Introduction to async/await

Kiva

<2020-07-31 Fri>

Contents

1 Abstract

One of the fundamental features of programming languages is multi-tasking, which is the ability to run part of your code concurrently. Some languages provided programmers with APIs that can manipulate threads (such as C, C++, Java) while some languages provided another way of writing multi-tasking programs, which is usually referred to as coroutine (such as Golang's goroutine and JavaScript's async/await).

In this paper, we will dive into the latter way of being concurrent. We take a detailed look how async/await works from the compiler's aspect, including the design of ***Future*** trait, the state machine and its transformation.

2 Introduction to multitasking

There are two forms of multitasking:

2.1 Preemptive multitasking

Preemptive multitasking uses operating system functionality to switch threads at arbitrary points in time by forcibly pausing them.

The idea behind preemptive multitasking is that the OS controls when to switch tasks. For that, it utilizes the fact that it regains control of the CPU on each interrupt. This makes it possible to switch tasks whenever new input is available to the system. For example, it would be possible to switch tasks when the mouse is moved or a network packet arrives. The OS can also determine the exact time that a task is allowed to run by configuring a hardware timer to send an interrupt after that time.

2.2 Cooperative multitasking

Cooperative multitasking requires tasks to regularly give up control of the CPU so that other tasks can make progress.

Instead of focibly pausing running tasks at arbitrary points in time, cooperative multitasking lets each task run until it voluntarily gives up control of the CPU. This allows tasks to pause themselves at convenient points in time, for example when it needs to wait for an IO operation anyway.

Cooperative multitasking is often used at the language level, for example in form of ***coroutines*** or ***async/await***. The idea is that either the programmer or the compiler inserts ***yield*** operations into the program, which is the command to give up control of the CPU and allow other tasks to run. For example, a yield could be inserted after each iteration of a complex loop.

It is very common to combine cooperative multitasking with asynchronous operations. Instead of waiting until an operation is finished and preventing other tasks to run in this time, asynchronous operations return a *not ready* status if the operation is not finished yet. In this case, the waiting task can execute a yield operation to let other tasks run.

2.3 Comparison between them

Let's take a look at preemptive multitasking. Since tasks are interrupted at arbitrary points in time, they might be in the middle of some calculations. In order to be able to resume them later, the OS must backup the whole state of the task, including its call stack and the values of all CPU registers. This time-consuming process is called a context-swtich.

Now let's move to cooperative multitasking. Since tasks define their pause points themselves, they don't need the OS to save their state. Instead they can save exactly the state they need for continuation before they pause themselves, which is often results in better performance. For example, a task that just finished a complex computation might only need to backup the final result of the computation since it does not need to intermediate results anymore. Language-supported implementations of cooperative tasks are often even able to backup the required parts of the call stack before pausing, which saved a lot of time.

3 Introduction to Future

We simply define the ***Future*** as a value that might not be available yet but in any near future. For example, an integer that is computed by another task or a file that is downloaded from the network. Instead of waiting until the value is available, futures make it possible to continue execution until the value is **needed** instead of **produced**.

A ***Future*** should have an operation to query whether the value is **produced**. We first define the *Future trait* (persudo code), which looks like this:

```
trait Future<T> {
    fn poll(&mut self) -> Poll<T>;
}
```

The generic type parameter T specifies the type of the asynchronous value. The *poll* is the name of the operation that returnes the status of the asynchronous value. It returns a *Poll* enum, which looks like this:

```
enum Poll<T> {
    Ready(T),
    Pending,
}
```

When the value is already available (e.g. the file was fully read from disk), it is returned wrapped in the *Ready* variant. Otherwise, the *Pending* variant is returned, which says the value is not yet available.

3.1 Waiting on futures

We define another operation on futures are *wait*, which we can simply define it as follows:

```
trait Future<T> {
    fn wait(self) -> T {
        loop {
            match self.poll() {
                Poll::Ready(value) => return value;
                Poll::Pending => (), // do nothing
            }
        }
    }
}
```

Here we wait for the future by calling *poll* over and over again in a loop. Although this solution works, it is very **inefficient*** because we keep the CPU busy until the value becomes available.

And this solution is opposite to the reason why we use futures. We want to continue the execution when the value is not available, but in this case, we kept waiting until the value becomes available.

3.2 Future Combinators

An alternative way of waiting futures is to use future combinators. Future combinators are methods like *map* that allow chaining and combining futures together. Instead of waiting, thus we don't need a *wait* operation.

If you are familiar with Haskell, you mihgt be thinking that a *Future* is also a *Functor*. We define the *fmap* of *Future* to be the following:

```
impl<T, U> Future<T> {
    fn then(self, mapper: Fn(T) -> U) -> Future<U>;
}
```

I omitted the body of the *fmap* function as we only care about the function signature: $fmap : Future_T \to (T \to U) \to Future_U$

3.2.1 Advantages

The big advantage of future combinators is that they keep the operations asynchronous. In combination with asynchronous IO interfaces, this approach can lead to very high performance. The fact that future combinators are implmented as normal structs with trait implementations, which allows the compiler to excessively optimize them.

