Tokio
Futures
Rust

—— wayslog

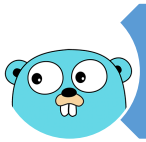

指点江山
口若悬河
云程序员来了
首先我没有写过
但我看教程很简单
你写不出来就是太菜

线程模型 & TLDR

开销分析

epoll 与 Rust

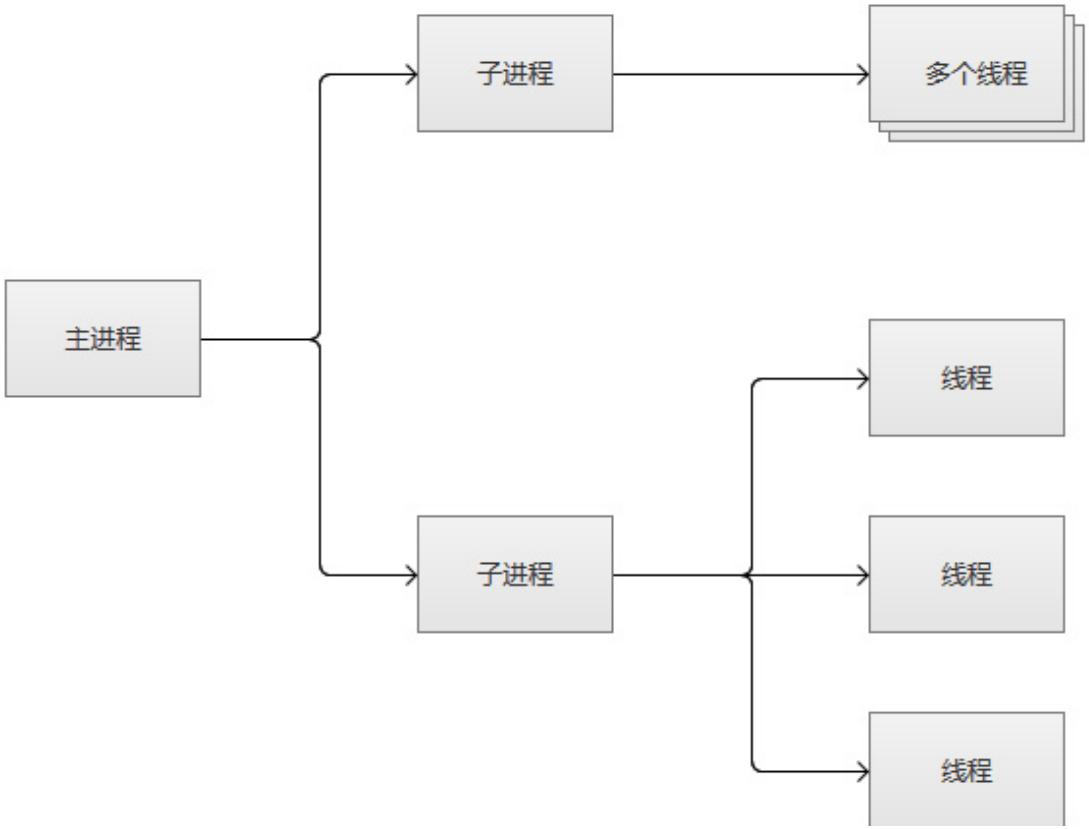
关于 tokio 与 futures



聊聊 coroutine

TLDR: 优化你的系统调用去吧。

线程模型

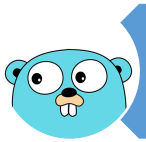


线程模型 & TLDR

开销分析

epoll 与 Rust

关于 tokio 与 futures



聊聊 coroutine

线程模型的问题探究

- 内核陷入开销太大 ...?
- 上下文切换开销太大 ...?

内核陷入开销探究

```
#[bench]
fn bench_getpid_syscall(b: &mut test::Bencher) {
    b.iter(|| {
        unsafe {
            libc::syscall(libc::SYS_getpid);
        }
    });
}

```

```
test bench_getpid_syscall ... bench:          57 ns/iter (+/- 0)
```

上下文切换开销(1)

