

# Systemnahe Programmierung in Rust

## - Fehlerbehandlung -

Hubert Högl

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

2022-12-12 13:49:30

- Behebbarer Fehler → `Option/Result`
  - File not found
  - `parse()` error
  - ...
- Nicht behebbarer Fehler → `panic!`
  - Index out of bounds
  - Divide by zero
  - Out of memory
  - Stack overflow
  - ...

“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

# Kombinatoren und Umwandler

## Kombinatoren

Option: `map()`, `unwrap_or()`, `unwrap_or_else()`, `and_then()`, `as_ref()`, `as_mut()`

Result: `map()`, `map_err()`, `unwrap_or()`, `unwrap_or_else()`, `and_then()`,  
`or()`, `or_else()`, `as_ref()`, `as_mut()`

## Umwandler

`ok_or()` : Option in Result

`ok` : Result in Option

Die Doku der Methoden finden Sie unter

- <https://doc.rust-lang.org/stable/std/option/enum.Option.html>
- <https://doc.rust-lang.org/stable/std/result/enum.Result.html>

## Kombinatoren und Umwandler (2)

```
Option<T> ---> map(f)
                / Some(x) ---> Some(f(x))
                -> Option<U>
                \ None ---> None
```

```
Option<T> ---> unwrap_or(d)
                / Some(o) ---> o
                \ None ---> d
```

## Kombinatoren und Umwandler (3)

```
Option<T> ---> unwrap_or_else(f)
                / Some(o) ---> o
                \ f: FnOnce() -> T
```

```
Option<T> ---> and_then(f)
                / Some(o) ---> f: FnOnce(o) -> Option<U>
                \ None ---> None
```

## Kombinatoren und Umwandler (4)

```
Option<T> ---> as_ref() ---> Option<&T>
```

```
Option<T> ---> as_mut() ---> Option<&mut T>
```

## Kombinatoren und Umwandler (5)

```
Result<T, E> ---> unwrap_or(d)
                    / Ok(o) ---> o
                    \ Err(e) ---> d
```

```
Result<T, E> ---> unwrap_or_else(f)
                    / Ok(o) ---> o
                    \ Err(e) ---> f: FnOnce(E) -> T
```

```
Result<T, E> ---> and_then(f)
                    / Ok(o) ---> f: FnOnce(o) -> Result<U, E>
                    \ Err(e)
```



## Kombinatoren und Umwandler (6)

```
Result<T, E> ---> or_else(f)
                    / Ok(o)
                    \ Err(e) ---> f: FnOnce(e) -> Result<T, F>
```

```
Result<T, E> ---> map(f)
                  / Ok(o) ---> f: FnOnce(o) -> U
                  -> Result<U, E>
                  \ Err(e)
```

```
Result<T, E> ---> map_err(f)
                  / Ok(o)
                  -> Result<T, F>
                  \ Err(e) ---> f: FnOnce(e) -> F
```

```
Result<T, E> ---> as_ref() ---> Result<&T, &E>
```

```
Result<T, E> ---> as_mut() ---> Result<&mut T, &mut E>
```

## Kombinatoren und Umwandler (7)

### *Umwandler*

```
Result<T> ---> ok()          / Ok(x) ---> Some(x)
              -> Option<T>
              \ error discarded ---> None
```

```
Option<T> ---> ok_or(e)      / Some(x) ---> Ok(x)
              -> Result<T, E>
              \ None ---> Err(e)
```

## Alias für Ergebnistyp

Beispiel: `std::io::Result`

```
pub type Result<T> = Result<T, Error>;
```

<https://doc.rust-lang.org/std/io/type.Result.html>

## Frühes Zurückkehren (early return)

aus [8]

```
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let username_file_result = File::open("hallo.txt");

    let mut username_file = match username_file_result {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut username = String::new();

    match username_file.read_to_string(&mut username) {
        Ok(_) => Ok(username),
        Err(e) => Err(e),
    }
}
```

## ? Operator

```
use std::fs::File;
use std::io;
use std::io::Read;

fn read_username_from_file() -> Result<String, io::Error> {
    let mut username_file = File::open("hallo.txt"?);
    let mut username = String::new();
    username_file.read_to_string(&mut username )?;
    Ok(username)
}
```

- Durch ? werden die Fehler automatisch falls nötig mit `from()` (From Trait) in den Fehlertyp der umgebenden Funktion umgewandelt.
- Für eigene Fehlertypen muss der From Trait implementiert werden.

