基于完成的 Rust 异步：compio 项目及其经验

Berrysoft（王宇逸）

2024 年 4 月 20 日

清华大学工程物理系

你的 CPU 占用比较松弛……

你的 CPU 占用比较松弛，但你的线程呢又很好地弥补了这一部分。如果要做高性能，把线程去除的话，可能会显得你的卡顿就比较大，会有一些卡死变成僵尸的情况。现在最好的办法就是你去掉线程的同时，开一个异步运行时。

常见误区

所谓的异步是针对 IO 而言的，计算密集型程序并不需要异步来加速。

一个简单的 IO 场景：隧道

A 和 B 希望通过一个~~中间人~~隧道通信。

图 1: 简化版的隧道。

A 和 B 在任意时刻均有可能向隧道通信，希望向另一端发送消息。隧道并不清楚二者的通信先后顺序。

简单的同步开发

由于不清楚 A 和 B 发送消息的先后顺序，一个简单的思路是开两个线程。

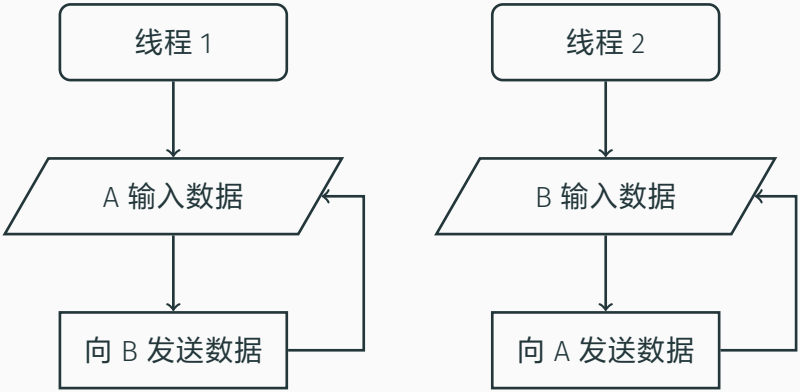


图 2: 多线程并发

简单的同步开发复盘

同步代码的缺点？

- 线程开销：创建、调度、切换。
- 软件的 CPU 利用率不高：大量时间用来做阻塞的系统调用。

简单的同步开发复盘

同步代码的缺点？

- 线程开销：创建、调度、切换。
- 软件的 CPU 利用率不高：大量时间用来做阻塞的系统调用。

阻塞的来源？

- Socket 尚未就绪。
  - 读：没有数据传入。
  - 写：缓冲区已满，需要发送。
- 数据在内核态与用户态之间复制。

简单的同步开发复盘

同步代码的缺点？

- 线程开销：创建、调度、切换。
- 软件的 CPU 利用率不高：大量时间用来做阻塞的系统调用。

阻塞的来源？

- Socket 尚未就绪。
  - 读：没有数据传入。
  - 写：缓冲区已满，需要发送。
- 数据在内核态与用户态之间复制。

如何消除阻塞？

- Socket 提供了一种非阻塞（non-blocking）模式。如果**尚未就绪**，系统调用会返回 EAGAIN 或 EWOULDBLOCK。

简单的异步开发

利用 `poll`、`epoll`、`kqueue` 可以在主流系统中等待多个 socket 就绪。

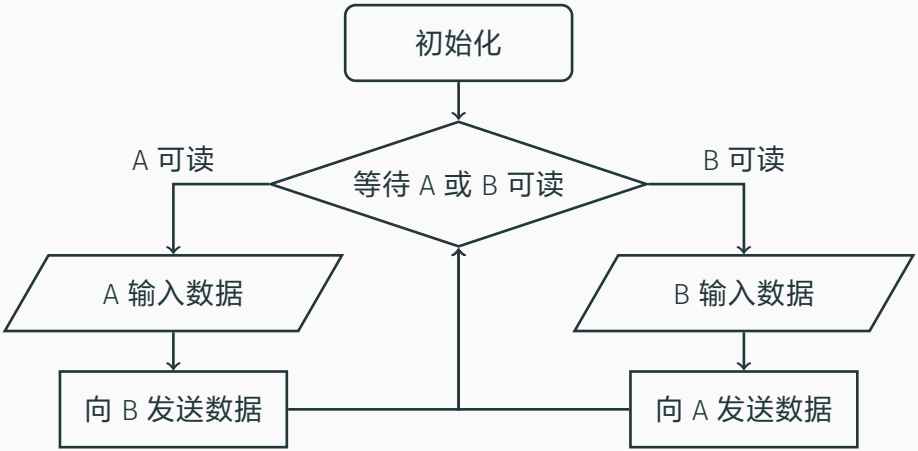


图 3: 单线程的简单异步，这样的模式也称作 reactor。

简单的异步开发复盘

异步开发的优点？

- 单线程，减少了线程开销。
- CPU 利用率提高。

简单的异步开发复盘

异步开发的优点？

- 单线程，减少了线程开销。
- CPU 利用率提高。

存在的提升空间？

- 异步不充分：发送数据前也可以先等待 socket 可写。参考：Corkscrew。
- 各种 poll 越写越麻烦，想要像同步代码一样写：需要一个异步框架。

简单的异步开发复盘

异步开发的优点？

- 单线程，减少了线程开销。
- CPU 利用率提高。

存在的提升空间？

- 异步不充分：发送数据前也可以先等待 socket 可写。参考：Corkscrew。
- 各种 poll 越写越麻烦，想要像同步代码一样写：需要一个异步框架。

……是不是忘了什么？

- 常规文件怎么办？
