

Monoio

基于 `io_uring` 的高效 Rust
异步运行时

CloudWeGo 开源团队出品

2022/11



个人介绍



茌海(Chi Hai)

GitHub: ihciah

- 毕业于复旦大学(但读研读到一半跳车跑路了)
- Rust 语言爱好者

字节跳动研发工程师

- Rust Async Runtime
- Rust RPC 框架(Volo)
- 公司内 Rust 基础库 / 基础设施

CONTENTS

目录

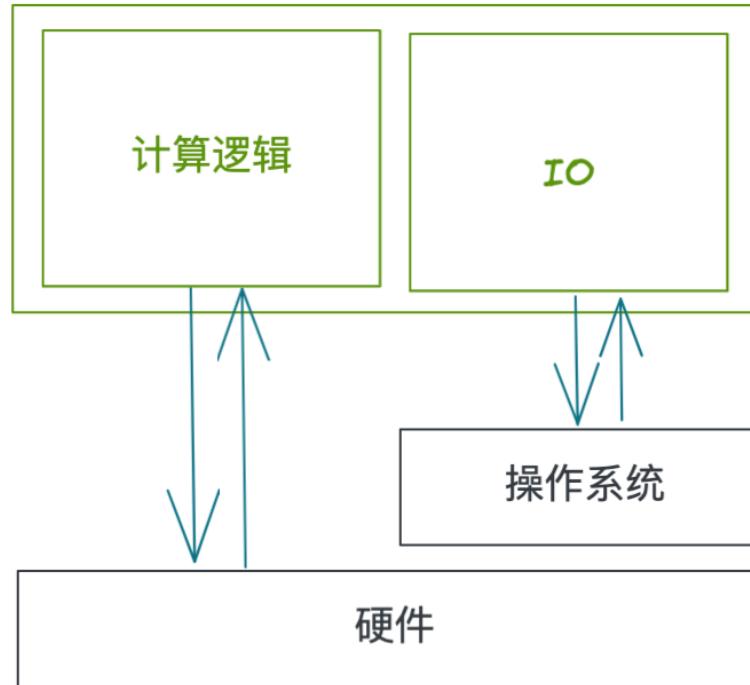
-  01. **Rust 异步机制**
介绍 Rust 语言中的异步机制
-  02. **Monoio 设计**
Monoio 的设计要点
-  03. **Runtime 对比、选型与应用**
有什么不同、有哪些局限性？怎么选？有哪些应用？

01

Rust 异步机制

介绍 Rust 语言中的异步机制

Rust 异步机制



语言 & 编译器 => 计算逻辑生成安全、高效的机器码

Runtime -> ? ? ? ? ? => 高效地和操作系统打交道?

Future + Async -> ? ? ? ? ? => 友好地表达计算逻辑和 IO 的关系?
(We don't like callback!)

Rust 异步机制 – Example

Download 2 files in parallel with 2 threads:

```
fn download_parallel() {  
    let t1 = thread::spawn(|| download_file("http://example.com/file_a"));  
    let t2 = thread::spawn(|| download_file("http://example.com/file_b"));  
  
    t1.join();  
    t2.join();  
}
```

相比线程高效的多！

Download 2 files in parallel with 2 tasks:

```
async fn download_parallel_async() {  
    let task1 = download_file_async("http://example.com/file_a");  
    let task2 = download_file_async("http://example.com/file_b");  
  
    join!(task1, task2);  
}
```

Rust 异步机制 – Example

像写同步函数一样写异步函数（面向过程而非面向状态）：

```
#[inline(never)]
async fn do_http() -> i32 {
    // do http request in async way
    1
}

pub async fn sum() -> i32 {
    do_http().await + do_http().await + 1
}
```

它也不需要写一堆 callback，
和写同步函数一样写异步就好！

Rust 异步机制 – Async Await 背后的秘密

```
#[inline(never)]
async fn do_http() -> i32 {
    // do http request in async way
    1
}

pub async fn sum() -> i32 {
    do_http().await + do_http().await + 1
}
```

Async + Await => Generator => State Machine
=>
Implement Future

Rust 异步机制 – Async Await 背后的秘密

```
#[inline(never)]
async fn do_http()
->
/*impl Trait*/ #[lang = "from_generator"](
// do http request in async way
move |mut _task_context| { { let _t = { 1 }; _t } })

async fn sum()
->
/*impl Trait*/ #[lang = "from_generator"](|move |mut _task_context|
{
    {
        let _t =
        {
            match #[lang = "into_future"](do_http()) {
                mut __awaitee =>
                loop {
                    match unsafe {
                        #[lang = "poll"](|#[lang = "new_unchecked"](&mut __awaitee),
                        #[lang = "get_context"](_task_context))
                    } {
                        #[lang = "Ready"] { 0: result } => break result,
                        #[lang = "Pending"] {} => {}
                    }
                    _task_context = (yield ());
                },
            } +
            match #[lang = "into_future"](do_http()) {
                mut __awaitee =>
                loop {
                    match unsafe {
                        #[lang = "poll"](|#[lang = "new_unchecked"](&mut __awaitee),
                        #[lang = "get_context"](_task_context))
                    } {
                        #[lang = "Ready"] { 0: result } => break result,
                        #[lang = "Pending"] {} => {}
                    }
                    _task_context = (yield ());
                },
            } + 1
        };
        _t
    }
})
```

