200行代码讲透Rust Futures的问题

stevenbai.top/rust/futures_explained_in_200_lines_of_rust2

rust

2020/4/13

引言

在翻译完本文以后很快就有人反馈作者实现的runtime有问题,然后作者进行了修复,没想到发下了更大的问题. 当然最终解决了. 我们来看看过程,同时加深对Future的理解.

作者关于此问题的讨论过程见github issue.

本文是在有问题的版本基础之上撰写,最终修订后,正常的版本在这里.

问题1: Future返回Pending一定要有唤醒机制

将main函数改写如下:

```
fn main() {
    let start = Instant::now();
    let reactor = Reactor::new();
    let reactor = Arc::new(Mutex::new(reactor));
    let future1 = Task::new(reactor.clone(), 1, 1);
    let future2 = Task::new(reactor.clone(), 2, 2);

    let start1 = start.clone();
    let fut1 = async move {
        let val = future1.await;
        let dur = (Instant::now() - start1).as_secs_f32();
        println!("Future got {} at time: {:.2}.", val, dur);
    };

    let start2 = start.clone();
    let fut2 = async move {
        let val = future2.await;
        let dur = (Instant::now() - start2).as_secs_f32();
        println!("Future got {} at time: {:.2}.", val, dur);
    };

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

    println!("Running on first executor");
    let mainfut = match block_on(futures::future::select(
        Box::pin(mainfut),
        futures::future::ready(()),
    )) {
        futures::future::Either::Left(_) => unreachable!(),
        futures::future::Either::Right(((), mainfut)) => mainfut,
    };

    println!("Running on second executor");
    std::thread::spawn(move || {
        block_on(mainfut);
    })
    .join();

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

Rust

然后尝试运行,你会发现第40行的join无法返回,为什么呢?

我们来分析一下task1的运行过程: 1. select的那次poll,这次因为没有注册,所以通过channel发送 `Event::Timeout` 给Reactor,Reactor则会启动一个线程T1运行该任务. 2. select这个future会提前结束,并没有等待刚刚创建的线程T1结束 3. 再次创建一个新的线程T2来运行task1,这时候task1的状态是已经注册,所以它就直接返回pending了.

然后,就再也不能唤醒了, 为什么呢,因为task1返回了Pending,但是没有任何唤醒当前线程的机制T2,因为T1结束后唤醒的是T1,而不是T2.

修复Future唤醒问题

知道了问题所在,相应的也就容易得多. 这也是赖智超的博客中说明的问题.

这里转述如下: > 在Rust的异步模型中，有一个极其关键的细节容易被忽略：在多次调用Future::poll方法时，Executor传递进去的Waker有可能是不一样的，因此每次返回Pending之前都需要把之前保存的Waker更新。比如考虑一个带工作窃取的Executor，Future刚开始在T1线程执行，注册了Waker，由于T1负载过高，Future被T2线程窃取执行，这时如果不更新Waker，当Future真正ready的时候，调用老的Waker，通知的却是T1线程，导致真正拥有Future的T2丢失了通知，Future永远无法结束。

那么解决方法的关键就是每次返回Pending的时候,保证Waker唤醒的线程就是block_on所在的那个线程即可.

这主要涉及到三个地方,下面分别描述.

Reactor的改动1

Reactor要保存每个task id对应的Waker,这个和初次运行Event::Timeout传递进来的Waker可能不是同一个. 所以Reactor的声明如下:

```
struct Reactor {
    dispatcher: Sender<Event>,
    handle: Option<JoinHandle<()>>,
    readylist: Arc<Mutex<Vec<usize>>>,
    wakers: Arc<Mutex<HashMap<usize, Waker>>>,
}
#[derive(Debug)]
enum Event {
    Close,
    Timeout(u64, usize),
}
```

Rust

注意到区别: 1. 多了一个wakers字段,其中的key是任务id,value则是关联的waker 2. Event的Timeout不再传递Waker,只是传递任务id

Reactor的改动2

说白了就是需要将任务的id和Waker关联起来,这样真正运行任务的线程结束后,可以调用相应的wake函数.

