Лекція 10

RefCell<T> та Interior Mutability

Мутабельність через іммутабельні посилання

RefCell • Cell • borrow • borrow_mut • runtime checks

🤖 Приклади: лічильники агентів, кеш, мок-об'єкти

Частина 1: RefCell<T> — основи

О. С. Бичков • 2025

План лекції (Частина 1)

1. Проблема: мутабельність через &self
2. Interior Mutability — концепція
3. RefCell<T> — базове використання
4. borrow() та Ref<T>
5. borrow_mut() та RefMut<T>
6. Правила borrowing у runtime
7. Panic при порушенні правил
8. try_borrow() — безпечна альтернатива
9. RefCell у структурах
10. Патерн: Rc<RefCell<T>>
11. 🤖 Лічильник операцій агента
12. 🤖 Кеш розрахунків
13. 🤖 Mock objects для тестів
14. 🤖 Логер з буфером
15. Lifetime правила для Ref/RefMut

Частина 2: Cell<T>, OnceCell, LazyCell, патерни та порівняння

```
struct Counter {
    count: u32,
}

impl Counter {
    // &mut self — потрібне мутабельне посилання
    fn increment(&mut self) {
        self.count += 1;
    }
}

// Проблема: Rc дає тільки &T, не &mut T!
use std::rc::Rc;

let counter = Rc::new(Counter { count: 0 });

// counter.increment();  // ❌ Error: cannot borrow as mutable
// Rc::get_mut(&mut counter) — тільки якщо count == 1
```

Рішення: Interior Mutability — мутабельність зсередини

```
// Як змінювати дані, що мають кілька власників?
// Як змінювати поле через &self (не &mut self)?
```

Interior Mutability — концепція

Interior Mutability — патерн, що дозволяє мутувати дані навіть коли є тільки іммутабельне посилання на них

Це "контрольоване порушення" правил borrowing:
- Звичайно: &T = read-only, &mut T = read-write
- Interior mutability: &T може давати write access

Compile-time (звичайне)

Компілятор перевіряє borrowing

✓ Гарантії на етапі компіляції
✗ Обмежена гнучкість

Runtime (interior mutability)

Перевірка під час виконання

✓ Більша гнучкість
✗ Panic при порушенні

UnsafeCell<T> — примітив; RefCell<T>, Cell<T> — безпечні обгортки

```rust
use std::cell::RefCell;

// RefCell обгортає значення, надаючи interior mutability
let data = RefCell::new(5);

// Зовні data — іммутабельний!
// Але ми можемо змінювати внутрішнє значення:

// borrow() — отримати &T (immutable reference)
{
    let r = data.borrow();  // Повертає Ref<T>
    println!("Value: {}", *r);  // 5
}  // r dropped — borrow закінчився

// borrow_mut() — отримати &mut T (mutable reference)
{
    let mut r = data.borrow_mut();  // Повертає RefMut<T>
    *r += 10;
    println!("Value: {}", *r);  // 15
}  // r dropped — borrow закінчився

println!("Final: {}", data.borrow());  // 15
```

```rust
use std::cell::{RefCell, Ref};

let data = RefCell::new(String::from("hello"));

// borrow() повертає Ref<T> — smart pointer для immutable borrow
let r1: Ref<String> = data.borrow();
let r2: Ref<String> = data.borrow();  // ✓ Багато immutable OK!

println!("{}, {}", *r1, *r2);  // hello, hello

// Ref автоматично реалізує Deref
println!("Length: {}", r1.len());  // Викликає String::len()

// Ref::map — трансформація посилання
let data = RefCell::new(vec![1, 2, 3]);
let first: Ref<i32> = Ref::map(data.borrow(), |v| &v[0]);
println!("First: {}", *first);  // 1

// Коли Ref dropped — borrow "звільняється"
drop(r1);
drop(r2);
// Тепер можна borrow_mut()
```

```rust
use std::cell::{RefCell, RefMut};

let data = RefCell::new(vec![1, 2, 3]);

// borrow_mut() повертає RefMut<T> — smart pointer для mutable borrow
let mut r: RefMut<Vec<i32>> = data.borrow_mut();

// RefMut реалізує DerefMut — можна мутувати
r.push(4);
r.push(5);
println!("{:?}", *r);  // [1, 2, 3, 4, 5]

drop(r);  // Звільняємо mutable borrow

// Тепер можна знову borrow
println!("Final: {:?}", data.borrow());

// RefMut::map — трансформація мутабельного посилання
let data = RefCell::new((1, String::from("hello")));
{
    let mut second = RefMut::map(data.borrow_mut(), |t| &mut t.1);
    second.push_str(" world");
}
println!("{:?}", data.borrow());  // (1, "hello world")
```

