RUST异步编程

协程编程目前已经逐步成为高并发，高性能编程的共识方案。语言或库对协程的支持使得程序员在编写多IO的程序时不必再去学习复杂的IO多路复用，以一种自然的直线思维方完成代码编写，但在运行时实际上使用IO多路复用的机制。协程机制将线程方案下的简单逻辑与IO多路复用的高性能巧妙融合在一起。
RUST对协程的支持采用的方案是语言仅仅提供最基础的语法async/await，Future trait。其余则留给框架编程人员自由发挥。

RUST的协程

RUST的协程的一个举例如下：

```
use tokio::net::TcpListener;
use tokio::io::{AsyncReadExt, AsyncWriteExt};

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let listener = TcpListener::bind("127.0.0.1:8080").await?;

    loop {
        let (mut socket, _) = listener.accept().await?;

        //此处类似go的协程，对于每一个接收到的tcp连接
        //都可以spawn一个tokio的task完成对其的处理
        //此处的tokio的task
        tokio::spawn(async move {
            let mut buf = [0; 1024];


            loop {
                let n = match socket.read(&mut buf).await {
                    // socket closed
                    Ok(n) if n == 0 => return,
                    Ok(n) => n,
                    Err(e) => {
                        eprintln!("failed to read from socket; err = {:?}", e);
                        return;
                    }
                };

                // Write the data back
                if let Err(e) = socket.write_all(&buf[0..n]).await {
                    eprintln!("failed to write to socket; err = {:?}", e);
```

```
                    return;
                }
            }
        });
    }
}
```

如上代码，可以看到，针对每个tcp连接都形成一个tokio的协程任务，在此任务中的代码几乎与同步IO的代码相同，仅仅多了几个`.await`，这个代码形式相比于IO多路复用，显得简单，明了，程序员也不用再费力去学习IO多路复用的知识。

RUST协程支持的设计

理解RUST针对异步IO协程的标准库支持，需要从整个协程实现的需求去考虑，否则无法理解清晰。 在一个进程内实现协程，实际上与操作系统实现线程管理的理念是近似的，只是在一些方面对协程做出了限制和简化，以匹配其轻量级的需求。

1. 协程的设计必然需要一个代码执行流重入的解决方案。RUST协程采用可重入的函数+状态这种设计来做重入,具体如下例：

```
//协程中异步函数编译后代码的示意
//除了完全可重入的函数外，其他所有可能
//导致协程退出执行，等待调度的函数都会编译成
//如下示意的形式
fn poll(&self, cx: &Context) {
    match(self) {
        Start: {
            //这个区间的代码都是同步代码
            //不会出现协程退出函数，等待调度
            ...
            ...
        }
        //第一个异步函数做.await的调用位置
        Await1: {
            self.state = Await1;
            //这个区间的代码都是同步代码
            //不会出现协程退出函数，等待调度
            ...
            ...
        }
        //第二个异步函数做.await的调用位置
        Await2: {
            self.state = Await2;
            //这个区间的代码都是同步代码
            //不会出现协程退出函数，等待调度
            ...
            ...
```

```
        }
        ...
        ...
    }
}
```

显然，这样的函数不应该由程序员来实现，事实上，async定义的函数及block会自动的被编译器编译成如上所示的形式。对于async中的异步函数或块做.await调用会自动的生成AwaitN状态。程序重入时也只可能利用这些状态逐级的进入到async的poll函数调用链，直到最后进入可完全重入的，无状态的poll函数(一般就是async实现的对系统调用的异步化封装)，这样就在用户空间实现了协程代码流的重入。

2. 协程的设计必然需要一个协程调度器，根据调度器设计实现的协程任务类型结构、集合类型机构及函数、方法。
3. 针对所有协程相关的操作系统资源必须建立相应的类型结构封装，类型结构应缓存在等待此资源的协程信息及操作信息，一般用多路IO复用侦听这些资源的事件，当事件发生时，如果此事件导致协程满足继续运行条件，则向调度器发送唤醒协程通知
4. 调度器不停的轮询或等待协程唤醒事件，调用协程的入口poll函数执行协程