```
    fn insert_or_swap(&mut self, id: usize, waker: Waker) {
        self.wakers
            .lock()
            .map(|mut wakers| {
                let _ = wakers.insert(id, waker);
            })
            .unwrap();
    }

    fn register(&mut self, duration: u64, waker: Waker, data: usize) {
        self.insert_or_swap(data, waker);
        self.dispatcher
            .send(Event::Timeout(duration, data))
            .unwrap();
    }
```

Rust

Task的poll实现

```
fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
        println!(
            "poll task {}, is_registered={},thread={:?}",
            self.id,
            self.is_registered,
            thread::current().id(),
        );
        let mut r = self.reactor.lock().unwrap();
        if r.is_ready(self.id) {
            Poll::Ready(self.id)
        } else if self.is_registered {
            r.insert_or_swap(self.id, cx.waker().clone());
            Poll::Pending
        } else {
            r.register(self.data, cx.waker().clone(), self.id);
            drop(r);
            self.is_registered = true;
            Poll::Pending
        }
    }
```

Rust

最关键的地方就是这里:

```
if self.is_registered {
            r.insert_or_swap(self.id, cx.waker().clone());
            Poll::Pending
        }
```

Rust

这样保证每次返回Pending,都是和当前的waker关联起来,因为这有可能发生变化.

休眠唤醒问题

如果我们仔细深入思考,这里其实还存在一个问题,就是当poll正在运行的时候被唤醒的问题. 也就是

```
r.register(self.data, cx.waker().clone(), self.id);
           drop(r);
           //--->这里有竞争冲突
           self.is_registered = true;
```

Rust

当然这里不是传统意义上的内存共享问题,更确切来说应该是死锁问题, 这里可能reactor中很快就已经结束了任务的执行,然后尝试去唤醒executor线程,但是这时候实际上executor线程正在运行着. 然后poll返回pending,executor线程进入休眠.

> 如果你用代码实验,就会发现不会发生死锁. 这是因为thread::park文当中写明了,先unpark后park,这时候park会立即返回. 但是如果你用其他导致线程休眠唤醒的方式就会出问题. 这里说的是这种情况应该避免.

潜在死锁问题的解决办法

知道了问题所在,解决起来就会容易. 实际上并发问题,难的是找到问题,理解问题. 大多数时候解决起来反而容易.

问题的本质是: 我们在executor在执行poll的过程,可能会被wake,但是这时候执行线程还没有休眠.

解决办法则是: 1. reactor线程唤醒的时候要持有reactor锁.

```
 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_millis(duration));
                            let reactor = reactor.upgrade().unwrap();
                            reactor.lock().map(|mut r| r.wake(id)).unwrap();
                            println!("wake {}", id);
                        });
                        handles.push(event_handle);
                    }
                }
            }
            handles
                .into_iter()
                .for_each(|handle| handle.join().unwrap());
        });
        reactor.lock().map(|mut r| r.handle = Some(handle)).unwrap();
        reactor
    }
```

Rust

1. poll的过程中也要持有reactor的锁.

```rust
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) {
        println!("POLL: TASK {} 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) {
        println!("POLL: REPLACED WAKER FOR TASK: {}", self.id);
        r.tasks
            .insert(self.id, TaskState::NotReady(cx.waker().clone()));
        Poll::Pending
    } else {
        println!(
            "POLL: REGISTERED TASK: {}, WAKER: {:?}",
            self.id,
            cx.waker()
        );
        r.register(self.data, cx.waker().clone(), self.id);
        Poll::Pending
    }
}
}
```

Rust

但是我不认为这个完整解决了问题,还是有可能发生唤醒发生在休眠之前的问题.

更好的方式是负责休眠/唤醒顺序的是executor,而不是具体的Future. 比如 https://github.com/stjepang/byo-executor/blob/master/examples/v1.rs.

相关源码完整版

问题1的完整代码

```
use futures::future::select;
// from https://cfsamson.github.io/books-futures-explained/8_finished_example.html