Async + Await => Generator => State Machine
=>
Implement Future

Rust 异步机制 – Async Await 背后的秘密

```
fn sum:{closure#0}(_1: Pin<&mut [static generator@src/lib.rs:7:27: 9:2]>, _2: ResumeTy) -> GeneratorState<(), i32> {
    debug _task_context => _32; // in scope 0 at src/lib.rs:7:27: 9:2
    let mut _0: std::ops::GeneratorState<(), i32>; // return place in scope 0 at src/lib.rs:7:27: 9:2
    let mut _3: i32; // in scope 0 at src/lib.rs:8:5: 8:38
    let mut _4: impl std::future::Future<Output = i32>; // in scope 0 at src/lib.rs:8:14: 8:20
    let mut _5: impl std::future::Future<Output = i32>; // in scope 0 at src/lib.rs:8:5: 8:14
    let mut _6: std::task::Poll<i32>; // in scope 0 at src/lib.rs:8:14: 8:20
    let mut _7: std::pin::Pin<&mut impl std::future::Future<Output = i32>>; // in scope 0 at src/lib.rs:8:14: 8:20
    let mut _8: &mut impl std::future::Future<Output = i32>; // in scope 0 at src/lib.rs:8:14: 8:20
    let mut _9: &mut impl std::future::Future<Output = i32>; // in scope 0 at src/lib.rs:8:14: 8:20
    let mut _10: &mut std::task::Context<'_>; // in scope 0 at src/lib.rs:8:5: 8:20
    let mut _11: &mut std::task::Context<'_>; // in scope 0 at src/lib.rs:8:5: 8:20
    let mut _12: std::future::ResumeTy; // in scope 0 at src/lib.rs:8:14: 8:20
    let mut _13: isize; // in scope 0 at src/lib.rs:8:14: 8:20
    let mut _15: std::future::ResumeTy; // in scope 0 at src/lib.rs:8:14: 8:20
    let mut _16: (); // in scope 0 at src/lib.rs:8:14: 8:20
    let mut _17: i32; // in scope 0 at src/lib.rs:8:23: 8:38
    let mut _18: impl std::future::Future<Output = i32>; // in scope 0 at src/lib.rs:8:32: 8:38
    let mut _19: impl std::future::Future<Output = i32>; // in scope 0 at src/lib.rs:8:23: 8:32
    let mut _20: std::task::Poll<i32>; // in scope 0 at src/lib.rs:8:32: 8:38
    let mut _21: std::pin::Pin<&mut impl std::future::Future<Output = i32>>; // in scope 0 at src/lib.rs:8:32: 8:38
    let mut _22: &mut impl std::future::Future<Output = i32>; // in scope 0 at src/lib.rs:8:32: 8:38
    let mut _23: &mut impl std::future::Future<Output = i32>; // in scope 0 at src/lib.rs:8:32: 8:38
    let mut _24: &mut std::task::Context<'_>; // in scope 0 at src/lib.rs:8:23: 8:38
    let mut _25: &mut std::task::Context<'_>; // in scope 0 at src/lib.rs:8:23: 8:38
    let mut _26: std::future::ResumeTy; // in scope 0 at src/lib.rs:8:32: 8:38
    let mut _27: isize; // in scope 0 at src/lib.rs:8:32: 8:38
    let mut _29: std::future::ResumeTy; // in scope 0 at src/lib.rs:8:32: 8:38
    let mut _30: (); // in scope 0 at src/lib.rs:8:32: 8:38
    let mut _31: i32; // in scope 0 at src/lib.rs:7:27: 9:2
    let mut _32: std::future::ResumeTy; // in scope 0 at src/lib.rs:7:27: 9:2
    let mut _33: u32; // in scope 0 at src/lib.rs:7:27: 9:2
    let mut _34: &mut [static generator@src/lib.rs:7:27: 9:2]; // in scope 0 at src/lib.rs:7:27: 9:2
    let mut _35: &mut [static generator@src/lib.rs:7:27: 9:2]; // in scope 0 at src/lib.rs:7:27: 9:2
    let mut _36: &mut [static generator@src/lib.rs:7:27: 9:2]; // in scope 0 at src/lib.rs:7:27: 9:2
    let mut _37: &mut [static generator@src/lib.rs:7:27: 9:2]; // in scope 0 at src/lib.rs:7:27: 9:2
    let mut _38: &mut [static generator@src/lib.rs:7:27: 9:2]; // in scope 0 at src/lib.rs:7:27: 9:2
    let mut _39: &mut [static generator@src/lib.rs:7:27: 9:2]; // in scope 0 at src/lib.rs:7:27: 9:2
    let mut _40: &mut [static generator@src/lib.rs:7:27: 9:2]; // in scope 0 at src/lib.rs:7:27: 9:2
    let mut _41: &mut [static generator@src/lib.rs:7:27: 9:2]; // in scope 0 at src/lib.rs:7:27: 9:2
    let mut _42: &mut [static generator@src/lib.rs:7:27: 9:2]; // in scope 0 at src/lib.rs:7:27: 9:2
    let mut _43: &mut [static generator@src/lib.rs:7:27: 9:2]; // in scope 0 at src/lib.rs:7:27: 9:2
    let mut _44: &mut [static generator@src/lib.rs:7:27: 9:2]; // in scope 0 at src/lib.rs:7:27: 9:2
    scope 1 {
        debug __awaitee => (((*_1.0: &mut [static generator@src/lib.rs:7:27: 9:2])) as variant#3).0: impl std::future:
        let _14: i32; // in scope 1 at src/lib.rs:8:5: 8:20
    }
```

Async + Await => Generator => State Machine
=>
Implement Future

Rust 异步机制 – Future 抽象

Rust 中对异步 Task 的核心抽象：Future

```
pub trait Future {  
    type Output;  
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;  
}  
  
pub enum Poll<T> {  
    Ready(T),  
    Pending,  
}
```

Future 描述状态机对外暴露的接口：

1. 推动状态机执行
2. 返回执行结果：
 1. 遇到了阻塞：Pending
 2. 执行完毕：Ready + 返回值

异步 Task 的本质：实现 Future 的状态机

Rust 异步机制 – 手动实现 Future

```
// auto generate
async fn do_http() -> i32 {
    // do http request in async way
    1
}

// manually impl
fn do_http() -> DoHTTPFuture { DoHTTPFuture }

struct DoHTTPFuture;
impl Future for DoHTTPFuture {
    type Output = i32;
    fn poll(self: Pin<&mut Self>, _cx: &mut Context<'_>) -> Poll<Self::Output> {
        Poll::Ready(1)
    }
}
```

虽然 Async + Await 可以生成 Future，我们依然可以自行实现…

事实上这种需要自行实现 Future 状态机的场景非常多

Rust 异步机制 – 手动实现 Future

```
// auto generate
async fn sum() -> i32 {
    do_http().await + do_http().await + 1
}

// manually impl
fn sum() -> SumFuture { SumFuture::FirstDoHTTP(DoHTTPFuture) }

enum SumFuture {
    FirstDoHTTP(DoHTTPFuture),
    SecondDoHTTP(DoHTTPFuture, i32),
}
impl Future for SumFuture {
    type Output = i32;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
        let this = self.get_mut();
        loop {
            match this {
                SumFuture::FirstDoHTTP(f) => {
                    let pinned = unsafe { Pin::new_unchecked(f) };
                    match pinned.poll(cx) {
                        Poll::Ready(r) => {
                            *this = SumFuture::SecondDoHTTP(DoHTTPFuture, r);
                        }
                        Poll::Pending => {
                            return Poll::Pending;
                        }
                    }
                }
                SumFuture::SecondDoHTTP(f, prev_sum) => {
                    let pinned = unsafe { Pin::new_unchecked(f) };
                    return match pinned.poll(cx) {
                        Poll::Ready(r) => Poll::Ready(*prev_sum + r + 1),
                        Poll::Pending => Poll::Pending,
                    };
                }
            }
        }
    }
}
```

