NAME

Capture::Tiny − Capture STDOUT and STDERR from Perl, XS or external programs

VERSION

version 0.48

SYNOPSIS

```
  use Capture::Tiny ':all';

  # capture from external command

  ($stdout, $stderr, $exit) = capture {
    system( $cmd, @args );
  };

  # capture from arbitrary code (Perl or external)

  ($stdout, $stderr, @result) = capture {
    # your code here
  };

  # capture partial or merged output

  $stdout = capture_stdout { ... };
  $stderr = capture_stderr { ... };
  $merged = capture_merged { ... };

  # tee output

  ($stdout, $stderr) = tee {
    # your code here
  };

  $stdout = tee_stdout { ... };
  $stderr = tee_stderr { ... };
  $merged = tee_merged { ... };
```

DESCRIPTION

Capture::Tiny provides a simple, portable way to capture almost anything sent to STDOUT or STDERR, regardless of whether it comes from Perl, from XS code or from an external program. Optionally, output can be teed so that it is captured while being passed through to the original filehandles. Yes, it even works on Windows (usually). Stop guessing which of a dozen capturing modules to use in any particular situation and just use this one.

USAGE

The following functions are available. None are exported by default.

capture

```
  ($stdout, $stderr, @result) = capture \&code;
  $stdout = capture \&code;
```

The `capture` function takes a code reference and returns what is sent to STDOUT and STDERR as well as any return values from the code reference. In scalar context, it returns only STDOUT. If no output was received for a filehandle, it returns an empty string for that filehandle. Regardless of calling context, all output is captured — nothing is passed to the existing filehandles.

It is prototyped to take a subroutine reference as an argument. Thus, it can be called in block form:

```
($stdout, $stderr) = capture {
  # your code here ...
};
```

Note that the coderef is evaluated in list context. If you wish to force scalar context on the return value, you must use the `scalar` keyword.

```
($stdout, $stderr, $count) = capture {
  my @list = qw/one two three/;
  return scalar @list; # $count will be 3
};
```

Also note that within the coderef, the `@_` variable will be empty. So don't use arguments from a surrounding subroutine without copying them to an array first:

```
sub wont_work {
  my ($stdout, $stderr) = capture { do_stuff( @_ ) };    # WRONG
  ...
}

sub will_work {
  my @args = @_;
  my ($stdout, $stderr) = capture { do_stuff( @args ) }; # RIGHT
  ...
}
```

Captures are normally done to an anonymous temporary filehandle. To capture via a named file (e.g. to externally monitor a long-running capture), provide custom filehandles as a trailing list of option pairs:

```
my $out_fh = IO::File->new("out.txt", "w+");
my $err_fh = IO::File->new("out.txt", "w+");
capture { ... } stdout => $out_fh, stderr => $err_fh;
```

The filehandles must be read/write and seekable. Modifying the files or filehandles during a capture operation will give unpredictable results. Existing IO layers on them may be changed by the capture.

When called in void context, `capture` saves memory and time by not reading back from the capture handles.

capture_stdout

```
($stdout, @result) = capture_stdout \&code;
$stdout = capture_stdout \&code;
```

The `capture_stdout` function works just like `capture` except only STDOUT is captured. STDERR is not captured.

capture_stderr

```
($stderr, @result) = capture_stderr \&code;
$stderr = capture_stderr \&code;
```

The `capture_stderr` function works just like `capture` except only STDERR is captured. STDOUT is not captured.

capture_merged

```
($merged, @result) = capture_merged \&code;
$merged = capture_merged \&code;
```

The `capture_merged` function works just like `capture` except STDOUT and STDERR are merged. (Technically, STDERR is redirected to the same capturing handle as STDOUT before executing the function.)

Caution: STDOUT and STDERR output in the merged result are not guaranteed to be properly ordered due to buffering.

tee

```
($stdout, $stderr, @result) = tee \&code;
$stdout = tee \&code;
```

The `tee` function works just like `capture`, except that output is captured as well as passed on to the original STDOUT and STDERR.

