

고려대학교 GDSC 스터디

A Tour of Rust, Part 3

Chris Ohk

utilForever@gmail.com

- 클로저
- 변수 캡처하기
 - 빌리는 클로저
 - 훔치는 클로저
- 함수와 클로저 타입
- 클로저 성능
- 클로저와 안전성
 - 드롭하는 클로저
 - FnOnce
 - FnMut
 - 클로저를 위한 Copy와 Clone

- 구조체 City를 정렬하고 싶다. 무엇을 기준으로 정렬해야 할까?
 - 컴파일해보면 City가 std::cmp::Ord를 구현하지 않았다고 오류를 출력한다.

```
struct City {  
    name: String,  
    population: i64,  
    country: String,  
}  
  
fn sort_cities(cities: &mut Vec<City>) {  
    cities.sort(); // ???  
}
```

- 구조체 City를 정렬하고 싶다. 무엇을 기준으로 정렬해야 할까?
 - 이 때는 정렬 순서를 지정해 주면 된다.

```
struct City {  
    name: String,  
    population: i64,  
    country: String,  
}  
  
fn city_population_descending(city: &City) -> i64 {  
    -city.population  
}  
  
fn sort_cities(cities: &mut Vec<City>) {  
    cities.sort_by_key(city_population_descending);  
}
```

- 구조체 City를 정렬하고 싶다. 무엇을 기준으로 정렬해야 할까?
 - 하지만 익명의 함수 표현식인 클로저를 작성하면 보다 간결한 코드를 작성할 수 있다.

```
struct City {  
    name: String,  
    population: i64,  
    country: String,  
}  
  
fn sort_cities(cities: &mut Vec<City>) {  
    cities.sort_by_key(|city| -city.population);  
}
```

- 클로저는 바깥쪽 함수에 속한 데이터를 사용할 수 있다.
- 아래 예제에서 클로저는 `sort_by_statistic`이 소유한 `stat`을 사용한다.
- 이를 가리켜 클로저가 `stat`을 캡처(Capture)한다고 말한다.



```
fn sort_by_statistic(cities: &mut Vec<City>, stat: Statistic) {  
    cities.sort_by_key(|city| -city.get_statistic(stat));  
}
```

- 다음 클로저는 어떤 방식으로 동작할까?
 - 클로저를 갖춘 대부분의 언어들에서는 가비지 컬렉션이 중요한 역할을 한다.
 - 러스트는 가비지 컬렉션이 없는데, 이 문제를 어떻게 해결할까?

```
function startSortingAnimation(cities, stat) {  
  function keyfn(city) {  
    return city.get_statistic(stat);  
  }  
  
  if (pendingSort)  
    pendingSort.cancel();  
  
  pendingSort = new SortingAnimation(cities, keyfn);  
}
```

- 앞에서 봤던 함수 `sort_by_statistic`을 다시 보자.
 - 러스트가 클로저를 생성할 때 `stat`의 레퍼런스를 자동으로 빌린다.
 - 클로저는 빌림과 수명에 관한 규칙을 적용받는다.
(앞서 나온 클로저는 `stat`의 레퍼런스를 갖고 있으므로 `stat`보다 더 오래 살아있을 수 없다.)
- 러스트는 가비지 컬렉터 대신 수명을 써서 안전성을 보장한다.



```
fn sort_by_statistic(cities: &mut Vec<City>, stat: Statistic) {  
    cities.sort_by_key(|city| -city.get_statistic(stat));  
}
```


- 이번 예제는 좀 더 까다롭다.
 - `thread::spawn`은 주어진 클로저를 새 시스템 스레드에서 호출한다.
 - 새 스레드는 호출부와 병렬로 실행되며, 클로저가 복귀하면 종료된다.
 - 이번 예제에서도 클로저 `key_fn`은 `stat`의 레퍼런스를 갖는다.

```
use std::thread;

fn start_sorting_thread(mut cities: Vec<City>, stat: Statistic)
    -> thread::JoinHandle<Vec<City>> {
    let key_fn = |city: &City| -> i64 { -city.get_statistic(stat) };

    thread::spawn(|| {
        cities.sort_by_key(key_fn);
        cities
    })
}
```

