Coroutine Intro with Rust

Zero-cost abstraction of async framework

- By Amanda Tian
Rust Language is ...

- a general purpose language for both system programming and applications, originated from Mozilla in 2006

- **high performance** and memory efficient without runtime or GC

- **Memory and thread safety** with type system and ownership model at compile time

- **Linux 6.1 officially adds support for Rust in kernel**
Agenda

- Asynchronous programming basics
- Rust async mechanism
- Runtime & task scheduling
- About stackful coroutine
Part 1
Asynchronous programming basics
Concurrency vs. Parallelism

Parallel: **doing** a lot of things at the same time

Concurrency: **dealing** a lot of things at the same time

Asynchronous: describe a language feature to enable concurrent or parallel programming
I/O types - synchronous blocking

Traditional OS threads with one thread per task blocks:

```rust
fn read_parallel() {
    let jh_1 = thread::spawn(|| read("file_A"));
    let jh_2 = thread::spawn(|| read("file_B"));
    jh_1.join();
    jh_2.join();
}
```

Pros:
- Simple, straightforward logic
- Free use with kernel's management

Cons:
- Limited number of tasks with large stack mem
- Context switch is bottleneck in high concurrency
I/O types - asynchronous non-blocking

Event driven + I/O multiplexing

Pros:
- No context switch, relative low mem cost

Cons:
- Callback hell, nested callback chains hard to maintain and understand
I/O types - synchronous non-blocking

Coroutines:
- Pausable cooperative multitask able to yield and resume
- Write async code just in synchronous manner

```rust
async fn read_parallel_async() {
    let fut_1 = read_file_async("file_A");
    let fut_2 = read_file_async("file_B");

    fut_1.await;
    fut_2.await;
}
```
“Coroutines are computer program components that generalize subroutines for non-preemptive multitasking, by allowing execution to be suspended and resumed.”

- Wikipedia

Variant of functions enable concurrency via cooperative multitasking
Preemptive vs. Cooperative Multitasking

**Preemptive:** system forcibly suspend running task and switch to another

**Cooperative:** task voluntarily yield control periodically or idle or blocked
Part 2
Rust async mechanism
Rust async mechanism - overview

Rust compiler:
- `async` keyword, `.await` syntax

Rust std:
- Basic `Future` trait for pausable task
- `Waker` type to wake up a task

Rust async runtime

```rust
def main() {
    let sum_fut = sum();
    let jh = tokio::spawn(sum_fut);
    block_on(jh);
}

#[inline(never)]
def get_val() -> impl Future<Output::usize> {
    // do the task asynchronously
    async {1}
}

def fn sum() -> usize {
    get_val().await + 1
}
Rust async mechanism - async/await

std::ops::generator example:

```rust
fn main() {
    let generator = || {
        let mut val = 1;
        yield val;
        val += 1;
        yield val;
        val += 1;
        return val;
    };
    assert_eq!(generator.resume(), GeneratorState::Yielded(1));
    assert_eq!(generator.resume(), GeneratorState::Yielded(2));
    assert_eq!(generator.resume(), GeneratorState::Complete(3));
}
```

async/await => generator
Rust async mechanism - async/await

async/await => generator => **statemachine** => impl Future
Rust async mechanism - Future trait

A future represents an asynchronous computation obtained by use of async.

```
pub trait Future {
    /// The type of value produced on completion.
    type Output;

    /// Attempt to resolve the future to a final value, registering
    /// the current task for wakeup if the value is not yet available.
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}

pub enum Poll<T> {  
    /// Represents that a value is immediately ready.
    Ready(T),

    /// represents a value is not ready yet
    Pending,
}
```

Future exposes `poll` method:
- Called by future to drive task execution
- Return `Pending` when blocked
- Return `Ready` with output when finished

Poll method defines the state machine of Future
Rust async mechanism - impl a Future

Futures implementation:
- Leaf future (I/O resource) usually by runtime
- Non-leaf future generated by compiler via async

Memory optimization:
- Zero-cost abstraction allow no heap allocation or dynamic dispatch
- Reuse memory for non-overlap variables
Rust async mechanism - Waker

