官方文档：、
https://tokio.rs/tokio/tutorial
https://tokio.rs/tokio/tutorial/hello-tokio

https://github.com/LinkTed/async-http-proxy.git

https://rust-book.junmajinlong.com/ch100/02_understand_tokio_task.html
Rust语言圣经(Rust Course) https://course.rs/about-book.html

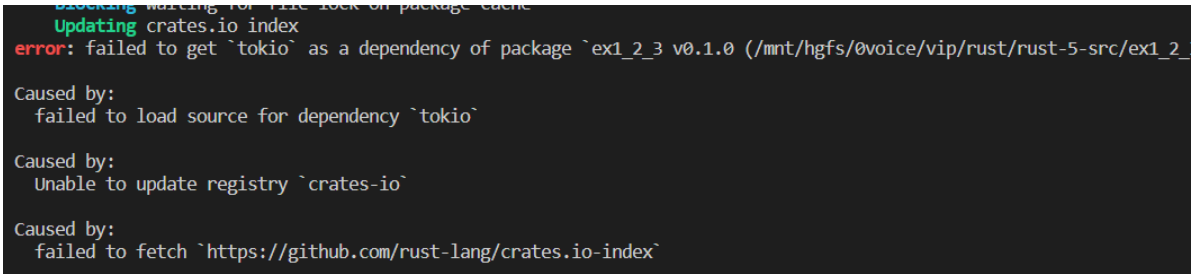
重点内容：

Tokio涉及的知识点非常多，本节课主要讲解基本的用法，以及如何进一步扩展学习。

1. **同步、异步编程概念**
2. **async**
3. **Future**
4. **await**
5. **runtime**
6. **异步通信**
7. **异步同步**

0 Rust更换Crates源

更换源



解决办法：更换Crates源
Rust开发时有时使用官方的源太慢，可以考虑更换使用国内中科大的源。更换方法如下：
在 $HOME/.cargo/config 中添加如下内容：

```
[source.crates-io]
replace-with = 'ustc'

[source.ustc]
registry = "git://mirrors.ustc.edu.cn/crates.io-index"
```

