Concurrency and Cargo in Rust

Hitesh Kumar Sharma
Instructor, Pluralsight

Agenda

- About Concurrency
- Threads
- Join Handles
- About Cargo
- Cargo Build
- Cargo Run

Concurrency

Another one of Rust's primary objectives is the safe and effective management of concurrent programming. As more computers utilize their numerous processors, concurrent programming where several parts of a program run concurrently and parallel programming where various parts of a program run simultaneously become more crucial. Rust aims to transform the historically challenging and error-prone nature of programming in these situations.

In this topic we are going to discuss following:-

- How to construct threads so that several sections of code can execute simultaneously?
- Concurrency with message-passing, in which channels transfer messages between threads.
- Multiple threads have access to the same piece of shared state data.
- Rust's concurrency guarantees are expanded to include both user-defined types and types from the standard library with the Sync and Send traits.

Threads

We can run multiple threads of code at once. The code of an executed program is run in a process in modern operating systems, and the operating system maintains numerous processes concurrently. You can also have independent components running concurrently within your program. Threads are the functions that control these separate components.

There are various ways programming languages implement threads, and many operating systems offer an API that languages can use to create new threads. One operating system thread is used for every one language thread in a program thanks to the 1:1 style of thread implementation used by the Rust standard library. Other threading models with different trade-offs from the 1:1 model are implemented by some crates.

Create Thread

To start a new thread, use the thread::spawn function. The parameter for the spawn function is a closure. The closure specifies the code that the thread should run. The example we have given in next slide prints text from a new thread as well as from the main thread.

The thread::sleep method makes a thread pause for a brief period of time so that another thread can continue to execute. Although it is likely that the threads will alternate, this is not a given and will rely on how the operating system schedules the threads.

Create Thread (Example)

The following example prints some text from a main thread and other text from a new thread.

```
use std::thread;
use std::time::Duration;
fn main() {
    thread::spawn(|| {
      for n in 1..4 {
         println!("Number {} is from PS thread!", n);
         thread::sleep(Duration::from_millis(2));
      }
   });
  for n in 1..4 {
      println!("Number {} is from main thread!", n);
      thread::sleep(Duration::from_millis(2));
   }
}
```

```
Output:

Number 1 is from main thread!
Number 1 is from PS thread!
Number 2 is from main thread!
Number 2 is from PS thread!
Number 3 is from main thread!
Number 3 is from PS thread!
```

The new thread will be stopped when the main thread ends. The output from this program might be a little different every time.

Join Handles

Because there is no certainty regarding the sequence in which threads run, we cannot ensure that the created thread will ever get to run because the code in thread example (in last slide) not only ends it prematurely most of the time owing to the main thread terminating!

By storing the return value of thread::spawn in a variable, we may resolve the issue of the spawned thread not running or terminating prematurely. JoinHandle is the return type for thread::spawn. A JoinHandle is an owned item that will wait for its thread to finish when the join method is called on it. In order to ensure that the thread we generated in last example ends before the main quits, in coming slide we will demonstrate how to use the JoinHandle of that thread and call join.

Create Thread (Example)

The following example prints some text from a main thread and other text from a new thread.

```
use std::thread;
use std::time::Duration;
fn main() {
   let handle = thread::spawn(|| {
      for i in 1..10 {
         println!("Number {} is from PS thread!", i);
         thread::sleep(Duration::from_millis(1));
      }
   });
   for i in 1..5 {
      println!("Number {} is from main thread!", i);
      thread::sleep(Duration::from_millis(1));
   }
   handle.join().unwrap();
}
```

```
Output:

Number 1 is from main thread!
Number 1 is from PS thread!
Number 2 is from PS thread
Number 2 is from main thread!
Number 2 is from PS thread!
Number 3 is from PS thread!
Number 3 is from main thread!
```

The main thread and spawned thread continue switching. The main thread waits for spawned thread to complete because of the call to the **join()** method.

Cargo

Rust's build system and package management are called cargo. Because Cargo does several activities for you, including creating your code and obtaining and constructing the libraries your code depends on, the majority of Rust aceans utilize this tool to manage their Rust projects.

If you used the official installers, Rust and Cargo are already installed. If you installed Rust in another way, type the following command in your terminal to see if Cargo was installed:

```
$ cargo --version
```

You have it if you see the version number! If you encounter a problem, such as a command not found error then you need to install Cargo individually.

Create Rust Project with Cargo

In the following section, we have shown how to build a new project with Cargo and examine how it differs from our initial "Hello, world!" project. Execute the following commands on any operating system:

```
$ cargo new PS_cargo
$ cd PS_cargo
```

- The first command creates a new directory and project called PS_cargo. We have named our project PS_cargo, and Cargo creates its files in a directory of the same name.
- You need to go into the PS_cargo directory and list the files. You will see that Cargo has generated two files and one directory for us: a Cargo.toml file and a src directory with a main.rs file inside.

Build and Run Cargo Project

In this section we are going to learn hoe we build and run the any project. In this section we are taking the same project named as “Hello, world!” program with Cargo! from your *PS_cargo* directory, build your project by entering the following command:

```
$ cargo build
   Compiling PS_cargo v0.1.0 (file:///projects/PS_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 2.85 secs
```

Instead of producing an executable file in your current directory, this command produces one in target/debug/PS_cargo (or targetdebugPS_cargo.exe on Windows). Cargo places the binary in the debug directory because the default build is a debug build. The command can be used to launch the executable is given in next slide.

Build and Run Cargo Project (contd..)

```
$ ./target/debug/hello_cargo # or .\target\debug\PS_cargo.exe on
Windows

Hello, world!
```

If all goes well, Hello, world! should print to the terminal. The first time you run cargo build, Cargo will also create a new file at the root level called `Cargo.lock`. The precise versions of the dependencies in your project are tracked in this file.

Build and Run Cargo Project

We recently created a project with cargo build and executed it with./target/debug/PS_cargo, however cargo run allows us to compile the code and execute the resulting executable in a single command:

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/MY_cargo`
Hello, world!
```

The majority of developers use cargo run because it is more convenient than having to remember to execute cargo build and then use the complete path to the binary.

End of Module