Tratamento de erros

Diogo Silveira Mendonça

Slides Baseados no Capítulo 9 do Rust Book
https://doc.rust-lang.org/book/ch09-00-error-handling.html

Introdução

- Rust possui diversos recursos para tratamento de erros.

- Em muitos casos, o Rust informa sobre um possível erro que deve ser corrigido, antes que o código possa ser compilado. Tornando o código mais robusto.

- No Rust existem duas categorias de erros:

    - Erros recuperáveis: Onde relata ao usuário um problema e vai tentar novamente depois. Exemplo: um erro de arquivo não encontrado

    - Erros irrecuperáveis: São sintomas de bugs, onde o rust encerra o programa imediatamente. Exemplo: tentar acessar um índice além do final de um array.

Introdução

- Maior parte das linguagens não fazem essa distinção dos erros dessa forma. Normalmente eles utilizam exceções.

- Rust não possui exceções, em vez disso, ele tem o tipo Result<T, E> para recuperação de erros e o macro panic! para erros irrecuperáveis.

- Também vamos ver algumas considerações que são levadas ao decidir se devemos tentar recuperar de um erro ou interromper a execução.

- Os exemplos de código podem ser encontrados nesse link.

Erros irrecuperáveis com panic!

- Existem duas formas de fazer o Rust entrar em pânico.
  - Ocasionar o erro, como acessar um índice inválido de um array.
  - Chamar explicitamente o macro panic!.

- Por padrão quando o Rust entra em pânico, será imprimido a mensagem de erro, vai resolver e limpar a pilha e terminará.

- Esse processo de resolver e limpar a pilha é bem trabalhoso. Então o Rust permite que você escolha a alternativa de abortar imediatamente, sem realizar essas ações.

- Nesse caso a limpeza da memória será feita pelo sistema operacional.

Erros irrecuperáveis com panic!

- Podemos chamar panic! no programa:

```
fn main() {
    panic!("crash and burn");
}
```

- Quando rodarmos o programa, teremos o seguinte erro:

```
$ cargo run
   Compiling panic v0.1.0 (file:///projects/panic)
    Finished dev [unoptimized + debuginfo] target(s) in 0.25s
     Running `target/debug/panic`
thread 'main' panicked at 'crash and burn', src/main.rs:2:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

- O Rust irá informar onde entrou em pânico, informando o arquivo, a linha e a coluna.

Usando um backtrace do panic

- Outro exemplo de código que gera pânico.

```
fn main() {
    let v = vec![1, 2, 3];

    v[99];
}
```

- Nessa situação, o programa entra em pânico, porque tentamos acessar um índice inválido da estrutura.

- Em C, existe a possibilidade de ocorrer um buffer overread, que é um problema de segurança, que ocorre quando você obtém o dado que corresponde a aquela localização da memória, que não pertence a estrutura.

Usando um backtrace do panic

- O Rust interrompe a execução, impedindo um buffer overread (vulnerabilidade).

```
fn main() {
        let v = vec![1, 2, 3];

        v[99];
}
```

- Executando o código, teremos a mensagem de erro:

```
$ cargo run
   Compiling panic v0.1.0 (file:///projects/panic)
       Finished dev [unoptimized + debuginfo] target(s) in 0.27s
       Running `target/debug/panic`
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 99',
src/main.rs:4:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

- A mensagem indica que tentamos acessar um índice inválido

Usando um backtrace do panic

- Acessando o Backtrace com “RUST_BACKTRACE=1 cargo run”

```
$ RUST_BACKTRACE=1 cargo run
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 99', src/main.rs:4:5
stack backtrace:
   0: rust_begin_unwind
             at /rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/std/src/panicking.rs:584:5
   1: core::panicking::panic_fmt
             at /rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/core/src/panicking.rs:142:14
   2: core::panicking::panic_bounds_check
             at /rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/core/src/panicking.rs:84:5
   3: <usize as core::slice::index::SliceIndex<[T]>>::index
             at /rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/core/src/slice/index.rs:242:10
   4: core::slice::index::<impl core::ops::index::Index<I> for [T]>::index
             at /rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/core/src/slice/index.rs:18:9
   5: <alloc::vec::Vec<T,A> as core::ops::index::Index<I>>::index
             at /rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/alloc/src/vec/mod.rs:2591:9
   6: panic::main
             at ./src/main.rs:4:5
   7: core::ops::function::FnOnce::call_once
             at /rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/core/src/ops/function.rs:248:5
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
```

Usando um backtrace do panic

- O backtrace pode ser diferente, dependendo do sistema operacional e da versão do Rust.

