

Systemnahe Programmierung in Rust

- Fehlerbehandlung (Blog Eintrag von Andrew Gallant) -

Hubert Högl

Hochschule Augsburg / Informatik
<https://hhoegl.informatik.hs-augsburg.de/hhwiki/FrontPage>

2022-12-05 23:46:48

aus [1]

Result type idiom: `std::result::Result` ↔ `io::Result<T>`

- Unwrap/expect “erlaubt” bei
 - ein paar Zeilen “Wegwerf-Code”
 - Hinweis auf Bug im Programm (Verletzung von Invarianten)
 - anfänglichem Konzentrieren auf den Programmablauf und nicht auf die Fehlerbehandlung
- Combinators
- Early returns
- ? Operator
- Own error types

Combinators: Option + Result

```
aus [1]

fn double_arg(mut argv: env::Args) -> Result<i32, String> {
    argv.nth(1)
        .ok_or("Please give at least one argument".to_owned())
        .and_then(|arg| arg.parse::<i32>().map_err(|err| err.to_string()))
        .map(|i| i * 2)
}
```

Limits of Combinators

aus [1]

```
fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, String> {
    File::open(file_path)
        .map_err(|err| err.to_string())
        .and_then(|mut file| {
            let mut contents = String::new();
            file.read_to_string(&mut contents)
                .map_err(|err| err.to_string())
                .map(|_| contents)
        })
        .and_then(|contents| {
            contents.trim().parse::<i32>()
                .map_err(|err| err.to_string())
        })
        .map(|n| 2 * n)
}
```

aus [1]

- explicit case analysis

? Operator + map_err()

- try! macro

Own error types

- Nachteile von String als Error Typ:
 - lossy (opaque)
 - Fehlertyp kann schlecht aus String abgeleitet werden
- Deshalb eigene Fehlertypen definieren, vor allem in Bibliotheken
- Beispiel: `io::Error` (enthält `io::ErrorKind`)

- `#[derive(Debug)]`

```
enum CliError {  
    Io(io::Error),  
    Parse(num::ParseIntError),  
}
```

```
fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, CliError> {  
    ...  
    ...map_err(CliError::Io)?  
    ...map_err(CliError::Parse)?  
}
```

Traits for error handling

`std::error::Error` (is implemented by all error types)

```
fn source(&self) -> Option<&(dyn Error + 'static)> { ... };  
fn description(&self) -> &str;  
fn cause(&self) -> Option<&dyn Error> { ... };  
fn provide(&'a self, demand: &mut Demand<'a>) { ... };
```

Trait objects: `Box<&dyn Error>`, `&dyn Error`

```
impl error::Error for CliError { ... }
```

`std::convert::From`

? Operator ruft `From::from` auf. Jeder Typ kann in `Box<&dyn Error>` gewandelt werden.

```
→ Result<i32, Box<&dyn Error>>
```

Nachteil: `Box<&dyn Error>` ist opaque. Man kann jedoch `description` und `cause` aufrufen (Anmerkung: Über *runtime reflection* könnte man den Typ herausbekommen).

Composing custom error types

- Vermeidet den Nachteil mit `Box<&dyn Error>`
- `-> Result<i32, CliError>`
- muss aber wieder `.map_err(CliError::Io)?;` schreiben
- Lösung:

```
impl From<io::Error> for CliError { ... }  
impl From<num::ParseIntError> for CliError { ... }
```

- Custom errors
 - Exposing repr <https://doc.rust-lang.org/std/io/enum.ErrorKind.html>
 - Hidden repr <https://doc.rust-lang.org/std/io/enum.ErrorKind.html>
 - Add information beyond String repr
- Minimum requirement: Implement Error trait (always has `fmt::Debug` and `fmt::Display`). Erlaubt zumindest einfache *error composition*.
- Implement `From` on the error types. This allows composition of detailed errors.
- Define result type alias (examples: `io::Result`, `fmt::Result`)

Case study: A program to read population data (`city-pop`)

<https://github.com/BurntSushi/rust-error-handling-case-study>

Summary

- short example code: `unwrap()`
- quick 'n' dirty without panicking: `Box<&dyn error>` or `anyhow crate, anyhow::Error`
- Own error types with `From` and `Error` impls to make ? more ergonomic
- Libraries
 - own error types, implement `std::error::Error`
 - when appropriate impl `From`
(coherence rule: callers can not impl `From` on libraries error type)
- Learn combinators defined on `Option` and `Result`. Try to find a healthy mix of ? operator and combinators like `and_then`, `map`, `unwrap_or`.

- [1] Andrew Gallant, Error Handling in Rust, 2015 - 2020
<https://blog.burntsushi.net/rust-error-handling>