Rust 프로그래밍 언어

성재용 2022.07.22

Advanced Rust

Closure

세부 내용

```
fn main() {
    let mut i = 0;
    let mut f = |x| {
        i += 1;
        x
    };

    i = 1; // compile error
    let a = f(1);
}
```

- Closure 에는 변수가 capture됨
- mutable reference가 캡쳐되면 closure도 mutable
  → 호출할 때 mutable reference 필요

Closure

관련 traits - FnOnce / FnMut / Fn

```rust
pub trait FnOnce<Args> {
    type Output;
    extern "rust-call" fn call(&mut self, args: Args) -> Self::Output;
}
```

```rust
pub trait FnMut<Args>: FnOnce<Args> {
    extern "rust-call" fn call(&mut self, args: Args) -> Self::Output;
}
```

```rust
pub trait Fn<Args>: FnMut<Args> {
    extern "rust-call" fn call(&self, args: Args) -> Self::Output;
}
```

Sized trait

개념

```rust
pub trait Sized { }
```

```rust
pub fn f<T: Sized>(t: T)
```

```rust
pub fn f<T: !Sized>(t: T)
```

```rust
pub fn f<T: ?Sized>(t: T)
```

- 컴파일 타임에 크기를 알 수 있는 자료형
- 기본적으로 대부분의 자료형에 자동으로 구현

dyn vs impl
비교

```rust
trait Bar {
    fn bar(&self) -> Vec<usize>;
}

impl Bar for Foo { ... }
impl Bar for Baz { ... }
```

```rust
fn f<B: Bar>(b: B) -> usize
```

```rust
fn f(b: impl Bar) -> usize
```

```rust
fn f(b: &dyn Bar) -> usize
```

```rust
fn f() -> impl Bar {
    Foo { ... }
}
```

함수형 프로그래밍 &
객체 지향 프로그래밍

객체 지향 프로그래밍

개념

- 모든 것이 객체이다.
- 특징
  - 추상화 (Abstraction): 중요한 정보만 추출
  - 캡슐화 (Encapsulation): 정보의 은닉
  - 상속 (Inheritance): 기존 클래스의 함수를 사용
  - 다형성 (Polymorphism): 한 요소에 다양한 개념 사용

객체 지향 프로그래밍

Java

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable {
    transient Object[] elementData;
    private int size;

    public ArrayList(int initialCapacity) {
        if (initialCapacity > 0) {
            this.elementData = new Object[initialCapacity];
        } else if (initialCapacity == 0) {
            this.elementData = {};
        } else {
            throw new IllegalArgumentException("Illegal Capacity: " +
                    initialCapacity);
        }
    }
}
```

함수형 프로그래밍

개념

- 람다 대수에 기반한 프로그래밍
- 모든 함수가 pure → side-effect가 없고 입력이 같으면 출력이 같음
- 특징
  - 병렬처리에 좋음
  - 새로운 시각
  - 추상화

함수형 프로그래밍

Haskell

```haskell
newtype State s a = State {runState :: s -> (s, a)}

instance Monad (State s) where
  return = pure
  m >>= k = State $ \s -> case runState m s of
    (s', x) -> runState (k x) s'
```

Concurrency & Parallelism

Concurrency vs Parallelism

비교

Concurrency

Parallelism

Concurrency vs Parallelism

비교

	Concurrency O	**Concurrency X**
Parallelism O	Web Servers	Python Multi-thread
Parallelism X	Async/Await	Simple Program

Send / Sync trait

개념

```
pub unsafe auto trait Send { }
```

```
pub unsafe auto trait Sync { }
```

- Send: 다른 쓰레드로 보낼 수 있음
- Sync: 다양한 쓰레드로 공유될 수 있음

Pin trait

개념

```
pub struct Pin<P> {
    pub pointer: P,
}
```

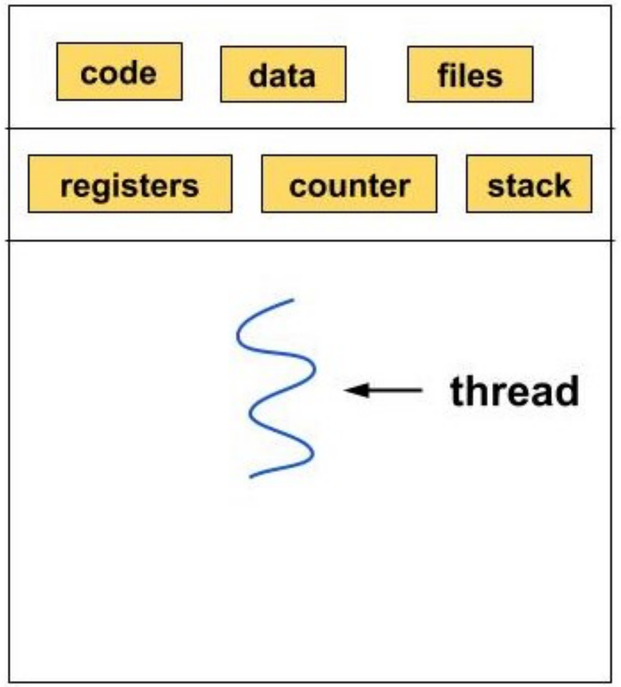
```
pub auto trait Unpin { }
```