- 이번 예제는 좀 더 까다롭다.
 - 하지만 이번에는 러스트가 레퍼런스의 안전한 사용을 보장할 수 없어서 컴파일 오류가 발생한다.
 - 컴파일 오류가 발생하는 이유는, `thread::spawn`이 생성한 새 스레드가 앞서 나온 함수의 끝에서 `cities`와 `stat`이 소멸되기 전에 자신의 일을 마치리라고 기대할 수 없기 때문이다.
 - 그렇다면 어떻게 해결해야 할까?

```
use std::thread;

fn start_sorting_thread(mut cities: Vec<City>, stat: Statistic)
    -> thread::JoinHandle<Vec<City>> {
    let key_fn = |city: &City| -> i64 { -city.get_statistic(stat) };

    thread::spawn(|| {
        cities.sort_by_key(key_fn);
        cities
    })
}
```

- 이번 예제는 좀 더 까다롭다.
 - 러스트에게 cities와 stat의 레퍼런스를 빌리지 말고 대신 이 둘을 앞서 나온 클로저 안으로 옮겨달라고 말하는 것이다.
 - move 키워드는 러스트에게 클로저가 자신이 사용하는 변수를 빌리지 않고 훔친다고 말한다.

```
use std::thread;

fn start_sorting_thread(mut cities: Vec<City>, stat: Statistic)
    -> thread::JoinHandle<Vec<City>> {
    let key_fn = move |city: &City| -> i64 { -city.get_statistic(stat) };

    thread::spawn(move || {
        cities.sort_by_key(key_fn);
        cities
    })
}
```

- 함수와 클로저에도 타입이 있다.
- 다음 함수는 &City 하나를 인수로 받아 i64를 반환하며, 타입은 fn(&City) -> i64다.

```
fn city_population_descending(city: &City) -> i64 {  
    -city.population  
}
```

- 함수는 다른 값들과 똑같이 취급해 사용할 수 있다.
- 함수를 변수에 저장할 수도 있고, 함수값을 계산할 때도 통상의 러스트 문법을 모두 사용할 수 있다.

```
let my_key_fn: fn(&City) -> i64 =  
    if user.prefs.by_population {  
        city_population_descending  
    } else {  
        city_monster_attack_risk_descending  
    };  
  
cities.sort_by_key(my_key_fn);
```

함수와 클로저 타입

- 함수와 클로저에도 타입이 있다.
 - 함수는 다른 함수를 인수로 받을 수 있다.

```
fn count_selected_cities(cities: &Vec<City>, test_fn: fn(&City) -> bool) -> usize {  
    let mut count = 0;  
  
    for city in cities {  
        if test_fn(city) {  
            count += 1;  
        }  
    }  
  
    count  
}  
  
fn has_monster_attacks(city: &City) -> bool {  
    city.monster_attack_risk > 0.0  
}  
  
let n = count_selected_cities(&my_cities, has_monster_attacks);
```

함수와 클로저 타입

- 하지만 클로저의 타입은 함수의 타입과 다르다.



```
let limit = preferences.acceptable_monster_risk();  
let n = count_selected_cities(&my_cities, |city| city.monster_attack_risk > limit);
```

- count_selected_cities 함수가 클로저를 지원하기 위해서는 타입 시그니처를 바꿔야 한다.



```
fn count_selected_cities(cities: &Vec<City>, test_fn: F) -> usize  
    where F: Fn(&City) -> bool  
{  
    let mut count = 0;  
  
    for city in cities {  
        if test_fn(city) {  
            count += 1;  
        }  
    }  
  
    count  
}
```

- 모든 클로저는 자기만의 고유한 타입을 갖는다.
 - 클로저가 바깥쪽 범위에서 빌리거나 훔친 값을 데이터로 가질 수도 있고, 또 가지고 있는 변수의 수와 타입의 조합이 전부 제각각일 수 있기 때문이다.
 - 따라서 모든 클로저는 컴파일러로부터 자신의 데이터가 전부 들어갈 만한 크기를 가진 임시 타입을 부여받는다.
 - 똑같은 타입을 가진 클로저란 있을 수 없지만, 모든 클로저는 Fn 트레잇을 구현한다.



```
fn(&City) -> bool    // fn 타입 (함수만 받는다.)  
Fn(&City) -> bool    // Fn 트레잇 (함수와 클로저를 모두 받는다.)
```

