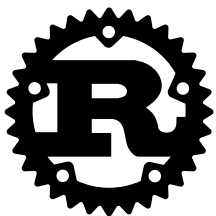






Rust 짱어먹기

Let's get Rusty



Ferris	Meaning
	This code does not compile!
	This code panics!
	This code block contains unsafe code.
	This code does not produce the desired behavior.

목차

1. 워밍업

- Rust의 강점

2. Rust의 매력

- match
- option
- result

3. emacs 의 rust 코드 구경하기

만들어진 계기



Mozilla developer *Graydon Hoare*



Rust의 강점

1. 안전한 메모리 관리
2. Ownership
3. 불변성
4. Shadowing
5. 예외와 에러 관리 (match, option, result)
6. 눈물 없이는 볼 수 없는 **감동적인 컴파일 에러 메시지**
 - 아래 예제에서 느낄 수 있습니다.

```
error[E0308]: mismatched types
  --> src\client\tenor_client.rs:39:17
39 | |                               cod: 500,
   | |                               ^^^ help: a field with a similar name exists: `code`
```

안전한 메모리 관리

```
{  
    let minsu_bank_account = BankAccount { // 민수의 계좌는 여기서부터 유효합니다  
        name: String::from("minsu"), // name: String,  
        balance: 5000, // balance: u32,  
        has_credit_card: false, // has_credit_card: bool  
        account_history: vec![4000, -6000, 3000] // Vec<i32>  
    };  
  
    // 민수의 계좌로 뭔가 합니다  
}  
// 이 스코프는 끝났고, 민수의 계좌는 더 이상  
// 유효하지 않습니다
```

1. 런타임에 운영체제로부터 메모리가 요청되어야 한다.
2. String의 사용이 끝났을 때 운영체제에게 메모리를 반납할 방법이 필요하다.

`String::from`, `Vec<i32>` 은 힙에 생성됩니다.

러스트는 `}` 괄호가 닫힐때 자동적으로 `drop` 을 호출합니다.

Ownership

- 힙에 생성되는 변수를 다른 변수에 할당할 경우
- Ownership은 복사되지 않고 (Not Copy)
- 이동(Move) 된다.
- 즉, Rust는 각각의 값은 해당 값의 Owner라고 불리는 변수를 딱 하나 가지고 있다.
- 할당할 때마다 함수에 넘겨주거나, 반환받는건 너무 불편하다!
 - *Reference/Borrow*

Ownership 예제 코드

```
pub fn fail_move_ownership() {
    let i_am_on_stack: i64 = 7427466391;
    let me_too = i_am_on_stack;

    println!("i_am_on_stack is {}", i_am_on_stack);
    println!("me_too is {}", me_too);

    let i_am_on_heap = vec![500, 60000];
    print_function(&i_am_on_heap);

    let me_too = i_am_on_heap;
    print_function(&me_too);
}

fn print_function(params: &Vec<i32>) {
    println!("{:?}", &params);
}

// 오류 발생 예제
11 |     let me_too = i_am_on_heap;
    |               ----- value moved here
12 |     print_function(&me_too);
13 |     print_function(&i_am_on_heap)
    |                   ^^^^^^^^^^^^^ value borrowed here after move

help: consider cloning the value if the performance cost is acceptable
11 |     let me_too = i_am_on_heap.clone();
    |                               ++++++++
```

불변성

1. 기본 변수는 immutable (불변)
 - 가변은 *가끔* 값을 나중에 변경하면 찾기가 어려움

변수의 불변성

- 기본 변수는 immutable (불변)

```
fn main() {  
    let x = 5; // let mut x = 5;  
    println!("The value of x is: {}", x);  
    x = 6;  
    println!("The value of x is: {}", x);  
}
```

error[E0384]: re-assignment of immutable variable `x`

--> src/main.rs:4:5

```
2 |         let x = 5;  
    - first assignment to `x`  
3 |         println!("The value of x is: {}", x);  
4 |         x = 6;  
    ^^^^^^ re-assignment of immutable variable
```