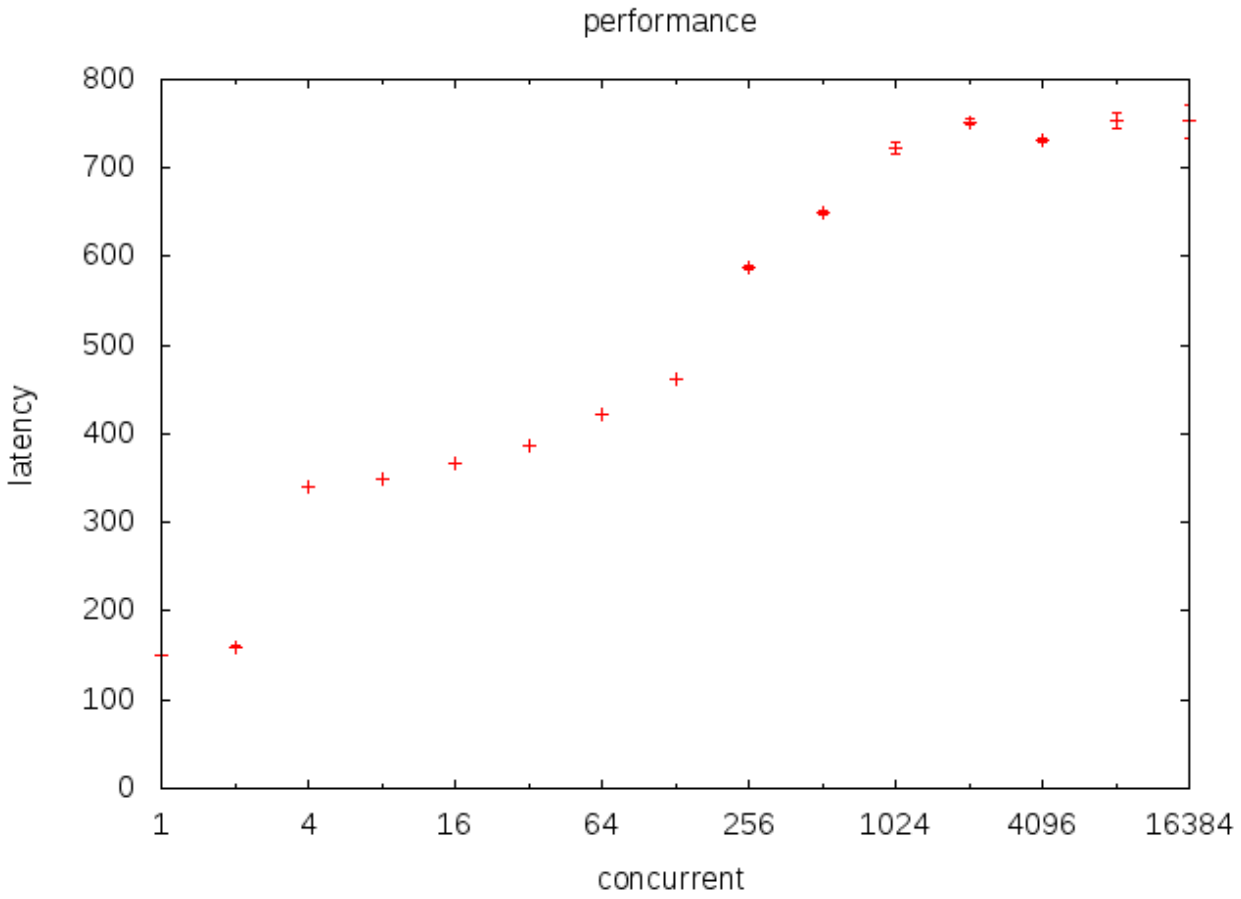
```rust
const MAX: i32 = 1_000_000;

fn bench_yield(n: i32) {
    let times = MAX / n;
    let ths: Vec<_> = (0..n)
        .into_iter()
        .map(|_| thread::spawn(move || for _ in 0..times {
            unsafe {
                libc::syscall(libc::SYS_sched_yield);
            }
        }))
        .collect();
    for th in ths {
        th.join().unwrap();
    }
}

#[bench]
fn bench_yield_x1_thread(b: &mut test::Bencher) {
    b.iter(|| bench_yield(1))
}
```

上下文切换开销(2)

```
1 test bench_yield_x1_thread   ... bench: 139,557,796 ns/iter (+/- 840,008)
```

也就是说，每次开销在139ns。
并且因为有57ns的陷入开销，
所以纯粹的上下文切换只有82ns

调度开销探究



yied 时间随着
线程的增加而增加

Context Switch Scheduler

- 进程调度会触发 context switch
- 进程三态变化才会触发 调度
- 如何让用户程序三态变化：运行态到阻塞态
- blocking syscall 是什么