RUST在标准库中没有实现统一的异步框架，而只是实现了最基础的支持：

1. `Future` trait，定义了异步代码的编译后的对外形态
2. `async` 自动实现了Future trait的类型，并将async block, async fn编译成了适合重入poll方法
3. `.await` 负责在async内部对下一级的async block或async fn的调用，帮助编译器生成具体的Future类型的状态
4. `Poll<T>, Ready<T>, Waker/Context<'a>/RawWaker` 是Future trait的支持结构，但也抽象了唤醒协程所使用的基本机制。

Future trait 分析

Future的定义如下:

```
pub trait Future {
    /// 是后继poll方法返回的类型，
    /// 对于async block，编译器会根据表达式的值来确定Output
    /// 对于async fn，Output即函数返回值
    type Output;

    //实现Future的类型结构会保存重入后正确执行的执行状态
    //在poll中如果返回Poll::Pending，会对类型结构的状态做正确的赋值
    //这样，再进入poll后，就可以根据self的状态及附带的参数回到上一次代码执行的位置
    //此处，poll使用Pin是编译器内部要求，大致上，self由于要在中断时保存执行状态，一定
需要定义一些
    //内部变量，而有些内部变量很可能会引用其他的内部变量，这就必须要用Pin来防止self内
```

```
存发生移动
    //Context实际存放唤醒调用本Future的协程的所有信息及操作。当poll阻塞时，可以将这个
信息
    //缓存到资源类型结构里，以便资源满足时对协程进行唤醒
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}

//Future的支持类型结构
pub enum Poll<T> {
    //表示轮询结束，并返回轮询结构变量
    Ready(T),

    //轮询不满足条件，需要暂时中止
    //条件满足时再次轮询
    Pending,
}
```

对于Future的支持结构 Context的分析如下：

```
//Context由调度器实现设计，在调度协程的根Future根据该协程的信息及调度
//需求生成，然后传入Future的poll函数，并且传入后继的每一个Future的poll
//函数，poll函数会从传入的cx获取唤醒本协程的实体，将之缓存到导致协程挂起
//的系统资源结构体中。以便系统资源事件处理时，利用其唤醒协程。
pub struct Context<'a> {
    //waker用于作为唤醒协程，会被clone后
    //缓存于导致协程挂起的资源的类型结构体里面
    //当协程运行条件满足时，被用于唤醒协程
    waker: &'a Waker,
    // Ensure we future-proof against variance changes by forcing
    // the lifetime to be invariant (argument-position lifetimes
    // are contravariant while return-position lifetimes are
    // covariant).
    _marker: PhantomData<fn(&'a ()) -> &'a ()>,
}

//Context的支持结构
pub struct Waker {
    waker: RawWaker,
}

//RawWaker保存了与某一协程调度相关的信息及唤醒协程需要的函数指针列表
//RawWaker实际上完全是一个C语言习惯的类型定义，由此可见，RUST语言类型
//实际上完全兼容C语言
pub struct RawWaker {
    /// 通过data，可以获得协程的信息，调度器唤醒协程需要的其他信息
    /// 可以认为这个data类似于C语言的 void*
    data: *const (),
    /// waker实现的函数列表，可以认为这个函数列表类似一个trait
    /// 这里没有办法采用trait的机制，因此使用了函数指针来实现接口
    vtable: &'static RawWakerVTable,
```

```rust
}

impl RawWaker {
    //由调度器根据自身设计的需要创建Waker
    pub const fn new(data: *const (), vtable: &'static RawWakerVTable) ->
RawWaker {
        RawWaker { data, vtable }
    }

    pub fn data(&self) -> *const () {
        self.data
    }

    pub fn vtable(&self) -> &'static RawWakerVTable {
        self.vtable
    }
}

//唤醒协程需要的接口函数列表
pub struct RawWakerVTable {
    //参数即RawWaker中的data
    //由调度器实现如何根据已有的数据复制一个RawWaker
    clone: unsafe fn(*const ()) -> RawWaker,

    //唤醒协程，可以消费掉传入的指针
    wake: unsafe fn(*const ()),

    //唤醒协程，不能消费传入的指针
    wake_by_ref: unsafe fn(*const ()),

    //释放传入的指针
    drop: unsafe fn(*const ()),
}

//创建一个函数指针列表
impl RawWakerVTable {
    pub const fn new(
        clone: unsafe fn(*const ()) -> RawWaker,
        wake: unsafe fn(*const ()),
        wake_by_ref: unsafe fn(*const ()),
        drop: unsafe fn(*const ()),
    ) -> Self {
        Self { clone, wake, wake_by_ref, drop }
    }
}

impl<'a> Context<'a> {
    //创建Context
    pub fn from_waker(waker: &'a Waker) -> Self {
        Context { waker, _marker: PhantomData }
    }
```

```
    pub fn waker(&self) -> &'a Waker {
        &self.waker
    }
}

impl Waker {
    pub fn wake(self) {
        let wake = self.waker.vtable.wake;
        let data = self.waker.data;

        //data会被后继的wake函数释放
        //不能再调用self的drop，会导致重复释放
        crate::mem::forget(self);

        unsafe { (wake)(data) };
    }

    pub fn wake_by_ref(&self) {
        unsafe { (self.waker.vtable.wake_by_ref)(self.waker.data) }
    }

    //如果两者是对同一个协程的唤醒，则相等
    pub fn will_wake(&self, other: &Waker) -> bool {
        self.waker == other.waker
    }

    pub unsafe fn from_raw(waker: RawWaker) -> Waker {
        Waker { waker }
    }

    pub fn as_raw(&self) -> &RawWaker {
        &self.waker
    }
}

impl Clone for Waker {
    #[inline]
    fn clone(&self) -> Self {
        Waker {
            //依赖于传入的clone函数完成
            waker: unsafe { (self.waker.vtable.clone)(self.waker.data) },
        }
    }
}

impl Drop for Waker {
    fn drop(&mut self) {
        //需要释放内部的data
        unsafe { (self.waker.vtable.drop)(self.waker.data) }
```

```
    }
}
```

