从头实现Rust异步执行器

stevenbai.top/rust/build_your_own_executor

2020/4/13

原文 Build your own executor

现在我们已经构建了block_on函数，是时候进一步将其转换为一个真正的执行器了。 我们希望我们的遗执行器不只是一次运行一个future，而是同时运行多个future！

这篇博文的灵感来自于 juliex，一个最小的执行器，作者也是Rust中的async/await功能的开拓者之一。 今天我们要从头开始写一个更现代、更清晰的 `juliex` 版本。

我们的执行器的目标是只使用简单和完全安全的代码，但是性能可以与现有的最佳执行器匹敌。

我们将用作依赖的crate包括 crossbeam、 async-task、 once_cell、 futures 和 num_cpus。

接口

执行器只有一个函数，就是运行一个future:

```
fn spawn<F, R>(future: F) -> JoinHandle<R>
where
    F: Future<Output = R> + Send + 'static,
    R: Send + 'static,
{
    todo!()
}
```

Rust

返回的JoinHandle是一种实现了Future的类型，在任务完成后可以取得其输出。

注意这个spawn()函数和 std::thread::spawn()之间的相似之处——它们几乎是等价的，除了一个产生异步任务，另一个产生线程。

下面是一个简单的例子，生成一个任务并等待它的输出:

```
fn main() {
    futures::executor::block_on(async {
        let handle = spawn(async { 1 + 2 });
        assert_eq!(handle.await, 3);
    });
}
```

Rust

将输出传递给JoinHandle

既然 JoinHandle是一个实现 Future 的类型，那么让我们暂先简单地将它定义为一个固定到堆上的future的别名:

```
type JoinHandle<R> = Pin<Box<dyn Future<Output = R> + Send>>;
```

Rust

这个方法目前可行，但是不要担心，稍后我们会将它作为一个新的结构清晰地重写，并手动实现 Future。

产生的 future 的输出必须以某种方式发送到 JoinHandle。 一种方法是创建一个 oneshot 通道，并在future完成时通过该通道发送输出。 那么 JoinHandle 就是一个等待来自通道的消息的future:

```
use futures::channel::oneshot;

fn spawn<F, R>(future: F) -> JoinHandle<R>
where
    F: Future<Output = R> + Send + 'static,
    R: Send + 'static,
{
    let (s, r) = oneshot::channel();
    let future = async move {
        let _ = s.send(future.await);
    };

    todo!()

    Box::pin(async { r.await.unwrap() })
}
```

Rust

下一步是在堆上分配future包装器，并将其推入某种全局任务队列，以便由执行程序处理。 我们称这种分配的future为一项任务。

任务的剖析

任务(task)包括future和它的状态。 我们需要跟踪状态，以了解任务是否计划运行、是否当前正在运行、是否已经完成等等。

下面是我们的Task类型的定义:

```
struct Task {
    state: AtomicUsize,
    future: Mutex<Pin<Box<dyn Future<Output = ()> + Send>>>,
}
```

Rust

我们还没有确定状态到底是什么，但它将是某种可以从任何线程更新的 AtomicUsize。 我们以后再说吧。

Future 的输出类型是()——这是因为 spawn ()函数将原始的 future 包装成一个将输出发送到 oneshot 通道，然后简单地返回()。

future被固定在堆上。 这是因为只有pin的future才能被轮询（poll）。 但是为什么它还被包装在Mutex中呢?

每个与任务相关联的 Waker 都会保存一个 Task 引用，这样它就可以通过将任务推入全局任务队列来唤醒任务。 问题就在这里: 任务实例在线程之间共享，但是轮询future需要对它的可变访问。 解决方案: 我们将future封装到互斥对象中，以获得对它的可变访问权。

如果这一切听起来让人困惑，不要担心，一旦我们完成了整个执行器，理解起来就会容易得多！

让我们来分配一个保存future和他的状态的Task来完成spawn函数:

```
fn spawn<F, R>(future: F) -> JoinHandle<R>
where
    F: Future<Output = R> + Send + 'static,
    R: Send + 'static,
{
    let (s, r) = oneshot::channel();
    let future = async move {
        let _ = s.send(future.await);
    };

    let task = Arc::new(Task {
        state: AtomicUsize::new(0),
        future: Mutex::new(Box::pin(future)),
    });
    QUEUE.send(task).unwrap();

    Box::pin(async { r.await.unwrap() })
}
```