Правила borrowing у runtime

```
let r1 = data.borrow();
let r2 = data.borrow();
// Два immutable — OK

drop(r1); drop(r2);
let m = data.borrow_mut();
// Один mutable — OK
```

❌ Panic!

Порушення = panic у runtime, не compile error!

```
let r1 = data.borrow();
let m = data.borrow_mut();
// ❌ panic: already borrowed
```

```rust
use std::cell::RefCell;

let data = RefCell::new(5);

// Це спричинить panic!
let r1 = data.borrow();      // immutable borrow
let r2 = data.borrow_mut();  // ❌ PANIC!

// Повідомлення:
// thread 'main' panicked at 'already borrowed: BorrowMutError'

// Типова помилка — забули drop
fn problematic(data: &RefCell<Vec<i32>>) {
    let first = data.borrow()[0];  // Temporary Ref живе до ;
    // Ref dropped тут, все OK

    // Але якщо зберегти:
    let r = data.borrow();
    // r все ще живе...
    data.borrow_mut().push(4);  // ❌ PANIC!
    // Рішення: drop(r) перед borrow_mut()
}
```

```rust
use std::cell::RefCell;

let data = RefCell::new(5);

// try_borrow() повертає Result, не паніку
match data.try_borrow() {
    Ok(r) => println!("Value: {}", *r),
    Err(_) => println!("Cannot borrow!"),
}

// try_borrow_mut() — для мутабельного
let r = data.borrow();  // immutable borrow active

match data.try_borrow_mut() {
    Ok(mut m) => *m += 1,
    Err(_) => println!("Cannot mutably borrow — already borrowed!"),
}
// Виведе: "Cannot mutably borrow — already borrowed!"

// Корисно для умовної логіки
if let Ok(mut m) = data.try_borrow_mut() {
    *m += 10;
} else {
    // Fallback логіка
}
```