Future trait的实现有两种方式： 一种是在代码中明确实现，这种情况一般发生于调用系统调用时，此时Future中的poll方法是没有状态的，可以反复重入。例如tokio中如下实现：

```
//ScheduledIo是IO资源的类型结构
impl ScheduledIo {
    ...

    //此函数由Future的poll函数调用，并传入cx
    //此函数是没有状态的，
    //连续执行并不会对执行流造成影响
    pub(super) fn poll_readiness(
        &self,
        cx: &mut Context<'_>,
        direction: Direction,
    ) -> Poll<ReadyEvent> {
        let curr = self.readiness.load(Acquire);

        let ready = direction.mask() &
Ready::from_usize(READINESS.unpack(curr));

        if ready.is_empty() {
            //这里waiters保存了cx的waker
            let mut waiters = self.waiters.lock();
            let slot = match direction {
                Direction::Read => &mut waiters.reader,
                Direction::Write => &mut waiters.writer,
            };

            //从cx中复制一个waker，放入self.waiters中
            match slot {
                Some(existing) => {
                    if !existing.will_wake(cx.waker()) {
                        *existing = cx.waker().clone();
                    }
                }
                None => {
                    *slot = Some(cx.waker().clone());
                }
            }

            let curr = self.readiness.load(Acquire);
            let ready = direction.mask() &
Ready::from_usize(READINESS.unpack(curr));

            if waiters.is_shutdown {
                Poll::Ready(ReadyEvent {
                    tick: TICK.unpack(curr) as u8,
```

```
                    ready: direction.mask(),
                })
            } else if ready.is_empty() {
                Poll::Pending
            } else {
                Poll::Ready(ReadyEvent {
                    tick: TICK.unpack(curr) as u8,
                    ready,
                })
            }
        } else {
            Poll::Ready(ReadyEvent {
                tick: TICK.unpack(curr) as u8,
                ready,
            })
        }
    }
    ...
}
```

另一种是通过async语法，由编译器自动实现。编译器以async包含的代码为基础生成Future的poll函数及实现Future trait的状态机类型结构。这一个课题官方的异步编程手册及course.rs的异步教程中有大量内容，请大家参考，本文不再赘述。

