PV281: Programování v Rustu

Obsah

Enumy

Základní varianta

```
enum Delivery {
    Pickup,
    Parcel,
}

fn main() {
    let delivery = Delivery::Pickup;
}
```

C varianta

```rust
// zacinaji nulou
enum Number {
    Zero,
    One,
    Two,
}

// enum s nastavenou hodnotou
enum Color {
    Red = 0xff0000,
    Green = 0x00ff00,
    Blue = 0x0000ff,
}

fn main() {
    // enum muze byt pretypovan na cele cislo
    println!("zero is {}", Number::Zero as i32);
    println!("one is {}", Number::One as i32);

    println!("roses are #{:06x}", Color::Red as i32);
    println!("violets are #{:06x}", Color::Blue as i32);
}
```

Monadicka varianta

```
enum Delivery {
    Pickup,
    Parcel(String),
}


fn main() {
    let delivery = Delivery::Parcel(String::from("Ceska 0, 60200 Brno"));
}
```

Zpracování hodnoty pomocí if

```
fn main() {
    let some_u8_value = Some(0u8);
    if let Some(3) = some_u8_value {
        println!("three");
    }
}
```

Pattern Matching

Výhody pattern matchingu

1. Kontrola všech variant větvení
   To pomáhá i při refaktoringu - nikdy nezpomenete změnit další místa v kódu
2. Lepší čitelnost

Zpracování enum pattern matchingem

```
fn deliver(delivery: Delivery) {
    match delivery {
        Delivery::Pickup => {
            println!("Vyzvednete to u nas v obchode!");
        },
        Delivery::Parcel(address) => {
            println!("Zbozi bude dorucena prepravou na adresu: {}!", address);
        },
        _ => {
                println!("Zatim neimplementovany zpusob doruceni!");
        }
    }
}
```

Porovnání s jinou proměnnou

```
fn main() {
    let x = Some(5);
    let y = 10;

    match x {
        Some(50) => println!("Got 50"),
        Some(y) => println!("Matched, y = {:?}", y),
        _ => println!("Default case, x = {:?}", x),
    }

    println!("at the end: x = {:?}, y = {:?}", x, y);
}
```

S využitím tuple nebo struktury

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}

fn main() {
    let msg = Message::ChangeColor(0, 160, 255);

    match msg {
        Message::Quit => {
            println!("The Quit variant has no data to destructure.")
        }
        Message::Move { x, y } => {
            println!(
                "Move in the x direction {} and in the y direction {}",
                x, y
            );
        }
        Message::Write(text) => println!("Text message: {}", text),
        Message::ChangeColor(r, g, b) => println!(
            "Change the color to red {}, green {}, and blue {}",
            r, g, b
        ),
    }
```

Match více explicitně zadaných variant

```
fn main() {
    let x = 1;

    match x {
        1 | 2 => println!("one or two"),
        3 => println!("three"),
        _ => println!("anything"),
    }
}
```

Match nad range

```
fn main() {
    let x = 'c';

    match x {
        'a'..='j' => println!("early ASCII letter"), // ..= znamená včetně
        'k'..='z' => println!("late ASCII letter"),
        _ => println!("something else"),
    }
}
```

Nad touple

```
fn main() {
    let numbers = (2, 4, 8, 16, 32);

    match numbers {
        (first, .., last) => {
            println!("Some numbers: {}, {}", first, last);
        }
    }
}
```

Struktury

Definice struktury

```
struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}

fn main() {
    // ...
}
```

Použití struktury

```
struct User {
    //...
}

fn main() {
    let mut user1 = User {
        email: String::from("someone@example.com"),
        username: String::from("someusername123"),
        active: true,
        sign_in_count: 1,
    };

    user1.email = String::from("anotheremail@example.com");
}
```

Funkce pro vytvoření struktury

```
fn build_user(email: String, username: String) -> User {
    User {
        email,
        username,
        active: true,
        sign_in_count: 1,
    }
}
```

Vytvoření z již existující struktury

```
fn main() {
    let user1 = User {
        email: String::from("someone@example.com"),
        username: String::from("someusername123"),
        active: true,
        sign_in_count: 1,
    };

    let user2 = User {
        email: String::from("another@example.com"),
        username: String::from("anotherusername567"),
        ...user1
    };
}
```

Makro Debug

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "The area of the rectangle is {} square pixels.",
        area(&rect1)
    );

    println!("rect1 is {:?}", rect1);
}

fn area(rectangle: &Rectangle) -> u32 {
```

Traity

Zjednodušeně můžeme trait považovat za interface v jiných programovacích jazycích.