涉及 await => 喜提状态机

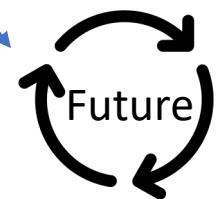
每个 await 点都可能返回 Pending
所以需要在每个 await 点都对应一个状态

Rust 异步机制 – Task, Future 和 Runtime 的关系

Task 的本质：包含用户逻辑的状态机



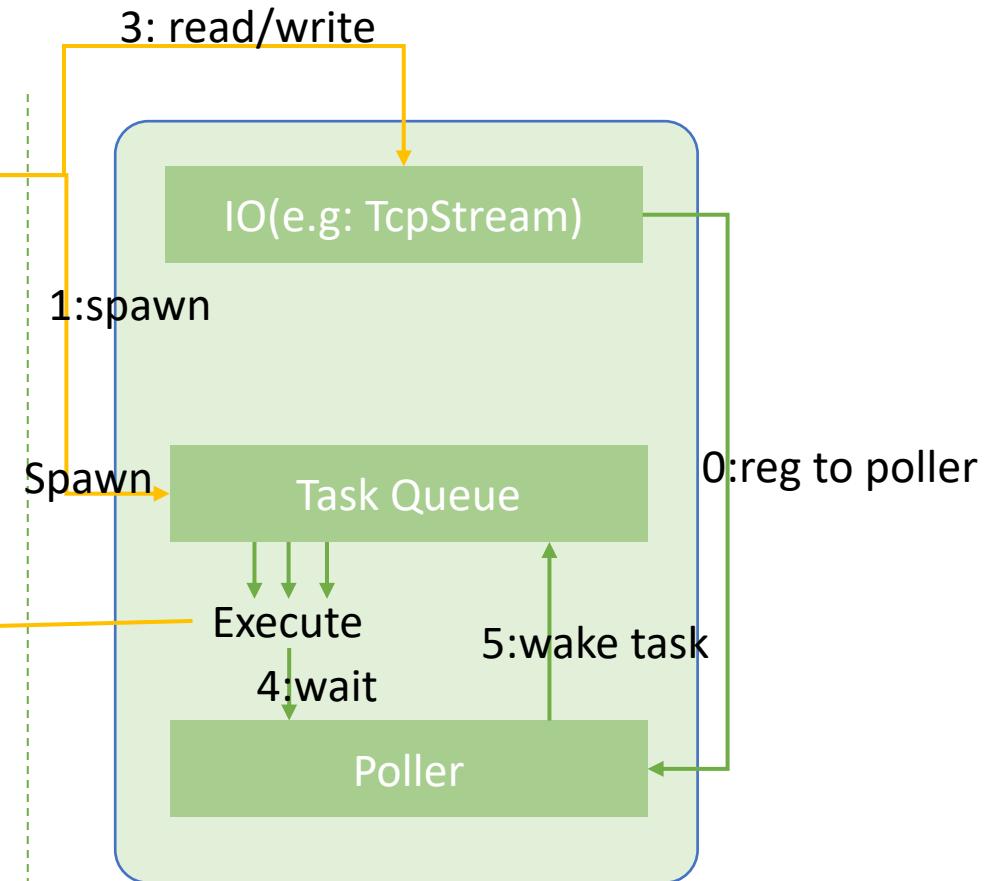
Task2



Future：状态机如何被驱动运转
Task 实现了 Future

Future & Waker(Waker 稍后讲)