- Por padrão os símbolos de depuração são habilitados, quando usados **cargo run** e **cargo build** sem a flag do sistema operacional **--release**.

- A linha 6 do backtrace, aponta onde foi ocasionado o problema.

```
6: panic::main
             at ./src/main.rs:4:5
```

- Mais para frente, veremos como utilizar o panic!, para lidar com condições de erros.

Erros recuperáveis com Result

- Em alguns casos não há a necessidade de encerrar um programa por conta de um erro.

- Por exemplo, quando o programa não encontra o arquivo, em vez de encerrar o programa, ele pode criar o arquivo.

- O enum result é definido por:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

- T e E são tipos genéricos, T representa o tipo que será retornado em caso de sucesso e E representa o tipo que será retornado em caso de falha.

Erros recuperáveis com Result

- Veremos o result, implementando o exemplo onde tentamos abrir um arquivo.

```
use std::fs::File;

fn main() {
    let greeting_file_result = File::open("hello.txt");
}
```

- O tipo de retorno de File::open é um Result<T, E>.
  - O tipo de T usado em caso é sucesso é std::fs::File, um identificador de arquivo.
  - O tipo de E usado em caso de erro é std::io::Error, contendo informações do erro.

- greeting_file_result em caso de sucesso terá uma instância do identificador de arquivo, em caso de erro terá uma instância com as informações do erro.

Erros recuperáveis com Result

- O enum result, assim como enum option, não precisa especificar “Result::” antes de Ok e de Err, depois do match.
- Depois de match podemos usar o identificador do arquivo para leitura ou escrita.
- Em caso de erro, o programa chamará o macro panic!, encerrando o programa.

Exemplo:

```
use std::fs::File;

fn main() {
    let greeting_file_result =
        File::open("hello.txt");

    let greeting_file =
        match greeting_file_result {
            Ok(file) => file,
            Err(error) => panic!
                ("Problem opening
            the file: {:?}", error),
        };
}
```

Usando match em diferentes erros

- io::Error é uma estrutura com o método kind que pode ser chamado com io::ErrorKind.
- O enum io::ErrorKind contém as variantes com os diferentes tipos de erro.
- Nesse caso utilizamos um match error.kind() procurando ErrorKind::NotFound, que representa que o arquivo não foi encontrado.
- Temos “match File::create” já que a criação do arquivo também pode falhar.

Exemplo:

```
  ---snip---
let greeting_file = match greeting_file_result {
        Ok(file) => file,
        Err(error) => match error.kind() {
                ErrorKind::NotFound =>
                        match File::create("hello.txt") {
                                Ok(fc) => fc,
                                Err(e) => panic!
                                        ("Problem creating
                                        the file: {:?}", e),
                        },
                other_error => {
                        panic!("Problem opening the file:
                                {:?}", other_error);
                }
        },
};
}
```

Atalhos em panic em erros: unwrap e expect

- Utilizar match pode deixar o código muito verboso e dificultar sua leitura.
- O tipo Result<T, E>, possui diversos métodos para auxiliar sua utilização.
- O método de atalho unwrap é implementado como a expressão match. se result tiver o valor ok, unwrap retorna o valor dentro de ok. Se o valor for err, unwrap chama o macro panic!.

```
use std::fs::File;

fn main() {
    let greeting_file = File::open("hello.txt").unwrap();
}
```

- Se rodarmos sem o arquivo, teremos a mensagem:

thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value: Os { code: 2, kind: NotFound, message: "No such file or directory" }', src/main.rs:4:49

Atalhos em panic em erros: unwrap e expect

- O método de atalho expect, funciona de forma similar, porém permite personalizar a mensagem de erro.

```
use std::fs::File;

fn main() {
    let greeting_file = File::open("hello.txt")
    .expect("hello.txt should be included in this project");
}
```

- Então teremos a mensagem de erro

```
thread 'main' panicked at 'hello.txt should be included in this project: Os {
code: 2, kind: NotFound, message: "No such file or directory" }',
src/main.rs:5:10
```

- Uma mensagem de erro com expect, pode dar mais contexto e ser mais útil que a mensagem padrão do unwrap, dando mais qualidade ao software.

