Things to know about Fearless Concurrency in Rust

Rust is distinguished by features for high performance, safe and efficient concurrency support. Rust's concurrency is based on the concept of 'fearless concurrency'.

Concurrency is the feature of a program to run multiple tasks concurrently on the same CPU core. Concurrent tasks run and complete in overlapping time without being in a particular order, other than parallelism or parallelism.

Rust is distinguished by features for high performance, safe and efficient concurrency support. Rust's concurrency is based on the concept of 'fearless concurrency', where the language aims to make it easy to write secure concurrent code through ownership and a borrowed system that enforces strict rules at compile time to prevent data traces & ensure memory safety.

What is concurrency in Rust?

Rust provides a number of concurrency primitives for writing concurrent programs, including thread, message passing, mutex, atomic, async/await for asynchronous programming.

Rust's homogeneous primitives overview:

1. **Threads** : Rust provides the std::thread module in its standard library for creating and managing threads. You can create a new thread using the thread::spawn function. This function receives a wrapper containing the executable code. You can also run threads in parallel. Rust provides synchronization of primitives to coordinate their implementation. Borrow checker ensures references don't lead to unexpected activities.
2. **Message Passing** : Rust's concurrency model supports passing messages between threads. You will use channels implemented through the std::sync::mpsc module to pass messages. A channel consists of a transmitter (Sender) and a receiver (Receiver). The subject can send messages through the transmitter and

receive them through the receiver. This provides a safe and synchronous way of communication between threads.

3. **Mutex and Atomic** types : Rust provides synchronization primitives, including mutex (std::sync::Mutex) and atom types (std::sync::atomic), to ensure exclusive shared access of data. Mutex allows multiple threads to access data concurrently, while blocking data race. Atomic types provide atomic operations on shared data, such as incrementing counters, without requiring an explicit lock.
4. **Async/Await** and **Future** : Rust's async/await syntax provides a feature for writing asynchronous code that you can execute concurrently. Asynchronous programs efficiently handle tasks associated with I/O, allowing programs to perform other tasks while waiting for another I/O operation. Rust's async/await syntax is based on futures, and you can "power" them with the async-std or tokio runtime libraries.

How to use spawn thread in Rust

You will use the std:thread module to create **threads** . **The std::thread::spawn** function allows you to create a new thread that will run concurrently with the main thread or any other existing thread in the program.

Here's how you can spawn a thread with the **std::thread::spawn** function :

```
use std::thread; fn main() { // Sinh m?t thread m?
i let thread_handle = thread::spawn(|| { // Code executed in the new thread goes
??i thread ?ã sinh hoàn t?t thread_handle.join().unwrap(); // Code ?ã th?
c thi trong lu?ng chính ti?p t?c ? ?
ây println!("Hello from the main thread!"); }
```

The main function creates a new thread with the **thread::spawn** function by passing in a closure containing the code that runs in the thread. That closure prints a message indicating that the new thread is running.

The join method on **thread_handle** allows the main thread to wait for the spawned thread to finish executing. By calling join, this function ensures that the main thread waits for the spawned thread to complete before continuing.

You can spawn multiple threads and use a loop or any other Rust control construct to create multiple closures and generate threads for each.

```
use std::thread; fn main() { let num_threads = 5; let mut thread_handles = vec![]
```

For loop generates 5 threads, each of which is assigned a unique identifier, **i** to the **loop** variable . Closures write the value of I with the move keyword to avoid ownership issues. **The thread_handles** vector contains the **threads** used later for the loop join.

After spawning all threads, the main function iterates over the **thread_handles** vector , calls join on each handle and waits for all threads to execute.

Pass notifications through channels

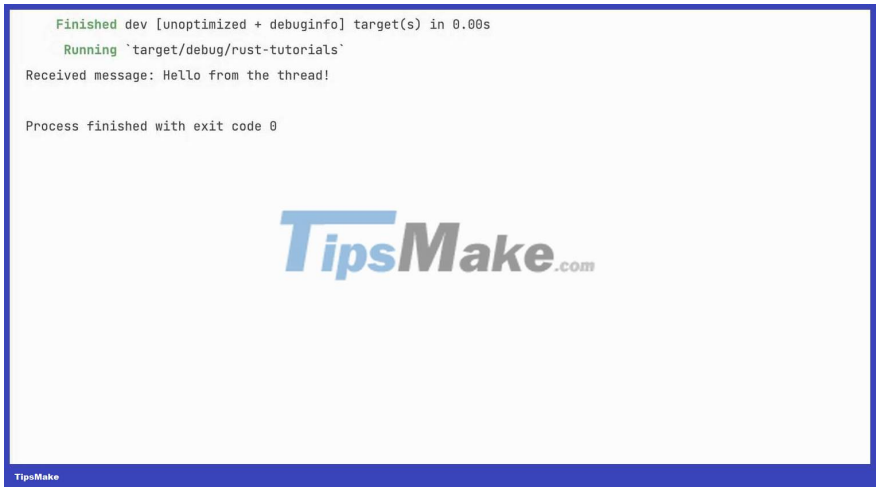
You can pass notifications across streams with their channels. Rust provides message passing in the std::sync::mpsc module. Here, mpsc stands for "multiple producer, single consumer", it allows communication between multiple threads by sending and receiving messages across channels.

Here's how you would implement communication across threads in your program:

```
use std::sync::mpsc; use std::thread; fn main() { // T?o m?
t kênh let (sender, receiver) = mpsc::channel(); // Spawn m?
t thread thread::spawn(move || { // Send a message through the channel sender.se
?
n thông báo trong thread chính let received_message = receiver.recv().unwrap(); p
```

The main function creates a channel with **mpsc::channel()** , returning a sender and a **receiver** . The sender sends the message to the receiver receiving the message. **The main** function continues spawning threads and transfers ownership of the Sender to the thread closure. Inside **the thread closure** , **the sender.send()** function sends the message over that channel.

The receiver.recv() function receives notifications by pausing execution until the thread receives the notification. **The main** function prints a message to the console after the invoice is successful.



Note that sending notifications through this channel costs the sender. If you need to send messages from multiple threads, you can clone the sender using the **sender.clone()** function .

In addition, the **mpsc** module provides other methods such as **try_recv()** , a 'non-blocking' method that attempts to receive messages, and **iter()** , which creates an iterator on received messages.

Passing messages across channels provides a safe and convenient environment for inter-thread communication, and at the same time, avoids **data races** and ensures proper synchronization.

Rust combines all of the above to provide a robust, secure, and consistent programming framework. Hope the article helps you to use Rust's Fearless Concurrency effectively.