- 대부분의 언어에서는 클로저가 힙에 할당되고, 동적으로 디스패치되고, 가비지 컬렉션된다.
- 따라서 클로저를 생성하고, 호출하고, 컬렉션할 때마다 약간의 추가 CPU 시간이 소모된다.
- 또한 클로저는 인라인 처리의 대상이 되지 못하는 경우가 많다.
- 러스트의 클로저는 가비지 컬렉션되지 않으며 box나 Vec 등의 컨테이너에 집어넣지 않는 이상 힙에 할당되지 않는다.
- 또한 각 클로저가 서로 다른 타입을 갖기 때문에, 러스트 컴파일러가 여러분이 호출하고 있는 클로저의 타입을 인식할 때마다 해당 클로저의 코드를 인라인 처리할 수 있다.

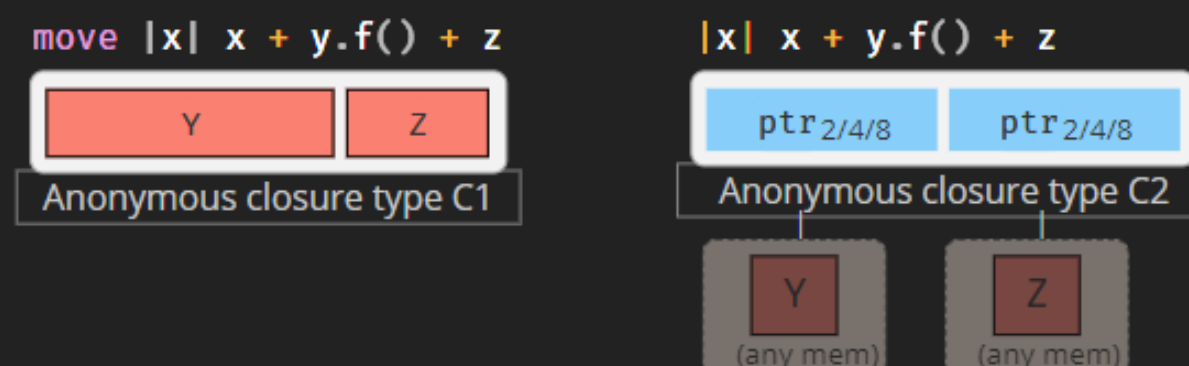
Closures

Ad-hoc functions with an automatically managed data block **capturing** ^{REF. 1} environment where closure was defined. For example, if you had:

```
let y = ...;
let z = ...;

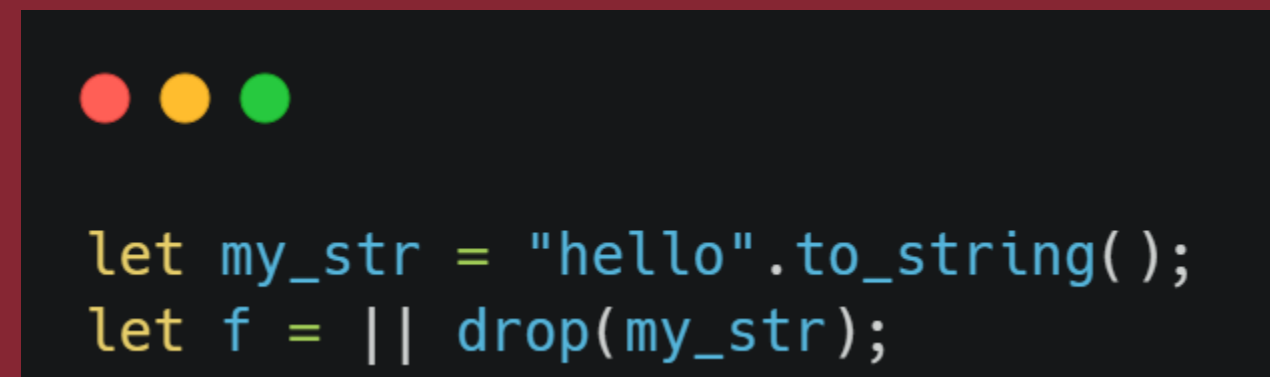
with_closure(move |x| x + y.f() + z); // y and z are moved into closure instance (of type C1)
with_closure(|x| x + y.f() + z); // y and z are pointed at from closure instance (of type C2)
```

Then the generated, anonymous closure types c1 and c2 passed to with_closure() would look like:



- 클로저는 항상 안전할까? 그렇지 않다.
 - 클로저가 캡처된 값을 드롭하거나 수정할 때 벌어지는 일들을 살펴봐야 한다.

