HSPACE Rust 특강

Rust Basic #2 - 레퍼런스

Chris Ohk

utilForever@gmail.com

레퍼런스

- 값의 레퍼런스
- 레퍼런스 다루기
- 레퍼런스 안전성
- 공유 vs 변경
- 오브젝트의 바다와 맞서기

들어가며

- Rust에는 **레퍼런스(Reference)**라는 소유권을 갖지 않는 포인터 타입이 있다.
  - 이들은 자신이 가리키는 대상의 수명에 아무런 영향을 주지 않는다.
  - 오히려 그 반대인데, 레퍼런스는 자신이 가리키는 대상보다 절대로 더 오래 살아 있으면 안 된다.
  - 따라서 모든 레퍼런스가 자신이 가리키는 값보다 더 오래 살아 있을 수 없다는 걸 코드에 명확히 해야 한다.
  - Rust는 이 점을 강조하기 위해서 어떤 값의 레퍼런스를 만드는 걸 두고 **빌려온다**고 표현한다.
  - 빌려온 것은 결국엔 주인에게 꼭 돌려줘야 한다.

값의 레퍼런스

- 이동 문법 때문에 변수가 미초기화 상태가 되어 사용할 수 없는 경우가 종종 있다.
  이때 레퍼런스를 쓰면 소유권에 영향을 주지 않고도 값에 접근할 수 있다.
- 레퍼런스의 종류에는 두 가지가 있다.
  - **공유된 레퍼런스(Shared Reference)** : 참조하는 대상을 읽을 수 있지만 수정할 수 없다.
    하지만 특정 값을 참조하는 공유된 레퍼런스의 수는 동시에 여러 개일 수 있다.
    표현식 `&e`는 `e`의 값을 참조하는 공유된 레퍼런스를 만든다. `e`의 타입이 `T`일 때 `&e`의 타입은 `&T`가 된다.
    공유된 레퍼런스는 `Copy`다.
  - **변경할 수 있는 레퍼런스(Mutable Reference)** : 참조하는 값을 읽을 수도 있고 수정할 수도 있다.
    하지만 그 기간 동안에는 같은 값을 참조하는 다른 모든 종류의 레퍼런스를 사용할 수 없게 된다.
    표현식 `&mut e`는 `e`의 값을 참조하는 변경할 수 있는 레퍼런스를 만든다. 변경할 수 있는 레퍼런스는 `Copy`가 아니다.

값의 레퍼런스

- 공유된 레퍼런스와 변경할 수 있는 레퍼런스를 따로 구분해 둔 이유

  → 컴파일 시점에 **멀티플 리더(Multiple Reader) 또는 싱글 라이터(Single Writer)** 규칙을 시행하기 위해서!

  - 공유된 레퍼런스가 존재하는 동안에는 참조하는 값이 잠기므로 소유자라 할 지라도 값을 수정할 수 없다.
  - 변경할 수 있는 레퍼런스가 존재하는 경우에는 그 레퍼런스가 참조하는 값의 독점적인 접근 권한을 갖는다.
    따라서 이 변경할 수 있는 레퍼런스가 사라질 때까지는 아무도 그 소유자를 사용할 수 없다.
  - 공유와 변경을 완전히 분리해 가져가는 건 메모리 안전성을 지키는 데 있어 가장 중요한 요소다.
- 전달하는 방식
  - **값 전달(Pass by Value)** : 값을 함수에 넘길 때 그 값의 소유권이 함께 넘어간다.
  - **레퍼런스 전달(Pass by Reference)** : 값이 아니라 그 값의 레퍼런스를 넘긴다.

