RUST CHINA CONF 2020

首届中国 Rust 开发者大会

2020.12.26–27 深圳

Rust China Conf 2020

Shenzhen, China

2020conf.rustcc.cn

Rust 异步和并发浅谈

自我介绍

姓名： 赖智超　　公司： Onchain　　职位： 区块链架构师

联系方式： github.com/laizy

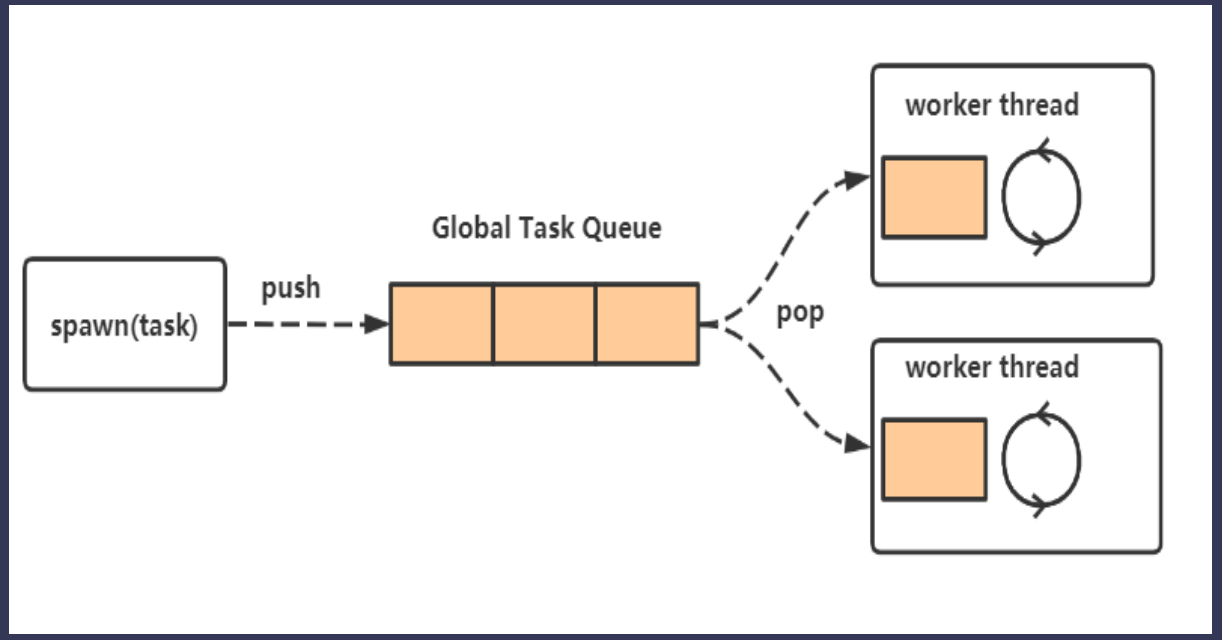
目前 Rust 在公司的应用情况：

1. 区块链 Wasm JIT 虚拟机：基于 cranelift/wasmtime；

2. 智能合约开发库和配套的工具链：目前合约开发都首选 Rust，开发效率高，迭代速度快；

3. 密码学库；

同步任务的多线程 Executor




```rust
use std::thread;
use crossbeam::channel::{unbounded, Sender};
use once_cell::sync::Lazy;

type Task = Box<dyn FnOnce() + Send + 'static>;

static QUEUE: Lazy<Sender<Task>> = Lazy::new(|| {
    let (sender, receiver) = unbounded::<Task>();
    for _ in 0..4 {
        let recv = receiver.clone();
        thread::spawn(|| {
            for task in recv {
                task();
            }
        });
    }
    sender
});

fn spawn<F>(task: F) where F: FnOnce()+ Send + 'static
{   QUEUE.send(Box::new(task)).unwrap();
}
```

异步任务的多线程 Executor

异步任务：不能立即完成，需要多次执行。

```
type Task = Box<dyn FnMut() -> bool + Send + 'static>;
```

Executor 不知道任务何时 ready，忙等肯定不行。因此提供一个 Waker，告诉 Task 如果你啥时候好了，可以通过它重新放进全局队列里去，以便再次执行。

```
type Task = Box<dyn FnMut(waker: &Waker) -> bool + Send + 'static>;
```

