CloudWeGo

# Rust 异步系统

## Rust 开发实训

茌海 / 字节跳动 Infra

2023/09/07

# 个人介绍

茌海(Chi Hai)
GitHub: ihciah

字节跳动研发工程师
- Rust Async Runtime
- Rust 通用网关框架
- 公司内 Rust 基础库 / 基础设施

# 目录

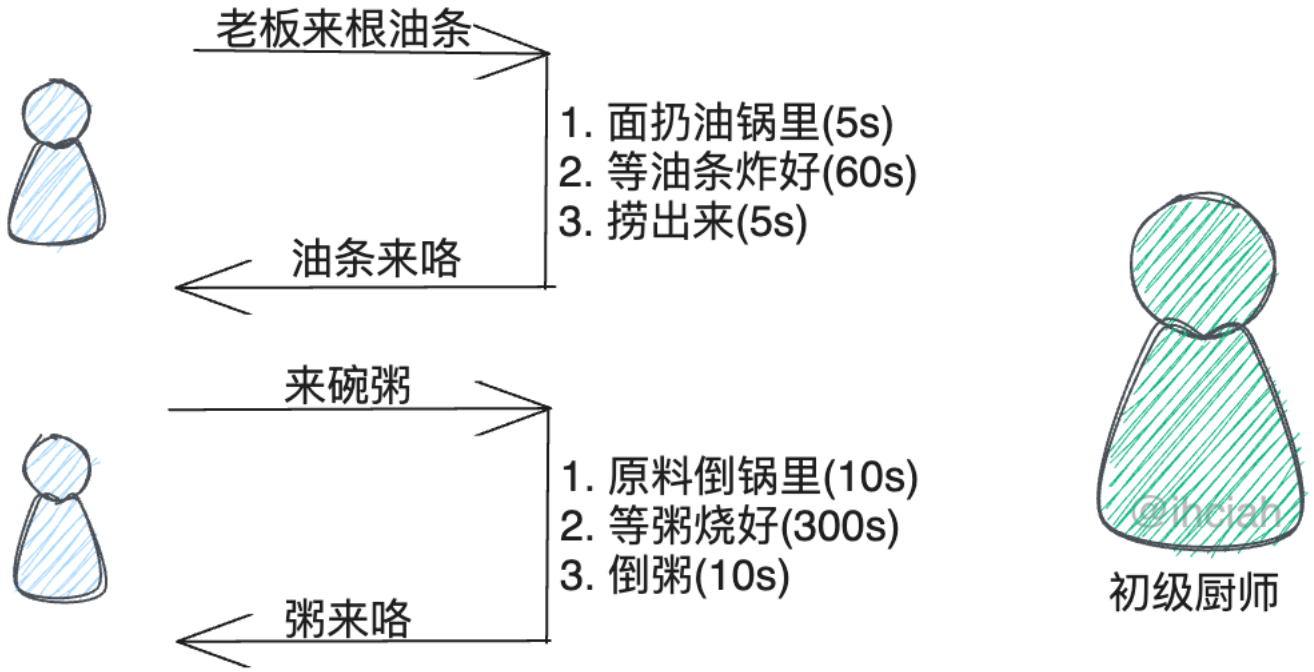
CONTENT

# 01 异步基础

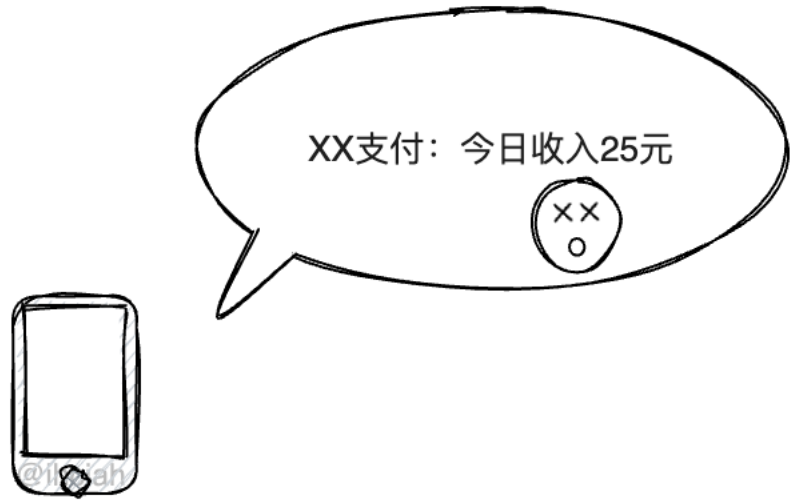
What and Why

# 异步基础

失业了，去卖早点吧！ -> 小海早餐店开业了！



# 异步基础



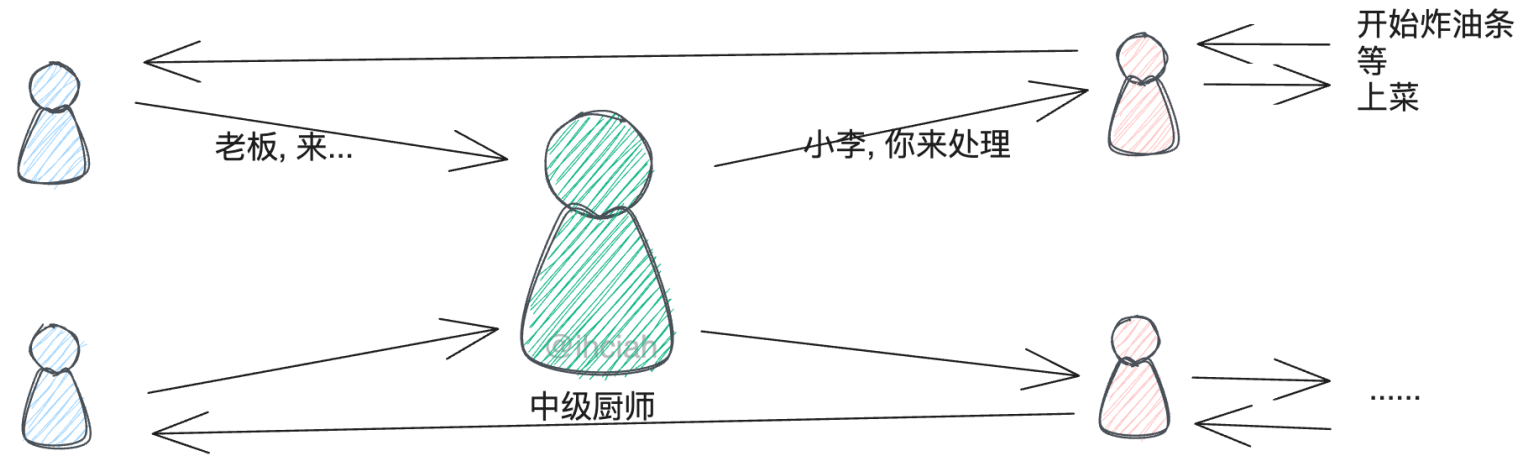

反思一下：摸鱼太多了！

炸油条操作时间：10s
等待时间：60s
摸鱼率：86%

熬粥操作时间：20s
等待时间：300s
摸鱼率：94%

