Crate env_logger

A simple logger that can be configured via environment variables, for use with the logging facade exposed by the `log` crate.

Despite having "env" in its name, **`env_logger`** can also be configured by other means besides environment variables. See the examples in the source repository for more approaches.

By default, `env_logger` writes logs to `stderr`, but can be configured to instead write them to `stdout`.

Example

```
use log::{debug, error, log_enabled, info, Level};

env_logger::init();

debug!("this is a debug {}", "message");
error!("this is printed by default");

if log_enabled!(Level::Info) {
    let x = 3 * 4; // expensive computation
    info!("the answer was: {}", x);
}
```

Assumes the binary is `main`:

```
$ RUST_LOG=error ./main
[2017-11-09T02:12:24Z ERROR main] this is printed by default


$ RUST_LOG=info ./main
[2017-11-09T02:12:24Z ERROR main] this is printed by default
[2017-11-09T02:12:24Z INFO main] the answer was: 12


$ RUST_LOG=debug ./main
[2017-11-09T02:12:24Z DEBUG main] this is a debug message
[2017-11-09T02:12:24Z ERROR main] this is printed by default
[2017-11-09T02:12:24Z INFO main] the answer was: 12
```

You can also set the log level on a per module basis:

```
$ RUST_LOG=main=info ./main
```

```
[2017-11-09T02:12:24Z ERROR main] this is printed by default
[2017-11-09T02:12:24Z INFO main] the answer was: 12
```

And enable all logging:

```
$ RUST_LOG=main ./main
[2017-11-09T02:12:24Z DEBUG main] this is a debug message
[2017-11-09T02:12:24Z ERROR main] this is printed by default
[2017-11-09T02:12:24Z INFO main] the answer was: 12
```

If the binary name contains hyphens, you will need to replace them with underscores:

```
$ RUST_LOG=my_app ./my-app
[2017-11-09T02:12:24Z DEBUG my_app] this is a debug message
[2017-11-09T02:12:24Z ERROR my_app] this is printed by default
[2017-11-09T02:12:24Z INFO my_app] the answer was: 12
```

This is because Rust modules and crates cannot contain hyphens in their name, although `cargo` continues to accept them.

See the documentation for the `log` crate for more information about its API.

Enabling logging

By default all logging is disabled except for the `error` level

The **`RUST_LOG`** environment variable controls logging with the syntax:

```
RUST_LOG=[target][=][level][,...]
```

Or in other words, its a comma-separated list of directives. Directives can filter by **target**, by **level**, or both (using `=`).

For example,

```
RUST_LOG=data=debug,hardware=debug
```

target is typically the path of the module the message in question originated from, though it can be overridden. The path is rooted in the name of the crate it was compiled for, so if your program is in a file called, for example, `hello.rs`, the path would simply be `hello`.

Furthermore, the log can be filtered using prefix-search based on the specified log target.

For example, `RUST_LOG=example` would match the following targets:

- `example`
- `example::test`

- `example::test::module::submodule`
- `examples::and_more_examples`

When providing the crate name or a module path, explicitly specifying the log level is optional. If omitted, all logging for the item will be enabled.

level is the maximum `log::Level` to be shown and includes:

- `error`
- `warn`
- `info`
- `debug`
- `trace`
- `off` (pseudo level to disable all logging for the target)

Logging level names are case-insensitive; e.g., `debug`, `DEBUG`, and `dEbuG` all represent the same logging level. For consistency, our convention is to use the lower case names. Where our docs do use other forms, they do so in the context of specific examples, so you won't be surprised if you see similar usage in the wild.

