200行代码讲透Rust Futures

stevenbai.top/rust/futures_explained_in_200_lines_of_rust

rust

2020/4/9

原文地址 Futures Explained in 200 Lines of Rust

一 引言

这本书的目的是用一个例子驱动的方法来解释Rust中的Futures，探索为什么他们被设计成这样，以及他们如何工作。 我们还将介绍在编程中处理并发性时的一些替代方案。

理解这本书中描述的细节， 不需要使用Rust中的 futures或async/await。 这是为那些好奇的人准备的，他们想知道这一切是如何运作的。

这本书涵盖的内容

这本书将试图解释你可能想知道的一切，包括不同类型的执行器(executor)和运行时(runtime)的主题。 我们将在本书中实现一个非常简单的运行时，介绍一些概念，但这已经足够开始了。

Stjepan Glavina 发表了一系列关于异步运行时和执行器的优秀文章，如果谣言属实，那么在不久的将来他会发表更多的文章。

你应该做的是先读这本书，然后继续阅读 stejpang 的文章，了解更多关于运行时以及它们是如何工作的，特别是: 1. 构建自的block_on 2. 构建自己的executor

我将自己限制在一个200行的主示例(因此才有了这个标题)中，以限制范围并介绍一个可以轻松进一步研究的示例。

然而，有很多东西需要消化，这并不像我所说的那么简单，但是我们会一步一步来，所以喝杯茶，放松一下。

> 这本书是在开放的，并欢迎贡献。 你可以在这里找到这本书。 最后的例子，你可以克隆或复制可以在这里找到。 任何建议或改进可以归档为一个公关或在问题追踪的书。 一如既往，我们欢迎各种各样的反馈。

阅读练习和进一步阅读

在最后一章)中，我冒昧地提出了一些小练习，如果你想进一步探索的话。

这本书也是我在 Rust 中写的关于并发编程的第四本书。 如果你喜欢它，你可能也想看看其他的:

感谢

我想借此机会感谢 mio，tokio，async std，Futures，libc，crossbeam 背后的人们，他们支撑着这个异步生态系统，却很少在我眼中得到足够的赞扬。

特别感谢 jonhoo，他对我这本书的初稿给予了一些有价值的反馈。 他还没有读完最终的成品，但是我们应该向他表示感谢。

二 背景资料

在我们深入研究 Futures in Rust 的细节之前，让我们快速了解一下处理并发编程的各种方法，以及每种方法的优缺点。

同时当涉及到并发性时,我们也会解释为什么这么做，这将使我们更容易深入理解Futures.

> 为了好玩，我在大多数示例中添加了一小段可运行代码。 如果你像我一样，事情会变得更有趣，也许你会看到一些你从未见过的东西。

线程

现在，实现这一点的一个方法就是让操作系统为我们处理好一切。 我们只需为每个要完成的任务生成一个新的操作系统线程，并像通常那样编写代码。

我们用来处理并发性的运行时就是操作系统本身。

优点:

- 简单
- 易用
- 在任务之间切换相当快
- 不需要付出即可得到并行支持

缺点:

- 操作系统级线程的堆栈相当大。 如果同时有许多任务等待(就像在负载很重的 web 服务器中那样)，那么内存将很快耗尽
- 这涉及到很多系统调用。当任务数量很高时，这可能会非常昂贵
- 操作系统有很多事情需要处理。 它可能不会像你希望的那样快速地切换回线程
- 某些系统可能不支持线程

在 Rust 中使用操作系统线程看起来像这样:

```
use std::thread;

fn main() {
    println!("So we start the program here!");
    let t1 = thread::spawn(move || {
        thread::sleep(std::time::Duration::from_millis(200));
        println!("We create tasks which gets run when they're finished!");
    });

    let t2 = thread::spawn(move || {
        thread::sleep(std::time::Duration::from_millis(100));
        println!("We can even chain callbacks...");
        let t3 = thread::spawn(move || {
            thread::sleep(std::time::Duration::from_millis(50));
            println!("...like this!");
        });
        t3.join().unwrap();
    });
    println!("While our tasks are executing we can do other stuff here.");

    t1.join().unwrap();
    t2.join().unwrap();
}
```

Rust

操作系统线程肯定有一些相当大的优势。 这也是为什么所有这些讨论“异步”和并发性把线程摆在首位?

首先。 为了使计算机有效率，它们需要多任务处理。 一旦你开始深入研究(比如操作系统是如何工作的)，你就会发现并发无处不在。 这是我们做任何事情的基础。

其次，我们有网络。

Webservers 是关于I/O和处理小任务(请求)的。 当小任务的数量很大时，由于它们所需的内存和创建新线程所涉及的开销，就不适合今天的操作系统线程。

如果负载是可变的，那么问题就更大了，这意味着程序在任何时间点的当前任务数量都是不可预测的。 这就是为什么今天你会看到如此多的异步web框架和数据库驱动程序。

然而有大量的问题,标准的线程通常是正确的解决方案。 因此在使用异步库之前，请三思而后行。

现在，让我们来看看多任务处理的其他选项。 它们都有一个共同点，那就是它们实现了一种多任务处理的方式，即拥有一个“用户界面”运行时:

绿色线程(Green threads)

绿色线程使用与操作系统相同的机制，为每个任务创建一个线程，设置一个堆栈，保存 CPU 状态，并通过“上下文切换”从一个任务(线程)跳转到另一个任务(线程)。

我们将控制权交给调度程序(在这样的系统中，调度程序是运行时的核心部分)，然后调度程序继续运行不同的任务。

Rust曾经支持绿色线程，但他们它达到1.0之前被删除了， 执行状态存储在每个栈中，因此在这样的解决方案中不需要 `async` , `await` , `Futures` 或者 `Pin` 。

典型的流程是这样的: 1. 运行一些非阻塞代码 2. 对某些外部资源进行阻塞调用 3. 跳转到main”线程，该线程调度一个不同的线程来运行，并“跳转”到该栈中 4. 在新线程上运行一些非阻塞代码，直到新的阻塞调用或任务完成 5. “跳转”回到“main”线程 ，调度一个新线程，这个新线程的状态已经是 `Ready` ,然后跳转到该线程

这些“跳转”被称为上下文切换，当你阅读这篇文章的时候，你的操作系统每秒钟都会做很多次。

优点:

1. 栈大小可能需要增长,解决这个问题不容易,并且会有成本.[^go中的栈] [^go中的栈]: 栈拷贝,指针等问题
2. 它不是一个零成本抽象(这也是Rust早期有绿色线程,后来删除的原因之一)
3. 如果您想要支持许多不同的平台，就很难正确实现

一个绿色线程的例子可以是这样的: > 下面的例子是一个改编的例子，来自我之前写的一本200行Rust说清绿色线程的 gitbook。 如果你想知道发生了什么，你会发现书中详细地解释了一切。 下面的代码非常不安全，只是为了展示一个真实的例子。 这绝不是为了展示“最佳实践”。 这样我们就能达成共识了。

```
 #![feature(asm, naked_functions)]
use std::ptr;

const DEFAULT_STACK_SIZE: usize = 1024 * 1024 * 2;
const MAX_THREADS: usize = 4;
static mut RUNTIME: usize = 0;

pub struct Runtime {
    threads: Vec<Thread>,
    current: usize,
}

  #[derive(PartialEq, Eq, Debug)]
enum State {
    Available,
    Running,
    Ready,
}

struct Thread {
    id: usize,
    stack: Vec<u8>,
    ctx: ThreadContext,
    state: State,
    task: Option<Box<dyn Fn()>>,
}

 #[derive(Debug, Default)]
 #[repr(C)]
struct ThreadContext {
    rsp: u64,
    r15: u64,
    r14: u64,
    r13: u64,
    r12: u64,
    rbx: u64,
    rbp: u64,
    thread_ptr: u64,
}

impl Thread {
    fn new(id: usize) -> Self {
        Thread {
            id,
            stack: vec![0_u8; DEFAULT_STACK_SIZE],
            ctx: ThreadContext::default(),
            state: State::Available,
            task: None,
        }
    }
}

impl Runtime {
    pub fn new() -> Self {
        let base_thread = Thread {
```

```
            id: 0,
            stack: vec![0_u8; DEFAULT_STACK_SIZE],
            ctx: ThreadContext::default(),
            state: State::Running,
            task: None,
        };

        let mut threads = vec![base_thread];
        threads[0].ctx.thread_ptr = &threads[0] as *const Thread as u64;
        let mut available_threads: Vec<Thread> = (1..MAX_THREADS).map(|i|
Thread::new(i)).collect();
        threads.append(&mut available_threads);

        Runtime {
            threads,
            current: 0,
        }
    }

    pub fn init(&self) {
        unsafe {
            let r_ptr: *const Runtime = self;
            RUNTIME = r_ptr as usize;
        }
    }

    pub fn run(&mut self) -> ! {
        while self.t_yield() {}
        std::process::exit(0);
    }

    fn t_return(&mut self) {
        if self.current != 0 {
            self.threads[self.current].state = State::Available;
            self.t_yield();
        }
    }

    fn t_yield(&mut self) -> bool {
        let mut pos = self.current;
        while self.threads[pos].state != State::Ready {
            pos += 1;
            if pos == self.threads.len() {
                pos = 0;
            }
            if pos == self.current {
                return false;
            }
        }

        if self.threads[self.current].state != State::Available {
            self.threads[self.current].state = State::Ready;
        }

        self.threads[pos].state = State::Running;
```

```
            let old_pos = self.current;
            self.current = pos;

            unsafe {
                switch(&mut self.threads[old_pos].ctx, &self.threads[pos].ctx);
            }
            true
        }

        pub fn spawn<F: Fn() + 'static>(f: F){
            unsafe {
                let rt_ptr = RUNTIME as *mut Runtime;
                let available = (*rt_ptr)
                    .threads
                    .iter_mut()
                    .find(|t| t.state == State::Available)
                    .expect("no available thread.");

                let size = available.stack.len();
                let s_ptr = available.stack.as_mut_ptr();
                available.task = Some(Box::new(f));
                available.ctx.thread_ptr = available as *const Thread as u64;
                ptr::write(s_ptr.offset((size - 8) as isize) as *mut u64, guard as
    u64);
                ptr::write(s_ptr.offset((size - 16) as isize) as *mut u64, call as
    u64);
                available.ctx.rsp = s_ptr.offset((size - 16) as isize) as u64;
                available.state = State::Ready;
            }
        }
    }

    fn call(thread: u64) {
        let thread = unsafe { &*(thread as *const Thread) };
        if let Some(f) = &thread.task {
            f();
        }
    }

     #[naked]
    fn guard() {
        unsafe {
            let rt_ptr = RUNTIME as *mut Runtime;
            let rt = &mut *rt_ptr;
            println!("THREAD {} FINISHED.", rt.threads[rt.current].id);
            rt.t_return();
        };
    }

    pub fn yield_thread() {
        unsafe {
            let rt_ptr = RUNTIME as *mut Runtime;
            (*rt_ptr).t_yield();
        };
    }
```

```
 #[naked]
 #[inline(never)]
unsafe fn switch(old: *mut ThreadContext, new: *const ThreadContext) {
    asm!("
        mov     %rsp, 0x00($0)
        mov     %r15, 0x08($0)
        mov     %r14, 0x10($0)
        mov     %r13, 0x18($0)
        mov     %r12, 0x20($0)
        mov     %rbx, 0x28($0)
        mov     %rbp, 0x30($0)

        mov     0x00($1), %rsp
        mov     0x08($1), %r15
        mov     0x10($1), %r14
        mov     0x18($1), %r13
        mov     0x20($1), %r12
        mov     0x28($1), %rbx
        mov     0x30($1), %rbp
        mov     0x38($1), %rdi
        ret
        "
    :
    : "r"(old), "r"(new)
    :
    : "alignstack"
    );
}
 #[cfg(not(windows))]
fn main() {
    let mut runtime = Runtime::new();
    runtime.init();
    Runtime::spawn(|| {
        println!("I haven't implemented a timer in this example.");
        yield_thread();
        println!("Finally, notice how the tasks are executed concurrently.");
    });
    Runtime::spawn(|| {
        println!("But we can still nest tasks...");
        Runtime::spawn(|| {
            println!("...like this!");
        })
    });
    runtime.run();
}
 #[cfg(windows)]
fn main() {
    let mut runtime = Runtime::new();
    runtime.init();
    Runtime::spawn(|| {
        println!("I haven't implemented a timer in this example.");
        yield_thread();
        println!("Finally, notice how the tasks are executed concurrently.");
    });
```

```
    Runtime::spawn(|| {
        println!("But we can still nest tasks...");
        Runtime::spawn(|| {
            println!("...like this!");
        })
    });
    runtime.run();
}
```