- 内核态与用户态传输数据也要异步！我们需要 proactor。

等待完成而非等待就绪：IOCP、io-uring、AIO

等待完成？

- 完全的异步：让内核完成任务，而非等待 IO 就绪后做同步的系统调用。
- 文件 IO：传统的 non-blocking 针对 socket 设计，普通文件始终是就绪的。然而 NFS 上的文件显然并不永远就绪。
- 并发任务：一次性等待多个相同文件的读取任务。
- 不局限于 IO：任何需要阻塞的任务都可以异步并等待。

等待完成而非等待就绪：IOCP、io-uring、AIO

等待完成？

- 完全的异步：让内核完成任务，而非等待 IO 就绪后做同步的系统调用。
- 文件 IO：传统的 non-blocking 针对 socket 设计，普通文件始终是就绪的。然而 NFS 上的文件显然并不永远就绪。
- 并发任务：一次性等待多个相同文件的读取任务。
- 不局限于 IO：任何需要阻塞的任务都可以异步并等待。

异步的必要性

- 考虑一个延迟 $114.514\,\mathrm{ms}$ 的服务器上的 NFS。

等待完成而非等待就绪：IOCP、io-uring、AIO

等待完成？

- 完全的异步：让内核完成任务，而非等待 IO 就绪后做同步的系统调用。
- 文件 IO：传统的 non-blocking 针对 socket 设计，普通文件始终是就绪的。然而 NFS 上的文件显然并不永远就绪。
- 并发任务：一次性等待多个相同文件的读取任务。
- 不局限于 IO：任何需要阻塞的任务都可以异步并等待。

异步的必要性

- 考虑一个延迟 $114.514\,\mathrm{ms}$ 的服务器上的 NFS。
- 创建 socket 是否需要异步？考虑在 NFS 的下面创建 Unix domain socket……
  - 并不鼓励这样的行为。
  - Socket 的类型有很多，可能存在会阻塞创建任务的奇妙驱动。

Compio：基于完成的 Rust 异步运行时

Compio[1] 跨平台，支持 Windows IOCP、Linux io-uring，并在其它情况下回退到传统的 non-blocking poll 机制。

```rust
use compio::{fs::File, io::AsyncReadAtExt};
#[compio::main]
async fn main() {
    let file = File::open("Cargo.toml").await.unwrap();
    let (read, buffer) = file
        .read_to_end_at(Vec::with_capacity(1024), 0)
        .await
        .unwrap();
    assert_eq!(read, buffer.len());
    let buffer = String::from_utf8(buffer).unwrap();
    println!("{}", buffer);
}
```

[1] https://github.com/compio-rs/compio

“妈妈给你把红包保管起来……”

