Typed Concurrency in Rust

Riccardo Sieve

November 12, 2025

University of Oslo

Recap on Rust

- Rust is a systems programming language focused on safety and performance.
- It achieves memory safety without a garbage collector.
- It uses a unique ownership model to manage memory.
- Rust has a strong type system that helps catch errors at compile time.
- Rust has built-in support for concurrency.
  - Threads
  - Message passing
- Ownership, borrowing, and lifetimes are key concepts for typed concurrency in Rust.
- Rust's type system includes the `Send` and `Sync` traits to ensure thread safety.
- Smart pointers like `Arc<T>` are used for shared ownership and thread-safe reference counting.
- Asynchronous programming is supported through the `Future` trait and the `async/await` syntax.

Mutability and Immutability in Rust

- By default, variables in Rust are immutable.
- This means that once a value is assigned to a variable, it cannot be changed.
- To make a variable mutable, you need to use the `mut` keyword.
- This helps prevent accidental changes to values.

Ownership in Rust

- Rust uses ownership to manage memory.
- This helps prevent data races at compile time.
- In other programming languages, memory is managed through garbage collection.
- In Rust, memory is managed through a set of ownership that enforces rules at compile time.
  - Each value in Rust has an owner.
  - A value can only have one owner at a time.
  - When the owner goes out of scope, the value is dropped.

Borrowing in Rust

- Borrowing allows you to use a value without taking ownership of it.
- You can borrow a value by creating a reference to it.
- We can also use references to borrow values.
- References can be either mutable or immutable.
- We can have either one mutable reference or multiple immutable references to a value at a time.

Reference Counted in Rust

- `Rc<T>` is a reference-counted smart pointer that enables multiple ownership of the same data.
- It keeps track of the number of references to the data, and when the reference count reaches zero, the data is automatically deallocated.
- `Rc<T>` is not thread-safe, meaning that it cannot be shared between threads.
- For thread-safe reference counting, Rust provides the `Arc<T>` type, which stands for atomic reference counting.

Atomic Reference Counted in Rust

- `Arc<T>` is a thread-safe reference-counted smart pointer that enables multiple ownership of the same data across threads.
- It uses atomic operations to manage the reference count, ensuring that it is safe to share between threads.
- Like `Rc<T>`, when the reference count reaches zero, the data is automatically deallocated.
- `Arc<T>` is often used in conjunction with `Mutex<T>` to provide thread-safe shared mutable state.
- Note that, in terms of performance, `Arc<T>` is generally slower than `Rc<T>` due to the overhead of atomic operations.

Send and Sync Traits

- The `Send` and `Sync` traits are embedded in the language rather than the standard library.
- The `Send` marker trait indicates that ownership of values of the type implementing `Send` can be transferred between threads.
- The `Sync` marker trait indicates that it is safe to reference values of the type from multiple threads.
- Most primitive types in Rust are both `Send` and `Sync`.
- Types that are not `Send` or `Sync` include raw pointers, `Rc<T>`, and `RefCell<T>`.

Send and Sync Traits (Cont.)

- Types that are composed entirely of `Send` types are automatically `Send`.
- The same applies to `Sync`.
- This means that most types in Rust are `Send` and `Sync`.
- Manually implementing these traits involves implementing unsafe Rust code.

