异步编程

jask

09/28/2024

一些模型

OS 线程，它最简单，也无需改变任何编程模型（业务/代码逻辑），因此非常适合作为语言的原生并发模型，我们在多线程章节也提到过，Rust 就选择了原生支持线程级的并发编程。但是，这种模型也有缺点，例如线程间的同步将变得更加困难，线程间的上下文切换损耗较大。使用线程池在一定程度上可以提升性能，但是对于 IO 密集的场景来说，线程池还是不够。

事件驱动（Event driven），这个名词你可能比较陌生，如果说事件驱动常常跟回调（Callback）一起使用，相信大家就恍然大悟了。这种模型性能相当的好，但最大的问题就是存在回调地狱的风险：非线性的控制流和结果处理导致了数据流向和错误传播变得难以掌控，还会导致代码可维护性和可读性的大幅降低，大名鼎鼎的 JavaScript 曾经就存在回调地狱。

协程（Coroutines）可能是目前最火的并发模型，Go 语言的协程设计就非常优秀，这也是 Go 语言能够迅速火遍全球的杀手锏之一。协程跟线程类似，无需改变编程模型，同时，它也跟 async 类似，可以支持大量的任务并发运行。但协程抽象层次过高，导致用户无法接触到底层的细节，这对于系统编程语言和自定义异步运行时是难以接受的

actor 模型是 erlang 的杀手锏之一，它将所有并发计算分割成一个一个单元，这些单元被称为 actor，单元之间通过消息传递的方式进行通信和数据传递，跟分布式系统的设计理念非常相像。由于 actor 模型跟现实很贴近，因此它相对来说更容易实现，但是一旦遇到流控制、失败重试等场景时，就会变得不太好用

async/await，该模型性能高，还能支持底层编程，同时又像线程和协程那样无需过多的改变编程模型，但有得必有失，async 模型的问题就是内部实现机制过于复杂，对于用户来说，理解和使用起来也没有线程和协程简单，好在前者的复杂性开发者们已经帮我们封装好，而理解和使用起来不够简单，正是本章试图解决的问题。

Rust 实现的 Async 与 JS 有所区别

Future 在 Rust 中是惰性的，只有在被轮询（poll）时才会运行，因此丢弃一个 future 会阻止它未来再被运行，你可以将 Future 理解为一个在未来某个时间点被调度执行的任务。

Async 在 Rust 中使用开销是零，意味着只有你能看到的代码（自己的代码）才有性能损耗，你看不到的（async 内部实现）都没有性能损耗，例如，你可以无需分配任何堆内存、也无需任何动态分发来使用 async，这对于热点路径的性能有非常大的好处，正是得益于此，Rust 的异步编程性能才会这么高。

Rust 没有内置异步调用所必需的运行时，但是无需担心，Rust 社区生态中已经提供了非常优异的运行时实现，例如大明星 tokio

运行时同时支持单线程和多线程，这两者拥有各自的优缺点，稍后会讲

Rust: async vs 多线程

虽然 async 和多线程都可以实现并发编程，后者甚至还能通过线程池来增强并发能力，但是这两个方式并不互通

OS 线程非常适合少量任务并发，因为线程的创建和上下文切换是非常昂贵的，甚至于空闲的线程都会消耗系统资源。虽说线程池可以有效的降低性能损耗，但是也无法彻底解决问题。当然，线程模型也有其优点，例如它不会破坏你的代码逻辑和编程模型，你之前的顺序代码，经过少量修改适配后依然可以在新线程中直接运行，同时在某些操作系统中，你还可以改变线程的优先级，这对于实现驱动程序或延迟敏感的应用（例如硬实时系统）很有帮助。

对于长时间运行的 CPU 密集型任务，例如并行计算，使用线程将更有优势。这种密集任务往往会让所在的线程持续运行，任何不必要的线程切换都会带来性能损耗，因此高并发反而在此时成为了一种多余。同时你所创建的线程数应该等于 CPU 核心数，充分利用 CPU 的并行能力，甚至还可以将线程绑定到 CPU 核心上，进一步减少线程上下文切换。