Rust 语言本身提供核心抽象



Runtime

外部实现

Rust 异步机制 – Waker

我知道 Future 本质是状态机没错，我也知道状态机每次运转可能返回 Pending / Ready…

但是当它遇到 io 阻塞返回 Pending 时，谁来感知 io 就绪？io 就绪后怎么重新驱动 Future 运转？

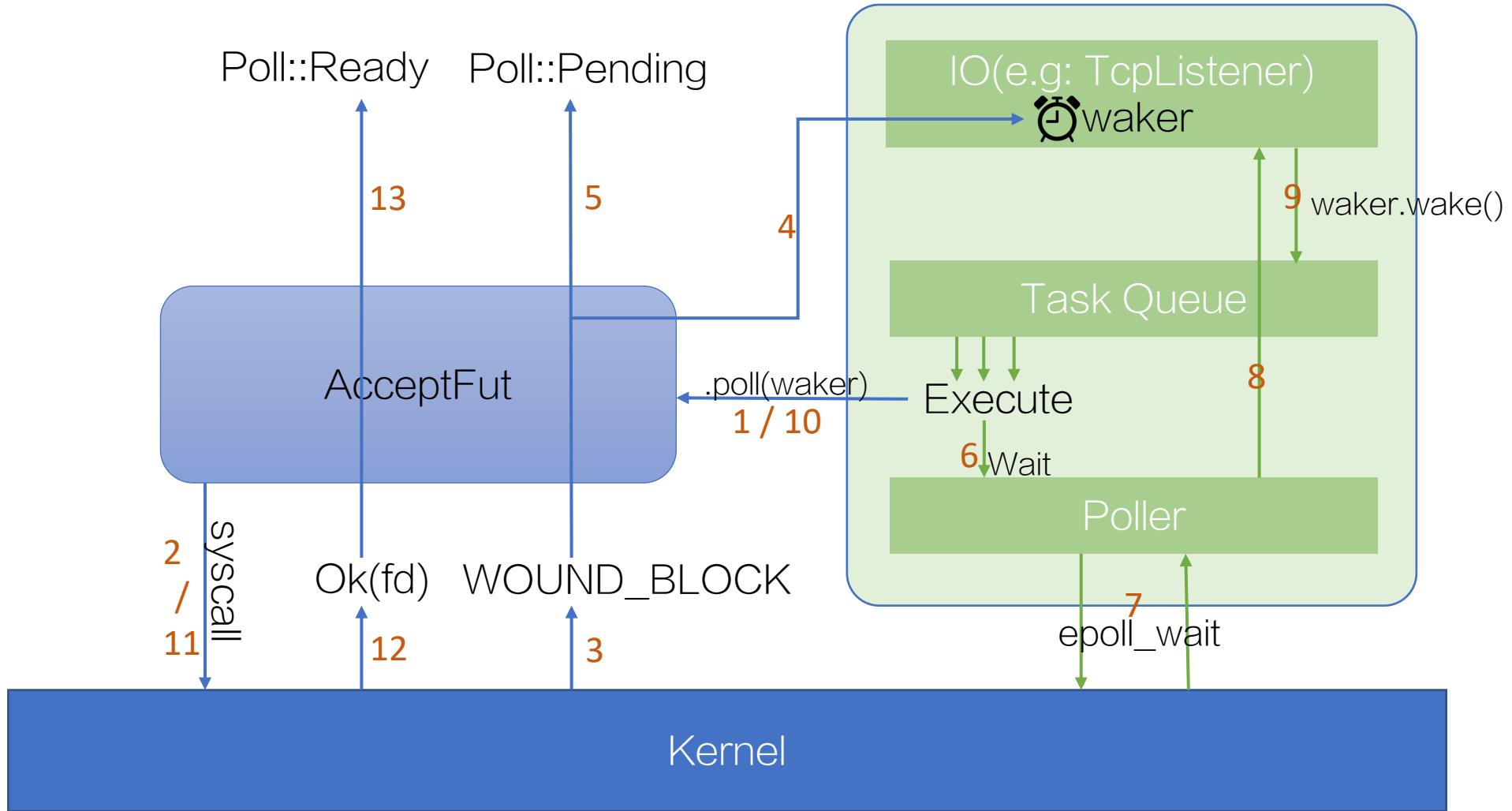
```
pub trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}

pub struct Context<'a> {
    // 可以拿到用于唤醒 Task 的 Waker
    waker: &'a Waker,

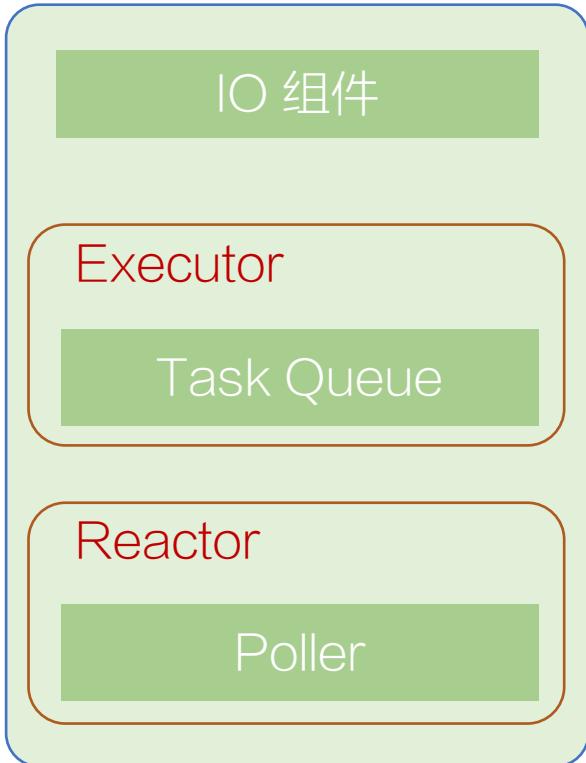
    // 标记字段，忽略即可
    _marker: PhantomData<fn(&'a ()) -> &'a ()>,
}
```

注：Waker 不是 Trait，但和 Trait 很像，它内部包含一个裸指针和一个 vtable。
它由 Runtime 构造，行为也由 Runtime 实现。
为了理解方便，可以暂时将它视作一个 Trait Object。

Rust 异步机制 – Waker



Rust 异步机制 – Runtime



IO 组件:

1. 提供异步接口
2. 并能将自己的 fd 注册到Reactor上
3. 在 IO 未就绪时，将 waker 放到关联任务中

Executor:

1. 取出并推动任务执行
2. 无事可做时将执行权交给 Reactor

Reactor:

1. 与 kernel 打交道，等待 IO 就绪
2. 标记 IO 就绪状态，并将就绪 IO 关联的任务唤醒（加入执行队列）
3. 执行权交还给 Executor

02

Monoio 设计

Monoio 的设计要点

01.

基于 GAT 的纯异步 IO 接口

基于 GAT+TAIT，统一 io_uring 和 epoll 操作模式

02.

上层无感知的 Driver 探测与切换

支持动态探测 io_uring 可用性，运行时选择 Driver
用户只需实现一套代码，即可在两种模式下运行
借助 OpAble 抽象统一组件实现

03.

性能与功能兼具

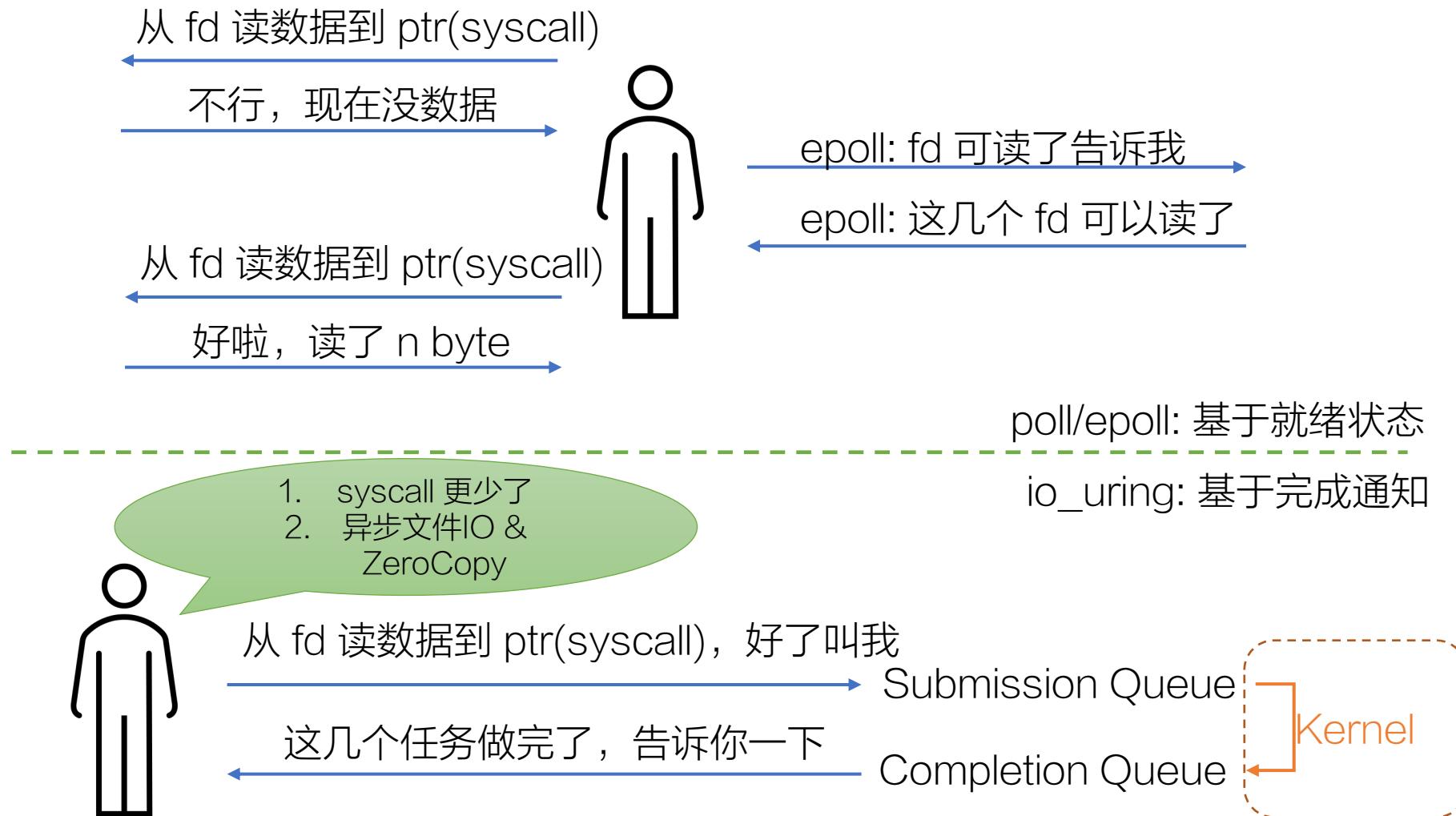
基于 thread-per-core 模型，热路径无锁无竞争
支持 io_uring / epoll / kqueue，高并发下降低 syscall 开销，同时兼容低版本内核
同时支持跨线程交互、等待与 spawn_blocking