RUST的IO多路复用

协程的基础仍然是IO多路复用，因此本章给出一个简略的说明。
IO多路复用的基础在于操作系统提供的支持IO多路复用的系统调用，在linux系统即为select, poll, epoll等系统调用，目前一般使用epoll。IO多路复用的思想实际很简单，linux中的IO事件即fd事件，fd事件有读、写及异常，分别代表fd有内容可读，fd可写，fd异常。在有事件发生时，对fd可以进行对应的操作。
IO多路复用的另一个基础是所有的fd可以设置为非阻塞状态，即在读/写如果没有达到期望时(读入要求的字节数目或写出要求的字节数目)，不会等待，而是直接返回一个错误标志。 IO多路复用程序一般针对多个fd注册需要操作的多个fd事件及超时时间，然后用epoll调用阻塞等待这些事件。epoll返回时，调用程序会轮询所有的IO是否发生注册的事件及超时事件，如果事件发生则进行相应处理。所有事件处理完毕后根据需求进行下一次事件注册，然后再次调用epoll。如此可以用一个线程处理所有的IO，减少了多线程导致的线程切换开销，规避了不同线程操作临界区时的复杂的锁操作。

以下代码出自mio库。

```
//路径: src/sys/unix/selector.rs
//多路复用的核心结构，对于linux就是一个fd
pub struct Selector {
    ep: RawFd,
}
```

```rust
impl Selector {
    //创建一个多路复用的核心文件
    pub fn new() -> io::Result<Selector> {
        //执行exec是需要主动关闭此文件
        let flag = libc::EPOLL_CLOEXEC;

        //详细请参考epoll的相关指南，此系统调用生成了一个epoll的fd
        syscall!(epoll_create1(flag)).map(|ep| Selector {
            ep,
        })
    }

    //做多路复用的阻塞调用，等待多个IO事件，并给定等待的超时时间
    pub fn select(&self, events: &mut Events, timeout: Option<Duration>) ->
io::Result<()> {
        const MAX_SAFE_TIMEOUT: u128 = libc::c_int::max_value() as u128;

        let timeout = timeout
            .map(|to| cmp::min(to.as_millis(), MAX_SAFE_TIMEOUT) as
libc::c_int)
            .unwrap_or(-1);

        //清除所有的事件，具体见线面Events的分析
        events.clear();
        syscall!(epoll_wait(
            self.ep,
            //调用返回时操作系统会填充此结构
            events.as_mut_ptr(),
            //最多一次接收的事件数目
            events.capacity() as i32,
            timeout,
        ))
        .map(|n_events| {
            //设置events为正确长度，此处重要，因为操作系统
            //不知道RUST的语法
            unsafe { events.set_len(n_events as usize) };
        })
    }

    //向多路复用增加一个等待事件
    pub fn register(&self, fd: RawFd, token: Token, interests: Interest) ->
io::Result<()> {
        //生成一个epoll_event的结构变量
        let mut event = libc::epoll_event {
            events: interests_to_epoll(interests),
            //用来做事件的唯一标识,此标识被设置进操作系统，
            //事件发生后，操作系统会返回此标识
            u64: usize::from(token) as u64,
        };
```

```
        //将事件注册到epoll等待的事件中去
        syscall!(epoll_ctl(self.ep, libc::EPOLL_CTL_ADD, fd, &mut
event)).map(|_| ())
    }
    ....
}


impl Drop for Selector {
    fn drop(&mut self) {
        if let Err(err) = syscall!(close(self.ep)) {
            error!("error closing epoll: {}", err);
        }
    }
}

//支持类型结构及函数、方法

//对于某些操作系统，多路复用的系统调用需要对每个事件设置唯一标识
//以便应用程序与系统能够彼此确定唯一的事件
pub struct Token(pub usize);

impl From<Token> for usize {
    fn from(val: Token) -> usize {
        val.0
    }
}

//用来表示对读/写/异常事件的兴趣
//此处用一个独立的数据结构是RUST处于安全考虑的习惯
pub struct Interest(NonZeroU8);

//读事件及写事件的位
const READABLE: u8 = 0b0_001;
const WRITABLE: u8 = 0b0_010;

impl Interest {
    /// 读事件
    pub const READABLE: Interest = Interest(unsafe {
NonZeroU8::new_unchecked(READABLE) });

    /// 写事件
    pub const WRITABLE: Interest = Interest(unsafe {
NonZeroU8::new_unchecked(WRITABLE) });

    //增加希望处理的事件
    pub const fn add(self, other: Interest) -> Interest {
        Interest(unsafe { NonZeroU8::new_unchecked(self.0.get() |
other.0.get()) })
    }
```

```
    //移除不希望处理的事件
    pub fn remove(self, other: Interest) -> Option<Interest> {