- 러스트에서 값을 드롭하는 가장 간단한 방법은 `drop()`을 호출하는 것이다.

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains two lines of Rust code:

```
let my_str = "hello".to_string();  
let f = || drop(my_str);
```

- 이때 클로저 `f`를 두 번 호출하면 무슨 일이 벌어질까?
 - `f`를 처음 호출하면 `my_str`이 드롭된다. 문자열이 저장되어 있던 메모리가 해제되어 시스템에 반환된다는 의미다.
 - 문제는 `f`를 다시 호출했을 때도 똑같은 일이 벌어진다는 거다. 이를 바로 중복 해제(Double Free)라고 한다.
- 러스트는 위의 클로저가 두 번 호출될 수 없다는 걸 알고 있다.
 - 값이 소모된다는(즉, 이동된다는) 발상은 러스트의 핵심 개념 중 하나다.
 - 클로저에도 이 개념이 적용된다.

- 러스트를 다시 한 번 속여서 String이 두 번 드롭되게 해보자.

```
fn call_twice<F>(closure: F) where F: Fn() {  
    closure();  
    closure();  
}
```

- 그런데 앞서 나온 제네릭 함수에 안전하지 않은 클로저를 전달하면 무슨 일이 벌어질까?

```
let my_str = "hello".to_string();  
let f = || drop(my_str);  
call_twice(f);
```

- 이번에도 이 클로저를 호출하면 `my_str`이 드롭된다.
따라서 이를 두 번 호출하면 중복 해제가 발생한다. 그러나 러스트는 속지 않는다.

```
error[E0525]: expected a closure that implements the `Fn` trait, but this closure only implements `FnOnce`
--> src/main.rs:11:13
11 |     let f = || drop(my_str);
    |             ^^          ----- closure is `FnOnce` because it moves the variable `my_str` out of its environment
    |             |
    |             this closure implements `FnOnce`, not `Fn`
12 |     call_twice(f);
    |     ----- the requirement to implement `Fn` derives from here
    |     required by a bound introduced by this call
```