04.

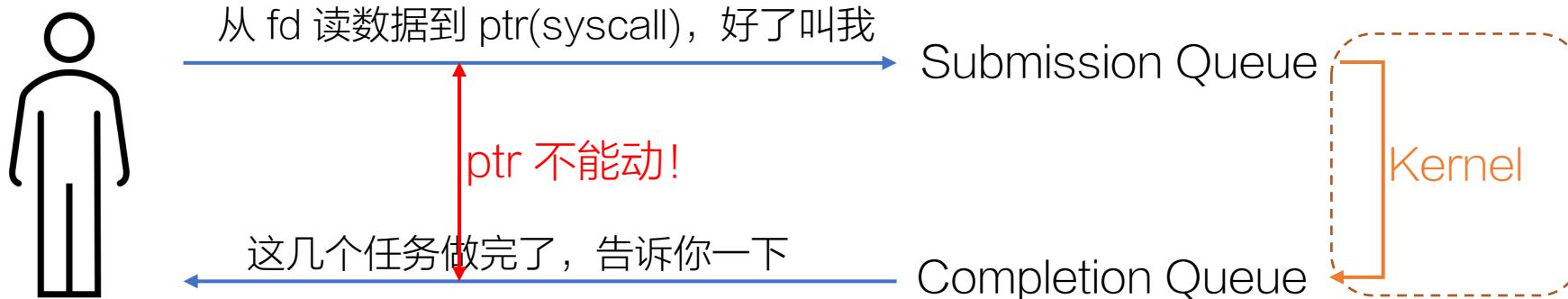
兼容接口

提供 IO 接口兼容组件，兼容已有生态

Monoio 设计 - 基于 GAT 的纯异步 IO 接口



Monoio 设计 - 基于 GAT 的纯异步 IO 接口



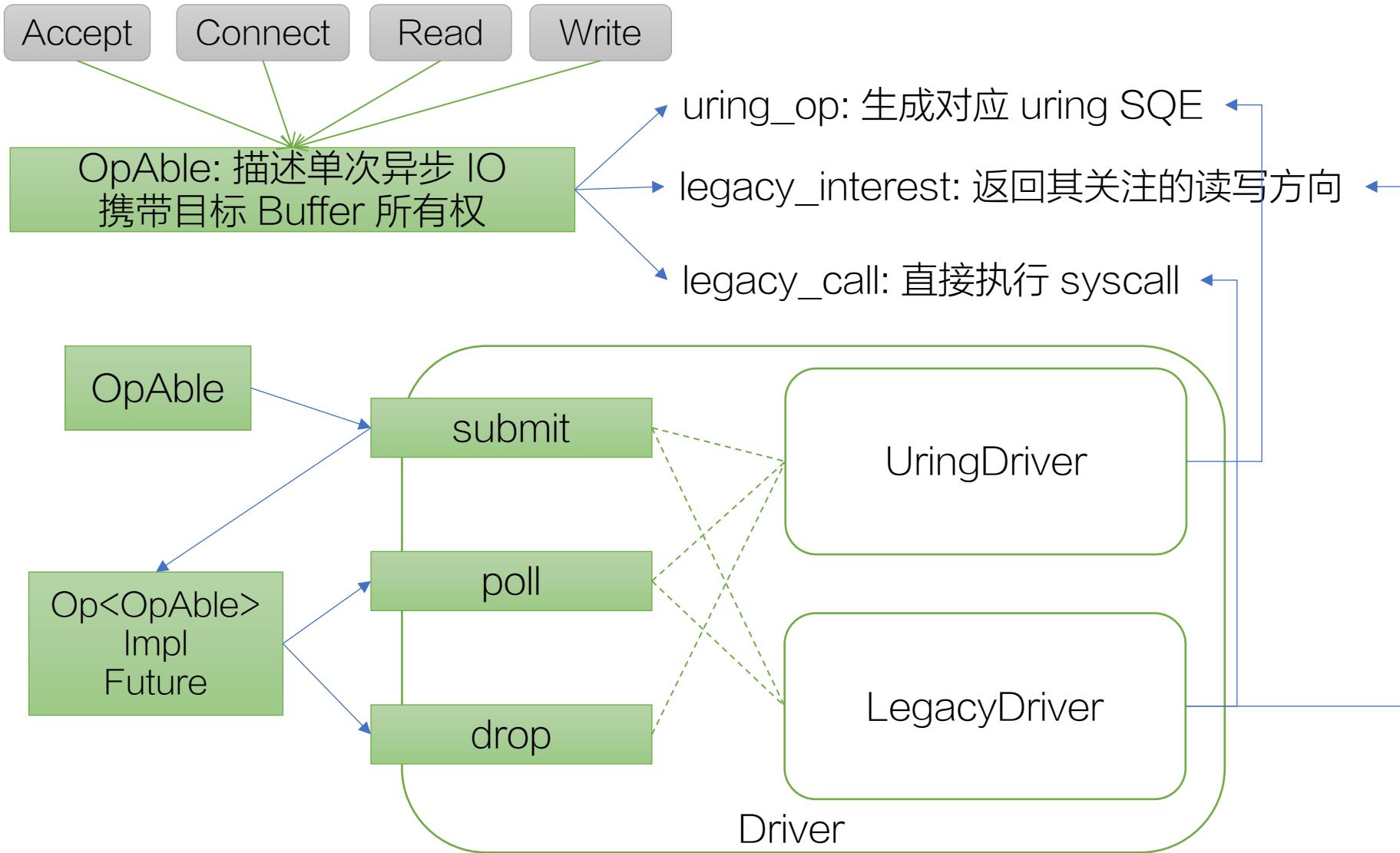
```
pub trait AsyncReadRent {  
    type ReadFuture<'a, T>: Future<Output = (std::io::Result<usize>, T)>  
    where  
        Self: 'a, T: IoBufMut + 'a;  
    type ReadvFuture<'a, T>: Future<Output = (std::io::Result<usize>, T)>  
    where  
        Self: 'a, T: IoVecBufMut + 'a;  
  
    fn read<T: IoBufMut>(&mut self, buf: T) -> Self::ReadFuture<'_, T>;  
    fn readv<T: IoVecBufMut>(&mut self, buf: T) -> Self::ReadvFuture<'_, T>;  
}
```