```rust
pub struct RawWaker {
    data: *const (),
    /// Virtual function pointer table with customized behavior.
    vtable: &'static RawWakerVTable,
}

pub struct Waker {
    waker: RawWaker,
}

impl Waker {
    /// Wake up the task related to this `Waker`.
    pub fn wake(self) {
        let wake = self.waker.vtable.wake;
        let data = self.waker.data;

        unsafe { (wake)(data) };
    }
}

pub struct Context<'a> {
    waker: &'a Waker,
    // marker field could be ignored
    _marker: PhantomData<fn(&'a ()) -> &'a ()>,
}
```

Waker:
- std defined interface to wake up a suspended task when related I/O event ready
- Runtime creates and defines data, vtable, i.e. HOW to wake a task up
- Passed around wrapped in a Context
Part 3
Runtime and task scheduling
Runtime overview

What’s runtime?
- Rust std provides minimal primitive: Future trait, async/await for pausable async tasks
- Runtime act as execution context to drive the futures to completion

What runtime consists?
- **Executor**: a task scheduler usually with task queue
- **Reactor**: I/O driver backed by system event queue (mio crate over epoll/kqueue/IOCP)
- **I/O components**: non-blocking APIs interact with Reactor

Rust community runtimes crates:

<table>
<thead>
<tr>
<th>Runtime</th>
<th>All-time downloads (July 2022)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tokio</td>
<td>59,048,636</td>
<td>An event-driven, non-blocking I/O platform for writing asynchronous I/O backed applications.</td>
</tr>
<tr>
<td>async-std</td>
<td>8,002,852</td>
<td>Async version of the Rust standard library</td>
</tr>
<tr>
<td>smol</td>
<td>1,491,204</td>
<td>A small and fast async runtime</td>
</tr>
</tbody>
</table>
Tokio interfaces

- `#[tokio_main]`: annotate the main function as async
- `block_on`: runtime’s entry point, runs a future to completion
- `tokio::spawn`: spawn new future as Tasks, executed by runtime concurrently
- `JoinHandle`: handle to spawned task to retrieve output on Task finish
- `tokio::spawn_blocking`: runs blocking functions on executor, usually on a separate thread pool from non-blocking tasks
- `tokio::block_in_place`: spawn blocking task and turn current thread to blocking thread, move existing tasks to other worker threads
Task scheduling (single thread)

Executor and reactor form an event-loop, loosely decoupled by Future and Waker

Waker in tokio:
- A reference to the task itself
- Wake pushes task to the queue
Task scheduling (single thread)

I/O component:
1. Provides non-blocking API
2. Register I/O event fd to reactor, with correlated Waker

Executor:
1. Poll each task on queue as far as possible
2. Give control to reactor when idle

Reactor (underlying Mio - metal I/O):
1. Waiting for I/O event blockingly
2. Wake up the task with event ready
3. Give control back to executor
Task scheduling (multi-thread)

Global queue

Separate local queue

Work-stealing model

ref
Handling multiple tasks - join

tokio::join!():
Waits on multiple **concurrent** branches, returning when all branches complete.
Handling multiple tasks - join

tokio::select!():
Waits on multiple concurrent branches, returning when the first branch completes, cancelling the remaining branches.

```rust
#[tokio::main]
async fn main() {
    let (tx1, rx1) = oneshot::Receiver::<u32>();
    let (tx2, rx2) = oneshot::Receiver::<u32>();

    tokio::select!{
        _ = rx1.recv() => {},
        _ = rx2.recv() => {},
    }
}

pub struct SelectTask {
    rx1: oneshot::Receiver<u32>,
    rx2: oneshot::Receiver<u32>,
}

impl Future for SelectTask {
    type Output = u32;
    fn poll(mut self: Pin<&mut Self>, cx: &mut Context<Self>) -> Poll<Self::Output> {
        if let Poll::Ready(val) = Pin::new(&mut self.rx1).poll(cx) {
            println!("rx1 completed first with {:?}", val);
            return Poll::Ready(val);
        }

        if let Poll::Ready(val) = Pin::new(&mut self.rx2).poll(cx) {
            println!("rx2 completed first with {:?}", val);
            return Poll::Ready(val);
        }

        Poll::Pending
    }
}
```
Part 4
About stackful coroutine
More about coroutines...

Coroutine classification:
- asymmetric vs. symmetric
- first-class object vs. constrained construct
- stackful vs. stackless

Stackful coroutine:
- Future state stored as call stack, allocated on heap
- Task switched by context switch
- Also known as fibers, green threads, e.g. Goroutine
Hack with context switch