Rust

还在坚持阅读本书？ 很好。 如果上面的代码很难理解，不要感到沮丧。 如果不是我自己写的，我可能也会有同样的感觉。 你随时可以回去读,稍后我还会解释。

基于回调的方法

你可能已经知道我们接下来要谈论Javascript，我想大多数人都知道。

> 如果你接触过 Javascript 回调会让你更早患上 PTSD，那么现在闭上眼睛，向下滚动 2-3秒。 你会在那里找到一个链接，带你到安全的地方。

基于回调的方法背后的整个思想就是保存一个指向一组指令的指针，这些指令我们希望以后在以后需要的时候运行。 针对Rust，这将是一个闭包。 在下面的示例中，我们将此信息保存在 `HashMap` 中，但这并不是唯一的选项。

不涉及线程作为实现并发性的主要方法的基本思想是其余方法的共同点。 包括我们很快就会讲到的 Rust 今天使用的那个。

优点: 1. 大多数语言中易于实现 2. 没有上下文切换 3. 相对较低的内存开销(在大多数情况下)

缺点:

1. 每个任务都必须保存它以后需要的状态，内存使用量将随着一系列计算中回调的数量线性增长
2. 很难理解，很多人已经知道这就是“回调地狱”
3. 这是一种非常不同的编写程序的方式，需要大量的重写才能从“正常”的程序流转变为使用“基于回调”的程序流
4. 在 Rust 使用这种方法时，任务之间的状态共享是一个难题，因为它的所有权模型

一个极其简单的基于回调方法的例子是:

```
fn program_main() {
    println!("So we start the program here!");
    set_timeout(200, || {
        println!("We create tasks with a callback that runs once the task
finished!");
    });
    set_timeout(100, || {
        println!("We can even chain sub-tasks...");
        set_timeout(50, || {
            println!("...like this!");
        })
    });
    println!("While our tasks are executing we can do other stuff instead of
waiting.");
}

fn main() {
    RT.with(|rt| rt.run(program_main));
}

use std::sync::mpsc::{channel, Receiver, Sender};
use std::{cell::RefCell, collections::HashMap, thread};

thread_local! {
    static RT: Runtime = Runtime::new();
}

struct Runtime {
    callbacks: RefCell<HashMap<usize, Box<dyn FnOnce() -> ()>>>,
    next_id: RefCell<usize>,
    evt_sender: Sender<usize>,
    evt_reciever: Receiver<usize>,
}

fn set_timeout(ms: u64, cb: impl FnOnce() + 'static) {
    RT.with(|rt| {
        let id = *rt.next_id.borrow();
        *rt.next_id.borrow_mut() += 1;
        rt.callbacks.borrow_mut().insert(id, Box::new(cb));
        let evt_sender = rt.evt_sender.clone();
        thread::spawn(move || {
            thread::sleep(std::time::Duration::from_millis(ms));
            evt_sender.send(id).unwrap();
        });
    });
}

impl Runtime {
    fn new() -> Self {
        let (evt_sender, evt_reciever) = channel();
        Runtime {
            callbacks: RefCell::new(HashMap::new()),
            next_id: RefCell::new(1),
            evt_sender,
            evt_reciever,
```

```
        }
    }

    fn run(&self, program: fn()) {
        program();
        for evt_id in &self.evt_reciever {
            let cb = self.callbacks.borrow_mut().remove(&evt_id).unwrap();
            cb();
            if self.callbacks.borrow().is_empty() {
                break;
            }
        }
    }
}
```

Rust

我们保持这种超级简单的方法，您可能想知道这种方法和使用 OS 线程直接将回调传递给 OS 线程的方法之间有什么区别。 不同之处在于，回调是在同一个线程上运行的。 这个例子中,我们创建的 OS 线程基本上只是用作计时器，但可以表示任何类型的我们将不得不等待的资源。

从回调到承诺 (promises)

你现在可能会想，我们什么时候才能谈论未来?

好吧，我们就快到了。你看， `promises` 、 `futures` 和其他延迟计算的名称经常被交替使用。

它们之间有形式上的区别，但是我们在这里不会涉及，但是值得解释一下 `promises` ，因为它们被广泛使用在 Javascript 中，并且与 Rusts Futures 有很多共同之处。

首先，许多语言都有 `promises` 的概念，但我将在下面的例子中使用来自 Javascript 的概念。

承诺是解决回调带来的复杂性的一种方法。

比如,下面的例子:

```
setTimer(200, () => {
  setTimer(100, () => {
    setTimer(50, () => {
      console.log("I'm the last one");
    });
  });
});
```

JavaScript

可以替换为promise:

```
function timer(ms) {
    return new Promise((resolve) => setTimeout(resolve, ms))
}

timer(200)
.then(() => return timer(100))
.then(() => return timer(50))
.then(() => console.log("I'm the last one));
```

JavaScript

深入原理可以看到变化更为显著。 您可以看到，promises 返回的状态机可以处于以下三种状态之一: `pending` 、 `fulfilled` 或 `rejected` 。

当我们在上面的例子中调用 `timer (200)` 时，我们得到一个状态 `pending` 的承诺。

由于承诺被重写为状态机，它们还提供了一种更好的语法，允许我们像下面这样编写最后一个示例:

```
async function run() {
    await timer(200);
    await timer(100);
    await timer(50);
    console.log("I'm the last one");
}
```

JavaScript

可以将 run 函数视为一个由几个子任务组成的可执行任务。 在每个 `await` 点上，它都将控制权交给调度程序(在本例中是众所周知的 Javascript 事件循环)。

一旦其中一个子任务将状态更改为 `fulfilled` 或 `rejected` ，则计划继续执行下一步。

从语法上讲，Rusts Futures 0.1很像上面的承诺示例，Rusts Futures 0.3很像我们上一个示例中的 async / await。

这也是与 Rusts Futures 相似的地方。 我们这样做的原因是通过上面的介绍,更加深刻的理解Rust的Futures。

> 为了避免以后的混淆: 有一点你应该知道。 Java script的承诺是立即执行(early evaluated)的。 这意味着一旦它被创建，它就开始运行一个任务。 与此相反,Rust的Futures是延迟执行(lazy evaluated)。 除非轮询(poll)一次,否则什么事都不会发生。

三 Rust中的Futures

概述

1. Rust中并发性的高级介绍
2. 了解 Rust 在使用异步代码时能提供什么，不能提供什么

3. 了解为什么我们需要 Rust 的运行时库
4. 理解“leaf-future”和“non-leaf-future”的区别
5. 了解如何处理 CPU 密集型任务

Futures

什么是 `Future`？ `Future` 是一些将在未来完成的操作。 Rust中的异步实现基于轮询,每个异步任务分成三个阶段: 1. 轮询阶段(The Poll phase). 一个 `Future` 被轮询后,会开始执行,直到被阻塞. 我们经常把轮询一个Future这部分称之为执行器(executor) 2. 等待阶段. 事件源(通常称为reactor)注册等待一个事件发生，并确保当该事件准备好时唤醒相应的 `Future` 3. 唤醒阶段. 事件发生,相应的 `Future` 被唤醒。 现在轮到执行器(executor),就是第一步中的那个执行器，调度 `Future` 再次被轮询，并向前走一步，直到它完成或达到一个阻塞点，不能再向前走, 如此往复,直到最终完成.

当我们谈论 `Future` 的时候，我发现在早期区分 `non-leaf-future` 和 `leaf-future` 是很有用的，因为实际上它们彼此很不一样。

Leaf futures

由运行时创建 `leaf futures` ，它就像套接字一样,代表着一种资源.

```
// stream is a **leaf-future**
let mut stream = tokio::net::TcpStream::connect("127.0.0.1:3000");
```

Rust

对这些资源的操作，比如套接字上的 Read 操作，将是非阻塞的，并返回一个我们称之为 `leaf-future` 的Future.之所以称之为 `leaf-future` ,是因为这是我们实际上正在等待的 Future.

除非你正在编写一个运行时，否则你不太可能自己实现一个 `leaf-future` ，但是我们将在本书中详细介绍它们是如何构造的。

您也不太可能将 `leaf-future` 传递给运行时，然后单独运行它直到完成，这一点您可以通过阅读下一段来理解。

Non-leaf-futures

Non-leaf-futures指的是那些我们用 `async` 关键字创建的Future.

异步程序的大部分是Non-leaf-futures，这是一种可暂停的计算。 这是一个重要的区别，因为这些 `Future` 代表一组操作。 通常，这样的任务由 `await` 一系列 `leaf-future` 组成.

```
// Non-leaf-future
let non_leaf = async {
    let mut stream = TcpStream::connect("127.0.0.1:3000").await.unwrap();// <-
yield
    println!("connected!");
    let result = stream.write(b"hello world\n").await; // <- yield
    println!("message sent!");
    ...
};
```

Rust

这些任务的关键是，它们能够将控制权交给运行时的调度程序，然后在稍后停止的地方继续执行。 与 `leaf-future` 相比，这些Future本身并不代表I/O资源。 当我们对这些Future进行轮询时, 有可能会运行一段时间或者因为等待相关资源而让度给调度器,然后等待相关资源ready的时候唤醒自己.

运行时(Runtimes)

像 c # ，JavaScript，Java，GO 和许多其他语言都有一个处理并发的运行时。 所以如果你来自这些语言中的一种，这对你来说可能会有点奇怪。

Rust 与这些语言的不同之处在于 Rust 没有处理并发性的运行时，因此您需要使用一个为您提供此功能的库。

很多复杂性归因于 Futures 实际上是来源于运行时的复杂性，创建一个有效的运行时是困难的。 学习如何正确使用一个也需要相当多的努力，但是你会看到这些类型的运行时之间有几个相似之处，所以学习一个可以使学习下一个更容易。

Rust 和其他语言的区别在于，在选择运行时时，您必须进行主动选择。大多数情况下，在其他语言中，你只会使用提供给你的那一种。

异步运行时可以分为两部分: 1. 执行器(The Executor) 2. reactor (The Reactor)

当 Rusts Futures 被设计出来的时候，有一个愿望，那就是将通知 `Future` 它可以做更多工作的工作与 `Future` 实际做工作分开。

你可以认为前者是reactor的工作，后者是执行器的工作。 运行时的这两个部分使用 `Waker` 进行交互。

写这篇文章的时候，未来最受欢迎的两个运行时是:

1. async-std
2. Tokio

Rust 的标准库做了什么

1. 一个公共接口， `Future trait`

2. 一个符合人体工程学的方法创建任务, 可以通过async和await关键字进行暂停和恢复 `Future`
3. `Waker` 接口, 可以唤醒暂停的 `Future`

这就是Rust标准库所做的。 正如你所看到的，不包括异步I/O的定义,这些异步任务是如何被创建的,如何运行的。

I/O密集型 VS CPU密集型任务

正如你们现在所知道的，你们通常所写的是 `Non-leaf-futures` 。 让我们以 pseudo-rust 为例来看一下这个异步块:

```
let non_leaf = async {
    let mut stream = TcpStream::connect("127.0.0.1:3000").await.unwrap(); // <--
yield

    // request a large dataset
    let result = stream.write(get_dataset_request).await.unwrap(); // <-- yield

    // wait for the dataset
    let mut response = vec![];
    stream.read(&mut response).await.unwrap(); // <-- yield

    // do some CPU-intensive analysis on the dataset
    let report = analyzer::analyze_data(response).unwrap();

    // send the results back
    stream.write(report).await.unwrap(); // <-- yield
};
```

Rust

现在，正如您将看到的，当我们介绍 Futures 的工作原理时，两个 `yield` 之间的代码与我们的执行器在同一个线程上运行。

这意味着当我们分析器处理数据集时，执行器忙于计算而不是处理新的请求。

幸运的是，有几种方法可以解决这个问题，这并不困难，但是你必须意识到: 1. 我们可以创建一个新的 `leaf future` ，它将我们的任务发送到另一个线程，并在任务完成时解析。我们可以像等待其他Future一样等待这个 `leaf-future` 。 2. 运行时可以有某种类型的管理程序来监视不同的任务占用多少时间，并将执行器本身移动到不同的线程，这样即使我们的分析程序任务阻塞了原始的执行程序线程，它也可以继续运行。 3. 您可以自己创建一个与运行时兼容的 `reactor` ，以您认为合适的任何方式进行分析，并返回一个可以等待的未来。