Monoio 设计 – 上层无感知的 Driver 探测与切换

1. 通过 Feature 与代码指定 Driver 并有条件地做运行时探测
2. 暴露统一的 IO 接口
3. 内部利用 OpAble 统一组件实现(对 Read、Write 等 Op 做抽象)

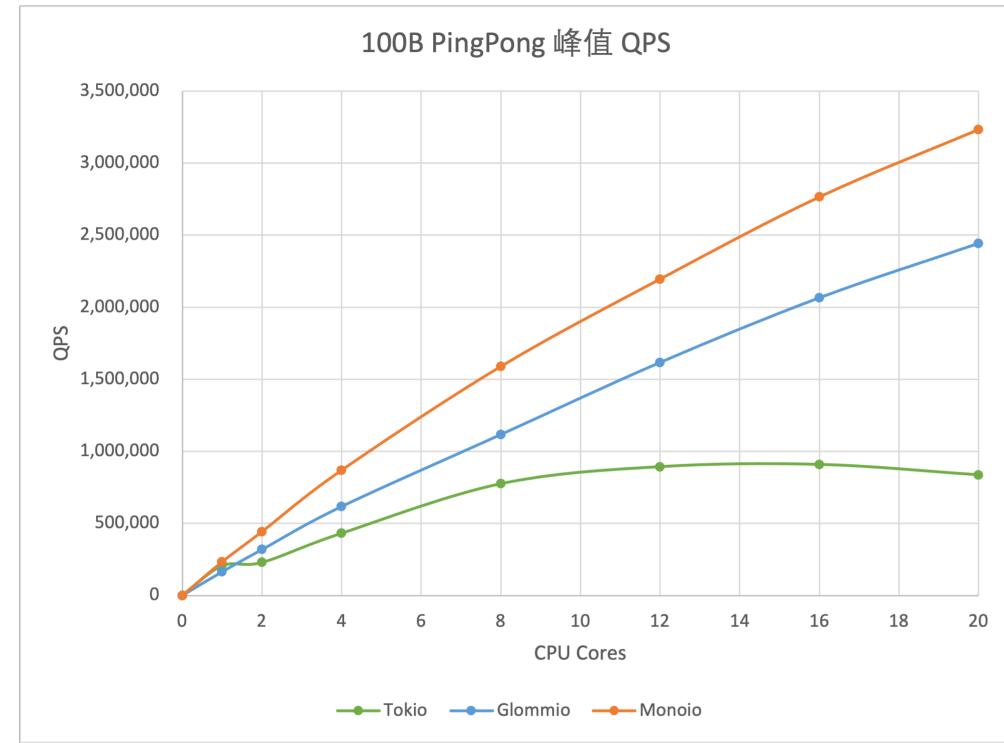
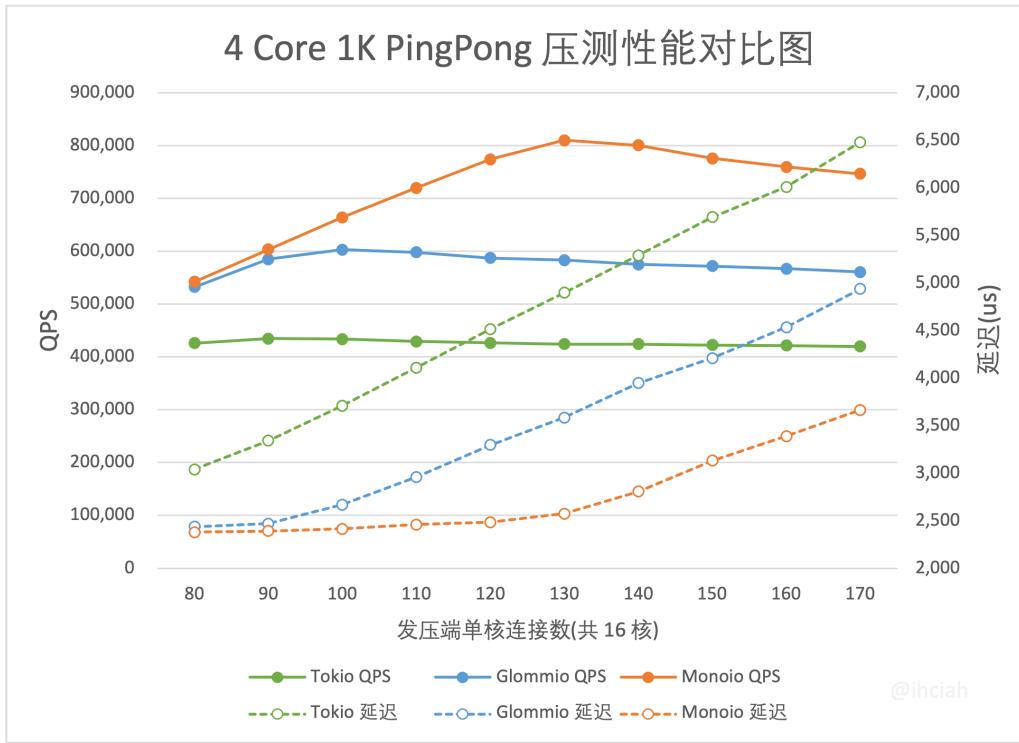
```
trait OpAble {  
    fn uring_op(&mut self) -> io_uring::squeue::Entry;  
  
    fn legacy_interest(&self) -> Option<(ready::Direction, usize)>;  
    fn legacy_call(&mut self) -> io::Result<u32>;  
}
```