Definujeme pomocí nich společnou funkcionalitu.

Implementace traitu

```
pub trait Summary {
    fn summarize(&self) -> String;
}

pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}, by {} ({})", self.headline,
        self.author, self.location)
    }
```

Implementace traitu pro druhou sturkturu

```
pub trait Summary {
    fn summarize(&self) -> String;
}

pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
```

Výchozí implementace

```
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more...)")
    }
}
```

Využití jiných metod ve výchozí implementaci

```
pub trait Summary {
    fn summarize_author(&self) -> String;

    fn summarize(&self) -> String {
        format!("(Read more from {}...)", self.summarize_author())
    }
}
```

Trait jako parametr

```
pub fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}
```

Trait Bound

Impl je syntaktické pozlátko pro delší zápis, kterému se říká trait bound. Následující výpis kódu je shodný s přechozím. Jen je delší a hůře čitelný. Proto doporučujeme používat impl.

```
pub fn notify<T: Summary>(item: &T) {
    println!("Breaking news! {}", item.summarize());
}
```

Více traitů

```
pub fn notify(item: &(impl Summary + Display)) {
}
```

Zápis pomocí where

Použijeme pro situace, kde máme více parametrů s různými kombinacemi traitů

```
fn some_function<T, U>(t: &T, u: &U) -> i32
    where T: Display + Clone,
          U: Clone + Debug
{
}
```

Ošetření chybových stavů

Typy chyb

1. chyby ze kterých se můžeme zotavit
2. a chyby po kterých to můžeme zabalit

Panika

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

Pozn. v Rustu při panice program sám projde stacka uklidí po sobě veškerá data. Je to za cenu větší binárky. Pokud chcete snížit velikost binárky a nechat úklid na operačním systému, tak můžete udělat následující konfiguraci v Cargo.toml:

```toml
[profile.release]
panic = 'abort'
```

Výčet Result

Funkce, kde může nastat chyba, vrací result. Z něm můžeme vyčíst výsledek nebo chybu

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

Příklad ošetření chybového stavu

Pokud chceme otevřít soubor, tak může dojít k chybě. Ošetřit ji můžeme následně:

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

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

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

Ošetření dílčích chyb

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

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

    let f = match f {
        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)
            }
        },
    };
```

Zpanikaření v případě chyby

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

fn main() {
    // Pri chybe ukonci program s generickou chybovou zpravou
    let f = File::open("hello.txt").unwrap();

    // Pri chybe ukonci program s nasi vlastni chybovou hlaskou
    let f = File::open("hello.txt").expect("hello.txt se nepovedlo otevřít");
}
```

Propagace chyb

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

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

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

    Ok(s)
}
```

Práce s CLI

Argumenty příkazové řádky

```
let pattern = std::env::args().nth(1).expect("no pattern given");
let path = std::env::args().nth(2).expect("no path given");
```

Uložení argumentů do struktury

```
struct Cli {
    pattern: String,
    path: std::path::PathBuf,
}

fn main() {
    let pattern = std::env::args().nth(1).expect("no pattern given");
    let path = std::env::args().nth(2).expect("no path given");

    let args = Cli {
        pattern: pattern,
        path: std::path::PathBuf::from(path),
    };
}
```

Jednodušší zpracování přes clap

```rust
use clap::Clap;

/// This doc string acts as a help message when the user runs '--help'
/// as do all doc strings on fields
#[derive(Clap)]
#[clap(version = "1.0", author = "L.G.")]
struct Opts {
    /// Sets a custom config file. Could have been an Option<T> with no default too
    #[clap(short = "c", long = "config", default_value = "default.conf")]
    config: String,
    /// Some input. Because this isn't an Option<T> it's required to be used
    input: String,
    /// A level of verbosity, and can be used multiple times
    #[clap(short = "v", long = "verbose", parse(from_occurrences))]
    verbose: i32,
}
```

Jednodušší zpracování přes clap

```
fn main() {
    let opts: Opts = Opts::parse();

    // Gets a value for config if supplied by user, or defaults to "default.conf"
    println!("Value for config: {}", opts.config);
    println!("Using input file: {}", opts.input);