RefCell

```
use std::cell::RefCell;

struct Counter {
    // RefCell дозволяє мутувати через &self
    count: RefCell<u32>,
    name: String,
}

impl Counter {
    fn new(name: &str) -> Self {
        Counter {
            count: RefCell::new(0),
            name: name.to_string(),
        }
    }

    // &self, не &mut self!
    fn increment(&self) {
        *self.count.borrow_mut() += 1;
    }

    fn get(&self) -> u32 {
        *self.count.borrow()
    }
}

let counter = Counter::new("operations");
counter.increment();  // Через &self!
counter.increment();
println!("{}: {}", counter.name, counter.get());  // operations: 2
```

```rust
use std::rc::Rc;
use std::cell::RefCell;

// Rc — спільне володіння
// RefCell — interior mutability
// Разом: спільне мутабельне володіння

type SharedData = Rc<RefCell<Vec<i32>>>;

let data: SharedData = Rc::new(RefCell::new(vec![1, 2, 3]));

// Кілька власників
let owner1 = Rc::clone(&data);
let owner2 = Rc::clone(&data);

// Будь-хто може мутувати
owner1.borrow_mut().push(4);
owner2.borrow_mut().push(5);

// Всі бачать зміни
println!("owner1: {:?}", owner1.borrow());  // [1, 2, 3, 4, 5]
println!("owner2: {:?}", owner2.borrow());  // [1, 2, 3, 4, 5]
println!("data: {:?}", data.borrow());      // [1, 2, 3, 4, 5]

// ⚠️ Runtime перевірки все ще діють!
```

```rust
use std::cell::RefCell;

struct AgentStats {
    moves: RefCell<u64>,
    scans: RefCell<u64>,
    messages_sent: RefCell<u64>,
    messages_received: RefCell<u64>,
}

impl AgentStats {
    fn new() -> Self {
        AgentStats {
            moves: RefCell::new(0),
            scans: RefCell::new(0),
            messages_sent: RefCell::new(0),
            messages_received: RefCell::new(0),
        }
    }

    fn record_move(&self) { *self.moves.borrow_mut() += 1; }
    fn record_scan(&self) { *self.scans.borrow_mut() += 1; }
    fn record_message_sent(&self) { *self.messages_sent.borrow_mut() += 1; }
    fn record_message_received(&self) { *self.messages_received.borrow_mut() += 1; }

    fn summary(&self) -> String {
        format!("Moves: {}, Scans: {}, Sent: {}, Received: {}",
            self.moves.borrow(), self.scans.borrow(),
            self.messages_sent.borrow(), self.messages_received.borrow())
    }
}
```

```rust
use std::cell::RefCell;
use std::collections::HashMap;

struct PathCache {
    // Кеш — мутується при доступі (lazy computation)
    cache: RefCell<HashMap<(Position, Position), Vec<Position>>>,
}

impl PathCache {
    fn new() -> Self {
        PathCache { cache: RefCell::new(HashMap::new()) }
    }

    /// Отримати шлях з кешу або обчислити
    fn get_path(&self, from: Position, to: Position) -> Vec<Position> {
        let key = (from, to);

        // Спочатку перевіряємо кеш (immutable borrow)
        if let Some(path) = self.cache.borrow().get(&key) {
            return path.clone();
        }

        // Не знайшли — обчислюємо
        let path = self.compute_path(from, to);

        // Зберігаємо в кеш (mutable borrow)
        self.cache.borrow_mut().insert(key, path.clone());

        path
    }
```

```rust
use std::cell::RefCell;

/// Mock сенсор для тестів
struct MockSensor {
    readings: RefCell<Vec<f64>>,
    call_count: RefCell<u32>,
}

impl MockSensor {
    fn new(readings: Vec<f64>) -> Self {
        MockSensor {
            readings: RefCell::new(readings),
            call_count: RefCell::new(0),
        }
    }
}

impl Sensor for MockSensor {
    fn read(&self) -> Option<f64> {
        *self.call_count.borrow_mut() += 1;
        self.readings.borrow_mut().pop()
    }
}

#[test]
fn test_agent_uses_sensor() {
    let mock = MockSensor::new(vec![1.0, 2.0, 3.0]);
    let agent = Agent::new(Box::new(mock));

    agent.scan();
```

```rust
