字节开源 Monoio： 基于 io-uring 的高性能 Rust Runtime

CloudWeGo
2023年04月18日 11:24 · 阅读 3140

关注

> 作者：**CloudWeGo Rust Team**
>
> GitHub: github.com/bytedance/m...

一、概述

尽管 Tokio 目前已经是 Rust 异步运行时的事实标准，但要实现极致性能的网络中间件还有一定距离。为了这个目标，CloudWeGo Rust Team 探索基于 io-uring 为 Rust 提供异步支持，并在此基

础上研发通用网关。

本文包括以下内容：

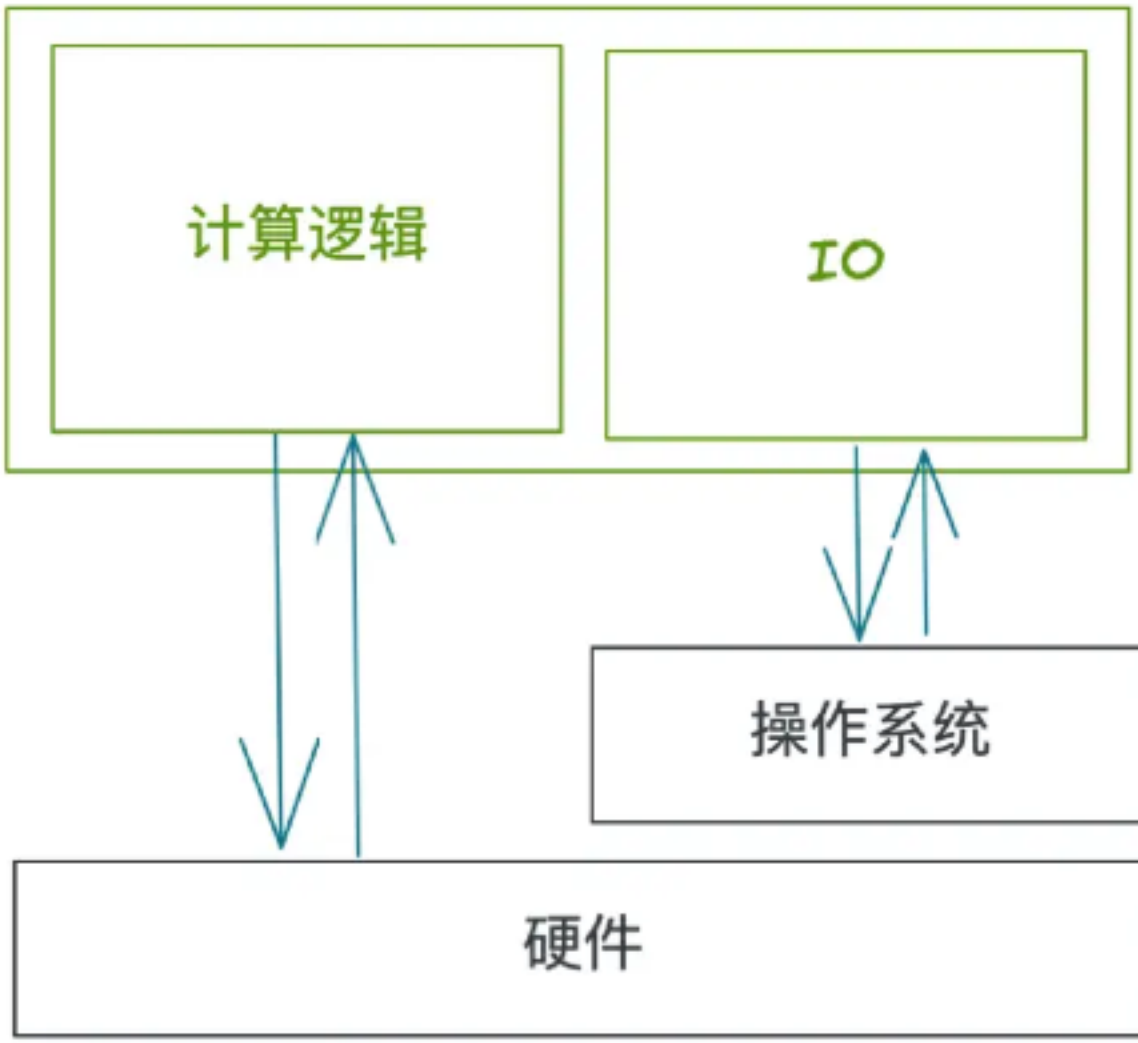
1. 介绍 Rust 异步 Runtime；
2. Monoio 的一些设计精要；
3. Runtime 对比选型与应用。

二、Rust 异步机制

借助 Rustc 和 llvm，Rust 可以生成足够高效且安全的机器码。但是一个应用程序除了计算逻辑以外往往还有 IO，特别是对于网络中间件，IO 其实是占了相当大比例的。

程序做 IO 需要和操作系统打交道，编写异步程序通常并不是一件简单的事情，在 Rust 中是怎么解决这两个问题的呢？比如，在 C++里面，可能经常会写一些 callback ，但是我们并不想在 Rust 里面这么做，这样的话会遇到很多生命周期相关的问题。

Rust 允许自行实现 Runtime 来调度任务和执行 syscall；并提供了 Future 等统一的接口；另外内置了 async-await 语法糖从面向 callback 编程中解放出来。



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？

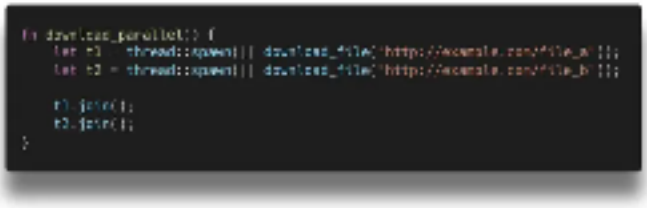
Example

这里从一个简单的例子入手，看一看这套系统到底是怎么工作的。

当并行下载两个文件时，在任何语言中都可以启动两个 Thread，分别下载一个文件，然后等待 thread 执行结束；但并不想为了 IO 等待启动多余的线程，如果需要等待 IO，我们希望这时线程可以去干别的，等 IO 就绪了再做就好。

这种基于事件的触发机制在 cpp 里面常常会以 callback 的形式遇见。Callback 会打断我们的连续逻辑，导致代码可读性变差，另外也容易在 callback 依赖的变量的生命周期上踩坑，比如在 callback 执行前提前释放了它会引用的变量。

但在 Rust 中只需要创建两个 task 并等待 task 执行结束即可。



Download 2 files in parallel with 2 threads

Download 2 files in parallel with 2 tasks

这个例子相比线程的话，异步 task 会高效很多，但编程上并没有因此复杂多少。

第二个例子，现在 mock 一个异步函数 do_http，这里直接返回一个 1，其实里面可能是一堆异步的远程请求；在此之上还想对这些异步函数做一些组合，这里假设是做两次请求，然后把两次的结果加起来，最后再加一个 1，就是这个例子里面的 sum 函数。通过 Async 和 Await 语法可以非常友好地把这些异步函数给嵌套起来。

rust 复制代码

```rust
#[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
}
```

这个过程和写同步函数是非常像的，也就说是在面向过程编程，而非面向状态编程。利用这种机制可以避开写一堆 callback 的问题，带来了编程的非常大的便捷性。

Async Await 背后的秘密

通过这两个例子可以得知 Rust 的异步是怎么用的，以及它写起来确实非常方便。那么它背后到底是什么原理呢?