## std::error::Error

```
fn input_num() -> Result<u32, Box<dyn std::error::Error>> {  
  
    println!("Zahl eingeben:");  
    let mut input = String::new();  
  
    // -> Result<usize>, bei Fehler std::io::Error  
    std::io::stdin().read_line(&mut input)?;  
  
    Ok(input.trim().parse()?) // bei Fehler ParseIntError  
}
```

- `std::error::Error` wird von jedem Trait implementiert → passt auf jeden Fehler!
- Aber: Konkreter Fehlertyp geht nach oben verloren (*type erasure*). Man kann ihn aber im Aufrufer wieder herausbekommen mit:

```
...  
if let Some(x) = e.downcast_ref::<std::io::Error>() {  
    println!("{}", x);  
}
```

## Anwendungsspezifische Fehler

aus [6]

Beste Lösung zur *error composition*

```
#[derive(Debug)]
enum UpstreamError{
    IO(io::Error),
    Parsing(net::AddrParseError),
}
```

- Mit `map_err()` Fehler umwandeln

```
fn main() -> Result<(), UpstreamError> {
    let _f = File::open("invisible.txt")
        .map_err(UpstreamError::IO)?; // (*)

    let _localhost = "::1"
        .parse::<Ipv6Addr>()
        .map_err(UpstreamError::Parsing)?; // (*)
}
```

`Ok(())`

## Anwendungsspezifische Fehler (2)

(\*): `UpstreamError::IO` hat eine Initialisierungsfunktion, die hier als Argument verwendet wird.



## Anwendungsspezifische Fehler (3)

Implementiere Display Trait für UpstreamError

Implementiere std::error::Error Trait

Nun map\_err() entfernen:

```
impl From<io::Error> for UpstreamError { ... }
impl From<net::AddrParseError> for UpstreamError { ... }

fn main() -> Result<(), UpstreamError> {
    let _f = File::open("invisible.txt");
    let _localhost = "::1".parse::<Ipv6Addr>()?;

    Ok(())
}
```

Tipp: Als anwendungsspezifischen Fehler nicht den String Typ verwenden (zu allgemein).

## main() verlassen

aus [7]

- Return Typen für main() müssen den Termination Trait implementieren (Funktion report() gibt ExitCode zurück)
- <https://doc.rust-lang.org/std/process/trait.Termination.html>
- std::process::Termination

```
fn main() -> Result<(), &'static str> {  
    let s = vec!["apple", "mango", "banana"];  
    let _fourth = s.get(4).ok_or("I got only 3 fruits");  
    Ok(())  
}
```

- Ausgabe

Error: "I got only 3 fruits"

- Quick and Dirty: `unwrap()` für kurze Beispiele
- Quick and Dirty ohne `panic!`: `Box<&dyn error>` oder `anyhow crate` (<https://crates.io/crates/anyhow>)
- Gut: Eigene Fehlertypen mit `From` und `Error` Implementierungen, vor allem für Bibliotheken.
- Lerne/übe Kombinatoren und Umwandler

- [1] Andrew Gallant, Error Handling in Rust, 2015 - 2020  
<https://blog.burntsushi.net/rust-error-handling>
- [2] Stephan Baumgartner, Error handling in Rust, 2021 <https://fettblog.eu/rust-error-handling>
- [3] Sheshbabu Chinnakonda, Beginner's guide to Error Handling in Rust, 2020  
<https://www.sheshbabu.com/posts/rust-error-handling>
- [4] Programming Rust, 2. Auflage 2021, Kapitel 7: Error Handling  
<https://learning.oreilly.com/library/view/programming-rust-2nd/9781492052586/ch07.html>
- [5] Jon Gjengset, Rust for Rustaceans, Kapitel 4: Error Handling (für Fortgeschrittene!)  
<https://learning.oreilly.com/library/view/rust-for-rustaceans/9781098129828/f01.xhtml>
- [6] McNamara, Rust in Action, 2021. Kapitel 8: Networking (darin wird auch die Fehlerbehandlung erklärt) <https://hhoegl.informatik.hs-augsburg.de/sysprog/Rust/RiA-Kap8.pdf>, S. 28-41
- [7] Rahul Sharma u.a., The Complete Rust Programming Reference, Kap. 6 (Error Handling)  
<https://learning.oreilly.com/library/view/the-complete-rust/9781838828103>
- [8] Die Programmiersprache Rust, Kapitel 9: Fehlerbehandlung  
<https://rust-lang-de.github.io/rustbook-de/ch09-00-error-handling.html>