
xasync
low ingress async runtime

每周工作

- 第一周 - 回顾
- 第二周 - 定目标
- 第三周 - 学习和实验
- 第四周 - 代码结合

xasync 实现分析

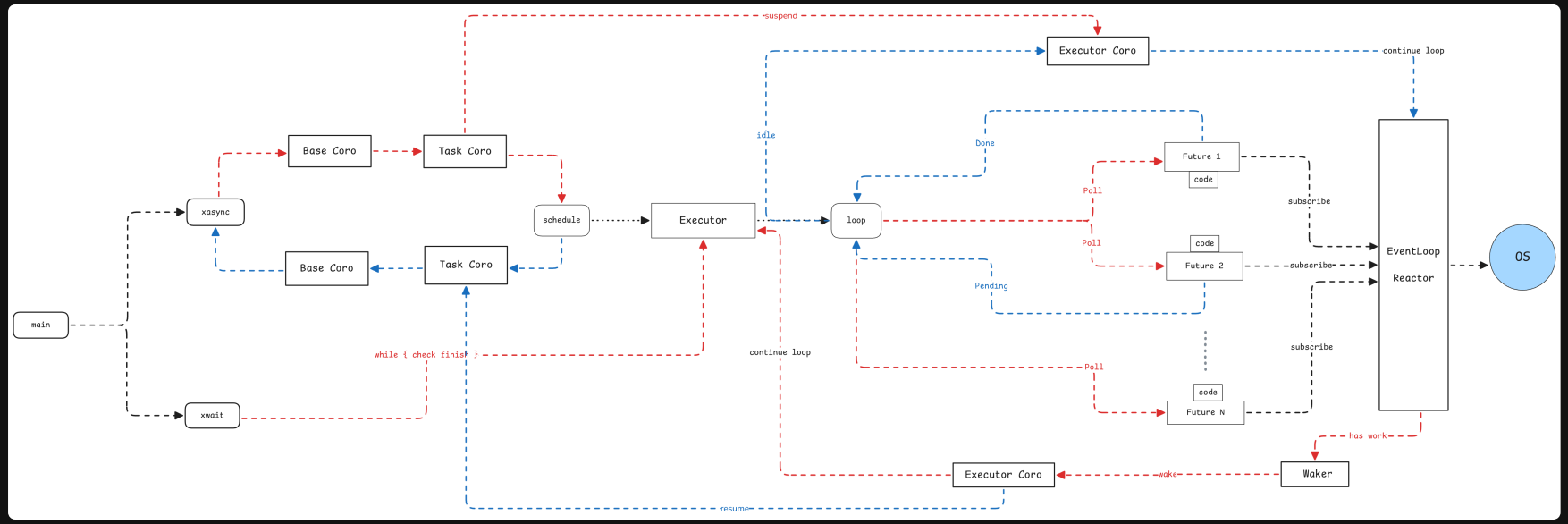
使用者角度

```
var is_async = true // 如果关闭后，底层会走阻塞逻辑
fn read(file) {
    if (is_async) {
        scheudle(future:run(sys_read(file))) // 生成 future，交给 executor 和 eventloop 调度处理
        suspend()
    } else {
        sys_read(file)
    }
}
fn long_time_action() {
    read("large file")
    sleep(100) // 这里依然不会阻塞
}
fn other_action() {

}
fn main() {
    let frame = xasync(long_time_action) // 使用者也可以用 xasync 来标记上层代码是异步的
    // xawait(frame) // 需要等待的时候才等待
    other_action()
}
```

xasync 实现分析

架构设计



xasync 实现分析

Future

其中一个测例

```
test "counter-chain-done" {
    const allocator = std.testing.allocator;

    var executor = Executor.init(allocator);
    defer executor.deinit();

    var counter = Counter.init(0, 5);
    const fut = runWithAllocator(allocator, Counter.doCount, &counter).chain(Counter.printNum); // 这里支持链式调用

    executor.schedule(fut);

    executor.run();
}
```