3.2.2 Drawbacks

While the future combinators make it possible to write very efficient code, they can be difficult to use in some complex situations. For example, JavaScript have callback-hell which looks like this:

```
fn test(response: Response) {
    async_read_file("hello.txt")
        .and_then(|content| response.send(content))
        .and_then(|result| {
            if result.is_ok() {
                println!("Send file successful");
            } else {
                println!("Send file failed");
            }
        });
}
```

Here we read the and use the $and\backslash_{then}$ combinator to chain a second future based on the file content which sends the content to client.

As you can imagine, this way of writing code is not the trivial way we often use. And this can quickly lead to very complex code for larger projects. It gets especially compilcated when many $and\backslash_{then}$ are applied.

3.3 The async/await Pattern

In order to write simplified code just like normal synchronous code, we introduce the async/await pattern, which looks like this:

```
async fn get() -> i32 {
    0
}
```

The *async* keyword here tells the compiler to turn this synchronous function into a asynchronous one which returns a future instead of the immediate value. The compiler may produce the following code after desugar:

```rust
fn get() -> Future<i32> {
    future::ready(0)
}
```

Here as the *get* function does not have complicated computations, it just returned a *ready* future which means the value is already available.

To wait on a future, we use the *.await* opeartor, which looks like this:

```rust
async fn test(response: Response) {
    let content = async_read_file("hello.txt").await;
    let result = response.send(content).await;
    if result.is_ok() {
        println!("Send file successful");
    } else {
        println!("Send file failed");
    }
}
```

This is a direct translation of the example above. As you can see, we write this code just like any other synchronous code. But this is actually asynchronous!

4 Future State Machine

Now let's move to the implementation part of futures. We will explore the basic idea of future and its corresponding state machine.

The compiler will convert the async function body into a state machine, each *.await* operation representing a new state. For example, the example above may be translated into the following code:

```rust
enum StateMachine {
    StartState(Resonse),
    WaitingOnFileState(Response, Future<String>),
    WaitingOnSendState(Future<Result>),
    EndState,
}
```

The state machine generated by the compiler may looks like the above code.

We have four states, one start state, one end state, and two waiting states (because we have two *.await* operations).

And the compiler will implementation *Future* trait for this state machine:

```
impl Future<()> for StateMachine {
    fn poll(&mut self) {
        loop {
            match self {
                StartState(response) => {},
                WaitingOnFileState(response, read_file_future) => {}
                WaitingOnSendState(send_future) => {}
                EndState => {}
            }
        }
    }
}
```

Because the async function body has the result type *()*, the compiler implements *StateMachine* the *Future<()>* trait, which means the result value is a *()*.

In the *poll* body, which is the core transformation of our state machine, we use a match statement on the current state inside a loop. The idea is that we switch to the next state as long as possible and use an explicit *return Poll::Pending* when we can't continue.

The state machine is in the *StartState* when it is right at the beginning of the function. In this case, we execute all the code from the body of the example function until the first *.await*. To handle the *.await* operation, we change the state of the self state machine to *WaitingOnFileState*, which includes save the needed *response* and the future returned by *async_read_file*.

```
// ...
match self {
    StartState(response) => {
        let read_file_future = async_read_file("hello.txt");
        *self = WaitingOnFileState(response, read_file_future);
    }
    // ...
}
// ...
```

Since the *match self {...}* statement is executed in a loop, the execution jumps to the *WaitingOnFileState* arm next. In this match arm we first call the *poll* function of the ***read_file_future***. If it is not ready, we exit the loop and return *Poll::Pending*. Since self stays in the *WaitingOnFileState* in this case, the next *poll* call on the state machine will enter the same match arm and retry polling the ***read_file_future***.

```
// ...
match self {
    // ...
    WaitingOnFileState(response, read_file_future) => match read_file_future.poll() {
        Poll::Ready(content) => {
            let send_future = response.send(content);
            ,*self = WaitingOnSendState(send_future);
        },
        Poll::Pending => return Poll::Pending,
    }
    // ...
}
// ...
```

When the ***read_file_future*** is ready, we create a new future by calling *response.send(...)*. And switch the current state to *WaitingOnSendState* which waits on it. But this time, we don't need *response* any more as it is not used in the later code.

```
// ...
match self {
    WaitingOnSendState(send_future) => match send_future.poll() {
        Poll::Ready(result) => {
            if result.is_ok() {
                println!("Send file successful");
            } else {
                println!("Send file failed");
            }
            *self = EndState;
            return Poll::Ready(());
        },
        Poll::Pending => return Poll::Pending,
    },
    // ...
}
// ...
```

When *send_future* is ready, we switch current state to *EndState* and return a *Poll::Ready*. Futures should not be polled again after returning *Poll::Ready*, so we just panic if we are already in the *EndState*.

```
// ...
match self {
    // ...
    EndState => panic!("poll was called() after future completed"),
    // ...
}
// ...
```

The compiler may generate different code. In fact, the compiler generated code is more complicated than we showed above because the compiler need to handle every lifetimes, borrow checkers and types. But we only showed the core part of the idea.

The last piece of the problem is to convert the *test* function itself. Remember, we define the *test* function like this:

```
async fn test(response: Response);
```

So we only need to return the *StartState* of our state machine (actually, we returned a future so other async functions can perform *.await* on it).

```
fn test(response: Response) -> Future<()> {
    StateMachine::StartState(response)
}
```

5 Reference

// TODO