程序设计训练之 Rust 编程语言

第七讲：I/O 与异步编程

韩文弢

清华大学计算机科学与技术系

2022 年 9 月 7 日

1

输入输出

用于输入输出的特型

```rust
pub trait Read {
    fn read(&mut self, buf: &mut [u8]) -> Result<usize>;
    // Other methods implemented in terms of read().
}

pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;
    // Other methods implemented in terms of write() and flush().
}
```

- 很多类型实现了标准 IO 特型：
  - `File`、`TcpStream`、`Vec<T>`、`&[u8]`
- 注意：返回类型是 `std::io::Result`，而不是 `std::Result`。
  - `type Result<T> = Result<T, std::io::Error>;`

std::io::Read

```rust
use std::io;
use std::io::prelude::*;
use std::fs::File;

let mut f = File::open("foo.txt")?;
let mut buffer = [0; 10];

// read up to 10 bytes
f.read(&mut buffer)?;
```

- `buffer` 是一个数组，因此可以读取的最大长度编码在类型信息中。
- `read` 成功时返回读取的字节数，失败时返回表示错误原因的 `Err`。
  - 返回值 `Ok(n)` 保证 `n <= buf.len()`。
  - 如果读取的来源是空，返回值可能为 `Ok(0)`。

读取方式

```rust
/// Required.
fn read(&mut self, buf: &mut [u8]) -> Result<usize>;

/// Reads to end of the Read object.
fn read_to_end(&mut self, buf: &mut Vec<u8>) -> Result<usize>

/// Reads to end of the Read object into a String.
fn read_to_string(&mut self, buf: &mut String) -> Result<usize>

/// Reads exactly the length of the buffer, or throws an error.
fn read_exact(&mut self, buf: &mut [u8]) -> Result<()>
```

- Read 提供了读取内容到不同缓冲区的方法。
  - 其他方法提供了调用 read 的默认实现。
- 注意不同的类型签名。

读取迭代器

```
fn bytes(self) -> Bytes<Self> where Self: Sized
```

- bytes 方法将 Read 按逐字节的方式转换成迭代器。
- 关联的 Item 是 Result<u8>。
  - 调用 next() 返回的是 Option<Result<u8>>。
  - 遇到 EOF 时会返回 None。

迭代器适配器

```
fn chain<R: Read>(self, next: R) -> Chain<Self, R>
    where Self: Sized
```

- `chain` 以第二个读取对象作为输入，返回的迭代器先迭代 `self`，再迭代 `next`。

```
fn take<R: Read>(self, limit: u64) -> Take<Self>
    where Self: Sized
```

- `take` 创建的迭代器的迭代范围是读取对象的前 `limit` 个字节。

std::io::Write

```rust
pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;
    // Other methods omitted.
}
```

- `Write` 特型有两个必须实现的方法 `write()` 和 `flush()`。
  - 其他方法有依赖于这两个方法的默认实现。
- `write`（试图）将 `buf` 里的内容写入写入对象，并返回写入（或加入队列）的字节数。
- `flush` 确保所有写入的数据都已经到达目标。
  - 写入可以用队列来优化行为。
  - 如果不是全部排队的数据都写入成功，会返回 `Err`。

写入操作

```rust
let mut buffer = File::create("foo.txt")?;

buffer.write("Hello, Ferris!")?;
```

- 写入方法

```rust
/// Attempts to write entire buffer into self.
fn write_all(&mut self, buf: &[u8]) -> Result<()> { ... }

/// Writes a formatted string into self.
/// Don't call this directly, use `write!` instead.
fn write_fmt(&mut self, fmt: Arguments) -> Result<()> { ... }
```

write!