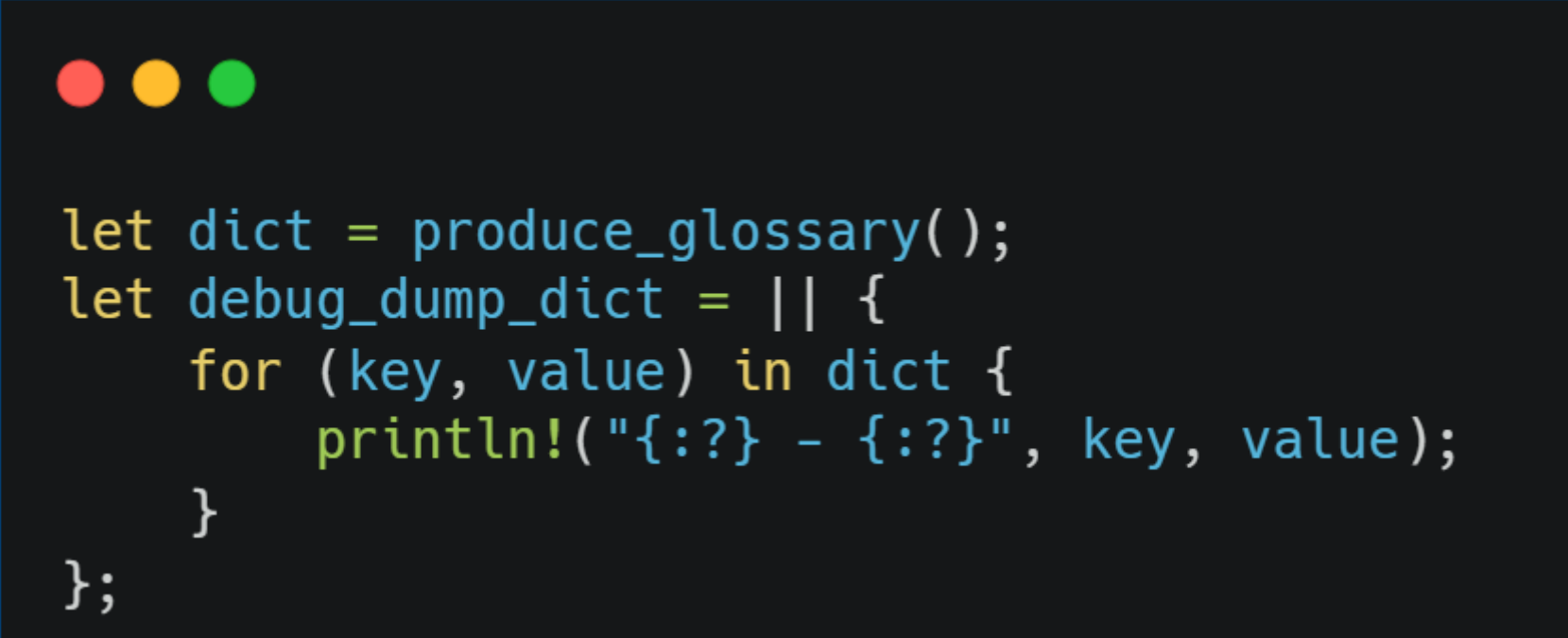
- 이 오류 메시지는 러스트가 '죽이는 클로저'를 다루는 법에 대해 자세히 알려준다.
- 죽이는 클로저를 언어에서 아예 금지할 수도 있지만, 경우에 따라서는 이런 식으로 뒷정리를 하는 클로저가 유용할 때도 있기 때문에 러스트는 이를 금지하는 대신 용도를 제한하고 있다.
- `f`처럼 값을 드롭하는 클로저는 `Fn`을 가질 수 없다. 말 그대로 `Fn`이 아닌 것이다.
그 대신 덜 강력한 트레이트인 `FnOnce`를 구현하는데, 이는 한 번만 호출될 수 있는 클로저의 트레이트이다.

- FnOnce 클로저는 처음 호출될 때 클로저 자체가 소모된다.
Fn과 FnOnce는 다음처럼 정의되어 있다고 보면 된다.

```
trait Fn() -> R {  
    fn call(&self) -> R;  
}  
  
trait FnOnce() -> R {  
    fn call_once(self) -> R;  
}
```

- 러스트는 closure()를 위 예에 나와 있는 두 트레이트 메서드 중 하나의 축약 표기로 취급한다.
- Fn 클로저의 경우에는 closure()가 closure.call()로 확장된다.
이 메서드는 self를 레퍼런스로 받으므로 클로저가 이동되지 않는다.
- 그러나 클로저가 처음 호출될 때만 안전한 경우에는 closure()가 closure.call_once()로 확장된다.
이 메서드는 self를 값으로 받으므로 클로저가 소모된다.

- 어쩌다 한 번쯤은 나도 모르게 값을 소모하는 클로저 코드를 작성할 수도 있다.



```
let dict = produce_glossary();
let debug_dump_dict = || {
    for (key, value) in dict {
        println!("{:?} - {:?}", key, value);
    }
};
```

- `debug_dump_dict()`를 두 번 이상 호출하면, 다음과 같은 오류 메시지가 표시된다.

```
error[E0525]: expected a closure that implements the `Fn` trait, but this closure only implements `FnOnce`
--> src\main.rs:11:13
11 |     let f = || drop(my_str);
    |             ^^          ----- closure is `FnOnce` because it moves the variable `my_str` out of its environment
    |             |
    |             this closure implements `FnOnce`, not `Fn`
12 |     call_twice(f);
    |     ----- - the requirement to implement `Fn` derives from here
    |     |
    |     required by a bound introduced by this call
17 |     debug_dump_dict();
    |     ----- `debug_dump_dict` moved due to this call
18 |     debug_dump_dict();
    |     ^^^^^^^^^^^^^^^^^ value used here after move

note: closure cannot be invoked more than once because it moves the variable `dict` out of its environment
--> src\main.rs:12:29
12 |         for (key, value) in dict {
    |                             ^^^^

note: this value implements `FnOnce`, which causes it to be moved when called
--> src\main.rs:17:5
17 |     debug_dump_dict();
    |     ^^^^^^^^^^^^^^^^^
```

