21闭包生成器异步的匿名类型生成

2020年3月9日 12:32

闭包,生成器和异步都能生成一些匿名的结构类型,让我们来仔细的看一下:

总结一下:三种都会生成匿名的类型,闭包和生成器是匿名结构体, 异步是生成匿名的enum.

闭包是返回的指针,trait 对象(Fn FnOnce FnMut), 生成器和异步返回的是类型(impl trait的类型,实现
了某些trait的类型.)

(1):闭包:
闭包在创建的时候,编译器帮我们生成了一个匿名struct类型.编译器通过自动分析闭包的内部逻辑,来决定改结构体包括哪些数据,以及这些数据改如何变化.

```
fn main() {
    let x = 1;
    let add_x = |a| x + a;
    let result = add_x(5);
    println!("result is {}", result);
}
```

编译器会自动生成下面的结构体:

```
struct Closure {
    inner1: i32,
}

impl Closure {
    fn call(&self, a: i32) -> i32 {
        self.inner1 + a
    }
}
```

如果发生了借用或者可变的借用,编译器会自动的实现下面的形式

```
struct Closure<'a, 'b> {
    inner1: i32,
    inner2: &'b i32,
    inner3: &'a mut i32, //这里假设是i32
}
```

因为有匿名结构体的存在,所以闭包是可以传递的,Box::new(|| {});
传递的就是这个匿名的结构体的值.
所以具体的编译器生成的大概的代码是:

```
// 类型的生命是一种全局的,类似一种全局的变量
// 因为闭包的参数是变化的,不定数的,所以每一种闭包都会生成这样的全局性
```

的结构体类型声明,

```
struct Closure {
    inner1: i32,
}

impl Closure {
    fn call(&self, a: i32) -> i32 {
        self.inner1 + a
    }
}

fn main() {
    let x = 1;
    let add_x = Closure{ inner1: x }; // 这个局部的,可以传递所有权,引用,可变引用
    let result = add_x.call(5);
    println!("result: {}", result);
}
```

因为闭包和函数指针能互相通用, 所以闭包返回的应该是一个指针,也就是一个实现了Fn() -> T, FnOnce(), FnMut(),
这些trait的一个对象,函数默认也是实现了的.所以能够通用了. 因为trait对象只是一个指针,对这个指针指向的所谓
的匿名结构体的大小一无所知,所以我们就不能修改匿名结构体内部的数据了.

(2): 生成器:

先来看下面的代码:

```
use std::ops::{Generator, GeneratorState};
fn main() {
    let mut g = || {
        let mut curr: u64 = 1;
        let mut next: u64 = 1;
        loop {
            let new_next = curr.checked_add(next);
            if let Some(new_next) = new_next {
                curr = next;
                next = new_next;
                yield curr; // 新的关键字
            } else {
                return;
            }
        }
    };

    loop {
        unsafe {
            match g.resume() {
                GeneratorState::Yielded(v) =>
                println!("{}", v),
                GeneratorState::Complete(_) => return,
```

```
            }
        }
    }
}
```

其实编译器生成的原理是:
let g= ||{}; //虽然没有添加返回值的类型声明,但是编译器会自动的判断出返回的类型.
这里是返回了一个实现了的Generator trait的struct类型.
这个trait中有resume方法. g就是一个实现了Generator trait的匿名struct类型,可以调用resume方法,
返回的是GeneratorState枚举.

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

enum GeneratorState<T1, T2> {
    Yielded(T1),
    Complete(T2),
}
```

上面的代码的生成过程是:

```
use std::ops::{ Generator, GeneratorState };
fn main() {
    let mut g = {
        enum _AnonymousGenerator {
            Start{curr: u64, next: u64},
            Yield{curr: u64, next: u64},
            Done,
        }

        impl Generator for _AnonymousGenerator {
            type Yield = u64;
            type Return = ();

            unsafe fn resume(&mut self) ->
            GeneratorState<Self::Yield, Self::Return> {
                use std::mem;
                match mem::replace(self,
                _AnonymousGenerator) {
                    // 交换,返回之前的
                    _AnonymousGenerator::Start{curr, next}
                    // 如果是Start,继续执行下面的
                    _AnonymousGenerator::Yield{curr, next}
                    => {
                        let new_next =
                        curr.checked_add(next);
```

```
                        if let Some(new_next) = new_next {
                            *self =
                            _AnonymousGenerator::Yield{curr:
                            next, next: new_next};
                            return
                            GeneratorState::Yielded(next);
                        } else {
                            *self =
                            _AnonymousGenerator::Done;
                            return
                            GenratorState::Complete(());
                        }
                    }

                    _AnonymousGenerator::Done => {
                        panic!("generator resumed after
                        completion.");
                    }
                }
            }
        }

        _AnonymousGenerator::Start{curr: 1, next: 1}
    };