现在，#1是通常的处理方式，但是一些执行器也实现了#2。 2的问题是，如果你切换运行时，你需要确保它也支持这种监督，否则你最终会阻塞执行者。

方式#3更多的是理论上的重要性，通常您会很乐意将任务发送到多数运行时提供的线程池。

大多数执行器都可以使用诸如 spawn blocking 之类的方法来完成#1。

这些方法将任务发送到运行时创建的线程池，在该线程池中，您可以执行 cpu 密集型任务，也可以执行运行时不支持的“阻塞”任务。

现在，有了这些知识，你已经在一个很好的方式来理解 `Future` ，但我们不会停止，有很多细节需要讨论。

休息一下或喝杯咖啡，准备好我们进入下一章的深度探索。

奖励部分

如果你发现并发和异步编程的概念一般来说令人困惑，我知道你是从哪里来的，我已经写了一些资源，试图给出一个高层次的概述，这将使之后更容易学习 Rusts Futures:

- Async Basics - The difference between concurrency and parallelism 异步基础-并发和并行之间的区别
- Async Basics - Async history 异步基础-异步历史
- Async Basics - Strategies for handling I/O 异步基础-处理 i / o 的策略
- Async Basics - Epoll, Kqueue and IOCP 异步基础-Epoll，Kqueue 和 IOCP

通过研究 `Future` 来学习这些概念会让它变得比实际需要难得多，所以如果你有点不确定的话，继续读这些章节。

你回来的时候我就在这儿。

如果你觉得你已经掌握了基本知识，那么让我们开始行动吧！

四 唤醒器和上下文(Waker and Context)

概述

1. 了解 Waker 对象是如何构造的
2. 了解运行时如何知道 `leaf-future` 何时可以恢复
3. 了解动态分发的基础知识和trait对象

`Waker` 类型在RFC#2592中介绍.

唤醒器

`Waker` 类型允许在运行时的reactor 部分和执行器部分之间进行松散耦合。

通过使用不与 `Future` 执行绑定的唤醒机制，运行时实现者可以提出有趣的新唤醒机制。例如，可以生成一个线程来执行一些工作，这些工作结束时通知 `Future` ，这完全独立于当前的运行时。

如果没有唤醒程序，执行程序将是通知正在运行的任务的唯一方式，而使用唤醒程序，我们将得到一个松散耦合，其中很容易使用新的 `leaf-future` 来扩展生态系统。

> 如果你想了解更多关于 Waker 类型背后的原因，我可以推荐Withoutboats articles series about them。

理解唤醒器

在实现我们自己的 `Future` 时，我们遇到的最令人困惑的事情之一就是我们如何实现一个唤醒器。 创建一个 Waker 需要创建一个 vtable，这个vtable允许我们使用动态方式调用我们真实的Waker实现.

> 如果你想知道更多关于Rust中的动态分发，我可以推荐 Adam Schwalm 写的一篇文章 Exploring Dynamic Dispatch in Rust.

让我们更详细地解释一下。

Rust中的胖指针

为了更好地理解我们如何在 Rust 中实现 Waker，我们需要退后一步并讨论一些基本原理。 让我们首先看看 Rust 中一些不同指针类型的大小。

运行以下代码:

```
trait SomeTrait { }

fn main() {
    println!("======== The size of different pointers in Rust: ========");
    println!("&dyn Trait:-----{}", size_of::<&dyn SomeTrait>());
    println!("&[&dyn Trait]:--{}", size_of::<&[&dyn SomeTrait]>());
    println!("Box<Trait>:-----{}", size_of::<Box<SomeTrait>>());
    println!("&i32:-----------{}", size_of::<&i32>());
    println!("&[i32]:---------{}", size_of::<&[i32]>());
    println!("Box<i32>:-------{}", size_of::<Box<i32>>());
    println!("&Box<i32>:------{}", size_of::<&Box<i32>>());
    println!("[&dyn Trait;4]:-{}", size_of::<[&dyn SomeTrait; 4]>());
    println!("[i32;4]:--------{}", size_of::<[i32; 4]>());
}
```

Rust

从运行后的输出中可以看到，引用的大小是不同的。 许多是8字节(在64位系统中是指针大小)，但有些是16字节。

16字节大小的指针被称为“胖指针”，因为它们携带额外的信息。

例如 `&[i32]` :- 前8个字节是指向数组中第一个元素的实际指针(或 slice 引用的数组的一部分) - 第二个8字节是切片的长度

例如 `&dyn SomeTrait` :

这就是我们将要关注的胖指针的类型。 `&dyn SomeTrait` 是一个trait的引用，或者 Rust 称之为一个trait对象。

指向 trait 对象的指针布局如下: - 前8个字节指向trait 对象的data - 后八个字节指向trait对象的 vtable

这样做的好处是，我们可以引用一个对象，除了它实现了 trait 定义的方法之外，我们对这个对象一无所知。 为了达到这个目的，我们使用动态分发。

让我们用代码而不是文字来解释这一点，通过这些部分来实现我们自己的 trait 对象:

```
// A reference to a trait object is a fat pointer: (data_ptr, vtable_ptr)
trait Test {
    fn add(&self) -> i32;
    fn sub(&self) -> i32;
    fn mul(&self) -> i32;
}

// This will represent our home brewn fat pointer to a trait object
   #[repr(C)]
struct FatPointer<'a> {
    /// A reference is a pointer to an instantiated `Data` instance
    data: &'a mut Data,
    /// Since we need to pass in literal values like length and alignment it's
    /// easiest for us to convert pointers to usize-integers instead of the other
way around.
    vtable: *const usize,
}

// This is the data in our trait object. It's just two numbers we want to operate
on.
struct Data {
    a: i32,
    b: i32,
}

// ====== function definitions ======
fn add(s: &Data) -> i32 {
    s.a + s.b
}
fn sub(s: &Data) -> i32 {
    s.a - s.b
}
fn mul(s: &Data) -> i32 {
    s.a * s.b
}

fn main() {
    let mut data = Data {a: 3, b: 2};
    // vtable is like special purpose array of pointer-length types with a fixed
    // format where the three first values has a special meaning like the
    // length of the array is encoded in the array itself as the second value.
    let vtable = vec![
        0,            // pointer to `Drop` (which we're not implementing here)
        6,            // lenght of vtable
        8,            // alignment

        // we need to make sure we add these in the same order as defined in the
Trait.
        add as usize, // function pointer - try changing the order of `add`
        sub as usize, // function pointer - and `sub` to see what happens
        mul as usize, // function pointer
    ];

    let fat_pointer = FatPointer { data: &mut data, vtable: vtable.as_ptr()};
    let test = unsafe { std::mem::transmute::<FatPointer, &dyn Test>(fat_pointer)
```

```
};

    // And voalá, it's now a trait object we can call methods on
    println!("Add: 3 + 2 = {}", test.add());
    println!("Sub: 3 - 2 = {}", test.sub());
    println!("Mul: 3 * 2 = {}", test.mul());
}
```

Rust

稍后，当我们实现我们自己的 Waker 时，我们实际上会像这里一样建立一个 vtable。 我们创造它的方式略有不同，但是现在你知道了规则特征对象是如何工作的，你可能会认识到我们在做什么，这使得它不那么神秘。

奖励部分

您可能想知道为什么Waker是这样实现的，而不仅仅是作为一个普通的trait.

原因在于灵活性。 以这里的方式实现 Waker，可以很灵活地选择要使用的内存管理方案。

“正常”的方法是使用 Arc 来使用引用计数来跟踪 Waker 对象何时可以被删除。 但是，这不是唯一的方法，您还可以使用纯粹的全局函数和状态，或者任何其他您希望的方法。

这在表中为运行时实现者留下了许多选项。

五 生成器和async/await

概述

1. 理解 async / await 语法在底层是如何工作的
2. 亲眼目睹(See first hand)我们为什么需要Pin
3. 理解是什么让 Rusts 异步模型的内存效率非常高

生成器的动机可以在 RFC#2033中找到。 它写得非常好，我建议您通读它(它谈论async/await的内容和谈论生成器的内容一样多)。

为什么要学习生成器

generators/yield和 async/await 非常相似，一旦理解了其中一个，就应该能够理解另一个。

对我来说，使用Generators而不是 Futures 来提供可运行的和简短的示例要容易得多，这需要我们现在引入很多概念，稍后我们将介绍这些概念，以便展示示例。

Async/await 的工作方式类似于生成器，但它不返回生成器，而是返回一个实现 Future trait 的特殊对象。

一个小小的好处是，在本章的最后，你将有一个很好的关于生成器和 async / await 的介绍。

基本上，在设计 Rust 如何处理并发时，主要讨论了三个选项: 1. Green Thread. 2. 使用组合符(Using combinators.) 3. Generator, 没有专门的栈

我们在背景信息中覆盖了绿色线程，所以我们不会在这里重复。 我们将集中在各种各样的无堆栈协同程序,这也就是Rust正在使用的.

组合子(Combinators)

在 `Futures 0.1` 中使用组合子.如果你曾经是用过Javascript中的 `Promises` ,那么你已经比较熟悉combinators了. 在Rust中,他们看起来如下:

```
let future = Connection::connect(conn_str).and_then(|conn| {
    conn.query("somerequest").map(|row|{
        SomeStruct::from(row)
    }).collect::<Vec<SomeStruct>>()
});

let rows: Result<Vec<SomeStruct>, SomeLibraryError> = block_on(future);
```

Rust

使用这个技巧主要有三个缺点: 1. 错误消息可能会冗长并且难懂 2. 不是最佳的内存使用(浪费内存) 3. Rust中不允许跨组合子借用.

其中第三点是这种方式的主要缺点.

不允许跨组合子借用，结果是非常不符合人体工程学的.为了完成某些任务，需要额外的内存分配或者复制，这很低效。

内存占用高的原因是，这基本上是一种基于回调的方法，其中每个闭包存储计算所需的所有数据。 这意味着，随着我们将它们链接起来，存储所需状态所需的内存会随着每一步的增加而增加。

无栈协程/生成器

这就是今天 Rust 使用的模型，它有几个显著的优点: 1. 使用 async/await 作为关键字，可以很容易地将普通的Rust代码转换为无堆栈的协程(甚至可以使用宏来完成) 2. 不需要上下文切换与保存恢复CPU状态 3. 不需要处理的动态栈分配 4. 内存效率高 5. 允许我们块暂停点(suspension)借用 这是啥意思啊

与Futures 0.1不一样，使用async/ await 我们可以这样做:

```
async fn myfn() {
    let text = String::from("Hello world");
    let borrowed = &text[0..5];
    somefuture.await;
    println!("{}", borrowed);
}
```

Rust

Rust中的异步使用生成器实现. 因此为了理解异步是如何工作的，我们首先需要理解生成器。 在Rust中，生成器被实现为状态机。

一个计算链的内存占用是由占用空间最大的那个步骤定义的。

这意味着在计算链中添加步骤可能根本不需要增加任何内存，这也是为什么Futures和Async 在 Rust 中的开销很小的原因之一。

生成器是如何工作的

在今天的 Nightly Rust 中，你可以使用关键词 yield。 在闭包中使用这个关键字，将其转换为生成器。 在介绍Pin之前，闭包是这样的:

```
 #![feature(generators, generator_trait)]
use std::ops::{Generator, GeneratorState};

fn main() {
    let a: i32 = 4;
    let mut gen = move || {
        println!("Hello");
        yield a * 2;
        println!("world!");
    };

    if let GeneratorState::Yielded(n) = gen.resume() {
        println!("Got value {}", n);
    }

    if let GeneratorState::Complete(()) = gen.resume() {
        ()
    };
}
```

Rust

早些时候，在人们对 Pin 的设计达成共识之前，编译完代码看起来类似于这样:

```
fn main() {
    let mut gen = GeneratorA::start(4);

    if let GeneratorState::Yielded(n) = gen.resume() {
        println!("Got value {}", n);
    }

    if let GeneratorState::Complete(()) = gen.resume() {
        ()
    };
}

// If you've ever wondered why the parameters are called Y and R the naming from
// the original rfc most likely holds the answer
enum GeneratorState<Y, R> {
    Yielded(Y),  // originally called `Yield(Y)`
    Complete(R), // originally called `Return(R)`
}

trait Generator {
    type Yield;
    type Return;
    fn resume(&mut self) -> GeneratorState<Self::Yield, Self::Return>;
}

enum GeneratorA {
    Enter(i32),
    Yield1(i32),
    Exit,
}

impl GeneratorA {
    fn start(a1: i32) -> Self {
        GeneratorA::Enter(a1)
    }
}

impl Generator for GeneratorA {
    type Yield = i32;
    type Return = ();
    fn resume(&mut self) -> GeneratorState<Self::Yield, Self::Return> {
        // lets us get ownership over current state
        match std::mem::replace(self, GeneratorA::Exit) {
            GeneratorA::Enter(a1) => {

          /*----code before yield----*/
                println!("Hello");
                let a = a1 * 2;

                *self = GeneratorA::Yield1(a);
                GeneratorState::Yielded(a)
            }

            GeneratorA::Yield1(_) => {
          /*-----code after yield-----*/
```

```
                println!("world!");

                *self = GeneratorA::Exit;
                GeneratorState::Complete(())
            }
            GeneratorA::Exit => panic!("Can't advance an exited generator!"),
        }
    }
}
```

Rust

关键词yield首先在RFC#1823和 RFC#1832中讨论。

既然您知道了现实中的 yield 关键字会将代码重写为状态机，那么您还将了解await 如何工作的,他们非常相似.