rust 复制代码

```rust
#[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
}
```

```rust
#[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
            };
```


刚才的例子使用 Async + Await 编写，其生成结构最终实现 Future trait 。

Async + Await 其实是语法糖，可以在 HIR 阶段被展开为 Generator 语法，然后 Generator 又会在 MIR 阶段被编译器展开成状态机。

```
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: 8:29
    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
```

Future 抽象

Future trait 是标准库里定义的。它的接口非常简单，只有一个关联类型和一个 poll 方法。

rust 复制代码

```rust
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：Poll 方法顾名思义就是去推动状态机执行，给定一个任务，就会推动这个任务做状态转换。

2. 返回执行结果：

    1. 遇到了阻塞：Pending
    2. 执行完毕：Ready + 返回值

可以看出，异步 task 的本质就是实现 Future 的状态机。程序可以利用 Poll 方法去操作它，它可能会告诉程序现在遇到阻塞，或者说任务执行完了并返回结果。

既然有了 Future trait，我们完全可以手动地去实现 Future。这样一来，实现出来的代码要比 Async、Await 语法糖去展开的要易读。下面是手动生成状态机的样例。如果用 Async 语法写，可能直接一个 async 函数返回一个 1 就可以；我们手动编写需要自定义一个结构体，并为这个结构体实现 Future。

rust 复制代码

```rust
// 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)
    }
}
```

Async fn 的本质就是返回一个实现了 Future 的匿名结构，这个类型由编译器自动生成，所以它的

名字不会暴露给我们。而我们手动实现就定义一个 Struct DoHTTPFuture，并为它实现 Future，它的 Output 和 Async fn 的返回值是一样的，都是 i32 。这两种写法是等价的。

由于这里只需要立刻返回一个数字 1，不涉及任何等待，那么我们只需要在 poll 实现上立刻返回 Ready(1) 即可。 前面举了 sum 的例子，它做的事情是异步逻辑的组合：调用两次 do http，最后再把两个结果再加一起。这时候如果要手动去实现的话，就会稍微复杂一些，因为会涉及到两个 await 点。一旦涉及到 await，其本质上就变成一个状态机。

为什么是状态机呢？因为每次 await 等待都有可能会卡住，而线程此时是不能停止工作并等待在这里的，它必须切出去执行别的任务；为了下次再恢复执行前面任务，它所对应的状态必须存储下来。这里我们定义了 FirstDoHTTP 和 SecondDoHTTP 两个状态。实现 poll 的时候，就是去做一个 loop，loop 里面会 match 当前状态，去做状态转换。

rust 复制代码