Rust

一旦任务被分配，我们将其推入 QUEUE，这是一个包含可运行任务的全局队列。 Spawn ()函数现在已经完成，所以让我们接下来定义 QUEUE..。

执行器线程

因为我们正在构建一个执行器，所以必须有一个后台线程池，它从队列中获取可运行的任务并运行它们，即轮询它们的future。

让我们定义全局任务队列，并且在它第一次被初始化时产生执行线程池:

```
use crossbeam::channel;
use once_cell::sync::Lazy;

static QUEUE: Lazy<channel::Sender<Arc<Task>>> = Lazy::new(|| {
    let (sender, receiver) = channel::unbounded::<Arc<Task>>();

    for _ in 0..num_cpus::get().max(1) {
        let receiver = receiver.clone();
        thread::spawn(move || receiver.iter().for_each(|task| task.run()));
    }

    sender
});
```

Rust

非常简单——执行器线程实际上是一行代码！ 任务队列是一个无界通道，而执行器线程则从这个通道接收任务并运行每个任务.

执行器线程的数量等于系统上的核心数量，该核心数量由 `nums_cpus` 提供。

现在我们已经有了任务队列和线程池，最后一个需要实现的部分是run()方法。

任务执行

运行一个任务仅仅意味着轮询它的future。 我们已经从我们实现 `block_on()` 的前一篇博客文章中知道如何轮询future。

Run()方法如下所示:

```
impl Task {
    fn run(self: Arc<Task>) {
        let waker = todo!();

        let cx = &mut Context::from_waker(&waker);
        self.future.try_lock().unwrap().as_mut().poll(cx);
    }
}
```

Rust

请注意，我们需要锁定future，以获得可变访问权并对其进行轮询。 根据设计，没有其他线程会同时持有锁，因此try_lock()必须总是成功。

但是我们如何创造一个唤醒者呢？ 我们将像上次一样使用 `async_task::waker_fn()` ，但唤醒函数应该做什么呢?

我们不能就这样把一个 `Arc<Task>` 放到QUEUE中，以下是我们应该考虑的潜在竞争冲突:

- 如果一个任务已经完成了，然后被唤醒了怎么办? Waker生命周期会超过他关联的Future,并且我们也不想在队列中包含已经完成的任务.

- 如果一个任务在运行之前,连续被唤醒两次会怎么样? 我们不希望在队列中同一个任务出现两次.
- 如果一个任务正在运行的时候被唤醒了怎么办? 如果这时候将其加入队列中,另一个执行线程可能试图运行它,这将导致一个任务同时在两个线程上运行.

如果我们仔细想想，我们会想出两个简单的规则，优雅地解决所有这些问题:

1. 如果还没有被唤醒并且当前没有正在运行，唤醒函数会安排此任务
2. 如果一个任务正在运行时被唤醒,由当前执行器线程(当前正在运行这个future的那个线程)重新调度它.

让我们勾勒出这些规则:

```
impl Task {
    fn run(self: Arc<Task>) {
        let waker = async_task::waker_fn(|| {
            todo!("schedule if the task is not woken already and is not running");
        });

        let cx = &mut Context::from_waker(&waker);
        self.future.try_lock().unwrap().as_mut().poll(cx);

        todo!("schedule if the task was woken while running");
    }
}
```

Rust

还记得我们在 Task 中定义的 AtomicUsize 类型的状态字段吗？ 现在是时候在其中存储一些有用的数据了。 关于任务，有两条信息可以帮助我们实现唤醒: 1. 任务是否已经被唤醒 2. 任务是否正在运行

这两个值都是 true / false 值，我们可以在 state 字段中用两个位表示它们:

```
const WOKEN: usize = 0b01;
const RUNNING: usize = 0b10;
```

Rust

唤醒函数设置“WOKEN”位。 如果两个位先前都是0(即任务既没有被唤醒也没有运行)，那么我们通过将引用推入队列来调度任务:

```
let task = self.clone();
let waker = async_task::waker_fn(move || {
    if task.state.fetch_or(WOKEN, Ordering::SeqCst) == 0 {
        QUEUE.send(task.clone()).unwrap();
    }
});
```

Rust

在轮询future之前，我们取消了WOKEN位的设置，并设置了RUNNING位:

```
self.state.store(RUNNING, Ordering::SeqCst);
let cx = &mut Context::from_waker(&waker);
let poll = self.future.try_lock().unwrap().as_mut().poll(cx);
```

Rust

在轮询future之后，我们取消RUNNING位的设置，并检查先前的状态是否已经设置了 WOKEN 和 RUNNING 位(即任务在运行时被唤醒)。 如果是这样，我们重新安排任务:

有趣的是，如果任务完成了(即它的future不再是pending)，我们就会让它永远处于 `RUNNING` 状态。 这样future以后被唤醒后就不可能再次进入队列。

我们现在有了一个真正的执行器——在v1.rs中看到完整的实现。

一点魔法

如果您发现处理 `Task` 结构体及其状态转换很有挑战，我感同身受。 但也有好消息,这些工作都不需要你亲自做,使用 `asyc-task` 即可！

我们只需要用 `async_task::Task()` 替换 `Arc<Task>` ，并用 `async-task::JoinHandle<()>` 替换 `oneshot` 通道。

这就是我们如何简化生成:

```
type Task = async_task::Task<()>;

fn spawn<F, R>(future: F) -> JoinHandle<R>
where
    F: Future<Output = R> + Send + 'static,
    R: Send + 'static,
{
    let (task, handle) = async_task::spawn(future, |t| QUEUE.send(t).unwrap(),
());
    task.schedule();
    Box::pin(async { handle.await.unwrap() })
}
```

Rust

`async_task::spawn()` 接受三个参数: 1. 待运行的future 2. 一个将任务放入队列的调度函数. 该函数可能被唤醒器执行,也可能被 `run()` 在轮询future后执行. 3. 一个包含任意信息的tag,这个tag信息会保存在task中. 这篇博客中我们不考虑只是简单的保存 `()` ,也就是忽略它.

然后构造函数返回两个值: 1. `async_task::Task<()>` ,其中 `()` 就是刚刚传入的tag. 2. `async_task::JoinHandle<R, ()>` ,这里的 `()` 还是刚刚的tag. 这个JoinHandle是一个future,它完成的时候会返回一个 `Option<R>` . 当返回None的时候表示任务发生了panic或者被取消了.

如果您想知道 `schedule()` 方法，它只需调用任务上的 schedule 函数将其推入队列。 我们也可以自己将任务推入QUEUE——最终结果是相同的。

综上所述，我们最终得到了这个非常简单的执行器:

```
static QUEUE: Lazy<channel::Sender<Task>> = Lazy::new(|| {
    let (sender, receiver) = channel::unbounded::<Task>();

    for _ in 0..num_cpus::get().max(1) {
        let receiver = receiver.clone();
        thread::spawn(move || receiver.iter().for_each(|task| task.run()));
    }

    sender
});

type Task = async_task::Task<()>;
type JoinHandle<R> = Pin<Box<dyn Future<Output = R> + Send>>;

fn spawn<F, R>(future: F) -> JoinHandle<R>
where
    F: Future<Output = R> + Send + 'static,
    R: Send + 'static,
{
    let (task, handle) = async_task::spawn(future, |t| QUEUE.send(t).unwrap(),
());
    task.schedule();
    Box::pin(async { handle.await.unwrap() })
}
```

Rust

完整的代码可以在 v2.rs 中找到。

这里使用 `async_task::spawn()` 的好处不仅仅是简单。它也比我们自己写的Task更有效率，也更健壮。 举一个健壮性的例子， `async_task::Task` 在完成后立即删除未来，而不是等待任务的所有引用都失效后才删除。

除此之外， `async-task` 还提供了一些有用的特性，比如tags和cancellation，但是我们今天不讨论这些。 还值得一提的是，async-task 是一个 `#[no_std]` crate，甚至可以在没有标准库的情况下使用。

改进的JoinHandle

如果你仔细观察我们最新的执行器，还有一个效率低下的实例——JoinHandle的冗余 `Box::pin()` 分配。

如果我们可以使用下面的类型别名就更好了，但是我们不能，因为 `async_task::JoinHandle<R>` 输出 `Option<R>` ，而JoinHandle 输出R:

```
type JoinHandle<R> = async_task::JoinHandle<R, ()>;
```

Rust

我们只能将 `async_task::JoinHandle` 封装到一个新的结构体中，如果任务发生panic或者被取消，它也会panic: >这句话感觉说不通呢,需要看看async_task源码才行

```
struct JoinHandle<R>(async_task::JoinHandle<R, ()>);

impl<R> Future for JoinHandle<R> {
    type Output = R;

    fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>
{
        match Pin::new(&mut self.0).poll(cx) {
            Poll::Pending => Poll::Pending,
            Poll::Ready(output) => Poll::Ready(output.expect("task failed")),
        }
    }
}
```

Rust

完整的执行器实现可以在v3.rs中找到。

处理恐慌(panic)