- 在应用程序中直接使用写入对象会不太方便，尤其是对输出内容有格式要求时。
- `write!` 宏提供了在 `write_fmt` 之上用字符串格式化的输出方式。
  - 返回 `Result`。

```rust
let mut buf = try!(File::create("foo.txt"));

write!(buf, "Hello {}!", "Ferris").unwrap();
```

IO 的速度问题

- 对比于处理器和内存，IO 操作非常慢。
- 处理器内的操作一般是几个时钟周期，ns 级别。
- 内存操作一般是几百个时钟周期，亚 s 级别。
- IO 操作的延迟（从发出 IO 指令到收到数据）:
  - 机械硬盘是十 ms 级别。
  - 固态硬盘是亚 ms 级别。
- IO 操作与处理器和内存有 3-7 个数量级的差距。

IO 缓冲机制

- 应用程序在操作系统下执行时是受限的。
- 对磁盘的读取操作由操作系统完成（通过系统调用）。
  - 此时，需要将运行状态切换至内核态，由操作系统去处理读取操作，并返回给应用程序。
  - 相比普通的内存访问，这个过程非常慢。
- 如果频繁读取时每次都这样操作会导致性能非常差。
  - 使用缓冲机制来提高性能。
  - 每次将一大块内容读到缓冲区里，然后应用程序根据需要进行访问。
- 写入的时候情况类似。

BufReader

```rust
fn new(inner: R) -> BufReader<R>;
```

```rust
let mut f = File::open("foo.txt")?;
let buffered_reader = BufReader::new(f);
```

- BufReader 可以给任意的读取对象添加缓冲机制。
- BufReader 实现了 Read 特型，因此可以透明地使用。

BufReader

- BufReader 还实现了另外一个接口 BufRead。

```rust
pub trait BufRead: Read {
    fn fill_buf(&mut self) -> Result<&[u8]>;
    fn consume(&mut self, amt: usize);

    // Other optional methods omitted.
}
```

BufReader

- 由于 BufReader 提前读取了更多的数据，可以做一些有用的事情。
- 它定义了两个新的读取方法，可以一直读到遇到某个特定的字节。

```
fn read_until(&mut self, byte: u8, buf: &mut Vec<u8>)
    -> Result<usize> { ... }
fn read_line(&mut self, buf: &mut String)
    -> Result<usize> { ... }
```

BufReader

- 还定义了两种迭代器。

```
fn split(self, byte: u8)
    -> Split<Self> where Self: Sized { ... }
fn lines(self)
    -> Lines<Self> where Self: Sized { ... }
```

BufWriter

- BufWriter 包裹了写入对象。

```
let f = File::create("foo.txt")?;
let mut writer = BufWriter::new(f);
buffer.write(b"Hello world")?;
```

- BufWriter 没有像 BufReader 那样实现第二个接口。
- 它直接缓存所有的写入数据，在失效前把所有缓存的数据写出。

Stdin

```rust
let mut buffer = String::new();

io::stdin().read_line(&mut buffer)?;
```

- 这是一个典型的从标准输入（终端输入）读取数据的方法。
- `io::stdin()` 返回类型是 `Stdin` 结构体的值。
- `stdin` 直接实现了 `read_line` 方法，没有实现 `BufRead` 特型。

StdinLock

- 多个线程从标准输入读取数据是一个数据共享问题，需要加锁。
- 所有的 `read` 方法会在内部调用 `self.lock()`。
- 也可以创建 `StdinLock` 并显式调用 `lock()` 方法。

```rust
let lock: io::StdinLock = io::stdin().lock();
```

StdinLock

```rust
use std::io::{self, BufRead};

fn main() -> io::Result<()> {
    let mut buffer = String::new();
    let stdin = io::stdin(); // We get `Stdin` here.
    {
        let mut handle = stdin.lock(); // We get `StdinLock` here.
        handle.read_line(&mut buffer)?;
    } // `StdinLock` is dropped here.
    Ok(())
}
```

Stdout

- `Stdout` 是标准输出的接口。
- 实现了 `Write` 特型。
- 一般来说不会直接使用 `stdout`，而是用 `print!` 和 `println!`。
- 也可以通过 `Stdout::lock()` 来获得标准输出的锁。

特殊 IO 构件

- `repeat(byte: u8)`：无限产生某个指定字节的读取对象。
  - 总是会充满缓冲区。
- `sink()`：将数据丢弃的写入对象。
- `empty()`：读取操作时总是返回 `Ok(0)` 的读取对象。
- `copy(reader: &mut R, writer: &mut W) -> Result<u64>`
  - 将所有字节从读取对象拷贝到写入对象。

2

异步编程的概念

为什么要异步执行?