```rust
// 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::0utput> {
        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 Pol::Pending;
```

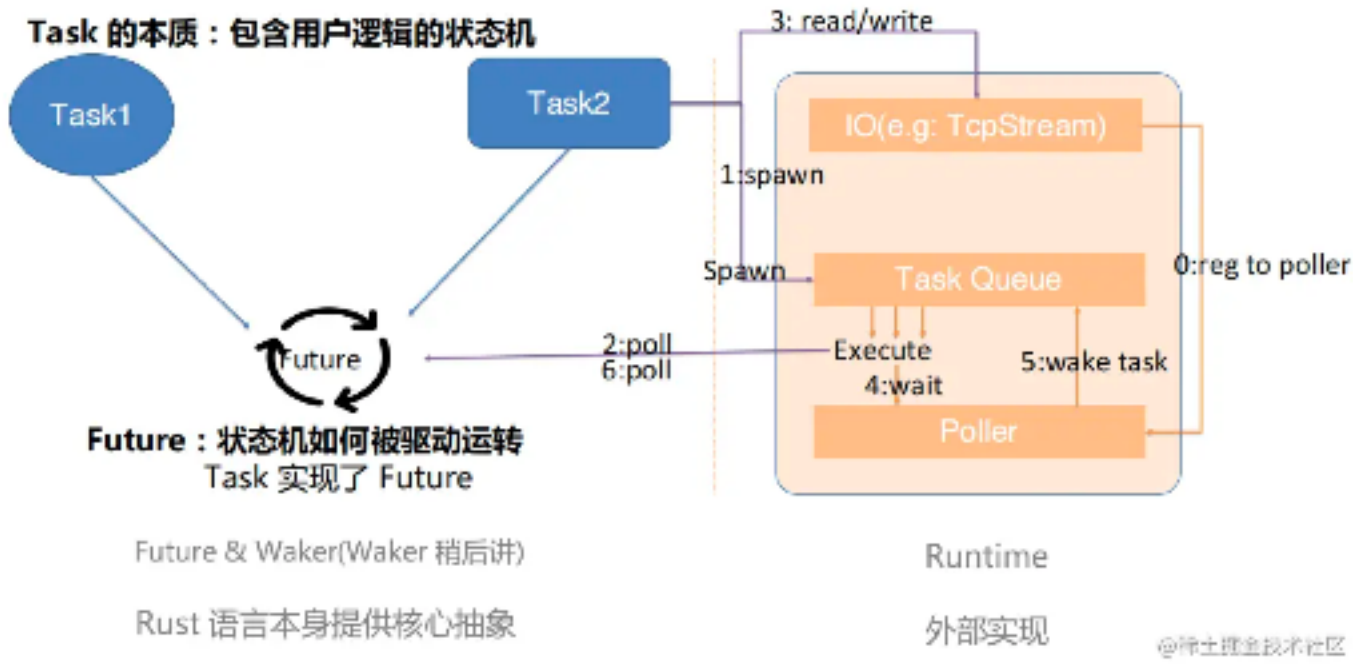
```
                        }
                    }
                }
                SumFuture::SecondDoHTTP(f,  prev_sum) => {
                    let pinned = unsafe { Pin::new_unchecked(f) };
                    return match pinned.poll( cx) {
                        Poll::Ready(r) => Poll::Ready(*prev_sum + r + 1),
                        Poll::Pending => Pol::Pending,
                    };
                }
            }
        }
    }
}
```

Task, Future 和 Runtime 的关系

我们这里以 TcpStream 的 Read/Write 为例梳理整个机制和组件的关系。

首先当我们创建 TCP stream 的时候，这个组件内部就会把它注册到一个 poller 上去，这个 poller 可以简单地认为是一个 epoll 的封装（具体使用什么 driver 是根据平台而异的）。

按照顺序来看，现在有一个 task，要把这个 task spawn 出去执行。那么 spawn 本质上就是把 task 放到了 runtime 的任务队列里，然后 runtime 内部会不停地从任务队列里面取出任务并且执行——执行就是推动状态机动一动，即调用它的 poll 方法，之后我们就来到了第2步。



我们执行它的 poll 方法，本质上这个 poll 方法是用户实现的，然后用户就会在这个 task 里面调用 TcpStream 的 read/write。这两个函数内部最终是调用 syscall 来实现功能的，但在执行 syscall 之前需要满足条件：这个 fd 可读/可写。如果它不满足这个条件，那么即便我们执行了 syscall 也只是拿到了 WOULD_BLOCK 错误，白白付出性能。初始状态下我们会设定新加入的 fd 本身就是可读/可写的，所以第一次 poll 会执行 syscall。当没有数据可读，或者内核的写 buffer 满了的时候，这个 syscall 会返回 WOULD_BLOCK 错误。在感知到这个错误后，我们会修改 readiness 记录，设定这个 fd 相关的读/写为不可读/不可写状态。这时我们只能对外返回 Pending。

之后来到第四步，当我们任务队列里面任务执行完了，我们现在所有任务都卡在 IO 上了，所有的 IO 可能都没有就绪，此时线程就会持续地阻塞在 poller 的 wait 方法里面，可以简单地认为它是一个 epoll_wait 一样的东西。当基于 io_uring 实现的时候，这可能对应另一个 syscall。

此时陷入 syscall 是合理的，因为没有任务需要执行，我们也不需要轮询 IO 状态，陷入 syscall 可以让出 CPU 时间片供同机的其他任务使用。如果有任何 IO 就绪，这时候我们就会从 syscall 返回，并且 kernel 会告诉我们哪些 fd 上的哪些事件已经就绪了。比如说我们关心的是某一个 FD 它的可读，那么这时候他就会把我们关心的 fd 和可读这件事告诉我们。 我们需要标记 fd 对应的 readiness 为可读状态，并把等在它上面的任务给叫醒。前面一步我们在做 read 的时候，有一个任务是等在这里的，它依赖 IO 可读事件，现在条件满足了，我们需要重新调度它。叫醒的本质就是把任务再次放到 task queue 里，实现上是通过 Waker 的 wake 相关方法做到的，wake 的处理

行为是 runtime 实现的，最简单的实现就是用一个 Deque 存放任务，wake 时 push 进去，复杂一点还会考虑任务窃取和分配等机制做跨线程的调度。

当该任务被 poll 时，它内部会再次做 TcpStream read，它会发现 IO 是可读状态，所以会执行 read syscall，而此时 syscall 就会正确执行，TcpStream read 对外会返回 Ready。

Waker

刚才提到了 Waker，接下来介绍 waker 是如何工作的。我们知道 Future 本质是状态机，每次推它转一转，它会返回 Pending 或者 Ready ，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?

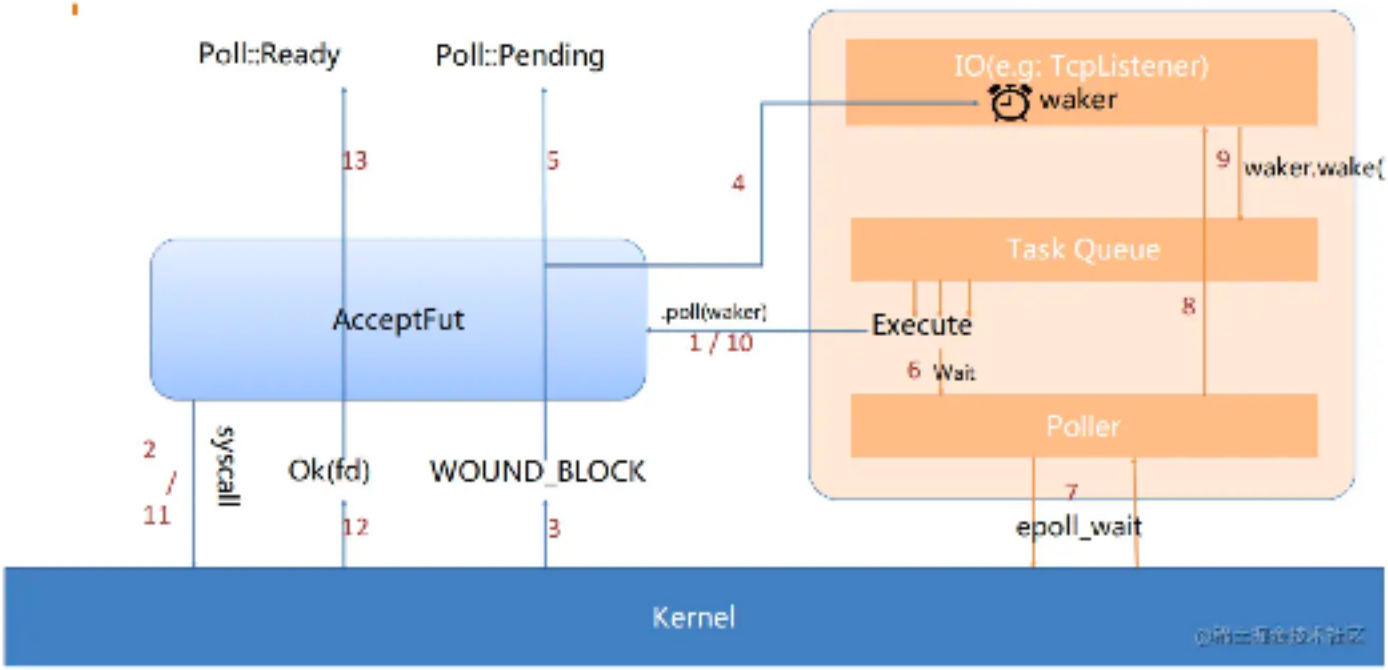
rust 复制代码