Compio 的 IO API 需要数据的所有权。

std API	compio API
`let n = socket.read(&mut buffer)?;`	`let (n, buffer) = socket.read(buffer).await?;`

“妈妈给你把红包保管起来……”

Compio 的 IO API 需要数据的所有权。

std API	compio API
`let n = socket.read(&mut buffer)?;`	`let (n, buffer) = socket.read(buffer).await?;`

在同步 API 与基于就绪的异步 API 中，数据传输始终是同步的，且发生在最后。用户态和内核态不会发生数据竞争，任务也可以通过 drop 安全取消。

“妈妈给你把红包保管起来……”

Compio 的 IO API 需要数据的所有权。

std API	compio API
`let n = socket.read(&mut buffer)?;`	`let (n, buffer) = socket.read(buffer).await?;`

在同步 API 与基于就绪的异步 API 中，数据传输始终是同步的，且发生在最后。用户态和内核态不会发生数据竞争，任务也可以通过 `drop` 安全取消。

基于完成的异步 API，不确定数据传输发生于何时，也不能保证任务正常取消。数据的生存期应当由内核控制。

“妈妈给你把红包保管起来……”

Compio 的 IO API 需要数据的所有权。

std API	compio API
`let n = socket.read(&mut buffer)?;`	`let (n, buffer) = socket.read(buffer).await?;`

在同步 API 与基于就绪的异步 API 中，数据传输始终是同步的，且发生在最后。用户态和内核态不会发生数据竞争，任务也可以通过 `drop` 安全取消。

基于完成的异步 API，不确定数据传输发生于何时，也不能保证任务正常取消。数据的生存期应当由内核控制。

妈妈随时可能用钱，所以把红包保管起来。即使你忘记了（forget）、搬家了（move）、消失了（drop），红包里的钱还在。

红包能不能放我这里？

传递所有权是很麻烦的，能不能像 std 或者 tokio 一样只传递数据的引用？在现有的 Rust 语言下不能，但是也许我们可以改进它。

红包能不能放我这里？

传递所有权是很麻烦的，能不能像 std 或者 tokio 一样只传递数据的引用？在现有的 Rust 语言下不能，但是也许我们可以改进它。

`!Move` 或许是一种解决思路

- 将 compio 的 future 设计为 `!Move`，让它无法手动 `drop`，从而避免在内核 IO 结束前释放数据，造成 UAF。
- `!Move` 也意味着无法调用 `forget`，也无法用 `Rc` 或 `Arc` 构造循环引用导致内存泄露，从而导致多个任务拿到数据的可变引用，造成数据竞争。
- 这样的任务一旦创建只能等待完成。运行时可以提供 API 使其安全取消。

红包能不能放我这里？

传递所有权是很麻烦的，能不能像 std 或者 tokio 一样只传递数据的引用？在现有的 Rust 语言下不能，但是也许我们可以改进它。

`!Move` 或许是一种解决思路

- 将 compio 的 future 设计为 `!Move`，让它无法手动 `drop`，从而避免在内核 IO 结束前释放数据，造成 UAF。
- `!Move` 也意味着无法调用 `forget`，也无法用 `Rc` 或 `Arc` 构造循环引用导致内存泄露，从而导致多个任务拿到数据的可变引用，造成数据竞争。
- 这样的任务一旦创建只能等待完成。运行时可以提供 API 使其安全取消。

Async drop 无法解决问题

- Async drop 禁止了手动 `drop`，一定要等待任务完成；但是没有禁止 `forget`，不传递所有权仍然会导致数据竞争。

红包还给你，钱用完了

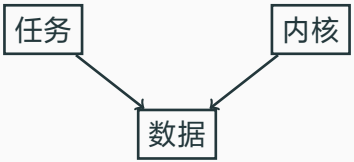
在发送取消请求时，任务可能已经完成。为了安全取消，下面是一种我喜欢的设计思路。

```rust
pub trait CancellableFuture {
    type Output;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll<Result<Self::Output>>;

    fn poll_cancel(self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll<Result<()>>;
}
```