而高并发更适合 IO 密集型任务，例如 web 服务器、数据库连接等等网络服务，因为这些任务绝大部分时间都处于等待状态，如果使用多线程，那线程大量时间会处于无所事事的状态，再加上线程上下文切换的高昂代价，让多线程做 IO 密集任务变成了一件非常奢侈的事。而使用 async，既可以有效的降低 CPU 和内存的负担，又可以让大量的任务并发的运行，一个任务一旦处于 IO 或者其他等待（阻塞）状态，就会被立刻切走并执行另一个任务，而这里的任务切换的性能开销要远远低于使用多线程时的线程上下文切换。

事实上，async 底层也是基于线程实现，但是它基于线程封装了一个运行时，可以将多个任务映射到少量线程上，然后将线程切换变成了任务切换，后者仅仅是内存中的访问，因此要高效的多。

不过 async 也有其缺点，原因是编译器会为 async 函数生成状态机，然后将整个运行时打包进来，这会造成我们编译出的二进制可执行文件体积显著增大。

总结

有大量 IO 任务需要并发运行时，选 async 模型

有部分 IO 任务需要并发运行时，选多线程，如果想要降低线程创建和销毁的开销，可以使用线程池

有大量 CPU 密集任务需要并行运行时，例如并行计算，选多线程模型，且让线程数等于或者稍大于 CPU 核心数

无所谓时，统一选多线程

Future Trait

首先，来给出 Future 的定义：它是一个能产出值的异步计算（虽然该值可能为空，例如 () ）。

我们来看看一个简化版的 Future 特征：

```
trait SimpleFuture{
  type Output;
  fn poll(&mut self,wake:fn())->Poll<Self::Output>;
}
enum Poll<T>{
  Ready(T),
  Pending,
}
```

我们提到过 Future 需要被执行器 poll(轮询) 后才能运行，诺，这里 poll 就来了，通过调用该方法，可以推进 Future 的进一步执行，直到被切走为止（ 这里不好理解，但是你只需要知道 Future 并不能保证在一次 poll 中就被执行完，后面会详解介绍)。

若在当前 poll 中，Future 可以被完成，则会返回 Poll::Ready(result) ，反之则返回 Poll::Pending，并且安排一个 wake 函数：当未来 Future 准备好进一步执行时，该函数会被调用，然后管理该 Future 的执行器（例如 futures 中的 block_on 函数）会再次调用 poll 方法，此时 Future 就可以继续执行了。

如果没有 wake 方法，那执行器无法知道某个 Future 是否可以继续被执行，除非执行器定期的轮询每一个 Future，确认它是否能被执行，但这种作法效率较低。而有了 wake，Future 就可以主动通知执行器，然后执行器就可以精确的执行该 Future。这种“事件通知 -> 执行”的方式要远比定期对所有 Future 进行一次全遍历来的高效。

下面的 SocketRead 结构体就是一个 Future：

```
pub struct SocketRead<'a> {
    socket: &'a Socket,
}

impl SimpleFuture for SocketRead<'_> {
    type Output = Vec<u8>;

    fn poll(&mut self, wake: fn()) -> Poll<Self::Output> {
        if self.socket.has_data_to_read() {
            // socket 有数据，写入 buffer 中并返回
            Poll::Ready(self.socket.read_buf())
        } else {
            // socket 中还没数据
            //
            // 注册一个`wake`函数，当数据可用时，该函数会被调用，
            // 然后当前 Future 的执行器会再次调用`poll`方法，此时就可以读取到数据
            self.socket.set_readable_callback(wake);
            Poll::Pending
        }
    }
}
```