```rust
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 ()>,
}
```

Future trait 里面除了有包含自身状态机的可变以借用以外，还有一个很重要的是 Context，Context 内部当前只有一个 Waker 有意义，这个 waker 我们可以暂时认为它就是一个 trait object ，由 runtime 构造和实现。它实现的效果，就是当我们去 wake 这个 waker 的时候，会把任务重新加回任务队列，这个任务可能立刻或者稍后被执行。

举另一个例子来梳理整个流程。



用户使用 listener.accept() 生成 AcceptFut 并等待:

1. fut.await 内部使用 cx 调用 Future 的 poll 方法
2. poll 内部执行 syscall
3. 当前无连接拨入，kernel 返回 WOULD_BLOCK
4. 将 cx 中的 waker clone 并暂存于 TcpListener 关联结构内
5. 本次 poll 对外返回 Pending
6. Runtime 当前无任务可做，控制权交给 Poller
7. Poller 执行 epoll_wait 陷入 syscall 等待 IO 就绪
8. 查找并标记所有就绪 IO 状态
9. 如果有关联 waker 则 wake 并清除
10. 等待 accept 的 task 将再次加入执行队列并被 poll
11. 再次执行 syscall
12. 12/13. kernel 返回 syscall 结果，poll 返回 Ready

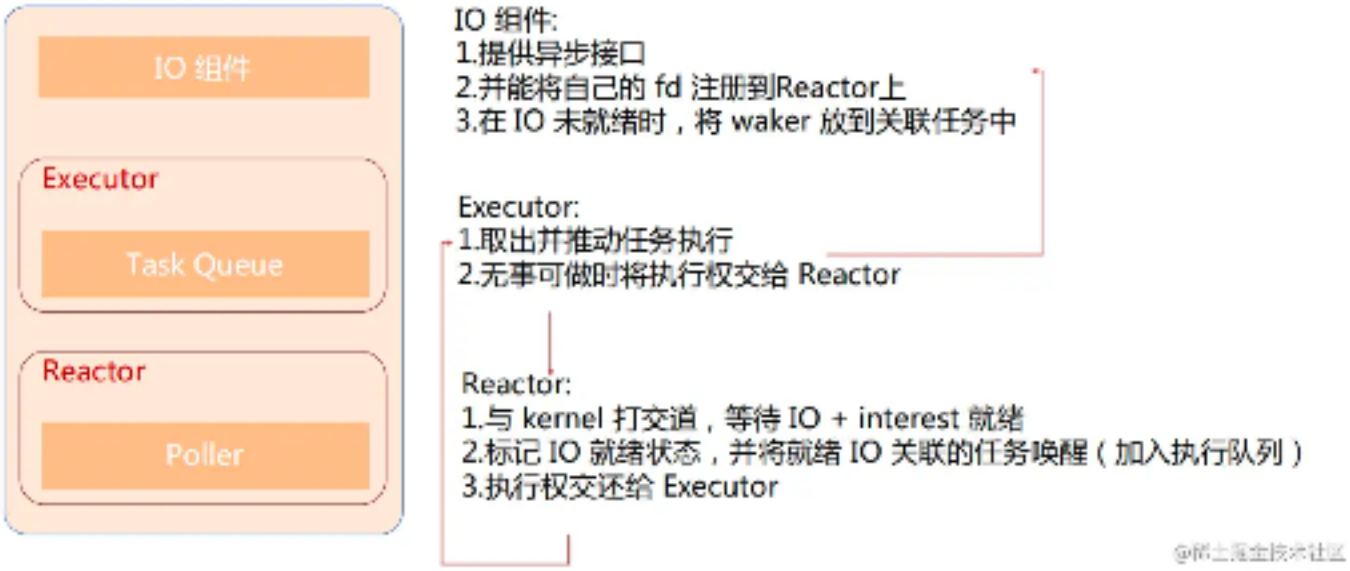
Runtime

1. 先从 executor 看起，它有一个执行器和一个任务队列，它的工作是不停地取出任务，推动任务运行，之后在所有任务执行完毕必须等待时，把执行权交给 Reactor。
2. Reactor 拿到了执行权之后，会与 kernel 打交道，等待 IO 就绪，IO就绪好了之后，我们需要

标记这个 IO 的就绪状态，并且把这个 IO 所关联的任务给唤醒。唤醒之后，我们的执行权又会重新交回给 executor 。在 executor 执行这个任务的时候，就会调用到 IO 组件所提供的一些能力。

3. IO 组件要能够提供这些异步的接口，比如说当用户想用 tcb stream 的时候，得用 runtime 提供的一个 TcpStream， 而不是直接用标准库的。第二，能够将自己的 fd 注册到 Reactor 上。第三，在 IO 没有就绪的时候，我们能把这个 waker 放到任务相关联的区域里。

整个 Rust 的异步机制大概就是这样。



三、Monoio 设计

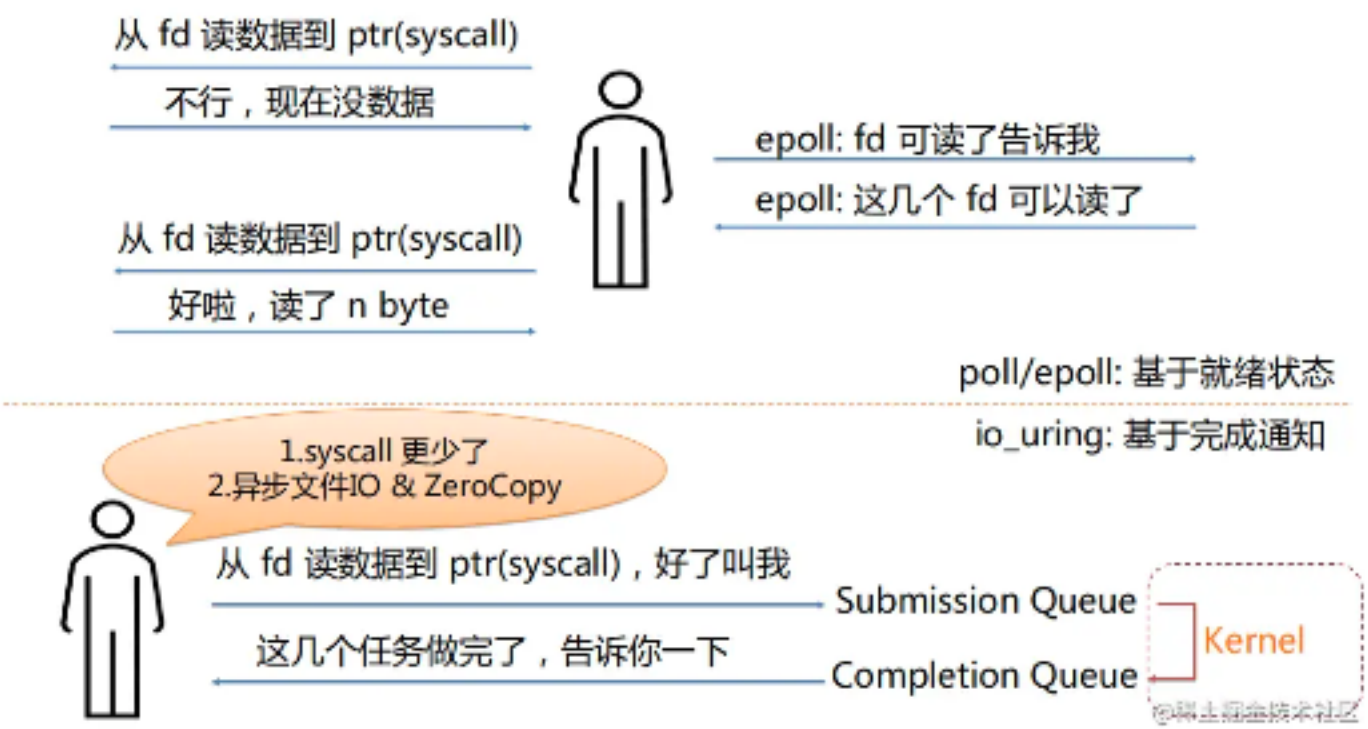
以下将会分为四个部分介绍 Monoio Runtime 的设计要点：

1. 基于 GAT(Generic associated types) 的异步 IO 接口；
2. 上层无感知的 Driver 探测与切换；
3. 如何兼顾性能与功能；
4. 提供兼容 Tokio 的接口

基于 GAT 的纯异步 IO 接口

首先介绍一下两种通知机制。第一种是和 epoll 类似的，基于就绪状态的一种通知。第二种是 io-

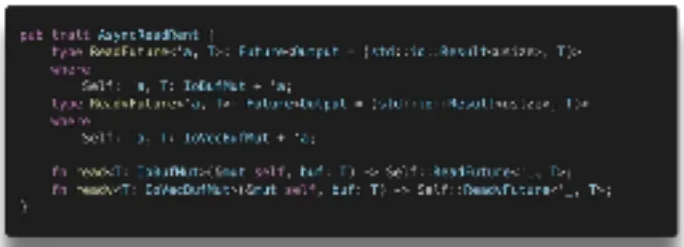
uring 的模式，它是一个基于"完成通知"的模式。



在基于就绪状态的模式下，任务会通过 epoll 等待并感知 IO 就绪，并在就绪时再执行 syscall。但在基于"完成通知"的模式下，Monoio 可以更懒：直接告诉 kernel 当前任务想做的事情就可以放手不管了。