# 异步基础

摸鱼也是无可奈何的事，毕竟有的工序很复杂。。。
招几个小弟吧!



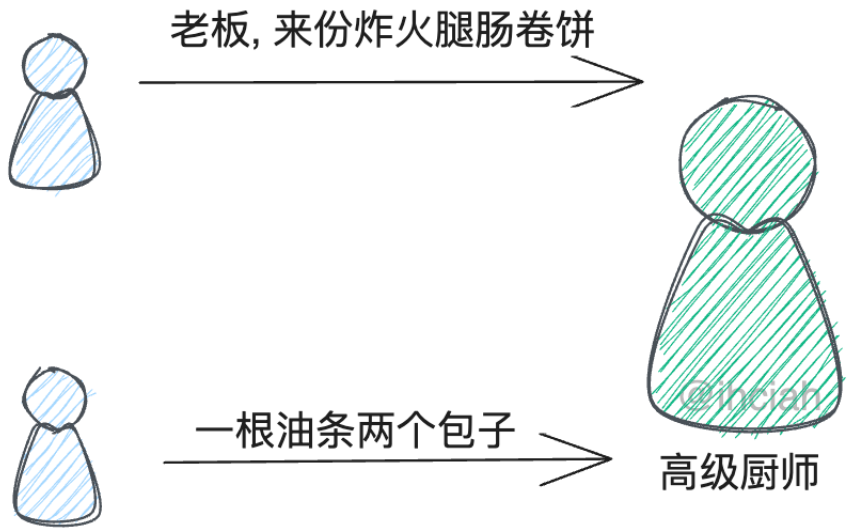
# 异步基础


虽然客人增多收入增多了，但员工工资也是要出的。。。

遇到客人瞬间变多的情况，还要花一大笔紧急招聘费；如果招不到，这个客人就要等着也会影响风评。。。

# 异步基础

算了，还是自己来吧。。。

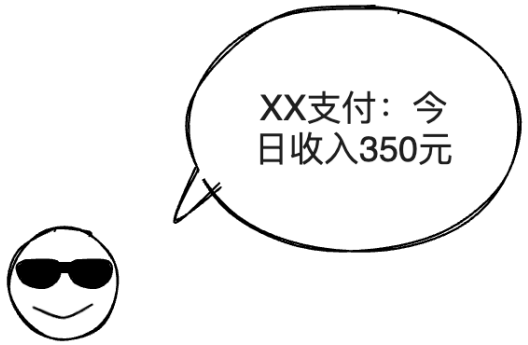



.....

1. 火腿肠扔油锅里
2. 面饼放平底锅上
3. 生油条扔油锅里

```
while let Some(事件) = 等待事件发生 {
        match 事件 {
                火腿肠炸好了 => 火腿肠捞出来,
                饼煎好了 => 卷上火腿肠给顾客,
                油条炸好了 => 和包子一起给顾客
        }
}
```

