Tokio入门之运行时介绍

Rust异步运行时, 认准Tokio就对了!

苏林

自我介绍

- 前折800互联网研发团队负责人, 10余年一线研发经验
- 目前是多点Dmall技术Leader
- 具有多年的软件开发经验, 熟悉Ruby、Java、Rust等开发语言
- 同时也参与过Rust中文社区日报维护工作.

分享内容

- 回顾上次公开课的内容
- Rust异步编程模型的实现, 和其它语言的对比
- async/await 底层的实现
- 谈谈对Rust异步运行时的认识
- Tokio介绍以及Tokio里的Executor、Reactor、Future
- 使用 Tokio 实现一个简单的服务端程序

期望公开课达到的目的

让大家对Rust异步编程生态有一个非常具体的理解并且对Rust异步编程生态中涉及到的各组件的作用有非常清晰的蓝图.

对Rust异步编程模型生态有一个非常清楚的认识.

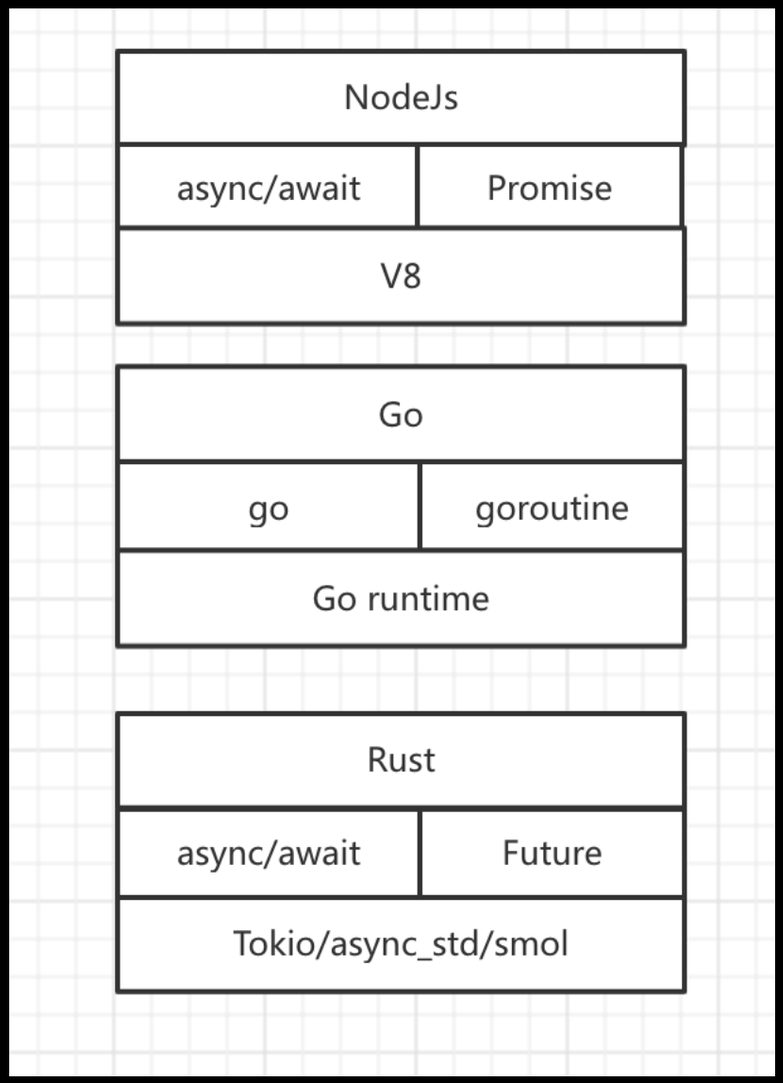
回顾上次公开课的内容

Rust异步编程模型的实现, 和其它语言的对比

NodeJS

Go

Rust



async/await 底层的实现

```rust
#![feature(generators, generator_trait)]
use std::pin::Pin;
use std::ops::Generator;

fn main() {
    let mut gen : impl Generator<...> = || {
        println!("执行第1次yield");
        yield 1;
        println!("执行第2次yield");
        yield 2;
        println!("执行第3次yield");
        yield 3;
        println!("执行第4次yield");
        return 4;
    };
    let c = Pin::new(&mut gen).resume(());
    println!("{:?}", c);
    let c = Pin::new(&mut gen).resume(());
    println!("{:?}", c);
    let c = Pin::new(&mut gen).resume(());
    println!("{:?}", c);
    let c = Pin::new(&mut gen).resume(());
    println!("{:?}", c);
}
```

async/await 底层的实现

```rust
#![feature(generators, generator_trait)]
use std::ops::{Generator, GeneratorState};
use std::pin::Pin;
enum __Gen {
    // (0) 初始状态
    Start,
    // (1) resume 方法执行之后
    State1(State1),
    // (2) resume 方法执行之后
    State2(State2),
    // (3) resume 方法执行之后
    State3(State3),
    // (4) resume 方法执行之后
    Done
}
struct State1 {x: u64}
struct State2 {x: u64}
struct State3 {x: u64}
impl Generator for __Gen {
    type Yield = u64;
    type Return = u64;

    fn resume(self: Pin<&mut Self>, _: ()) -> GeneratorState<u64, u64> {
        let mut_ref = self.get_mut();
        match std::mem::replace(mut_ref, __Gen::Done) {
            __Gen::Start =>
                {
                *mut_ref = __Gen::State1(State1 { x: 1 });
                GeneratorState::Yielded(1)
            }
            __Gen::State1(State1 { x: 1 }) => {
                *mut_ref = __Gen::State2(State2 { x: 2 });
```