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

fn main() {
    let start = Instant::now();
    let reactor = Reactor::new();
    let reactor = Arc::new(Mutex::new(reactor));
    let future1 = Task::new(reactor.clone(), 1, 1);
    let future2 = Task::new(reactor.clone(), 2, 2);

    let start1 = start.clone();
    let fut1 = async move {
        let val = future1.await;
        let dur = (Instant::now() - start1).as_secs_f32();
        println!("Future got {} at time: {:.2}.", val, dur);
    };

    let start2 = start.clone();
    let fut2 = async move {
        let val = future2.await;
        let dur = (Instant::now() - start2).as_secs_f32();
        println!("Future got {} at time: {:.2}.", val, dur);
    };

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

    println!("Running on first executor");
    let mainfut = match block_on(futures::future::select(
        Box::pin(mainfut),
        futures::future::ready(()),
    )) {
        futures::future::Either::Left(_) => unreachable!(),
        futures::future::Either::Right(((), mainfut)) => mainfut,
    };

    println!("Running on second executor");
    std::thread::spawn(move || {
        block_on(mainfut);
    })
    .join();
    reactor.lock().map(|mut r| r.close()).unwrap();
}

// ============================= EXECUTOR ====================================
fn block_on<F: Future>(mut future: F) -> F::Output {
    let mywaker = Arc::new(MyWaker {
        thread: thread::current(),
    });
    let waker = waker_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) };
    let val = loop {
        match Future::poll(future.as_mut(), &mut cx) {
            Poll::Ready(val) => break val,
            Poll::Pending => thread::park(),
        };
    };
    val
}

// ====================== FUTURE IMPLEMENTATION ==============================
#[derive(Clone)]
struct MyWaker {
    thread: thread::Thread,
}

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

fn mywaker_wake(s: &MyWaker) {
    let waker_ptr: *const MyWaker = s;
    let waker_arc = unsafe { Arc::from_raw(waker_ptr) };
    println!("wake thread {:?}", waker_arc.thread.id());
    waker_arc.thread.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 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<Reactor>>, data: u64, id: usize) -> Self {
        Task {
            id,
            reactor,
            data,
            is_registered: false,
        }
    }
}

impl Future for Task {
    type Output = usize;
    fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>
{
        println!(
            "poll task {}, is_registered={},thread={:?}",
            self.id,
            self.is_registered,
            thread::current().id(),
        );
        let mut r = self.reactor.lock().unwrap();
        if r.is_ready(self.id) {
            Poll::Ready(self.id)
        } else if self.is_registered {
            Poll::Pending
        } else {
            r.register(self.data, cx.waker().clone(), self.id);
            drop(r);
            self.is_registered = true;
            Poll::Pending
        }
    }
}

// =============================== REACTOR ===================================
struct Reactor {
    dispatcher: Sender<Event>,
    handle: Option<JoinHandle<()>>,
    readylist: Arc<Mutex<Vec<usize>>>,
}
#[derive(Debug)]
enum Event {
    Close,
    Timeout(Waker, u64, usize),
}
```

```
impl Reactor {
    fn new() -> Self {
        let (tx, rx) = channel::<Event>();
        let readylist = Arc::new(Mutex::new(vec![]));
        let rl_clone = readylist.clone();
        let mut handles = vec![];
        let handle = thread::spawn(move || {
            // This simulates some I/O resource
            for event in rx {
                println!("REACTOR: {:?}", event);
                let rl_clone = rl_clone.clone();
                match event {
                    Event::Close => break,
                    Event::Timeout(waker, duration, id) => {
                        let event_handle = thread::spawn(move || {
                            thread::sleep(Duration::from_secs(duration));
                            rl_clone.lock().map(|mut rl| rl.push(id)).unwrap();
                            waker.wake();
                        });

                        handles.push(event_handle);
                    }
                }
            }

            for handle in handles {
                handle.join().unwrap();
            }
        });

        Reactor {
            readylist,
            dispatcher: tx,
            handle: Some(handle),
        }
    }

    fn register(&mut self, duration: u64, waker: Waker, data: usize) {
        self.dispatcher
            .send(Event::Timeout(waker, duration, data))
            .unwrap();
    }