# 异步基础




要尽可能地服务(zhuan)客户(qian)，就不能闲着

# 异步基础

提升处理能力的通解：多进程 / 多线程

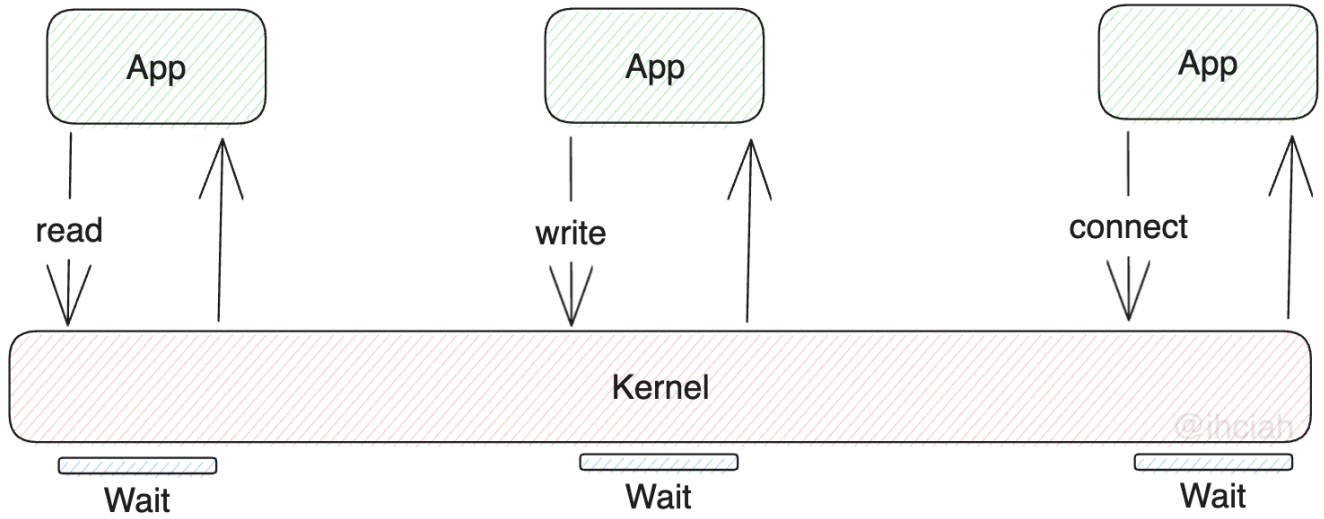
**Naïve 版本:**
为每个请求创建一个新的进程 / 线程
创建开销大

**更好的版本:**
使用线程池
有效复用了线程，但内核的调度器压力会变大
（对应中级厨师的例子）

# 异步基础

除了创建多个线程（多招人）外，为什么不能想办法提升线程利用率（少摸鱼）呢？

一个关键的问题：同步 syscall 等待期间线程不可用




## epoll + non-blocking

non-blocking: 将 fd 设置为 non-blocking 后，对其执行 read、write 等 syscall 时，如果未就绪，则 kernel 立刻返回 WOULD_BLOCK 而非等待。

# 异步基础

## epoll 简明科普

kernel 提供的同时等待多个 fd 就绪的机制
1. epoll_create: 创建一个 epoll fd
2. epoll_ctl: 向 epoll 中添加或删除 interest(= fd + direction)
3. epoll_wait: 陷入内核，等待并得到就绪事件

其他平台也提供了类似的机制，例如 macos 上有 kqueue。
Rust 生态中，mio 提供了跨平台的 poller 封装。
一个例子: https://github.com/tokio-rs/mio/blob/master/examples/tcp_server.rs

**epoll 触发模式**：边缘触发 和 水平触发
边缘触发 = 只在 fd 由非就绪状态变为就绪状态时通知用户
水平触发 = 只要 fd 就绪就通知用户

边缘触发更高效，但需要在用户态维护 fd 状态。

# 异步基础

解决完线程利用率问题，我们还要足够容易地使用它。

## Callback 模式

向 poller 注册回调函数。
实现较为容易，但用起来不太友好（callback hell 警告）。

用户需要将一段逻辑按照等待点打断为多段逻辑，并将他们用 callback 串起来。

典型代表：libevent(C)

## 绿色线程（有栈协程）模式

劫持系统调用，并将它们改为非阻塞模式。用户创建协程时，由 Runtime 在堆上分配函数栈空间，并由 Runtime 控制切换。

使用起来用户几乎可以将其当作普通线程使用。

典型代表：Goroutine(go)、gevent(py)

## 无栈协程模式

编译器将用户代码展开为状态机，在需要等待时将控制权交还上层。最外层 Runtime 负责调度任务。

效率最高，写异步逻辑也简便，但对用户使用姿势有要求：不能调用同步 syscall。

典型代表：Rust、Coroutine of C++20

# 02 Rust 异步结构

## Future, Waker and Runtime

# 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();
}
```

## 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）：

```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 异步机制 – Async Await 背后的秘密

```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
}
```

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

```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
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