Async drop 也无法解决安全取消问题，用户无法拿到取消之后的任务结果。

Compio 的生存期管理方式

Compio 中数据保存一个初始值为 2 的引用计数，记录 future 和内核对它的引用。

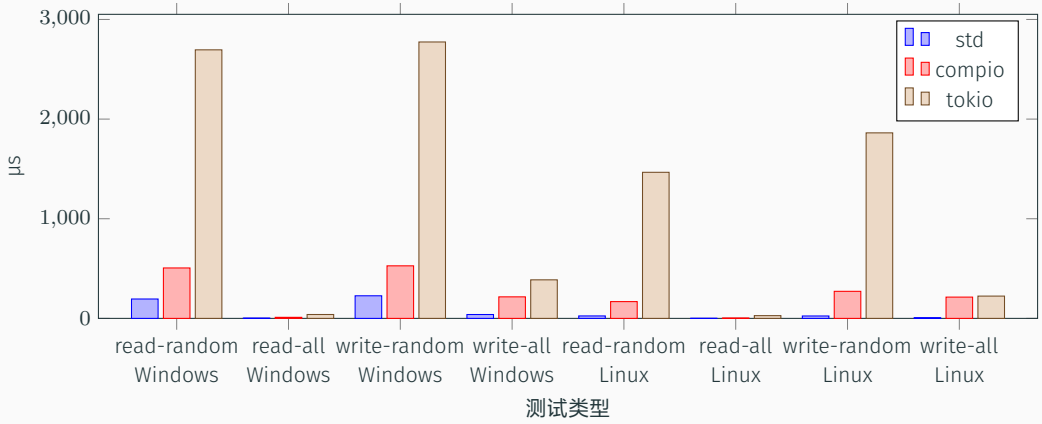


如果任务被取消，或者任务完成，引用计数均 -1。这样做引入了额外的装箱，因为涉及到对于数据的类型擦除。

不解决前述的  问题，装箱无法避免。

文件 IO 性能

众所周知，tokio 的文件 IO 性能极差。这是因为它大量使用了 `spawn_blocking` 来模拟异步操作。Compio 在 Windows 与 Linux 的文件 IO 轻松碾压 tokio。



高层 API 和底层 API

Compio 同时提供了高层和底层 API，方便使用者自行组合。如果只想要用一个 proactor，可以用 `compio-driver`，手动调度异步任务。

基于高层 API，我们自己也探索了一些更花样的拓展。

- cyper 这是一个基于 compio 和 hyper 项目的 HTTP 客户端。不过由于 hyper 太复杂，还处在 beta 阶段。
- winio 这是我们的愚人节玩笑。这是基于 compio 的 native GUI 库，旨在将 GUI 的事件机制和 runtime 的循环结合起来。
  - IO 不会导致 UI 卡死了，但是 UI 重绘时间太长可能导致 IO 完成了但是无法调度。

IOCP 就是个 channel

IOCP 在漫长的历史中已经被各路资料妖魔化了，然而其本质很简单：一个 MPMC channel。内核完成任务之后向 channel 发送 pack，用户可以在用户态接收。

IOCP 就是个 channel

IOCP 在漫长的历史中已经被各路资料妖魔化了，然而其本质很简单：一个 MPMC channel。内核完成任务之后向 channel 发送 pack，用户可以在用户态接收。

然而，Windows API 真正的问题在于：

- IOCP 并非唯一的异步机制。
  - Win32 Event 事件机制，提供的 API 可以同时等待最多 64 个事件。广泛用于 mutex、信号量、线程、进程等对象的等待。
  - 回调机制。一部分使用了线程池的 API 使用回调。
  - APC。类似 Unix 的信号，但是只在允许的位置打断线程。
  - Message Queue。Win32 GUI 的主循环。
  - IoRing。Windows 11 新增用于文件 IO 的异步 API。

IOCP 就是个 channel

IOCP 在漫长的历史中已经被各路资料妖魔化了，然而其本质很简单：一个 MPMC channel。内核完成任务之后向 channel 发送 pack，用户可以在用户态接收。