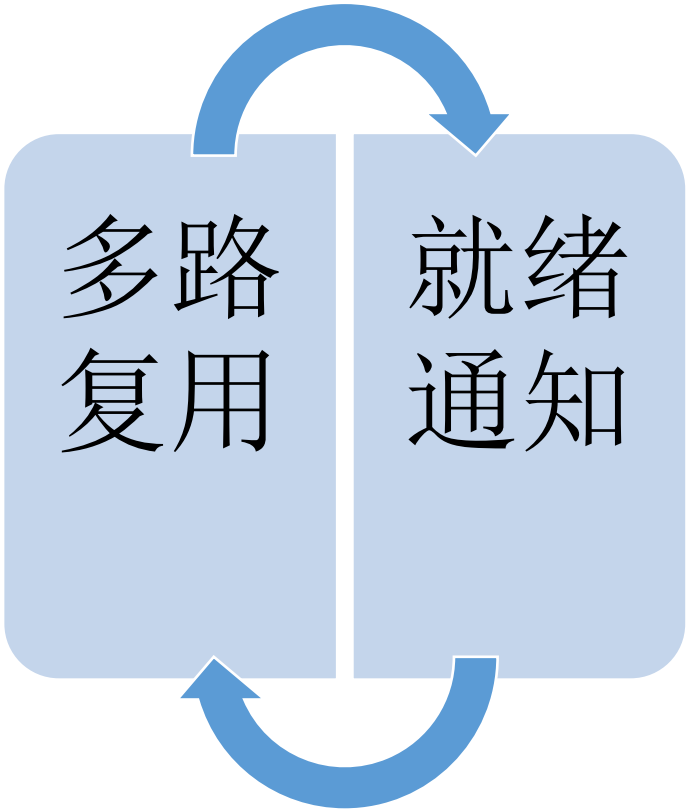
Context Switch Scheduler

- syscall 不能完成的时候会阻塞
- context switch 会造成CPU空转
- 其开销只与活跃线程数有关

Context Switch Scheduler

- 核心：减少context switch的数量
- 手段：就绪通知
- 手段：用户态缓冲区
- 手段：writev/readv
- 手段：减小锁临界区
- 手段：使用atomic(合理内存序)

如何解决

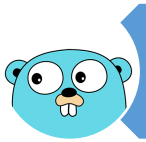


线程模型 & TLDR

开销分析

epoll 与 Rust

关于 tokio 与 futures



聊聊 coroutine

epoll 的典型模型

- 1. 1 event loop + N worker thread
- 2. N (event loop + worker) + reuseport
- 3. coroutine

Rust 与 epoll

- mio: epoll/kqueue/iocp抽象
- futures: 任务抽象库
- tokio: mio+futures 生的娃

Future

Future 表示一种状态，将来可能完成的状态。定义如下：

```
pub trait Future {
    type Item;
    type Error;
    fn poll(&mut self) -> Result<Async<Self::Item>, Self::Error>;
}

```

Future的状态转移

- 1. Err(Self::Error) 表示发生了错误，并且返回对应的错误类型。
- 2. Ok(Async::NotReady) 表示本次计算还没有完成，需要等会儿才能完成。需要由调用方重试。
- 3. Ok(Async::Ready(T)) 表示本次计算完成了，并且返回了对应的结果。