io_uring 允许用户和内核共享两个无锁队列，submission queue 是用户态程序写，内核态消费；completion queue 是内核态写，用户态消费。通过 enter syscall 可以将队列中放入的 SQE 提交给 kernel，并可选地陷入并等待 CQE。

在 syscall 密集的应用中，使用 io_uring 可以大大减少上下文切换次数，并且 io_uring 本身也可以减少内核中数据拷贝。



这两种模式的差异会很大程度上影响 Runtime 的设计和 IO 接口。在第一种模式下，等待时是不需要持有 buffer 的，只有执行 syscall 的时候才需要 buffer，所以这种模式下可以允许用户在真正调用 poll 的时候（如 poll_read）传入 &mut Buffer；而在第二种模式下，在提交给 kernel 后，kernel 可以在任何时候访问 buffer，Monoio 必须确保在该任务对应的 CQE 返回前 Buffer 的有效性。

如果使用现有异步 IO trait（如 tokio/async-std 等），用户在 read/write 时传入 buffer 的引用，可能会导致 UAF 等内存安全问题：如果在用户调用 read 时将 buffer 指针推入 uring SQ，那么如果用户使用 read(&mut buffer) 创建了 Future，但立刻 Drop 它，并 Drop buffer，这种行为不违背 Rust 借用检查，但内核还将会访问已经释放的内存，就可能会踩踏到用户程序后续分配的内存块。 所以这时候一个解法，就是去捕获它的所有权，当生成 Future 的时候，把所有权给 Runtime，这时候用户无论如何都访问不到这个 buffer 了，也就保证了在 kernel 返回 CQE 前指针的有效性。这个解法借鉴了 tokio-uring 的做法。 Monoio 定义了 AsyncReadRent 这个 trait。所谓的 Rent，即租借，相当于是 Runtime 先把这个 buffer 从用户手里拿过来，待会再还给用户。这里的 type read future 是带了生命周期泛型的，这个泛型其实是 GAT 提供了一个能力，现在 GAT 已经稳定了，已经可以在 stable 版本里面去使用它了。当要实现关联的 Future 的时候，借助 TAIT 这个 trait 可以直接利用 async-await 形式来写，相比手动定义 Future 要方便友好很多，这个 feature 目前还没稳定（现在改名叫 impl trait in assoc type 了）。

当然，转移所有权会引入新的问题。在基于就绪状态的模式下，取消 IO 只需要 Drop Future 即可；这里如果 Drop Future 就可能导致连接上数据流错误（Drop Future 的瞬间有可能 syscall 刚好已经成功），并且一个更严重的问题是一定会丢失 Future 捕获的 buffer。针对这两个问题 Monoio 支持了带取消能力的 IO trait，取消时会推入 CancelOp，用户需要在取消后继续等待原 Future 执行结束（由于它已经被取消了，所以会预期在较短的时间内返回），对应的 syscall 可能执行成功或失败，并返还 buffer。

上层无感知的 Driver 探测和切换

第二个特性是支持上层无感知的 Driver 探测和切换。

rust 复制代码

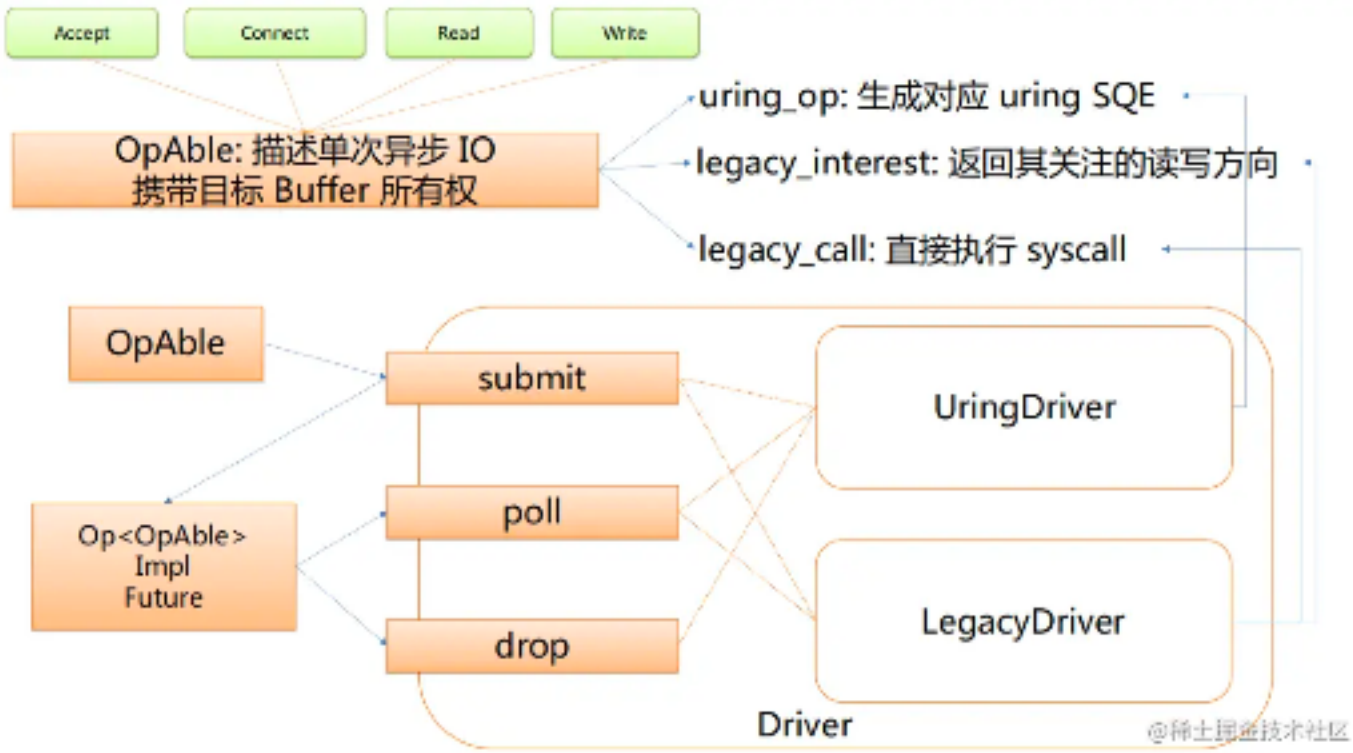
```rust
trait OpAble {
    fn uring_op(&mut self) -> io_uring::squeue::Entry;
    fn legacy_interest(&self) -> Option<(ready::Diirection, usize)>;
```

