从头实现Rust异步block_on

stevenbai.top/rust/build_your_own_block_on

2020/4/12

原文地址 Build your own block_on()

你是否曾经想过futures中的block_on是如何工作的呢?今天我们就来实现你自己的block_on版本.

这篇博客灵感应该来自两个crate, 分别是wakeful和extreme. `wakeful` 提供了一种简单的直接从一个函数创建Waker的方法, `extreme` 则提供了一种非常简洁的block_on实现.

我们的实现目标将与 `extreme` 略有不同。 我们不再追求零依赖性和最少的代码行数，而是追求一个安全、高效但仍然相当简单的实现。

我们将使用的依赖项有 pin-utils、 crossbeam 和 async-task。

签名

block_on的签名如下所示。 我们以一个 future 作为参数，在当前线程上运行它(当它pending时阻塞)，然后返回它的输出:

```
fn block_on<F: Future>(future: F) -> F::Output {
    todo!()
}
```

Rust

现在让我们实现遗漏的 todo! ()部分。

初次尝试

请注意，标准库的[Future]()中的poll的参数是一个 `Pin<&mut Future>` 。 所以我们需要先把它固定(Pin)住。 虽然有一种方法可以安全地使用 `Box::pin()` 来实现这一点，但是我们更愿意把Future放在栈上而不是堆上(译者注 Box::Pin会把Future分配到堆上,然后Pin)。

不幸的是，安全地将Future固定在栈上的唯一方法是使用 `pin-utils` :

```
pin_utils::pin_mut!(future);
```

Rust

pin_mut这个宏会将一个类型为 `F` 的 `future` 转换为 `Pin<&mut F>` ,并将其固定在栈上.

接下来我们需要详细说明当这个 `future` 被唤醒时会发生什么。 在这种情况下，唤醒应该只是解除运行 `future` 的线程的阻塞。

构造一个唤醒器可能很丑陋——只要看一眼 `extreme` 的实现就可以了。 这是手工构建 `Waker` 最简单的方法！ 到处都是原始指针和不安全的代码... ... 让我们暂时跳过这一部分，以后再填空。

```
let waker = todo!();
```

Rust

最后，我们从 `Waker` 创建一个任务上下文，并在循环中轮询 `future` 。 如果完成，返回输出。 如果它挂起，阻止当前线程:

```
let cx = &mut Context::from_waker(&waker);
loop {
    match future.as_mut().poll(cx) {
        Poll::Ready(output) => return output,
        Poll::Pending => thread::park(),
    }
}
```

Rust

请别对Contex类型感到困惑,它就是 Waker 的一个包装器——没有什么比这更好的了。 当在 Rust 中设计 async / await 时，我们不确定除了传递 Waker给poll ()之外，传递其他任何东西是否有用，所以我们想出了这个包装器，它可能在 Rust 的未来版本中包含更多的东西。

不管怎样... 我们差不多完成了。让我们回到刚才的block_on，并将todo替换为上面的代码。

仔细想想，Waker 实际上是 `Arc<dyn Fn () + Send + Sync>` 的精心优化版本，wake() 调用这个函数。 换句话说，Waker 是一个回调函数，当将来可以继续执行时，它就会被调用。

因为 Waker 是如此难以构造，所以 sagebind 提供了 waker_fn() ，这是一种将任何函数转换为 Waker 的直接方法。 不幸的是， `wakeful` 似乎此刻被猛拉，所以我借用了 `wakerfn()` 并将其放入我的 `async-task` 中。

在我们的代码块中，回调函数就是唤醒future所在线程:

```
let thread = thread::current();
let waker = async_task::waker_fn(move || thread.unpark());
```

Rust

简单多了! 比起处理RawWaker 和 RawWakerVTable 好多了。

在内部，waker_fn()构造函数实际上创建了 `Arc<impl Fn () + Send + Sync>` ，然后用不安全的代码将其转换为 Waker，这些代码看起来与我们在extreme中看到的类似。

下面是对block_on的完整实现:

```
fn block_on<F: Future>(future: F) -> F::Output {
    pin_utils::pin_mut!(future);

    let thread = thread::current();
    let waker = async_task::waker_fn(move || thread.unpark());

    let cx = &mut Context::from_waker(&waker);
    loop {
        match future.as_mut().poll(cx) {
            Poll::Ready(output) => return output,
            Poll::Pending => thread::park(),
        }
    }
}
```

Rust

完整代码见v1.rs

park问题

但是，现在还不是庆祝的时候。 有个问题。 如果future的用户代码也使用 park / unpark API，它可能会从回调函数中获取并“偷取” unpark 通知。 阅读这个问题可以得到更详细的解释。

> 译者注 说白了就是因为都在用unpark,会让导致不该唤醒的时候被唤醒,或者收不到唤醒通知.

一个可能的解决方案是使用一种不同于 std: : thread 模块中的线程的park/unpark方法。这样，future的代码就不会干扰唤醒。

在crossbeam中有一个非常类似的 park / unpark 机制，只不过它允许我们创建任意多的独立的 parkers，而不是每个线程都有一个。 让我们为block_on的每一次调都独立创建一个parker:

```
fn block_on<F: Future>(future: F) -> F::Output {
    pin_utils::pin_mut!(future);

    let parker = Parker::new();
    let unparker = parker.unparker().clone();
    let waker = async_task::waker_fn(move || unparker.unpark());

    let cx = &mut Context::from_waker(&waker);
    loop {
        match future.as_mut().poll(cx) {
            Poll::Ready(output) => return output,
            Poll::Pending => parker.park(),
        }
    }
}
```

Rust

就这样! 问题解决了。 完整的代码在v2.rs,你可以去运行他.