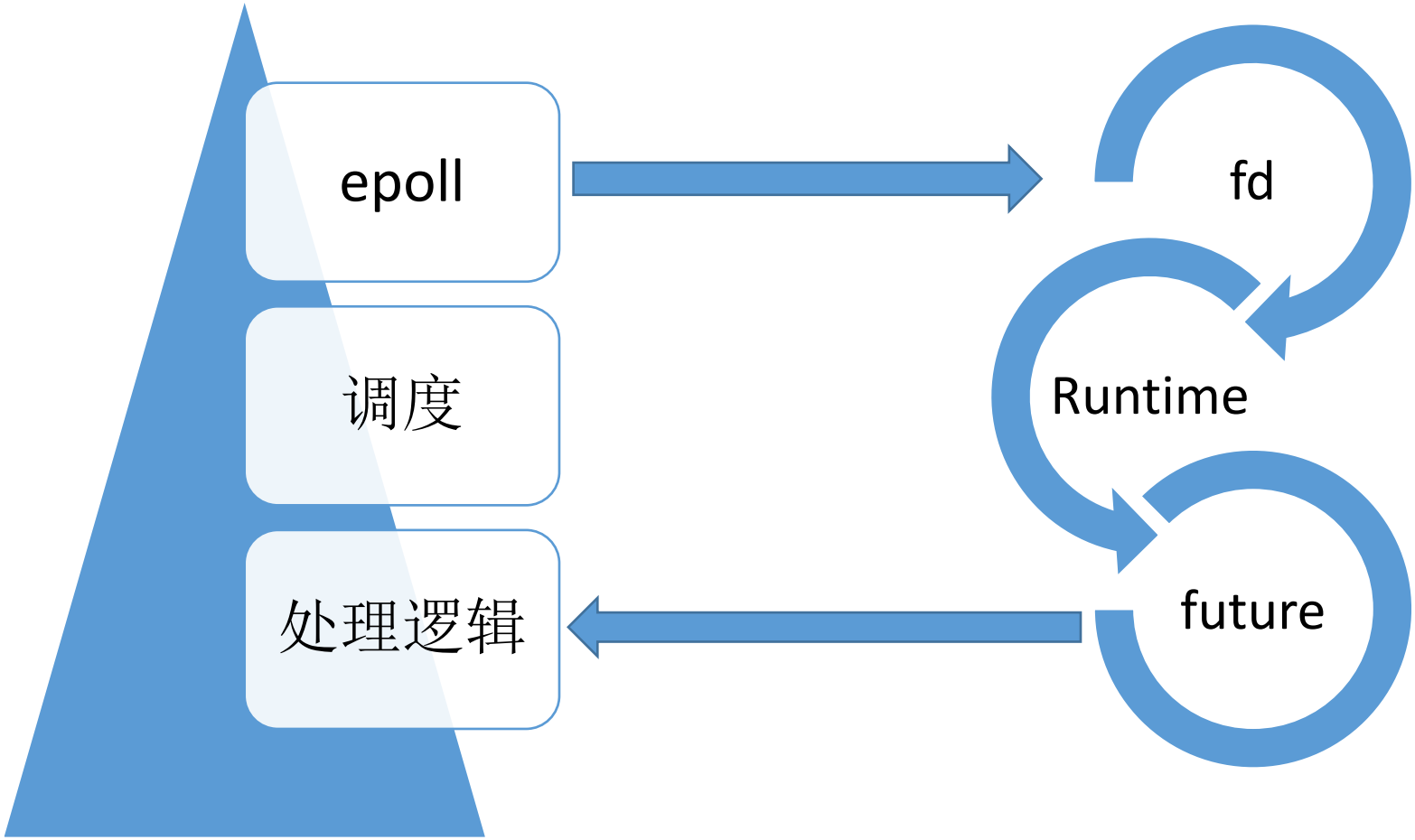
Stream

```rust
pub trait Stream {
    type Item;
    type Error;

    fn poll(&mut self) -> Result<Async<Option<Self::Item>>, Self::Error>;
}
```

Stream 的 状态转移

- 1. Err(Self::Error) 表示发生了错误，终止此 Stream。
- 2. Ok(Async::NotReady) 表示Stream暂时没有数据。
- 3. Ok(Async::Ready(Some(Self::Item))) 表示正确取到了数据。
- 4. Ok(Async::Ready(None)) 表示没收到数据且本 Stream 已经被关闭，不会再有数据过来了。

Future与mio的关联

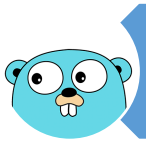


线程模型 & TLDR

开销分析

epoll 与 Rust

关于 tokio 与 futures



聊聊 coroutine

案例分析(1)

https://sourcegraph.com/github.com/wayslog/aster/-/blob/libaster/src/proxy/cluster.rs#L546-600

```
1 let fut = lazy()
2    .and_then(...)
3    .and_then(...)
4    .and_then(...);
5 current_thread::block_on_all(fut).unwrap();
6
```

