Failing in Rust

Armin @*mitsuhiko* Ronacher


SENTRY

we show you your crashes


“Only those who dare to fail greatly
can ever achieve greatly.”
— Robert F. Kennedy

WHY DO WE CARE?

Errors are Important

- Errors are part of your API
- Exceptions let you forget about this easily
- A lot more relevant when you can catch them and there are multiple versions of libraries involved

WAYS TO FAIL GREATLY

Mechanisms

- `Result<T, E>`
- `Option<T>`
- `panic!`

Result Propagation vs Panic

- Results/Options are for handling
- panics are for recovering at best

Examples of Panics

- out of bound access
- runtime

Examples of Option

- safe signalling absence of data
- "the one obvious error"

— Douglas Adams

But when you do

- `panic!("…");`
- `unreachable!();`

LET'S TALK RESULTS

But if you don't panic …
how do you result?

```rust
fn square_a_number() -> Result<f32, E> {
    let num = get_a_random_float()?;
    Ok(num * num)
}
```

```
let val = expr?;
```

```
let val = match Try::into_result(expr) {
    Ok(v) => v,
    Err(e) => return
        Try::from_error(From::from(e));
};
```

error propagation can be hooked!

The Err in Result can be anything :-/

So let's use some traits for Err

```rust
pub trait Error: Debug + Display {
    fn description(&self) -> &str;
    fn cause(&self) -> Option<&Error>;
}
```

```
impl Error + 'static {
    pub fn downcast_ref<T>(&self) -> Option<&T>
        where T: Error + 'static;
}
```


“To kill std::error is as good a service as, and sometimes even better than, the establishing of a new trait”
— Charles Darwin