异步任务执行没 ready 时，可以将拿到的 Waker 注册到能监控任务状态的 Reactor 中，如 io epoll, timer 等。

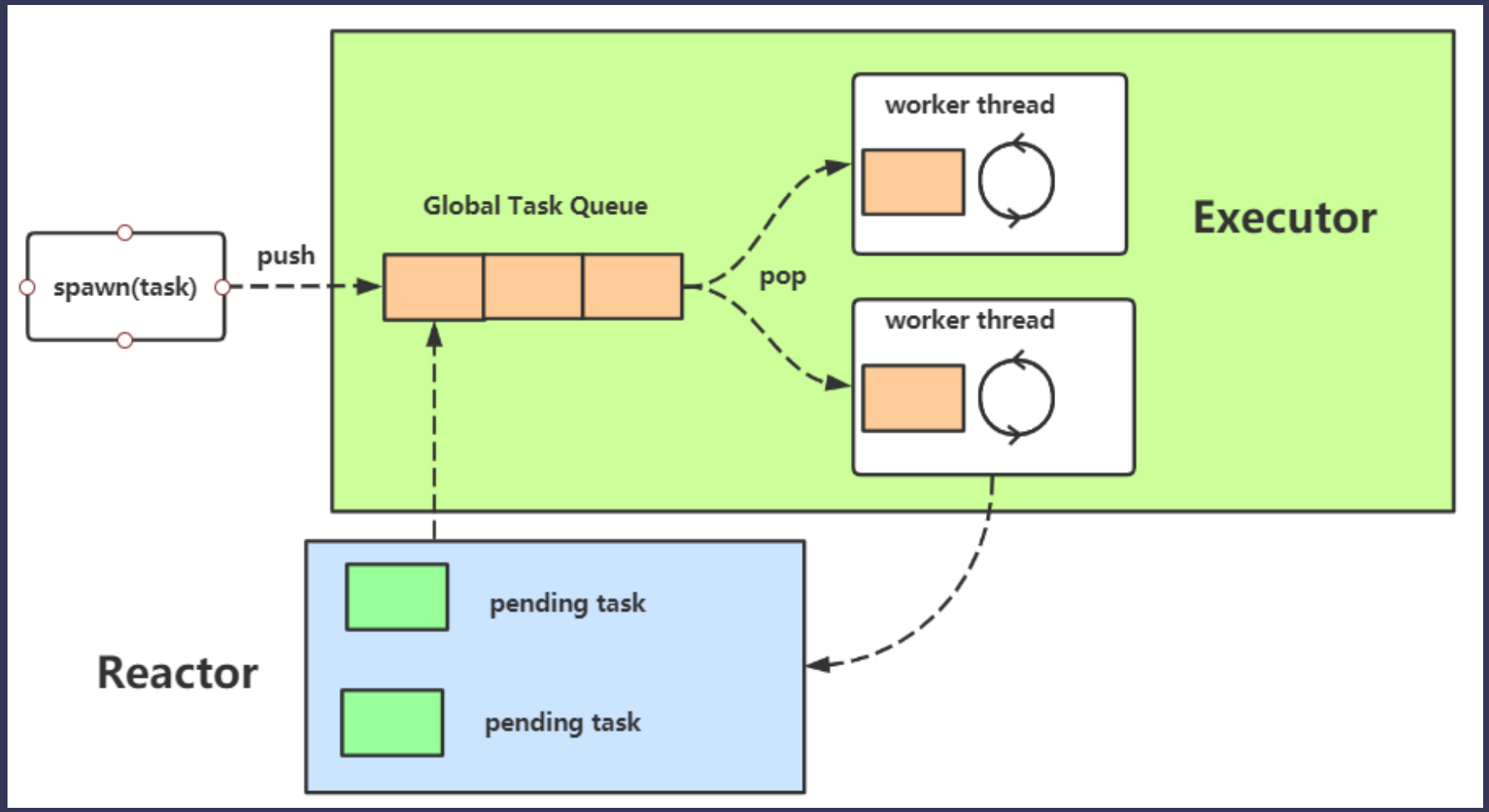
异步任务的多线程 Executor

异步计算的标准定义：

```rust
pub enum Poll<T> {
    Ready(T),
    Pending,
}

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

异步任务的多线程 Executor



Waker 接口的要求

使用方：

异步任务可能对应 io/timer 等多种底层 Reactor 。

Executor 和 Reactor 可能不在同一个线程。

多个 Reactor 可能同时触发任务唤醒。