上述简单的状态机中有一些限制,当跨yield发生借用的时候会发生什么呢?

我们可以禁止这样做，但async/await 语法的主要设计目标之一就是允许这样做。 这些类型的借用是不可能使用 `Futures 0.1` ，所以我们不能让这个限制存在。

与其在理论上讨论它，不如让我们来看看一些代码。

> 我们将使用目前 Rust 中使用的状态机的优化版本。 更深入的解释见 Tyler Mandry 的文章: Rust 如何优化async/await

```
let mut generator = move || {
        let to_borrow = String::from("Hello");
        let borrowed = &to_borrow;
        yield borrowed.len();
        println!("{} world!", borrowed);
    };
```

Rust

我们将手工编写一些版本的状态机，这些状态机表示生成器定义的状态机。

在每个示例中，我们都是“手动”逐步完成每个步骤，因此它看起来非常陌生。 我们可以添加一些语法糖，比如为我们的生成器实现 Iterator trait，这样我们就可以这样做:

```
while let Some(val) = generator.next() {
    println!("{}", val);
}
```

Rust

这是一个相当微不足道的改变，但是这一章已经变得很长了。我们继续前进的时候，请牢牢记住这点。

现在，我们的重写状态机在这个示例中看起来是什么样子的?

```
 #![allow(unused_variables)]
fn main() {
enum GeneratorState<Y, R> {
    Yielded(Y),
    Complete(R),
}

trait Generator {
    type Yield;
    type Return;
    fn resume(&mut self) -> GeneratorState<Self::Yield, Self::Return>;
}

enum GeneratorA {
    Enter,
    Yield1 {
        to_borrow: String,
        borrowed: &String, // uh, what lifetime should this have?
    },
    Exit,
}

impl GeneratorA {
    fn start() -> Self {
        GeneratorA::Enter
    }
}

impl Generator for GeneratorA {
    type Yield = usize;
    type Return = ();
    fn resume(&mut self) -> GeneratorState<Self::Yield, Self::Return> {
        // lets us get ownership over current state
        match std::mem::replace(self, GeneratorA::Exit) {
            GeneratorA::Enter => {
                let to_borrow = String::from("Hello");
                let borrowed = &to_borrow; // <--- NB!
                let res = borrowed.len();

                *self = GeneratorA::Yield1 {to_borrow, borrowed};
                GeneratorState::Yielded(res)
            }

            GeneratorA::Yield1 {to_borrow, borrowed} => {
                println!("Hello {}", borrowed);
                *self = GeneratorA::Exit;
                GeneratorState::Complete(())
            }
            GeneratorA::Exit => panic!("Can't advance an exited generator!"),
        }
    }
}
}
```

Rust

如果你试图编译这个，你会得到一个错误。

字符串的生命周期是什么。 这和Self的生命周期是不一样的。 它不是静态的。 事实证明，我们不可能用Rusts语法来描述这个生命周期，这意味着，为了使这个工作成功，我们必须让编译器知道，我们自己正确地控制了它。

这意味着必须借助unsafe。

让我们尝试编写一个使用unsafe的实现。 正如您将看到的，我们最终将使用一个自引用结构, 也就是将引用保存在自身中的结构体。

正如您所注意到的，这个编译器编译得很好！

```
 #![allow(unused_variables)]
fn main() {
enum GeneratorState<Y, R> {
    Yielded(Y),
    Complete(R),
}

trait Generator {
    type Yield;
    type Return;
    fn resume(&mut self) -> GeneratorState<Self::Yield, Self::Return>;
}

enum GeneratorA {
    Enter,
    Yield1 {
        to_borrow: String,
        borrowed: *const String, // NB! This is now a raw pointer!
    },
    Exit,
}

impl GeneratorA {
    fn start() -> Self {
        GeneratorA::Enter
    }
}
impl Generator for GeneratorA {
    type Yield = usize;
    type Return = ();
    fn resume(&mut self) -> GeneratorState<Self::Yield, Self::Return> {
            match self {
            GeneratorA::Enter => {
                let to_borrow = String::from("Hello");
                let borrowed = &to_borrow;
                let res = borrowed.len();
                *self = GeneratorA::Yield1 {to_borrow, borrowed:
std::ptr::null()};

                // NB! And we set the pointer to reference the to_borrow string
here
                if let GeneratorA::Yield1 {to_borrow, borrowed} = self {
                    *borrowed = to_borrow;
                }

                GeneratorState::Yielded(res)
            }

            GeneratorA::Yield1 {borrowed, ..} => {
                let borrowed: &String = unsafe {&**borrowed};
                println!("{} world", borrowed);
                *self = GeneratorA::Exit;
                GeneratorState::Complete(())
            }
```

```
            GeneratorA::Exit => panic!("Can't advance an exited generator!"),
        }
    }
}
}
```

Rust

请记住，我们的例子是我们生成的生成器，它的原始文件像这样:

```
let mut gen = move || {
        let to_borrow = String::from("Hello");
        let borrowed = &to_borrow;
        yield borrowed.len();
        println!("{} world!", borrowed);
    };
```

Rust

下面是我们如何运行这个状态机的示例，正如您所看到的，它完成了我们所期望的任务。但这仍然存在一个巨大的问题:

```
pub fn main() {
    let mut gen = GeneratorA::start();
    let mut gen2 = GeneratorA::start();

    if let GeneratorState::Yielded(n) = gen.resume() {
        println!("Got value {}", n);
    }

    if let GeneratorState::Yielded(n) = gen2.resume() {
        println!("Got value {}", n);
    }

    if let GeneratorState::Complete(()) = gen.resume() {
        ()
    };
}
enum GeneratorState<Y, R> {
    Yielded(Y),
    Complete(R),
}

trait Generator {
    type Yield;
    type Return;
    fn resume(&mut self) -> GeneratorState<Self::Yield, Self::Return>;
}

enum GeneratorA {
    Enter,
    Yield1 {
        to_borrow: String,
        borrowed: *const String,
    },
    Exit,
}

impl GeneratorA {
    fn start() -> Self {
        GeneratorA::Enter
    }
}
impl Generator for GeneratorA {
    type Yield = usize;
    type Return = ();
    fn resume(&mut self) -> GeneratorState<Self::Yield, Self::Return> {
            match self {
            GeneratorA::Enter => {
                let to_borrow = String::from("Hello");
                let borrowed = &to_borrow;
                let res = borrowed.len();
                *self = GeneratorA::Yield1 {to_borrow, borrowed:
std::ptr::null()};

                // We set the self-reference here
                if let GeneratorA::Yield1 {to_borrow, borrowed} = self {
```

```
                    *borrowed = to_borrow;
                }

                GeneratorState::Yielded(res)
            }

            GeneratorA::Yield1 {borrowed, ..} => {
                let borrowed: &String = unsafe {&**borrowed};
                println!("{} world", borrowed);
                *self = GeneratorA::Exit;
                GeneratorState::Complete(())
            }
            GeneratorA::Exit => panic!("Can't advance an exited generator!"),
        }
    }
}
```

Rust

问题在于，如果在Safe Rust代码中,我们这样做:

```
 #![feature(never_type)] // Force nightly compiler to be used in playground
// by betting on it's true that this type is named after it's stabilization
date...
pub fn main() {
    let mut gen = GeneratorA::start();
    let mut gen2 = GeneratorA::start();

    if let GeneratorState::Yielded(n) = gen.resume() {
        println!("Got value {}", n);
    }

    std::mem::swap(&mut gen, &mut gen2); // <--- Big problem!

    if let GeneratorState::Yielded(n) = gen2.resume() {
        println!("Got value {}", n);
    }

    // This would now start gen2 since we swapped them.
    if let GeneratorState::Complete(()) = gen.resume() {
        ()
    };
}
enum GeneratorState<Y, R> {
    Yielded(Y),
    Complete(R),
}

trait Generator {
    type Yield;
    type Return;
    fn resume(&mut self) -> GeneratorState<Self::Yield, Self::Return>;
}

enum GeneratorA {
    Enter,
    Yield1 {
        to_borrow: String,
        borrowed: *const String,
    },
    Exit,
}

impl GeneratorA {
    fn start() -> Self {
        GeneratorA::Enter
    }
}
impl Generator for GeneratorA {
    type Yield = usize;
    type Return = ();
    fn resume(&mut self) -> GeneratorState<Self::Yield, Self::Return> {
            match self {
            GeneratorA::Enter => {
                let to_borrow = String::from("Hello");
                let borrowed = &to_borrow;
```

```
                let res = borrowed.len();
                *self = GeneratorA::Yield1 {to_borrow, borrowed:
std::ptr::null()};

                // We set the self-reference here
                if let GeneratorA::Yield1 {to_borrow, borrowed} = self {
                    *borrowed = to_borrow;
                }

                GeneratorState::Yielded(res)
            }

            GeneratorA::Yield1 {borrowed, ..} => {
                let borrowed: &String = unsafe {&**borrowed};
                println!("{} world", borrowed);
                *self = GeneratorA::Exit;
                GeneratorState::Complete(())
            }
            GeneratorA::Exit => panic!("Can't advance an exited generator!"),
        }
    }
}
```

Rust

运行代码并比较结果。你看到问题了吗?

等等? “Hello”怎么了? 为什么我们的代码出错了?

事实证明，虽然上面的例子编译得很好，但是我们在使用安全Rust时将这个API的使用者暴露在可能的内存未定义行为和其他内存错误中。 这是个大问题！

实际上，我已经强制上面的代码使用编译器的夜间版本。 如果您在playground上运行上面的示例，您将看到它在当前稳定状态(1.42.0)上运行时没有panic，但在当前夜间状态(1.44.0)上panic。 太可怕了！

我们将在下一章用一个稍微简单一点的例子来解释这里发生了什么，我们将使用 Pin 来修复我们的生成器，所以不用担心，您将看到到底出了什么问题，看看 Pin 如何能够帮助我们在一秒钟内安全地处理自引用类型。

在我们详细解释这个问题之前，让我们通过了解生成器和 async 关键字之间的关系来结束本章。

异步和生成器

Futures 在Rust中被实现为状态机，就像生成器是状态机一样。

您可能已经注意到异步块中使用的语法和生成器中使用的语法的相似之处:

```
let mut gen = move || {
        let to_borrow = String::from("Hello");
        let borrowed = &to_borrow;
        yield borrowed.len();
        println!("{} world!", borrowed);
    };
```

Rust

比较一下异步块的类似例子:

```
let mut fut = async {
        let to_borrow = String::from("Hello");
        let borrowed = &to_borrow;
        SomeResource::some_task().await;
        println!("{} world!", borrowed);
    };
```

Rust

不同之处在于，Futures 的状态与 Generator 的状态不同。

异步块将返回一个 Future 而不是 Generator，但是 Future 的工作方式和 Generator 的内部工作方式是相似的。

我们不调用 `Generator::resume` ，而是调用 `Future::poll` ，并且不返回 generated 或 Complete，而是返回 Pending 或 Ready。 Future中的每一个await就像生成器中的一个yield。

你看到他们现在是怎么联系起来的了吗?