```
    fn legacy_call(&mut self) -> io::Result<u32>;
}
```

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

具体来说，比如想做 accept、connect 或者 read、write 之类的，这些 op 是实现了 OpAble 的，实际对应这三个 fn：

1. uring_op：生成对应 uring SQE
2. legacy_interest：返回其关注的读写方向
3. legacy_call：直接执行syscall



整个流程会将一个实现了 opable 的结构 submit 到的 driver 上，然后会返回一个实现了 future 的东西，之后它 poll 的时候和 drop 的时候具体地会分发到两个 driver 实现中的一个，就会用这三个函数里面的一个或者两个。

性能

性能是 Monoio 的出发点和最大的优点。除了 io_uring 带来的提升外，它设计上是一个 thread-per-core 模式的 Runtime。

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

高性能主要源于两个方面：

1. Runtime内部高性能：基本等价于裸对接syscall

2. 用户代码高性能：结构尽量 thread local 不跨线程

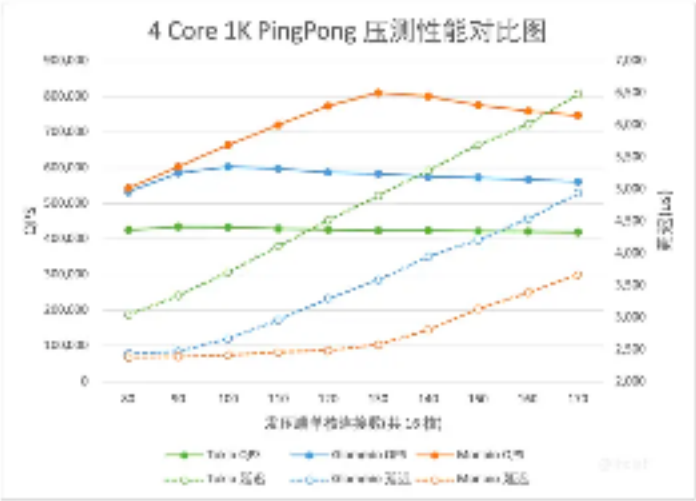
任务窃取和 thread-per-core 两种机制的对比：

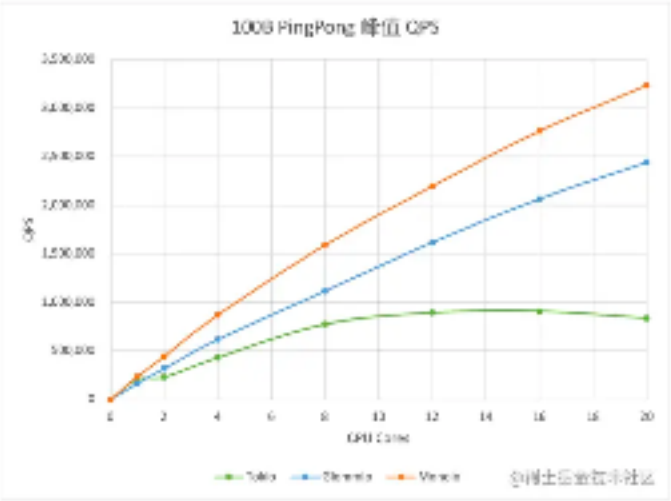
如果用 tokio 的话，可能某一个线程上它的任务非常少，可能已经空了，但是另一个线程上任务非常多。那么这时候比较闲的线程就可以把任务从比较忙的任务上偷走，这一点和 Golang 非常像。这种机制可以较充分的利用 CPU，应对通用场景可以做到较好的性能。

但跨线程本身会有开销，多线程操作数据结构时也会需要锁或无锁结构。但无锁也不代表没有额外开销，相比纯本线程操作，跨线程的无锁结构会影响缓存性能，CAS 也会付出一些无效 loop。除此之外，更重要的是这种模式也会影响用户代码。