레퍼런스 다루기

- C++ 레퍼런스 vs Rust 레퍼런스
  - C++ 레퍼런스
    - 레퍼런스가 변환에 의해서 암시적으로 만들어진다.
    - 레퍼런스의 역참조가 암시적으로 이뤄진다.
  - Rust 레퍼런스
    - 레퍼런스가 `&` 연산자를 통해서 명시적으로 만들어진다.
    - 레퍼런스의 역참조가 `*` 연산자를 통해서 명시적으로 이뤄진다.
    - 변경할 수 있는 레퍼런스는 `&mut` 연산자를 써서 만든다.
    - 단, `.` 연산자는 필요에 따라 자신의 왼쪽에 있는 피연산자를 암시적으로 역참조하게 되어 있다.
      또한 메소드 호출 시에 필요할 경우 암시적으로 자신의 왼쪽에 있는 피연산자의 레퍼런스를 빌려올 수 있다.
  - 정리
    - C++에서는 레퍼런스와 L-value 간의 변환이 필요한 모든 곳에서 암시적으로 일어난다.
    - Rust에서는 `&`와 `*` 연산자를 써서 레퍼런스를 만들고 따라가야 하며, `.` 연산자를 쓸 때만 암시적인 차용과 역참조가 일어난다.

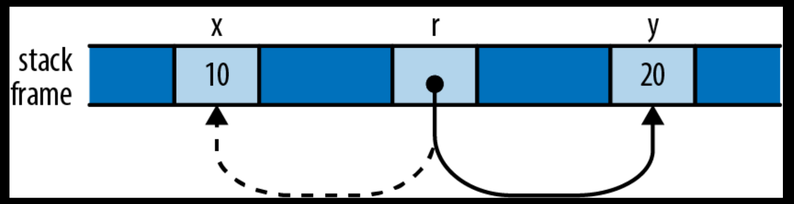
레퍼런스 다루기

- 레퍼런스 배정하기
  - 레퍼런스를 변수에 배정하면 그 변수는 새 위치를 가리키게 된다.
  - C++ 레퍼런스는 한 번 초기화되고 나면 절대로 다른 것을 가리킬 수 없다.

```rust
fn main() {
    let x = 10;
    let y = 20;
    let b = true;
    let mut r = &x;

    if b {
        r = &y;
    }

    assert!(*r == 10 || *r == 20);
}
```

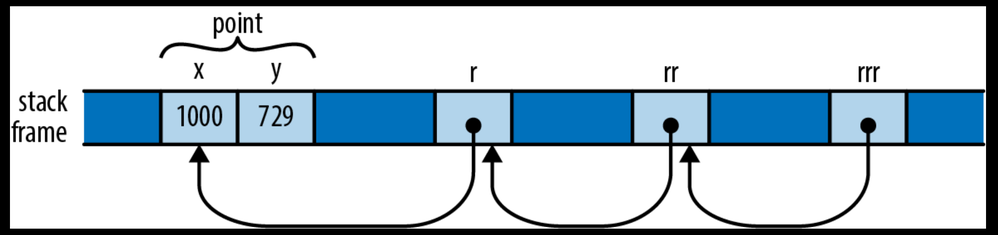


레퍼런스 다루기

- 레퍼런스의 레퍼런스
  - Rust는 레퍼런스의 레퍼런스를 만들 수 있다.
  - . 연산자는 레퍼런스가 최종적으로 가리키는 대상을 찾을 때까지 거듭해서 주어진 연결고리를 따라간다.

```rust
fn main() {
    struct Point {
        x: i32,
        y: i32,
    }

    let point = Point { x: 1000, y: 729 };
    let r: &Point = &point;
    let rr: &&Point = &r;
    let rrr: &&&Point = &rr;
}
```



레퍼런스 다루기

- 레퍼런스 비교하기
  - Rust의 비교 연산자도 `.` 연산자처럼 레퍼런스가 최종적으로 가리키는 대상을 찾을 때까지 주어진 연결고리를 따라간다.
  - 만약 두 레퍼런스가 같은 메모리를 가리키는지 알고 싶다면, 둘의 주소를 비교하는 `std::ptr::eq`를 쓰면 된다.
  - 비교할 때 대상이 되는 피연산자는 반드시 같은 타입이어야 하며, 레퍼런스의 경우도 마찬가지다.

레퍼런스 다루기

- Rust 레퍼런스는 절대로 널이 될 수 없다.
  - Rust에는 C의 `NULL`이나 C++의 `nullptr`에 해당하는 것이 없고, 레퍼런스를 위한 초깃값도 없다.
    또한 Rust는 정수를 레퍼런스로 변환하는 일이 없으므로 `0`을 레퍼런스로 변환할 수도 없다.
  - C, C++ 코드는 널 포인터를 써서 값이 없는 상태를 나타낼 때가 많다.
    Rust에서는 무언가의 레퍼런스일 수도 있고 아닐 수도 있는 값이 필요할 때 `Option<&T>` 타입을 쓴다.
  - `Option<&T>`는 널을 허용하는 C, C++ 포인터만큼 효율적이면서도
    쓰기 전에 `None`인지 아닌지를 꼭 확인해야 하므로 더 안전하다.