```
impl Waker {
    pub fn wake(self);
}

impl Clone for Waker;

impl Send for Waker;

impl Sync for Waker;
```

Waker 接口的要求

提供方：

不同的 Executor 有不同的内部实现，要构造统一的 Waker 必然涉及多态，因此采用自定义虚表的方式：

```
impl Waker {
  pub unsafe fn from_raw(waker: RawWaker) -> Waker
}

pub struct RawWaker {
    data: *const (),
    vtable: &'static RawWakerVTable,
}
pub struct RawWakerVTable {
    clone: unsafe fn(*const ()) -> RawWaker,
    wake: unsafe fn(*const ()),
    wake_by_ref: unsafe fn(*const ()),
    drop: unsafe fn(*const ()),
}
```

Waker 实现需要考虑的并发问题

wake 调用之间的并发：确保只压入执行队列一次。

wake 调用和 poll 之间的并发：确保在 poll 期间不压入队列，设置标志位委托 executor 在 poll 结束后压入。

async-task 完美地处理了这些并发问题，同时提供了优雅的 api 。
源码解析见： https://zhuanlan.zhihu.com/p/92679351

异步任务的多线程 Executor

```rust
use std::thread;
use crossbeam::channel::{unbounded, Sender};
use once_cell::sync::Lazy;

type Task = Box<dyn FnOnce() + Send + 'static>;

static QUEUE: Lazy<Sender<Task>> = Lazy::new(|| {
    let (sender, receiver) = unbounded::<Task>();
    for _ in 0..4 {
        let recv = receiver.clone();
        thread::spawn(|| {
            for task in recv {
                task();
            }
        });
    }
    sender
});

fn spawn<F>(task: F) where F: FnOnce()+ Send + 'static
{   QUEUE.send(Box::new(task)).unwrap();
}
```

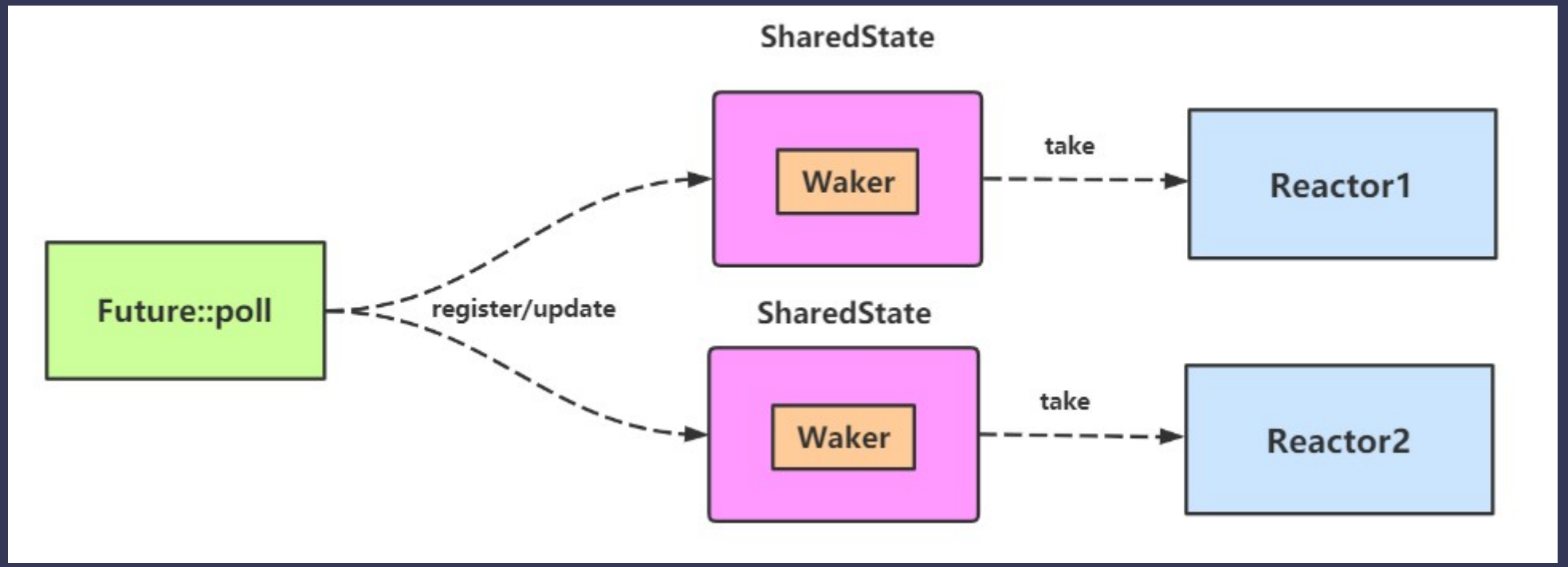
```rust
static QUEUE: Lazy<Sender<async_task::Task<()>>> = Lazy::new(|| {
    let (sender, receiver) = unbounded::<async_task::Task<()>>();
    for _ in 0..4 {
        let recv = receiver.clone();
        thread::spawn(|| {
            for task in recv {
                task.run();
            }
        });
    }