这种 Future 模型允许将多个异步操作组合在一起，同时还无需任何内存分配。不仅仅如此，如果你需要同时运行多个 Future 或链式调用多个 Future ，也可以通过无内存分配的状态机实现，例如：

```rust
trait SimpleFuture {
    type Output;
    fn poll(&mut self, wake: fn()) -> Poll<Self::Output>;
}

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

/// 一个 SimpleFuture，它会并发地运行两个 Future 直到它们完成
///
/// 之所以可以并发，是因为两个 Future 的轮询可以交替进行，一个阻塞，另一个就可以立刻执行，反之亦然
pub struct Join<FutureA, FutureB> {
    // 结构体的每个字段都包含一个 Future，可以运行直到完成.
    // 等到 Future 完成后，字段会被设置为 `None`. 这样 Future 完成后，就不会再被轮询
    a: Option<FutureA>,
    b: Option<FutureB>,
}

impl<FutureA, FutureB> SimpleFuture for Join<FutureA, FutureB>
where
    FutureA: SimpleFuture<Output = ()>,
    FutureB: SimpleFuture<Output = ()>,
{
    type Output = ();
    fn poll(&mut self, wake: fn()) -> Poll<Self::Output> {
        // 尝试去完成一个 Future `a`
        if let Some(a) = &mut self.a {
            if let Poll::Ready(()) = a.poll(wake) {
                self.a.take();
            }
        }

        // 尝试去完成一个 Future `b`
        if let Some(b) = &mut self.b {
            if let Poll::Ready(()) = b.poll(wake) {
                self.b.take();
            }
        }

        if self.a.is_none() && self.b.is_none() {
            // 两个 Future 都已完成 - 我们可以成功地返回了
            Poll::Ready(())
        } else {
            // 至少还有一个 Future 没有完成任务，因此返回 `Poll::Pending`.
            // 当该 Future 再次准备好时，通过调用`wake()`函数来继续执行
            Poll::Pending
        }
    }
}
```

真实的 Future Trait

```rust
trait Future {
    type Output;
    fn poll(
        // 首先值得注意的地方是，`self`的类型从`&mut self`变成了`Pin<&mut Self>`:
        self: Pin<&mut Self>,
        // 其次将`wake: fn()`修改为 `cx: &mut Context<'_>`:
        cx: &mut Context<'_>,
```

```
    ) -> Poll<Self::Output>;
}
```

首先这里多了一个 `Pin` ，关于它我们会在后面章节详细介绍，现在你只需要知道使用它可以创建一个无法被移动的 `Future` ，因为无法被移动，所以它将具有固定的内存地址，意味着我们可以存储它的指针（如果内存地址可能会变动，那存储指针地址将毫无意义！），也意味着可以实现一个自引用数据结构： `struct MyFut { a: i32, ptr_to_a: *const i32 }`。而对于 `async/await` 来说，`Pin` 是不可或缺的关键特性。

其次，从 `wake: fn()` 变成了 `&mut Context<' _>` 。意味着 `wake` 函数可以携带数据了，为何要携带数据？考虑一个真实世界的场景，一个复杂应用例如 `web` 服务器可能有数千连接同时在线，那么同时就有数千 `Future` 在被同时管理着，如果不能携带数据，当一个 `Future` 调用 `wake` 后，执行器该如何知道是哪个 `Future` 调用了 `wake` ，然后进一步去 `poll` 对应的 `Future` ？没有办法！那之前的例子为啥就可以使用没有携带数据的 `wake` ？因为足够简单，不存在歧义性。

总之，在正式场景要进行 `wake` ，就必须携带上数据。而 `Context` 类型通过提供一个 `Waker` 类型的值，就可以用来唤醒特定的的任务。

Waker

对于 `Future` 来说，第一次被 `poll` 时无法完成任务是很正常的。但它需要确保在未来一旦准备好时，可以通知执行器再次对其进行 `poll` 进而继续往下执行，该通知就是通过 `Waker` 类型完成的。

`Waker` 提供了一个 `wake()` 方法可以用于告诉执行器：相关的任务可以被唤醒了，此时执行器就可以对相应的 `Future` 再次进行 `poll` 操作。

Executor

`Rust` 的 `Future` 是惰性的：只有屁股上拍一拍，它才会努力动一动。其中一个推动它的方式就是在 `async` 函数中使用`.await` 来调用另一个 `async` 函数，但是这个只能解决 `async` 内部的问题，那么这些最外层的 `async` 函数，谁来推动它们运行呢？答案就是我们之前多次提到的执行器 `executor` 。

执行器需要从一个消息通道（ `channel` ）中拉取事件，然后运行它们。当一个任务准备好后（可以继续执行），它会将自己放入消息通道中，然后等待执行器 `poll` 。