레퍼런스 다루기

- 임의의 표현식을 가리키는 레퍼런스 빌려 오기
  - 다음 코드를 보자. 이 상황에서 Rust는 표현식의 값을 보관할 익명 변수를 만들고 이를 가리키는 레퍼런스를 만든다.
    이 익명 변수의 수명은 레퍼런스로 무얼 하는지에 따라 달라진다.
    - `let` 문에서 레퍼런스를 변수에 바로 대입하면 Rust는 익명 변수의 수명을 `let`이 초기화하는 변수의 수명과 같아지도록 만든다.
      앞의 예에서는 `r`이 가리키는 대상이 여기에 해당한다.
    - 그렇지 않으면 익명 변수는 바깥쪽 실행문이 끝날 때까지만 살아 있게 된다.
      아래 예제에서 `1009`를 보관하기 위해 생성되는 익명 변수는 `assert_eq!` 문이 끝날 때까지만 지속된다.

```rust
fn main() {
    fn factorial(n: usize) -> usize {
        (1..=n).product()
    }

    let r = &factorial(6);

    assert_eq!(r + &1009, 1729);
}
```

레퍼런스 다루기

- 슬라이스 레퍼런스와 트레잇 오브젝트
  - 지금까지 살펴본 레퍼런스는 모두 단순한 주소였다.

    그러나 Rust는 두 가지 종류의 **팻 포인터(Fat Pointer)**를 더 갖고 있다.
    - 팻 포인터는 어떤 값의 주소와 그 값을 사용하는데 필요한 추가 정보를 갖는 2워드 크기의 값이다.
    - 슬라이스 레퍼런스는 슬라이스의 시작 주소와 길이를 갖는 팻 포인터다.
    - 또 다른 종류의 팻 포인터인 **트레잇 오브젝트(Trait Object)**는 트레잇을 구현하고 있는 값의 레퍼런스다.

레퍼런스 안전성

- 레퍼런스는 C / C++에 있는 일반적인 포인터와 비슷하다. 하지만 포인터는 안전하지 않다. Rust는 과연 어떤 식으로 레퍼런스를 통제하고 있는 걸까?
  - Rust가 어떤 식으로 함수 본문 안에 있는 레퍼런스의 올바른 쓰임을 보장하는지 보여 주는 사례부터 짚어 본다.
  - 레퍼런스를 함수 간에 전달하는 법, 이를 데이터 구조 안에 저장하는 법을 알아본다.
    이 과정에서 **수명 매개 변수(Lifetime Parameter)**를 함수와 데이터 타입에 부여하는 과정이 수반된다.
  - Rust가 제공하는 자주 있는 사용 패턴을 단순화시켜주는 축약 표기 몇 가지를 살펴 본다.

레퍼런스 안전성

- 지역 변수 빌려오기
  - 지역 변수의 레퍼런스를 빌려올 때는 레퍼런스를 그 변수의 범위 밖으로 가지고 나갈 수 없다.

```
fn main() {
    {
        let r;

        {
            let x = 1;
            r = &x;          // `x` does not live long enough
        }

        assert_eq!(*r, 1);  // Bad: Reads memory `x` used to occupy
    }
}
```

