[翻译]async: 什么是blocking

2021-05-09 | 2362 | 1 Comment

原文来自:https://ryhl.io/blog/async-what-is-blocking/

作者:Alice Ryhl

翻译 by abaabaqua

`Rust`的`async`/`await`特性是通过一种称为协作式调度(cooperative scheduling)的机制来实现的，这对于编写异步`Rust`代码的人来说有一些重要的影响。

这篇博文的目标读者是异步`Rust`的新用户。我将使用`Tokio`运行时作为示例，但这里提出的观点适用于任何异步运行时。

如果你只从这篇文章中记住一件事，那应该是:
异步代码不应该长时间不到达`.await`。(注:指的是运行中)

Blocking vs. non-blocking code

编写一个可以同时处理许多事情的应用程序的质朴的方法是为每个任务生成一个新线程。如果任务的数量很少，这就是一个完美的解决方案，但是随着任务的数量变得越来越多，你最终会因为数量过多的线程而遇到问题。这个问题在不同的编程语言中有不同的解决方案，但它们都归结为同一件事: 非常快速地切换每个线程上当前运行的任务，这样所有的任务都有机会运行。在`Rust`中，这种切换发生在当你`.await`一些事上。

在写异步rust 时，短语`阻塞线程(blocking the thread)`意味着`阻止运行时切换当前任务`。这可能是一个主要问题，这意味着同一运行时上的其他任务将停止运行，直到线程不再被阻塞。为了防止这种情况，我们应该编写能够快速切换的代码，也就是不要长时间不进行`.await`。

让我们来举个例子:

```
use std::time::Duration;
```

```rust
#[tokio::main]
async fn main() {
    println!("Hello World!");

    // No .await here!
    std::thread::sleep(Duration::from_secs(5));

    println!("Five seconds later...");
}
```

上面的代码看起来是正确的，运行它将看起来正常工作。但它有一个致命的缺陷: 它阻塞了线程。上述情况没有其他任务，所以看不出问题，但在真正的程序中就不一样了。为了说明这一点，考虑下面的例子:

```rust
use std::time::Duration;

async fn sleep_then_print(timer: i32) {
    println!("Start timer {}.", timer);

    // No .await here!
    std::thread::sleep(Duration::from_secs(1));

    println!("Timer {} done.", timer);
}

#[tokio::main]
async fn main() {
    // The join! macro lets you run multiple things concurrentl
    tokio::join!(
        sleep_then_print(1),
        sleep_then_print(2),
        sleep_then_print(3),
    );
}
```

```
Start timer 1.
Timer 1 done.
Start timer 2.
Timer 2 done.
```

```
Start timer 3.
Timer 3 done.
```

这个例子将花费三秒钟运行，timer将在没有任何并发的情况下一个接一个地运行。原因很简单：`Tokio`运行时无法将一个任务切换为另一个任务，因为这种切换只能发生在`.await`。因为`sleep_then_print`中没有`.await`，在运行时就不会发生切换。

然而，如果我们使用`Tokio`的sleep函数，它使用一个`.await`来睡眠，那么这个函数就会正常工作:

```
use tokio::time::Duration;

async fn sleep_then_print(timer: i32) {
    println!("Start timer {}.", timer);

    tokio::time::sleep(Duration::from_secs(1)).await;
//                                             ^ execution can b

    println!("Timer {} done.", timer);
}

#[tokio::main]
async fn main() {
    // The join! macro lets you run multiple things concurrentl
    tokio::join!(
        sleep_then_print(1),
        sleep_then_print(2),
        sleep_then_print(3),
    );
}
```

```
Start timer 1.
Start timer 2.
Start timer 3.
Timer 1 done.
Timer 2 done.
Timer 3 done.
```

这段代码只需一秒钟就可以运行，并且如预期地在同一时间正确地运行所有三个函数。

要知道，事情并不总是这么明显。通过使用tokio::join!，这三个任务都保证在同一个线程上运行，但如果你用tokio::spawn替换它，并使用一个多线程的运行时，你将能够同时运行多个阻塞任务直到用完线程。默认的`Tokio`运行时按照核心数生成线程，你的电脑通常会有8个`CPU`。这足以使你在本地测试时忽略这个问题，并在真实场景下运行代码时非常快地耗尽线程。

为了给出多少时间算的上过久的定义，一个好的经验法则是每个`.await`的时间间隔不超过10到100微秒。这取决于你正在编写的应用程序的类型。

如果我想要阻塞呢?