Monoio 设计 – 上层无感知的 Driver 探测与切换



Monoio 设计 – 性能

- 所有 Task 均仅在固定线程运行，无任务窃取
- Task Queue 为 thread local 结构，操作无锁无竞争



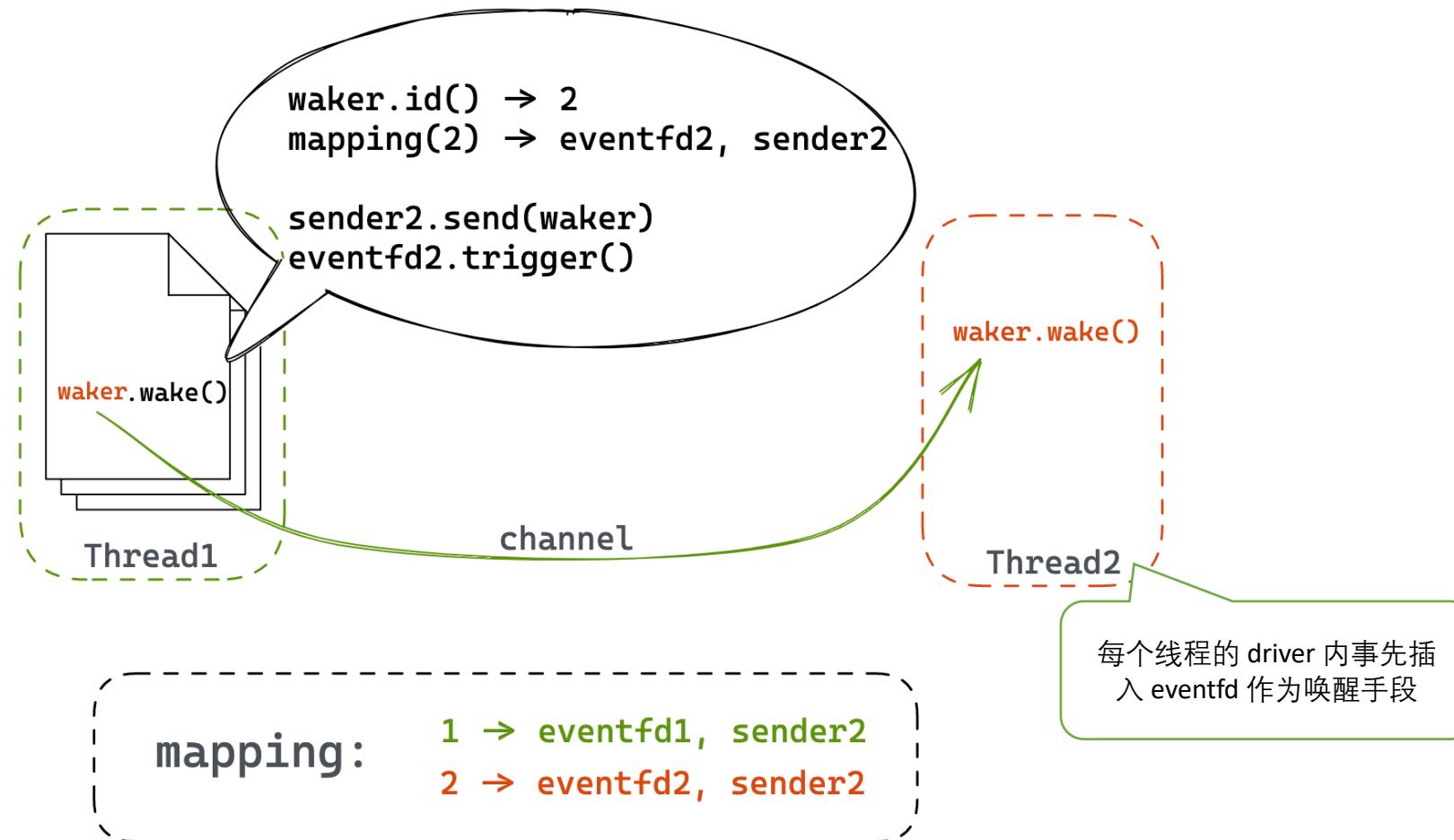
高性能的两个方面：

- Runtime 内部高性能：基本等价于裸对接 syscall
- 用户代码高性能：结构尽量 thread local 不跨线程 (Rc<RefCell<T>> / thread_local! vs Arc<Mutex<T>>)

