

第七课: Rust异步编程

Future、async/await、内部实现

苏林



回顾上一次公开课的内容

Rust编译过程 - - - 编译器前端(Rustc) 编译器后端(LLVM) Rust宏解释器

过程宏 — 属性式(proc_macro_attribute)、派生式(proc_macro_derive)、函数式(proc_macro)

过去的公开课我们讲了很多关于Rust异步编程

今天公开课内容

- 1、Future以及处理Future的async/await
- 2、async/await关键字背后的原理

为什么需要Future

```
1 use anyhow::Result;
2 use serde_yaml::Value;
3 use std::fs;
4
5 fn main() -> Result<()> {
6     // 读取 Cargo.toml, IO 操作 1
7     let content1 = fs::read_to_string("./Cargo.toml")?;
8     // 读取 Cargo.lock, IO 操作 2
9     let content2 = fs::read_to_string("./Cargo.lock")?;
10
11     // 计算
12     let yaml1 = toml2yaml(&content1)?;
13     let yaml2 = toml2yaml(&content2)?;
14
15     // 写入 /tmp/Cargo.yaml, IO 操作 3
16     fs::write("/tmp/Cargo.yaml", &yaml1)?;
17     fs::write("/tmp/Cargo.lock", &yaml2)?;
18
19     // 打印
20     println!("{}", yaml1);
21     println!("{}", yaml2);
22
23     Ok(())
24 }
25
26 fn toml2yaml(content: &str) -> Result<String> {
27     let value: Value = toml::from_str(&content)?;
28     Ok(serde_yaml::to_string(&value)?)
29 }
```

为什么需要Future

```
1 use anyhow::{anyhow, Result};
2 use serde_yaml::Value;
3 use std::{
4     fs,
5     thread::{self, JoinHandle},
6 };
7
8 /// 封装一下 JoinHandle, 这样可以提供额外方法
9 struct MyJoinHandle<T> { join_handle: JoinHandle<Result<T>> };
10
11 impl<T> MyJoinHandle<T> {
12     /// 等待 thread 执行完 (类似 await)
13     pub fn thread_await(self) -> Result<T> {
14         self.join().map_err(|_| anyhow!("failed"))?
15     }
16 }
17
18 fn main() -> Result<()> {
19     // 读取 cargo.toml, IO 操作 1
20     let t1 = thread::spawn(|| {
21         // 读取 Cargo.lock, IO 操作 2
22         let t2 = thread::spawn(|| {
23             // 计算
24             let content1 = t1.thread_await()?;
25             let content2 = t2.thread_await()?;
26
27             // 计算
28             let yaml1 = toml::from_str(&content1)?;
29             let yaml2 = toml::from_str(&content2)?;
30
31             // 写入 /tmp/Cargo.yaml, IO 操作 3
32             let t3 = thread::spawn(|| {
33                 // 写入 /tmp/Cargo.lock, IO 操作 4
34                 let t4 = thread::spawn(|| {
35                     let yaml1 = t3.thread_await()?;
36                     let yaml2 = t4.thread_await()?;
37
38                     fs::write("/tmp/Cargo.yaml", &yaml1)?;
39                     fs::write("/tmp/Cargo.lock", &yaml2)?;
40
41                     // 打印
42                     println!("{}", yaml1);
43                     println!("{}", yaml2);
44
45                     Ok(())
46                 })
47             })
48         })
49     });
50
51     let handle = thread::spawn(move || {
52         let s = fs::read_to_string(filename)?;
53         Ok(s), anyhow::Error(s)
54     });
55     MyJoinHandle(handle)
56 }
57
58 fn thread_write(filename: &static str, content: String) -> MyJoinHandle<String> {
59     let handle = thread::spawn(move || {
60         fs::write(filename, &content)?;
61         Ok(s), anyhow::Error(s)
62     });
63     MyJoinHandle(handle)
64 }
65
66 fn toml::from_str(content: &str) -> Result<String> {
67     let value: Value = toml::from_str(&content)?;
68     Ok(serde_yaml::to_string(&value)?)
69 }
```