```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)
    }
}
```

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

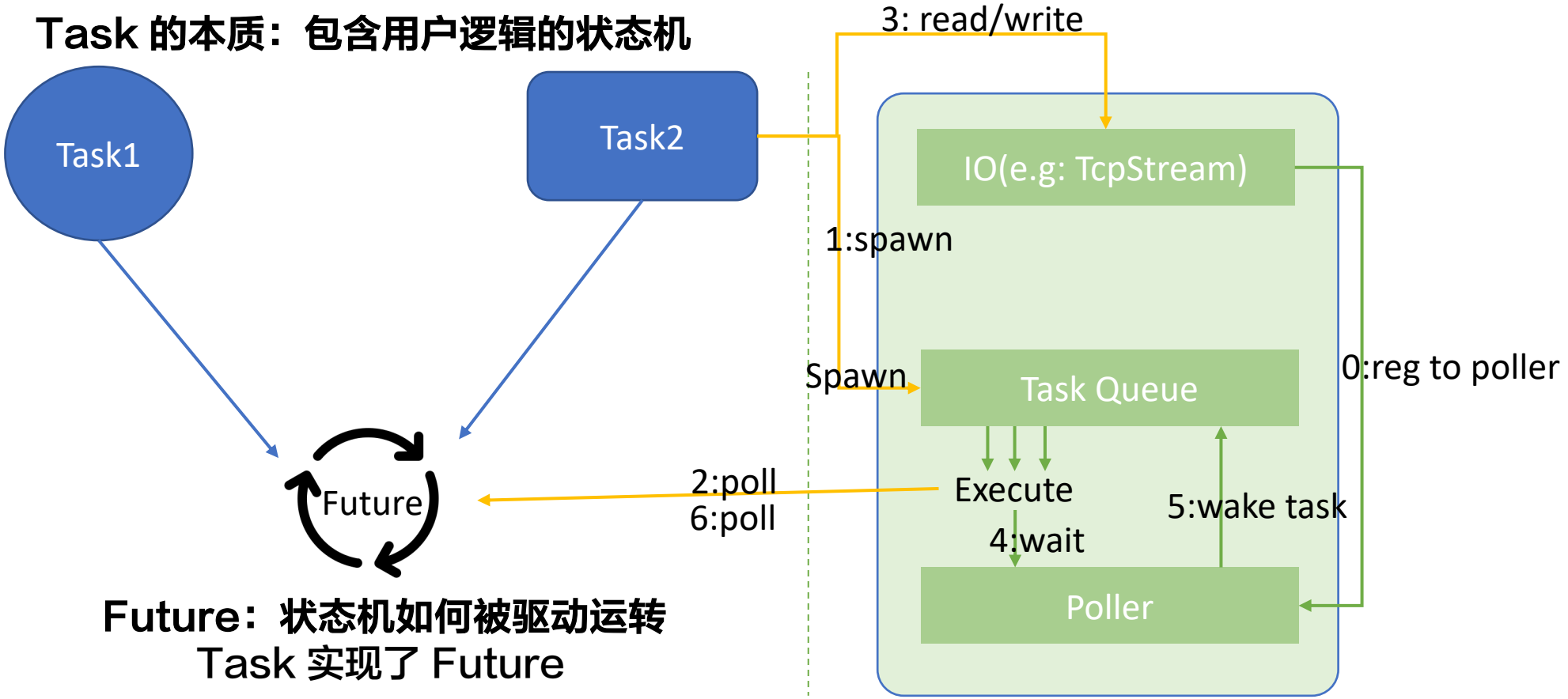
# Rust 异步机制 – 手动实现 Future

```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::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




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

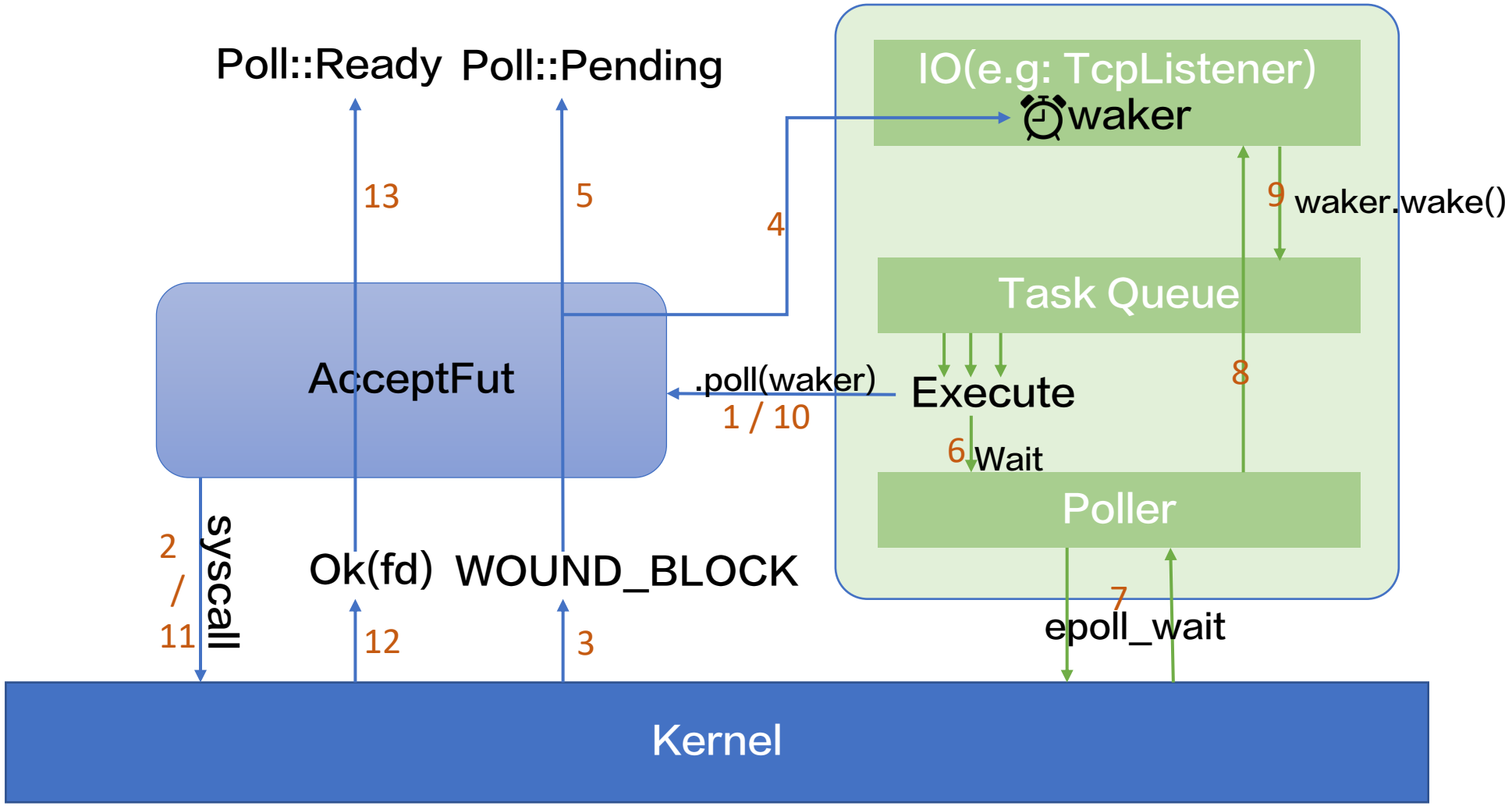
我知道 Future 本质是状态机没错，我也知道状态机每次运转可能返回 Pending / Ready…