这就是为什么理解了生成器如何工作以及他需要面对的挑战,也就理解了Future如何工作以及它需要面对的挑战。

跨yield/await的借用就是这样.

奖励部分-正在使用的自引用生成器

感谢 PR#45337 ,你可以在nightly版本中使用static关键字运行上面的例子. 你可以试试:

> 要注意的是，API可能会发生改变。 在我撰写本书时，生成器API有一个更改，添加了对“ resume”参数的支持，以便传递到生成器闭包中。 可以关注RFC#033的相关问题#4312的进展。

```
 #![feature(generators, generator_trait)]
use std::ops::{Generator, GeneratorState};


pub fn main() {
    let gen1 = static || {
        let to_borrow = String::from("Hello");
        let borrowed = &to_borrow;
        yield borrowed.len();
        println!("{} world!", borrowed);
    };

    let gen2 = static || {
        let to_borrow = String::from("Hello");
        let borrowed = &to_borrow;
        yield borrowed.len();
        println!("{} world!", borrowed);
    };

    let mut pinned1 = Box::pin(gen1);
    let mut pinned2 = Box::pin(gen2);

    if let GeneratorState::Yielded(n) = pinned1.as_mut().resume(()) {
        println!("Gen1 got value {}", n);
    }

    if let GeneratorState::Yielded(n) = pinned2.as_mut().resume(()) {
        println!("Gen2 got value {}", n);
    };

    let _ = pinned1.as_mut().resume(());
    let _ = pinned2.as_mut().resume(());
}
```

Rust

六 Pin

概述

> 译者注: Pin是在使用Future时一个非常重要的概念,我的理解是: 通过使用Pin,让用户无法安全的获取到 `&mut T` ,进而无法进行上述例子中的swap. 如果你觉得你的和这个struct没有自引用的问题,你可以自己实现UnPin.

1. 了解如何使用Pin以及当你自己实现 `Future` 的时候为什么需要Pin
2. 理解如何让自引用类型被安全的使用
3. 理解跨`await`借用是如何实现的
4. 制定一套实用的规则来帮助你使用Pin

Pin是在RFC#2349中被提出的.

让我们直接了当的说吧,Pin是这一系列概念中很难一开始就搞明白的,但是一旦你理解了其心智模型,就会觉得非常容易理解.

定义

Pin只与指针有关,在Rust中引用也是指针.

Pin有 `Pin` 类型和 `Unpin` 标记组成(UnPin是Rust中为数不多的几个auto trait). Pin存在的目的就是为了让那些实现了 `!UnPin` 的类型遵守特定的规则.

是的，你是对的，这里是双重否定 `!Unpin` 的意思是“not-un-pin”。

> 这个命名方案是 Rusts 的安全特性之一，它故意测试您是否因为太累而无法安全地使用这个标记来实现类型。 如果你因为 `UnPin` 开始感到困惑，或者甚至生气，那么你就应该这样做！ 是时候放下工作，以全新的心态重新开始明天的生活了，这是一个好兆头。

更严肃地说，我认为有必要提到，选择这些名字是有正当理由的。 命名并不容易，我曾经考虑过在这本书中重命名 `Unpin` 和 `!UnPin` ,使他们更容易理解。

然而，一位经验丰富的Rust社区成员让我相信，当简单地给这些标记起不同的名字时，有太多的细微差别和边缘情况需要考虑，而这些很容易被忽略，我相信我们将不得不习惯它们并按原样使用它们。

如果你愿意，你可以从内部讨论中读到一些讨论。

Pinning和自引用结构

让我们从上一章(生成器那一章)停止的地方开始，通过使用一些比状态机更容易推理的自引用结构，使我们在生成器中看到的使用自引用结构的问题变得简单得多:

现在我们的例子是这样的:

```
use std::pin::Pin;

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

impl Test {
    fn new(txt: &str) -> Self {
        let a = String::from(txt);
        Test {
            a,
            b: std::ptr::null(),
        }
    }

    fn init(&mut self) {
        let self_ref: *const String = &self.a;
        self.b = self_ref;
    }

    fn a(&self) -> &str {
        &self.a
    }

    fn b(&self) -> &String {
        unsafe {&*(self.b)}
    }
}
```

Rust

让我们来回顾一下这个例子，因为我们将在本章的其余部分使用它。

我们有一个自引用结构体 `Test` 。 Test需要创建一个init方法，这个方法很奇怪，但是为了尽可能简短，我们需要这个方法。

Test 提供了两种方法来获取字段 a 和 b 值的引用。 因为 b 是 a 的参考，所以我们把它存储为一个指针，因为 Rust 的借用规则不允许我们定义这个生命周期。

现在，让我们用这个例子来详细解释我们遇到的问题:

```
fn main() {
    let mut test1 = Test::new("test1");
    test1.init();
    let mut test2 = Test::new("test2");
    test2.init();

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

}
```

Rust

在main函数中,我们首先实例化Test的两个实例,然后输出test1和test2各字段的值,结果如我们所料:

```
a: test1, b: test1
a: test2, b: test2
```

让我们看看，如果我们将存储在 test1指向的内存位置的数据与存储在 test2指向的内存位置的数据进行交换，会发生什么情况，反之亦然。

```
fn main() {
    let mut test1 = Test::new("test1");
    test1.init();
    let mut test2 = Test::new("test2");
    test2.init();

    println!("a: {}, b: {}", test1.a(), test1.b());
    std::mem::swap(&mut test1, &mut test2);
    println!("a: {}, b: {}", test2.a(), test2.b());

}
```

Rust

我们可能会认为会打印两边test1,比如:

```
a: test1, b: test1
a: test1, b: test1
```

Rust

但是实际上我们得到的是:

```
a: test1, b: test1
a: test1, b: test2
```

指向 test2.b 的指针仍然指向test1内部的旧位置。 该结构不再是自引用的，它保存指向不同对象中的字段的指针。 这意味着我们不能再依赖test2.b的生存期与test2的生存期绑定在一起。

如果你仍然不相信，这至少可以说服你:

```
fn main() {
    let mut test1 = Test::new("test1");
    test1.init();
    let mut test2 = Test::new("test2");
    test2.init();

    println!("a: {}, b: {}", test1.a(), test1.b());
    std::mem::swap(&mut test1, &mut test2);
    test1.a = "I've totally changed now!".to_string();
    println!("a: {}, b: {}", test2.a(), test2.b());

}
```

Rust

这是不应该发生的。 目前还没有严重的错误，但是您可以想象，使用这些代码很容易创建严重的错误。

我创建了一个图表来帮助可视化正在发生的事情:

图1: 交换前后

正如你看到的,这不是我们想要的结果. 这很容易导致段错误,也很容易导致其他意想不到的未知行为以及失败.

固定在栈上

现在，我们可以通过使用 `Pin` 来解决这个问题。 让我们来看看我们的例子是什么样的:

```
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 {
        let a = String::from(txt);
        Test {
            a,
            b: std::ptr::null(),
            // This makes our type `!Unpin`
            _marker: PhantomPinned,
        }
    }
    fn init<'a>(self: Pin<&'a mut Self>) {
        let self_ptr: *const String = &self.a;
        let this = unsafe { self.get_unchecked_mut() };
        this.b = self_ptr;
    }

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

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

Rust

现在，我们在这里所做的就是固定到一个栈地址。如果我们的类型实现了 `!UnPin` ，那么它将总是 `unsafe` 。

我们在这里使用相同的技巧，包括需要 init。 如果我们想要解决这个问题并让用户避免 `unsafe` ，我们需要将数据钉在堆上，我们马上就会展示这一点。

让我们看看如果我们现在运行我们的例子会发生什么：

```
pub fn main() {
    // test1 is safe to move before we initialize it
    let mut test1 = Test::new("test1");
    // Notice how we shadow `test1` to prevent it from beeing accessed again
    let mut test1 = unsafe { Pin::new_unchecked(&mut test1) };
    Test::init(test1.as_mut());

    let mut test2 = Test::new("test2");
    let mut test2 = unsafe { Pin::new_unchecked(&mut test2) };
    Test::init(test2.as_mut());

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

Rust

现在，如果我们尝试使用上次使我们陷入麻烦的问题，您将得到一个编译错误。

```
pub fn main() {
    let mut test1 = Test::new("test1");
    let mut test1 = unsafe { Pin::new_unchecked(&mut test1) };
    Test::init(test1.as_mut());

    let mut test2 = Test::new("test2");
    let mut test2 = unsafe { Pin::new_unchecked(&mut test2) };
    Test::init(test2.as_mut());

    println!("a: {}, b: {}", Test::a(test1.as_ref()), Test::b(test1.as_ref()));
    std::mem::swap(test1.get_mut(), test2.get_mut());
    println!("a: {}, b: {}", Test::a(test2.as_ref()), Test::b(test2.as_ref()));
}
```

Rust

正如您从运行代码所得到的错误中看到的那样，类型系统阻止我们交换固定指针。

> 需要注意的是，栈pinning总是依赖于我们所在的当前栈帧，因此我们不能在一个栈帧中创建一个自引用对象并返回它，因为任何指向“self”的指针都是无效的。 如果你把一个值固定在一个栈上，这也会让你承担很多责任。 一个很容易犯的错误是，忘记对原始变量进行阴影处理，因为这样可以在初始化后drop固定的指针并访问原来的值:
>
> ```
> fn main() {
>    let mut test1 = Test::new("test1");
>    let mut test1_pin = unsafe { Pin::new_unchecked(&mut test1) };
>    Test::init(test1_pin.as_mut());
>    drop(test1_pin);
>
>    let mut test2 = Test::new("test2");
>    mem::swap(&mut test1, &mut test2);
>    println!("Not self referential anymore: {:?}", test1.b);
> }
> ```
>
> Rust

固定在堆上

为了完整性，让我们删除一些不安全的内容，通过以堆分配为代价来消除 `init` 方法。 固定到堆是安全的，这样用户不需要实现任何不安全的代码:

```
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 a = String::from(txt);
        let t = Test {
            a,
            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<'a>(self: Pin<&'a Self>) -> &'a str {
        &self.get_ref().a
    }

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

pub fn main() {
    let mut test1 = Test::new("test1");
    let mut 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());
}
```

Rust

事实上就算是 `!Unpin` 有意义,固定一个堆分配的值也是安全的。 一旦在堆上分配了数据，它就会有一个稳定的地址。

作为 API 的用户，我们不需要特别注意并确保自引用指针保持有效。

也有一些方法能够对固定栈上提供一些安全保证，但是现在我们使用pin_project这个包来实现这一点。

Pinning的一些实用规则

1. 针对 `T:UnPin` (这是默认值), `Pin<'a,T>` 完全定价与 `&'a mut T` . 换句话说: `UnPin` 意味着这个类型即使在固定时也可以移动，所以Pin对这个类型没有影响。
2. 针对 `T:!UnPin` ,从 `Pin< T>` 获取到 `&mut T` ,则必须使用unsafe. 换句话说, `!Unpin` 能够阻止API的使用者移动T,除非他写出unsafe的代码.
3. Pinning对于内存分配没有什么特别的作用，比如将其放入某个“只读”内存或任何奇特的内存中。 它只使用类型系统来防止对该值进行某些操作。
4. 大多数标准库类型实现 Unpin。 这同样适用于你在 Rust 中遇到的大多数“正常”类型。 `Future` 和 `Generators` 是两个例外。
5. Pin的主要用途就是自引用类型,Rust语言的所有这些调整就是为了允许这个. 这个API中仍然有一些问题需要探讨.
6. `!UnPin` 这些类型的实现很有可能是不安全的. 在这种类型被钉住后移动它可能会导致程序崩溃。 在撰写本书时，创建和读取自引用结构的字段仍然需要不安全的方法(唯一的方法是创建一个包含指向自身的原始指针的结构)。
7. 当使用nightly版本时,你可以在一个使用特性标记在一个类型上添加 `!UnPin` . 当使用stable版本时,可以将std: : marker: : PhantomPinned 添加到类型上。
8. 你既可以固定一个栈上的对象也可以固定一个堆上的对象.
9. 将一个 `!UnPin` 的指向栈上的指针固定需要unsafe.
10. 将一个 `!UnPin` 的指向堆上的指针固定,不需要unsafe,可以直接使用 `Box::Pin` .