一个优化

创建一个 Parker 和 Waker 并不是免费的——这两者都会引起内存分配， 我们能改进吗?

为什么不在线程本地存储器中缓存它们，而不是在每次调用block_on时构造 Parker 和 Waker 呢? 这样，线程就可以在 block on ()的所有调用中重用相同的实例:

```
fn block_on<F: Future>(future: F) -> F::Output {
    pin_utils::pin_mut!(future);

    thread_local! {
        static CACHE: (Parker, Waker) = {
            let parker = Parker::new();
            let unparker = parker.unparker().clone();
            let waker = async_task::waker_fn(move || unparker.unpark());
            (parker, waker)
        };
    }

    CACHE.with(|(parker, waker)| {
        let cx = &mut Context::from_waker(&waker);
        loop {
            match future.as_mut().poll(cx) {
                Poll::Ready(output) => return output,
                Poll::Pending => parker.park(),
            }
        }
    })
}
```

Rust

如果future能够快速执行，这个小小的更改将使 block on ()效率大大提升！

完整的代码在v3.rs中

如何处理递归

是否还有其他问题?

如果future在block_on的内部块再次递归地调用block_on会怎样? 当然我们可以允许也可以禁止递归。

如果我们选择允许递归，那么我们还需要确保 block on ()的递归调用不共享相同的 Parker 和 Waker 实例，否则就无法知道哪个 block on ()调用会被唤醒。

futures的block on 发生递归调用会panic。 对于允许递归还是禁止递归，我没有强烈的意见—- 这两种行为都是明智的。 但是，既然我们在模仿futures的版本，那么就不要使用递归。

为了检测递归调用，我们可以引入另一个线程本地变量来指示我们当前是否在 block on() 中。 但这是一个很大的工作量。

这里有一个很酷的技巧，它只需对代码进行更少的更改。 让我们把(Parker，Waker)包装到 RefCell 中，如果多次发生可变的借用，程序就会panic:

```
fn block_on<F: Future>(future: F) -> F::Output {
    pin_utils::pin_mut!(future);

    thread_local! {
        static CACHE: RefCell<(Parker, Waker)> = {
            let parker = Parker::new();
            let unparker = parker.unparker().clone();
            let waker = async_task::waker_fn(move || unparker.unpark());
            RefCell::new((parker, waker))
        };
    }

    CACHE.with(|cache| {
        let (parker, waker) = &mut *cache.try_borrow_mut().ok()
            .expect("recursive `block_on`");

        let cx = &mut Context::from_waker(&waker);
        loop {
            match future.as_mut().poll(cx) {
                Poll::Ready(output) => return output,
                Poll::Pending => parker.park(),
            }
        }
    })
}
```

Rust

终于。 现在我们真的结束了，我保证！ 最终的实现是正确的，健壮的，高效的。 差不多吧。 :)

完整的代码见v4.rs

Benchmarks

为了测试block_on的效率，让我们将它与futures的进行基准测试比较。

但是首先，我们将编写一个 helper future 类型，它可以反复被轮询，直到完成:

```
struct Yields(u32);

impl Future for Yields {
    type Output = ();

    fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<()> {
        if self.0 == 0 {
            Poll::Ready(())
        } else {
            self.0 -= 1;
            cx.waker().wake_by_ref();
            Poll::Pending
        }
    }
}
```

Rust

举个例子，为了测试轮询十次的性能,可以这样写:

```
#[bench]
fn custom_block_on_10_yields(b: &mut Bencher) {
    b.iter(|| block_on(Yields(10)));
}
```

Rust

让我们设定一组三个基准，轮询次数分别为0、10和50。 我们使用自定义的block_on，然后使用 futures 的。 您可以在 yield.rs 中找到完整的基准测试代码。

下面是我的机器上的结果:

```
test custom_block_on_0_yields   ... bench:           3 ns/iter (+/- 0)
test custom_block_on_10_yields  ... bench:         130 ns/iter (+/- 12)
test custom_block_on_50_yields  ... bench:         638 ns/iter (+/- 20)

test futures_block_on_0_yields  ... bench:          10 ns/iter (+/- 0)
test futures_block_on_10_yields ... bench:         236 ns/iter (+/- 10)
test futures_block_on_50_yields ... bench:       1,139 ns/iter (+/- 30)
```

结果显示，在这个特定的基准测试中，我们的block_on比futures的大约快2到3倍，这一点都不差！

结论

Async Rust 之所以令人生畏，是因为它包含了太多的机制: Future trait、 pinning、Context、 Waker 及其相关的RawWaker 和 RawWakerVTable、 Async 和 await 的语法糖背后的机制、不安全的代码、原始指针等等。

但问题是，许多丑陋的东西甚至不是那么重要——实际上它们只是无聊的样板文件，可以用像 pin-utils、async-task和 crossbeam这样的crate绕开。

实际上，今天我们已经在几行安全代码中构建了一个高效的block_on，而不需要理解大部分的样板文件。 在另一篇博文中，我们将建立一个真正的执行器..。

转载说明