- 이를 디버깅하려면 클로저가 FnOnce인 이유를 알아내야 한다.
 - 컴파일러는 앞의 코드에서 참조하고 있는 유일한 값인 dict를 유력한 용의자로 지목하고 있다.
 - 바로 여기에 버그가 있다. dict를 직접 반복 처리하는 바람에 값이 소모되고 있는 것이다.
 - 따라서 그냥 dict 대신 &dict를 반복 처리하게 고쳐서 레퍼런스로 값을 접근하도록 만들어야 한다.

```
let dict = produce_glossary();
let debug_dump_dict = || {
    for (key, value) in &dict {
        println!("{:?} - {:?}", key, value);
    }
};
```

- 이렇게 하면 문제가 해결된다. 이제 이 함수는 Fn이므로 여러 번 호출할 수 있다.

- 클로저의 종류에는 변경할 수 있는 데이터가 mut 레퍼런스를 갖고 있는 클로저도 있다.
- 러스트는 mut가 아닌 값을 여러 스레드에 공유해도 안전하다고 본다.

그러나 mut 데이터를 갖는 mut가 아닌 클로저를 공유하는 건 안전하지 않다.

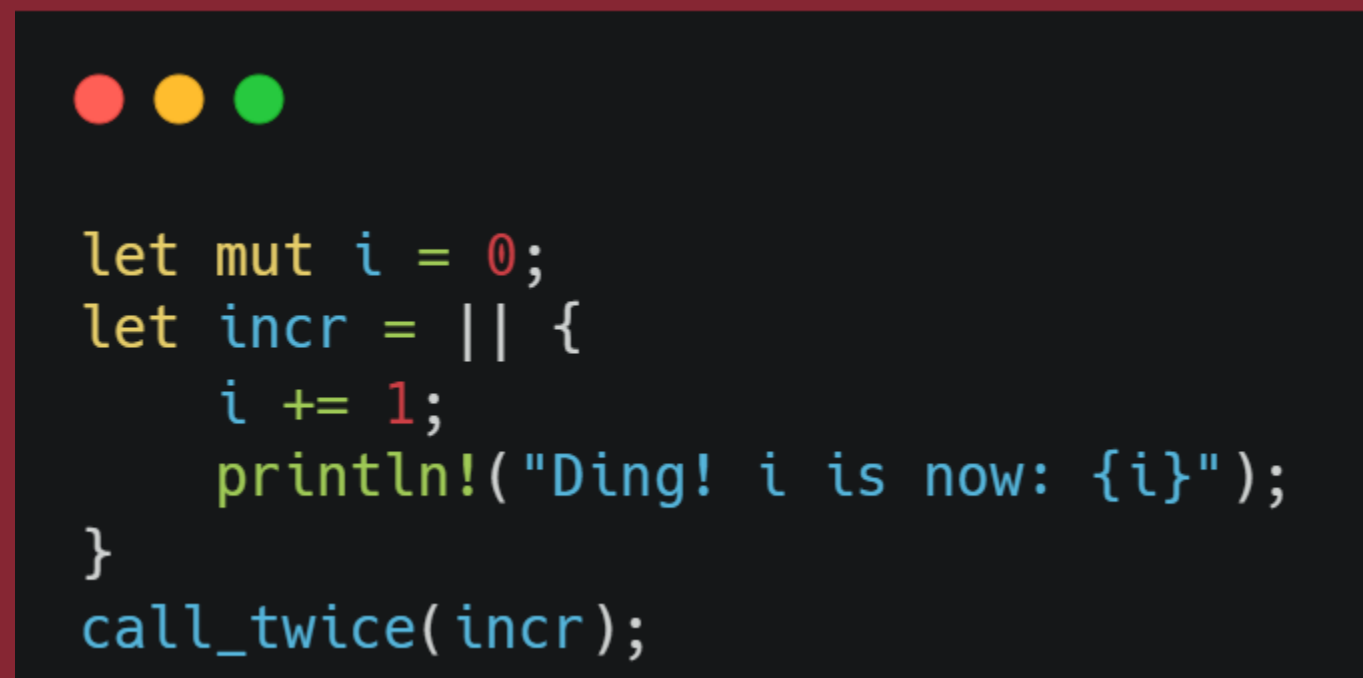
- 이런 클로저를 여러 스레드에서 호출하게 되면 스레드들이 동시에 같은 데이터를 읽고 쓰려 하기 때문에 온갖 종류의 경합 상태가 발생할 수 있기 때문이다.