> 不安全的代码并不意味着它真的“unsafe”，它只是减轻了通常从编译器得到的保证。一个不安全的实现可能是完全安全的，但是您没有编译器保证的安全网。

映射/结构体的固定

简而言之，投影是一个编程语言术语。 Mystruct.field1是一个投影。 结构体的固定是在每一个字段上使用Pin。 这里有一些注意事项，您通常不会看到，因此我参考相关文档。

Pin和Drop

Pin保证从值被固定到被删除的那一刻起一直存在。 而在Drop实现中，您需要一个可变的self 引用，这意味着在针对固定类型实现 Drop 时必须格外小心。

把它们放在一起

当我们实现自己的 `Futures` 的时候,这正是我们要做的，我们很快就完成了。

奖励部分

修复我们实现的自引用生成器以及学习更多的关于Pin的知识.

但是现在，让我们使用 Pin 来防止这个问题。 我一直在评论，以便更容易地发现和理解我们需要做出的改变。

```
 #![feature(optin_builtin_traits, negative_impls)] // needed to implement `!Unpin`
use std::pin::Pin;

pub fn main() {
    let gen1 = GeneratorA::start();
    let gen2 = GeneratorA::start();
    // Before we pin the pointers, this is safe to do
    // std::mem::swap(&mut gen, &mut gen2);

    // constructing a `Pin::new()` on a type which does not implement `Unpin` is
    // unsafe. A value pinned to heap can be constructed while staying in safe
    // Rust so we can use that to avoid unsafe. You can also use crates like
    // `pin_utils` to pin to the stack safely, just remember that they use
    // unsafe under the hood so it's like using an already-reviewed unsafe
    // implementation.

    let mut pinned1 = Box::pin(gen1);
    let mut pinned2 = Box::pin(gen2);

    // Uncomment these if you think it's safe to pin the values to the stack
instead
    // (it is in this case). Remember to comment out the two previous lines first.
    //let mut pinned1 = unsafe { Pin::new_unchecked(&mut gen1) };
    //let mut pinned2 = unsafe { Pin::new_unchecked(&mut gen2) };

    if let GeneratorState::Yielded(n) = pinned1.as_mut().resume() {
        println!("Gen1 got value {}", n);
    }

    if let GeneratorState::Yielded(n) = pinned2.as_mut().resume() {
        println!("Gen2 got value {}", n);
    };

    // This won't work:
    // std::mem::swap(&mut gen, &mut gen2);
    // This will work but will just swap the pointers so nothing bad happens here:
    // std::mem::swap(&mut pinned1, &mut pinned2);

    let _ = pinned1.as_mut().resume();
    let _ = pinned2.as_mut().resume();
}

enum GeneratorState<Y, R> {
    Yielded(Y),
    Complete(R),
}

trait Generator {
    type Yield;
    type Return;
    fn resume(self: Pin<&mut Self>) -> GeneratorState<Self::Yield, Self::Return>;
}

enum GeneratorA {
    Enter,
```

```
    Yield1 {
        to_borrow: String,
        borrowed: *const String,
    },
    Exit,
}

impl GeneratorA {
    fn start() -> Self {
        GeneratorA::Enter
    }
}

// This tells us that the underlying pointer is not safe to move after pinning.
// In this case, only we as implementors "feel" this, however, if someone is
// relying on our Pinned pointer this will prevent them from moving it. You need
// to enable the feature flag ` #![feature(optin_builtin_traits)]` and use the
// nightly compiler to implement `!Unpin`. Normally, you would use
// `std::marker::PhantomPinned` to indicate that the struct is `!Unpin`.
impl !Unpin for GeneratorA { }

impl Generator for GeneratorA {
    type Yield = usize;
    type Return = ();
    fn resume(self: Pin<&mut Self>) -> GeneratorState<Self::Yield, Self::Return> {
        // lets us get ownership over current state
        let this = unsafe { self.get_unchecked_mut() };
            match this {
            GeneratorA::Enter => {
                let to_borrow = String::from("Hello");
                let borrowed = &to_borrow;
                let res = borrowed.len();
                *this = GeneratorA::Yield1 {to_borrow, borrowed:
std::ptr::null()};

                // Trick to actually get a self reference. We can't reference
                // the `String` earlier since these references will point to the
                // location in this stack frame which will not be valid anymore
                // when this function returns.
                if let GeneratorA::Yield1 {to_borrow, borrowed} = this {
                    *borrowed = to_borrow;
                }

                GeneratorState::Yielded(res)
            }

            GeneratorA::Yield1 {borrowed, ..} => {
                let borrowed: &String = unsafe {&**borrowed};
                println!("{} world", borrowed);
                *this = GeneratorA::Exit;
                GeneratorState::Complete(())
            }
            GeneratorA::Exit => panic!("Can't advance an exited generator!"),
        }
```

```
    }
}
```

Rust

现在，正如你所看到的，这个 API 的使用者必须: 1. 将值装箱，从而在堆上分配它 2. 使用 `unafe` 然后把值固定到栈上。 用户知道如果他们事后移动了这个值，那么他们在就违反了当他们使用unsafe时候做出的承诺,也就是一直持有.

希望在这之后，你会知道当你在一个异步函数中使用 `yield` 或者 `await` 关键词时会发生什么，以及如果我们想要安全地跨yield/await借用时。,为什么我们需要 `Pin`

七 实现Futures–主要例子

我们将用一个伪reactor和一个简单的执行器创建我们自己的 `Futures` ，它允许你在浏览器中编辑和运行代码

我将向您介绍这个示例，但是如果您想更深入的研究它，您可以克隆存储库并自己处理代码，或者直接从下一章复制代码。

readme文件中解释了几个分支，其中有两个分支与本章相关。 主分支是我们在这里经过的例子， `basic_example_commented` 分支是这个具有大量注释的例子

> 如果您希望跟随我们的步骤，可以通过创建一个新的文件夹初始化一个新的 cargo 项目，并在其中运行 cargo init。所有的一切都在main.rs文件中.

实现我们自己的Futures

让我们先从引入依赖开始:

```
use std::{
    future::Future, pin::Pin, sync::{mpsc::{channel, Sender}, Arc, Mutex},
    task::{Context, Poll, RawWaker, RawWakerVTable, Waker},
    thread::{self, JoinHandle}, time::{Duration, Instant}
};
```

Rust

执行器

执行器的责任是获取一个或多个 `Future` 然后运行他们到完成。

执行器拿到 `Future` 后的第一件事就是轮询它.

轮询后可以发现以下三种情况: 1. `Future` 返回Ready,然后就可以调度其他任何后续操作. 2. 这个 `Future` 从未被轮询过,所以传入一个 `Waker` ,然后将它挂起 3. 这个 `Future` 已经被轮询过,但是返回 `Pending`

Rust通过 `Waker` 为Reactor和执行器提供了通信方式. reactor存储这个 `Waker` ,然后在 `Future` 等待的事件完成的时候调用 `Waker: : wake ()` ,这样 `Future` 就会被再次轮询.

我们的执行器会是这个样子:

```
// Our executor takes any object which implements the `Future` trait
fn block_on<F: Future>(mut future: F) -> F::Output {

    // the first thing we do is to construct a `Waker` which we'll pass on to
    // the `reactor` so it can wake us up when an event is ready.
    let mywaker = Arc::new(MyWaker{ thread: thread::current() });
    let waker = waker_into_waker(Arc::into_raw(mywaker));

    // The context struct is just a wrapper for a `Waker` object. Maybe in the
    // future this will do more, but right now it's just a wrapper.
    let mut cx = Context::from_waker(&waker);

    // So, since we run this on one thread and run one future to completion
    // we can pin the `Future` to the stack. This is unsafe, but saves an
    // allocation. We could `Box::pin` it too if we wanted. This is however
    // safe since we shadow `future` so it can't be accessed again and will
    // not move until it's dropped.
    let mut future = unsafe { Pin::new_unchecked(&mut future) };

    // We poll in a loop, but it's not a busy loop. It will only run when
    // an event occurs, or a thread has a "spurious wakeup" (an unexpected wakeup
    // that can happen for no good reason).
    let val = loop {

        match Future::poll(pinned, &mut cx) {

            // when the Future is ready we're finished
            Poll::Ready(val) => break val,

            // If we get a `pending` future we just go to sleep...
            Poll::Pending => thread::park(),
        };
    };
    val
}
```

Rust

在本章的所有例子中，我都选择了对代码进行广泛的注释。 我发现沿着这条路走会更容易一些，所以我不会在这里重复自己的话，只关注一些可能需要进一步解释的重要方面。

现在你已经阅读了这么多关于生成器和 Pin 的内容，这应该很容易理解。 `Future` 是一个状态机，每一个 `await` 点也是一个 `yield` 点。 我们可以跨越 `await` 借用，我们遇到的问题与跨 `yield` 借用时完全一样。

> `Context` 只是 `Waker` 的包装器, 至少在我写这本书的时候，它仅仅是这样。 在未来， `Context` 对象可能不仅仅是包装一个 `Waker` (译者注,原文是Future,应该有误)，因此这种额外的抽象可以提供一些灵活性。

正如在关于生成器的章节中解释的那样，我们使用Pin来保证允许 `Future` 有自引用。

实现Future

`Future` 有一个定义良好的接口，这意味着他们可以用于整个生态系统。

我们可以将这些 `Future` 连接起来，这样一旦 `leaf-future` 准备好了，我们就可以执行一系列操作，直到任务完成或者我们到达另一个 `leaf-future` ，我们将等待并将控制权交给调度程序。

我们 `Future` 的实现是这样的:

```
// This is the definition of our `Waker`. We use a regular thread-handle here.
// It works but it's not a good solution. It's easy to fix though, I'll explain
// after this code snippet.
#[derive(Clone)]
struct MyWaker {
    thread: thread::Thread,
}

// This is the definition of our `Future`. It keeps all the information we
// need. This one holds a reference to our `reactor`, that's just to make
// this example as easy as possible. It doesn't need to hold a reference to
// the whole reactor, but it needs to be able to register itself with the
// reactor.
#[derive(Clone)]
pub struct Task {
    id: usize,
    reactor: Arc<Mutex<Box<Reactor>>>,
    data: u64,
}

// These are function definitions we'll use for our waker. Remember the
// "Trait Objects" chapter earlier.
fn mywaker_wake(s: &MyWaker) {
    let waker_ptr: *const MyWaker = s;
    let waker_arc = unsafe {Arc::from_raw(waker_ptr)};
    waker_arc.thread.unpark();
}

// Since we use an `Arc` cloning is just increasing the refcount on the smart
// pointer.
fn mywaker_clone(s: &MyWaker) -> RawWaker {
    let arc = unsafe { Arc::from_raw(s) };
    std::mem::forget(arc.clone()); // increase ref count
    RawWaker::new(Arc::into_raw(arc) as *const (), &VTABLE)
}

// This is actually a "helper funtcion" to create a `Waker` vtable. In contrast
// to when we created a `Trait Object` from scratch we don't need to concern
// ourselves with the actual layout of the `vtable` and only provide a fixed
// set of functions
const VTABLE: RawWakerVTable = unsafe {
    RawWakerVTable::new(
        |s| mywaker_clone(&*(s as *const MyWaker)),     // clone
        |s| mywaker_wake(&*(s as *const MyWaker)),      // wake
        |s| mywaker_wake(*(s as *const &MyWaker)),      // wake by ref
        |s| drop(Arc::from_raw(s as *const MyWaker)),   // decrease refcount
    )
};

// Instead of implementing this on the `MyWaker` object in `impl Mywaker...` we
// just use this pattern instead since it saves us some lines of code.
fn waker_into_waker(s: *const MyWaker) -> Waker {
    let raw_waker = RawWaker::new(s as *const (), &VTABLE);
    unsafe { Waker::from_raw(raw_waker) }
}
```

```
impl Task {
    fn new(reactor: Arc<Mutex<Box<Reactor>>>, data: u64, id: usize) -> Self {
        Task { id, reactor, data }
    }
}

// This is our `Future` implementation
impl Future for Task {
    type Output = usize;

    // Poll is the what drives the state machine forward and it's the only
    // method we'll need to call to drive futures to completion.
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {

        // We need to get access the reactor in our `poll` method so we acquire
        // a lock on that.
        let mut r = self.reactor.lock().unwrap();

        // First we check if the task is marked as ready
        if r.is_ready(self.id) {

            // If it's ready we set its state to `Finished`
            *r.tasks.get_mut(&self.id).unwrap() = TaskState::Finished;
            Poll::Ready(self.id)

        // If it isn't finished we check the map we have stored in our Reactor
        // over id's we have registered and see if it's there
        } else if r.tasks.contains_key(&self.id) {

            // This is important. The docs says that on multiple calls to poll,
            // only the Waker from the Context passed to the most recent call
            // should be scheduled to receive a wakeup. That's why we insert
            // this waker into the map (which will return the old one which will
            // get dropped) before we return `Pending`.
            r.tasks.insert(self.id, TaskState::NotReady(cx.waker().clone()));
            Poll::Pending
        } else {

            // If it's not ready, and not in the map it's a new task so we
            // register that with the Reactor and return `Pending`
            r.register(self.data, cx.waker().clone(), self.id);
            Poll::Pending
        }

        // Note that we're holding a lock on the `Mutex` which protects the
        // Reactor all the way until the end of this scope. This means that
        // even if our task were to complete immidiately, it will not be
        // able to call `wake` while we're in our `Poll` method.

        // Since we can make this guarantee, it's now the Executors job to
        // handle this possible race condition where `Wake` is called after
        // `poll` but before our thread goes to sleep.
    }
}
```