为什么需要Future

```
1 use anyhow::Result;
2 use serde_yaml::Value;
3 use tokio::{fs, try_join};
4
5 #[tokio::main]
6 - async fn main() -> Result<()> {
7     // 读取 Cargo.toml, IO 操作 1
8     let f1 = fs::read_to_string("./Cargo.toml");
9     // 读取 Cargo.lock, IO 操作 2
10    let f2 = fs::read_to_string("./Cargo.lock");
11    let (content1, content2) = try_join!(f1, f2)?;
12
13    // 计算
14    let yaml1 = toml2yaml(&content1)?;
15    let yaml2 = toml2yaml(&content2)?;
16
17    // 写入 /tmp/Cargo.yml, IO 操作 3
18    let f3 = fs::write("/tmp/Cargo.yml", &yaml1);
19    // 写入 /tmp/Cargo.lock, IO 操作 4
20    let f4 = fs::write("/tmp/Cargo.lock", &yaml2);
21    try_join!(f3, f4)?;
22
23    // 打印
24    println!("{}", yaml1);
25    println!("{}", yaml2);
26
27    ok(())
28 }
29
30 - fn toml2yaml(content: &str) -> Result<String> {
31     let value: Value = toml::from_str(&content)?;
32     ok(serde_yaml::to_string(&value)?)
33 }
34
```

深入了解Future/async/await

```
6   #[tokio::main]
7   ▶ async fn main() -> Result<()> {
8       // 读取 Cargo.toml, IO 操作 1
9       let f1 : impl Future<Output=Result<...>> = fs :read_to_string( path: "./Cargo.toml");
10      // 读取 Cargo.lock, IO 操作 2
11      let f2 : impl Future<Output=Result<...>> = fs :read_to_string( path: "./Cargo.lock");
12      let (content1 : String , content2 : String ) = try_join!(f1, f2)?;
13  }
```

Trait std::future::Future

```
pub trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}
```


深入了解Future/async/await

```
1 use futures::executor::block_on;
2 use std::future::Future;
3
4 #[tokio::main]
5 async fn main() {
6     let name1 = "Tyr".to_string();
7     let name2 = "Lindsey".to_string();
8
9     say_hello1(&name1).await;
10    say_hello2(&name2).await;
11
12    // Future 除了可以用 await 来执行外, 还可以用 executor 执行
13    block_on(say_hello1(&name1));
14    block_on(say_hello2(&name2));
15 }
16
17 async fn say_hello1(name: &str) -> usize {
18     println!("Hello {}", name);
19     42
20 }
21
22 // async fn 关键字相当于一个返回 impl Future<Output> 的语法糖
23 fn say_hello2<'fut>(name: &'fut str) -> impl Future<Output = usize> + 'fut {
24     async move {
25         println!("Hello {}", name);
26         42
27     }
28 }
```

什么是executor?

把 executor 大致想象成一个 Future 的调度器

很多在语言层面支持协程的编程语言，比如 Golang / Erlang，都自带一个用户态的调度器。Rust 虽然也提供 Future 这样的协程，但它在语言层面并不提供 executor，把要不要使用 executor 和使用什么样的 executor 的自主权交给了开发者。好处是，当我的代码中不需要使用协程时，不需要引入任何运行时；而需要使用协程时，可以在生态系统中选择最合适应用的 executor。

常见的 executor 有

futures 库自带的很简单的 executor；

tokio 提供的 executor，当使用 `#[tokio::main]` 时，就引入了 tokio 的 executor；

async-std 提供的 executor，和 tokio 类似；

smol 提供的 async-executor，主要提供了 `block_on`。

什么是reactor?

Reactor pattern 包含三部分:

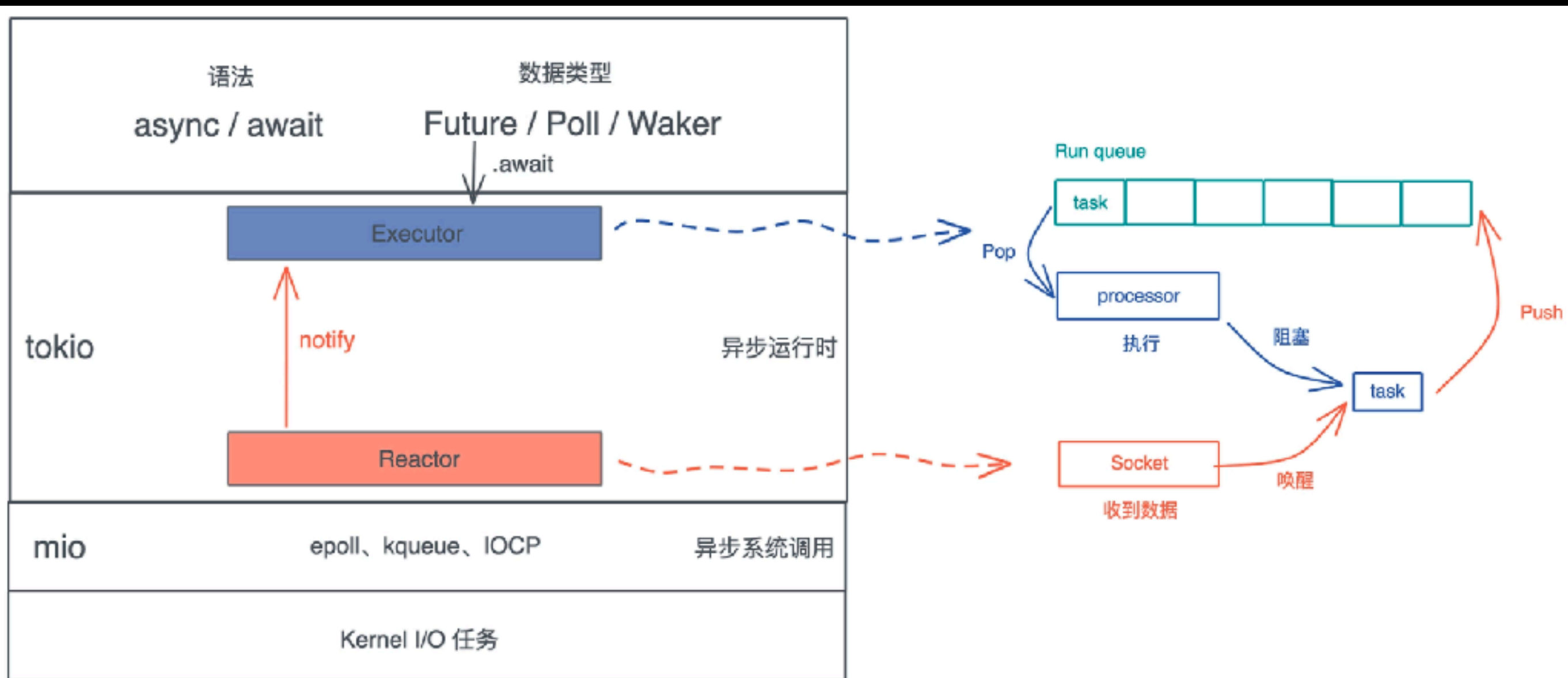
task, 待处理的任务。Future

executor, 一个调度器。ready queue, wait queue;

reactor, 维护事件队列。

executor 会调度执行待处理的任务, 当任务无法继续进行却又没有完成时, 它会挂起任务, 并设置好合适的唤醒条件。之后, 如果 reactor 得到了满足条件的事件, 它会唤醒之前挂起的任务, 然后 executor 就有机会继续执行这个任务。这样一直循环下去, 直到任务执行完毕。

