When is the right time (and the wrong time) to use backticks?

Asked 16 years, 3 months ago Modified 4 months ago Viewed 18k times

15

Many beginning programmers write code like this:

```
sub copy_file ($$) {
  my $from = shift;
  my $to = shift;

  `cp $from $to`;
}
```

Is this bad, and why? Should backticks ever be used? If so, how?

perl

Share Improve this question Follow

asked Sep 22, 2008 at 16:12
raldi
22.1k ● 33 ● 78 ● 86

14 Answers

Sorted by: Highest score (default)

22

A few people have already mentioned that you should only use backticks when:

- You need to capture (or supress) the output.
- There exists no built-in function or Perl module to do the same task, or you have a good reason not to use the module or built-in.
- You sanitise your input.
- You check the return value.

Unfortunately, things like checking the return value *properly* can be quite challenging. Did it die to a signal? Did it run to completion, but return a funny exit status? The standard ways of trying to interpret `$?` are just awful.

I'd recommend using the IPC::System::Simple module's `capture()` and `system()` functions rather than backticks. The `capture()` function works just like backticks, except that:

- It provides detailed diagnostics if the command doesn't start, is killed by a signal, or returns an unexpected exit value.
- It provides detailed diagnostics if passed tainted data.
- It provides an easy mechanism for specifying acceptable exit values.
- It allows you to call backticks without the shell, if you want to.
- It provides reliable mechanisms for avoiding the shell, even if you use a single argument.

The commands also work consistently across operating systems and Perl versions, unlike Perl's built-in `system()` which may not check for tainted data when called with multiple arguments on older versions of Perl (eg, 5.6.0 with multiple arguments), or which may call the shell anyway under Windows.

As an example, the following code snippet will save the results of a call to `perldoc` into a scalar, avoids the shell, and throws an exception if the page cannot be found (since perldoc returns 1).

```
#!/usr/bin/perl -w
use strict;
use IPC::System::Simple qw(capture);

# Make sure we're called with command-line arguments.
@ARGV or die "Usage: $0 arguments\n";

my $documentation = capture('perldoc', @ARGV);
```

IPC::System::Simple is pure Perl, works on 5.6.0 and above, and doesn't have any dependencies that wouldn't normally come with your Perl distribution. (On Windows it depends upon a Win32:: module that comes with both ActiveState and Strawberry Perl).

Disclaimer: I'm the author of IPC::System::Simple, so I may show some bias.

Share Improve this answer Follow

answered Sep 23, 2008 at 3:13

pjf
6,001 ● 27 ● 43

FYI if someone wants to use something that is in the perl base distribution (i.e., you don't want to install modules or require your downstream to install a particular module), look at IPC::Cmd, which is more complex but not oppressively so. – raindog308 Nov 15, 2014 at 17:08

The rule is simple: never use backticks if you can find a built-in to do the same job, or if their is a robust module on the CPAN which will do it for you. Backticks often rely on

11

unportable code and even if you untaint the variables, you can still open yourself up to a lot of security holes.

Never use backticks with user data unless you have very tightly specified what is allowed (not what is disallowed -- you'll miss things)! This is very, very dangerous.

Share Improve this answer Follow

answered Sep 22, 2008 at 16:30
Ovid
11.7k ● 9 ● 49 ● 76

6

Backticks should be used if and only if you need to capture the output of a command. Otherwise, `system()` should be used. And, of course, if there's a Perl function or CPAN module that does the job, this should be used instead of either.

In either case, two things are strongly encouraged:

First, **sanitize all inputs:** Use Taint mode (`-T`) if the code is exposed to possible untrusted input. Even if it's not, make sure to handle (or prevent) funky characters like space or the three kinds of quote.