Rust

这大部分都是直截了当的。 令人困惑的部分是我们需要构建 Waker 的奇怪方式，但是由于我们已经从原始部分创建了我们自己的 trait 对象，这看起来很熟悉。 事实上，这更简单。

我们在这里使用一个Arc来传递一个引用计数的MyWaker的借用。 这是相当正常的，并且使得这个操作变得简单和安全。 克隆一个Waker只是增加一个计数。Drop一个Waker只是简单地减少一个计数.

在我们这种特定场景下,我们选择不使用 `Arc` . 而使用这种更低层次方式实现的Waker才可以允许我们这么做.

事实上，如果我们只使用 Arc，那么我们就没有理由费尽心思去创建自己的 vtable 和 RawWaker。 我们可以实现一个普通的trait。

幸运的是，将来在标准库中也可以实现这个功能。 目前这个特性仍然在实验中，但是我猜想在成熟之后，这个特性将会成为标准库的一部分。

我们选择在这里传入一个整个reactor的引用, 这不正常。 reactor通常是一个全局性的资源，让我们注册感兴趣的事而不需要传入一个引用.

> **为什么在一个Lib中使用park/unpark是一个坏主意**
>
> 他很容易死锁,因为任何人都可以获得执行器所在线程的句柄,然后调用park/unpark.
>
> 1. 一个future可以在另一个不同的线程上unpark执行器线程
> 2. 我们的执行器认为数据准备好了,然后醒来去轮询这个 `Future`
> 3. 当被轮询时,这个 `Future` 还没有准备好,但是恰在此时, `Reactor` 收到事件,调用了 `Wake()` 来unpark我们的线程.
> 4. 这可能发生在我们再次睡眠之前,因为这些操作完全是并行的.
> 5. 我们的reactor已经调用过 `wake` ,但是我们的线程仍然在睡眠,因为刚刚调用wake的时候,我们的线程是醒着的.
> 6. 我们发生了死锁,然后我们的程序停止工作.
>
> 有一种情况是，我们的线程可能会出现所谓的虚假唤醒(可能会出乎意料地发生)，如果我们运气不好，这可能会导致同样的死锁

有几种更好的方案,比如:

- std::sync::CondVar
- crossbeam::sync::Parker

Reactor

这是最后的冲刺阶段，并不完全与 `Future` 相关，但是我们需要它来让我们的例子运行起来。

由于大多数时候并发只有在与外部世界(或者至少是一些外围设备)进行交互时才有意义，因此我们需要一些东西来抽象这些异步的交互.

这就是reacotor的工作. 大多数时候你看到的reactor都是用Mio这个库. 它早多个平台上提供了非阻塞API和事件通知机制.

reactor通常会提供类似于TcpStream(或任何其他资源)的东西，只不过您用TcpStream来创建I/O请求,而用reactor来创建Future.

我们的示例任务是一个计时器，它只生成一个线程，并将其置于休眠状态，休眠时间为我们指定的秒数。 我们在这里创建的reactor将创建一个表示每个计时器的 `leaf-future` 。作为回报，reactor接收到一个唤醒器，一旦任务完成reactor将调用这个唤醒器。

为了能够在浏览器中运行这里的代码，没有太多真正的I/O，我们可以假装这实际上代表了一些有用的I/O操作。

我们的reactor看起来像这样:

```
// This is a "fake" reactor. It does no real I/O, but that also makes our
// code possible to run in the book and in the playground
// The different states a task can have in this Reactor
enum TaskState {
    Ready,
    NotReady(Waker),
    Finished,
}

// This is a "fake" reactor. It does no real I/O, but that also makes our
// code possible to run in the book and in the playground
struct Reactor {

    // we need some way of registering a Task with the reactor. Normally this
    // would be an "interest" in an I/O event
    dispatcher: Sender<Event>,
    handle: Option<JoinHandle<()>>,

    // This is a list of tasks
    tasks: HashMap<usize, TaskState>,
}

// This represents the Events we can send to our reactor thread. In this
// example it's only a Timeout or a Close event.
#[derive(Debug)]
enum Event {
    Close,
    Timeout(u64, usize),
}

impl Reactor {

    // We choose to return an atomic reference counted, mutex protected, heap
    // allocated `Reactor`. Just to make it easy to explain... No, the reason
    // we do this is:
    //
    // 1. We know that only thread-safe reactors will be created.
    // 2. By heap allocating it we can obtain a reference to a stable address
    // that's not dependent on the stack frame of the function that called `new`
    fn new() -> Arc<Mutex<Box<Self>>> {
        let (tx, rx) = channel::<Event>();
        let reactor = Arc::new(Mutex::new(Box::new(Reactor {
            dispatcher: tx,
            handle: None,
            tasks: HashMap::new(),
        })));

        // Notice that we'll need to use `weak` reference here. If we don't,
        // our `Reactor` will not get `dropped` when our main thread is finished
        // since we're holding internal references to it.

        // Since we're collecting all `JoinHandles` from the threads we spawn
        // and make sure to join them we know that `Reactor` will be alive
        // longer than any reference held by the threads we spawn here.
        let reactor_clone = Arc::downgrade(&reactor);
```

```
        // This will be our Reactor-thread. The Reactor-thread will in our case
        // just spawn new threads which will serve as timers for us.
        let handle = thread::spawn(move || {
            let mut handles = vec![];

            // This simulates some I/O resource
            for event in rx {
                println!("REACTOR: {:?}", event);
                let reactor = reactor_clone.clone();
                match event {
                    Event::Close => break,
                    Event::Timeout(duration, id) => {

                        // We spawn a new thread that will serve as a timer
                        // and will call `wake` on the correct `Waker` once
                        // it's done.
                        let event_handle = thread::spawn(move || {
                            thread::sleep(Duration::from_secs(duration));
                            let reactor = reactor.upgrade().unwrap();
                            reactor.lock().map(|mut r| r.wake(id)).unwrap();
                        });
                        handles.push(event_handle);
                    }
                }
            }

            // This is important for us since we need to know that these
            // threads don't live longer than our Reactor-thread. Our
            // Reactor-thread will be joined when `Reactor` gets dropped.
            handles.into_iter().for_each(|handle| handle.join().unwrap());
        });
        reactor.lock().map(|mut r| r.handle = Some(handle)).unwrap();
        reactor
    }

    // The wake function will call wake on the waker for the task with the
    // corresponding id.
    fn wake(&mut self, id: usize) {
        self.tasks.get_mut(&id).map(|state| {

            // No matter what state the task was in we can safely set it
            // to ready at this point. This lets us get ownership over the
            // the data that was there before we replaced it.
            match mem::replace(state, TaskState::Ready) {
                TaskState::NotReady(waker) => waker.wake(),
                TaskState::Finished => panic!("Called 'wake' twice on task: {}",
id),
                _ => unreachable!()
            }
        }).unwrap();
    }

    // Register a new task with the reactor. In this particular example
    // we panic if a task with the same id get's registered twice
```

```
    fn register(&mut self, duration: u64, waker: Waker, id: usize) {
        if self.tasks.insert(id, TaskState::NotReady(waker)).is_some() {
            panic!("Tried to insert a task with id: '{}', twice!", id);
        }
        self.dispatcher.send(Event::Timeout(duration, id)).unwrap();
    }

    // We send a close event to the reactor so it closes down our reactor-thread
    fn close(&mut self) {
        self.dispatcher.send(Event::Close).unwrap();
    }

    // We simply checks if a task with this id is in the state `TaskState::Ready`
    fn is_ready(&self, id: usize) -> bool {
        self.tasks.get(&id).map(|state| match state {
            TaskState::Ready => true,
            _ => false,
        }).unwrap_or(false)
    }
}

impl Drop for Reactor {
    fn drop(&mut self) {
        self.handle.take().map(|h| h.join().unwrap()).unwrap();
    }
}
```

Rust

虽然代码量很大，但实际上我们只是产生了一个新线程，并让它休眠一段时间，这是我们在创建任务时指定的。

虽然代码量很大，但实际上我们只是产生了一个新线程，并让它休眠一段时间，这是我们在创建任务时指定的。

在最后一章中，我们在一个可编辑的窗口中有整整200行，你可以按照自己喜欢的方式进行编辑和修改。

```
fn main() {
    // This is just to make it easier for us to see when our Future was resolved
    let start = Instant::now();

    // Many runtimes create a glocal `reactor` we pass it as an argument
    let reactor = Reactor::new();

    // Since we'll share this between threads we wrap it in a
    // atmically-refcounted- mutex.
    let reactor = Arc::new(Mutex::new(reactor));

    // We create two tasks:
    // - first parameter is the `reactor`
    // - the second is a timeout in seconds
    // - the third is an `id` to identify the task
    let future1 = Task::new(reactor.clone(), 1, 1);
    let future2 = Task::new(reactor.clone(), 2, 2);

    // an `async` block works the same way as an `async fn` in that it compiles
    // our code into a state machine, `yielding` at every `await` point.
    let fut1 = async {
        let val = future1.await;
        let dur = (Instant::now() - start).as_secs_f32();
        println!("Future got {} at time: {:.2}.", val, dur);
    };

    let fut2 = async {
        let val = future2.await;
        let dur = (Instant::now() - start).as_secs_f32();
        println!("Future got {} at time: {:.2}.", val, dur);
    };

    // Our executor can only run one and one future, this is pretty normal
    // though. You have a set of operations containing many futures that
    // ends up as a single future that drives them all to completion.
    let mainfut = async {
        fut1.await;
        fut2.await;
    };

    // This executor will block the main thread until the futures is resolved
    block_on(mainfut);

    // When we're done, we want to shut down our reactor thread so our program
    // ends nicely.
    reactor.lock().map(|mut r| r.close()).unwrap();
}
```

Rust

我添加了一个reactor感兴趣的事件的调试输出，这样我们可以观察到两件事:

1. `Waker` 这个对象如何像前面我们讨论的trai对象

2. 事件以何种顺序向reactor注册感兴趣的信息

Async/Await和并发Async/Await

Async 关键字可以用在 async fn (...)中的函数上，也可以用在 async { ... }中的块上。 两者都可以讲一个函数或者代码块转换成一个 `Future`

这些 `Future` 是相当简单的。 想象一下几章前我们的生成器。

每一个await就像一个yield,只不过不是生成一个值,而是生成Future,然后当轮询的时候返回响应的结果.

我们的 `mainfut` 包含两个 `non-leaf-future` ，它将在轮询中调用。 `non-leaf-future` 有一个 `poll` 方法, 这个方法简单的轮询他自己的内部Future,它内部的Future会被继续轮询,直到 `leaf-future` 返回 `Ready` 或者 `Pending` .