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？

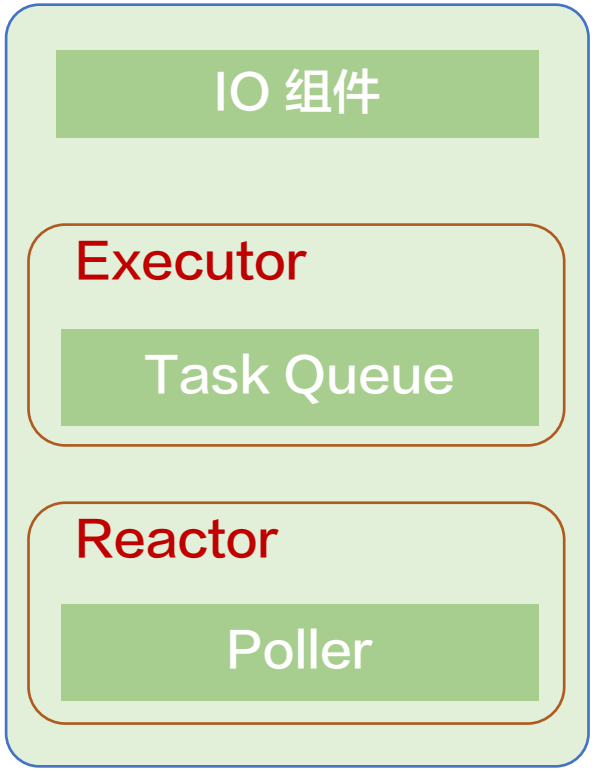
```rust
pub trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}

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 ()>,
}
```

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

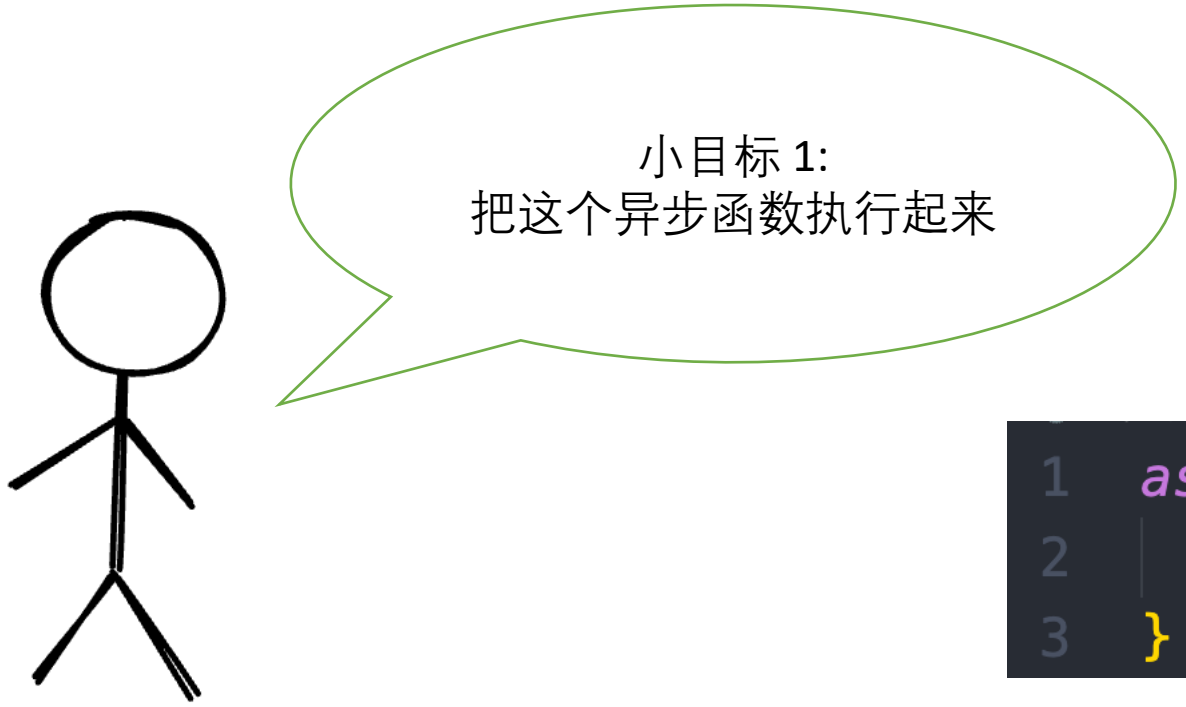
# 03

# 上手 Runtime

## 上手写一个极简 Runtime

~~君 Rust 本当上手~~

# 异步 Runtime – Simple block_on




```rust
async fn demo() {
    println!("hello world!");
}
```

# 异步 Runtime – Simple block_on

```rust
async fn demo() {
    println!("hello world!");
}
```



Just poll it!

```rust
use std::future::Future;

1 implementation
struct Demo;

impl Future for Demo {
    type Output = ();

    fn poll(self: std::pin::Pin<&mut Self>, _cx: &mut std::task::Context<'_>) -> std::task::Poll<Self::Output> {
        println!("hello world!");
        std::task::Poll::Ready(())
    }
}
```