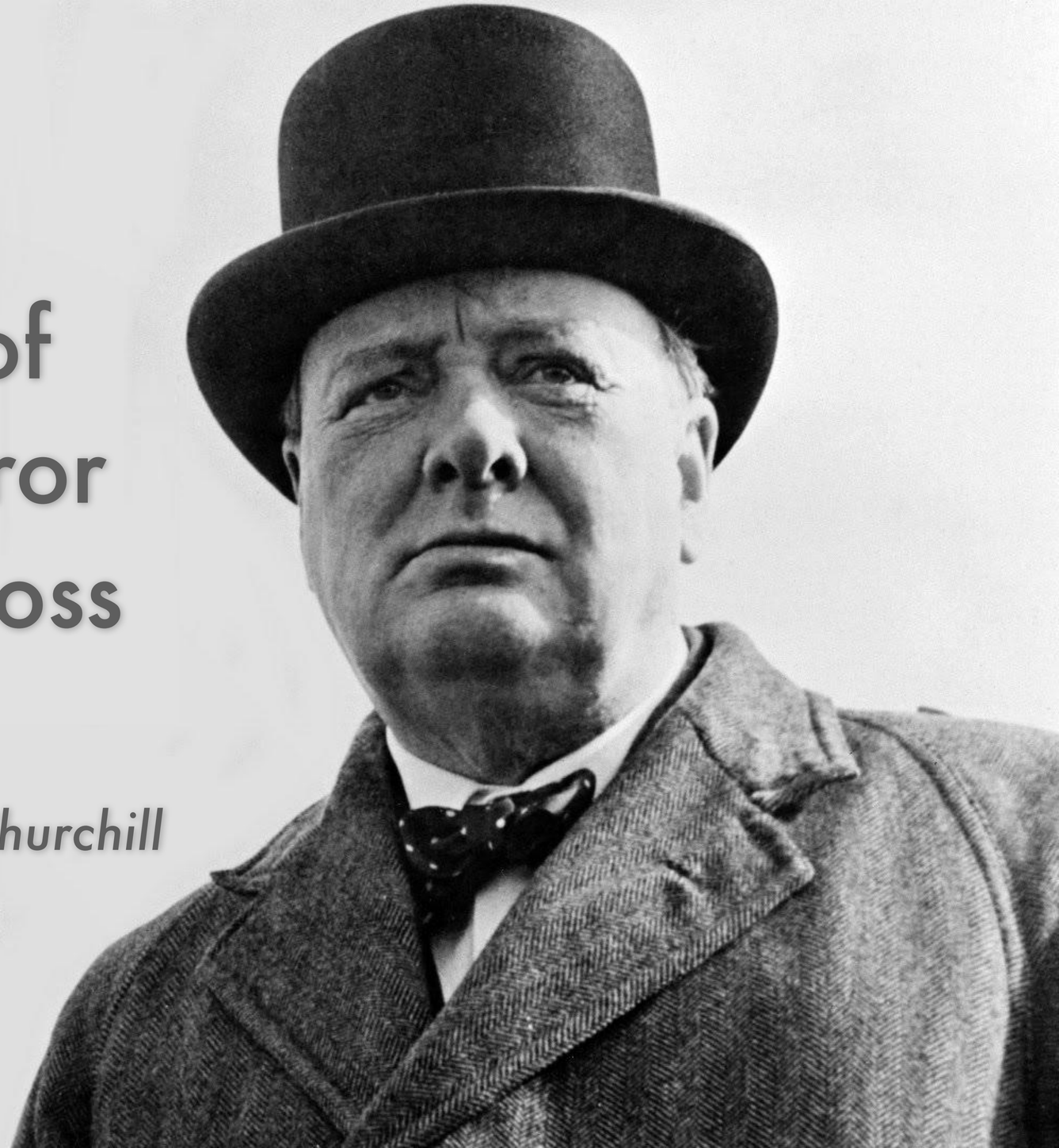
Problems

- Generic errors give no guarantees
  - no Send / Sync / Debug
- causes() returns non static errors
- description() is useless
- no backtraces

ENTER FAILURE

"Success consists of going from std::error to failure without loss of enthusiasm"

– *Winston Churchill*



nice!

some std errors are fails

```
impl<E> Fail for E
  where E: StdError + Send + Sync + 'static
```


failure 0.1 ➠ failure 1.0

```rust
pub trait Fail: Display + Debug + Send + Sync + 'static {
    fn cause(&self) -> Option<&Fail>;
    fn backtrace(&self) -> Option<&Backtrace>;
    fn context<D>(self, context: D) -> Context<D>
    where
        D: Display + Send + Sync + 'static,
        Self: Sized;
}
```

Fail can be derived

```
#[derive(Fail, Debug)]
#[fail(display = "my failure happened")]
pub struct MyFailure;
```

```
#[derive(Fail, Debug)]
#[fail(display = "my failure happened")]
pub struct MyFailure {
    backtrace: failure::Backtrace,
}
```

```
#[derive(Fail, Debug)]
#[fail(display = "my failure happened")]
pub struct MyFailure {
    backtrace: failure::Backtrace,
    #[fail(cause)] io_cause: ::std::io::Error,
}
```

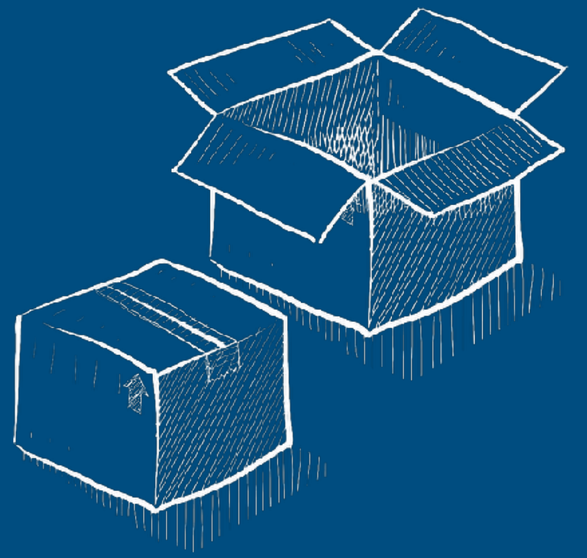
Fail & Error

Fail ⟷ Error

Fail for libraries

Error for applications

WHAT'S IN THE PACKAGE



Main Functionality

- Fail trait
- Error type
- Context

Bonus Points

- Fail works with `no_std`
- Fail works with many `std::error`s
- error-chain is deprecating itself for failure
- actix and others are already using it!


an Error is not a Fail
— rustc

Error to &Fail

failure 0.1:

```
error.cause()
```

failure 1.0:

```
error.as_fail()
```

EXAMPLES

Parse Errors

```
#[derive(Debug, Fail, PartialEq, Eq, PartialOrd, Ord)]
#[fail(display = "invalid value for project id")]
pub struct ProjectIdParseError;
```

Complex Parse Errors

```
#[derive(Debug, Fail)]
pub enum DsnParseError {
    #[fail(display = "no valid url provided")] InvalidUrl,
    #[fail(display = "no valid scheme")] InvalidScheme,
    #[fail(display = "username is empty")] NoUsername,
    #[fail(display = "no project id")] NoProjectId,
    #[fail(display = "invalid project id")]
    InvalidProjectId(#[fail(cause)] ProjectIdParseError),
}
```

Mapping Errors

```
fn parse(url: Url) -> Result<Dsn, DsnParseError> {
    let project_id: i64 = url.path()
        .trim_matches('/')
        .parse()
        .map_err(DsnParseError::InvalidProjectId)?;
    Ok(Dsn { project_id })
}
```

Error Kinds

```
#[derive(Debug, Fail, Copy, Clone, PartialEq, Eq, Hash)]
pub enum ErrorKind {
    #[fail(display = "governor spawn failed")]
    TroveGovernSpawnFailed,
    #[fail(display = "governor shutdown failed")]
    TroveGovernShutdownFailed,
}
```

Custom Errors

```
#[derive(Debug)]
pub struct Error {
    inner: Context<ErrorKind>,
}
```