Future做异步处理



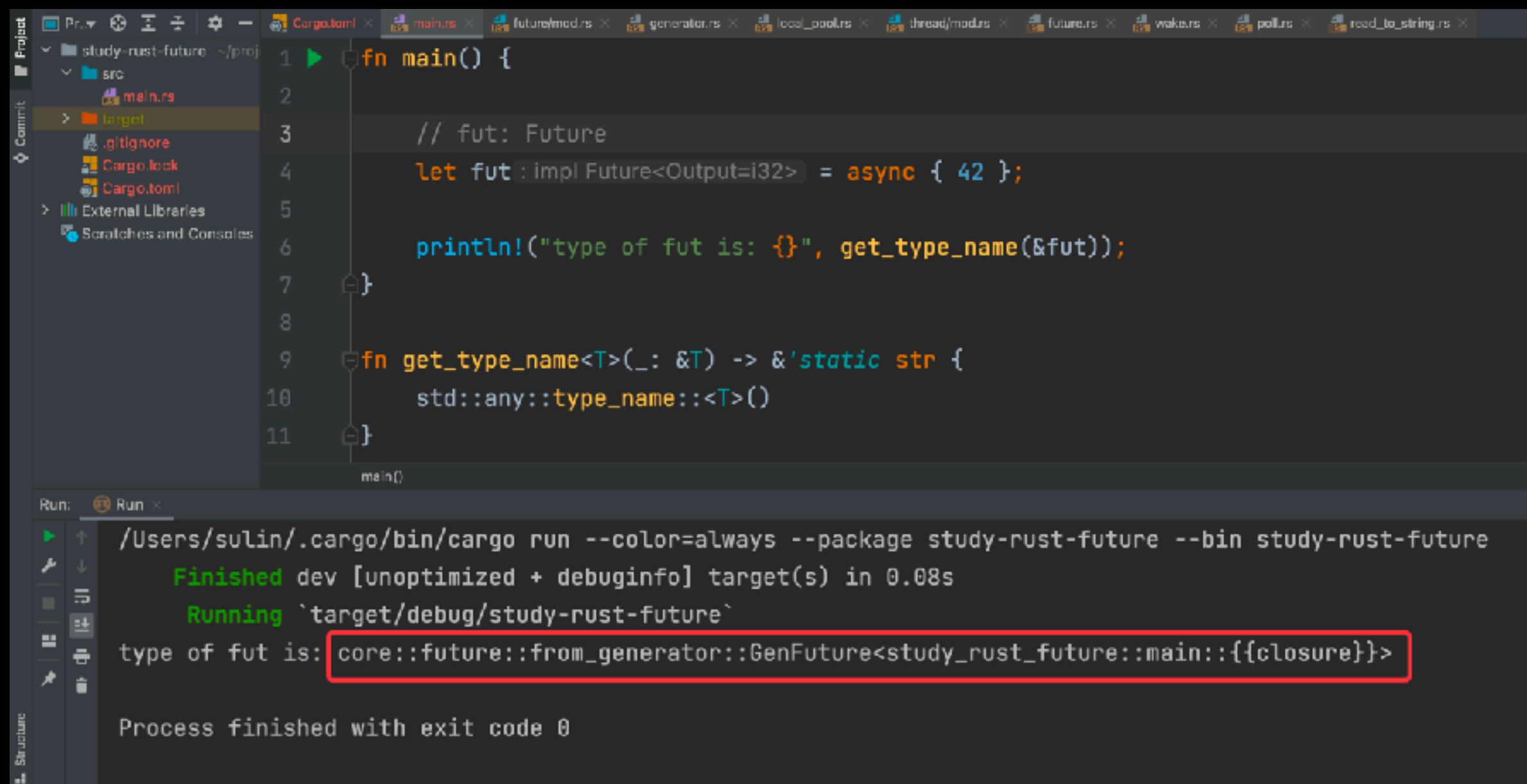
async/await这两个关键字背后的原理

1、围绕着 Future 这个接口, 探讨一些原理

Trait std::future::Future

```
pub trait Future {  
    type Output;  
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;  
}
```

async生成了什么?



The screenshot shows an IDE with a project named 'study-rust-future'. The code in 'main.rs' defines a `main` function that creates a `Future` using `async` and prints its type. A helper function `get_type_name` is also defined. The output window shows the command used to run the program, the completion of the build, and the printed output: `type of fut is: core::future::from_generator::GenFuture<study_rust_future::main::{{closure}}>`, which is highlighted with a red box. The process finished with exit code 0.

```
1 fn main() {
2
3     // fut: Future
4     let fut : impl Future<Output=i32> = async { 42 };
5
6     println!("type of fut is: {}", get_type_name(&fut));
7 }
8
9 fn get_type_name<T>(_: &T) -> &'static str {
10     std::any::type_name::<T>()
11 }
```

Run: Run x

```
/Users/sulin/.cargo/bin/cargo run --color=always --package study-rust-future --bin study-rust-future
Finished dev [unoptimized + debuginfo] target(s) in 0.08s
Running `target/debug/study-rust-future`
type of fut is: core::future::from_generator::GenFuture<study_rust_future::main::{{closure}}>
Process finished with exit code 0
```

QA环节

加群一起交流Rust & Databend