```rust
struct ThreadContext {
    rsp: u64,
}

fn main() {
    let mut ctx = ThreadContext::default();
    let mut stack = vec![0_u8; SSIZE as usize];

    unsafe {
        let stack_bottom = stack.as_mut_ptr().offset(SSIZE);
        let sb_aligned = (stack_bottom as usize & !15) as *mut u8;
        std::ptr::write(sb_aligned.offset(-16) as *mut u64, hello as u64);
        ctx.rsp = sb_aligned.offset(-16) as u64;
        gt_switch(&mut ctx);
    }
}

fn hello() -> ! {
    println!("I LOVE WAKING UP ON A NEW STACK!");
}

unsafe fn gt_switch(new: *const ThreadContext) {
    asm!(
        "mov rsp, [{0} + 0x00]",
        "ret",
        in(reg) new,
    );
}
```

---

**Figure 3.3: Stack Frame with Base Pointer**

<table>
<thead>
<tr>
<th>Position</th>
<th>Contents</th>
<th>Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>8h+16(%rbp)</td>
<td>memory argument eightbyte n</td>
<td>Previous</td>
</tr>
<tr>
<td>16(%rbp)</td>
<td>memory argument eightbyte 0</td>
<td></td>
</tr>
<tr>
<td>0(%rbp)</td>
<td>return address</td>
<td></td>
</tr>
<tr>
<td>-8(%rbp)</td>
<td>previous %rbp value</td>
<td></td>
</tr>
<tr>
<td>-8(%esp)</td>
<td>unspecified</td>
<td></td>
</tr>
<tr>
<td>-128(%esp)</td>
<td>variable size</td>
<td></td>
</tr>
<tr>
<td>-128(%esp)</td>
<td>red zone</td>
<td></td>
</tr>
</tbody>
</table>
Stackful coroutine - a toy

```
struct Thread {
    id: usize,
    stack: Vec<u8>,
    ctxx: ThreadContext,
    state: State,
}

// Registers %rbx,
// %rbp, and %r12-r15 are callee-save registers, meaning that they are saved across function
// calls. Register %rsp is used as the stack pointer, a pointer to the topmost element in the stack.
struct ThreadContext {
    rsp: u64,
    r15: u64,
    r14: u64,
    r13: u64,
    r12: u64,
    rbp: u64
}

impl Thread {
    pub fn new(id: usize) -> Self {
        Thread {
            id,
            stack: vec![0_u8, DEFAULT_STACK_SIZE as u8],
            ctx: ThreadContext::default(),
            state: State::Available
        }
    }
}
```

Thread:
- abstraction of coroutine holds its stack and context with register values
Stackful coroutine - a toy

```rust
code snippet
pub struct Runtime {
    threads: Vec<Thread>.,
    current: usize,
}

impl Runtime {
    pub fn run(&mut self) -> {
        while self.current != self.threads.len() {
            let mut pos = self.current;
            while self.threads[pos].state == State::Ready {
                if pos == self.threads.len() - 1 {
                    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 {
                let old = self.threads[old_pos].ctx;
                let new = self.threads[pos].ctx;
                asm("call switch", in("rdi") old, in("rsi") new, ctlobber_ab("C"));
            }
        }
    }

    pub fn spawn(&mut self, f: fn()) {
        let available = self.threads;
        let mut i = 0;
        let t = available[i].state == State::Available;
        let t_id = available[i].id;
        let t_size = available[i].size;
        unsafe {
            let s_ptr = available stack as mut ptr, offset as isize;
            let s_ptr = (s_ptr as usize & 15) as mut u64;
            std::ptr::write(s_ptr.offset(-16) as *mut u64, guard as u64);
            std::ptr::write(s_ptr.offset(-24) as *mut u64, skip as u64);
        }
    }
}
```

Runtime:
- API to spawn new threads
- main loop to trigger the execution of threads
- perform context switch when a thread is not Ready
Stackful coroutine - a toy

Context switch:
- Store the current registers to rdi (old thread)
- Load from rsi (new thread) to current registers
Stackful vs. stackless coroutine

Stackless coroutine:
- Lightweight with zero-cost abstraction backed with state machine
- No context switch on task scheduling

Stackful coroutine:
- With call stack stored, capable to yield at any time
- Allow preemptive scheduling on bad actors
- Memory cost on stack growth could be an issue
Last but not least...

Coroutine is powerful, but not suitable in any situations.

Good for:
- Obviously web servers
- UI (wait for user response while doing background tasks)
- Filesystems
- ...

Not best choice:
- CPU intensive computations
- Long running tasks without yielding
Q & A