        NonZeroU8::new(self.0.get() & !other.0.get()).map(Interest)
    }

    //是否希望处理读
    pub const fn is_readable(self) -> bool {
        (self.0.get() & READABLE) != 0
    }

    //是否希望处理写
    pub const fn is_writable(self) -> bool {
        (self.0.get() & WRITABLE) != 0
    }

}

impl ops::BitOr for Interest {
    type Output = Self;

    fn bitor(self, other: Self) -> Self {
        self.add(other)
    }
}

impl ops::BitOrAssign for Interest {
    fn bitor_assign(&mut self, other: Self) {
        self.0 = (*self | other).0;
    }
}

//将Interest转化为epoll的对应事件标志设置
fn interests_to_epoll(interests: Interest) -> u32 {
    let mut kind = EPOLLET;

    if interests.is_readable() {
        kind = kind | EPOLLIN | EPOLLRDHUP;
    }

    if interests.is_writable() {
        kind |= EPOLLOUT;
    }

    kind as u32
}

//epoll返回后的事件集
pub type Event = libc::epoll_event;
pub type Events = Vec<Event>;

pub mod event {
```

```
    use std::fmt;

    use crate::sys::Event;
    use crate::Token;

    //由Event获得Token,用于比较确定唯一的事件
    //此token是应用在注册是设置进操作系统的
    pub fn token(event: &Event) -> Token {
        Token(event.u64 as usize)
    }

    //判断event是否是读事件
    pub fn is_readable(event: &Event) -> bool {
        (event.events as libc::c_int & libc::EPOLLIN) != 0
            || (event.events as libc::c_int & libc::EPOLLPRI) != 0
    }

    //判断event是否是写事件
    pub fn is_writable(event: &Event) -> bool {
        (event.events as libc::c_int & libc::EPOLLOUT) != 0
    }

    //判断event是否是异常事件
    pub fn is_error(event: &Event) -> bool {
        (event.events as libc::c_int & libc::EPOLLERR) != 0
    }

    //判断event是否是输入fd关闭事件
    pub fn is_read_closed(event: &Event) -> bool {
        event.events as libc::c_int & libc::EPOLLHUP != 0
            || (event.events as libc::c_int & libc::EPOLLIN != 0
                && event.events as libc::c_int & libc::EPOLLRDHUP != 0)
    }

    //判断event是否是输出fd关闭事件
    pub fn is_write_closed(event: &Event) -> bool {
        event.events as libc::c_int & libc::EPOLLHUP != 0
            || (event.events as libc::c_int & libc::EPOLLOUT != 0
                && event.events as libc::c_int & libc::EPOLLERR != 0)
            || event.events as libc::c_int == libc::EPOLLERR
    }

    //判断是否有优先级
    pub fn is_priority(event: &Event) -> bool {
        (event.events as libc::c_int & libc::EPOLLPRI) != 0
    }

}
```

以上的使用实例如下:

```
const MAXEVENTS = 32;
fn main() {
    let events = Events::with_capacity(MAXEVENTS);
    let pool = Selector::new()?;

    pool.register(stdin().as_raw_fd(), Token::from(0), Interest::READABLE)?
    pool.register(stdout().as_raw_fd(), Token::from(1), Interest::WRITABLE)?
    pool.register(stderr().as_raw_fd(), Token::from(3), Interest::WRITABLE)?

    loop {
        pool.select(&events, None)?

        for(event in events.iter()) {
            match event.token() {
                Token::from(0) => {...},
                Token::from(1) => {...},
                Token::from(2) => {...}
            }
        }
    }
}
```

通常的情况下，如果IO文件众多，则针对每种类型的文件，需要涉及更合理的类型结构及其函数及方法来进行实现。但整体上多路复用的架构就是如上所述。虽然偏底层，但从实例的代码来看，并不是多复杂的一个知识。
IO多路复用编程的最大问题实际上是对底层抽象不够，编程者仍然要处理大量的底层IO的细节。