举个例子，我们内部需要一个 SDK 去收集本程序的一些打点，并把这些打点聚合之后去上报。在基于 tokio 的实现下，要做到极致的性能就比较困难。如果在 thread-per-core 结构的 Runtime 上，我们完全可以将聚合的 Map 放在 thread-local 中，不需要任何锁，也没有任何竞争问题，只需要在每个线程上启动一个任务，让这个任务定期清空并上报 thread local 中的数据。而在任务可能跨线程的场景下，我们就只能用全局的结构来聚合打点，用一个全局的任务去上报数据。聚合用的数据结构就很难不使用锁。

所以这两种模式各有各的优点，thread-per-core 模式下对于可以较独立处理的任务可以达到更好的性能。共享更少的东西可以做到更好的性能。但是 thread-per-core 的缺点是在任务本身不均匀的情况下不能充分利用 CPU。对于特定场景，如网关代理等，thread-per-core 更容易充分利用硬件性能，做到比较好的水平扩展性。当前广泛使用 nginx 和 envoy 都是这种模式。



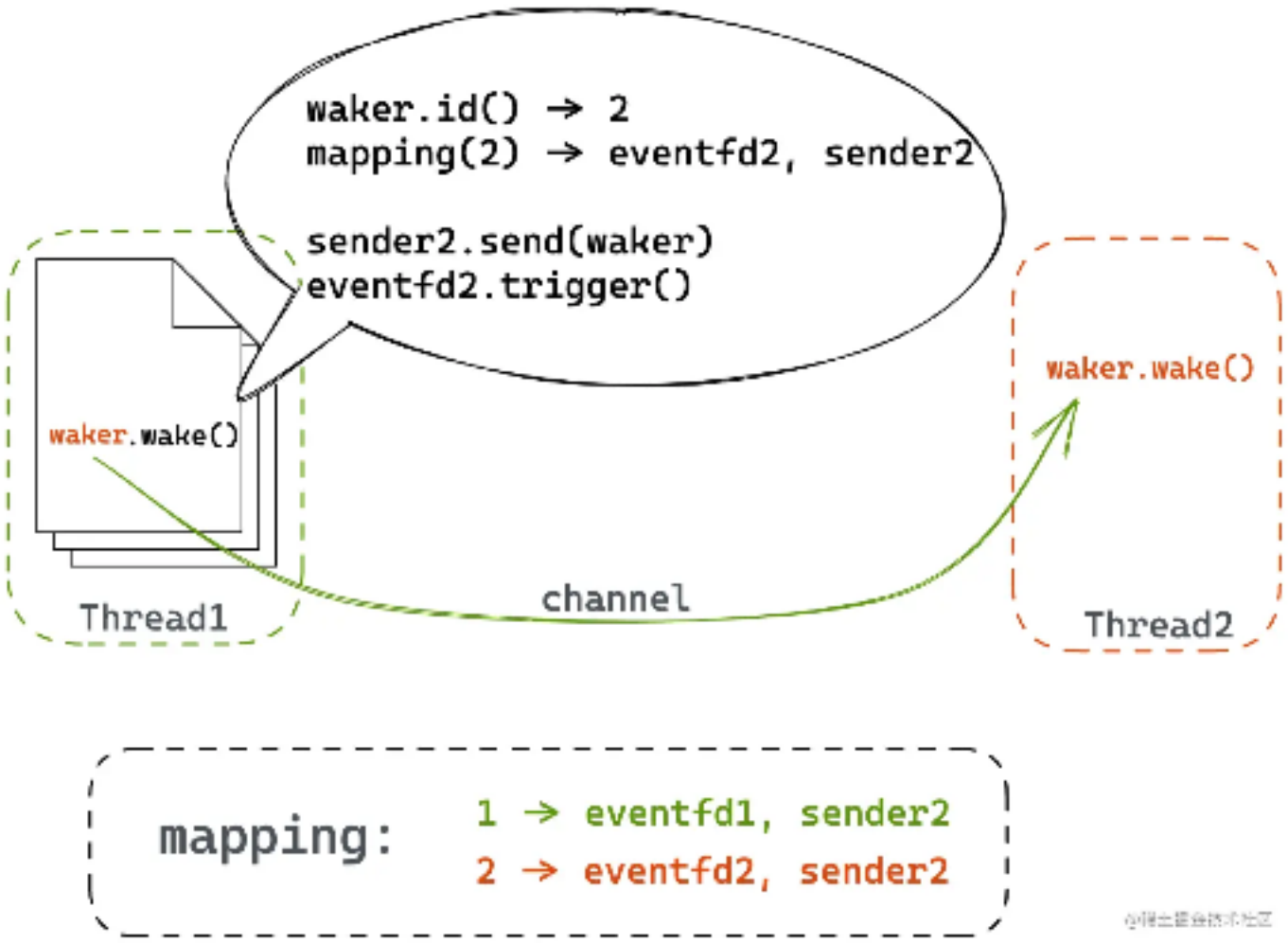


我们做了一些 benchmark，Monoio 的性能水平扩展性是非常好的。当 CPU 核数增加的时候，只需要增加对应的线程就可以了。

功能性

Thread-per-core 不代表没有跨线程能力。用户依旧可以使用一些跨线程共享的结构，这些和 Runtime 无关；Runtime 提供了跨线程等待的能力。

任务在本线程执行，但可以等待其他线程上的任务，这个是一个很重要的能力。举例来说，用户需要用单线程去拉取远程配置，并下发到所有线程上。基于这个能力，用户就可以非常轻松地实现这个功能。

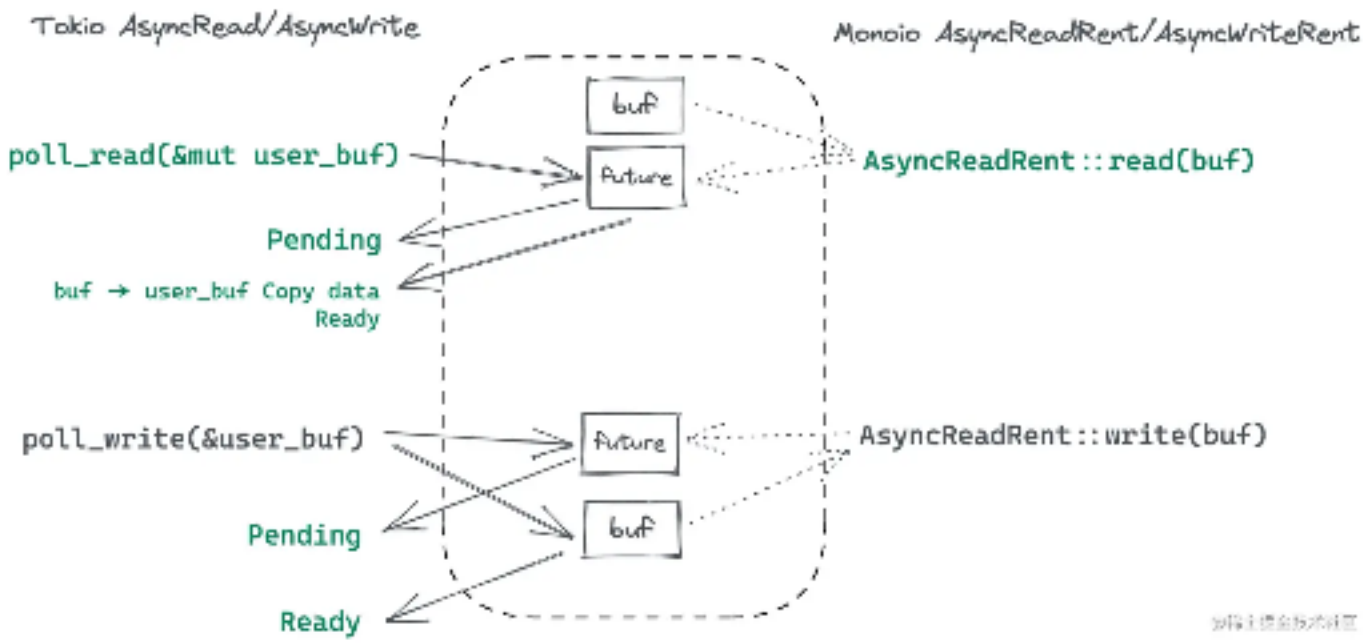


跨线程等待的本质是在别的线程唤醒本线程的任务。实现上我们在 Waker 中标记任务的所属权，如果当前线程并不是任务所属线程，那么 Runtime 会通过无锁队列将任务发送到其所属线程上；如果此时目标线程处于休眠状态（陷入 syscall 等待 IO），则利用事先安插的 eventfd 将其唤醒。唤醒后，目标线程会处理跨线程 waker 队列。

除了提供跨线程等待能力外，Monoio 也提供了 spawn_blocking 能力，供用户执行较重的计算逻辑，以免影响到同线程的其他任务。

兼容接口

由于目前很多组件（如 hyper 等）绑定了 tokio 的 IO trait，而前面讲了由于地层 driver 的原因这两种 IO trait 不可能统一，所以生态上会比较困难。对于一些非热路径的组件，需要允许用户以兼容方式使用，即便付出一些性能代价。



rust 复制代码