# 异步 Runtime – Simple block_on

```rust
fn dummy_waker() -> Waker {
    static DATA: () = ();
    unsafe { Waker::from_raw(waker: RawWaker::new(&DATA, &VTABLE)) }
}

const VTABLE: RawWakerVTable =
    RawWakerVTable::new(vtable_clone, vtable_wake, vtable_wake_by_ref, vtable_drop);

unsafe fn vtable_clone(_p: *const ()) -> RawWaker {
    RawWaker::new(data: _p, &VTABLE)
}

unsafe fn vtable_wake(_p: *const ()) {}

unsafe fn vtable_wake_by_ref(_p: *const ()) {}

unsafe fn vtable_drop(_p: *const ()) {}
```

构建 Waker 空实现: 现在我们并不需要它做什么

Tips: Waker::noop() provides a similar impl(unstable)

# 异步 Runtime – Simple block_on

```rust
fn block_on<F: Future>(future: F) -> F::Output {
    let mut fut: Pin<&mut F> = std::pin::pin!(future);
    let waker: Waker = dummy_waker();

    let mut cx: Context<'_> = Context::from_waker(&waker);
    loop {
        if let Poll::Ready(output: <F as Future>::Output) = fut.as_mut().poll(&mut cx) {
            return output;
        }
    }
}
```

实现 block_on: 简单 loop poll 即可

# 异步 Runtime – Simple block_on

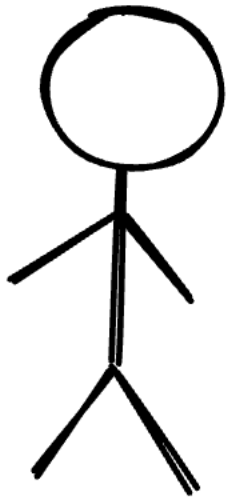
```rust
fn block_on<F: Future>(future: F) -> F::Output {
    let mut fut: Pin<&mut F> = std::pin::pin!(future);
    let waker: Waker = dummy_waker();

    let mut cx: Context<'_> = Context::from_waker(&waker);
    loop {
        if let Poll::Ready(output: <F as Future>::Output) = fut.as_mut().poll(&mut cx) {
            return output;
        }
    }
}
```

实现 block_on: 简单 loop poll 即可

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target/debug/playground`
hello world!
 *  Terminal will be reused by tasks, press any key to close it.
```

# 异步 Runtime – block_on with Waker

小目标 2:
允许任务被外部唤醒，且不 spin

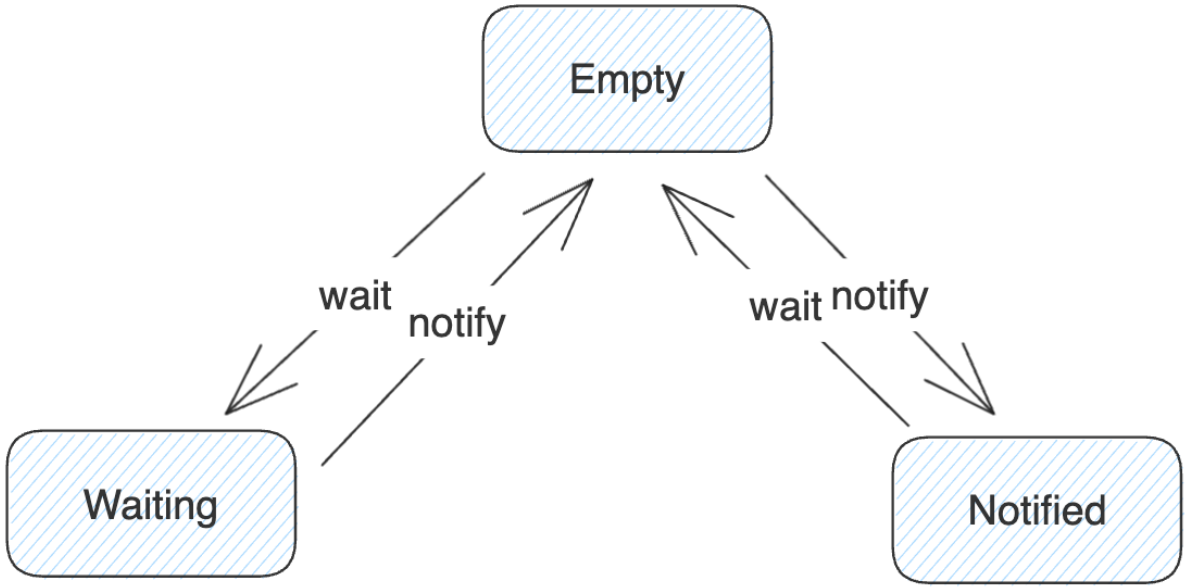