Propagando erros

- Quando se tem uma função que chama algo que possa falhar e em vez dela tratar o erro, ela retorna o erro para o código. Isso é conhecido como propagação de erro.
- O código chamador pode ter mais informação útil ou instruções para resolver o erro.
- O exemplo tenta pegar o nome de usuário dentro do arquivo, caso o arquivo não exista ou não possa ser lido ele retornará os erros.

Exemplo:

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

fn read_username_from_file() -> Result<String, io::Error> {
    let username_file_result = File::open("hello.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),
    }
}
```

Propagando erros

- O tipo de retorno dessa função é Result<String, io::Error>, ou seja, está retornando Result<T, E>.

- Caso a função File::open retorne Err, o programa não será encerrado, em vez disso, a função retornará o erro.

- O método read_to_string, não precisa explicitar o retorno do resultado, já que é a última expressão na função.

Exemplo:

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

fn read_username_from_file() -> Result<String, io::Error> {
    let username_file_result = File::open("hello.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),
    }
}
```

Atalho para propagar erros: operando “?”

- Temos o mesmo exemplo anterior, porém de forma simplificada, utilizando “?”

Exemplo:

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

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

- Quando uma “?” é colocado após result, é tratado praticamente da mesma forma que match.

- Se Result for Ok, a expressão irá retornar o valor dentro de Ok e continuar a execução do código.

- Se Result for Err, toda a função retornará Err, como se tivesse um return.

Atalho para propagar erros: operando “?”

- A diferença entre match e “?” , é que “?” utiliza a função from, da biblioteca padrão do Rust.
- Essa função é utilizada para converter um tipo em outro.
- Ou seja, quando “?” chama from, o tipo de erro recebido é convertido no tipo definido na função.
- Isso pode ser útil quando temos uma função que retorna um tipo de erro para representar todas as maneiras que ela pode falhar, mesmo que por razões muito diferentes.
- Por exemplo, poderíamos modificar a função read_username_from_file para retornar um tipo personalizado OurError que definimos. E instanciamos OurError com “impl From<io::Error> for OurError”. Então a função read_username_from_file passaria a retornar OurError nos casos de falha.

Atalho para propagar erros: operando “?”

- A utilização de “?” elimina a redundância no código e deixa o código mais simples.

- Poderíamos simplificar ainda mais o código, encadeando a chamadas de métodos após “?”.

Exemplo:

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

fn read_username_from_file() -> Result<String, io::Error> {
    let mut username = String::new();

    File::open("hello.txt")?.read_to_string(&mut username)?;

    Ok(username)
```

- Movemos a definição de username para o início. Encadeamos read_to_string com File::open. Em caso de sucesso, tanto File::open quanto read_to_string vão retornar Ok.

Atalho para propagar erros: operando “?”

- Podemos simplificar ainda mais o código, utilizando fs::read_to_string

  Exemplo:

```
use std::fs;
use std::io;

fn read_username_from_file() -> Result<String, io::Error> {
    fs::read_to_string("hello.txt")
}
```

- A biblioteca padrão fornece a função fs::read_to_string, que abre um arquivo, cria uma string, coloca seu conteúdo na string e retorna a string.

- Em caso de falha, ela irá retornar o erro.

Onde o operando “?” pode ser utilizado?

- O operando “?” só pode ser utilizado em funções onde o tipo retornado é compatível com o valor de “?”.

- O código ao abaixo apresenta um problema, já que a função main tem o retorno (), que é incompatível.

- Podemos resolver isso, mudando o retorno da função, ou adotando o uso de match ou de outro método de Result<T, E> para tratar o Result<T, E> de forma apropriada.

Exemplo:

```
use std::fs::File;

fn main() {
    let greeting_file = File::open("hello.txt")?;
}
```

Onde o operando “?” pode ser utilizado?