When called in void context, `tee` saves memory and time by not reading back from the capture handles, except when the original STDOUT OR STDERR were tied or opened to a scalar handle.

tee_stdout

```
($stdout, @result) = tee_stdout \&code;
$stdout = tee_stdout \&code;
```

The `tee_stdout` function works just like `tee` except only STDOUT is teed. STDERR is not teed (output goes to STDERR as usual).

tee_stderr

```
($stderr, @result) = tee_stderr \&code;
$stderr = tee_stderr \&code;
```

The `tee_stderr` function works just like `tee` except only STDERR is teed. STDOUT is not teed (output goes to STDOUT as usual).

tee_merged

```
($merged, @result) = tee_merged \&code;
$merged = tee_merged \&code;
```

The `tee_merged` function works just like `capture_merged` except that output is captured as well as passed on to STDOUT.

Caution: STDOUT and STDERR output in the merged result are not guaranteed to be properly ordered due to buffering.

LIMITATIONS

Portability

Portability is a goal, not a guarantee. `tee` requires fork, except on Windows where `system(1, @cmd)` is used instead. Not tested on any particularly esoteric platforms yet. See the CPAN Testers Matrix <http://matrix.cpantesters.org/?dist=Capture-Tiny> for test result by platform.

PerlIO layers

Capture::Tiny does its best to preserve PerlIO layers such as ':utf8' or ':crlf' when capturing (only for Perl 5.8.1+) . Layers should be applied to STDOUT or STDERR *before* the call to `capture` or `tee`. This may not work for tied filehandles (see below).

Modifying filehandles before capturing

Generally speaking, you should do little or no manipulation of the standard IO filehandles prior to using Capture::Tiny. In particular, closing, reopening, localizing or tying standard filehandles prior to capture may cause a variety of unexpected, undesirable and/or unreliable behaviors, as described below. Capture::Tiny does its best to compensate for these situations, but the results may not be what you desire.

Closed filehandles

Capture::Tiny will work even if STDIN, STDOUT or STDERR have been previously closed. However, since they will be reopened to capture or tee output, any code within the captured block that depends on finding them closed will, of course, not find them to be closed. If they started closed, Capture::Tiny will close them again when the capture block finishes.

Note that this reopening will happen even for STDIN or a filehandle not being captured to ensure that the filehandle used for capture is not opened to file descriptor 0, as this causes problems on various platforms.

Prior to Perl 5.12, closed STDIN combined with PERL_UNICODE=D leaks filehandles and also breaks *tee()* for undiagnosed reasons. So don't do that.

Localized filehandles

If code localizes any of Perl's standard filehandles before capturing, the capture will affect the localized filehandles and not the original ones. External system calls are not affected by localizing a filehandle in Perl and will continue to send output to the original filehandles (which will thus not be captured).

Scalar filehandles

If STDOUT or STDERR are reopened to scalar filehandles prior to the call to `capture` or `tee`, then Capture::Tiny will override the output filehandle for the duration of the `capture` or `tee` call and then, for `tee`, send captured output to the output filehandle after the capture is complete. (Requires Perl 5.8)

Capture::Tiny attempts to preserve the semantics of STDIN opened to a scalar reference, but note that external processes will not be able to read from such a handle. Capture::Tiny tries to ensure that external processes will read from the null device instead, but this is not guaranteed.

Tied output filehandles

If STDOUT or STDERR are tied prior to the call to `capture` or `tee`, then Capture::Tiny will attempt to override the tie for the duration of the `capture` or `tee` call and then send captured output to the tied filehandle after the capture is complete. (Requires Perl 5.8)

Capture::Tiny may not succeed resending UTF−8 encoded data to a tied STDOUT or STDERR filehandle. Characters may appear as bytes. If the tied filehandle is based on Tie::StdHandle, then Capture::Tiny will attempt to determine appropriate layers like `:utf8` from the underlying filehandle and do the right thing.