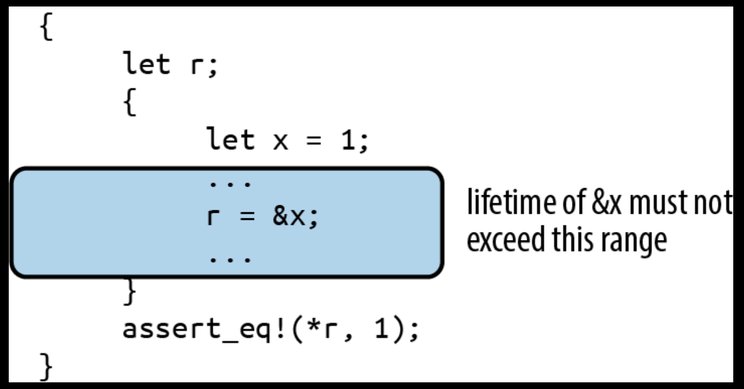
레퍼런스 안전성

- 지역 변수 빌려오기
  - 지역 변수의 레퍼런스를 빌려올 때는 레퍼런스를 그 변수의 범위 밖으로 가지고 나갈 수 없다.

```
error: `x` does not live long enough
  --> references_dangling.rs:8:5
   |
 7 |         r = &x;
   |             ^^ borrowed value does not live long enough
 8 |     }
   |     - `x` dropped here while still borrowed
 9 |     assert_eq!(*r, 1); // bad: reads memory `x` used to occupy
10 | }
```

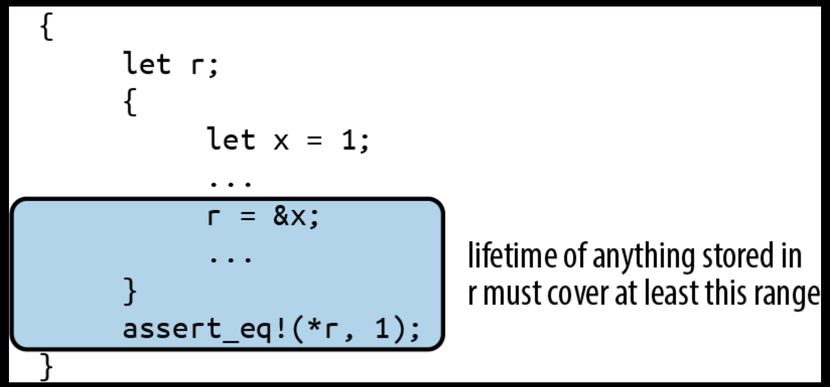
레퍼런스 안전성

- 지역 변수 빌려오기
  - Rust는 프로그램에 있는 모든 레퍼런스 타입을 대상으로 각 타입의 쓰임새에 맞는 제약 조건이 반영된 **수명(Lifetime)**을 부여하려고 한다. (컴파일 시점에만 존재하는 가상의 개념이다.)
    수명이란 실행문, 표현식, 변수 범위 등 프로그램에서 레퍼런스가 안전하게 쓰일 수 있는 구간을 말한다.
  - 변수의 수명은 자신에게서 차용된 레퍼런스의 수명을 반드시 **포함(Contain)**하거나 **에워싸야(Enclose)** 한다.



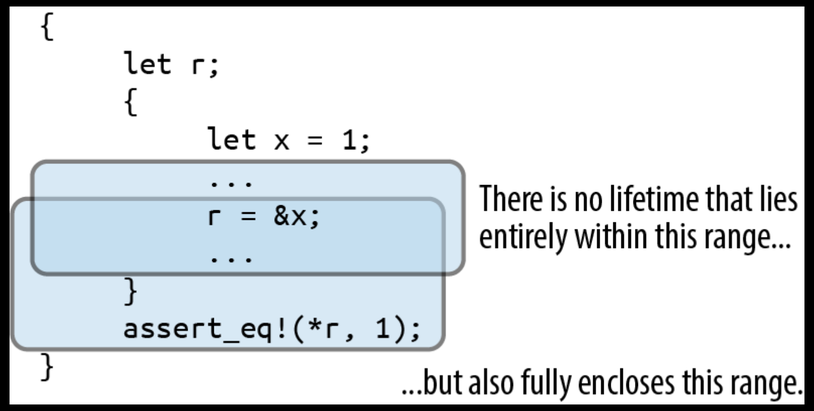
레퍼런스 안전성

- 지역 변수 빌려오기
  - Rust는 프로그램에 있는 모든 레퍼런스 타입을 대상으로 각 타입의 쓰임새에 맞는 제약 조건이 반영된 **수명(Lifetime)**을 부여하려고 한다. (컴파일 시점에만 존재하는 가상의 개념이다.)
    수명이란 실행문, 표현식, 변수 범위 등 프로그램에서 레퍼런스가 안전하게 쓰일 수 있는 구간을 말한다.
  - 변수의 수명은 자신에게서 차용된 레퍼런스의 수명을 반드시 **포함(Contain)**하거나 **에워싸야(Enclose)** 한다.