```
$ cargo run
   Compiling error-handling v0.1.0 (file:///projects/error-handling)
error[E0277]: the `?` operator can only be used in a function that returns `Result`
or `Option` (or another type that implements `FromResidual`)
 --> src/main.rs:4:48
  |
3 | fn main() {
  | --------- this function should return `Result` or `Option` to accept `?`
4 |    let greeting_file = File::open("hello.txt")?;
  |                                               ^ cannot use the `?` operator in a function that
returns `()`
  |
  = help: the trait `FromResidual<Result<Infallible, std::io::Error>>` is not
implemented for `()`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `error-handling` due to previous error
```

Onde o operando “?” pode ser utilizado?

- A mensagem de erro mencionou que também podemos utilizar “?” com valores de Option<T>.

- “?” só pode ser utilizado em uma Option, caso a função retorne um Option.

  Exemplo:

```
fn last_char_of_first_line(text: &str) -> Option<char> {
    text.lines().next()?.chars().last()
}
```

- Comportamento semelhante quando se utiliza com Result<T,E>

- Se retornar None, será retornado precocemente da função nesse ponto.

- Se retornar Some, O valor resultante será o valor dentro de Some e continuará a função.

Chamar panic! ou não chamar panic!

- Se chamarmos panic!, o programa será encerrado. Fazendo com que erros recuperável se tornem irreversíveis.

- Se utilizarmos Result, o programa poderá tentar se recuperar do erro.

- Então por padrão é mais interessante utilizar Result, a não ser que seja um caso onde não seja interessante ou possível de recuperar o erro.

- Em uma situações como exemplos, código de protótipo e testes é mais interessante utilizar panic!.

Exemplos, código de protótipo e testes

- Em exemplos, não é tão interessante colocar a recuperação de erros, por conta que eles podem deixar o código mais robusto e complicado.

- Normalmente, em caso de exemplos, utilização métodos que podem entrar em pânico, como unwrap, é entendido como um espaço que deveria ter a recuperação de código.

- Em protótipos, também se entende que unwrap e expect pode ser marcações onde deveriam ter a recuperação de código, que o desenvolvedor irá trabalhar futuramente para deixar o código mais robusto.

- Em Testes, se uma chamada de um método falhar, você vai querer que todo o teste falhe, mesmo que não seja um teste para aquele método. Então utilizar o macro panic! seria o ideal.

Casos onde você tem mais informação que o compilador

- Em alguns casos é correto chamar unwrap e expect, quando se tem alguma lógica que garanta que o valor de Result seja Ok, mas a lógica não é algo que o compilador irá entender.

- Apesar disso ainda devemos utilizar expect, para tratar o Err, mesmo que nunca caia nesse caso. O expect pode ter uma explicação do porque não deve cair no caso de falha.

Casos onde você tem mais informação que o compilador

- No exemplo abaixo, temos um IpAddr hardcoded, que sempre será válido.

- Entretanto, no parse ainda obtemos um Result e o compilador vai exigir que cuidemos do caso de falha.

```
fn main() {
    use std::net::IpAddr;

    let home: IpAddr = "127.0.0.1"
    .parse()
    .expect("Hardcoded IP address should be valid");
}
```

Diretrizes para tratamento de erros

- É aconselhável que você utilize panic!, quando existir a possibilidade do seu código ficar em um estado ruim.

- Temos um estado ruim quando alguma suposição, garantia, contrato ou invariante for quebrado, como valores inválidos, contraditórios ou ausentes.
  - Um estado ruim é inesperado, não é algo previsível como a possibilidade do usuário digitar o formato do input de forma incorreta.

  - Se a partir de um ponto, seu código precisa acreditar que não está em um estado ruim, em vez de verificar por problemas a cada passo.

  - Quando não se tem uma maneira de codificar os tipos que você utiliza.

Diretrizes para tratamento de erros

- Se alguém chama o código e passa valores que não fazem sentido, é melhor retornar um erro.

- Isso possibilita o usuário da biblioteca decidir o que fazer em caso de erro.

Referências

Slides Baseados no Capítulo 9 do Rust Book
https://doc.rust-lang.org/book/ch09-00-error-handling.html