```
pub struct PhantomPinned;

impl !Unpin for PhantomPinned {}
```

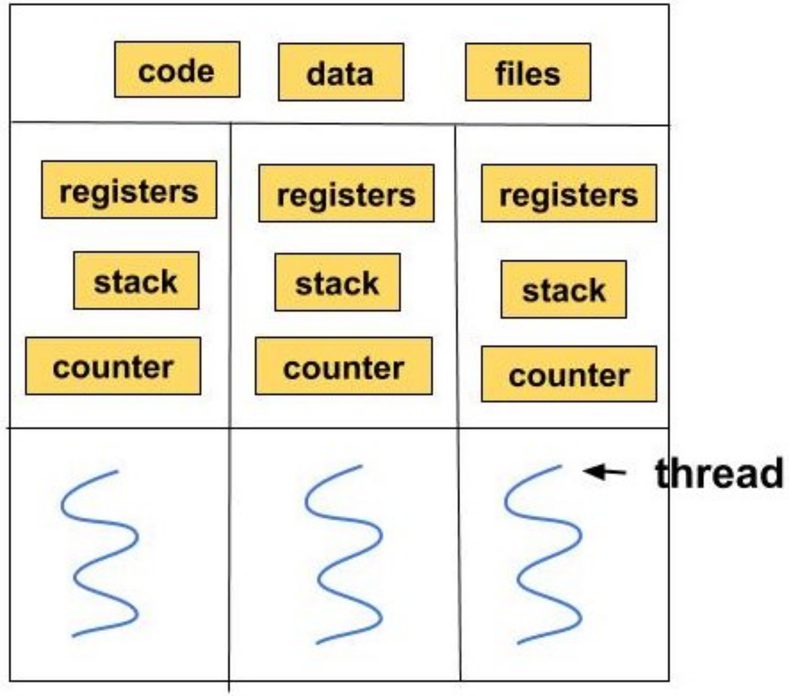
- Pin은 메모리에서 움직일 수 없게끔 고정
- Rust에서 모든 값은 기본적으로 움직일 수 있음
- Pin은 pointer를 이용해 이를 구현

Threads

개념



Single-threaded process



Multi-threaded process

Threads

예제

```
fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

Threads

종류

```
fn main() {
    let flag = Arc::new(AtomicBool::new(false));
    let flag2 = Arc::clone(&flag);

    let parked_thread = thread::spawn(move || {
        while !flag2.load(Ordering::Acquire) {
            thread::park();
        }
    });

    thread::sleep(Duration::from_millis(10));
    flag.store(true, Ordering::Release);

    parked_thread.thread().unpark();
    parked_thread.join().unwrap();
}
```

- park: 쓰레드를 주차, 다른 곳에서 unpark를 하면 다시 재개
- yield: scheduler에게 현재 쓰레드를 넘김
- sleep: 지정된 시간 동안 중지, 다른 쓰레드가 실행될 가능성 있음

Channel

개념

```rust
fn main() {
    let (tx, rx) = mpsc::channel();
    thread::spawn(move || {
        tx.send(10).unwrap();
    });
    assert_eq!(rx.recv().unwrap(), 10);
}
```

- Thread 사이의 정보를 주고 받을 수 있는 통로
- 다양한 구현이 있지만 std에 구현되어 있는 mpsc (multi-producer single-consumer)은 sender만 clone가능

Arc

개념

```
fn main() {
    let apple = Arc::new("the same apple");

    for _ in 0..10 {
        let apple = Arc::clone(&apple);

        thread::spawn(move || {
            println!("{:?}", apple);
        });
    }
}
```