到目前为止，我们还没有真正考虑过当任务感到恐慌时会发生什么，即调用 poll()时会发生恐慌。 现在 `run ()` 方法只是将恐慌传播到执行器中。 我们应该思考这是否是我们真正想要的。

明智的做法是以某种方式处理这些恐慌。 例如，我们可以简单地忽略恐慌，继续运行。 这样它们只是屏幕上打印信息，但不会崩溃整个进程——恐慌的线程的工作方式完全相同。

为了忽略恐慌，我们将run()包装成catch_unwind() :

```
use std::panic::catch_unwind;

static QUEUE: Lazy<channel::Sender<Task>> = Lazy::new(|| {
    let (sender, receiver) = channel::unbounded::<Task>();

    for _ in 0..num_cpus::get().max(1) {
        let receiver = receiver.clone();
        thread::spawn(move || {
            receiver.iter().for_each(|task| {
                let _ = catch_unwind(|| task.run());
            })
        });
    }

    sender
});
```

Rust

在v4.rs 中可以找到忽略恐慌的完整执行器代码。

有许多明智的应对恐慌的策略。下面是一些在 `async-task` 库中提供的例子:

- 忽略恐慌 – 恐慌直接被忽略,当 `JoinHandle<R>` 在await时也会发生恐慌
- 传播恐慌 panic被重新放入在等待 `JoinHandle<R>` 结果的那个任务中.
- 输出恐慌 `JoinHandle<R>` 输出 `std::thread::Result<R>` .

实现任何你想要的恐慌处理策略都是很容易的。 这完全由你来决定哪一个是最好的！

执行器的效率

当前的代码简短、简单、安全，但它有多快呢?

async_task::spawn ()分配的任务只是一个分配，存储任务状态、future以及future完成后的输出。 没有其他隐藏成本了——spawn的速度实际上已经到了极限！

其他执行器，如 async-std 和 tokio，分配任务的方式完全相同。 我们的执行器的基础本质上是一个最优的实现，现在我们离与流行的执行器竞争只有一步之遥: 任务窃取。

现在，所有的执行器线程共享相同的任务队列。 如果所有线程都在同时访问队列，则由于争用，性能将受到影响。 任务窃取背后的想法是为每个执行器线程分配一个不同的队列。 这样执行器线程只需要在自己的队列为空时从其他队列中窃取任务，这意味着争用只会很少发生，而不是一直发生。

我将在另一篇博客文章中更多地谈论任务窃取。

正确性

每个人都告诉我们，并发是困难的。 Go语言提供了一个内置的竞争检测器，tokio创建了自己的并发检查器 loom 来寻找并发错误，而crossbeam在某些情况下甚至采用了形式证明。 听起来很可怕！

但是我们可以坐下来，放松，不用担心。 竞争检测器，消毒器，甚至miri(译者 Miri是一个实验性的 Rust MIR解释器。它可以运行Rust二进制文件,对其进行测试,可以检查出某些未定义的行为)或loom，都不能在我们的遗嘱执行器上捕捉到bug。 原因是我们只编写了安全代码，而安全代码是内存安全的，也就是说它不能包含数据竞争。 Rust的类型系统已经证明我们的执行器是正确的。

确保内存安全的负担完全由依赖的crate承担，更确切地说是aysnc-task和crossbeam。 请放心，两者都非常重视正确性。 `async-task` 有一个覆盖所有边缘情况的广泛测试套件，crossbeam的通道有许多测试，甚至通过Go和std::sync::mpsc测试套件，工作窃取双向队列基于一个经过形式证明的实现，而基于epoch的垃圾收集器也有正确性证明。

适用于所有人的执行器

自从Alex和Aaron在2016年首次designed zero-cost futures以来，他们的计划就是每个spawn的future只进行一次内存分配:

> 每个“任务”需要一个分配，结果通常是每个连接需要一个分配。

然而，单次分配任务是一个善意的谎言——我们花了好几年才真正得到它们。 比如tokio 0.1版本中spawn时需要分配一个future，然后分配任务状态，最后分配一个oneshot通道。 也就是每个spawn三个分配点！

然后，在2019年8月,async-task诞生了。 有史以来第一次，我们成功地将future、任务状态和通道的分配压缩为单次分配。 之所以花费这么长时间，是因为任务内部的手动分配和状态转换管理非常复杂。 但是现在已经完成了，你再也不用担心任何事情了。

此后不久，在2019年10月，tokio也采用了类似于 async-task 的实现方法。

现在，任何人都可以通过单次分配任务来构建一个高效的执行器。 曾经的火箭科学现在已经不复存在了。

转载说明