```rust
async fn demo() {
    let (tx: Sender<()>, rx: Receiver<()>) = async_channel::bounded::<()>(cap: 1);
    std::thread::spawn(move|| {
        std::thread::sleep(dur: Duration::from_secs(20));
        tx.send_blocking(msg: ())
    });
    let _ = rx.recv().await;
    println!("hello world!");
}

▶ Run | Debug
fn main() {
    block_on(future: demo());
}
```

```
Main  I/O
  PID USER      PRI  NI  VIRT   RES   SHR S  CPU%▽MEM%   TIME+  Command
849812 ihciah    20   0 71140  3968  1792 R  99.3  0.0  0:09.79 target/debug/playground
```

# 异步 Runtime – block_on with Waker

我们需要某种机制来等待通知，而且不能 spin，那么这一定是个 syscall
Condvar(基于 futex)、EventFd 等都可以满足需求；这里方便起见我们直接使用 std 提供的 Condvar
最终封装为 Signal 结构，暴露 wait 和 notify 接口。




```rust
struct Signal {
    state: Mutex<State>,
    cond: Condvar,
}

0 implementations
enum State {
    Empty,
    Waiting,
    Notified,
}
```

# 异步 Runtime – block_on with Waker

```rust
53      fn wait(&self) {
54          let mut state: MutexGuard<'_, State> = self.state.lock().unwrap();
55          match *state {
56              State::Notified => *state = State::Empty,
57              State::Waiting => {
58                  panic!("multiple wait");
59              }
60              State::Empty => {
61                  *state = State::Waiting;
62                  while let State::Waiting = *state {
63                      state = self.cond.wait(guard: state).unwrap();
64                  }
65              }
66          }
67      }
68
69      fn notify(&self) {
70          let mut state: MutexGuard<'_, State> = self.state.lock().unwrap();
71          match *state {
72              State::Notified => {}
73              State::Empty => *state = State::Notified,
74              State::Waiting => {
75                  *state = State::Empty;
76                  self.cond.notify_one();
77              }
78          }
79      }
```

基于 Condvar 实现 wait 和 notify

Waker 被 wake 时调用 notify;
Runtime 空置时调用 wait

# 异步 Runtime – block_on with Waker

```rust
impl Wake for Signal {
    fn wake(self: Arc<Self>) {
        self.notify();
    }
}
```

标准库提供了便捷的函数，将 Arc<impl Wake + ...> 转换为 Waker

```rust
#[stable(feature = "wake_trait", since = "1.51.0")]
impl<W: Wake + Send + Sync + 'static> From<Arc<W>> for Waker {
    /// Use a `Wake`-able type as a `Waker`.
    ///
    /// No heap allocations or atomic operations are used for this conversion.
    fn from(waker: Arc<W>) -> Waker {
        // SAFETY: This is safe because raw_waker safely constructs
        // a RawWaker from Arc<W>.
        unsafe { Waker::from_raw(waker: raw_waker(waker)) }
    }
}
```

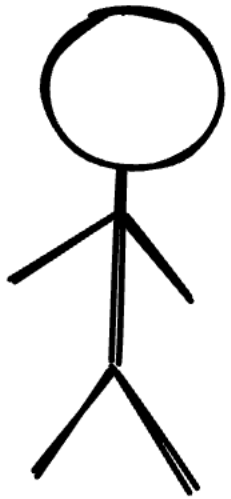
```rust
fn main() {
    block_on(future: demo());
}

fn block_on<F: Future>(future: F) -> F::Output {
    let mut fut: Pin<&mut F> = std::pin::pin!(future);
    let signal: Arc<Signal> = Arc::new(data: Signal::new());
    let waker: Waker = Waker::from(signal.clone());

    let mut cx: Context<'_> = Context::from_waker(&waker);
    loop {
        if let Poll::Ready(output: <F as Future>::Output) = fut.as_mut().poll(&mut cx) {
            return output;
        }
        signal.wait();
    }
}
```

最终组装一下即可，等待期间 CPU 占用率为 0

```
ihciah      20   0 71144  3976  1792 S   0.0  0.0  0:00.05 target/debug/playground
```

# 异步 Runtime – MultiTasks

小目标 3:
支持 spawn task



```rust
async fn demo() {
    spawn(future: demo2());
    println!("hello world!");
}

async fn demo2() {
    println!("hello world2!");
}

▶ Run | Debug
fn main() {
    block_on(future: demo());
}
```

# 异步 Runtime – MultiTasks

Question1: Task 如何表示?

```
41  struct Task {
42      future: RefCell<BoxFuture<'static, ()>>,
43      signal: Arc<Signal>,
44  }
45  unsafe impl Send for Task {}
46  unsafe impl Sync for Task {}
47
48  impl Wake for Task {
49      fn wake(self: Arc<Self>) {
50          RUNNABLE.with(|runnable: &Mutex<VecDeque<Arc<Task>>>| runnable.lock().unwrap().push_back(self.clone()));
51          self.signal.notify();
52      }
53  }
```

定义 Task: 包含 Future 主体，和唤醒用的 signal

# 异步 Runtime – MultiTasks

Question2: Task 之间什么关系？

```
loop {
    if let Poll::Ready(output: <F as Future>::Output) = main_fut.as_mut().poll(&mut cx) {
        return output;
    }
    while let Some(task: Arc<Task>) = runnable.lock().unwrap().pop_front() {
        let waker: Waker = Waker::from(task.clone());
        let mut cx: Context<'_> = Context::from_waker(&waker);
        let _ = task.future.borrow_mut().as_mut().poll(&mut cx);
    }
    signal.wait();
}
```

Task 分为主 Task 与附属其他 Tasks，主 Task 结束则返回

# 异步 Runtime – MultiTasks

Question3: 任务如何存储 / 销毁?

