Embassy 组件化

— Arceos 生态

吕粤蒙

OpenCamp 春夏开源内核训练营

2025-07-02

介绍

Embassy **模块化**

Axtask 兼容

Embassy 提供 Executor 作为运行时，其中 Spawner 作为 Executor 的引用提供 spawn() 函数生成异步任务。如下为标准的运行时结构:

```
#[percpu::def_percpu]
static SIGNAL_WORK_THREAD_MODE: AtomicBool = AtomicBool::new(false);

#[unsafe(export_name = "__pender")]
fn __pender(_context: *mut ()) {
    SIGNAL_WORK_THREAD_MODE.with_current(|m| {
        m.store(true, Ordering::SeqCst);
    });
}

pub struct Executor {
    inner: raw::Executor,
    not_send: PhantomData<*mut ()>,
}
```

Axtask 兼容

兼容线程意味着中断的条件从 `irq` 至线程的抢占中断:

```
pub fn run(...) -> ! {
  self.inner.poll();

  if SIGNAL_WORK_THREAD_MODE.load(Ordering::SeqCst) {
      SIGNAL_WORK_THREAD_MODE.store(false, Ordering::SeqCst);
  } else {
      axhal::arch::wait_for_irqs();
  }
}
```

修改的核心仅在于让出条件的改变:

```
pub fn run(...) -> ! {
  self.inner.poll();
  let polled = SIGNAL_WORK_THREAD_MODE.with_current(|m| m.load(Ordering::Acquire));
  if polled {
      SIGNAL_WORK_THREAD_MODE.with_current(|m| {
          m.store(false, Ordering::SeqCst);
      });
  } else {
      // park_current_task();
      axtask::yield_now();
  }
}
```

Axtask 兼容

我们选择单运行时的线程管理，运行时所在的线程初始化时，将其引用存储于一全局变量中。

```
pub(crate) static SPAWNER: Mutex<OnceCell<SendSpawner>> = Mutex::new(OnceCell::new());

#[embassy_executor::task]
async fn init_task() {
    use crate::asynch;

    let spawner = asynch::Spawner::for_current_executor().await;
    asynch::set_spawner(spawner.make_send());
    log::info!("Initialize spawner... ");
}
```

用户在获取 `spawner` 时，线程调度会初始化运行时线程：

```
pub fn spawner() -> SendSpawner {
    let sp = SPAWNER.lock();
    if let Some(inner) = sp.get() {
        *inner
    } else {
        drop(sp);
        init_spawn();
        yield_now();
        // initialize the spawner if not
        let sp = SPAWNER.lock();
        *sp.get().expect("Reinitialize the spawner failed")
    }
}
```

异步共享

异步的共享必须局限于同一运行时内，在单线程的运行时管理下这种共享是便捷的，但是一旦存在多线程的运行时管理，则存在一定的心智负担。

```rust
pub struct SameExecutorCell<T> {
    /// The executor id
    id: usize,
    inner: T,
}

type MutexSignal<T> = Signal<CriticalSectionRawMutex, T>;

pub struct Delegate<T> {
    send: MutexSignal<SameExecutorCell<*mut T>>,
    reply: MutexSignal<()>,
    state: AtomicU8,
    _not_send: PhantomData<*const ()>,
}
```

异步共享

我们使用 send 用于数据传输，reply 用于信息传达，即 lend 过程的传递和结束。

代码的形式通常如下：

```rust
pub async fn lend<'a, 'b: 'a>(&'a self, target: &'b mut T) -> Result<(), DelegateError> {
  ...
  match self.state.compare_exchange(
      New as u8,
      Lent as u8,
      core::sync::atomic::Ordering::AcqRel,
      core::sync::atomic::Ordering::Acquire,
  ) {
      Ok(_) => {}
      Err(_) => return Err(LendInvalid),
  }
  let sp = Spawner::for_current_executor().await;
  let ptr = ptr::from_mut(target);
  self.send.signal(SameExecutorCell::new(ptr, sp));

  self.reply.wait().await;
  ...
}
```

其中我们通过 enum 构建 NEW → LENT → CONSUMED 的状态机。

时间驱动

AxDriver 定义了时间相关的唤醒功能，用于支撑 Embassy 的时间管理。其中 Embassy 使用同一的 tick 单位定义时间流逝。

```
struct AxDriver {
    queue: SpinNoIrq<RefCell<Queue>>,
    // static period interval
    period_nanos: AtomicU64,
}

/// Dequeue expired timer and return nanos of next expiration
pub fn next_expiration(&self, period: u64) -> u64 {
    ...
    // queue wakes and remove expired task.
    let ticks_next_expired = queue.next_expiration(ticks_now);
    let nanos_next_expired = ticks_to_nanos(ticks_next_expired);
    nanos_next_expired
}
```