    sender
});

fn spawn<F, R>(future: F) -> async_task::JoinHandle<R, ()>
where
    F: Future<Output = R> + Send + 'static,
    R: Send + 'static,
{
    let schedule = |task| QUEUE.send(task).unwrap();
    let (task, handle) = async_task::spawn(future, schedule, ());
    task.schedule();
    handle
}
```

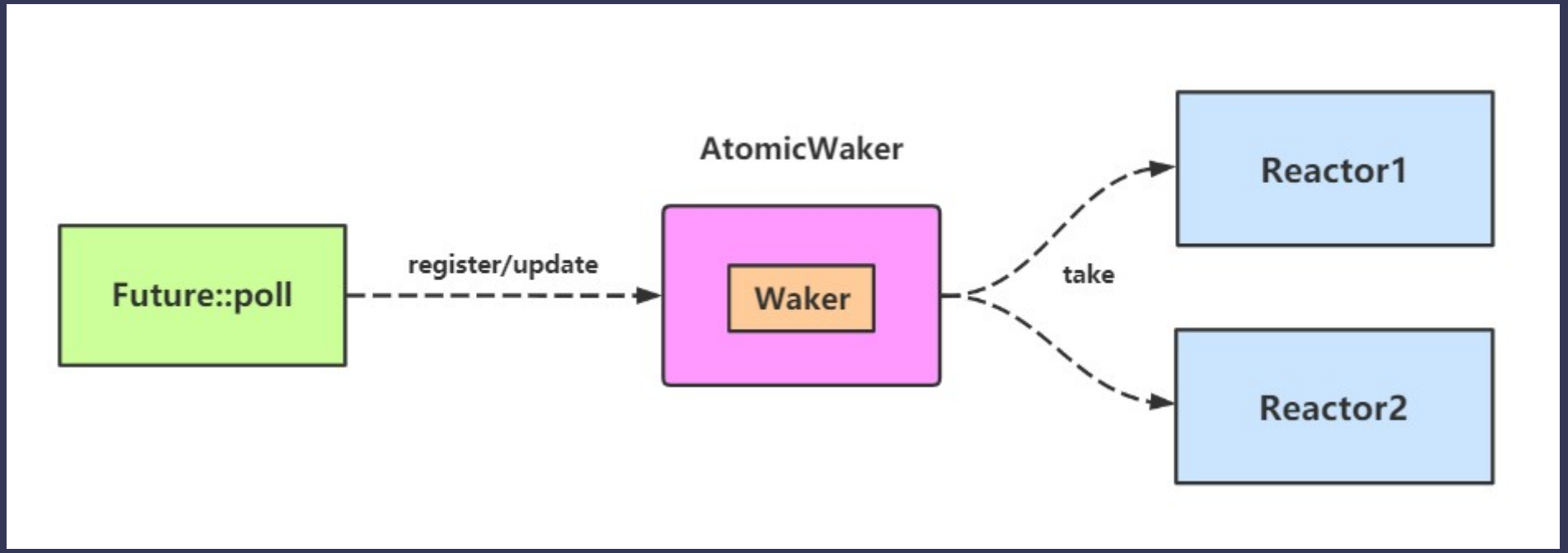
Future 和 Reactor 之间的并发

Waker 的过期：每次 poll 调用传递的 Waker 可能是不同的，因此每次 poll 都需要更新到 Reactor ，以确保能够唤醒 Task 。



Future 和 Reactor 之间的并发

AtomicWaker: spmc 并发模式， lock-free

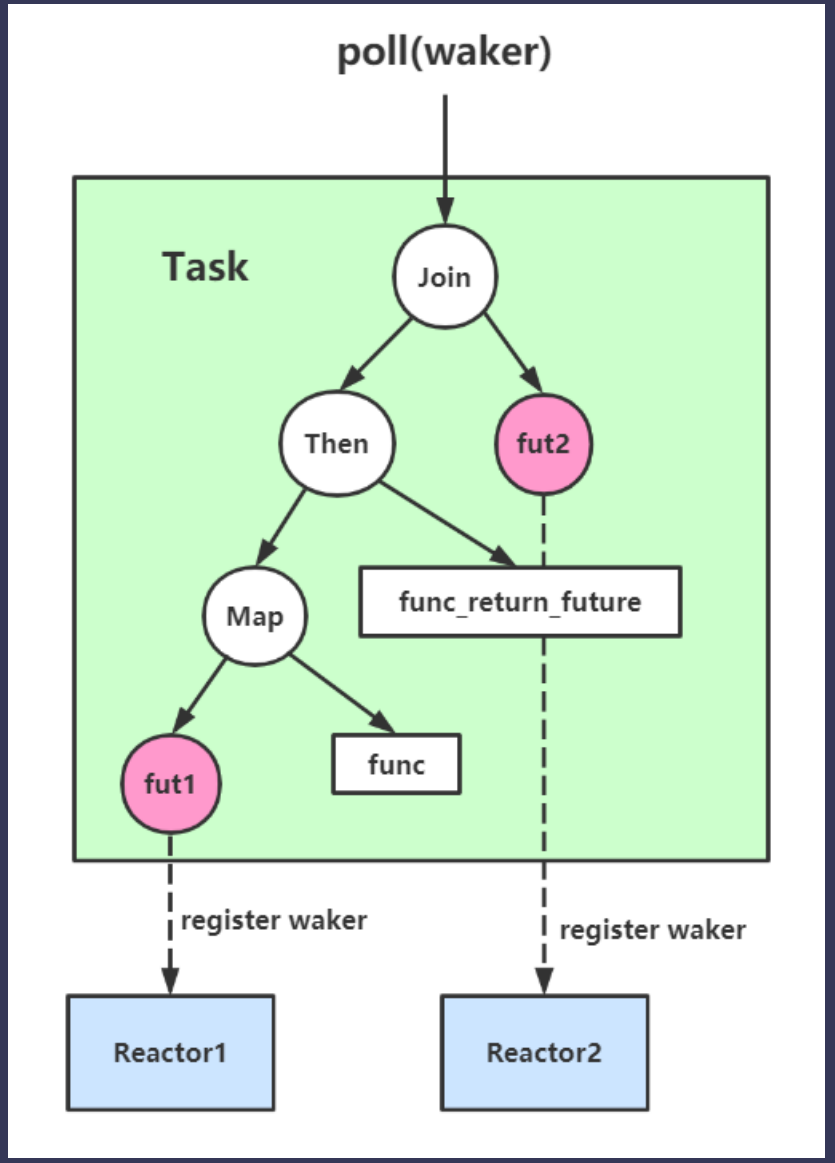


Future 的可组合性