- 程序任务的两种类型:
  - CPU 密集型 (CPU-bound) 任务:文件压缩、视频编码、图形处理等
  - IO 密集型 (IO-bound) 任务:文件读写、网络请求处理等
- CPU 密集型任务可以利用多核(甚至是多 CPU、多机)来获得整体性能。
- IO 密集型任务中如何提高 CPU 的使用率?
  - 希望程序在发出 IO 请求后,在等待 IO 动作完成的过程中,能够利用 CPU 做其他任务。

同步和多线程执行的例子

- 假设有三个任务要执行：任务 1、任务 2、任务 3
  - 其中，任务 1 中有长时间的 IO 操作。

同步方式

- 使用一个线程，依次执行：
  1. 任务 1（中间会阻塞一段时间）
  2. 任务 2
  3. 任务 3

多线程方式

- 使用两个线程，其中线程 1 执行：
  1. 任务 1（中间会阻塞一段时间）
- 同时，线程 2 执行：
  1. 任务 2
  2. 任务 3

异步执行的例子

- 同样是之前的例子，使用异步方式来执行：

异步方式

- 使用一个线程，执行：
  1. 任务 1（发出 IO 指令）
  2. 任务 2（同时任务 1 的 IO 操作正在执行）
  3. 任务 1（获得 IO 结果并处理）
  4. 任务 3

Web 服务器的例子

- 同步实现的缺点：同一时刻只能处理一个请求。
- 多线程实现：针对每个请求，开启一个新的线程来处理。
  - 优点：提高性能，可同时处理多个请求。
  - 缺点：引入各种开销和并发的问题。
    - 开新线程代价高（用线程池）。
    - CPU 在处理 IO 时会处于空闲状态。
    - 执行顺序无法预测。
    - 对于全局状态会有共享和竞争的问题。
    - 死锁……

异步 Web 服务器

- 异步 Web 服务器给每个请求建立一个任务。
- 由异步运行时来调度任务，把当前可以执行的任务放到可用的 CPU 上执行。
- 优点：
  - 减小开销。
  - 充分利用 CPU 资源。
- 缺点：
  - 逻辑上不容易理解。
  - 编程复杂。
- 能否完全解决并发的问题?
  - 根据异步运行时的调度方式决定，是否使用多个 CPU 核。

3

Rust 的异步基础

异步编程的编程支持

- 实现异步编程需要编程语言能够支持：
  - 把一段代码变成异步任务。
  - 调度异步任务的执行。
- 上述两项工作，从语言设计的角度如何划分?

异步代码

- 使用 `async` 语法来创建异步代码：

```
async fn do_something() { /* ... */ }
```

- `async fn` 返回的结果是一个 `Future` 对象。
- 需要用执行器 (executor) 来执行 `Future` 对象。

执行异步代码

```rust
use futures::executor::block_on;

async fn hello_world() {
    println!("Hello, world!");
}

fn main() {
    let future = hello_world(); // Nothing is printed
    block_on(future); // `future` is run and "Hello, world!" is printed
}
```

异步调用的可能性

```rust
async fn learn_song() -> Song { /* ... */ }
async fn sing_song(song: Song) { /* ... */ }
async fn dance() { /* ... */ }

fn main() {
    let song = block_on(learn_song());
    block_on(sing_song(song));
    block_on(dance());
}
```

- 以上代码仍然是同步执行的。
- 需要一种手段，在异步代码中调用其他异步代码时，可以选择性地调度任务。

异步调用

```rust
async fn learn_and_sing() {
    let song = learn_song().await;
    sing_song(song).await;
}

async fn async_main() {
    let f1 = learn_and_sing();
    let f2 = dance();

    futures::join!(f1, f2);
}

fn main() {
    block_on(async_main());
}
```