- Rc의 multi-thread 버전
- 성능은 Rc보다 느림

AtomicT

개념

```rust
fn main() {
    let spinlock = Arc::new(AtomicUsize::new(1));

    let spinlock_clone = Arc::clone(&spinlock);
    let thread = thread::spawn(move || {
        spinlock_clone.store(0, Ordering::SeqCst);
    });

    while spinlock.load(Ordering::SeqCst) != 0 {
        hint::spin_loop();
    }

    if let Err(panic) = thread.join() {
        println!("Thread had an error: {panic:?}");
    }
}
```

- 값 변경을 할 때 한 쓰레드만 가능
- 더 이상 연산을 쪼갤 수 없다는 의미의 원자의 의미
- Primitive Type들에 대해 지원

Atomic Ordering

개념

```
pub struct Mutex {
    is_acquired: AtomicBool,
}

impl Mutex {
    pub fn acquire(&self) {
        while !self.is_acquired.swap(true, Ordering::AcqRel) {
            hint::spin_loop();
        }
    }

    pub fn release(&self) {
        self.is_acquired.store(false, Ordering::Release);
    }
}
```

- 컴파일러가 코드의 순서를 바꾸는 것을 막기 위함
- 종류
  - Relaxed, Release, Acquire, AcqRel, SeqCst

Mutex

개념

```
fn main() {
    let data = Arc::new(Mutex::new(0));

    let (tx, rx) = channel();
    for _ in 0..10 {
        let (data, tx) = (Arc::clone(&data), tx.clone());
        thread::spawn(move || {
            let mut data = data.lock().unwrap();
            *data += 1;
            if *data == 10 {
                tx.send(()).unwrap();
            }
        });
    }

    rx.recv().unwrap();
}
```

- 한 곳에서만 데이터를 읽고 쓸 수 있게함
- 다른 곳에서 권한을 가지고 있으면 권한이 끝날 때까지 대기
- Box의 multi-thread 버전

RwLock

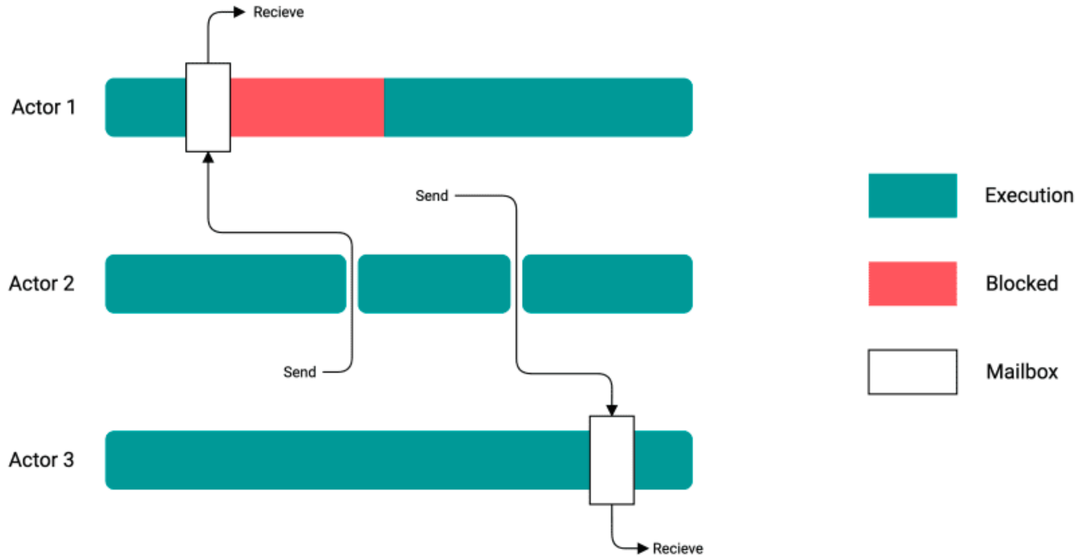
개념

```
fn main() {
    let lock = RwLock::new(5);

    {
        let r1 = lock.read().unwrap();
        let r2 = lock.read().unwrap();
        assert_eq!(*r1, 5);
        assert_eq!(*r2, 5);
    }

    {
        let mut w = lock.write().unwrap();
        *w += 1;
        assert_eq!(*w, 6);
    }
}
```

- RefCell의 multi-thread 버전
- 원하는 권한을 얻을 때까지 대기

Actor Model

개념



- 모든 것이 Actor
- Actor 끼리는 Message를 주고 받고 각 Actor은 순차적으로 Message를 처리

Actix

개념

- Rust에서 가장 유명한 Actor 모델 crate
- 이를 기반으로 만들어진 actix-web이 존재
- https://github.com/actix/actix
- https://github.com/actix/actix-web