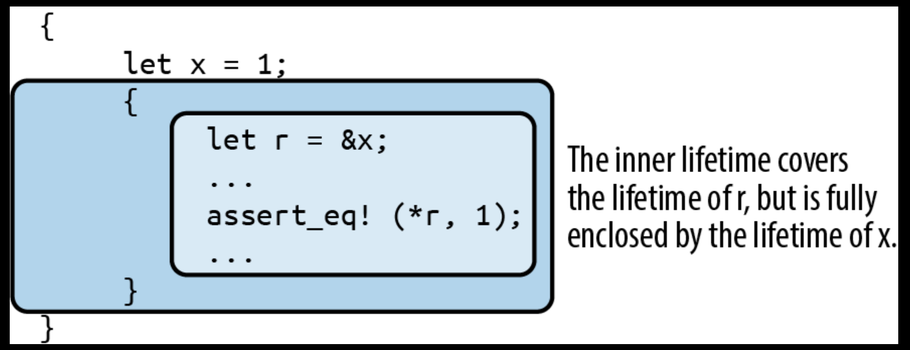
레퍼런스 안전성

- 지역 변수 빌려오기
  - Rust는 프로그램에 있는 모든 레퍼런스 타입을 대상으로 각 타입의 쓰임새에 맞는 제약 조건이 반영된 **수명(Lifetime)**을 부여하려고 한다. (컴파일 시점에만 존재하는 가상의 개념이다.)
    수명이란 실행문, 표현식, 변수 범위 등 프로그램에서 레퍼런스가 안전하게 쓰일 수 있는 구간을 말한다.
  - 변수의 수명은 자신에게서 차용된 레퍼런스의 수명을 반드시 **포함(Contain)**하거나 **에워싸야(Enclose)** 한다.



레퍼런스 안전성

- 지역 변수 빌려오기
  - Rust는 프로그램에 있는 모든 레퍼런스 타입을 대상으로 각 타입의 쓰임새에 맞는 제약 조건이 반영된 **수명(Lifetime)**을 부여하려고 한다. (컴파일 시점에만 존재하는 가상의 개념이다.)
    수명이란 실행문, 표현식, 변수 범위 등 프로그램에서 레퍼런스가 안전하게 쓰일 수 있는 구간을 말한다.
  - 변수의 수명은 자신에게서 차용된 레퍼런스의 수명을 반드시 **포함(Contain)**하거나 **에워싸야(Enclose)** 한다.



레퍼런스 안전성

- 레퍼런스를 갖는 구조체
  - 레퍼런스 타입을 다른 타입의 정의 안에 둘 때는 반드시 수명을 함께 기재해야 한다.

```
struct S<'a> {
    r: &'a i32,
}
```

  - 수명 매개 변수를 가진 타입을 다른 타입 안에 둘 때도 반드시 수명을 함께 기재해야 한다.
    (그 타입에서 다른 필드들이 레퍼런스 타입을 사용하지 않더라도 말이다.)

```
struct D<'a> {
    s: &S<'a>,
}
```

레퍼런스 안전성

- 고유한 수명 매개 변수
  - 다음과 같이 두 개의 레퍼런스를 갖는 구조체를 정의했다고 하자.

```
struct S<'a> {
    x: &'a i32,
    y: &'a i32,
}
```

  - 이 경우에는 코드가 다음과 같은 일을 하려 할 때 문제가 될 수 있다.

```
let x = 10;
let r;

{
    let y = 20;
    {
        let s = S { x: &x, y: &y };
        r = s.x;
    }
}
```

레퍼런스 안전성

- 고유한 수명 매개 변수
  - 앞의 코드가 댕글링 포인터를 만드는 건 아니다. 그렇다면 Rust가 걱정하는 이유는 뭘까?
    - S의 두 필드는 같은 수명 `'a`를 갖는 레퍼런스이므로, Rust는 `s.x`와 `s.y` 모두에게 적합한 수명 하나를 찾아야 한다.
    - 대입문 `r = s.x`는 `'a`가 r의 수명을 에워싸야 한다는 제약 조건을 만든다.
    - `s.y`를 `&y`로 초기화하는 구문은 `'a`가 y의 수명보다 더 길면 안 된다는 제약 조건을 만든다.
  - 즉, y의 범위보다는 짧고 r의 범위보다는 긴 그런 수명은 존재하지 않으므로,
    위의 제약 조건을 만족시킬 수 없다고 판단한 Rust가 컴파일을 중단시키는 것이다.