use std::cell::RefCell;

struct BufferedLogger {
    buffer: RefCell<Vec<String>>,
    max_size: usize,
}

impl BufferedLogger {
    fn new(max_size: usize) -> Self {
        BufferedLogger {
            buffer: RefCell::new(Vec::with_capacity(max_size)),
            max_size,
        }
    }

    /// Логування через &self
    fn log(&self, message: &str) {
        let mut buffer = self.buffer.borrow_mut();
        buffer.push(format!("[{}] {}", timestamp(), message));

        if buffer.len() >= self.max_size {
            self.flush_internal(&mut buffer);
        }
    }

    fn flush(&self) {
        self.flush_internal(&mut self.buffer.borrow_mut());
    }

    fn flush_internal(&self, buffer: &mut Vec<String>) {
```

```rust
use std::cell::RefCell;
use std::rc::Rc;

struct Agent {
    id: u32,
    // Публічні дані — звичайні
    pub position: Position,
    pub battery: u8,
    // Внутрішній стан — через RefCell
    internal_state: RefCell<InternalState>,
}

struct InternalState {
    path_cache: HashMap<Position, Vec<Position>>,
    message_queue: Vec<Message>,
    stats: AgentStats,
}

impl Agent {
    /// Обробити повідомлення — мутує internal_state через &self
    fn receive_message(&self, msg: Message) {
        self.internal_state.borrow_mut().message_queue.push(msg);
        self.internal_state.borrow_mut().stats.messages_received += 1;
    }

    /// Отримати статистику — читає internal_state
    fn get_stats(&self) -> AgentStats {
        self.internal_state.borrow().stats.clone()
    }
}
```

```rust
use std::cell::RefCell;

let data = RefCell::new(String::from("hello"));

// Ref живе поки не dropped
{
    let r = data.borrow();
    // r активний тут

    // data.borrow_mut();  // ❌ Panic! r ще живий
}  // r dropped тут

data.borrow_mut();  // ✓ OK — r вже dropped

// ВАЖЛИВО: Temporary lifetimes
let len = data.borrow().len();  // ✓ Ref dropped після ;

// Але не можна зберігати посилання на внутрішні дані!
// let s: &str = &*data.borrow();  // ❌ Ref dropped, s dangling

// Правильно — через Ref::map
use std::cell::Ref;
let first_char: Ref<char> = Ref::map(data.borrow(), |s| {
    &s.chars().next().unwrap_or(' ')
});
```

```rust
use std::cell::RefCell;

struct Outer {
    inner: RefCell<Inner>,
}

struct Inner {
    value: i32,
    data: RefCell<Vec<i32>>,
}

let outer = Outer {
    inner: RefCell::new(Inner {
        value: 42,
        data: RefCell::new(vec![1, 2, 3]),
    }),
};

// Доступ до вкладених RefCell
{
    let inner = outer.inner.borrow();
    // inner.value — доступний напряму
    println!("Value: {}", inner.value);

    // inner.data — ще один RefCell
    inner.data.borrow_mut().push(4);
}

// Обережно з вкладеними borrow!
// Тримайте borrow якомога коротше
```

```rust
use std::cell::RefCell;

// into_inner() — споживає RefCell, повертає T
let cell = RefCell::new(vec![1, 2, 3]);
let vec: Vec<i32> = cell.into_inner();
// cell більше не існує
println!("{:?}", vec);  // [1, 2, 3]

// get_mut() — отримати &mut T якщо маємо &mut RefCell<T>
let mut cell = RefCell::new(5);

// Якщо маємо exclusive access — не потрібен runtime check
let value: &mut i32 = cell.get_mut();
*value += 10;
println!("{}", cell.into_inner());  // 15

// replace() — замінити значення
let cell = RefCell::new(5);
let old = cell.replace(10);
println!("old: {}, new: {}", old, cell.borrow());  // old: 5, new: 10

// swap() — обміняти значення двох RefCell
let a = RefCell::new(1);
let b = RefCell::new(2);
a.swap(&b);
println!("a: {}, b: {}", a.borrow(), b.borrow());  // a: 2, b: 1
```

```rust
// Ref::map — корисно для доступу до полів
let data = RefCell::new((1, String::from("hello")));
let second: Ref<String> = Ref::map(data.borrow(), |t| &t.1);
println!("{}", *second);  // hello
```