xasync 实现分析

Executor

两个队列

- ready_queue: std.ArrayList(*Future) - 调度队列，供调用者放入 Future 任务
- futs: std.ArrayList(*Future) - 执行队列，实际调度器处理的 Future 任务

xasync 实现分析

Coroutine

- 支持上线文切换
- 支持参数传递

```
var base_coro: Coroutine = undefined;
var count_coro: Coroutine = undefined;
var count: i32 = 1;

fn addCount() void {
    count += 1;
    base_coro.resumeFrom(&count_coro);
    count += 1;
    base_coro.resumeFrom(&count_coro);
    count += 1;
    base_coro.resumeFrom(&count_coro);
}

test "simple counter suspend and resume coroutine" {
    const allocator = std.testing.allocator;

```

xasync 实现分析

Eventloop

```
pub fn poll(self: *Self, timeout_ms: i32) !usize {
    try self.events.resize(16); // 预分配事件数组，先这么写

    const n = std.posix.epoll_wait(self.epfd, self.events.items, timeout_ms);

    for (self.events.items[0..n]) |event| {
        const fd = event.data.fd;

        if (self.callbacks.get(fd)) |callback| {
            if (event.events & std.posix.system.EPOLL.IN != 0) {
                var buf: [8]u8 = undefined;
                _ = std.posix.read(fd, &buf) catch {}; // 这里目前只处理了 timer 的情况

                if (callback.callback_fn) |func| {
                    func(callback.user_data);
                }
            }
        }
    }
```

xasync 使用

Timer

这部分注册一个 TimerHandle 到 event loop 当中

```
const Timer = struct {
    const Self = @This();

    handle: TimerHandle,
    completed: bool = false,
    waker: ?*const Waker = null,

    fn init(nanoseconds: u64) !Self {
        const handle = try TimerHandle.init(&global_event_loop, nanoseconds); // 注册给 event_loop
        return .{
            .handle = handle,
        };
    }

    fn deinit(self: *Self) void {
        self.handle.deinit();
    }

    fn timerCompletedCallback(data: ?*anyopaque) void { // event_loop 回调
```

xasync 使用

Sleep

这部分把 Timer 包装成 Future

```
fn sleep(nanoseconds: u64) void {
    std.debug.print("sleep comes in\n", .{});
    if (!sys_is_block) {
        const timer_ptr = global_runtime.allocator.create(Timer) catch unreachable;
        timer_ptr.* = Timer.init(nanoseconds) catch unreachable;

        const callback = EventCallback{
            .callback_fn = Timer.timerCompletedCallback,
            .user_data = timer_ptr,
        };
        timer_ptr.handle.setCallback(callback) catch unreachable;

        const timer_fut = future.runWithAllocator(global_runtime.allocator, Timer.poll, timer_ptr).chain(struc
            fn thenFn(_: ?*anyopaque, ctx: *Context) *Future {
                const timer = @as(*Timer, @ptrCast(@alignCast(ctx.payload)));
                ctx.allocator.destroy(timer);
                return future.done(null);
            }
        }.thenFn);
```

xasync 使用

main

```
xasync(delay);
// xawait(); // 需要等待的时候开启
std.debug.print("hello xasync\n", .{});
```

xasync 使用

运行效果

不等待完成

```
delay comes in
sleep comes in
hello xasync                - 注意这里，没有等待 timer 异步执行结束，而是直接返回
timer callback completed!
poll timer is completed     - 注意这里，timer 结束了
main will quit
event loop quit
```

等待完成

```
delay comes in
sleep comes in
timer callback completed!
poll timer is completed
hello xasync                - 注意这里，虽然底层是异步协程执行，但是这里等待 timer 执行完毕才打印
main will quit
event loop quit
```

xasync

总结

从目前执行效果和 API 的调用方式看符合预期，基本达成了本期的目标：`实现简单的异步协程运行时（zig）`，按照这种方式解决`函数着色问题`是有希望的。

虽然本期目标基本达成，但是中间学习的过程中还是有很多技术细节没有完全搞懂，有些学习资料没有完全看完，后续还要继续努力。

Thanks