Tied input filehandle

Capture::Tiny attempts to preserve the semantics of tied STDIN, but this requires Perl 5.8 and is not entirely predictable. External processes will not be able to read from such a handle.

Unless having STDIN tied is crucial, it may be safest to localize STDIN when capturing:

```
  my ($out, $err) = do { local *STDIN; capture { ... } };
```

Modifying filehandles during a capture

Attempting to modify STDIN, STDOUT or STDERR *during* `capture` or `tee` is almost certainly going to cause problems. Don't do that.

Forking inside a capture

Forks aren't portable. The behavior of filehandles during a fork is even less so. If Capture::Tiny detects that a fork has occurred within a capture, it will shortcut in the child process and return empty strings for captures. Other problems may occur in the child or parent, as well. Forking in a capture block is not recommended.

Using threads

Filehandles are global. Mixing up I/O and captures in different threads without coordination is going to cause problems. Besides, threads are officially discouraged.

Dropping privileges during a capture

If you drop privileges during a capture, temporary files created to facilitate the capture may not be cleaned up afterwards.

No support for Perl 5.8.0

It's just too buggy when it comes to layers and UTF−8. Perl 5.8.1 or later is recommended.

Limited support for Perl 5.6

Perl 5.6 predates PerlIO. UTF−8 data may not be captured correctly.

ENVIRONMENT

PERL_CAPTURE_TINY_TIMEOUT

Capture::Tiny uses subprocesses internally for `tee`. By default, Capture::Tiny will timeout with an error if such subprocesses are not ready to receive data within 30 seconds (or whatever is the value of `$Capture::Tiny::TIMEOUT`). An alternate timeout may be specified by setting the `PERL_CAPTURE_TINY_TIMEOUT` environment variable. Setting it to zero will disable timeouts. **NOTE**, this does not timeout the code reference being captured — this only prevents Capture::Tiny itself from hanging your process waiting for its child processes to be ready to proceed.

SEE ALSO

This module was inspired by IO::CaptureOutput, which provides similar functionality without the ability to tee output and with more complicated code and API. IO::CaptureOutput does not handle layers or most of the unusual cases described in the ''Limitations'' section and I no longer recommend it.

There are many other CPAN modules that provide some sort of output capture, albeit with various limitations that make them appropriate only in particular circumstances. I'm probably missing some. The long list is provided to show why I felt Capture::Tiny was necessary.

- IO::Capture
- IO::Capture::Extended
- IO::CaptureOutput
- IPC::Capture
- IPC::Cmd
- IPC::Open2
- IPC::Open3
- IPC::Open3::Simple
- IPC::Open3::Utils
- IPC::Run
- IPC::Run::SafeHandles
- IPC::Run::Simple
- IPC::Run3
- IPC::System::Simple
- Tee
- IO::Tee
- File::Tee
- Filter::Handle
- Tie::STDERR
- Tie::STDOUT
- Test::Output

SUPPORT

Bugs / Feature Requests

Please report any bugs or feature requests through the issue tracker at <https://github.com/dagolden/Capture–Tiny/issues>. You will be notified automatically of any progress on your issue.

Source Code

This is open source software. The code repository is available for public review and contribution under the terms of the license.

<https://github.com/dagolden/Capture–Tiny>

```
  git clone https://github.com/dagolden/Capture-Tiny.git
```

AUTHOR

David Golden <dagolden@cpan.org>

CONTRIBUTORS

- Dagfinn Ilmari Mannsåker <ilmari@ilmari.org>
- David E. Wheeler <david@justatheory.com>
- fecundf <not.com+github@gmail.com>
- Graham Knop <haarg@haarg.org>
- Peter Rabbitson <ribasushi@cpan.org>

COPYRIGHT AND LICENSE

This software is Copyright (c) 2009 by David Golden.

This is free software, licensed under:

```
  The Apache License, Version 2.0, January 2004
```