执行器会管理一批 `Future`（最外层的 `async` 函数），然后通过不停地 `poll` 推动它们直到完成。最开始，执行器会先 `poll` 一次 `Future` ，后面就不会主动去 `poll` 了，而是等待 `Future` 通过调用 `wake` 函数来通知它可以继续，它才会继续去 `poll` 。这种 `wake` 通知然后 `poll` 的方式会不断重复，直到 `Future` 完成。

Pin 和 Unpin

在 `Rust` 中，所有的类型可以分为两类：

类型的值可以在内存中安全地被移动，例如数值、字符串、布尔值、结构体、枚举，总之你能想到的几乎所有类型都可以落入到此范畴内

自引用类型，大魔王来了，大家快跑，在之前章节我们已经见识过它的厉害

```
struct SelfRef{
  value: String,
  pointer_to_value: *mut String,
}
```

在上面的结构体中，`pointer_to_value` 是一个裸指针，指向第一个字段 `value` 持有的字符串 `String` 。很简单对吧？现在考虑一个情况，若 `String` 被移动了怎么办？

此时一个致命的问题就出现了：新的字符串的内存地址变了，而 `pointer_to_value` 依然指向之前的地址，一个重大 `bug` 就出现了！

灾难发生，英雄在哪？只见 `Pin` 闪亮登场，它可以防止一个类型在内存中被移动。再来回忆下之前在 `Future` 章节中，我们提到过在 `poll` 方法的签名中有一个 `self: Pin<&mut Self>` ，那么为何要在这里使用 `Pin` 呢？

为什么需要 Pin

其实 `Pin` 还有一个小伙伴 `UnPin` ，与前者相反，后者表示类型可以在内存中安全地移动。

Unpin

事实上，绝大多数类型都不在意是否被移动（开篇提到的第一种类型），因此它们都自动实现了 `Unpin` 特征。

从名字推测，大家可能以为 `Pin` 和 `Unpin` 都是特征吧？实际上，`Pin` 不按套路出牌，它是一个结构体：

```rust
pub struct Pin<P> {
    pointer: P,
}
```

它包裹一个指针，并且能确保该指针指向的数据不会被移动，例如 `Pin<&mut T>` , `Pin<&T>` , `Pin<Box>` ，都能确保 `T` 不会被移动。

而 `Unpin` 才是一个特征，它表明一个类型可以随意被移动，那么问题来了，可以被 `Pin` 住的值，它有没有实现什么特征呢？答案很出乎意料，可以被 `Pin` 住的值实现的特征是`!Unpin` ，大家可能之前没有见过，但是它其实很简单，`!` 代表没有实现某个特征的意思，`!Unpin` 说明类型没有实现 `Unpin` 特征，那自然就可以被 `Pin` 了。

那是不是意味着类型如果实现了 `Unpin` 特征，就不能被 `Pin` 了？其实，还是可以 `Pin` 的，毕竟它只是一个结构体，你可以随意使用，但是不再有任何效果而已，该值一样可以被移动！

例如 `Pin<&mut u8>` ，显然 `u8` 实现了 `Unpin` 特征，它可以在内存中被移动，因此 `Pin<&mut u8>` 跟 `&mut u8` 实际上并无区别，一样可以被移动。

因此，一个类型如果不能被移动，它必须实现`!Unpin` 特征。

Pin 的实际应用

将值固定在栈上

```rust
use std::pin::Pin;
use std::marker::PhantomPinned;

#[derive(Debug)]
struct Test {
    a: String,
    b: *const String,
    _marker: PhantomPinned,
}


impl Test {
    fn new(txt: &str) -> Self {
        Test {
            a: String::from(txt),
            b: std::ptr::null(),
            _marker: PhantomPinned, // 这个标记可以让我们的类型自动实现特征`!Unpin`
        }
    }

    fn init(self: Pin<&mut Self>) {
        let self_ptr: *const String = &self.a;
        let this = unsafe { self.get_unchecked_mut() };
        this.b = self_ptr;
    }

    fn a(self: Pin<&Self>) -> &str {
        &self.get_ref().a
    }

    fn b(self: Pin<&Self>) -> &String {
        assert!(!self.b.is_null(), "Test::b called without Test::init being called first");
        unsafe { &*(self.b) }
    }
}
```