如果所处的环境中不允许使用 git 协议，可以把上述地址改为：

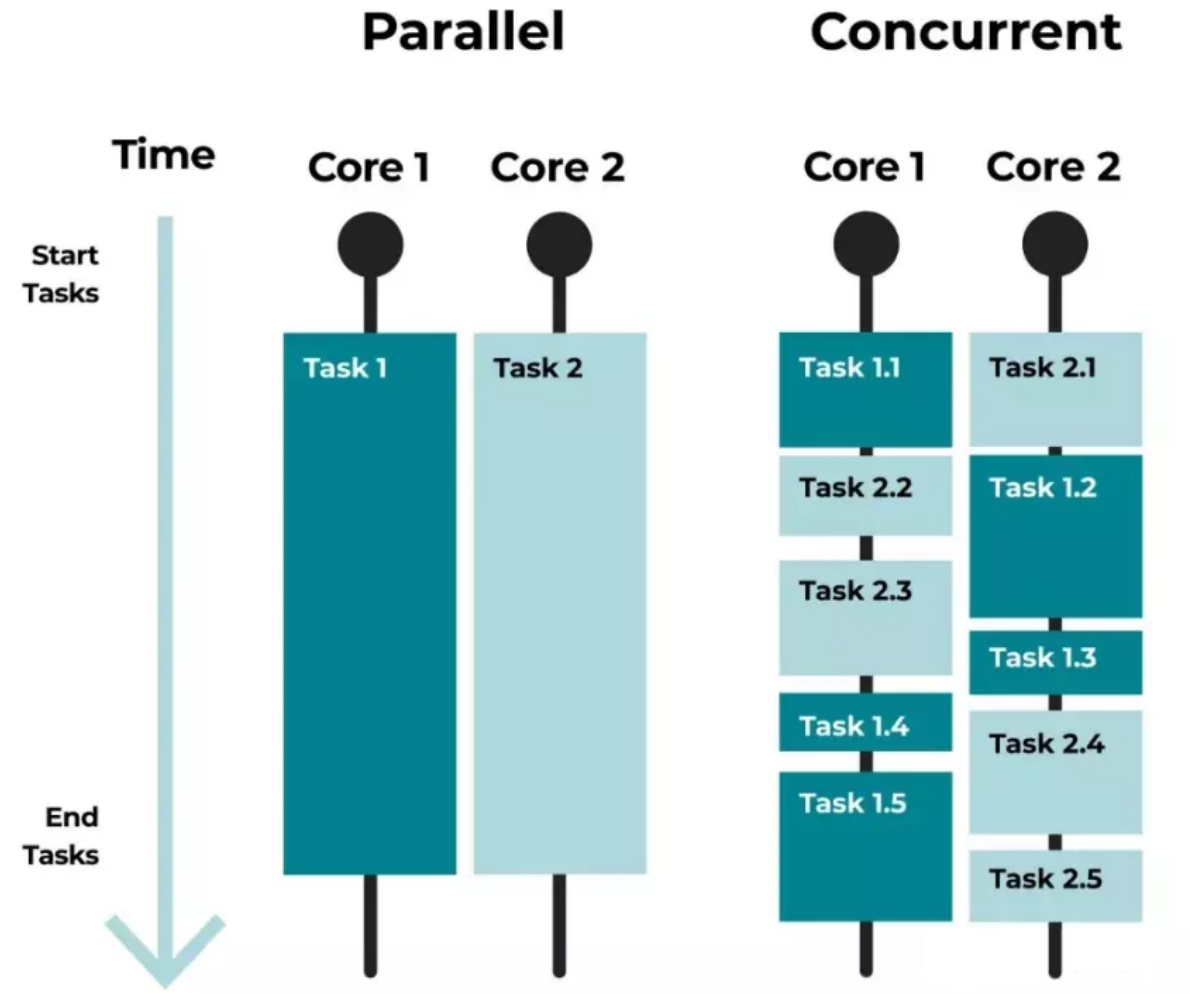
```
registry = "https://mirrors.ustc.edu.cn/crates.io-index"
```

为什么这么配置可以参考The Cargo Book/Source Replacement.

