RUST 열거형과 패턴 매칭

김상원

https://youtu.be/g7lJsDoOIyY

한성대학교 HANSUNG UNIVERSITY

CryptoCraft LAB

열거형

match 흐름 제어 연산자

if let을 사용한 간결한 흐름 제어

Q & A

열거형의 특징

- 하나의 타입이 가질 수 있는 값들을 열거 함으로써 타입을 정의
- 명시적인 값 집합 정의
- 타입 안전성 보장
- 패턴 매칭의 용이성
- 자기 문서화
- 확장성

열거형의 특징

- 명시적인 값 집합 정의
  - 열거형을 사용하면 하나의 변수가 취할 수 있는 가능한 모든 값을 명확하게 정의할 수 있다. 이는 코드의 가독성을 높이고, 의도하지 않은 값의 할당을 방지하여 오류를 줄일 수 있다.

열거형의 특징

- 타입 안전성 보장
  - 열거형을 사용하면 컴파일 시간에 타입 체크가 가능하여, 잘못된 타입의 값이 할당되는 것을 방지할 수 있다. 이는 프로그램의 안정성을 크게 향상시킨다.

열거형의 특징

- 패턴 매칭의 용이성
  - 많은 프로그래밍 언어에서 열거형은 패턴 매칭과 결합하여 사용된다. 이를 통해 열거형의 값에 따라 다른 동작을 쉽게 정의할 수 있으며, 코드의 가독성과 유지 보수성이 향상된다.

열거형의 특징

- 자기 문서화
  - 열거형의 각 값은 코드 내에서 특정 상태나 옵션을 명확하게 설명한다. 이는 코드를 읽는 사람이 프로그램의 의도를 더 쉽게 이해할 수 있도록 돕는다.

열거형의 특징

- 확장성
  - 일부 프로그래밍 언어에서는 열거형이 데이터를 가질 수 있다. 이는 열거형이 단순한 상수 집합을 넘어서 복잡한 데이터 구조를 표현할 수 있게 해주며, 프로그램의 확장성과 유연성을 높인다.

Enum의 기본 구조

- 간단한 열거형은 ‘enum’ 키워드, 그 뒤에 이름, 중괄호로 묶인 변형 집합을 사용하여 정의됨

```
enum Direction {
    Up,
    Down,
    Left,
    Right,
}
```

데이터 열거형

- Rust 열거형은 데이터를 저장할 수도 있기 때문에 다른 많은 프로그래밍 언어의 열거형보다 더 다양함
- 열거형의 각 변형은 서로 다른 유형과 관련 데이터의 양을 가질 수 있음

데이터 열거형

```
struct Message {
    message_type: String,
    text: Option<String>,
    x: Option<i32>,
    y: Option<i32>,
    color: Option<(i32, i32, i32)>,
}
```

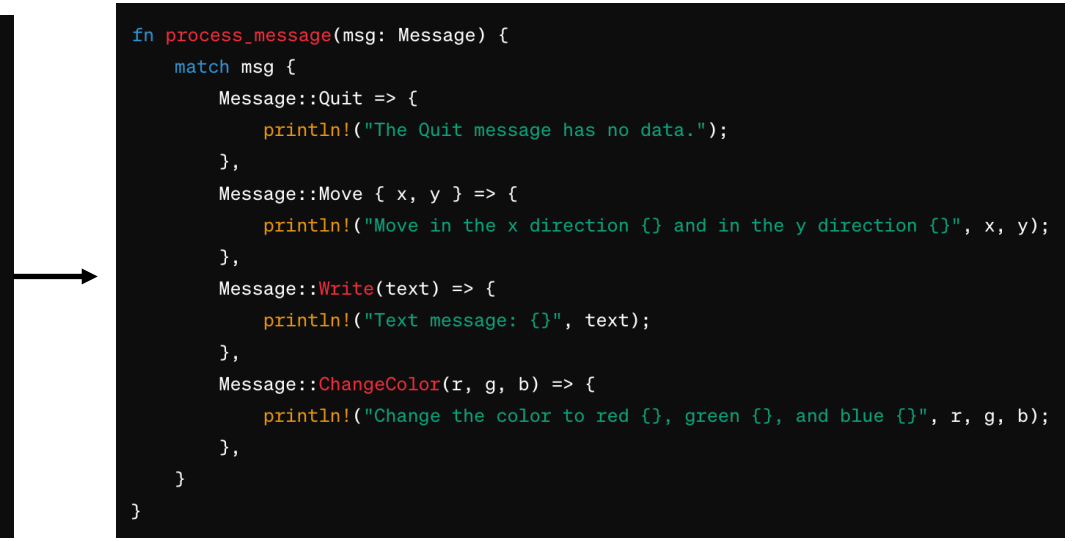


```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

- Message 열거형에서 Quit 변형은 데이터를 전달하지 않고, Move는 두 개의 필드가 있는 명명된 구조체를 포함하고, Write는 단일 String을 포함하고, ChangeColor는 세 개의 i32 값을 포함한다.

match 흐름제어 연산자

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



```rust
fn process_message(msg: Message) {
    match msg {
        Message::Quit => {
            println!("The Quit message has no data.");
        },
        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);
        },
    }
}
```

- Rust 열거형의 가장 강력한 기능 중 하나는 'match' 문을 통한 Rust 의 패턴 일치와의 통합이다. 이를 통해 열거형 값의 변형에 따라 다양한 코드를 실행할 수 있다.

match 흐름제어 연산자

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

```rust
fn process_message(msg: Message) {
    match msg {
        Message::Quit => {
            println!("The Quit message has no data.");
        },
        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);
        },
    }
}
```

- ‘if let’ 접근 방식은 열거형 변형의 수가 증가함에 따라 더 반복적이고 읽기 어려운 코드로 이어질 수 있음
- ‘match’와 달리 ‘if let’은 완전성 검사를 시행하지 않아, 열거형의 변형을 처리하지 못할 경우 컴파일러가 경고할 수 없음

If let을 사용한 간결한 흐름 제어

```rust
fn process_message(msg: Message) {
    match msg {
        Message::Quit => {
            println!("The Quit message has no data.");
        },
        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);
        },
    }
}
```

```rust
fn process_message(msg: Message) {
    if let Message::Write(text) = msg {
        println!("Text message: {}", text);
    }
    // No need to explicitly handle other cases
}
```

- ‘message’ 열거형에서 ‘Write’ 변형만 처리할 경우 if let을 사용하는 게 match를 사용하는 것보다 보다 간결함
- 특정 변형에 초점을 맞춰 코드를 단순화 하고 철저한 처리가 필요하지 않을 때 유용함

Q & A