可以解决指针指向的数据被移动的问题。

上面代码中，我们使用了一个标记类型 `PhantomPinned` 将自定义结构体 `Test` 变成了`!Unpin` (编译器会自动帮我们实现)，因此该结构体无法再被移动。

一旦类型实现了`!Unpin` ，那将它的值固定到栈 (`stack`) 上就是不安全的行为，因此在代码中我们使用了 `unsafe` 语句块来进行处理，你也可以使用 `pin_utils` 来避免 `unsafe` 的使用。

固定到堆上

将一个!Unpin 类型的值固定到堆上，会给予该值一个稳定的内存地址，它指向的堆中的值在 Pin 后是无法被移动的。而且与固定在栈上不同，我们知道堆上的值在整个生命周期内都会被稳稳地固定住。

```
use std::pin::Pin;
use std::marker::PhantomPinned;

#[derive(Debug)]
struct Test {
    a: String,
    b: *const String,
    _marker: PhantomPinned,
}

impl Test {
    fn new(txt: &str) -> Pin<Box<Self>> {
        let t = Test {
            a: String::from(txt),
            b: std::ptr::null(),
            _marker: PhantomPinned,
        };
        let mut boxed = Box::pin(t);
        let self_ptr: *const String = &boxed.as_ref().a;
        unsafe { boxed.as_mut().get_unchecked_mut().b = self_ptr };

        boxed
    }

    fn a(self: Pin<&Self>) -> &str {
        &self.get_ref().a
    }

    fn b(self: Pin<&Self>) -> &String {
        unsafe { &*(self.b) }
    }
}

pub fn main() {
    let test1 = Test::new("test1");
    let test2 = Test::new("test2");

    println!("a: {}, b: {}",test1.as_ref().a(), test1.as_ref().b());
    println!("a: {}, b: {}",test2.as_ref().a(), test2.as_ref().b());
}
```

将固定住的 Future 变为 Unpin

之前的章节我们有提到 async 函数返回的 Future 默认就是!Unpin 的。

但是，在实际应用中，一些函数会要求它们处理的 Future 是 Unpin 的，此时，若你使用的 Future 是!Unpin 的，必须要使用以下的方法先将 Future 进行固定：

Box::pin，创建一个 Pin<Box<T>>
pin_utils::pin_mut!，创建一个 Pin<&mut T>

固定后获得的 Pin<Box> 和 Pin<&mut T> 既可以用于 Future，又会自动实现 Unpin。

```
use pin_utils::pin_mut; // `pin_utils` 可以在 crates.io 中找到

// 函数的参数是一个`Future`，但是要求该`Future`实现`Unpin`
fn execute_unpin_future(x: impl Future<Output = ()> + Unpin) { /* ... */ }

let fut = async { /* ... */ };
```

```rust
// 下面代码报错：默认情况下，`fut` 实现的是`!Unpin`，并没有实现`Unpin`
// execute_unpin_future(fut);

// 使用`Box`进行固定
let fut = async { /* ... */ };
let fut = Box::pin(fut);
execute_unpin_future(fut); // OK

// 使用`pin_mut!`进行固定
let fut = async { /* ... */ };
pin_mut!(fut);
execute_unpin_future(fut); // OK
```

总结

若 T: Unpin (Rust 类型的默认实现)，那么 Pin<'a, T> 跟 &'a mut T 完全相同，也就是 Pin 将没有任何效果，该移动还是照常移动

绝大多数标准库类型都实现了 Unpin ，事实上，对于 Rust 中你能遇到的绝大多数类型，该结论依然成立，其中一个例外就是：async/await 生成的 Future 没有实现 Unpin

你可以通过以下方法为自己的类型添加!Unpin 约束：

使用文中提到的 std::marker::PhantomPinned

使用 nightly 版本下的 feature flag

可以将值固定到栈上，也可以固定到堆上

将!Unpin 值固定到栈上需要使用 unsafe

将!Unpin 值固定到堆上无需 unsafe ，可以通过 Box::pin 来简单的实现

当固定类型 T: !Unpin 时，你需要保证数据从被固定到被 drop 这段时期内，其内存不会变得非法或者被重用