第二十一讲：异步编程 (Asynchronous Programming)

第 2 节：Futures in Rust

向勇、陈渝

清华大学计算机系

xyong,yuchen@tsinghua.edu.cn

2020 年 5 月 6 日

提纲

1 第 2 节：Futures in Rust

Ref:

- Futures Explained in 200 Lines of Rust, by Carl Fredrik Samson
- Writing an OS in Rust - Async/Await, by Philipp Oppermann
- Zero-cost futures in Rust, by Aaron Turon
- Rust’s Journey to Async/Await, by Steve Klabnik
- Asynchronous Programming in Rust

零成本异步 I/O

Future 的设计目标

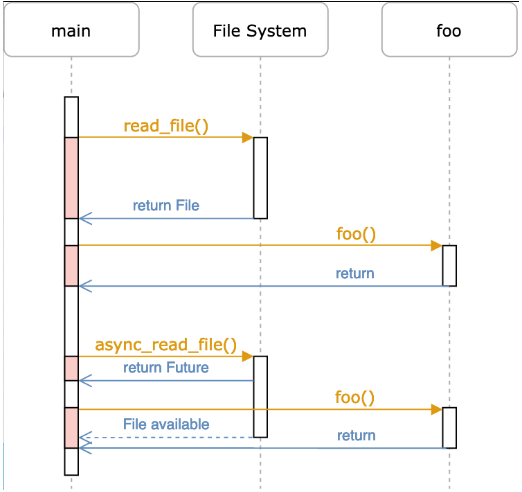
- 调用 I/O 时，系统调用会立即返回，然后你可以继续进行其他工作
- I/O 完成时，回到调用该异步 I/O 暂停的那个任务线上
- 一种通过对异步 I/O 的良好抽象形成的基于库的解决方案
  - 它不是语言的一部分，也不是每个程序附带的运行时的一部分，只是可选的并按需使用的库

零成本抽象

- 不给不使用该功能的用户增加成本
- 使用该功能时，它的速度不会比不使用它的速度慢

Concept of Future

A future is a representation of some operation which will complete in the future.



Concept of Future

A future is a representation of some operation which will complete in the future.

```
use futures::executor::block_on;
async fn foo() -> u8 { 5 }
async fn hello_world() {
    let x: u8 = foo().await;
    println!("{} hello, world!",x);
}
fn main() {
    let future = hello_world(); // Nothing is printed
    block_on(future); // print something
}
```

Rust future example

```
//rust code
fn main() {
  let _ = example(100);
}
async fn example(min_len: usize) -> String {
  let content = async_read_file("foo.txt").await;
  if content.len() < min_len {
     content + &async_read_file("bar.txt").await
  } else {
    content
  }
}
fn async_read_file(name: &str) -> impl Future<Output = String> {
  future::ready(String::from(name))
}
```

Async Lifetimes

- ‘async fn‘s which take references or other non-‘’static‘ arguments return a ‘Future‘ which is bounded by the lifetime of the arguments.

```
//rust code
// This function:
async fn foo(x: &u8) -> u8 { *x }

// Is equivalent to this function:
fn foo_expanded<'a>(x: &'a u8) -> impl Future<Output = u8> + 'a {
  async move { *x }
}
```

Async Lifetimes

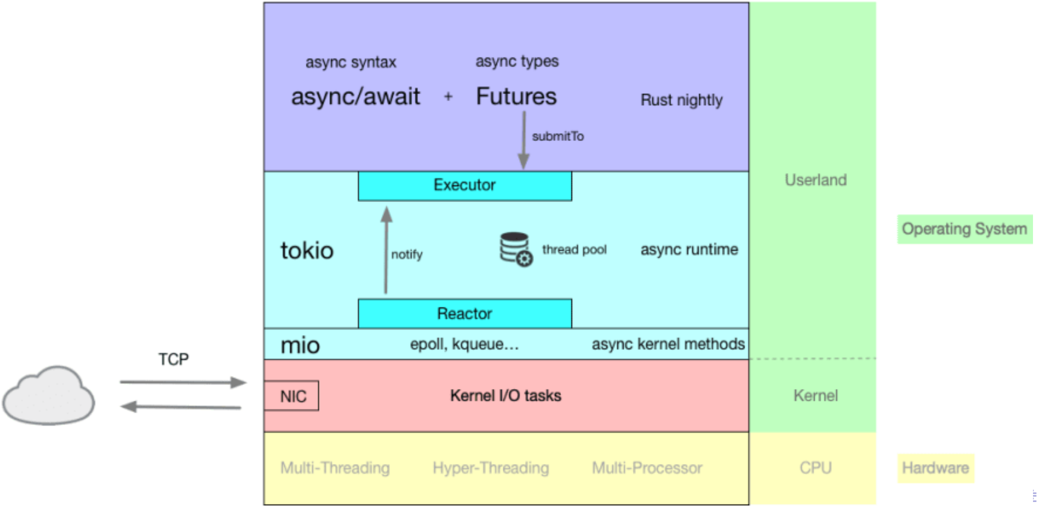
- By moving the argument into the ‘async‘ block, we extend its lifetime to match that of the ‘Future‘ returned

```
//rust code
fn bad() -> impl Future<Output = u8> {
  let x = 5;
  borrow_x(&x) // ERROR: `x` does not live long enough
}

fn good() -> impl Future<Output = u8> {
  async {
    let x = 5;
    borrow_x(&x).await
  }
}
```