```rust
scoped_thread_local!(static SIGNAL: Arc<Signal>);
scoped_thread_local!(static RUNNABLE: Mutex<VecDeque<Arc<Task>>>);
```

```rust
let runnable: Mutex<VecDeque<Arc<Task>>> = Mutex::new(VecDeque::with_capacity(1024));
SIGNAL.set(t: &signal, f: || {
    RUNNABLE.set(t: &runnable, f: || {
        loop {
            if let Poll::Ready(output: <F as Future>::Output) = main_fut.as_mut().poll(&mut cx) {
                return output;
            }
            while let Some(task: Arc<Task>) = runnable.lock().unwrap().pop_front() {
                let waker: Waker = Waker::from(task.clone());
                let mut cx: Context<'_> = Context::from_waker(&waker);
                let _ = task.future.borrow_mut().as_mut().poll(&mut cx);
            }
            signal.wait();
        }
    })
})
```

Task 存储在全局的 VecDeque 中，销毁由 Arc 控制

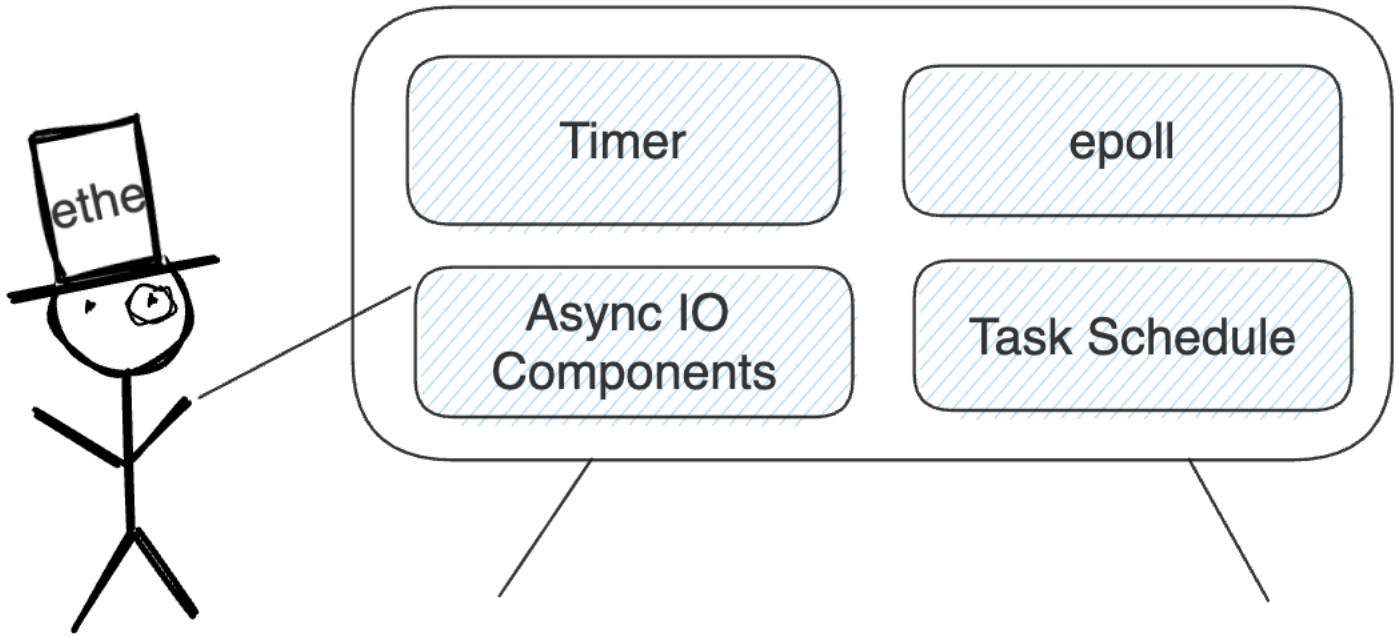
# 异步 Runtime – MultiTasks

```rust
async fn demo() {
    let (tx: Sender<()>, rx: Receiver<()>) = async_channel::bounded(cap: 1);
    spawn(future: demo2(tx));
    println!("hello world!");
    let _ = rx.recv().await;
}

async fn demo2(tx: async_channel::Sender<()>) {
    println!("hello world2!");
    let _ = tx.send(msg: ()).await;
}

▶ Run | Debug
fn main() {
    block_on(future: demo());
}
```

```
   Compiling playground v0.1.0 (/home/ihciah/code/ihciah/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 0.52s
     Running `target/debug/playground`
hello world!
hello world2!
 *  Terminal will be reused by tasks, press any key to close it.
```

运行 spawn 并等待的 demo: It Works!

# 异步 Runtime – What’s Next?



# 异步 Runtime – 实践

1. 完成本课程内的单线程 Runtime，要求性能尽可能好
2. (可选) 基于前面的实现，为该 Runtime 增加多线程运行任务的能力，并实现较为基础的任务调度

# THANKS

CloudWeGo

# 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：

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

```rust
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