1 同步编程和异步编程基础概念

1.1 并发和并行

程序不按顺序执行就是**并发**（concurrency），而同时执行多个任务是**并行**（parallelism）。



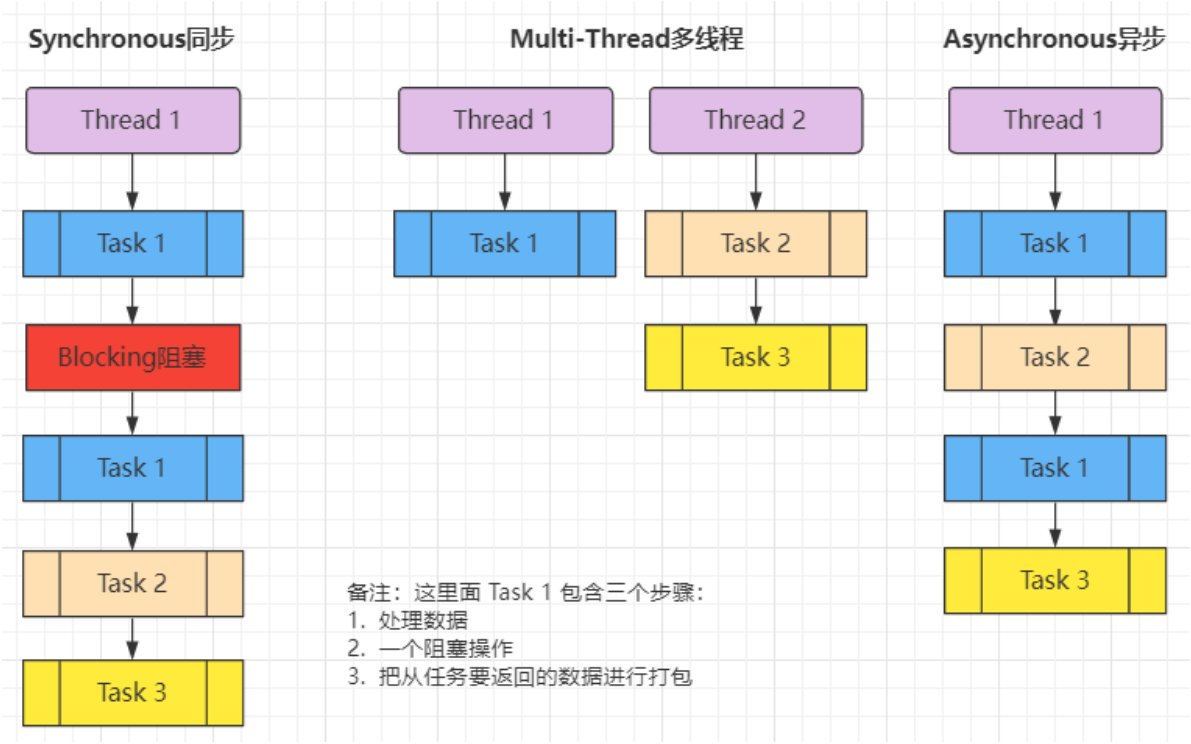
软件程序处理任务的两种类型：

- **CPU密集型**：占用 CPU 资源的任务，例如文件压缩、视频编码、图形处理和计算加密证明。
- **IO密集型**：占用 IO 资源的任务，例如从文件系统或数据库访问数据，以及处理网络TCP/HTTP请求。

1. 对于 CPU 密集型的任务，通常可以利用多 CPU 或多核进行处理。
2. 对于 IO 密集型，我们以 Web 项目中处理请求的任务为例。在 Web 项目中，我们通过 CRUD 操作把数据从数据库里传递过来，这就要求 CPU 等待数据写入到磁盘，但磁盘很慢。所以，如果程序从数据库请求 10000 笔数据，它会向操作系统请求磁盘的访问，而与此同时，CPU 就是在干等。那么程序员应该怎么利用 CPU 的这段等待的时间？那就是让 CPU 执行其它的任务。

另一个典型的例子就是网络请求的处理，客户端建立连接发送请求，服务器端处理请求返回响应并关闭连接。如果CPU还在处理前一个请求，而第二个的请求却已经到达，那么第二个的请求必须在队列中等待着第一个请求处理完成吗？或者我们可以把第二个请求安排到其他可用的CPU或内核上？