레퍼런스 안전성

- 고유한 수명 매개 변수
  - 문제는 S 안에 있는 두 레퍼런스가 같은 수명 ‘a를 갖는다는 데 있다.
    따라서 각 레퍼런스가 고유한 수명을 갖도록 S의 정의를 바꾸면 모든 문제가 해결된다.
    - 이 정의에서는 `s.x`와 `s.y`가 독립된 수명을 갖는다.
      이제 `s.x`로 무엇을 하든지 `s.y`에 저장한 것에 영향을 주지 않으므로 제약 조건을 만족시키기가 쉽다.

```
struct S<'a, 'b> {
    x: &'a i32,
    y: &'b i32,
}
```

레퍼런스 안전성

- 수명 매개 변수 생략하기
  - 코드가 아주 단순할 때는 매개 변수의 수명을 기재하지 않아도 된다.
    Rust가 필요한 곳에 알아서 고유한 수명을 배정해 준다.

```
struct S<'a, 'b> {
    x: &'a i32,
    y: &'b i32,
}

fn sum_r_xy(r: &i32, s: S) -> i32 {
    r + s.x + s.y
}
```

  - 이 함수의 시그니처는 다음처럼 써야 하는 것이 줄어든 것이다.

```
fn sum_r_xy<'a, 'b, 'c>(r: &'a i32, s: S<'b, 'c>) -> i32
```

레퍼런스 안전성

- 수명 매개 변수 생략하기
  - 함수가 어떤 타입의 메소드면서 `self` 매개 변수를 레퍼런스로 받는 경우에는 에외 규정이 적용되어 `self`의 수명이 반환 값에 배정된다.

```rust
struct StringTable {
    elements: Vec<String>,
}

impl StringTable {
    fn find_by_prefix(&self, prefix: &str) -> Option<&String> {
        for i in 0..self.elements.len() {
            if self.elements[i].starts_with(prefix) {
                return Some(&self.elements[i]);
            }
        }

        None
    }
}
```

레퍼런스 안전성

- 수명 매개 변수 생략하기
  - 여기서 `find_by_prefix` 메소드의 시그니처는 다음처럼 써야 하는 것이 줄어든 것이다.

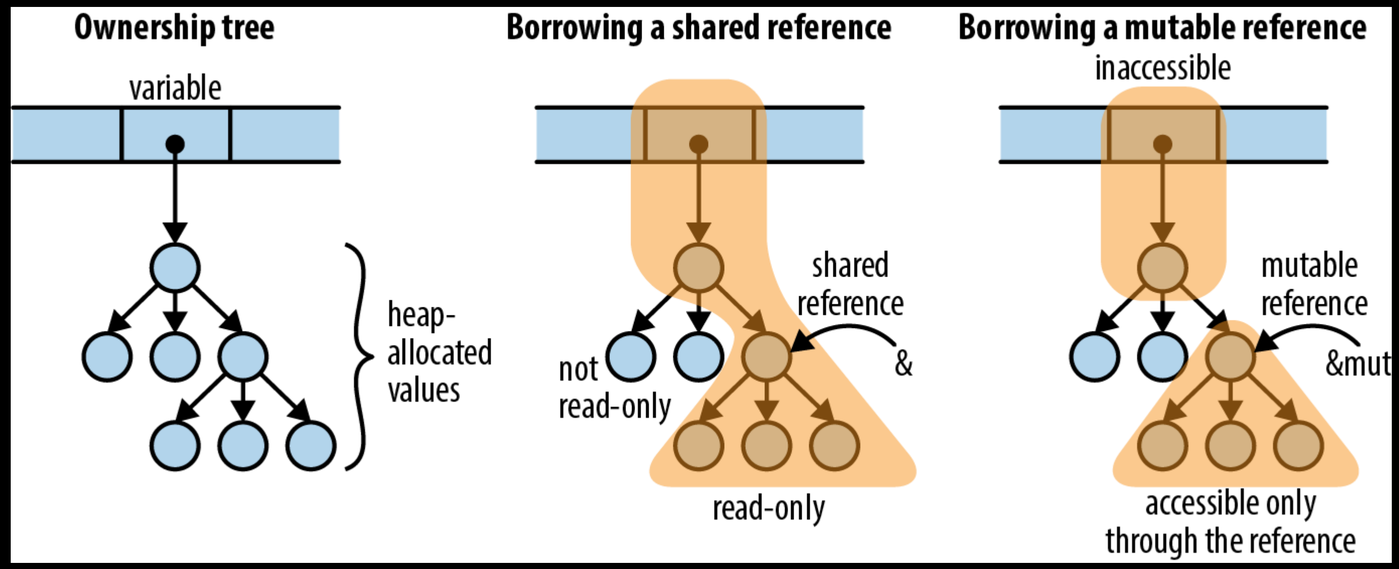
```
fn find_by_prefix<'a, 'b>(&'a self, prefix: &'b str) -> Option<&'a String>
```