Second, **check the return code** to make sure the command succeeded. Here is an example of how to do so:

```
my $cmd = "./do_something.sh foo bar";
my $output = `$cmd`;

if ($?) {
   die "Error running [$cmd]";
}
```

Share
Improve this answer
Follow

edited Aug 14 at 13:47
smonff
3,489 ● 3 ● 39 ● 48

answered Sep 22, 2008 at 16:13
raldi
22.1k ● 33 ● 78 ● 86

5

Another way to capture stdout(in addition to pid and exit code) is to use [IPC::Open3](http://search.cpan.org/~nwclark/perl-5.8.8/lib/IPC/Open3.pm) possibily negating the use of both system and backticks.

Share
Improve this answer
Follow

edited Jun 20, 2020 at 9:12
Community Bot
1 ● 1

answered Sep 22, 2008 at 17:16
Erik Johansen
130 ● 2 ● 8

5

Use backticks when you want to collect the output from the command.

Otherwise `system()` is a better choice, especially if you don't need to invoke a shell to handle metacharacters or command parsing. You can avoid that by passing a list to system(), eg `system('cp', 'foo', 'bar')` (however you'd probably do better to use a module for that *particular* example :))

Share

Improve this answer

Follow

edited Oct 4, 2008 at 2:35

answered Sep 22, 2008 at 16:14

Dan

63.3k ● 10 ● 66 ● 80

4

In Perl, there's always more than one way to do anything you want. The primary point of backticks is to get the standard output of the shell command into a Perl variable. (In your example, anything that the cp command prints will be returned to the caller.) The downside of using backticks in your example is you don't check the shell command's return value; cp could fail and you wouldn't notice. You can use this with the special Perl variable $?. When I want to execute a shell command, I tend to use **system**:

```
system("cp $from $to") == 0
    or die "Unable to copy $from to $to!";
```

(Also observe that this will fail on filenames with embedded spaces, but I presume that's not the point of the question.)

Here's a contrived example of where backticks might be useful:

```
my $user = `whoami`;
chomp $user;
print "Hello, $user!\n";
```

For more complicated cases, you can also use **open** as a pipe:

```
open WHO, "who|"
    or die "who failed";
while(<WHO>) {
    # Do something with each line
}
close WHO;
```

Share Improve this answer Follow

answered Sep 22, 2008 at 16:19

Commodore Jaeger

33.3k ● 4 ● 57 ● 44

3

From the "perlop" manpage:

> That doesn't mean you should go out of your way to avoid backticks when they're the right way to get something done. Perl was made to be a glue language, and one of the things it glues together is commands. Just understand what you're getting yourself into.

Share Improve this answer Follow

answered Sep 22, 2008 at 17:06

Michael Cramer
5,230 ● 1 ● 22 ● 16

3

For the case you are showing using the [File::Copy]() module is probably best. However, to answer your question, whenever I need to run a system command I typically rely on [IPC::Run3](). It provides a lot of functionality such as collecting the return code and the standard and error output.

Share Improve this answer Follow

answered Sep 22, 2008 at 18:23

Josh McAdams
605 ● 4 ● 5

2

Whatever you do, as well as sanitising input and checking the return value of your code, make sure you call any external programs with their explicit, full path. e.g. say

```
my $user = `/bin/whoami`;
```

or

```
my $result = `/bin/cp $from $to`;
```

Saying just "whoami" or "cp" runs the risk of accidentally running a command other than what you intended, if the user's path changes - which is a security vulnerability that a malicious attacker could attempt to exploit.

Share Improve this answer Follow

answered Sep 22, 2008 at 20:14

Sam Kington
1,200 ● 11 ● 14

Your example's bad because there are perl builtins to do that which are portable and usually more efficient than the backtick alternative.

1

They should be used only when there's no Perl builtin (or module) alternative. This is both for backticks and system() calls. Backticks are intended for capturing output of the executed command.

