them to the bottom of your `.bashrc` file. If you want to have the environment variables available to remote sessions, such as SSH connections, you'll need to add them to your `.bash_profile` file, as well.

The format of the environment variable definition is the same for both files. To add a definition to your `.bash_profile` file, type this in your home directory:

```
gedit .bashrc
```

We've added an environment variable called `INHERITED_VAR`. Note the word "export" at the start of the line.

Save and close your file after you finish editing. You could log out and back in again, or you can cause the shell to re-read the `.bash_profile` file using the dot command (`.`) like this:

```
. .bashrc
```

Now, let’s create an environment variable on the command line:

```
LOCAL_VAR="This session only"
```

If we use `echo`, we can see that both environment variables are accessible to us:

```
echo $LOCAL_VAR
```

```
echo $INHERITED_VAR
```

You'll notice the definition of the `INHERITED_VAR` environment variable had the word "export" at the start of the line. This means the environment variable will be inherited by child processes of the current shell. If we launch another one using the `bash` command, we can check the two variables again, from inside the child shell:

```
bash
```

```
echo $LOCAL_VAR
```

```
echo $INHERITED_VAR
```

As you can see, the `INHERITED_VAR` is accessible in the child shell, but `LOCAL_VAR` is not. We simply get a blank line.

Although “export” adds the environment variable part to the environment that child processes inherit, `INHERITED_VAR` is not a global environment variable. For example, user `mary` cannot reference it:

```
echo $INHERITED_VAR
```

To close our child `bash` session, we use `exit`:

```
exit
```

Inherited environments affect scripts, too. Here's a simple script that writes the values of our three environment variables to the terminal window:

```
#!/bin/bash

echo "WEBSITE" $WEBSITE
echo "LOCAL_VAR" $LOCAL_VAR
echo "INHERITED_VAR" $INHERITED_VAR
```

This was saved to a file called `envtest.sh`, and then made executable with the following:

```
chmod +x envtest.sh
```

When we run the script, it can access two out of three environment variables:

```
./envtest.sh
```

The script can see the `WEBSITE` global environment variable and the `INHERITED_VAR` exported environment variable. It cannot access `LOCAL_VAR`, even though the script is running in the same shell where the variable was created.

If we need to, we can export an environment variable from the command line. We'll do that to our `LOCAL_VAR`, and then run the script again:

```
export LOCAL_VAR
```

```
./envtest.sh
```

The environment variable has been added to the environment of the current shell, and so it appears in the environment that is inherited by the script. The script can reference that environment variable, too.

Remote Connections

Global environment variables are accessible to remote login sessions, but if you want your locally defined environment variables available to you remotely, you must add them to your `.bash_profile` file. You can set the same environment variable in the `.bashrc` and `.bash_profile` files, with different values. This could be picked up by a script, say, to modify its behavior for people using the system locally or remotely.

(At the risk of confusing matters, there's also a `.profile` file. It can hold environment variable definitions, too. However, the `.profile` file is not read if the `.bash_profile` file is present. So, the safest thing to do—and the `bash`-compliant way—is to use the `.bash_profile` file.)

To edit the `.bash_profile` file, we'll use `gedit` again:

```
gedit .bash_profile
```

We’re going to add the same environment variable with the same value we used before.

Save your changes and close `gedit`.

On another computer, we'll make an SSH connection to the test computer.

```
ssh dave@howtogeek.local
```

Once we're connected, we'll run the script once more:

```
./envtest.sh
```

The `.bash_profile` file has been read as part of the initialization of the remote login, and the `INHERITED_VAR` environment variable is accessible to us and the script.

Unsetting an Environment Variable

To unset an environment variable use the `unset` command. If we unset the global environment variable, `WEBSITE`, and the exported environment variable, `INHERITED_VAR`, they'll no longer be available on the command line, nor in child processes:

```
unset WEBSITE
```

```
unset INHERITED_VAR
```

```
./envtest.sh
```

```
echo $WEBSITE
```

A point to note is this only changes the availability of global environment variables for you in this session. Another person who's logged in simultaneously will still be able to access his instance of that global environment variable. His instance was initialized and read from the `/etc/environment` file during his login process, and is independent of anyone else's copy of the variable.

As an example, user `mary` can still access the `WEBSITE` environment variable and read its value, even though user `dave` has `unset` it in his session:

```
echo $WEBSITE
```

Environmental Control

Environment variables can be used to let scripts and applications know how they should behave. They can be used to store settings or small amounts of data. For example, a script can populate an environment with a value that can be referenced by other scripts without having to write them to a file.

READ NEXT

› What Is a “Surveillance” or “NAS” Hard Drive?

› How Do Music Identification Apps Like Shazam Work?

› How to Use a Digital Camera as a Webcam

› How to Remove Email Accounts From the Mail App on iPhone and iPad

› How to Set Up an Old Laptop for Kids

DAVE MCKAY

Dave McKay po raz pierwszy używał komputerów, kiedy perforowana taśma papierowa była w modzie i od tego czasu programuje. Po ponad 30 latach pracy w branży IT jest teraz pełnoetatowym dziennikarzem technologicznym. W trakcie swojej kariery pracował jako niezależny programista, kierownik międzynarodowego zespołu programistycznego, kierownik projektu usług informatycznych, a ostatnio jako inspektor ochrony danych. Dave jest ewangelistą Linuksa i orędownikiem open source. PRZECZYTAJ PEŁNE BIOGRAFIE »

Powyższy artykuł może zawierać linki partnerskie, które pomagają wspierać maniakiem porad.

How-To Geek to miejsce, w którym zwracasz się, gdy chcesz, aby eksperci wyjaśniali technologię. Od momentu wprowadzenia na rynek w 2006 r. Nasze artykuły zostały przeczytane ponad 1 miliard razy. Chcieć wiedzieć więcej?