공유 vs 변경

- Rust의 변경과 공유에 관한 규칙
  - **공유된 접근은 읽기 전용 접근이다.**
    - 공유된 레퍼런스로 빌려온 값은 읽을 수만 있다.
      공유된 레퍼런스가 살아 있는 동안에는 **그 무엇도** 참조 대상이나 참조 대상을 통해 도달할 수 있는 다른 대상을 변경할 수 없다.
      소유자 역시 읽기 전용으로 설정되기 때문에 이 구조에 관여된 대상의 변경할 수 있는 레퍼런스가 아예 존재할 수 없다.
      한마디로 동결 상태라고 보면 된다.
  - **변경할 수 있는 접근은 배타적인 접근이다.**
    - 변경할 수 있는 레퍼런스로 빌려온 값은 그 레퍼런스를 통해서만 접근할 수 있다.
      변경할 수 있는 레퍼런스가 살아 있는 동안에는 참조 대상이나 참조 대상을 통해 도달할 수 있는 다른 대상에 접근할 수 있는 경로가 없다.
      변경할 수 있는 레퍼런스와 수명이 겹칠 수 있는 유일한 레퍼런스는 변경할 수 있는 레퍼런스 그 자체에서 빌려온 것들 뿐이다.

공유 vs 변경

- 레퍼런스는 유형에 따라서 참조 대상에 이르는 소유 경로상의 값들과 참조 대상을 통해 도달할 수 있는 값들을 가지고 할 수 있는 일이 다르다.



공유 vs 변경

- 간단한 예를 통해서 확인해 보자.

```rust
let mut x = 10;
let r1 = &x;
let r2 = &x;                    // OK: Multiple shared borrows permitted
x += 10;                        // Error: Cannot assign to `x` because it is borrowed
let m = &mut x;                 // Error: Cannot borrow `x` as mutable because it is also borrowed as immutable
println!("{r1}, {r2}, {m}");    // The references are used here, so their lifetimes must last at least this long

let mut y = 20;
let m1 = &mut y;
let m2 = &mut y;                // Error: Cannot borrow as mutable more than once
let z = y;                      // Error: Cannot use `y` because it was mutably borrowed
println!("{m1}, {m2}, {z}");    // References are used here
```

공유 vs 변경

- 공유된 레퍼런스에게서 다시 공유된 레퍼런스를 빌려오는 건 문제없다.

```
let mut w = (107, 109);
let r = &w;
let r0 = &r.0;      // OK: Reborrowing shared as shared
let m1 = &mut r.1;  // Error: Can't reborrow shared as mutable
println!("{r0}");   // r0 gets used here
```