```
1 future1
2    .map(func)
3    .then(func_return_future)
4    .join(future2);
```

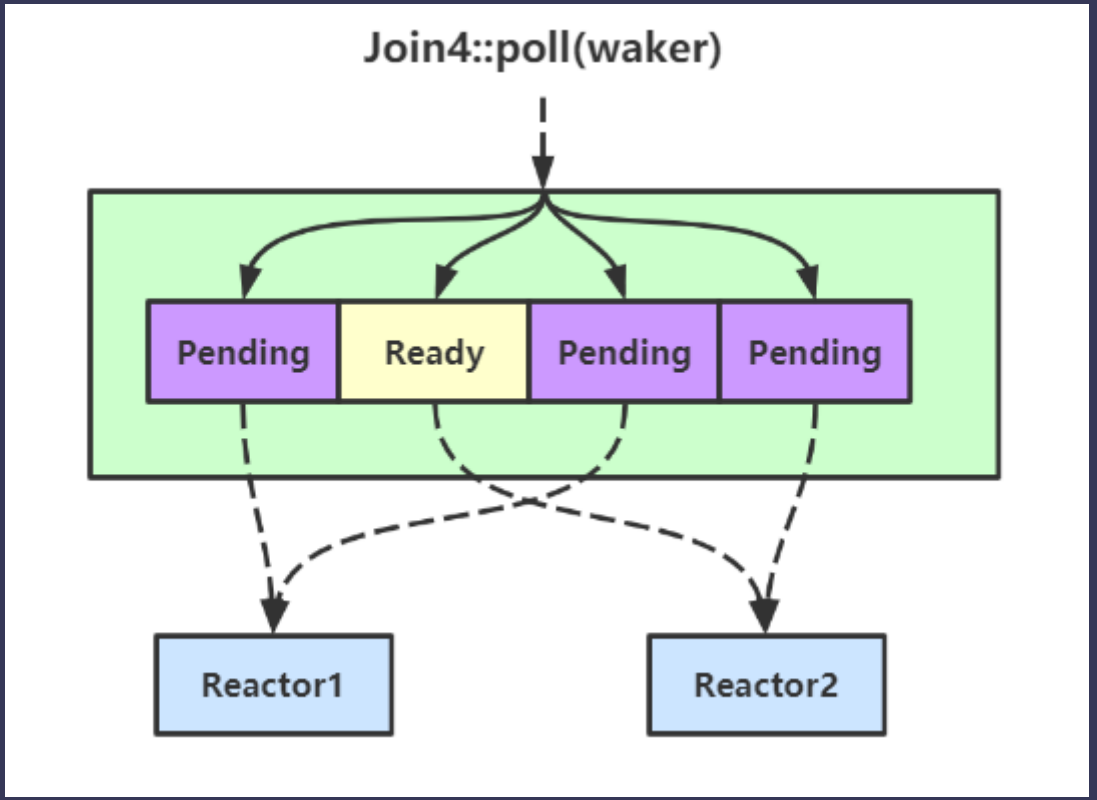
等价于