异步调用的机制

- 在异步代码中，可以使用 `.await` 来等待另外一个 `Future` 对象完成。
- 不同于 `block_on`，`.await` 是异步等待，不阻塞当前线程，此时可以调度其他异步任务。
- 上述代码中，
  - `learn_song()` 和 `sing_song()` 必须按顺序发生。
  - 而 `dance()` 可以和 `learn_and_sing()` 同时发生。

async 和 .await

```rust
// `foo()` returns a type that implements `Future<Output = u8>`.
// `foo().await` will result in a value of type `u8`.
async fn foo() -> u8 { 5 }

fn bar() -> impl Future<Output = u8> {
    // This `async` block results in a type that implements
    // `Future<Output = u8>`.
    async {
        let x: u8 = foo().await;
        x + 5
    }
}
```

async

- async 有两种语法：
  - async fn 将一个函数转换为异步函数，返回 Future 对象。
  - async 放在代码段前面，将代码段转换为异步任务，也返回 Future 对象。
- 需要注意与 async 相关的所有权和生命周期的问题。

Future

```rust
trait Future {
    type Output;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Outpu
}

enum Poll<T> {
    Ready(T),
    Pending,
}
```

- Future 特型是 Rust 异步编程的核心。
- Future 对象是一项异步计算任务，能够返回一个结果。

Future 的基本原理

```rust
trait SimpleFuture {
    type Output;
    fn poll(&mut self, wake: fn()) -> Poll<Self::Output>;
}
```

- 以简化版的 SimpleFuture 为例，可以通过调用 poll() 方法来驱动任务前进。
  - 如果任务完成，则返回 Poll::Ready(result)。
  - 如果还无法完成，则返回 Poll::Pending，同时安排在取得进展时调用 wake() 函数。
- wake() 函数被调用时，执行器会再次 poll() 这个 Future 对象。
- 如果没有 wake() 函数这个机制，执行器只能盲目地轮询所有正在执行的 Future 对象。

多线程执行异步任务

- 当使用多线程执行器时，`Future` 对象可能会在线程之间转移。
  - 因此 `async` 中用到的变量必须是 `Send` 的。
- `std::sync::Mutex` 可能会引起死锁。
  - 使用 `futures::lock::Mutex`。

4

Rust 的异步生态

库的支持

- Rust 中，
  - `async` 和 `.await` 是语言支持的。
  - `Future` 在标准库中。
- 没有对异步任务进行调度执行的支持。

异步运行时

- Rust 将异步运行时交给第三方库来提供。
- 一般来说，异步运行时会包含：
  - 一个反应器 (reactor)，提供对外部事件的监听机制（如 IO、过程间通信、计时器等）。
  - 一个或多个执行器，处理任务的调度和执行。
- 运行时需要做的事情包括：
  - 维护当前正在运行和挂起的任务。
  - 查询任务的进展。
  - 唤醒任务，推动任务执行。

futures 箱

- futures 箱提供编写异步代码需要的特型和函数：
  - `Stream`、`Sink`、`AsyncRead`、`AsyncWrite` 等
- 最终可能会成为标准库的组成部分。
- 有执行器，但没有反应器，无法应对外部事件。

常用的异步运行时

- Tokio：一个比较流行的异步框架，支持 HTTP、gRPC 等。
- async-std：提供标准库中一些组件的异步实现版本。
- smol：一个轻量级的异步运行时，提供 `Async` 包裹特型来实现异步功能。
- fuchsia-async：Fuchsia OS 所使用的执行器。

Tokio 库

Tokio 是一个 Rust 的异步运行时，适合来写网络服务端代码，提供：

- 执行异步代码的多线程运行时。
- 标准库的异步版本。
- 由很多相关库组成的生态系统。

看一个用 Tokio 实现简化版键值数据库的例子。

Tokio：基本用法

```rust
use mini_redis::{client, Result};

#[tokio::main]
async fn main() -> Result<()> {
    // Open a connection to the mini-redis address.
    let mut client = client::connect("127.0.0.1:6379").await?;
    // Set the key "hello" with value "world"
    client.set("hello", "world".into()).await?;
    // Get key "hello"
    let result = client.get("hello").await?;
    println!("got value from the server; result={:?}", result);
    Ok(())
}
```