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

    fn is_ready(&self, id_to_check: usize) -> bool {
        self.readylist
            .lock()
            .map(|rl| rl.iter().any(|id| *id == id_to_check))
            .unwrap()
    }
}
```

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

Rust

修复Future唤醒问题的完整源码

```
use futures::future::select;
// from https://cfsamson.github.io/books-futures-explained/8_finished_example.html

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

fn main() {
    let start = Instant::now();
    let reactor = Reactor::new();
    let reactor = Arc::new(Mutex::new(reactor));
    let future1 = Task::new(reactor.clone(), 1, 1);
    let future2 = Task::new(reactor.clone(), 2, 2);

    let start1 = start.clone();
    let fut1 = async move {
        let val = future1.await;
        let dur = (Instant::now() - start1).as_secs_f32();
        println!("Future got {} at time: {:.2}.", val, dur);
    };

    let start2 = start.clone();
    let fut2 = async move {
        let val = future2.await;
        let dur = (Instant::now() - start2).as_secs_f32();
        println!("Future got {} at time: {:.2}.", val, dur);
    };

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

    println!("Running on first executor");
    let mainfut = match block_on(futures::future::select(
        Box::pin(mainfut),
        futures::future::ready(()),
    )) {
        futures::future::Either::Left(_) => unreachable!(),
        futures::future::Either::Right(((), mainfut)) => mainfut,
    };

    println!("Running on second executor");
    std::thread::spawn(move || {
        block_on(mainfut);
    })
    .join();

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

// ============================= EXECUTOR ====================================
fn block_on<F: Future>(mut future: F) -> F::Output {
    let mywaker = Arc::new(MyWaker {
        thread: thread::current(),
    });
    let waker = waker_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) };
    let val = loop {
        match Future::poll(future.as_mut(), &mut cx) {
            Poll::Ready(val) => break val,
            Poll::Pending => thread::park(),
        };
    };
    val
}

// ====================== FUTURE IMPLEMENTATION ==============================
#[derive(Clone)]
struct MyWaker {
    thread: thread::Thread,
}

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

fn mywaker_wake(s: &MyWaker) {
    let waker_ptr: *const MyWaker = s;
    let waker_arc = unsafe { Arc::from_raw(waker_ptr) };
    println!("wake thread {:?}", waker_arc.thread.id());
    waker_arc.thread.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 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<Reactor>>, data: u64, id: usize) -> Self {
        Task {
            id,
            reactor,
            data,
            is_registered: false,
        }
    }
}

impl Future for Task {
    type Output = usize;
    fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>
{
        println!(
            "poll task {}, is_registered={},thread={:?}",
            self.id,
            self.is_registered,
            thread::current().id(),
        );
        let mut r = self.reactor.lock().unwrap();
        if r.is_ready(self.id) {
            Poll::Ready(self.id)
        } else if self.is_registered {
            r.insert_or_swap(self.id, cx.waker().clone());
            Poll::Pending
        } else {
            r.register(self.data, cx.waker().clone(), self.id);
            drop(r);
            self.is_registered = true;
            Poll::Pending
        }
    }
}

// =============================== REACTOR ===================================
struct Reactor {
    dispatcher: Sender<Event>,
    handle: Option<JoinHandle<()>>,
    readylist: Arc<Mutex<Vec<usize>>>,
    wakers: Arc<Mutex<HashMap<usize, Waker>>>,
}
#[derive(Debug)]
enum Event {
    Close,
    Timeout(u64, usize),
}

impl Reactor {
    fn new() -> Self {
        let (tx, rx) = channel::<Event>();
        let readylist = Arc::new(Mutex::new(vec![]));
        let wakers: Arc<Mutex<HashMap<usize, Waker>>> =
Arc::new(Mutex::new(HashMap::new()));
        let rl_clone = readylist.clone();
        let wakers_clone = wakers.clone();
        let mut handles = vec![];
        let handle = thread::spawn(move || {
            // This simulates some I/O resource
            for event in rx {
                println!("REACTOR: {:?}", event);
                let rl_clone = rl_clone.clone();
                let wakers_clone = wakers_clone.clone();
                match event {
                    Event::Close => break,
                    Event::Timeout(duration, id) => {
                        let event_handle = thread::spawn(move || {
                            thread::sleep(Duration::from_secs(duration));
                            rl_clone.lock().map(|mut rl| rl.push(id)).unwrap();
                            wakers_clone
                                .clone()
                                .lock()
                                .map(|mut wakers| wakers.remove(&id).map(|w|
w.wake()))
                                .unwrap();
                        });