Crossbeam

개념

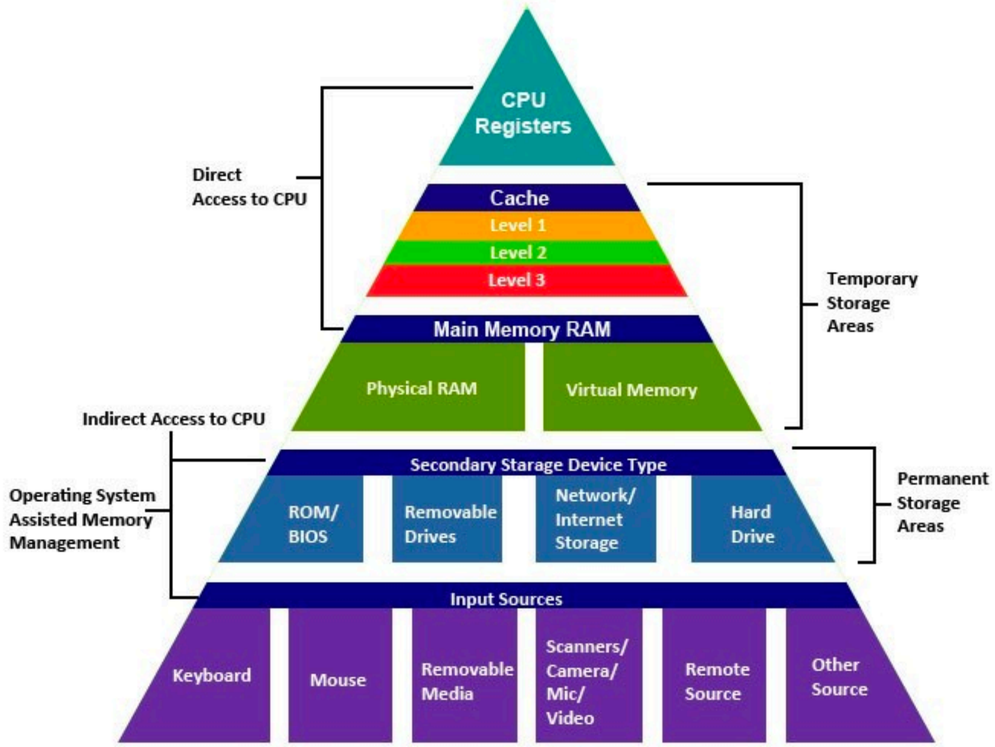
- Rust의 대표적인 concurrent 라이브러리
- no_std, MPMC, epoch GC, 등 지원
- https://github.com/crossbeam-rs/crossbeam

Rayon

개념

- Rust의 대표적인 parallelism 라이브러리
- 매우 간단하게 사용 가능 (drop-dead simple)
- https://github.com/rayon-rs/rayon

Async/Await
개념



- IO로 인해 컴퓨터는 쉬는 cycle이 존재
- 인터넷은 사람이 체감 가능
- 낭비되는 cycle을 낭비되지 않게끔

Async/Await

Future trait

```
pub trait Future {
    type Output;

    fn poll(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>
    ) -> Poll<Self::Output>;
}
```

- 비는 cycle에 다른 연산을 진행

Tokio

개념

- Rust의 대표적인 async runtime 라이브러리
- https://github.com/tokio-rs/tokio

Tokio
런타임

```rust
pub(crate) fn block_on<F: Future>(&mut self, f: F) -> Result<F::Output, ParkError> {
    use std::task::Context;
    use std::task::Poll::Ready;

    // `get_unpark()` should not return a Result
    let waker = self.get_unpark()?.into_waker();
    let mut cx = Context::from_waker(&waker);

    pin!(f);

    loop {
        if let Ready(v) = crate::coop::budget(|| f.as_mut().poll(&mut cx)) {
            return Ok(v);
        }

        self.park()?;
    }
}
```

Async File I/O

개념

```rust
use tokio::fs::File;
use tokio::io::{AsyncReadExt, AsyncWriteExt};

#[tokio::main]
async fn main() -> std::io::Result<()> {
    let mut file = File::create("foo.txt").await?;
    file.write_all(b"hello, world!").await?;

    let mut file = File::open("foo.txt").await?;

    let mut contents = vec![];
    file.read_to_end(&mut contents).await?;

    println!("len = {}", contents.len());

    Ok(())
}
```

- tokio의 fs모듈을 사용하여 async하게 읽기/쓰기 가능
- 여러 파일을 concurrently하게 읽어올 때에는 성능 이점