Example of Programming with Send and Sync Traits in Rust

Rust

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0)); let mut handles = vec![];
    for _ in 0..10 {
        let counter_clone = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter_clone.lock().unwrap();
            *num += 1; // Increment the counter
        });
        handles.push(handle);
    }
    for handle in handles {
        handle.join().unwrap();
    }
    println!("Final counter value: {}", *counter.lock().unwrap());
}
```

Send and Sync Traits (Cont.)

- `Mutex` is a synchronization primitive that is `Send` and `Sync` as long as the data it protects is also `Send` and `Sync`.
- `Arc` is an atomic reference-counted smart pointer that is `Send` and `Sync` as long as the data it points to is also `Send` and `Sync`.
- The closure passed to `thread::spawn` must be `Send` because it will be executed in a different thread.
- `Arc::clone` creates a new reference to the same `Mutex`-protected data, allowing multiple threads to share ownership of the data safely.
- The `lock` method on `Mutex` returns a guard that provides mutable access to the data, ensuring that only one thread can access the data at a time.
- While no `unlock` method is needed, the lock is automatically released when the guard goes out of scope.

Scope in `std::thread`

- Rust's standard library provides the `std::thread` module for creating and managing threads.
- The function `scope` creates a scope for spawning threads that can borrow data from the parent thread.
- The function passed to `scope` will be provided a `Scope` through which it can spawn threads.
- Threads spawned within the scope can borrow non-static data.
- This is due to the fact that scope guarantees all threads to be joined at the end of the scope.
- If some threads haven't been manually joined, they will be joined automatically when the function returns.

Example of Scopes in Rust (using std::thread::scope)

Rust

```rust
use std::thread;
fn main() {
    let mut a = vec![1, 2, 3];
    let mut x = 0;

    thread::scope(|s| {
        s.spawn(|| {
            a.push(4);
        });
        s.spawn(|| {
            x += 1;
        });
    }); // All threads are joined here

    println!("a: {:?}, x: {}", a, x);
    assert_eq!(a, vec![1, 2, 3, 4]);
    assert_eq!(x, 1);
}
```

Futures and Async/Await in Rust

- A future is a value that may not be available yet, but will be at some point in the future.
- Rust provides a `Future` trait as a building block so that different async operations can be implemented with different data structures but with a common interface.
- You can apply the `async` keyword to blocks and functions to specify that they can be interrupted and resumed.
- Within an async block or async function, you can use the `await` keyword to await a future.
- The process of checking with a future to see if its value is available yet is called **polling**.

The tokio crate

- `tokio` is a crate that provides an asynchronous runtime for Rust.
- A crate[a] is a package of Rust code. It can be a library or a binary.
- It provides a way to write asynchronous code that is efficient and scalable.
- `tokio` provides a number of features, including:
  - An event loop that can handle a large number of concurrent tasks.
  - A set of asynchronous I/O primitives, such as TCP and UDP sockets, file I/O, and timers.
  - A task scheduler that can manage the execution of asynchronous tasks.
- `tokio` is widely used in the Rust community for building high-performance network applications, such as web servers and databases.
- The `tprl` (short for *The Rust Programming Language*) crate from Rust also uses `tokio` under the hood.

[a] https://doc.rust-lang.org/book/ch07-01-packages-and-crates.html

Example of an Asynchronous program in Rust (using Tokio)

Rust

```rust
#[tokio::main]
async fn main() {
    println!("Hello from the main async function!");
    say_world().await;
    // Spawn a new asynchronous task
    tokio::spawn(async {
        println!("Hello from a spawned task!");
    });
    // Introduce a delay using tokio::time::sleep
    tokio::time::sleep(tokio::time::Duration::from_secs(1)).await;
    println!("Main function finished.");
}

async fn say_world() {
    println!("World from another async function!");
}
```

The tokio crate (Cont.)

- `#[tokio::main]` is the macro that transform the `async main` function into a regular main function that sets up a Tokio runtime and starts executing the asynchronous code within it.
- `say_world().await` calls the asynchronous function `say_world` and waits for it to complete before proceeding.
- `tokio::spawn(async ... )` spawn a new asynchronous task that runs concurrently with the main function.
- `tokio::time::sleep(...).await` introduces a delay of 1 second in the execution of the main function.

Writing the Dining Philosophers Problem in Rust

- Let's now write the Dining Philosophers Problem in Rust using the `tokio` crate for asynchronous programming.
- We will use `Arc<Mutex<T>>` to manage shared state and ensure thread safety.
- For thoughts and eating, we will use asynchronous functions to simulate the actions of the philosophers.
- While the chopsticks will be represented as shared resources that philosophers will need to acquire before eating.

Defining the Philosopher Struct in Rust

Rust