    // main里面对g这个实现了Generator trait的调用
    loop {
        match g.resume() {
            GeneratorState::Yielded(v) => println!("{}",
            v),
            GeneratorState::Completd(_) => return,
        }
    }
}
```

(3):异步:

我们都知道rust在实现闭包时, 编译器通过隐式地创建一个匿名的struct保存捕获到的变量,并对struct
实现call方法实现函数调用,实现async/await函数时,由于需要记录当前所处的状态(每次await的时候都会导致一个状态),所以编译器往往生成一个匿名的enum,每个enum变体成员保存从外部或之前的await点捕获的变量.
需要我们注意的是: 生成器是异步的底层实现工具,也就是说异步的实现方式也是用生成器的方式实现的.

我们来看看在生成器的基础上,研发出来的异步或者叫做协程.
协程的设计,核心是async和await两个关键字,以及Future这个trait.

```
pub enum Poll<T> {
    Ready(T),
```

```
    Pending,
}

pub trait Future {
    type Output;
    fn poll(self: PinMut<Self>, cx: &mut Context) ->
    Poll<Self::Output>;
}
```

详细步骤:
对于实现了Future trait的类型,每次调用这个poll方法,其实就是查看这个类型的实例当前的状态
是什么,该状态为正在执行或者已经执行完毕.

Future可以组合,一个Future可以由其他的一个或者多个Future包装而成.比如我们可以实现一个新的
Future,他的结果是多个Future按照顺序得到的.或者实现一个Future,它的结果是两个Future
先返回的那个.

然后 我们还需要一个调度器Executor,标准库中有一个Executor的trait,具体实现由第三方库实现.
它应该有一个主事件循环,不断调用最外层每个收到了事件通知的Future的poll方法,外层
的Future的poll方法被调用时,它就会调用内部Future的poll方法,不断嵌套.如果这个Future
处于Pending状态,那么这个Future就应该设置好自己需要监听的事件信息,然后马上返回,
放弃占用CPU,等到合适的事件发生时,调度器则应该再次调用这个Future的poll方法,
驱动这个Future从上次退出的那里继续往下执行.

async和await关键字:

```
async fn async_fn(x: u8) -> u8 {
    let msg = read_from_network().await;
    let result = calculate(msg, x).await;
    result
}
```

read_from_network()和calculate()函数都是异步的,async_fn函数当然也是异步的.
当代码执行到read_from_network().await里面的时候,发现里面异步操作还没有完成,
它会直接退出当前这个函数,把CPU让给其他任务执行,当这个数据从网上传输完成了,调度器
会再次调用这个函数,它会从上次中断的地方恢复执行.

```
async_fn f1(arg: u8) -> u8 {}
```

类似于:
fn f1(arg: u8) -> impl Future<Output=u8> {} 返回一个实现了Future trait的类型

上面的两种写法是一样的,凡是被async修饰的函数,返回的都是一个实现了Future trait
的类型. 由async修饰的闭包也是一样的.async代码块同样类似.
async关键字不仅对函数签名做了一个改变,而且还对函数体也自动做了一个包装,被async关键字包
起来的部分,会自动产生一个Generator,并把这个Generator包装成一个满足Future约束的结构体,
在函数体中,用户需要返回的是Future::Output.

```
macro_rules! await {
    ($e: expr) => {
        let mut pinned = $e; //当前Future

        // 构建一个PinMut<impl Future>
        let mut pinned = unsafe
        { $crate::mem::PinMut::new_uncheched(&mut
        pinned) };
        loop {
            // 这里是和生成Generator的时候一样的写法
            match $crate::future::poll_in_task_cx(&mut
            pinned) {
                $crate::task::Poll::Pending => yield,
                $crate::task::Poll::Ready(x) => break x,
            }
        }
    }
}
```

从上面的代码结合Generator的知识,我们知道,await一定只能出现在async函数或者代码块中出现的,所以
它实际上是被包在一个Generator里面的.
async生成了一个结构体__Anynmous,这个结构体里面应该保存了一些状态,它实现了Generator trait,同时
也实现了Future,按照上面的例子,应该是read_from_network和calculate函数都是Ready的时候返回Ready,
否则返回Pending.

也就是说,满足Future的类型我们可以自己制作,不用添加async,但是await必须在async中使用,因为他们是一体
的Generator.

async负责Generator和Future的实现, await负责Generator的yield语法部分, 调度器负责resume的调用.
poll函数的调用位置:
        调度器resume的时候,调用poll.会发生yield.

我们在代码中并未看到resume的调用,只是看到了poll的调用,所以,resume的调用时在poll
调用的时候自动添加,这个不需要我们手动的添加resume的实现.为什么呢?因为Future这个
trait是编译器照顾的trait.
#[lang = "future_trait"]
pub trait Future {} 这是一个编译器照顾的类型.