```
1 Join::new(
2   Then::new(
3      Map::new(future1, func),
4       func_return_future
5    ),
6    future2
7 );
```



JoinN 组合的效率

Waker 只用于唤醒 Task，没有携带为何唤醒的信息，因此 JoinN 组合在执行时只能挨个遍历内部的子 Future，不适用大规模的场合。

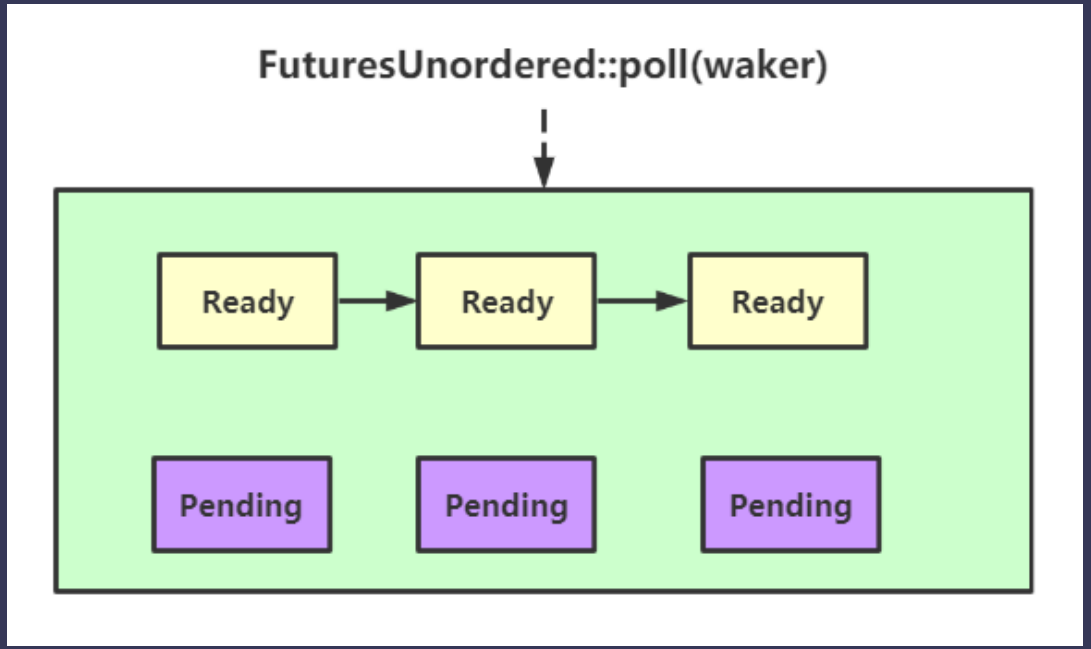


Waker 的拦截和包装

FuturesUnordered 内置一个并发队列，维护已经 ready 的子 Future

调用 poll 时将 Executor 传递的
waker 进行拦截包装