1.2 同步、多线程、异步的区别

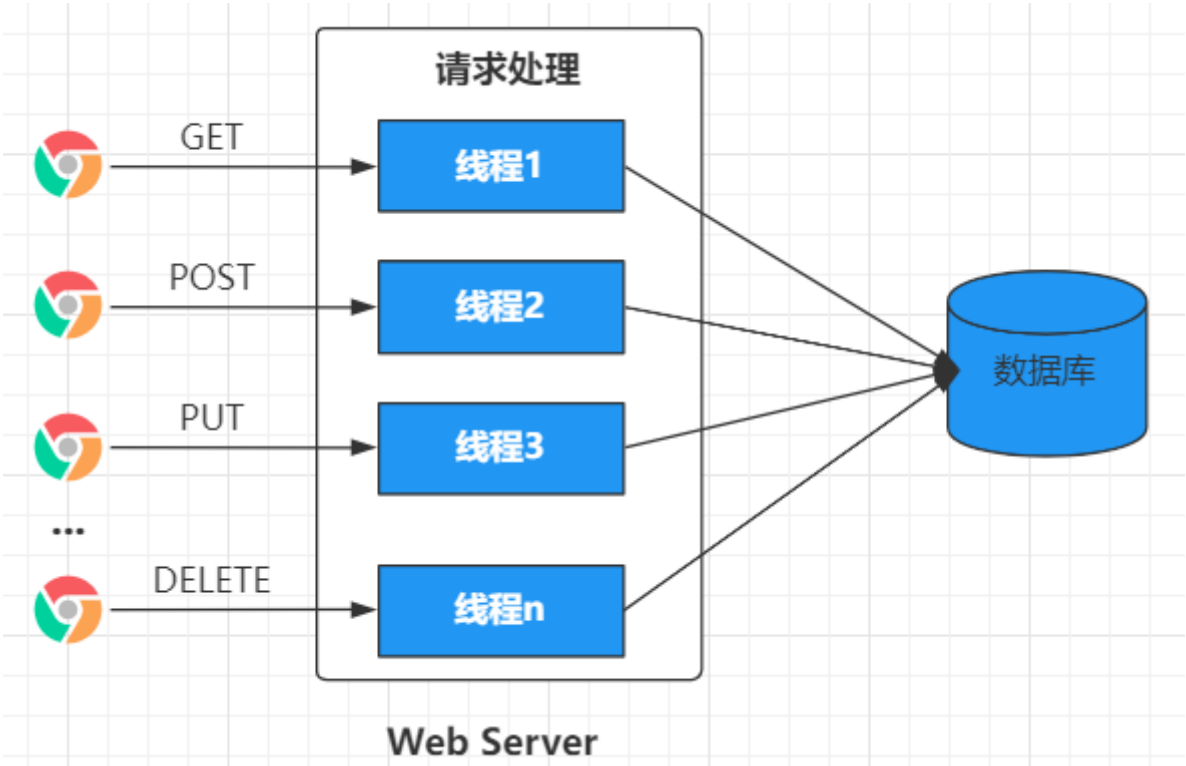


1. 同步执行。执行完 Task 1 的第一步处理数据后，CPU 等着 Task 1 的阻塞操作，然后再执行后续的步骤；
2. 多线程。包含阻塞操作的 Task 1 被安排在单独的系统线程执行，其它的任务在另外的线程中执行；
3. 异步。它会执行 Task 1 直到它开始阻塞等待 I/O。这时异步运行时（例如Tokio）会安排执行Task 2，而当Task 1的阻塞操作结束时，Task 1又被安排到CPU上来运行。

注：哪一种方式更好，需要结合实际业务情况。

1.3 多线程访问模型

1:1模型



第一种方式是使用多线程，针对每个请求都开启一个原生的系统线程。这确实会提高性能，但却引入了新的复杂性：

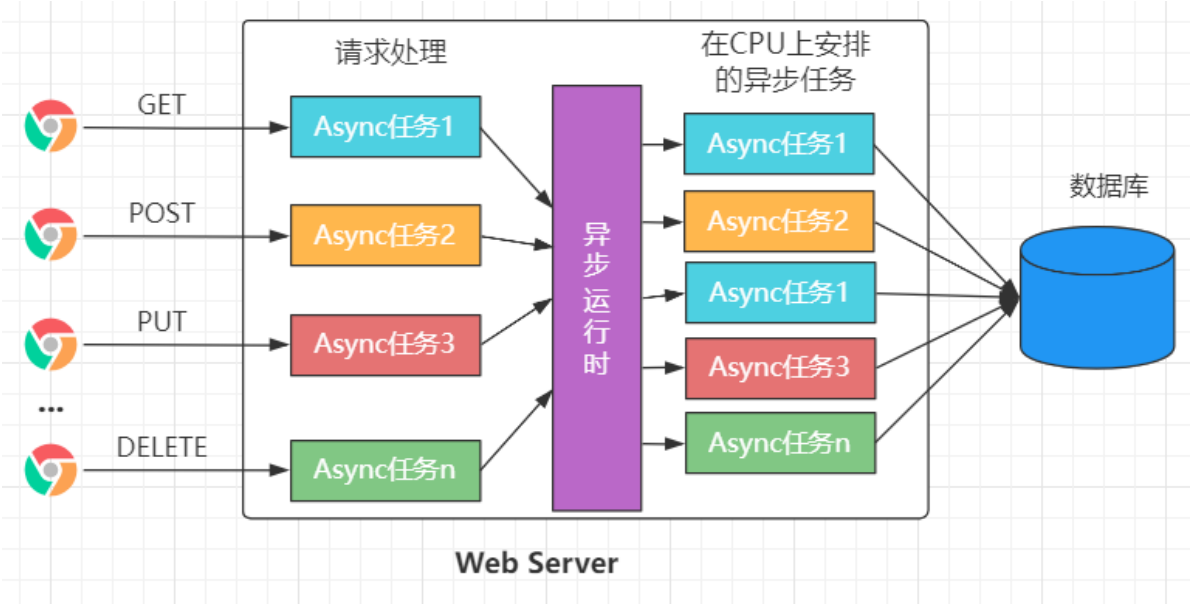
- 执行的顺序无法预测；
- 死锁，多线程同时尝试访问内存中同一块数据；
- 竞态条件（race condition），例如一个线程刚读取完数据并进行计算，而另一个线程却更新了该数据。