Методи RefCell<T>

Метод	Опис	Panic?
borrow()	Immutable borrow → Ref<T>	Так
borrow_mut()	Mutable borrow → RefMut<T>	Так
try_borrow()	Safe immutable → Result	Ні
try_borrow_mut()	Safe mutable → Result	Ні
into_inner()	Consume → T	Ні
get_mut()	&mut RefCell → &mut T	Ні
replace(v)	Замінити значення	Так*
swap(&other)	Обміняти з іншим	Так*
take()	Замінити на Default	Так*

* Panic якщо вже borrowed

Типові помилки з RefCell

```
let s: &str = &*data.borrow();
// ❌ Ref dropped, s dangling!
```

✓ Використовуйте Ref::map

```
let s: Ref<str> = Ref::map(
    data.borrow(), |d| &**d);
```

Підсумок: Частина 1

RefCell<T> — interior mutability:
• Мутабельність через &self
• Runtime borrow checking
• Panic при порушенні правил

Основні методи:
• borrow() → Ref<T>
• borrow_mut() → RefMut<T>
• try_borrow() / try_borrow_mut() — без panic

Патерн Rc<RefCell<T>>:
• Спільне володіння + мутабельність

🤖 MAC: лічильники, кеш, mock objects, логери

→ Частина 2: Cell<T>, OnceCell, LazyCell, порівняння

Лекція 10 (продовження)

Cell<T>, OnceCell та LazyCell

Альтернативи RefCell для різних сценаріїв

Cell • OnceCell • LazyCell • Lazy • OnceLock

🤖 Приклади: прапорці стану, lazy config, singleton

Частина 2: Cell та однократна ініціалізація

О. С. Бичков • 2025

План лекції (Частина 2)