就我们现在的例子来看，它并不比常规的同步代码好多少。 对于我们来说，如果需要在同一时间等待多个 `Future` ，我们需要 `spawn` 它们，以便执行器同时运行它们。

现在我们的例子返回如下结果:

```
Future got 1 at time: 1.00.
Future got 2 at time: 3.00.
```

Rust

```
Future got 1 at time: 1.00.
Future got 2 at time: 2.00.
```

Rust

> 请注意，这并不意味着它们需要并行运行。 它们可以并行运行，但没有要求。 请记住，我们正在等待一些外部资源，这样我们就可以在一个线程上发出许多这样的调用，并在事件发生时处理每个事件

现在，我将向您介绍一些更好的资源，以实现一个更好的执行器。 现在你应该已经对 `Future` 的概念有了一个很好的理解。

下一步应该是了解更高级的运行时是如何工作的，以及它们如何实现不同的运行 Futures 的方式。

如果我是你，我接下来就会读这篇文章，并试着把它应用到我们的例子中去。

我希望在阅读完这篇文章后，能够更容易地探索Future和异步，我真的希望你们能够继续深入探索。

别忘了最后一章的练习。

奖励部分-暂停线程的更好办法

正如我们在本章前面解释的那样，仅仅调用 `thread::sleep` 并不足以实现一个合适的反应器。 你也可以使用类似crossbeam::sync::Parker中的Parker 这样的工具.

因为我们自己创建一个这样的Parker也不需要很多行代码，所以我们将展示如何通过使用 Condvar 和 Mutex 来解决这个问题。

我们自己的Parker:

```
#[derive(Default)]
struct Parker(Mutex<bool>, Condvar);

impl Parker {
    fn park(&self) {

        // We aquire a lock to the Mutex which protects our flag indicating if we
        // should resume execution or not.
        let mut resumable = self.0.lock().unwrap();

            // We put this in a loop since there is a chance we'll get woken, but
            // our flag hasn't changed. If that happens, we simply go back to
sleep.
            while !*resumable {

                // We sleep until someone notifies us
                resumable = self.1.wait(resumable).unwrap();
            }

        // We immidiately set the condition to false, so that next time we call
`park` we'll
        // go right to sleep.
        *resumable = false;
    }

    fn unpark(&self) {
        // We simply acquire a lock to our flag and sets the condition to
`runnable` when we
        // get it.
        *self.0.lock().unwrap() = true;

        // We notify our `Condvar` so it wakes up and resumes.
        self.1.notify_one();
    }
}
```

Rust

在 Rust 中的 Condvar 被设计为与互斥对象一起工作。 通常，您会认为在我们进入休眠之前, `self.0.lock().unwrap()` 不会释放锁, 这意味着我们的 `unpark` 永远获取不到锁,我们会陷入死锁。

使用 Condvar 我们可以避免这种情况，因为 Condvar 会消耗我们的锁，所以它会在我们睡觉的时候释放。 当我们再次恢复时，我们的 Condvar 会重新持有锁，这样我们就可以继续操作它。 这意味着我们需要对我们的执行器做一些非常细微的改变，比如:

```
fn block_on<F: Future>(mut future: F) -> F::Output {
    let parker = Arc::new(Parker::default()); // <--- NB!
    let mywaker = Arc::new(MyWaker { parker: parker.clone() }); <--- NB!
    let waker = mywaker_into_waker(Arc::into_raw(mywaker));
    let mut cx = Context::from_waker(&waker);

    // SAFETY: we shadow `future` so it can't be accessed again.
    let mut future = unsafe { Pin::new_unchecked(&mut future) };
    loop {
        match Future::poll(future.as_mut(), &mut cx) {
            Poll::Ready(val) => break val,
            Poll::Pending => parker.park(), // <--- NB!
        };
    }
}
```

Rust

我们需要像这样改变我们的唤醒器:

```
#[derive(Clone)]
struct MyWaker {
    parker: Arc<Parker>,
}

fn mywaker_wake(s: &MyWaker) {
    let waker_arc = unsafe { Arc::from_raw(s) };
    waker_arc.parker.unpark();
}
```

Rust

> 你可以查看由park/unpark引起的微妙问题的连接. 你可以在这里查看我们最终的版本如何避免了这个问题.

八 完整的例子

```
 fn main() {
    let start = Instant::now();
    let reactor = Reactor::new();

    let fut1 = async {
        let val = Task::new(reactor.clone(), 1, 1).await;
        println!("Got {} at time: {:.2}.", val, start.elapsed().as_secs_f32());
    };

    let fut2 = async {
        let val = Task::new(reactor.clone(), 2, 2).await;
        println!("Got {} at time: {:.2}.", val, start.elapsed().as_secs_f32());
    };

    let mainfut = async {
        fut1.await;
        fut2.await;
    };

    block_on(mainfut);
    reactor.lock().map(|mut r| r.close()).unwrap();
}

use std::{
    future::Future, sync::{ mpsc::{channel, Sender}, Arc, Mutex, Condvar},
    task::{Context, Poll, RawWaker, RawWakerVTable, Waker}, mem, pin::Pin,
    thread::{self, JoinHandle}, time::{Duration, Instant}, collections::HashMap
};
// ============================= EXECUTOR ====================================
#[derive(Default)]
struct Parker(Mutex<bool>, Condvar);

impl Parker {
    fn park(&self) {
        let mut resumable = self.0.lock().unwrap();
            while !*resumable {
                resumable = self.1.wait(resumable).unwrap();
            }
        *resumable = false;
    }

    fn unpark(&self) {
        *self.0.lock().unwrap() = true;
        self.1.notify_one();
    }
}

fn block_on<F: Future>(mut future: F) -> F::Output {
    let parker = Arc::new(Parker::default());
    let mywaker = Arc::new(MyWaker { parker: parker.clone() });
    let waker = mywaker_into_waker(Arc::into_raw(mywaker));
    let mut cx = Context::from_waker(&waker);

    // SAFETY: we shadow `future` so it can't be accessed again.
    let mut future = unsafe { Pin::new_unchecked(&mut future) };
```

```
    loop {
        match Future::poll(future.as_mut(), &mut cx) {
            Poll::Ready(val) => break val,
            Poll::Pending => parker.park(),
        };
    }
}
// ====================== FUTURE IMPLEMENTATION ==============================
#[derive(Clone)]
struct MyWaker {
    parker: Arc<Parker>,
}

#[derive(Clone)]
pub struct Task {
    id: usize,
    reactor: Arc<Mutex<Box<Reactor>>>,
    data: u64,
}

fn mywaker_wake(s: &MyWaker) {
    let waker_arc = unsafe { Arc::from_raw(s) };
    waker_arc.parker.unpark();
}

fn mywaker_clone(s: &MyWaker) -> RawWaker {
    let arc = unsafe { Arc::from_raw(s) };
    std::mem::forget(arc.clone()); // increase ref count
    RawWaker::new(Arc::into_raw(arc) as *const (), &VTABLE)
}

const VTABLE: RawWakerVTable = unsafe {
    RawWakerVTable::new(
        |s| mywaker_clone(&*(s as *const MyWaker)),   // clone
        |s| mywaker_wake(&*(s as *const MyWaker)),    // wake
        |s| mywaker_wake(*(s as *const &MyWaker)),    // wake by ref
        |s| drop(Arc::from_raw(s as *const MyWaker)), // decrease refcount
    )
};

fn mywaker_into_waker(s: *const MyWaker) -> Waker {
    let raw_waker = RawWaker::new(s as *const (), &VTABLE);
    unsafe { Waker::from_raw(raw_waker) }
}

impl Task {
    fn new(reactor: Arc<Mutex<Box<Reactor>>>, data: u64, id: usize) -> Self {
        Task { id, reactor, data }
    }
}

impl Future for Task {
    type Output = usize;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
        let mut r = self.reactor.lock().unwrap();
```

```
            if r.is_ready(self.id) {
                *r.tasks.get_mut(&self.id).unwrap() = TaskState::Finished;
                Poll::Ready(self.id)
            } else if r.tasks.contains_key(&self.id) {
                r.tasks.insert(self.id, TaskState::NotReady(cx.waker().clone()));
                Poll::Pending
            } else {
                r.register(self.data, cx.waker().clone(), self.id);
                Poll::Pending
            }
        }
    }
    // ============================= REACTOR ===================================
    enum TaskState {
        Ready,
        NotReady(Waker),
        Finished,
    }
    struct Reactor {
        dispatcher: Sender<Event>,
        handle: Option<JoinHandle<()>>,
        tasks: HashMap<usize, TaskState>,
    }

    #[derive(Debug)]
    enum Event {
        Close,
        Timeout(u64, usize),
    }

    impl Reactor {
        fn new() -> Arc<Mutex<Box<Self>>> {
            let (tx, rx) = channel::<Event>();
            let reactor = Arc::new(Mutex::new(Box::new(Reactor {
                dispatcher: tx,
                handle: None,
                tasks: HashMap::new(),
            })));

            let reactor_clone = Arc::downgrade(&reactor);
            let handle = thread::spawn(move || {
                let mut handles = vec![];
                for event in rx {
                    let reactor = reactor_clone.clone();
                    match event {
                        Event::Close => break,
                        Event::Timeout(duration, id) => {
                            let event_handle = thread::spawn(move || {
                                thread::sleep(Duration::from_secs(duration));
                                let reactor = reactor.upgrade().unwrap();
                                reactor.lock().map(|mut r| r.wake(id)).unwrap();
                            });
                            handles.push(event_handle);
                        }
                    }
```

```
            }
            handles.into_iter().for_each(|handle| handle.join().unwrap());
        });
        reactor.lock().map(|mut r| r.handle = Some(handle)).unwrap();
        reactor
    }

    fn wake(&mut self, id: usize) {
        let state = self.tasks.get_mut(&id).unwrap();
        match mem::replace(state, TaskState::Ready) {
            TaskState::NotReady(waker) => waker.wake(),
            TaskState::Finished => panic!("Called 'wake' twice on task: {}", id),
            _ => unreachable!()
        }
    }

    fn register(&mut self, duration: u64, waker: Waker, id: usize) {
        if self.tasks.insert(id, TaskState::NotReady(waker)).is_some() {
            panic!("Tried to insert a task with id: '{}', twice!", id);
        }
        self.dispatcher.send(Event::Timeout(duration, id)).unwrap();
    }

    fn close(&mut self) {
        self.dispatcher.send(Event::Close).unwrap();
    }

    fn is_ready(&self, id: usize) -> bool {
        self.tasks.get(&id).map(|state| match state {
            TaskState::Ready => true,
            _ => false,
        }).unwrap_or(false)
    }
}

impl Drop for Reactor {
    fn drop(&mut self) {
        self.handle.take().map(|h| h.join().unwrap()).unwrap();
    }
}
```

Rust

九 结论和练习

我们的实现采取了一些明显的捷径，可以使用一些改进。 实际上，深入研究代码并自己尝试一些事情是一个很好的学习方式。 如果你想探索更多，这里有一些很好的练习

编码park

使用 Thread: : park 和 Thread: : unpark 的大问题是用户可以从自己的代码访问这些相同的方法。 尝试使用另一种方法来挂起线程并根据我们的命令再次唤醒它。 一些提示:

- std::sync::CondVar
- crossbeam::sync::Parker

编码传递reactor

避免包装整个Reactor in a mutex and pass it around 在互斥对象中传递

首先，保护整个reactor并传递它是过分的。 我们只对同步它包含的部分信息感兴趣。 尝试将其重构出来，只同步访问真正需要的内容。

我建议您看看async_std驱动程序是如何实现的，以及tokio调度程序是如何实现的，以获得一些灵感。

- 你想使用Arc来传递这些信息的引用?
- 你是否想创建一个全局的reactor,这样他可以随时随地被访问?

创建一个更好的执行器

现在我们只支持一个一个Future运行. 大多数运行时都有一个spawn函数,让你能够启动一个future,然后await它. 这样你就可以同时运行多个Future.

正如我在本书开头所建议的那样，访问stjepan 的博客系列，了解如何实现你自己的执行者，是我将要开始的地方，并从那里开始。

进一步阅读

- The official Asyc book
- The async_std book
- Aron Turon: Designing futures for Rust
- Steve Klabnik’s presentation: Rust’s journey to Async/Await
- The Tokio Blog
- Stjepan’s blog with a series where he implements an Executor
- Jon Gjengset’s video on The Why, What and How of Pinning in Rust 有中英文字幕的B站链接
- Withoutboats blog series about async/await

转载说明