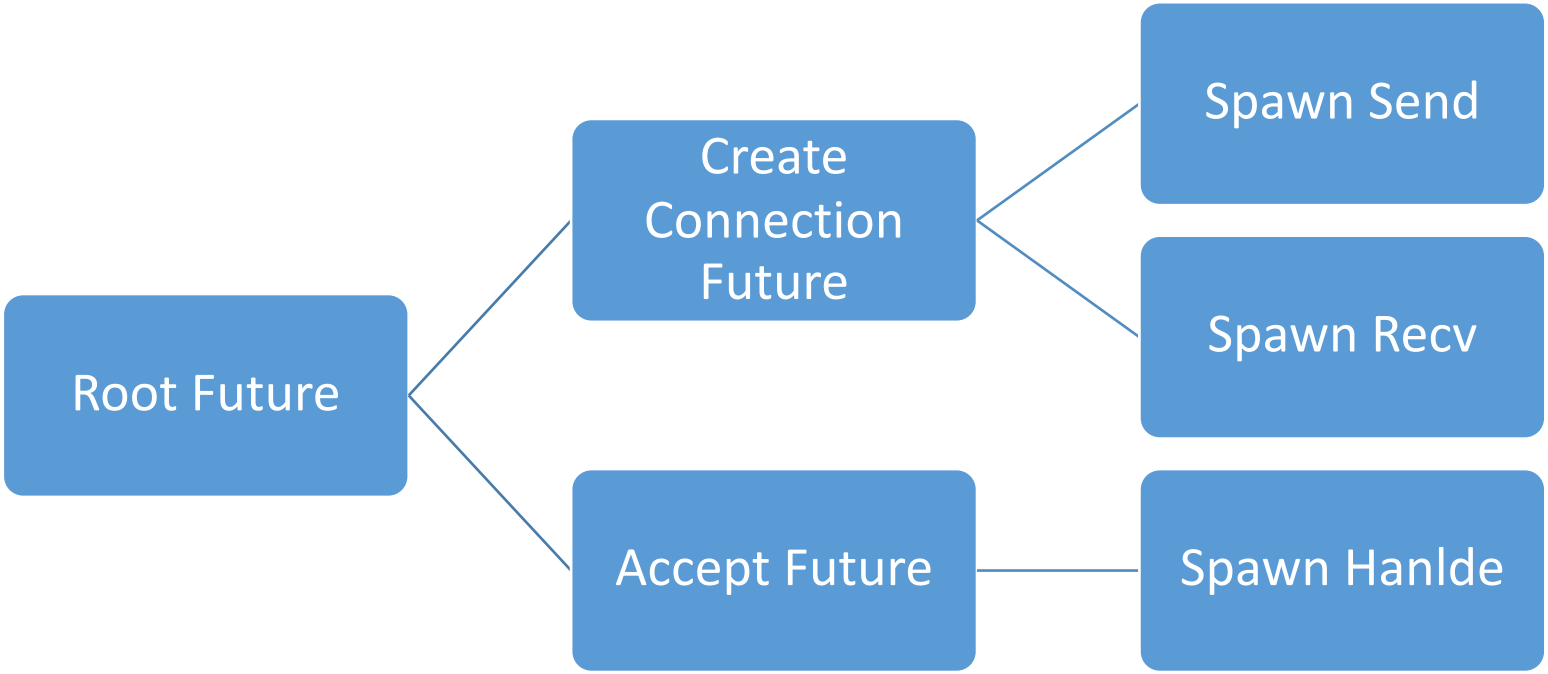
案例分析(2)

```
1 current_thread::spawn(amt);
2 Ok(())
```

```
1 current_thread::spawn(amt);
2 println!("i am baka");
3 Ok(())
```

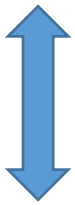
```
1 INFO 2018-11-29T11:31:10Z: libaster::proxy: setup ping for cluster test-redis-proxy
2 i am baka
3 DEBUG 2018-11-29T11:31:10Z: libaster::cluster::fetcher: trying to execute cmd to 127.0.0.1:7000
```