1. Cell<T> — для Copy типів
2. Cell методи: get, set, replace
3. Cell vs RefCell
4. Коли використовувати Cell
5. OnceCell — одноразова ініціалізація
6. OnceCell методи
7. LazyCell — lazy evaluation
8. Lazy від once_cell crate
9. OnceLock — thread-safe OnceCell
10. LazyLock — thread-safe Lazy
11. 🤖 Прапорці стану агента
12. 🤖 Lazy конфігурація
13. 🤖 Singleton координатора
14. 🤖 Ініціалізація ресурсів
15. Порівняння всіх Cell типів
16. Практичні рекомендації

```rust
use std::cell::Cell;

// Cell<T> — простіша альтернатива RefCell для Copy типів
// Не потребує borrow tracking!

let cell = Cell::new(5);

// get() — копіює значення (T: Copy)
let value = cell.get();  // 5

// set() — замінює значення
cell.set(10);
println!("New value: {}", cell.get());  // 10

// Ніяких Ref/RefMut — просто копіювання!
// Тому T має бути Copy

let cell_str = Cell::new("hello");  // &str is Copy
cell_str.set("world");

// ❌ String не Copy — використовуйте RefCell
// let cell_string = Cell::new(String::from("hello"));
// cell_string.get();  // Error: String is not Copy
```

```rust
use std::cell::Cell;

let cell = Cell::new(5i32);

// get() — копіює значення
let a = cell.get();  // 5
let b = cell.get();  // 5 (ще раз копія)

// set() — замінює значення, нічого не повертає
cell.set(10);

// replace() — замінює і повертає старе значення
let old = cell.replace(20);
println!("old: {}, new: {}", old, cell.get());  // old: 10, new: 20

// take() — замінює на Default і повертає старе
let cell = Cell::new(42);
let taken = cell.take();  // 42
println!("taken: {}, cell: {}", taken, cell.get());  // taken: 42, cell: 0

// swap() — обмін значеннями двох Cell
let a = Cell::new(1);
let b = Cell::new(2);
a.swap(&b);
println!("a: {}, b: {}", a.get(), b.get());  // a: 2, b: 1
```

```rust
use std::cell::Cell;

let cell = Cell::new(5);

// update() — застосувати функцію (Rust 1.50+)
cell.update(|x| x + 1);
println!("{}", cell.get());  // 6

// Еквівалент:
// let old = cell.get();
// cell.set(old + 1);

// Приклад: лічильник
struct Counter {
    count: Cell<u32>,
}

impl Counter {
    fn new() -> Self {
        Counter { count: Cell::new(0) }
    }

    fn increment(&self) {  // &self, не &mut self!
        self.count.update(|c| c + 1);
    }

    fn get(&self) -> u32 {
        self.count.get()
    }
}
```

Cell<T> vs RefCell<T>

	Cell<T>	RefCell<T>
Вимога до T	T: Copy	Будь-який T
Доступ	get()/set() — копіювання	borrow()/borrow_mut()
Runtime cost	Мінімальний	Borrow counter
Panic можливий?	Ніколи	Так (при порушенні)
Розмір overhead	0 bytes	usize (counter)
Підходить для	i32, bool, f64, &str	String, Vec, structs

Правило вибору:
- Copy типи (числа, bool, посилання) → Cell
- Все інше (String, Vec, складні структури) → RefCell

```rust
use std::cell::Cell;

// 1. Лічильники
struct Stats {
    reads: Cell<u64>,
    writes: Cell<u64>,
}

// 2. Прапорці стану
struct CacheEntry<T> {
    value: T,
    accessed: Cell<bool>,  // Чи був доступ
    access_count: Cell<u32>,
}

// 3. ID генератори
struct IdGenerator {
    next_id: Cell<u64>,
}

impl IdGenerator {
    fn next(&self) -> u64 {
        let id = self.next_id.get();
        self.next_id.set(id + 1);
        id
    }
}

// 4. Кешовані обчислення (для Copy результатів)
struct Cached {
```

```rust
use std::cell::OnceCell;

// OnceCell — ініціалізується максимум один раз
let cell: OnceCell<String> = OnceCell::new();

// Спочатку порожній
assert!(cell.get().is_none());

// set() — встановити значення (один раз)
cell.set(String::from("hello")).unwrap();

// Тепер є значення
assert_eq!(cell.get(), Some(&String::from("hello")));

// Повторний set — помилка!
assert!(cell.set(String::from("world")).is_err());
// Значення не змінилось
assert_eq!(cell.get(), Some(&String::from("hello")));

// get_or_init — отримати або ініціалізувати
let cell: OnceCell<String> = OnceCell::new();
let value = cell.get_or_init(|| {
    println!("Initializing...");
    expensive_computation()
});
// "Initializing..." виведеться тільки перший раз
```