然而，Windows API 真正的问题在于：

- IOCP 并非唯一的异步机制。
  - Win32 Event 事件机制，提供的 API 可以同时等待最多 64 个事件。广泛用于 mutex、信号量、线程、进程等对象的等待。
  - 回调机制。一部分使用了线程池的 API 使用回调。
  - APC。类似 Unix 的信号，但是只在允许的位置打断线程。
  - Message Queue。Win32 GUI 的主循环。
  - IoRing。Windows 11 新增用于文件 IO 的异步 API。
- IOCP **无法取关**。
  - IOCP handle 保存在文件或 socket handle 之中，只能保存一次，无法更改或删除。
  - 所以最自然的使用方法是所有工作线程均使用同一个 IOCP 调度。

禁酒令🍷❓

Tokio 在经历了痛苦的 IOCP 适配之后，改用 wepoll 来满足其基于就绪的设计，然而它使用的文件路径 `\Device\Afd` 是 undocumented API[2]，在 wine 上并未实现。在 mio 的某个版本之后，tokio 就既不支持 wine 也无法在 UWP 上使用。

Compio 力求解决这一痛点。在使用了基于完成的异步设计之后，成功基于 IOCP 实现了运行时，没有使用任何 undocumented API，能够满足 UWP 的要求。

[2] `https://notgull.net/device-afd/`

禁酒令🍷❓

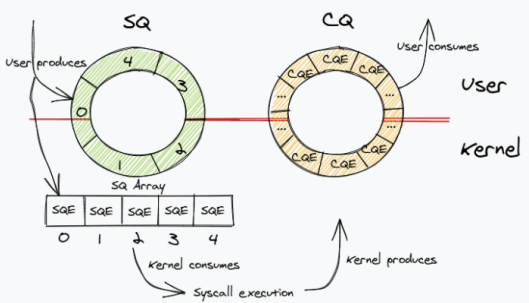
Tokio 在经历了痛苦的 IOCP 适配之后，改用 wepoll 来满足其基于就绪的设计，然而它使用的文件路径 `\Device\Afd` 是 undocumented API[2]，在 wine 上并未实现。在 mio 的某个版本之后，tokio 就既不支持 wine 也无法在 UWP 上使用。

Compio 力求解决这一痛点。在使用了基于完成的异步设计之后，成功基于 IOCP 实现了运行时，没有使用任何 undocumented API，能够满足 UWP 的要求。

然而 compio 还是在 wine 上无法运行，（也许）因为使用了 thread-local。

[2] `https://notgull.net/device-afd/`

仍需要进化的 io-uring



- io-uring 的 ring buffer 是个固定大小的循环链表。如果 cqe 堆满，则 submit 任务可能会失败。
- 一些 feature 无法直接判断是否打开，只能通过 opcode 的支持情况判断内核版本。
- Rust 的封装是 tokio 团队开发，更新不算快。（tokio-uring 项目似乎停滞了。）

不堪大用的 AIO

POSIX 有一个 AIO，Linux 有一个不太好用的 AIO，glibc 用线程池实现了一个 AIO。Linux 的内核 AIO 只能读写 `O_DIRECT` 打开的文件。

在 POSIX AIO 实现中，只有 FreeBSD 的 AIO 能够和 kqueue 协作。在虚拟机上测试过发现相比 tokio 的性能提升不到一个数量级，遂搁置。

参考：`#241`。

能否用 io-uring 实现一个 POSIX AIO？
可以试试，但是恐怕效果不太好。POSIX AIO 并没有强制要求使用 `aio_suspend` 等待任务完成，因此想要在用户态做包装，可能必须额外开线程。

总结

- 对于 IO 密集型程序，异步是提升性能的好选择。尤其是文件方向的 IO，基于完成的异步是唯一的高性能选择。
- Compio 项目是我们对于跨平台、基于完成的异步的框架的探索。通过这一实践我们发现了许多 Rust 设计以及操作系统设计中可以改进的空间。
  - IOCP 的 API 设计非常烂，io-uring 的一些操作也只是对 C 友好（比如 `SEND_ZC` 似乎还是很难用）。
  - Compio 成功让 **Rust 众** 的群友不再呼吁 tokio 2.0，转而研究 async cancel 和 async drop。
- 感谢项目的各位贡献者，尤其是 `@Pop`，他主要贡献了 polling driver、运行在 Linux 上的 fusion driver 和 `compio-io`，还承担了大量的 code review 工作。