Share Improve this answer Follow

answered Sep 22, 2008 at 16:14

Vinko Vrsalovic
340k ●55 ●340 ●373

My example comes from the real world: this is exactly how beginners misuse backticks. No input checks, no return code checks, and no effort to look for a CPAN module or a built-in. – raldi Sep 22, 2008 at 16:29

Actually it's not as bad as it looks - Perl's File::Copy is horribly broken (for example it doesn't copy file permission on Unix and other stuff), so many "high profile" hackers actually use system('cp', $target, $destination) or similar stuff – moritz Sep 22, 2008 at 16:33

Even then, moritz, one of the things people ought to know is that backticks means "capture output," which is more than simply "run a command." – skiphoppy Sep 22, 2008 at 17:50

1

Backticks are only supposed to be used when you want to capture output. Using them here "looks silly." It's going to clue anyone looking at your code into the fact that you aren't very familiar with Perl.

Use backticks if you want to capture output. Use system if you want to run a command. One advantage you'll gain is the ability to check the return status. Use modules where possible for portability. In this case, File::Copy fits the bill.

Share Improve this answer Follow

answered Sep 22, 2008 at 16:26

skiphoppy
102k ●75 ●179 ●220

1

In general, it's best to use **system** instead of backticks because:

1. **system** encourages the caller to check the return code of the command.
2. **system** allows "indirect object" notation, which is more secure and adds flexibility.
3. Backticks are culturally tied to shell scripting, which might not be common among readers of the code.
4. Backticks use minimal syntax for what can be a heavy command.

One reason users might be temped to use backticks instead of **system** is to hide STDOUT from the user. This is more easily and flexibly accomplished by redirecting the STDOUT stream:

```
my $cmd = 'command > /dev/null';
system($cmd) == 0 or die "system $cmd failed: $?"
```

Further, getting rid of STDERR is easily accomplished:

```
my $cmd = 'command 2> error_file.txt > /dev/null';
```

In situations where it makes sense to use backticks, I prefer to use the **qx{}** in order to emphasize that there is a heavy-weight command occurring.

On the other hand, having Another Way to Do It can really help. Sometimes you just need to see what a command prints to STDOUT. Backticks, when used as in shell scripts are just the right tool for the job.

Share Improve this answer Follow

answered Sep 22, 2008 at 17:04
Jon Ericson
21.5k ● 12 ● 102 ● 151

Don't ever use indirect syntax: shadow.cat/blog/matt-s-trout/indirect-but-still-fatal – SineSwiper
Jun 17, 2015 at 14:22

0

Perl has a split personality. On the one hand it is a great scripting language that can replace the use of a shell. In this kind of one-off I-watching-the-outcome use, backticks are convenient.

When used a programming language, backticks are to be avoided. This is a lack of error checking and, if the separate program backticks execute can be avoided, efficiency is gained.

Aside from the above, the system function should be used when the command's output is not being used.

Share Improve this answer Follow

answered Sep 22, 2008 at 21:03
user19115
9 ● 1

0

Backticks are for amateurs. The bullet-proof solution is a "Safe Pipe Open" (see "man perlipc"). You exec your command in another process, which allows you to first futz with STDERR, setuid, etc. Advantages: it does *not* rely on the shell to parse @ARGV, unlike open("$cmd $args|"), which is unreliable. You can redirect STDERR and change user priviliges without changing the behavior of your main program. This is more verbose than backticks but you can wrap it in your own function like run_cmd($cmd,@args);

```
sub run_cmd {
  my $cmd = shift @_;
  my @args = @_;

  my $fh; # file handle
  my $pid = open($fh, '-|');
  defined($pid) or die "Could not fork";
  if ($pid == 0) {
    open STDERR, '>/dev/null';
    # setuid() if necessary
    exec ($cmd, @args) or exit 1;
  }
  wait; # may want to time out here?
  if ($? >> 8) { die "Error running $cmd: [$?]"; }
  while (<$fh>) {
    # Have fun with the output of $cmd
  }
  close $fh;
}
```

Share

Improve this answer

Follow

edited Sep 23, 2008 at 19:14

answered Sep 23, 2008 at 18:58

user21308

11 ● 1