Tokio：产生任务

```rust
#[tokio::main]
async fn main() {
    // Bind the listener to the address
    let listener = TcpListener::bind("127.0.0.1:6379").await.unwrap();

    loop {
        // The second item contains the IP and port of the new connection.
        let (socket, _) = listener.accept().await.unwrap();
        process(socket).await;
    }
}
```

Tokio：实现任务

```rust
async fn process(socket: TcpStream) {
    // The `Connection` lets us read/write redis **frames** instead of
    // byte streams. The `Connection` type is defined by mini-redis.
    let mut connection = Connection::new(socket);

    if let Some(frame) = connection.read_frame().await.unwrap() {
        println!("GOT: {:?}", frame);

        // Respond with an error
        let response = Frame::Error("unimplemented".to_string());
        connection.write_frame(&response).await.unwrap();
    }
}
```

Tokio：共享状态

```rust
use bytes::Bytes;
use std::collections::HashMap;
use std::sync::{Arc, Mutex};

type Db = Arc<Mutex<HashMap<String, Bytes>>>;
```

Tokio：创建共享状态

```rust
use tokio::net::TcpListener;
use std::collections::HashMap;
use std::sync::{Arc, Mutex};

#[tokio::main]
async fn main() {
    let listener = TcpListener::bind("127.0.0.1:6379").await.unwrap();

    println!("Listening");

    let db = Arc::new(Mutex::new(HashMap::new()));

    loop {
        let (socket, _) = listener.accept().await.unwrap();
        // Clone the handle to the hash map.
        let db = db.clone();

        println!("Accepted");
        tokio::spawn(async move {
            process(socket, db).await;
        });
    }
}
```

Tokio：传递共享状态

```rust
use tokio::net::TcpStream;
use mini_redis::{Connection, Frame};

async fn process(socket: TcpStream, db: Db) {
    use mini_redis::Command::{self, Get, Set};

    // Connection, provided by `mini-redis`, handles parsing frames from
    // the socket
    let mut connection = Connection::new(socket);

    while let Some(frame) = connection.read_frame().await.unwrap() {
        // Process...

        // Write the response to the client
        connection.write_frame(&response).await.unwrap();
    }
}
```

Tokio：使用共享状态

```rust
let response = match Command::from_frame(frame).unwrap() {
    Set(cmd) => {
        let mut db = db.lock().unwrap();
        db.insert(cmd.key().to_string(), cmd.value().clone());
        Frame::Simple("OK".to_string())
    }
    Get(cmd) => {
        let db = db.lock().unwrap();
        if let Some(value) = db.get(cmd.key()) {
            Frame::Bulk(value.clone())
        } else {
            Frame::Null
        }
    }
    cmd => panic!("unimplemented {:?}", cmd),
};
```

Tokio 中的通道

- mpsc：多生产者、单消费者通道，可以发送多个值。
- oneshot：单生产者、单消费者通道，只能发送单个值。
- broadcast：多生产者、多消费者通道，可以发送多个值，每个接收方都能收到每个值。
- watch：单生产者、多消费者通道，可以发送多个值，但是不保留历史，接收方只能看到最新发送的值。

Tokio：使用通道

```rust
use tokio::sync::mpsc;

#[tokio::main]
async fn main() {
    let (tx, mut rx) = mpsc::channel(32);
    let tx2 = tx.clone();

    tokio::spawn(async move {
        tx.send("sending from first handle").await;
    });

    tokio::spawn(async move {
        tx2.send("sending from second handle").await;
    });

    while let Some(message) = rx.recv().await {
        println!("GOT = {}", message);
    }
}
```

Tokio：异步 IO

```rust
use tokio::fs::File;
use tokio::io::{self, AsyncReadExt};

#[tokio::main]
async fn main() -> io::Result<()> {
    let mut f = File::open("foo.txt").await?;
    let mut buffer = [0; 10];

    // read up to 10 bytes
    let n = f.read(&mut buffer[..]).await?;

    println!("The bytes: {:?}", &buffer[..n]);
    Ok(())
}
```

使用异步的场景与得失

- 任务的类型
- 编程的难度

5

小结

本讲小结

- 输入输出
- 异步编程基础
- Rust 的异步基础
- Rust 的异步生态