- 따라서 러스트는 클로저의 범주를 하나 더 마련해 뒀는데, 쓰는 클로저를 위한 FnMut다.
- FnMut 클로저는 mut 레퍼런스를 통해 호출되며, 다음과 같이 정의되어 있다.

```
trait Fn() -> R {  
    fn call(&self) -> R;  
}  
  
trait FnMut() -> R {  
    fn call_mut(&mut self) -> R;  
}  
  
trait FnOnce() -> R {  
    fn call_once(self) -> R;  
}
```

- FnMut 클로저는 값을 mut 접근 권한을 필요로 하지만 아무런 값도 드롭하지 않는 클로저다.

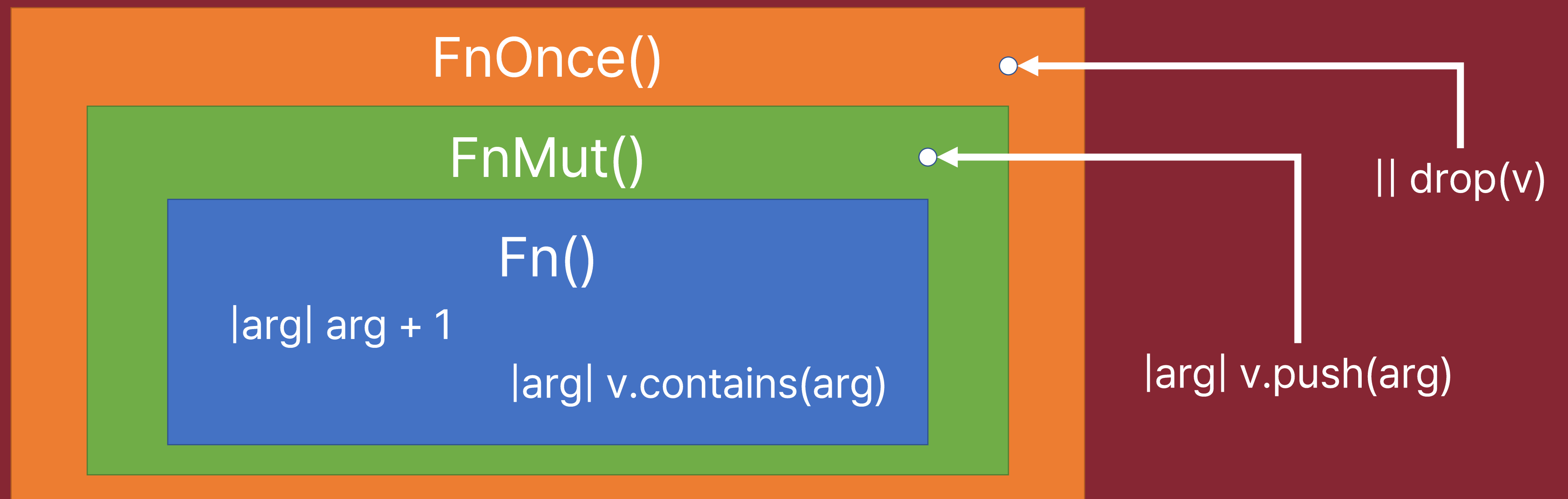
- 다음 예제를 보자.



```
let mut i = 0;
let incr = || {
    i += 1;
    println!("Ding! i is now: {i}");
}
call_twice(incr);
```

- 앞서 작성한 `call_twice`는 `Fn`을 요구한다. 그러나 `incr`은 `Fn`이 아니라 `FnMut`이므로 앞의 코드는 컴파일되지 않는다.
- 이 문제의 해결책을 이해하려면 지금까지 살펴본 러스트 클로저의 세 가지 범주에 관한 내용을 살펴봐야 한다.