有时候我们只是想阻塞线程。这是完全正常的。有两个常见的原因:

1.昂贵的`CPU-Bound`(受限于CPU资源的)计算
2.Synchronous IO. 同步IO

在这两种情况下，我们都在处理一个会在一段时间内阻止任务`.await`的操作。为了解决这个问题，我们必须将阻塞操作移动到`Tokio`线程池外部的线程中去。关于这一点有三种形式可以使用:

1.使用tokio::task::spawn_blocking函数
2.使用rayon crate
3.通过std::thread::spawn创建一个专门的线程

让我们仔细浏览下每个解决方案，看看什么时候应该使用它。

spawn_blocking函数

`Tokio`运行时包含一个单独的线程池，专门用于运行阻塞函数，你可以使用spawn_blocking在其上运行任务。这个线程池的上限是大约500个线程，因此可以在这个线程池上进行大量阻塞操作。

由于线程池有如此多的线程，它最适合用于阻塞`I/O`(如与文件系统交互)或使用阻塞数据库库(如diesel)。

线程池不适合昂贵的`CPU-bound`计算，因为线程池的线程数量比计算机上的 CPU核心数量多得多。当线程数等于CPU核心数时，`CPU-bound`的计算运行效率最高。也就是说，如果只需要运行几个`CPU-bound`的计算，我不会责怪你用`spawn_blocking`运行它们，因为这样做非常简单。

```
#[tokio::main]
async fn main() {
    // This is running on Tokio. We may not block here.

    let blocking_task = tokio::task::spawn_blocking(|| {
        // This is running on a thread where blocking is fine.
        println!("Inside spawn_blocking");
    });

    // We can wait for the blocking task like this:
    // If the blocking task panics, the unwrap below will propa
    // panic.
    blocking_task.await.unwrap();
}
```

The rayon crate

rayon是一个著名的库，它提供了一个专门用于昂贵的`CPU-bound`计算的线程池，你可以将它与`Tokio`一起用于此目的。与`spawn_blocking`不同，`rayon`的线程池的最大线程数量很少，这就是为什么它适合进行昂贵计算的原因。

我们将计算大列表的和作为例子，但是请注意在实践中，除非数组非常非常大，否则只计算和是足够便宜的，可以直接在`Tokio`运行。

使用`rayon`的主要危险是，在等待`rayon`完成时，必须注意不要阻塞线程。要做到这一点，可以将rayon::spawn 和tokio::sync::oneshot结合起来，就像这样:

```
async fn parallel_sum(nums: Vec<i32>) -> i32 {
    let (send, recv) = tokio::sync::oneshot::channel();

    // Spawn a task on rayon.
    rayon::spawn(move || {
```

```
        // Perform an expensive computation.
        let mut sum = 0;
        for num in nums {
            sum += num;
        }

        // Send the result back to Tokio.
        let _ = send.send(sum);
    });

    // Wait for the rayon task.
    recv.await.expect("Panic in rayon::spawn")
}

#[tokio::main]
async fn main() {
    let nums = vec![1; 1024 * 1024];
    println!("{}", parallel_sum(nums).await);
}
```

上述例子使用了`rayon`的线程池来运行昂贵的操作。请注意，例子中对 `parallel_sum` 的每个调用只使用了`rayon`线程池中的一个线程。如果你的应用程序中有很多对`parallel_ sum`的调用，那么这是有意义的，但是也可以使用`rayon`的并行迭代器在几个线程上计算和:

```
use rayon::prelude::*;

// Spawn a task on rayon.
rayon::spawn(move || {
    // Compute the sum on multiple threads.
    let sum = nums.par_iter().sum();

    // Send the result back to Tokio.
    let _ = send.send(sum);
});
```

注意，当使用并行迭代器时，仍然需要`rayon::spawn`调用，因为并行迭代器是阻塞的。

生成一个专用线程

如果阻塞操作是需要永久运行下去的，那么应该在专用线程上运行它。例如，考虑一个管理数据库连接的线程，他需要使用通道来接收要执行的操作。因为这个线程在循环中监听该通道，所以它永远不会退出。

在上述两个线程池中的任何一个上运行这样的任务都会是一个问题，因为它实际上从池中永久地带走了一个线程。一旦这样做了几次，线程池中就没有更多的线程了，所有其他阻塞任务都无法执行。

当然，如果你不在意每次启动一个新线程的成本，也可以使用专用线程来运行较短生命周期的任务。

总结

如果你忘了，以下是你需要记住的主要事情:
异步代码不应该长时间不到达`.await`。

下面你将看到一张备忘单，上面列出了当你想要阻塞的时候可以使用的方法:

	CPU-bound computation	Synchronous IO	Running forever
`spawn_blocking`	次优	OK	No
`rayon`	OK	No	No
专用线程	OK	OK	OK

最后，我建议阅读`Tokio`教程中关于共享状态的章节。本章将解释如何在异步代码中正确使用 std::sync::Mutex，并更深入地说明为什么即使锁定`Mutex`是阻塞的，也可以这样做。(剧透一下: 如果你的阻塞时间很短，它真的算阻塞吗?)

我还强烈推荐`Tokio`博客中的一篇文章：Reducing tail latencies with automatic cooperative task yielding。

感谢Chris Krycho和snocl阅读本文草稿并提供有用的建议。所有的错误都是我自己的。

原文来自:https://ryhl.io/blog/async-what-is-blocking/

作者:Alice Ryhl

翻译 by abaabaqua

rust # 翻译