Shadowing

1. 이전에 선언한 변수와 같은 이름의 *새 변수*를 선언 할 수 있음

```
let mut vec = Vec::new();  
vec.push(1i);  
vec.push(2i);  
let vec = vec;
```

참조자(References)

1. `&` 기호가 참조자를 의미
2. **소유권**(Ownership)을 넘기지 않고 참조 가능
3. 즉, 소유권이 없음 (스코프 밖에서 메모리 반납되지 않음)
4. 댕글링 참조자(Dangling References)가 되지 않도록 보장

```
let s1 = String::from("hello");

let len = calculate_length(&s1);

fn calculate_length(s: &String) -> usize { // s는 String의 참조자입니다
    s.len()
} // 여기서 s는 스코프 밖으로 벗어났습니다. 하지만 가리키고 있는 값에 대한 소유권이 없기
// 때문에, 아무런 일도 발생하지 않습니다. (메모리가 반납되지 않음)
```

빌림(Borrowing)

1. 함수의 파라미터로 참조자를 만드는 것을 **빌림**이라고 함
2. 빌리고 용무가 끝나면 돌려주어야함
3. 빌린 값은 고칠 수 없음 (가변 참조자는 가능)

```
fn calculate_length(s: &String) -> usize {  
    s.len()  
}
```

빌림(Borrowing)

- 변수가 불변인 것 처럼 참조자도 불변

```
fn main() {  
    let s = String::from("hello");  
  
    change(&s); // 가변 참조자: change(&mut s);  
}  
  
fn change(some_string: &String) { // 가변 참조자: fn change(some_string: &mut String)  
    some_string.push_str(", world");  
}
```

error: cannot borrow immutable borrowed content `*some_string` as mutable

--> error.rs:8:5

```
8 | | some_string.push_str(", world");  
  | | ^^^^^^^^^^^^^^^
```

가변 참조자(Mutable References, data race 방지)

1. 어떤 변수에 대해 실제 사용되는 읽기 전용 참조는 여러 개 존재할 수 있다.
2. 어떤 변수에 대해 실제 사용되는 변경 가능 참조는 단 한 개만 존재할 수 있다.
3. 어떤 변수에 대해 실제 사용되는 변경 가능 참조와, 실제 사용되는 읽기 전용 참조는 동시에 존재할 수 없다.

가변 참조자 예시 코드

```
pub fn mutable_references() {  
    let mut vector: Vec<i32> = Vec::new();  
  
    let vector1 = &mut vector;  
    let vector2 = &vector;  
  
    vector1.push(500);  
    println!("{:?}", vector2);  
}
```

```
error[E0502]: cannot borrow `vector` as immutable because it is also borrowed as mutable
```

```
4 | let vector1 = &mut vector;
   | ----- mutable borrow occurs here
5 | let vector2 = &vector;
   | ^^^^^^^^ immutable borrow occurs here
6 |
7 | vector1.push(500);
   | ----- mutable borrow later used here
```

자동 추론 (타입 추론)

- let 키워드를 사용하여 타입 자동 추론

```
fn variable_examples() {  
    let unsigned_int : u32 = 123_u32;  
  
    let a : u64 = 123;  
  
    let pi : f64 = 3.14159265358979323846264338327950288;  
  
    let small_pi : f32 = 3.14;  
  
    let url : &str = "https://httpbin.org/ip";  
  
    let tenor_key : String = env::var(key: "TENOR_API_KEY")  
        .unwrap_or_else(|_| String::from(s: "<default_api_key>"));  
  
    let signed_int : i32 = 0xff_ff_ff_ff; // ???  
}
```


match

- switch와 비슷하지만, match는 모든 케이스를 표현해야함 (안전성)
- `if`, `else if`, `else` 를 가독성 있게 변경