Future spawn wait



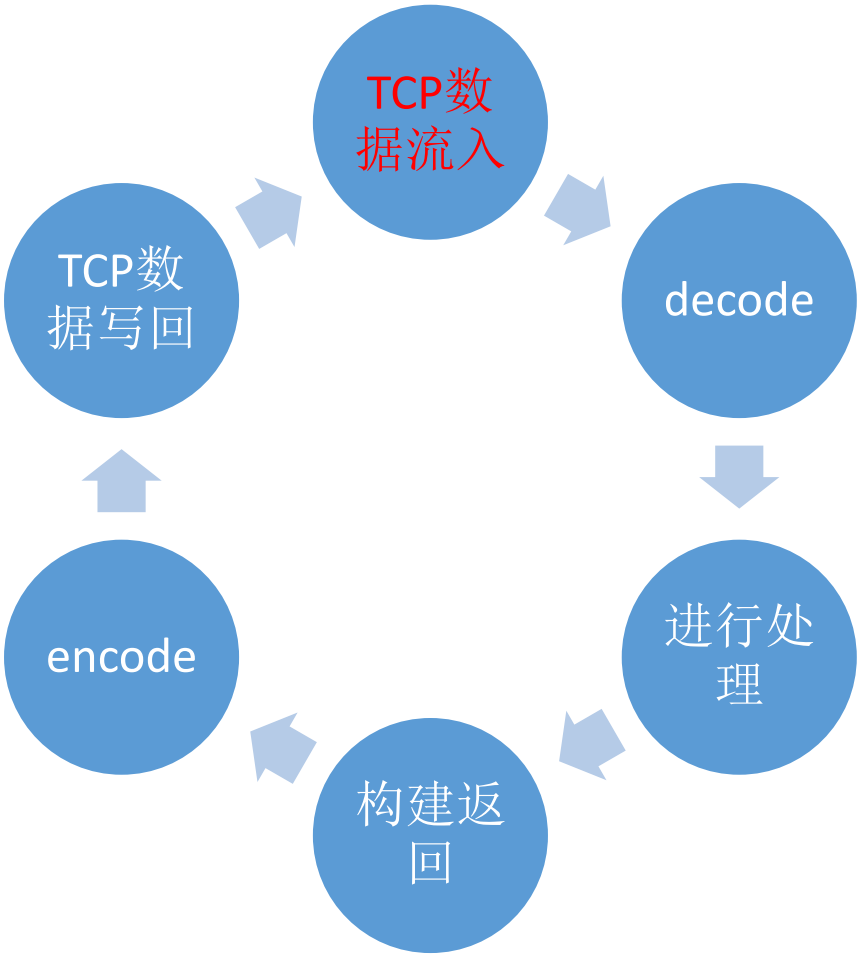
案例分析(3)

```
let amt = listen.incoming().for_each(...);
```



```
Stream.for_each() => Future
```

典型服务的模型



```rust
pub trait Decoder {
    type Item;
    type Error: From<Error>;
    fn decode(&mut self, src: &mut BytesMut) -> Result<Option<Self::Item>, Self::Error>;
}

pub trait Encoder {
    type Item;
    type Error: From<Error>;
    fn encode(&mut self, item: Self::Item, dst: &mut BytesMut) -> Result<(), Self::Error>;
}
```

案例分析(4)

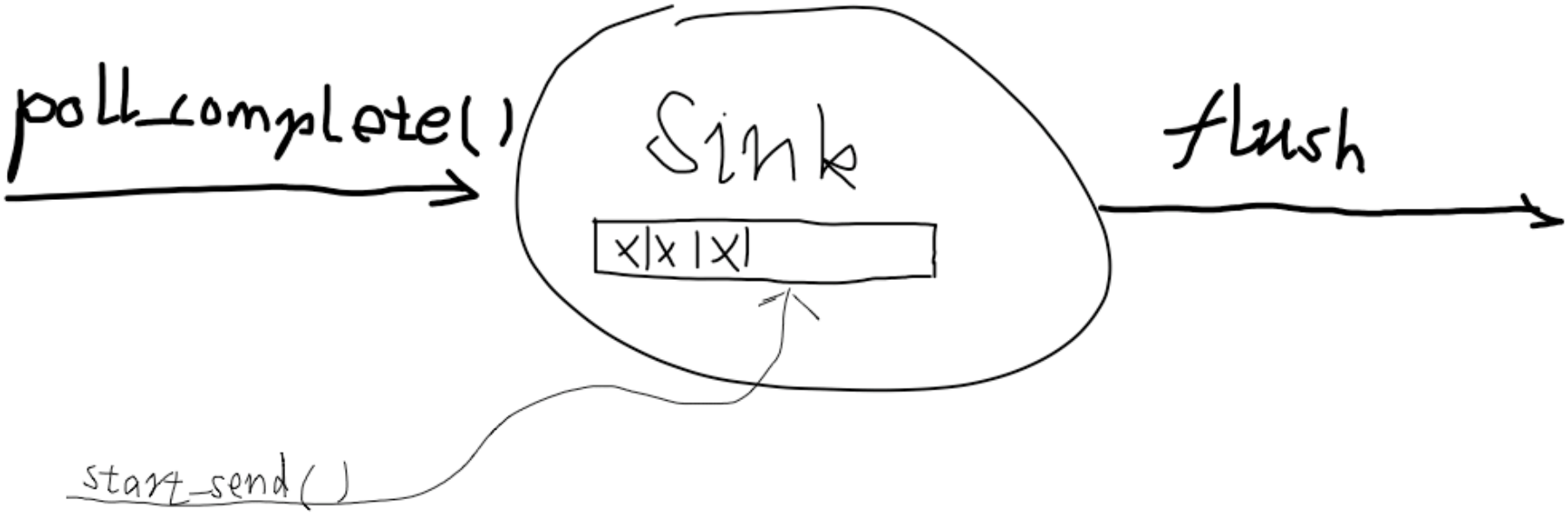
```
1 let codec = CmdCodec::default();
2 let (cmd_tx, resp_rx) = codec.framed(sock).split();
```

