Rust Enums

1. Basics

what?

Enums (enumerations) represent a type that can be one of several variants

- Each variant can carry different data letting you capture different states in a single type
- Provide type safety through pattern matching (handling all possible variants)

key concept: enums provides a way to express `sum types` (multiple possible values, each possibly with data)

Syntax

```
enum Message {
    Quit,                          // Unit variant -- zero sized type
    Move { x: i32, y: i32 },      // Struct variant -- equivalent to a struct
    Write(String),                 // Tuple variant -- wrapped around a string
    ChangeColor(i32, i32, i32),   // Tuple variant with multiple values -- tuple
with 3 integers
}
```

---

2. Enum Variants

Types of Variants

1. **Unit Variants**
   stateless - hold no data beyond their identity
   small memory footprint

```
   enum Direction {
       North,
       South,
       East,
       West,
   }
```

```
enum ConnectionStatus {
    Connected,
    Disconnected,
}
```

---

2. **Tuple Variants**

```
    enum Temperature {
        Celsius(f64),
        Fahrenheit(f64),
        Kelvin(f64),
    }
// in this case, each variant is carrying a single f64 value
```

---

3. **Struct Variants**

```
    enum Shape {
        Circle { radius: f64, center: (f64, f64) },
        Rectangle { width: f64, height: f64 },
    }
```

---

3. Working with Enums

Pattern Matching

```
fn process_message(msg: Message) {
    match msg {
        Message::Quit => println!("Quitting"),
        Message::Move { x, y } => println!("Moving to ({}, {})", x, y),
        Message::Write(text) => println!("Text message: {}", text),
        Message::ChangeColor(r, g, b) => println!("Color changed to RGB({},{},
{})", r, g, b),
    }
}
```

---

if let

well, just want to handle one pattern

```
let some_value = Some(3);
if let Some(value) = some_value {
    println!("Found value: {}", value);
}
```

---

4. Advanced Enum Features

Generic Enums

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

enum Option<T> {
    Some(T),
    None,
}
```

- handle `value-or-none` or `success-or-error` without null or exceptions.

Methods on Enums

```
enum Status {
    Active(String), // holding a string value
    Inactive, // no value
}

impl Status {
    // is status active?
    fn is_active(&self) -> bool {
        matches!(self, Status::Active(_))
    }

    // return name if active, none if inactive
    fn get_name(&self) -> Option<&String> {
        match self {
            Status::Active(name) => Some(name),
            Status::Inactive => None,
        }
    }
}
```

---

5. Best Practices

The Option Enum

Option

```
fn divide(numerator: f64, denominator: f64) -> Option<f64> {
    if denominator == 0.0 {
        None
    } else {
        Some(numerator / denominator)
    }
}
```

avoiding null reference errors and force the caller to handle the possible absence of a value

---

The Result Enum

- core error handling pattern

```
fn parse_port(s: &str) -> Result<u16, std::num::ParseIntError> {
    s.parse()
}
```

- caller must handle either `Ok(u16)` or `Err(ParseIntError)`
- can also use `?` to propagate errors neatly

Combining Different Enum Patterns

```
enum NetworkEvent {
    Connection {
        id: u32,
        status: Status,
    },
    Message {
        content: String,
        priority: Option<u8>,
    },
    Error(NetworkError),
}

enum NetworkError {
    Timeout(u32),
    InvalidProtocol(String),
    ConnectionLost,
}
```

---

6. Memory and performance considerations

Null Pointer Optimization

- example, certain enums like `Option<&T>`, optimized to the size of a single pointer
- a plus? you don't pay extra emory overhead for using `Option` over raw pointers

```
// Takes same space as *const T
// in that it occupies the same space as a single pointer
let x: Option<&T> = None;
```

Memory Layout

```
enum Packet {
    Small(u32),
    Large(String),
}
// Size of Packet = size of largest variant + discriminant
```

7. enum implementation samples

State Machine Implementation

```
enum State {
    Start,
    Processing { progress: f32 },
    Done(String),
    Error(String),
}

impl State {
    fn next(self, input: &str) -> Self {
        match self {
            State::Start => State::Processing { progress: 0.0 },
            State::Processing { progress } if progress >= 1.0 => {
                State::Done(input.to_string())
            }
            State::Processing { progress } => {
                State::Processing { progress: progress + 0.1 }
            }
            State::Done(_) | State::Error(_) => self,
        }
    }
}

fn main() {
    let mut current_state = State::Start;
    println!("Initial: {:?}", current_state);

    for _ in 0..12 {
        current_state = current_state.next("Finished!");
        println!("Current: {:?}", current_state);
    }
}
```

Command Pattern

```
enum Command {
    Save { filename: String },
    Load { filename: String },
    Undo,
    Redo,
    Copy { text: String },
    Paste { position: usize },
}

impl Command {
    fn execute(&self) -> Result<(), String> {
        match self {
            Command::Save { filename } => {
                println!("Saving to {}", filename);
```

```
                Ok(())
            }
            Command::Load { filename } => {
                println!("Loading from {}", filename);
                Ok(())
            }
            Command::Undo => {
                println!("Undoing last action");
                Ok(())
            }
            Command::Redo => {
                println!("Redoing last undone action");
                Ok(())
            }
            Command::Copy { text } => {
                println!("Copying text: {}", text);
                Ok(())
            }
            Command::Paste { position } => {
                println!("Pasting at position {}", position);
                Ok(())
            }
        }
    }
}

fn main() {
    let commands = vec![
        Command::Save { filename: String::from("file1.txt") },
        Command::Undo,
        Command::Copy { text: String::from("some text") },
        Command::Paste { position: 42 },
        Command::Redo,
        Command::Load { filename: String::from("file2.txt") },
    ];

    for cmd in commands {
        if let Err(e) = cmd.execute() {
            eprintln!("Command failed: {}", e);
        }
    }
}
```

---

8. Common Pitfalls and solutions maybe...

Avoiding Pattern Match Exhaustion

```
// Bad: Missing variants
fn process_status(status: Status) {
    match status {
        Status::Active(_) => println!("Active"),
        // Missing Inactive case - won't compile
    }
}
```

```rust
// Good: Using wildcard pattern
fn process_status(status: Status) {
    match status {
        Status::Active(name) => println!("Active: {}", name),
        _ => println!("Inactive"),
    }
}
```

Dealing with Large Enum Variants

If one variant is significantly larger than others, consider `boxing` that data to keep the enum size smaller. leading to more efficient memory usage when you store many enum instances.

- btw, see boxing as like putting your stuff in a storage unit instead of keeping it all at home

```rust
// Better memory efficiency for large data
enum Message {
    Text(Box<String>),  // Heap-allocated
    Binary(Box<Vec<u8>>),
}
```

- Option<fn()> (function pointers)
- Non-null raw pointers (NonNull)

---

github - nyakiomaina

X - @nyakiomaina11

ig - @nyakio.codes