- 러스트 클로저의 세 가지 범주
 - Fn은 제한 없이 여러 번 호출할 수 있는 클로저와 함수군이다. 모든 fn 함수 역시 이 최상위 범주에 포함된다.
 - FnMut는 클로저 자체를 mut로 선언한 경우에 여러 번 호출할 수 있는 클로저군이다.
 - FnOnce는 호출부가 클로저를 소유한 경우에 한 번만 호출할 수 있는 클로저군이다.
- 모든 Fn은 FnMut의 요건을 충족하고, 모든 FnMut는 FnOnce의 요건을 충족한다.



- Fn()은 FnMut()의 서브트레잇이고 FnMut()는 FnOnce()의 서브트레잇이다.
 - 따라서 Fn은 가장 배타적인 범주이자 가장 강력한 범주다.
 - FnMut와 FnOnce는 용도에 제한이 있는 클로저를 포함하는 더 넓은 범주다.
- 이제 해결책이 명확히 보인다. 다음처럼 call_twice 함수가 모든 FnMut 클로저를 받도록 고쳐서 수용할 수 있는 클로저의 범주를 넓히면 된다.

```
fn call_twice<F>(mut closure: F) where F: FnMut() {  
    closure();  
    closure();  
}
```

클로저를 위한 Copy와 Clone

- 러스트는 한 번만 호출될 수 있는 클로저를 알아서 파악했던 것처럼, Copy와 Clone을 구현할 수 있는 클로저와 그럴 수 없는 클로저를 파악할 수 있다.
- 클로저는 (move 클로저일 경우) 자신이 캡처하는 변수의 값이나 (move가 아닌 클로저일 경우) 그 값의 레퍼런스를 갖는 구조체로 표현된다.
- 클로저를 위한 Copy와 Clone 규칙은 일반적인 구조체를 위한 Copy와 Clone 규칙과 같다.
- 변수를 변경하지 않는 move가 아닌 클로저는 Clone이면서 Copy인 공유된 레퍼런스만 취하므로, 클로저 역시 Clone이면서 Copy다.

```
let y = 10;
let add_y = |x| x + y;
let copy_of_add_y = add_y;
assert_eq!(add_y(copy_of_add_y(22)), 42);
```

클로저를 위한 Copy와 Clone

- 러스트는 한 번만 호출될 수 있는 클로저를 알아서 파악했던 것처럼, Copy와 Clone을 구현할 수 있는 클로저와 그럴 수 없는 클로저를 파악할 수 있다.
- 반면 값을 변경하는 move가 아닌 클로저는 내부 표현 안에 변경할 수 있는 레퍼런스를 갖는다.
- 변경할 수 있는 레퍼런스는 Clone도 아니고 Copy도 아니므로 이를 사용하는 클로저 역시 둘 다 아닌 게 된다.

```
let mut x = 0;
let mut add_to_x = |n| { x += n; x };

let copy_of_add_to_x = add_to_x;
assert_eq!(add_to_x(copy_of_add_to_x(1)), 2);
```

클로저를 위한 Copy와 Clone

- 러스트는 한 번만 호출될 수 있는 클로저를 알아서 파악했던 것처럼, Copy와 Clone을 구현할 수 있는 클로저와 그럴 수 없는 클로저를 파악할 수 있다.
- move 클로저의 경우에는 규칙이 훨씬 더 단순하다.
- move 클로저가 캡처하는 게 전부 Copy면 클로저도 Copy고, 전부 Clone이면 클로저도 Clone이다.

```
let mut greeting = String::from("Hello, ");
let greet = move |name| {
    greeting.push_str(name);
    println!("{}", greeting);
};

greet.clone()("Alfred");
greet.clone()("Bruce");
```


- <https://www.rust-lang.org/>
- <https://doc.rust-lang.org/book/>
- <https://tourofrust.com/>
- The Rust Programming Language (No Starch Press, 2019)
- Programming Rust, 2nd Edition (O'Reilly, 2021)

Thank you!