案例分析(5)

```rust
pub struct CmdCodec {...}

impl Decoder for CmdCodec {
    type Item = Resp;
    type Error = Error;
    fn decode(&mut self, src: &mut BytesMut)
        -> Result<Option<Self::Item>, Self::Error> {...}
}

impl Encoder for CmdCodec {
    type Item = Cmd;
    type Error = Error;
    fn encode(&mut self, item: Self::Item, dst: &mut BytesMut)
        -> Result<(), Self::Error> {...}
}
```

Stream 组合：Sink

```rust
pub trait Sink {
    type SinkItem;
    type SinkError;
    fn start_send(&mut self, item: Self::SinkItem)
        -> Result<AsyncSink<Self::SinkItem>, Self::SinkError>;

    fn poll_complete(&mut self) -> Result<Async<()>, Self::SinkError>;
    fn close(&mut self) -> Result<Async<()>, Self::SinkError> { ... }
}
```


poll_complete()
Sink
flush
x|x|x|
start_send()

案例分析(6)

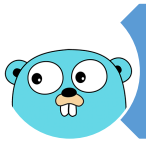
```rust
pub fn dispatch_all(&self, cmds: &mut VecDeque<Cmd>) -> Result<usize, AsError> {
    let mut count = 0usize;
    loop {
        ...
        if let Some(sender) = conns.get_mut(&addr).map(|x| x.sender()) {
            match sender.start_send(cmd) {
                Ok(AsyncSink::Ready) => {
                    // trace!("success start command into backend");
                    count += 1;
                }
                Ok(AsyncSink::NotReady(cmd)) => {
                    cmd.borrow_mut().add_cycle();
                    cmds.push_front(cmd);
                    return Ok(count);
                }
                Err(se) => {
                    let cmd = se.into_inner();
                    cmd.borrow_mut().add_cycle();
                    cmds.push_front(cmd);
                    self.connect(&addr, &mut conns)?;
                    return Ok(count);
                }
            }
        } else { ... }
    }
}
```

线程模型 & TLDR

开销分析

epoll 与 Rust

关于 tokio 与 futures



聊聊 coroutine

- 每次重入都需要翻一翻小本本
- 鱼的记忆只有7秒
- Future没有记忆
- 不能保存上下文

Rust的协程

无栈

- 无法简单的递归自己
- 不能被 move

async/await 语法

- 来自巨硬的赠与
- 调度需要靠自己

终于不吵了

- 2019年终于stable了
- 但是完美的错过了edition-2018

The `.await` is over, `async fn`s are here

Previously in Rust 1.36.0, we announced that the `Future` trait is here. Back then, we noted that:

> With this stabilization, we hope to give important crates, libraries, and the ecosystem time to prepare for `async` / `.await`, which we'll tell you more about in the future.

A promise made is a promise kept. So in Rust 1.39.0, we are pleased to announce that `async` / `.await` is stabilized! Concretely, this means that you can define `async` functions and blocks and `.await` them.

An `async` function, which you can introduce by writing `async fn` instead of `fn`, does nothing other than to return a `Future` when called. This `Future` is a suspended computation which you can drive to completion by `.await`ing it. Besides `async fn`, `async { ... }` and `async move { ... }` blocks, which act like closures, can be used to define "async literals".

For more on the release of `async` / `.await`, read Niko Matsakis's blog post.

thanks

Rust 编程语言社区水群

扫一扫二维码，加入群聊。

Rust 编程语言社区 2 群

扫一扫二维码，加入群聊。