M:N模型

此外，多线程还有另外的问题。多线程分为两种模型：

- 1:1 模型，一个语言线程对应一个系统线程；
- **M:N 模型**，M个准（**紫色**）线程对应N个系统线程。

而 Rust 标准库实现的是 1:1 模型。一般来说操作系统对总线程数都有限制，它受到栈内存和虚拟内存量影响。而且线程切换时还有上下文切换的成本和管理线程的其他资源成本。



并发（concurrent)方式，或者叫异步编程。每个 HTTP 请求被异步 Web Server 接收，Web Server 会生成一个任务来处理它，并由异步运行时安排各个异步任务在可用的 CPU 上执行。很明显在该场景下，异步的方式更加合适。

1.4 async/await 异步编程基础概念

1.4.1 async vs 其它并发模型

由于并发编程在现代社会非常重要，因此每个主流语言都对自己的并发模型进行过权衡取舍和精心设计，Rust 语言也不例外。下面的列表可以帮助大家理解不同并发模型的取舍:

- **OS 线程**, 它最简单，也无需改变任何编程模型(业务/代码逻辑)，因此非常适合作为语言的原生并发模型，我们在多线程章节也提到过，Rust 就选择了原生支持线程级的并发编程。但是，这种模型也有缺点，例如线程间的同步将变得更加困难，**线程间的上下文切换损耗较大**。使用线程池在一定程度上可以提升性能，但是对于 IO 密集的场景来说，线程池还是不够。
- **事件驱动(Event driven)**, 这个名词你可能比较陌生，如果说事件驱动常常跟回调(Callback)一起使用，相信大家就恍然大悟了。这种模型性能相当的好，但最大的问题就是存在回调地狱的风险：非线性的控制流和结果处理导致了数据流向和错误传播变得难以掌控，还会导致代码可维护性和可读性的大幅降低，大名鼎鼎的 JavaScript 曾经就存在回调地狱。
- **协程(Coroutines)** 可能是目前最火的并发模型，Go 语言的协程设计就非常优秀，这也是 Go 语言能够迅速火遍全球的杀手锏之一。协程跟线程类似，无需改变编程模型，同时，它也跟 async 类似，可以支持大量的任务并发运行。但协程抽象层次过高，导致用户无法接触到底层的细节，这对于系统编程语言和自定义异步运行时是难以接受的
- **actor 模型**是 erlang 的杀手锏之一，它将所有并发计算分割成一个一个单元，这些单元被称为 actor , 单元之间通过消息传递的方式进行通信和数据传递，跟分布式系统的设计理念非常相像。由于 actor 模型跟现实很贴近，因此它相对来说更容易实现，但是一旦遇到流控制、失败重试等场景时，就会变得不太好用
- **async/await**， 该模型性能高，还能支持底层编程，同时又像线程和协程那样无需过多的改变编程模型，但有得必有失，async 模型的问题就是内部实现机制过于复杂，对于用户来说，**理解和使用起来也没有线程和协程简单**，好在前者的复杂性开发者们已经帮我们封装好，而理解和使用起来不够简单，正是本章试图解决的问题。