Some examples of valid values of `RUST_LOG` are:

- `RUST_LOG=hello` turns on all logging for the `hello` module
- `RUST_LOG=trace` turns on all logging for the application, regardless of its name
- `RUST_LOG=TRACE` turns on all logging for the application, regardless of its name (same as previous)
- `RUST_LOG=info` turns on all info logging
- `RUST_LOG=INFO` turns on all info logging (same as previous)
- `RUST_LOG=hello=debug` turns on debug logging for `hello`
- `RUST_LOG=hello=DEBUG` turns on debug logging for `hello` (same as previous)
- `RUST_LOG=hello,std::option` turns on `hello`, and std's option logging
- `RUST_LOG=error,hello=warn` turn on global error logging and also warn for `hello`
- `RUST_LOG=error,hello=off` turn on global error logging, but turn off logging for `hello`
- `RUST_LOG=off` turns off all logging for the application
- `RUST_LOG=OFF` turns off all logging for the application (same as previous)

Filtering results

A `RUST_LOG` directive may include a regex filter. The syntax is to append `/` followed by a regex. Each message is checked against the regex, and is only logged if it matches. Note that the matching is done after formatting the log string but before adding any logging meta-data. There is a single filter for all modules.

Some examples:

- `hello/foo` turns on all logging for the 'hello' module where the log message includes 'foo'.
- `info/f.o` turns on all info logging where the log message includes 'foo', 'f1o', 'fao', etc.
- `hello=debug/foo*foo` turns on debug logging for 'hello' where the log message includes 'foofoo' or 'fofoo' or 'fooooooofoo', etc.

- `error,hello=warn/[0-9]scopes` turn on global error logging and also warn for hello. In both cases the log message must include a single digit number followed by 'scopes'.

Capturing logs in tests

Records logged during `cargo test` will not be captured by the test harness by default. The `Builder::is_test` method can be used in unit tests to ensure logs will be captured:

```
#[cfg(test)]
mod tests {
    use log::info;

    fn init() {
        let _ = env_logger::builder().is_test(true).try_init();
    }

    #[test]
    fn it_works() {
        init();

        info!("This record will be captured by `cargo test`");

        assert_eq!(2, 1 + 1);
    }
}
```

Enabling test capturing comes at the expense of color and other style support and may have performance implications.

Disabling colors

Colors and other styles can be configured with the `RUST_LOG_STYLE` environment variable. It accepts the following values:

- `auto` (default) will attempt to print style characters, but don't force the issue. If the console isn't available on Windows, or if TERM=dumb, for example, then don't print colors.
- `always` will always print style characters even if they aren't supported by the terminal. This includes emitting ANSI colors on Windows if the console API is unavailable.
- `never` will never print style characters.

Tweaking the default format

Parts of the default format can be excluded from the log output using the `Builder`. The following example excludes the timestamp from the log output:

```
env_logger::builder()
    .format_timestamp(None)
    .init();
```

Stability of the default format

The default format won't optimise for long-term stability, and explicitly makes no guarantees about the stability of its output across major, minor or patch version bumps during `0.x`.

If you want to capture or interpret the output of `env_logger` programmatically then you should use a custom format.

Using a custom format

Custom formats can be provided as closures to the `Builder`. These closures take a `Formatter` and `log::Record` as arguments:

```
use std::io::Write;

env_logger::builder()
    .format(|buf, record| {
        writeln!(buf, "{}: {}", record.level(), record.args())
    })
    .init();
```

See the `fmt` module for more details about custom formats.

Specifying defaults for environment variables

`env_logger` can read configuration from environment variables. If these variables aren't present, the default value to use can be tweaked with the `Env` type. The following example defaults to log `warn` and above if the `RUST_LOG` environment variable isn't set:

```
use env_logger::Env;

env_logger::Builder::from_env(Env::default().default_filter_or("warn")).init();
```

Re-exports

```
pub use super::Target;
pub use super::TimestampPrecision;
pub use super::WriteStyle;
```

Modules

filter	Filtering for log records.
fmt	Formatting for log records.

Structs

Builder	`Builder` acts as builder for initializing a `Logger`.
Env	Set of environment variables to configure from.
Logger	The env logger.

Constants

DEFAULT_FILTER_ENV	The default name for the environment variable to read filters from.
DEFAULT_WRITE_STYLE_ENV	The default name for the environment variable to read style preferences from.

Functions

builder	Create a new builder with the default environment variables.
from_env Deprecated	Create a builder from the given environment variables.
init	Initializes the global logger with an env logger.
init_from_env	Initializes the global logger with an env logger from the given environment variables.
try_init	Attempts to initialize the global logger with an env logger.
try_init_from_env	Attempts to initialize the global logger with an env logger from the given environment variables.