Error Pass Through

```
impl Fail for Error {
    fn cause(&self) -> Option<&Fail> { self.inner.cause() }
    fn backtrace(&self) -> Option<&Backtrace> { self.inner.backtrace() }
}

impl fmt::Display for Error {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        fmt::Display::fmt(&self.inner, f)
    }
}
```

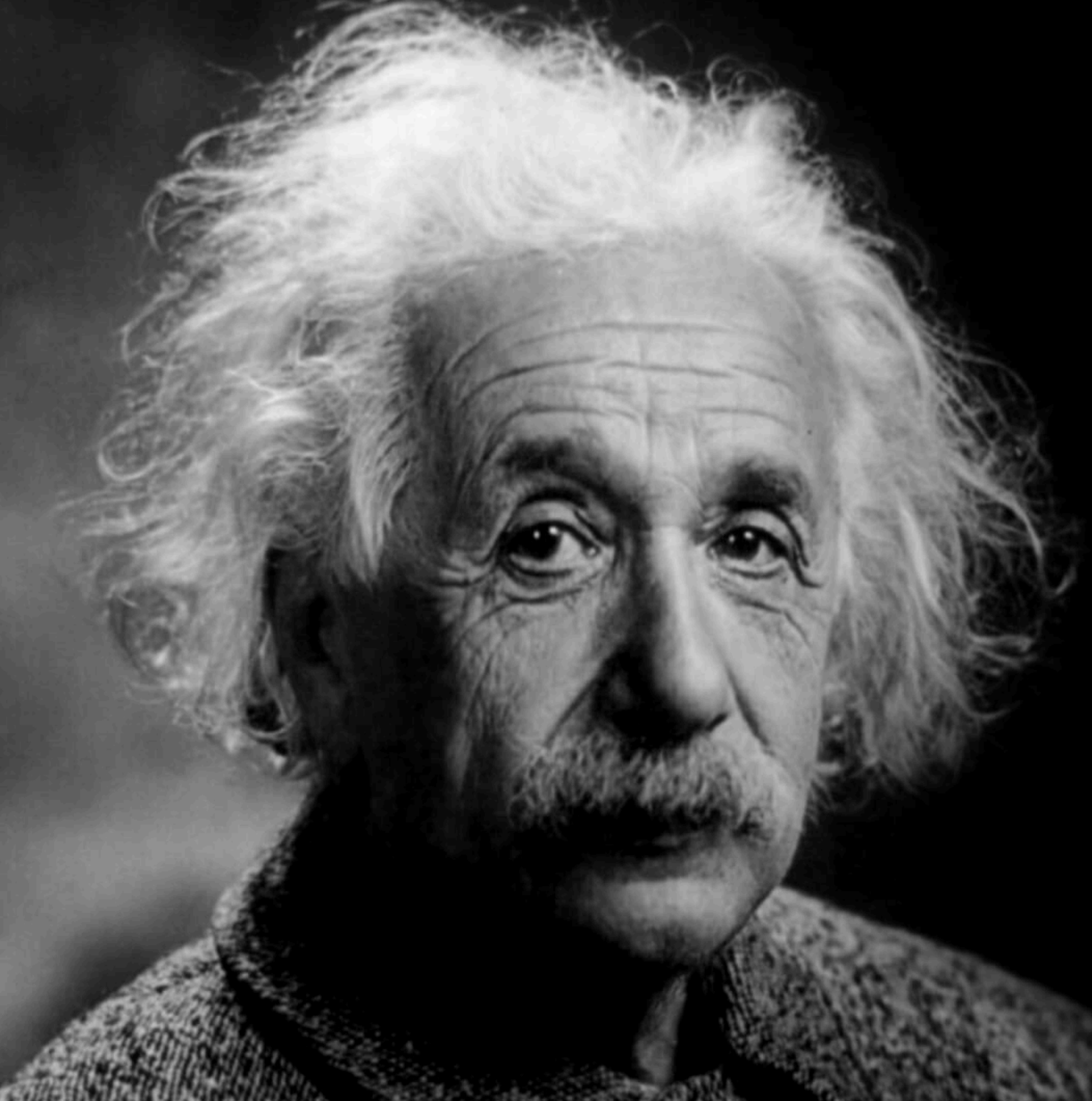
Example Usage

```
pub fn run(config: Config) -> Result<(), Error> {
    let trove = Arc::new(Trove::new(config));
    trove.govern().context(ErrorKind::TroveGovernSpawnFailed)?;
    // …
}
```

User Facing with Error

```
use failure::{Error, ResultExt};


pub fn attach_logfile(&mut self, logfile: &str)
    -> Result<(), Error>
{
    let f = fs::File::open(logfile)
        .context("Could not open logfile")?;
    let reader = BufReader::new(f);
    for line in reader.lines() {
        let line = line?;
```



A person who never made a mistake never had to write an error API

Albert Einstein