OnceCell методи

Метод	Опис	Результат
new()	Створити порожній	OnceCell<T>
get()	Отримати &T якщо є	Option<&T>
get_mut()	Отримати &mut T якщо є	Option<&mut T>
set(value)	Встановити (раз)	Result<(), T>
get_or_init(f)	Отримати або ініціалізувати	&T
get_or_try_init(f)	Те саме з Result	Result<&T, E>
into_inner()	Consume → Option<T>	Option<T>
take()	Забрати значення	Option<T>

OnceCell гарантує: значення встановлюється максимум один раз

```rust
use std::cell::LazyCell;

// LazyCell (Rust 1.80+) — lazy ініціалізація з closure
let lazy: LazyCell<String> = LazyCell::new(|| {
    println!("Computing...");
    expensive_computation()
});

// Closure ще не викликався!
println!("Before access");

// Перший доступ — викликає closure
println!("Value: {}", *lazy);  // Computing... \n Value: ...

// Наступні доступи — просто повертає значення
println!("Value: {}", *lazy);  // Value: ... (без Computing)

// LazyCell реалізує Deref — можна використовувати як &T
fn use_lazy(lazy: &LazyCell<String>) {
    // Автоматично ініціалізується при першому доступі
    println!("Length: {}", lazy.len());
}

// Зручно для глобальних констант
static CONFIG: LazyCell<Config> = LazyCell::new(|| {
    Config::load_from_file("config.toml")
});
```

```rust
// До Rust 1.70+ OnceCell/LazyCell були в crate once_cell
// Тепер більшість є в std, але crate має додаткові можливості

use once_cell::sync::Lazy;  // Thread-safe Lazy
use once_cell::sync::OnceCell;  // Thread-safe OnceCell

// Lazy — як LazyCell, але thread-safe
static GLOBAL_CONFIG: Lazy<Config> = Lazy::new(|| {
    Config::load()
});

// OnceCell — thread-safe версія
static INSTANCE: OnceCell<Database> = OnceCell::new();

fn get_database() -> &'static Database {
    INSTANCE.get_or_init(|| {
        Database::connect("localhost:5432")
    })
}

// Різниця з std:
// once_cell::unsync::{OnceCell, Lazy} — як std::cell
// once_cell::sync::{OnceCell, Lazy} — thread-safe (як std::sync)
```

```rust
use std::sync::OnceLock;

// OnceLock (Rust 1.70+) — OnceCell для багатопотокового коду
// Можна використовувати в static!

static CONFIG: OnceLock<Config> = OnceLock::new();

fn get_config() -> &'static Config {
    CONFIG.get_or_init(|| {
        Config::load_from_env()
    })
}

// У будь-якому потоці:
let config = get_config();  // Ініціалізується один раз

// Безпечно з кількох потоків:
use std::thread;

let handles: Vec<_> = (0..10).map(|_| {
    thread::spawn(|| {
        let config = get_config();
        // Всі потоки отримають той самий Config
    })
}).collect();

for h in handles {
    h.join().unwrap();
}
```

```rust
use std::sync::LazyLock;

// LazyLock (Rust 1.80+) — thread-safe lazy initialization
static HASHMAP: LazyLock<HashMap<String, i32>> = LazyLock::new(|| {
    let mut m = HashMap::new();
    m.insert("one".to_string(), 1);
    m.insert("two".to_string(), 2);
    m
});

fn main() {
    // Ініціалізується при першому доступі
    println!("one = {}", HASHMAP.get("one").unwrap());

    // Наступні доступи — просто читання
    println!("two = {}", HASHMAP.get("two").unwrap());
}

// LazyLock реалізує Deref<Target=T>
// Тому можна використовувати як &T

// Практичний приклад: regex
use regex::Regex;

static EMAIL_REGEX: LazyLock<Regex> = LazyLock::new(|| {
    Regex::new(r"^[\w.-]+@[\w.-]+\.\w+$").unwrap()
});
```

```rust
use std::cell::Cell;

struct AgentFlags {
    is_active: Cell<bool>,
    is_moving: Cell<bool>,
    needs_recharge: Cell<bool>,
    has_target: Cell<bool>,
    emergency_mode: Cell<bool>,
}

impl AgentFlags {
    fn new() -> Self {
        AgentFlags {
            is_active: Cell::new(true),
            is_moving: Cell::new(false),
            needs_recharge: Cell::new(false),
            has_target: Cell::new(false),
            emergency_mode: Cell::new(false),
        }
    }

    fn check_battery(&self, level: u8) {
        self.needs_recharge.set(level < 20);
        if level < 5 {
            self.emergency_mode.set(true);
        }
    }

    fn can_accept_mission(&self) -> bool {
        self.is_active.get()
```

```rust
use std::sync::LazyLock;
use std::collections::HashMap;

#[derive(Debug)]
struct SwarmConfig {
    max_agents: usize,
    communication_range: f64,
    formation_distance: f64,
    behaviors: HashMap<String, BehaviorConfig>,
}

// Глобальна конфігурація — lazy load
static SWARM_CONFIG: LazyLock<SwarmConfig> = LazyLock::new(|| {
    println!("Loading swarm configuration...");

    // Завантаження з файлу, env, або defaults
    let config_str = std::fs::read_to_string("swarm.toml")
        .unwrap_or_else(|_| include_str!("default_config.toml").to_string());

    toml::from_str(&config_str).expect("Invalid config")
});

// Доступ з будь-якого місця в коді
fn create_agent() -> Agent {
    let max = SWARM_CONFIG.max_agents;  // Lazy init тут
    Agent::new(&SWARM_CONFIG)
}
```

```rust
use std::sync::OnceLock;
use std::sync::Mutex;

struct Coordinator {
    agents: Mutex<Vec<AgentId>>,
    next_id: Mutex<u32>,
}

// Singleton pattern
static COORDINATOR: OnceLock<Coordinator> = OnceLock::new();

impl Coordinator {
    fn global() -> &'static Coordinator {
        COORDINATOR.get_or_init(|| {
            println!("Initializing coordinator...");
            Coordinator {
                agents: Mutex::new(Vec::new()),
                next_id: Mutex::new(1),
            }
        })
    }

    fn register_agent(&self) -> AgentId {
        let mut next_id = self.next_id.lock().unwrap();
        let id = *next_id;
        *next_id += 1;

        self.agents.lock().unwrap().push(id);
        id
    }
```

```rust
use std::cell::OnceCell;

struct Agent {
    id: u32,
    // Ресурси ініціалізуються лише при потребі
    path_planner: OnceCell<PathPlanner>,
    sensor_fusion: OnceCell<SensorFusion>,
    communication: OnceCell<CommModule>,
}

impl Agent {
    fn new(id: u32) -> Self {
        Agent {
            id,
            path_planner: OnceCell::new(),
            sensor_fusion: OnceCell::new(),
            communication: OnceCell::new(),
        }
    }

    fn get_path_planner(&self) -> &PathPlanner {
        self.path_planner.get_or_init(|| {
            println!("Agent {}: Initializing path planner", self.id);
            PathPlanner::new()
        })
    }

    fn get_communication(&self) -> &CommModule {
        self.communication.get_or_init(|| {
            println!("Agent {}: Initializing communication", self.id);
```

```rust
use std::sync::LazyLock;
use std::sync::atomic::{AtomicU64, Ordering};

struct GlobalStats {
    total_messages: AtomicU64,
    total_moves: AtomicU64,
    total_scans: AtomicU64,
    active_agents: AtomicU64,
}

static STATS: LazyLock<GlobalStats> = LazyLock::new(|| {
    GlobalStats {
        total_messages: AtomicU64::new(0),
        total_moves: AtomicU64::new(0),
        total_scans: AtomicU64::new(0),
        active_agents: AtomicU64::new(0),
    }
});

impl GlobalStats {
    fn record_message() {
        STATS.total_messages.fetch_add(1, Ordering::Relaxed);
    }

    fn record_move() {
        STATS.total_moves.fetch_add(1, Ordering::Relaxed);
    }

    fn summary() -> String {
        format!("Messages: {}, Moves: {}, Scans: {}, Active: {}",
```