时间驱动

我们将相关函数注册进中断:

```
fn update_timer() {
  ...
  #[cfg(feature = "embassy-timer")]
  {
      use axembassy::AxDriverAPI;
      let next_expired =
AxDriverAPI::next_expiration(PERIODIC_INTERVAL_NANOS);
      if deadline >= next_expired {
          deadline = next_expired;
      }
  }
}
```

我们期望更为优秀的设计方式，例如异步。

优先级

因为 Executor 通过头插法插入 task，并通过遍历的方式唤醒任务，则通过改变任务抽取先后顺序的方法不太实际。主要尝试通过独立的具有优先级判断的 Future 改变抽取的概率分布，在任务不变的情况下会稳定分布。

设定一优先级水平，高于此的优先级可以通过，反之则不然。 以$[0, 100]$的整数区间模拟两位小数,且优先级越小，数字越大。

```
fn poll(...) -> ... {
  ...
  let cur_prio = s.cur_prio;
  let prio = this.prio;
  let tol = Prio::weight(prio, Prio::TOL);

  s.adjust_cur_prio(prio);

  // If future prio > cur_prio + tolerance, park it
  let threshold: u64 = prio.as_u64().saturating_sub(cur_prio);
  if threshold > tol {
      s.park_future(this.id);
      cx.waker().wake_by_ref();
      return Poll::Pending;
  } else {
      s.unpark_future(this.id);
  }
}
```

优先级

这一优先级水平则必须动态调整以保证低优先级任务不会饿死。我们推测，每一优先级都期望将优先级拉至其水平线，而拉取的“幅度”应当反比于优先级和同一优先级正运行的任务数。

```
pub fn adjust_cur_prio(&mut self, prio: Prio) {
  ...
  let prio: u64 = prio.into();
  let cur_prio = self.cur_prio.into();
  let factor = (Prio::weight(norm_prio, Prio::PRIO_EFFECT)
      .saturating_add(Prio::weight(norm_active, Prio::ACTIVE_EFFECT)))
  .clamp(Prio::CLAMP_MIN, Prio::CLAMP_MAX);
  if prio > cur_prio {
      self.cur_prio += Prio::weight(prio - cur_prio, factor);
  } else {
      self.cur_prio -= Prio::weight(cur_prio - prio, factor);
  }
}
```