```rust
// tokio way
let tcp = tokio::net::TcpStream::connect("1.1.1.1:80").await.unwrap();
// monoio way(with monoio-compat)
let tcp = monoio_compat::StreamWrapper::new(monoio_tcp);
let monoio_tcp = monoio::net::TcpStream::connect("1.1.1.1:80").await.unwrap();
// both of them implements tokio:: io::AsyncReadd and tokio:: io: AsyncWrite
```

我们提供了一个 Wrapper，内置了一个 buffer，用户使用时需要多付出一次内存拷贝开销。通过这种方式，我们可以为 monoio 的组件包装出 tokio 的兼容接口，使其可以使用兼容组件。

四、Runtime 对比&应用

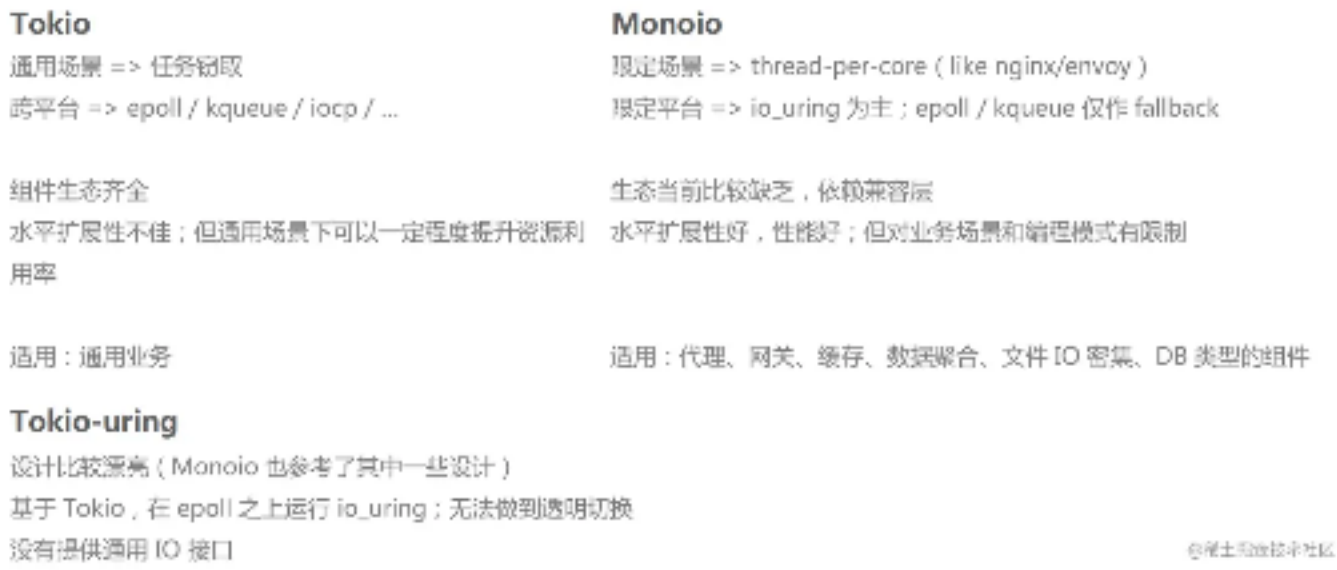
这部分介绍 runtime 的一些对比选型和应用。

前面已经提到了关于均匀调度和 thread-per-core 的一些对比，这里主要说一下应用场景。对于较大量的轻任务，thread-per-core 模式是适合的。特别是代理、网关和文件 IO 密集的应用，使用 Monoio 就非常合适。

还有一点，Tokio 致力于一个通用跨平台，但是 Monoio 设计之初就是为了极致性能，所以是期望以 io_uring 为主的。虽然也可以支持 epoll 和 kqueue，但仅作 fallback。比如 kqueue 其实就是为了让用户能够在 Mac 上去开发的便利性，其实不期望用户真的把它跑在这（未来将支持

Windows）。

生态部分，Tokio 的生态是比较全的，Monoio 的比较缺乏，即便有兼容层，兼容层本身是有开销的。Tokio 有任务窃取，可以在较多的场景表现很好，但其水平扩展性不佳。Monoio 的水平扩展就比较好，但是对这个业务场景和编程模型其实是有限制的。所以 Monoio 比较适合的一些场景就是代理、网关还有缓存数据聚合等。以及还有一些会做文件 io 的，因为 io_uring 对文件 io 非常好。如果不用 io_uring 的话，在 Linux 下其实是没有真异步的文件 io 可以用的，只有用 io_uring 才能做到这一点。还适用于这种文件 io 比较密集的，比如说像 DB 类型的组件。



Tokio-uring 其实是一个构建在 tokio 之上的一层，有点像是一层分发层，它的设计比较漂亮，我们也参考了它里面的很多设计，比如说像那个传递所有权的这种形式。但是它还是基于 tokio 做的，在 epoll 之上运行 uring，没有做到用户透明。当组件在实现时，只能在使用 epoll 和使用 uring 中二选一，如果选择了 uring，那么编译产物就无法在旧版本 linux 上运行。而 Monoio 很好的支持了这一点，支持动态探测 uring 的可用性。

Monoio 应用

1. Monoio Gateway: 基于 Monoio 生态的网关服务，我们优化版本 Benchmark 下来性能优于 Nginx；
2. Volo: CloudWeGo Team 开源的 RPC 框架，目前在集成中，PoC 版本性能相比基于 Tokio 提升 26%

我们也在内部做了一些业务业务试点，未来我们会从提升兼容性和组件建设上入手，就是让它更好用。

项目地址

- Monoio GitHub： github.com/bytedance/m...
- CloudWeGo：github.com/cloudwego
- CloudWeGo 官网：www.cloudwego.io

分类： 开发工具 标签： 开源

相关小册

IntelliJ IDE 插件开发指南

洪进锋 839购买

¥14.95 ~~¥29.9~~ 首单券后价

VIP Neovim 配置实战：从0到1打造自己的IDE

nshen LV.2 2316购买

¥14.95 ~~¥29.9~~ 首单券后价

评论

看完啦， 登录 分享一下感受吧～

相关推荐

Go学堂 6小时前 后端 Go 开源

「Go工具箱」go-mask：一个对数据脱敏处理的包

191 点赞 评论

萌萌哒草头将军 23小时前 开源 前端 React.js

dumi踩坑实录

550 2 评论

HelloGitHub 2天前 GitHub 开源

《HelloGitHub》第 85 期

1686 4 2

HelloGitHub 6天前 GitHub 开源

有了这些 AI 工具，健康和财富兼得「GitHub 热点速览」

3091 22 评论

字节跳动云原生计算 4天前 开源 分布式 机器学习

字节跳动正式开源分布式训练调度框架 Primus

1923 点赞 评论

YongGit 3天前 前端 开源 Vue.js

Vue.js 推出 "官方考试"，技能认证

826 2 12

天行无忌 1天前 ChatGPT OpenAI 开源

HuggingChat：ChatGPT开源机器人的替代品

155 2 评论

小嘴叭叭儿 2天前 Go 开源

go-parallel: 一个并发任务处理工具

138 1 评论

前端小付 11天前 前端 开源 Icon

开发中使用iconfont太麻烦了，于是我写了这款vscode插件，支持react和vue。

2536 72 21

HelloGitHub 13天前 GitHub 开源

一款能"干掉" ChatGPT 的应用「GitHub 热点速览」

3874 61 1

马丁玩编程 1月前 开源

发现一个商城开源项目，谷粒商城外又多了个选择

1.2w 213 14

DevUI团队 26天前 前端 JavaScript 开源

💥前端组件库视觉版本大更新，这波DevUI值得收藏！

7692 113 56

DevUI团队　10天前　前端　开源　动效

🎯DevUI这几种动效，教你打造有温度的人机交互体验

1867　32　11

字节跳动技术团队　24天前　后端　开源

字节跳动开源 Shmipc：基于共享内存的高性能 IPC

7239　20　1

HashTang　13天前　ChatGPT　人工智能　掘金·金石计划

精讲批量无限制注册&升级 ChatGPT 终极攻略｜建议收藏

2422　13　13

xbhog　7天前　后端　Java　开源

【源码分析】XXL-JOB执行器的注册流程

834　5　评论

A等天晴　29天前　程序员　开源

13个程序员常用开发工具用途推荐整理

6923　25　6

橘子没了　22天前　Android　开源　APP

时隔2年终于开源了基于RecyclerView的阅读器动画方案

2819　92　21

花生Peadar　4月前　JavaScript　Webpack　开源

维护8年的前端开源项目长啥样?

4.3w　326　54