async/await 底层的实现

```rust
#![feature(generators, generator_trait)]
use std::pin::Pin;
use std::ops::{Generator, GeneratorState};

fn main() {
    let mut gen : impl Generator<...> = || {
        println!("执行第1次yield");
        yield 1;
        println!("执行第2次yield");
        yield 2;
        println!("执行第3次yield");
        yield 3;
        println!("执行第4次yield");
        return 4;
    };

    if let GeneratorState::Yielded(v) = Pin::new(&mut gen).resume(()) {
        println!("resume {:?} : Pending", v);
    }

    if let GeneratorState::Yielded(v) = Pin::new(&mut gen).resume(()) {
        println!("resume {:?} : Pending", v);
    }

    if let GeneratorState::Yielded(v) = Pin::new(&mut gen).resume(()) {
        println!("resume {:?} : Pending", v);
    }

    if let GeneratorState::Complete(v) = Pin::new(&mut gen).resume(()) {
        println!("resume {:?} : Ready", v);
    }
}
```

谈谈对Rust异步运行时生态的认识

Tokio -> 生产级

Async-std

Smol -> small

Glommio

Bastion

Tokio介绍

Tokio 是一个 Rust 异步运行时库，底层基于 epoll/kqueue 这样的跨平台多路复用 IO 以及 event loop，目前正在支持 io_uring.

它的 scheduler 和 Erlang/Go 实现的 N:M threads 类似，线程会执行 Task，可以充分利用多核.

Task 是 Rust 基于 Future 抽象出的一种绿色线程，因为不需要预先分配多余的栈内存，可以创建大量 task，很适合做 IO 密集型应用.

Tokio介绍

Tokio 是一个 Rust 异步运行时库，底层基于 epoll/kqueue 这样的跨平台多路复用 IO 以及 event loop，目前正在支持 io_uring.

它的 scheduler 和 Erlang/Go 实现的 N:M threads 类似，线程会执行 Task，可以充分利用多核.

Task 是 Rust 基于 Future 抽象出的一种绿色线程，因为不需要预先分配多余的栈内存，可以创建大量 task，很适合做 IO 密集型应用.

官方示例代码

```rust
use std::error::Error;
use tokio::net::TcpListener;
use tokio::io::{AsyncReadExt, AsyncWriteExt};

#[tokio::main]
async fn main() -> Result<(), Box<dyn Error>> {
    let listener : TcpListener = TcpListener::bind( addr: "127.0.0.1:8080").await?;  // listen
    loop {
        let (mut socket : TcpStream , _) = listener.accept().await?; // async wait for incoming tcp socket

        tokio::spawn( future: async move {                    // create async task and let Tokio process it
            let mut buf : Vec<u8> = vec![0; 1024];

            loop {                                            // read and write data back until EOF
                let n : usize = match socket.read(&mut buf).await {
                    // socket closed
                    Ok(n : usize ) if n == 0 => return,
                    Ok(n : usize ) => n,
                    Err(e : Error ) => {
                        eprintln!("failed to read from socket; err = {:?}", e);
                        return;
                    }
                };

                // Write the data back
                if let Err(e : Error ) = socket.write_all( src: &buf[0..n]).await {
                    eprintln!("failed to write to socket; err = {:?}", e);
                    return;
                }// async wait socket is ready to write and write data
            }
        });
    }
}
```

Tokio库 代码目录和结构

.github	ci: fail if valgrind complains (#4066)	12 days ago
benches	bench: update spawn benchmarks (#3927)	2 months ago
bin	chore: script updating versions in links to docs.rs (#1249)	2 years ago
examples	net: add read/try_read etc methods to NamedPipeServer (#3899)	2 months ago
stress-test	deps: update rand to 0.8, loom to 0.4 (#3307)	9 months ago
tests-build	macros: fix wrong error messages (#4067)	11 days ago
tests-integration	runtime: fix memory leak/growth when creating many runtimes (#3564)	6 months ago
tokio-macros	macros: fix wrong error messages (#4067)	11 days ago
tokio-stream	stream: add `From<Receiver<T>>` impl for receiver streams (#4080)	7 days ago
tokio-test	fs: document performance considerations (#3920)	2 months ago
tokio-util	chore: prepare tokio-util v0.6.8 (#4087)	2 days ago
tokio	time: document paused time details better (#4061)	4 days ago
.cirrus.yml	util: remove path deps (#3413)	8 months ago
.clippy.toml	macros: suppress clippy::default_numeric_fallback lint in generated c...	3 months ago
.gitignore	Rename to tokio-core, add in futures-io	5 years ago
CODE_OF_CONDUCT.md	chore: improve discoverability of CoC (#2180)	2 years ago
CONTRIBUTING.md	chore: mention fix for building docs in contributing guide (#3618)	6 months ago
Cargo.toml	Move stream items into `tokio-stream` (#3277)	9 months ago
LICENSE	chore: update years in all licenses (#3665)	5 months ago
README.md	chore: prepare Tokio v1.11.0 (#4083)	5 days ago

Tokio运行时

Tokio 是一个 Rust 异步运行时库，底层基于 epoll/kqueue 这样的跨平台多路复用 IO 以及 event loop，目前正在支持 io_uring.

它的 scheduler 和 Erlang/Go 实现的 N:M threads 类似，线程会执行 Task，可以充分利用多核.

Task 是 Rust 基于 Future 抽象出的一种绿色线程，因为不需要预先分配多余的栈内存，可以创建大量 task，很适合做 IO 密集型应用.

QA环节

加群一起交流Rust & Datafuse

Datafuse

Rust语言中文社区

知数堂
zhishutang.com