    // Vary the output based on how many times the user used the "verbose" flag
    // (i.e. 'myprog -v -v -v' or 'myprog -vvv' vs 'myprog -v'
    match opts.verbose {
        0 => println!("No verbose info"),
        1 => println!("Some verbose info"),
        2 => println!("Tons of verbose info"),
        3 | _ => println!("Don't be crazy"),
    }
}
```

Zpracování přes builder pattern

```
use clap::App;

fn main() {
    let matches = App::new("myapp")
        .version("1.0")
        .author("LG")
        .about("Does awesome things")
        .arg("-c, --config=[FILE] 'Sets a custom config file'")
        .arg("<INPUT>              'Sets the input file to use'")
        .arg("-v...                'Sets the level of verbosity'")
        .get_matches();

    if let Some(i) = matches.value_of("INPUT") {
        println!("Value for input: {}", i);
    }

    if let Some(c) = matches.value_of("config") {
        println!("Value for config: {}", c);
    }

    match matches.occurrences_of("v") {
        0 => println!("Verbose mode is off"),
        1 => println!("Verbose mode is kind of on"),
        2 => println!("Verbose mode is on"),
        _ => println!("Don't be crazy"),
```

Definice argumentů v yaml

```yaml
name: myapp
version: "1.0"
author: LG
about: Does awesome things
args:
    - config:
        short: c
        long: config
        value_name: FILE
        about: Sets a custom config file
        takes_value: true
```

```rust
use clap::{App, load_yaml};

fn main() {
    let yaml = load_yaml!("cli.yaml");
    let matches = App::from(yaml).get_matches();
```

Stdin

```
use std::io;

let mut input = String::new();
match io::stdin().read_line(&mut input) {
    Ok(n) => {
        println!("{} bytes read", n);
        println!("{}", input);
    }
    Err(error) => println!("error: {}", error),
}
```

Stdin

```
#![feature(stdin_forwarders)]
use std::io;

let lines = io::stdin().lines();
for line in lines {
    println!("got a line: {}", line.unwrap());
}
```

Práce se soubory

Vytvoření souboru a zápis

```
use std::fs::File;
use std::io::prelude::*;

fn main() -> std::io::Result<()> {
    let mut file = File::create("foo.txt")?;
    file.write_all(b"Hello, world!")?;
    Ok(())
}
```

Načtení obsahu ze souboru

```
use std::fs::File;
use std::io::prelude::*;

fn main() -> std::io::Result<()> {
    let mut file = File::open("foo.txt")?;
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;
    assert_eq!(contents, "Hello, world!");
    Ok(())
}
```

Práce přes buffer

```rust
use std::fs::File;
use std::io::BufReader;
use std::io::prelude::*;

fn main() -> std::io::Result<()> {
    let file = File::open("foo.txt")?;
    let mut buf_reader = BufReader::new(file);
    let mut contents = String::new();
    buf_reader.read_to_string(&mut contents)?;
    assert_eq!(contents, "Hello, world!");
    Ok(())
}
```

Načtení řádky

```rust
use std::fs::File;
use std::io::{self, prelude::*, BufReader};

fn main() -> io::Result<()> {
    let file = File::open("foo.txt")?;
    let reader = BufReader::new(file);

    for line in reader.lines() {
        println!("{}", line?);
    }

    Ok(())
}
```

Synchronizace na disk

```
use std::fs::File;
use std::io::prelude::*;

fn main() -> std::io::Result<()> {
    let mut f = File::create("foo.txt")?;
    f.write_all(b"Hello, world!")?;

    f.sync_all()?;
    Ok(())
}
```

Flush bufferu

```
use std::io::prelude::*;
use std::io::BufWriter;
use std::fs::File;

fn main() -> std::io::Result<()> {
    let mut buffer = BufWriter::new(File::create("foo.txt")?);

    buffer.write_all(b"some bytes")?;
    buffer.flush()?;
    Ok(())
}
```

Zápis přes buffer

```
use std::fs::File;
use std::io::{BufWriter, Write};

fn main() {
    let data = "Some data!";
    let f = File::create("/tmp/foo").expect("Unable to create file");
    let mut f = BufWriter::new(f);
    f.write_all(data.as_bytes()).expect("Unable to write data");
}
```

Dotazy?

Děkuji za pozornost