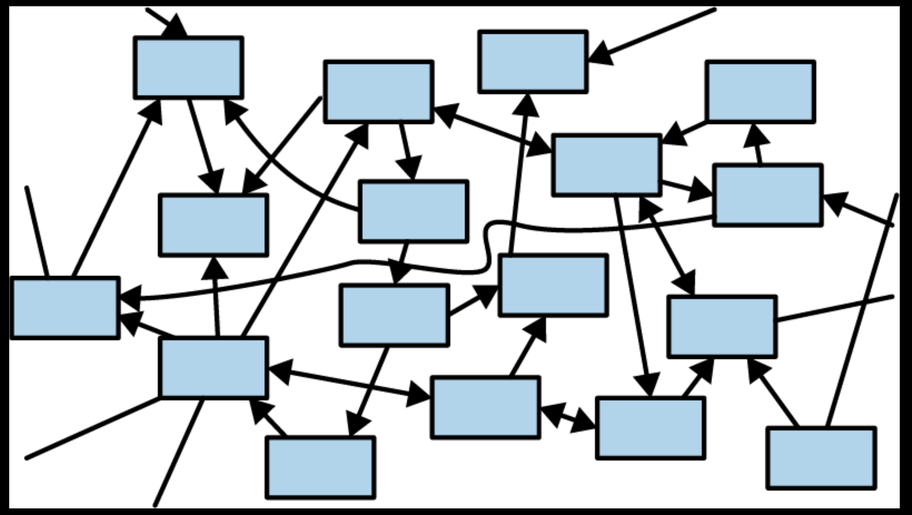
공유 vs 변경

- 변경할 수 있는 레퍼런스에게서는 어떤 레퍼런스든 다시 빌려올 수 있다.

```rust
let mut v = (136, 139);
let m = &mut v;
let m0 = &mut m.0;  // OK: reborrowing mutable from mutable
*m0 = 137;
let r1 = &m.1;      // OK: Reborrowing shared from mutable, and doesn't overlap with m0
v.1;                // Error: Access through other paths still forbidden
println!("{r1}");   // r1 gets used here
```

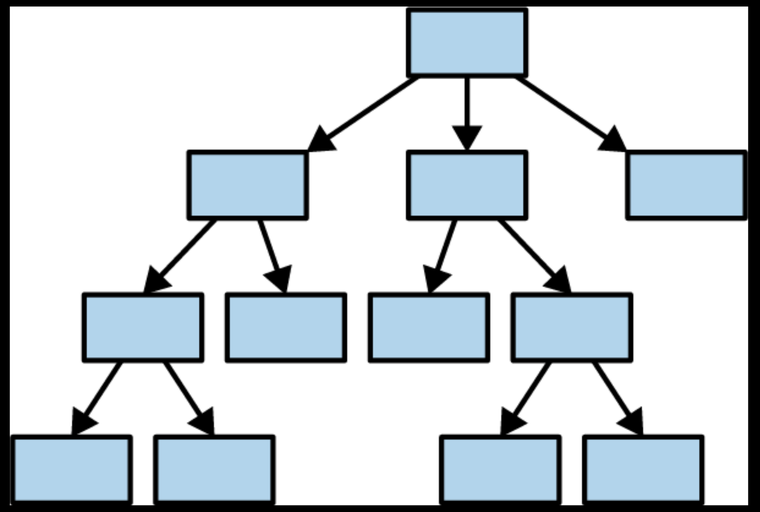
오브젝트의 바다와 맞서기

- 1990년대에 자동 메모리 관리가 등장하면서 이후 모든 프로그램은 **오브젝트의 바다(Sea of Objects)**를 기본 아키텍처로 삼아 왔다.
  - 초기 진행 속도가 빠르고, 여기저기 끼워 맞추기 쉬우며,
    언젠가 완전히 다시 만들고 싶어지더라도 큰 어려움 없이 정당화할 수 있다는 장점이 있다.
  - 하지만 모두가 모두에게 의존하는 구조는 테스트하기가 까다롭고, 확장하는 데도 한계가 따르며,
    심지어 구성 요소를 따로 분리해 생각하는 것조차 어렵다.



오브젝트의 바다와 맞서기

- Rust의 한 가지 흥미로운 점은 소유 모델이
  이 '지옥행 고속도로'에 과속 방지턱을 설치한다는 것이다.
  - Rust에서 두 값이 서로 상대방을 가리키는 순환 구조를 만들려면 Rc 같은 스마트 포인터 타입과 함께
    아직 다루지 않은 주제인 **내부 가변성(Interior Mutability)**을 활용해야 하는 등 약간의 노력이 필요하다.



감사합니다.

utilForever@gmail.com

https://github.com/utilForever

Facebook, Twitter: @utilForever