Monoio 设计 – 功能性

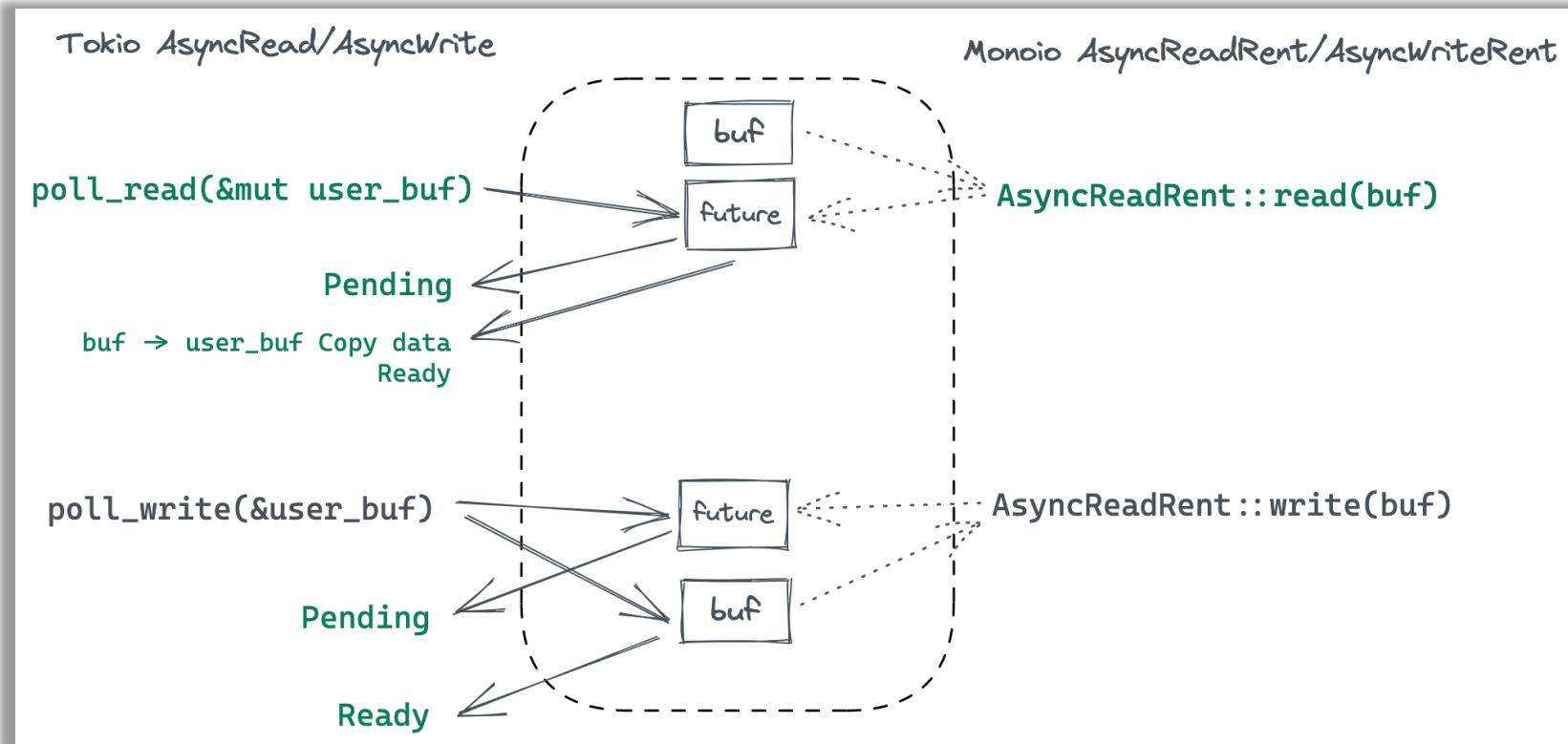
thread-per-core 并不意味着不能做跨线程通信：

1. 支持通过其他线程唤醒异步任务
2. 支持通过 spawn_blocking 将重计算任务或阻塞 IO 扔到独立线程池运行



Monoio 设计 – 兼容接口

主流生态对接 AsyncRead/AsyncWrite,
我们需要能够将 AsyncReadRent/AsyncWriteRent 包装为 AsyncRead/AsyncWrite



读：使用内置 `buf` 读，并等待读结束，之后拷贝至传入 `user_buf`。

写：拷贝至内置 `buf`，立刻 ack。后续调用时首先等待 `future` 执行完毕。

Monoio 设计 – 兼容接口

```
// tokio way
let tcp = tokio::net::TcpStream::connect("1.1.1.1:80").await.unwrap();

// monoio way(with monoio-compat)
let monoio_tcp = monoio::net::TcpStream::connect("1.1.1.1:80").await.unwrap();
let tcp = monoio_compatible::StreamWrapper::new(monoio_tcp);

// both of them implements tokio::io::AsyncRead and tokio::io::AsyncWrite
```

03

Runtime 对比、选型与应用

有什么不同、有哪些局限性？怎么选？有哪些应用？

Runtime 对比、选型

Tokio

通用场景 => 任务窃取

跨平台 => epoll / kqueue / iocp / ...

组件生态齐全

水平扩展性不佳；但通用场景下可以一定程度提升资源利用率

适用：对性能没有极高的要求的通用业务

Tokio-uring

设计比较漂亮（Monoio 也参考了其中一些设计）

基于 Tokio，在 epoll 之上运行 io_uring，无法做到用户透明

没有提供通用 IO 接口

Monoio

限定场景 => thread-per-core (like nginx/envoy)

限定平台 => io_uring 为主；epoll / kqueue 仅作 fallback

生态当前比较缺乏，依赖兼容层

水平扩展性好，性能好；但对业务场景和编程模式有限制

适用：代理、网关、缓存、数据聚合、文件 IO 密集、DB 类型的组件

Monoio 应用

Monoio Gateway

基于 Monoio 实现的通用网关系统

<https://github.com/monoio-rs/monoio-gateway>

Volo

[WIP] 支持以 feature 形式集成至 Volo 框架

<https://github.com/cloudwego/volo>

内部试点业务

[WIP] 网关接入层代理

未来将从提升兼容性和组件建设上入手，让 Monoio 更好用，让基础组件开发能够更容易上手。

THANKS



补充：Rust Async vs Goroutine

Rust 异步与 Golang 有什么区别？

Goroutine 是有栈的，而 Rust 的是无栈的；

Goroutine 一定是 Send + Sync 的，同样无法有效利用 thread local 等；且没有精确感知/控制 thread 的可能性。

有栈：可抢占、容错性更好；但性能略差。

无栈：性能好；但不可抢占、遇到错误调用的同步 syscall 会卡住线程影响其他任务。

我能把写 Goroutine 的习惯迁移到 Rust 中吗？

基本上是可以的；但某些场景下有性能更好的写法（状态机是可组合的，而 Goroutine 并不可以）。

例子：

```
ch := make(chan int)
go task1(ch)
go task2(ch)
first_result := <-ch
second_result := <-ch
```

```
let fut1 = task1();
let fut2 = task2();
let (result1, result2) = join!(fut1, fut2);
```

补充： Rust Poll or Async?

poll_xxx? 手动实现 Future? async fn? 有点晕！

来捋一捋 ☺ 这个问题的重点在于 谁存储状态。

- 组件内部存储状态，则组件直接提供 poll 接口（poll 接口表达能力更强，可以零成本地通过 poll_fn 转为 Future）
- 外部存储状态，则返回 Future 状态机，调用方负责存储，Future 提供 poll 接口（手动/自动实现均可）

内部存储 or 外部存储？简单来说，状态可共享则可内部存储。

- 例如所有 poll_read/poll_write 调用均等待 io read/write ready，则可暴露为 poll-like
- client.get(url) 不同的调用对应不同的状态（比如连接不共享），则只能将状态机丢出去外部存储