```rust
use std::sync::Arc;
use tokio::sync::{Mutex, mpsc};
use tokio::time;

struct Chopstick;

#[derive(Clone)]
struct Philosopher {
    name: String,
    // left_chopstick: ...
    // right_chopstick: ...
    // thoughts: ...
    left_chopstick: Arc<Mutex<Chopstick>>,
    right_chopstick: Arc<Mutex<Chopstick>>,
    thoughts: mpsc::Sender<String>,
}
```

Implementing thinking a in Philosopher Struct in Rust

Rust

```rust
impl Philosopher {
    async fn think(&self) {
        self.thoughts
            .send(format!("Eureka! {} has a new idea!", &self.name))
            .await
            .unwrap();
    }

    async fn eat(&self) {
        ...
    }
}
```

Implementing eating in Philosopher Struct in Rust

Rust

```rust
impl Philosopher {
    async fn think(&self) {
        ...
    }

    async fn eat(&self) {
        println!("{} is hungry...", &self.name);
        // Keep trying until we have both chopsticks
        let _left = self.left_chopstick.lock().await;
        println!("{} picked up left chopstick.", &self.name);
        let _right = self.right_chopstick.lock().await;
        println!("{} picked up right chopstick.", &self.name);

        println!("{} is eating...", &self.name);
        time::sleep(time::Duration::from_millis(5)).await;
        println!("{} is done eating.", &self.name);
    }
}
```

Structuring the main Function

Rust

```rust
static PHILOSOPHERS: &[&str] = &["Socrates", "Hypatia",
        "Aristoteles", "Plato", "Hegel", "Kant"];

#[tokio::main]
async fn main() {
    // Create chopsticks
    let chopsticks: Vec<_> = (0..PHILOSOPHERS.len())
        .map(|_| Arc::new(Mutex::new(Chopstick)))
        .collect();

    // Create philosophers and their thought receivers
        ...
}
```

Creating the philosopher

Rust

```rust
#[tokio::main]
async fn main() {
        ...
    // Create philosophers and their thought receivers
    let mut receivers = Vec::new();
    let philosophers: Vec<_> = PHILOSOPHERS
        .iter().enumerate().map(|(i, &name)| {
            let left = chopsticks[i].clone();
                        let right = chopsticks[(i + 1) % chopsticks.len()].clone();
            let (thoughts_tx, thoughts_rx) = mpsc::channel(PHILOSOPHERS.len());
            receivers.push((name.to_string(), thoughts_rx));
            Philosopher {
                name: name.to_string(), left_chopstick: left,
                right_chopstick: right, thoughts: thoughts_tx,
            }
        }).collect();
        ...
}
```

Making the philosophers think and eat

Rust

```rust
#[tokio::main]
async fn main() {
        ...

    // Make them think and eat
    let mut handles = Vec::new();
    for philosopher in philosophers {
        let handle = tokio::spawn(async move {
            philosopher.think().await;
            philosopher.eat().await;
        });
        handles.push(handle);
    }

    ...
}
```

Output the thoughts and wait for all tasks to finish

Rust

```rust
#[tokio::main]
async fn main() {
        ...
    // Output their thoughts
    for (name, mut thoughts_rx) in receivers {
        let handle = tokio::spawn(async move {
            while let Some(thought) = thoughts_rx.recv().await {
                println!("{} thinks: {}", name, thought);
            }
        });
        handles.push(handle);
    }

    // Wait for all tasks to finish
    for handle in handles {
        let _ = handle.await;
    }
}
```

Summary

- Rust provides a powerful type system that helps ensure memory safety and thread safety.
- `Arc<T>` and `Mutex<T>` are essential for managing shared state in a thread-safe manner.
- The `Send` and `Sync` traits are key to ensuring that types can be safely shared between threads.
- Asynchronous programming in Rust is facilitated by the `Future` trait and the `async/await` syntax.
- The `tokio` crate provides a robust runtime for building asynchronous applications in Rust.
- In the next lecture, we will do a recap of the entire course.