                        handles.push(event_handle);
                    }
                }
            }

            for handle in handles {
                handle.join().unwrap();
            }
        });

        Reactor {
            readylist,
            dispatcher: tx,
            handle: Some(handle),
            wakers,
        }
    }
    fn insert_or_swap(&mut self, id: usize, waker: Waker) {
        self.wakers
            .lock()
            .map(|mut wakers| {
                let _ = wakers.insert(id, waker);
            })
            .unwrap();
    }

    fn register(&mut self, duration: u64, waker: Waker, data: usize) {
        self.insert_or_swap(data, waker);
        self.dispatcher
            .send(Event::Timeout(duration, data))
            .unwrap();
    }

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

    fn is_ready(&self, id_to_check: usize) -> bool {
        self.readylist
            .lock()
            .map(|rl| rl.iter().any(|id| *id == id_to_check))
            .unwrap()
    }
}

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

Rust

完整的poll时wake问题

```rust
fn main() {
    let start = Instant::now();
    let reactor = Reactor::new();
    let future1 = Task::new(reactor.clone(), 1, 1);
    let future2 = Task::new(reactor.clone(), 2, 2);

    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);
    };

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

    block_on(mainfut);
    reactor.lock().map(|mut r| r.close()).unwrap();
}
use std::{
    collections::HashMap,
    future::Future,
    mem,
    pin::Pin,
    sync::{
        mpsc::{channel, Sender},
        Arc, Mutex,
    },
    task::{Context, Poll, RawWaker, RawWakerVTable, Waker},
    thread::{self, JoinHandle},
    time::{Duration, Instant},
};

// ============================= EXECUTOR ====================================
fn block_on<F: Future>(mut future: F) -> F::Output {
    let mywaker = Arc::new(MyWaker {
        thread: thread::current(),
    });
    let waker = waker_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) };
    let val = loop {
        match Future::poll(future.as_mut(), &mut cx) {
            Poll::Ready(val) => break val,
            Poll::Pending => {
                thread::sleep(std::time::Duration::from_millis(100));
                println!("executor parked");
                thread::park()
            }
        };
    };
    val
}

// ===================== FUTURE IMPLEMENTATION ==============================
#[derive(Clone)]
struct MyWaker {
    thread: thread::Thread,
}

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

fn mywaker_wake(s: &MyWaker) {
    let waker_ptr: *const MyWaker = s;
    let waker_arc = unsafe { Arc::from_raw(waker_ptr) };
    waker_arc.thread.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 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 }
    }
}

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) {
            println!("POLL: TASK {} 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) {
            println!("POLL: REPLACED WAKER FOR TASK: {}", self.id);
            r.tasks
                .insert(self.id, TaskState::NotReady(cx.waker().clone()));
            Poll::Pending
        } else {
            println!(
                "POLL: REGISTERED TASK: {}, WAKER: {:?}",
                self.id,
                cx.waker()
            );
            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_millis(duration));
                            let reactor = reactor.upgrade().unwrap();
                            reactor.lock().map(|mut r| r.wake(id)).unwrap();
                            println!("wake {}", id);
                        });
                        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) {
        self.tasks
            .get_mut(&id)
            .map(|state| match mem::replace(state, TaskState::Ready) {
                TaskState::NotReady(waker) => waker.wake(),
                TaskState::Finished => panic!("Called 'wake' twice on task: {}",
id),
                _ => unreachable!(),
            })
            .unwrap();
    }

    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

转载说明