match (tuple)

```
fn serve_coffee(coffee: bool, ice: bool, water: bool) -> &str {  
    match (coffee, ice, water) {  
        (false, _, _) => "차가운 물만 드세요",  
        (_, false, _) => "따뜻한 아메리카노를 드세요",  
        (_, _, false) => "아이스 에스프레소를 드세요",  
        _ => "아이스 아메리카노를 드세요",  
    }  
}  
  
fn main() {  
    let coffee = true;  
    let ice = false;  
    let water = true;  
  
    let drink = serve_coffee(coffee, ice, water);  
    println!("Serve: {}", drink); // Serve: 따뜻한 아메리카노를 드세요  
}
```

match (enum)

```
#[allow(dead_code)]
pub enum Weather {
    Rain,
    Snowing,
    Foggy,
    Earthquake,
    Typhoon
}

impl Weather {
    pub fn weather_forecasting_stone(&self) -> &str {
        match self {
            Self::Rain => "Stone is wet",
            Self::Snowing => "White on top",
            Self::Foggy => "Can't see stone",
            Self::Earthquake => "Stone jumping up & down",
            Self::Typhoon => "Stone gone"
        }
    }
}

fn main() {
    println!("{:?}", weather_forecasting_stone(&Weather::Earthquake))
}
```

Option

1. `null` 이거나 `null` 이 아닐 때 사용

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

Option 예제 코드

```
fn find_element_index(arr: &[i32], target: i32) -> Option<usize> {
    for (index, &element) in arr.iter().enumerate() {
        if element == target {
            return Some(index);
        }
    }
    None
}

fn main() {
    let numbers = [1, 2, 3, 4, 5];
    let target = 3;

    let index = find_element_index(&numbers, target);
    match index {
        Some(i) => println!("target: `{target}`의 위치는: {i}"),
        None => println!("원하는 숫자를 찾지 못했습니다!"),
    }
    // target: `3`의 위치는: 2
}
```

result (bool)

- 실패할 가능성이 있는 값을 반환하는 Generic enum
- 파일을 읽고 실패(없는 경우)할 경우 파일을 생성하는 경우

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

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

error[E0308]: mismatched types

--> src/main.rs:4:18

```
4 |         let f: u32 = File::open("hello.txt");  
    |                   ^^^^^^^^^^^^^^^^^^^^^^^^^ expected u32, found enum
```

`std::result::Result`

= note: expected type `u32`

= note: found type `std::result::Result<std::fs::File, std::io::Error>`

result (파일 읽기)

- 오류 수정 후

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

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

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

result

- emacs 의 rust 코드

```
fn op_system_memory_info(
    state: &mut OpState,
) -> Result<Option<MemInfo>, AnyError> {
    super::check_unstable(state, "Deno.systemMemoryInfo");
    state.borrow_mut::<Permissions>().env.check_all()?;
    match sys_info::mem_info() {
        Ok(info) => Ok(Some(MemInfo {
            total: info.total,
            free: info.free,
            available: info.avail,
            buffers: info.buffers,
            cached: info.cached,
            swap_total: info.swap_total,
            swap_free: info.swap_free,
        })),
        Err(_) => Ok(None),
    }
}
```