```
1 impl WrappedWaker {
2   fn wake(self) {
3     queue.enqueue(self.node);
4     self.parent_waker.wake();
5   }
6 }
```



异步任务间的同步

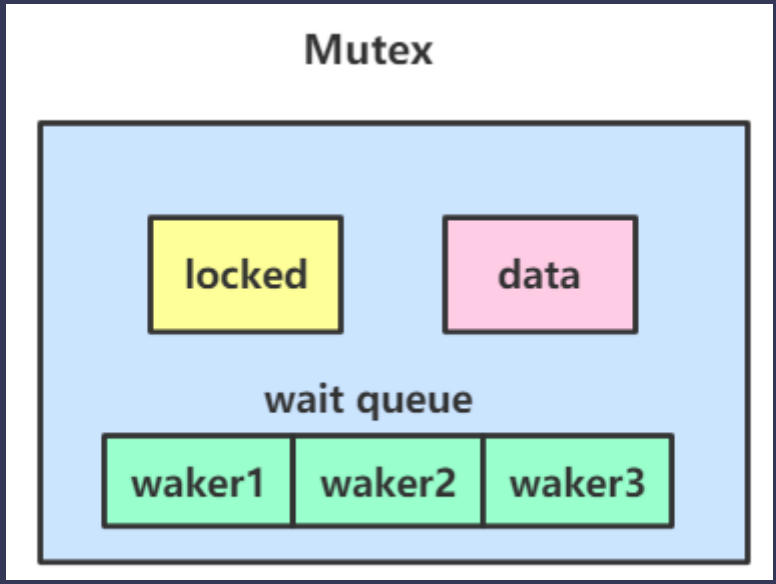
	线程	Task
睡眠	thread::park	return Pending
唤醒	Thread::unpark	Waker::wake
获取方式	thread::current()	poll 的参数

```rust
impl Future for SpinLock {
  type Output = ();
  fn poll(&mut self, waker:&Waker) -> Poll<()> {
    if self.locked.swap(true, AcqRel) {
      waker.wake_by_ref();
      Pending;
    } else {
      Ready(())
    }
  }
}
```

```rust
fn spin_lock(&mut self, waker:&Waker) {
  loop {
    let waker = thread::current();
    if self.locked.swap(true, AcqRel) {
      waker.unpark();
      thread::park();
    } else {
      return;
    }
  }
}
```