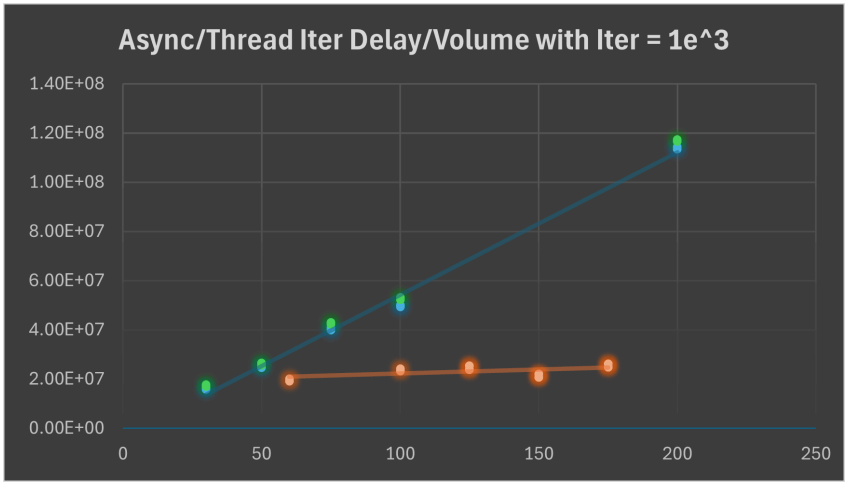
我们首先将优先级和该优先级所在的任务数进行正规化,并进行权重加和。

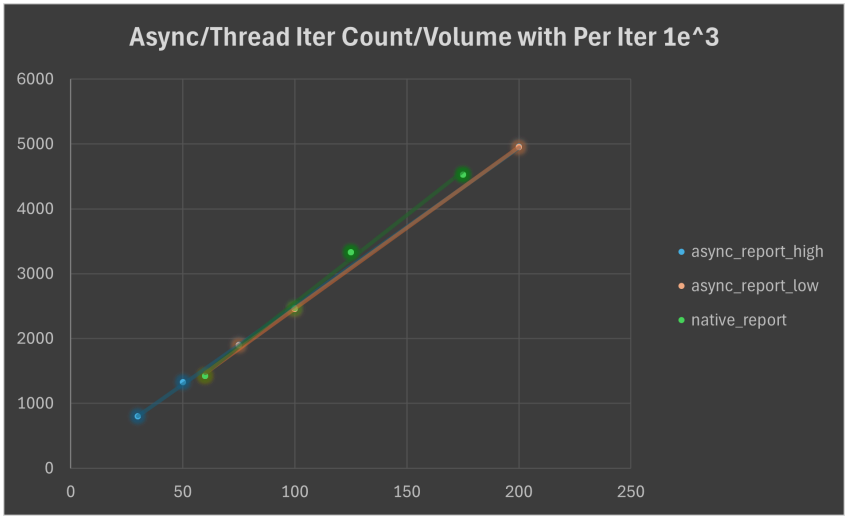
再通过参数化 $\mathrm{cur} = \mathrm{factor}(\mathrm{new} - \mathrm{cur}) + \mathrm{cur} \quad 0 < \mathrm{factor} < 1$,调整权重。

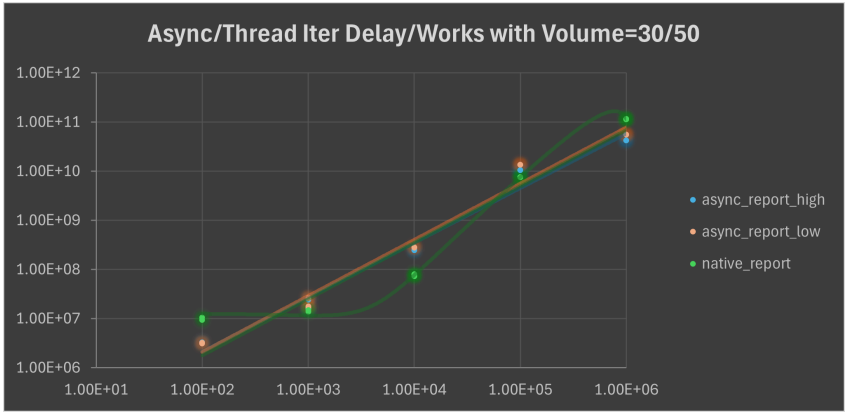
性能测试

循环延迟

测试线程与协程在一定工作量和容量下的循环延迟。其中我们可以观察到即使随容量上升，任务的调度延迟呈线性增加，但是在工作量上并未下降。与此同时可以发现随工作量增加，线程的调度开始隐约出现指数上升的情况,这是由于上下文开销导致的结果。

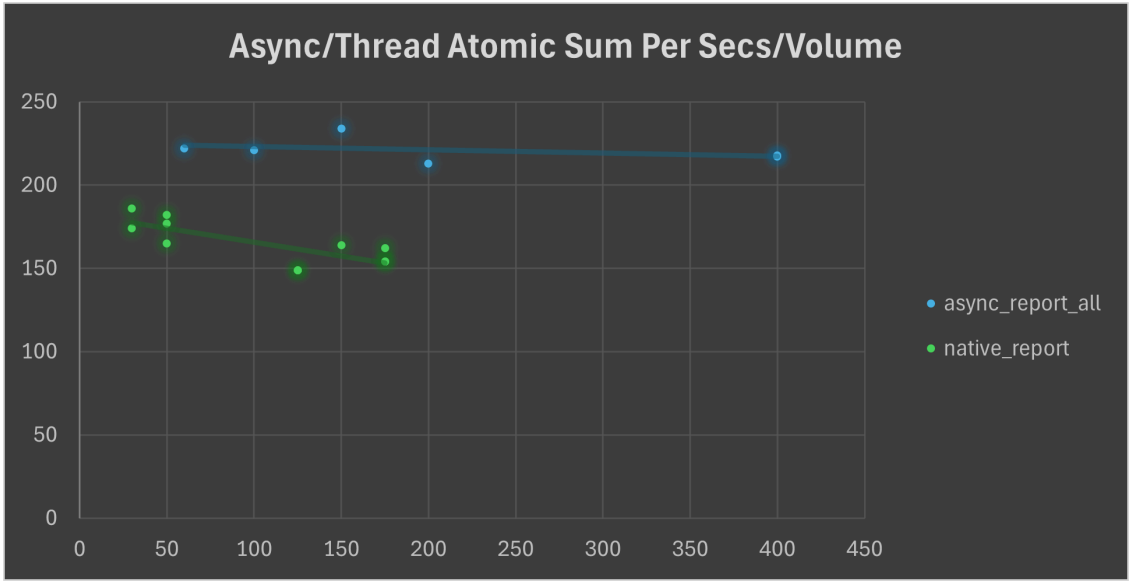






原子读写

因为线程的竞争问题，容量增加后反而呈现下降的趋势，而协程因为各任务独立调度，容量增加后仍保持稳定。



感谢!!