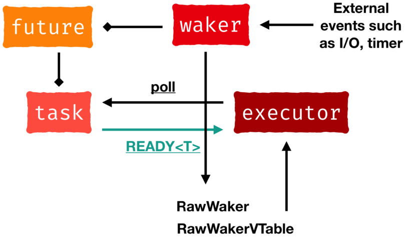
Concept of Future

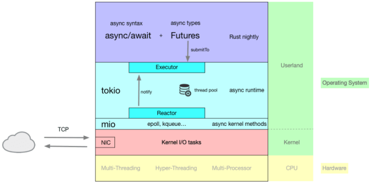
Async Architecture of rust



Concept of Future

Async Architecture of rust

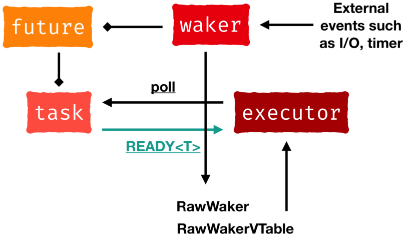


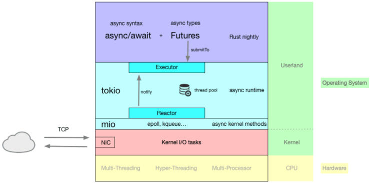


- Executor: A Future is polled which result in the task progressing
  - Until a point where it can no longer make progress

Concept of Future

Async Architecture of rust

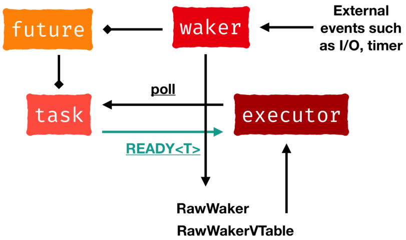


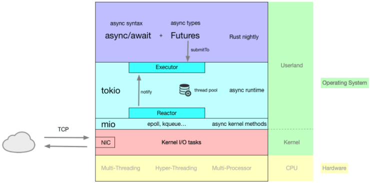


- Executor: A Future is polled which result in the task progressing
  - Until a point where it can no longer make progress
- Reactor: Register an event source that a Future is waiting for
  - Makes sure that it will wake the Future when that event is ready

Concept of Future

Async Architecture of rust





- Executor: A Future is polled which result in the task progressing
  - Until a point where it can no longer make progress
- Reactor: Register an event source that a Future is waiting for
  - Makes sure that it will wake the Future when that event is ready
- Waker: The event happens and the Future is woken up
  - Wake up to the executor which polled the Future
  - Schedule the future to be polled again and make further progress

Leaf futures & Non-leaf-futures

- Leaf future
  - Runtimes create *leaf futures* which represents a resource like a socket
  - Operations on these resources will be non-blocking and return a future which we call a leaf future

```
// stream is a leaf-future
let mut stream = tokio::net::TcpStream::connect("127.0.0.1:3000");
```

Leaf futures & Non-leaf-futures

- Non-leaf-future
  - The bulk of an async program will consist of non-leaf-futures, which are a kind of pause-able computation
  - Non-leaf-futures represents a set of operations (leaf or no-leaf)

```
// Non-leaf-future
async fn example(min_len: usize) -> String {
  let content = async_read_file("foo.txt").await;
  if content.len() < min_len {
    content + &async_read_file("bar.txt").await
  } else {
    content
  }
}
```

Runtimes

- Languages like C#, JavaScript, Java, GO and many others comes with a standard runtime for handling async
- Rust uses a special library for handling async
- The two most popular library for Futures:
  - async-std
  - Tokio

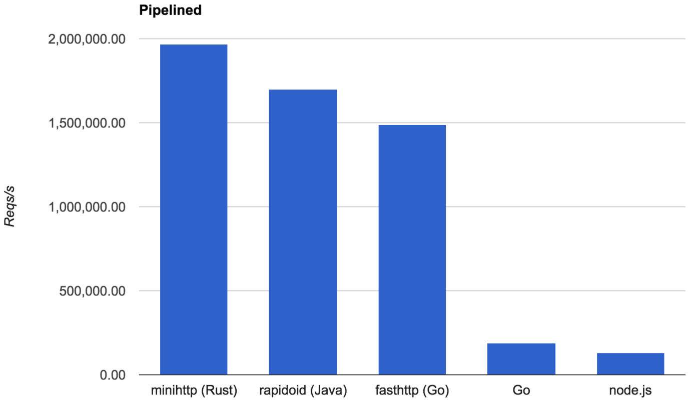
Runtimes

What Rust’s standard library takes care of

- A common interface representing an operation which will be completed in the future through the ‘Future‘ trait.
- An ergonomic way of creating tasks which can be suspended and resumed through the ‘async‘ and ‘await‘ keywords.
- A defined interface wake up a suspended task through the ‘Waker‘ type.

Zero-cost futures in Rust

Here are the results, in number of “Hello world!”s served per second on an 8 core Linux machine.


Pipelined
Reqs/s
minihttp (Rust)
rapidoid (Java)
fasthttp (Go)
Go
node.js