UnsafeCell — єдиний спосіб отримати &mut T з &T у safe Rust

```rust
use std::cell::UnsafeCell;

// UnsafeCell — єдиний спосіб отримати &mut T з &T у safe Rust
// Всі Cell, RefCell, Mutex побудовані на ньому

// ⚠️ Використання UnsafeCell напряму — unsafe!
let cell = UnsafeCell::new(5);

unsafe {
    // get() повертає *mut T
    let ptr: *mut i32 = cell.get();
    *ptr = 10;
}

// Чому це unsafe?
// Ви відповідаєте за дотримання правил borrowing!
// Компілятор не перевіряє

// Практично ніколи не потрібен напряму
// Використовуйте: Cell, RefCell, Mutex, RwLock

// Коли потрібен UnsafeCell:
// - Реалізація власних Cell-подібних типів
// - FFI з C кодом
// - Дуже низькорівнева оптимізація
```

Порівняння всіх Cell типів

Тип	T вимоги	Thread	Особливість
Cell<T>	T: Copy	❌	get/set копіювання
RefCell<T>	—	❌	Runtime borrow check
OnceCell<T>	—	❌	Init once
LazyCell<T>	—	❌	Lazy init з closure
OnceLock<T>	—	✓	Thread-safe OnceCell
LazyLock<T>	—	✓	Thread-safe LazyCell
Mutex<T>	—	✓	Lock-based
RwLock<T>	—	✓	Read-write lock

Single-thread: Cell/RefCell. Multi-thread: OnceLock/LazyLock/Mutex

Як вибрати правильний тип?

Питання для вибору:

1. Чи потрібна thread-safety?
   ❌ → Cell, RefCell, OnceCell, LazyCell
   ✓ → OnceLock, LazyLock, Mutex, RwLock

2. Чи T: Copy?
   ✓ → Cell (простіше, швидше)
   ❌ → RefCell

3. Чи потрібна одноразова ініціалізація?
   ✓ → OnceCell / OnceLock
   ✓ + lazy → LazyCell / LazyLock
   ❌ → Cell / RefCell

4. Чи багато читачів + один писач?
   ✓ → RwLock (multi-thread)
   ❌ → Mutex / RefCell

Типові помилки

```
// Для простих лічильників
count: RefCell<u32>  // Надлишково
count: Cell<u32>     // ✓ Краще
```

✓ Правильний вибір

```
flags: Cell<bool>      // Copy
data: RefCell<Vec<T>>  // не Copy
config: OnceLock<C>    // global init
```

Практичні рекомендації

✓ Використовуйте Cell для Copy типів (числа, bool)
✓ RefCell — тільки коли справді потрібна interior mutability
✓ Тримайте borrow якомога коротше
✓ Використовуйте try_borrow() для безпечного коду
✓ OnceCell/LazyCell для lazy initialization
✓ OnceLock/LazyLock для глобальних static

❌ Не зловживайте interior mutability
❌ Cell/RefCell НЕ для багатопотоковості
❌ Не ігноруйте можливість panic у RefCell

🤖 Для MAC:
• Cell — прапорці, лічильники агентів
• RefCell — кеш, внутрішній стан
• OnceLock — глобальна конфігурація
• LazyLock — singleton координатора

Підсумок лекції

Interior Mutability — мутабельність через &self:

Cell<T> (T: Copy):
• get/set — копіювання
• Ніколи не panic

RefCell<T>:
• borrow/borrow_mut
• Runtime borrow checking
• Може panic

OnceCell / OnceLock:
• Одноразова ініціалізація
• get_or_init pattern

LazyCell / LazyLock:
• Lazy evaluation з closure
• Автоматична ініціалізація при доступі

→ Наступна лекція: Threads — багатопотоковість

Завдання для самостійної роботи

1. Counter: Реалізуйте Counter з Cell<u32>.
   Методи: increment(&self), decrement(&self), get(&self).

2. Cache: Створіть LRU кеш з RefCell:
   - get_or_compute(key, f) — lazy computation
   - Обмеження розміру кешу

3. Mock: Реалізуйте MockDatabase для тестів:
   - RefCell для запису викликів
   - Перевірка кількості викликів у тестах

4. Singleton: Глобальний Logger з OnceLock:
   - Lazy initialization
   - Thread-safe доступ

5. MAC: AgentRegistry з LazyLock:
   - Глобальний реєстр агентів
   - Методи: register, unregister, get_all

Дякую за увагу!

RefCell • Cell • OnceCell • LazyCell

Питання?