느낀점

1. Fast and Safe

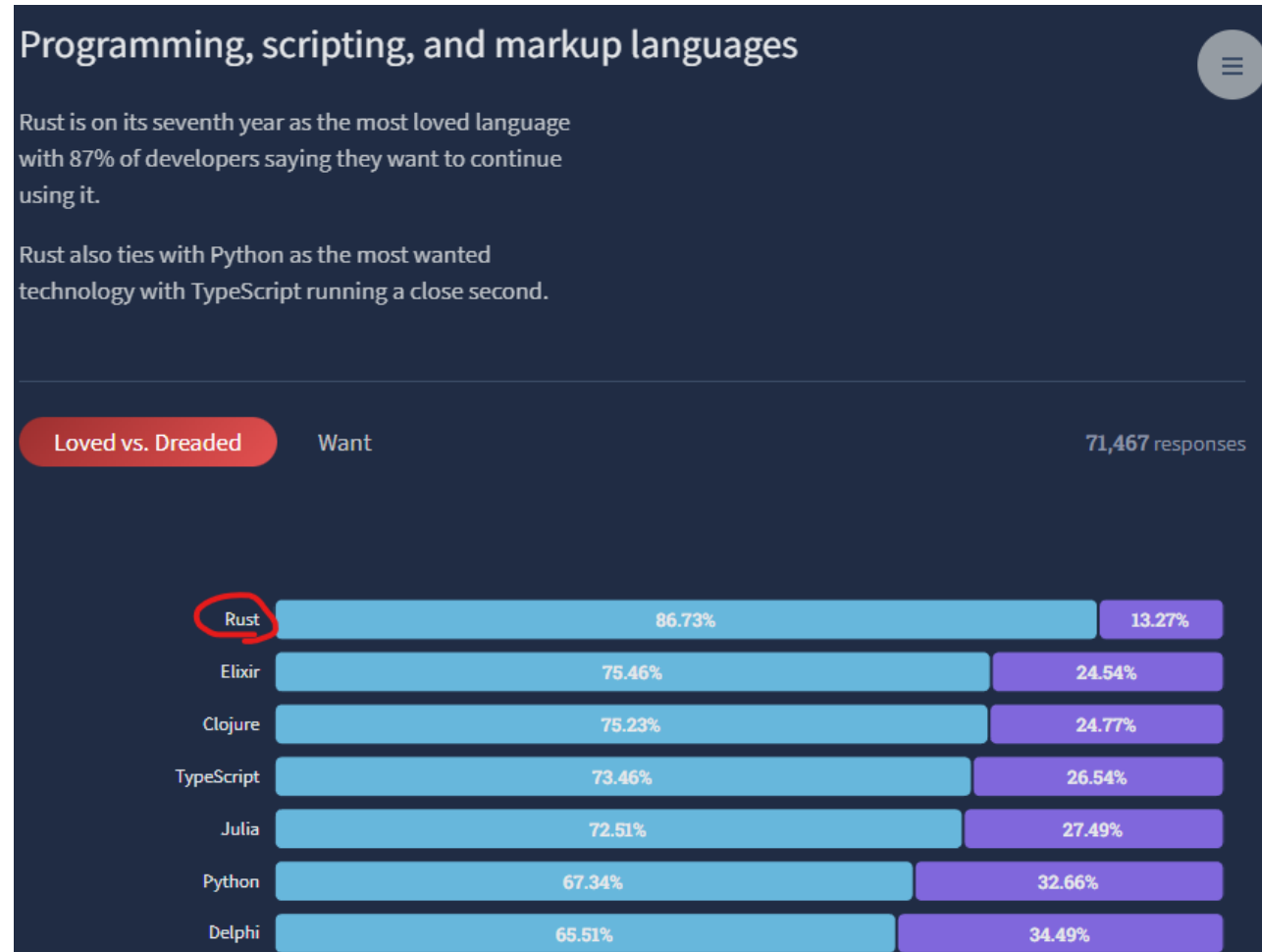
- 메모리 문제를 겪는 프로젝트가 Rust 넘어가고 있음

2. 생태계와 라이브러리

- 메이저한 언어(Java, C++, Python ..)에 비해 정보 부족

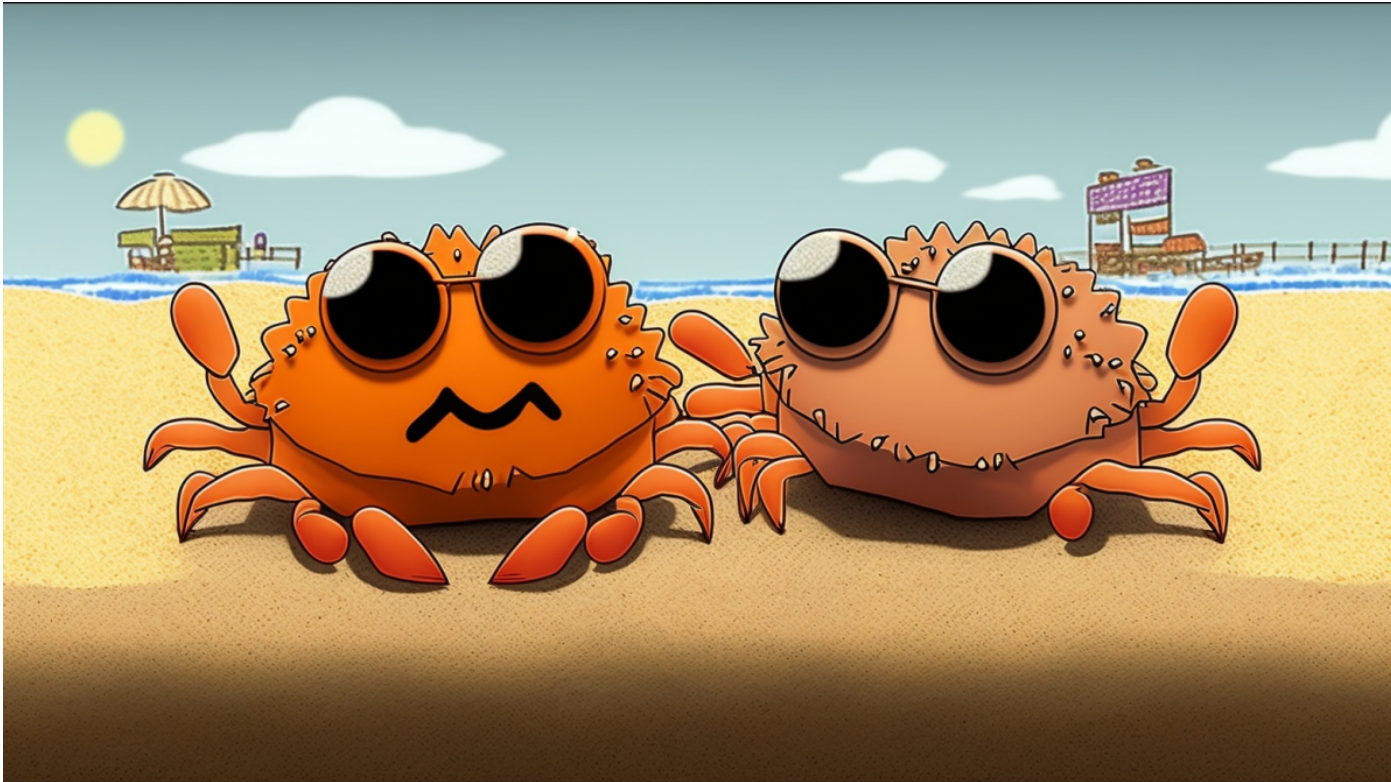
3. 동시성에 특화되어있는 언어

- 그 중 비동기 프로그래밍은 너무 어려움



끝

Rust SIG 모집 중



참고자료

<https://rinthe1.github.io/rust-lang-book-ko/>

<https://prev.rust-lang.org/ko-KR/faq.html>

<https://betterprogramming.pub/reading-and-writing-a-file-in-rust-47d2bc7086ac>

<https://github.com/dani-garcia/vaultwarden>

<https://github.com/emacs-ng/deno>

<https://www.techspot.com/news/97654-how-broken-elevator-led-one-most-loved-programming.html>

<https://survey.stackoverflow.co/2022#section-most-loved-dreaded-and-wanted-programming-scripting-and-markup-languages>

<https://github.com/HomoEfficio/dev-tips/blob/59409afe4f0378f6404e85c3afeb6f4734551a41/Rust%20%EB%A7%9B%EB%B3%B4%EA%B8%B0.md>



Marp

Markdown Presentation Ecosystem