```rust
fn spin_lock(&self) {
  while self.locked.swap(true, AcqRel) {
    // thread::yield_now();
  }
}
```

异步任务间的同步 -Mutex

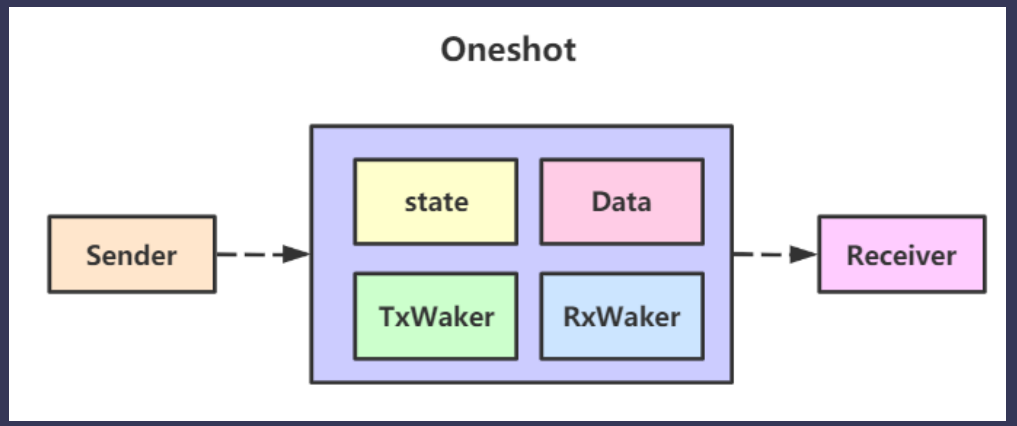


拿锁：尝试原子修改 locked， 失败则将 waker 注册 / 更新进等待队列

释放：修改 locked， 从队列中取一个唤醒

等待队列实现内部一般采用同步锁。因此如果锁保护的单纯是数据，那么应该优先使用同步锁。保护 IO 资源的时候需要使用易步锁。

异步任务间的同步 -Onshot



发送端

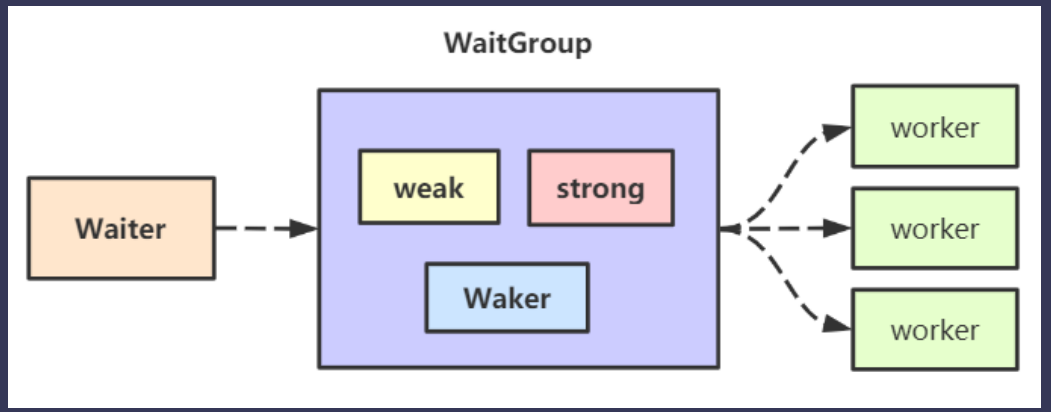
发送数据：设置 data 和原子修改 state， 通知 RxWaker
drop： 原子修改 state ， 通知 RxWaker

接收端

poll： 检查 state， 有数据则读取，没有则更新 RxWaker 和 state

drop： 原子修改 state ， 通知 RxWaker

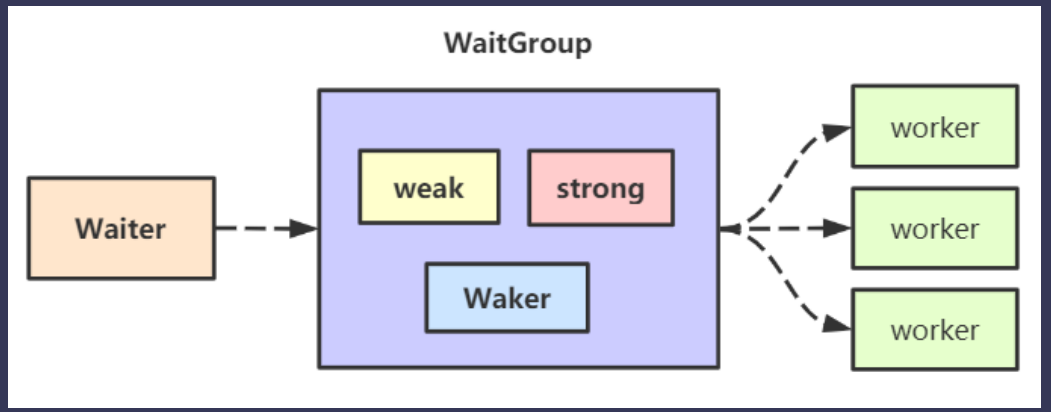
异步任务间的同步 -WaitGroup




```rust
use waitgroup::WaitGroup;
use async_std::task;
async {
    let wg = WaitGroup::new();
    for _ in 0..100 {
        let w = wg.worker();
      //thread::spawn(move || {
        task::spawn(async move {
          // do work
          drop(w); // drop w means task finished
        });
    }

    wg.wait().await;
}
```

异步任务间的同步 -WaitGroup




```rust
struct Inner {
  waker: AtomicWaker,
}
impl Drop for Inner {
  fn drop(&mut self) {
    self.waker.wake();
  }
}
```

```rust
#[derive(Clone)]
pub struct Worker {
  inner: Arc<Inner>,
}
pub struct WaitGroupFuture {
  inner: Weak<Inner>,
}
```

```rust
impl Future for WaitGroupFuture {
  type Output = ();

  fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
    match self.inner.upgrade() {
      Some(inner) => {
        inner.waker.register(cx.waker());
        Poll::Pending
      }
      None => Poll::Ready(()),
    }
  }
}
```

RUST CHINA CONF 2020

首届中国 Rust 开发者大会

2020.12.26–27 深圳