总之，Rust 经过权衡取舍后，最终选择了同时提供多线程编程和 async 编程:

- 前者通过标准库实现，当你无需那么高的并发时，例如需要并行计算时，可以选择它，优点是线程内的代码执行效率更高、实现更直观更简单，这块内容已经在多线程章节进行过深入讲解，不再赘述
- 后者通过语言特性 + 标准库 + 三方库的方式实现，在你需要高并发、异步 I/O 时，选择它就对了

1.4.2 async: Rust vs 其它语言

目前已经有诸多语言都通过 async 的方式提供了异步编程，例如 JavaScript ，但 Rust 在实现上有所区别:

- **Future 在 Rust 中是惰性的**，只有在被轮询(poll)时才会运行， 因此丢弃一个 future 会阻止它未来再被运行**, 你可以将Future理解为一个在未来某个时间点被调度执行的任务。**
- **Async 在 Rust 中使用开销是零**， 意味着只有你能看到的代码(自己的代码)才有性能损耗，你看不到的(async 内部实现)都没有性能损耗，例如，你可以无需分配任何堆内存、也无需任何动态分发来使用 async ，这对于热点路径的性能有非常大的好处，正是得益于此，Rust 的异步编程性能才会这么高。
- **Rust 没有内置异步调用所必须的运行时**，但是无需担心，Rust 社区生态中已经提供了非常优异的运行时实现，例如大明星 tokio
- **运行时同时支持单线程和多线程**，这两者拥有各自的优缺点，稍后会讲

1.4.3 Rust: async vs 多线程

虽然 async 和多线程都可以实现并发编程，后者甚至还能通过线程池来增强并发能力，但是这两个方式并不互通，从一个方式切换成另一个需要大量的代码重构工作，因此**提前为自己的项目选择适合的并发模型就变得至关重要。**

OS 线程非常适合少量任务并发，因为线程的创建和上下文切换是非常昂贵的，甚至于空闲的线程都会消耗系统资源。虽说线程池可以有效的降低性能损耗，但是也无法彻底解决问题。当然，线程模型也有其优点，例如它不会破坏你的代码逻辑和编程模型，你之前的顺序代码，经过少量修改适配后依然可以在新线程中直接运行，同时在某些操作系统中，你还可以改变线程的优先级，这对于实现驱动程序或延迟敏感的应用(例如硬实时系统)很有帮助。

对于长时间运行的 CPU 密集型任务，例如**并行计算，使用线程将更有优势。** 这种密集任务往往会让所在的线程持续运行，任何不必要的线程切换都会带来性能损耗，因此高并发反而在此时成为了一种多余。同时你所创建的线程数应该等于 CPU 核心数，充分利用 CPU 的并行能力，**甚至还可以将线程绑定到 CPU 核心上，进一步减少线程上下文切换。**

而高并发更适合 IO 密集型任务，例如 web 服务器、数据库连接等等网络服务，因为这些任务绝大部分时间都处于等待状态，如果使用多线程，那线程大量时间会处于无所事事的状态，再加上线程上下文切换的高昂代价，让多线程做 IO 密集任务变成了一件非常奢侈的事。而使用async，既可以有效的降低 CPU 和内存的负担，又可以让大量的任务并发的运行，一个任务一旦处于IO或者其他等待(阻塞)状态，就会被立刻切走并执行另一个任务，而这里的任务切换的性能开销要远远低于使用多线程时的线程上下文切换。

事实上, **async 底层也是基于线程实现，但是它基于线程封装了一个运行时**，可以将多个任务映射到少量线程上，然后将线程切换变成了任务切换，后者仅仅是内存中的访问，因此要高效的多。

不过async也有其缺点，原因是编译器会为async函数生成状态机，然后将整个运行时打包进来，这会造成我们编译出的**二进制可执行文件体积显著增大。**

总之，async编程并不一定比多线程更好，最终需要根据你的使用场景作出合适的选择，如果无需高并发，或者也不在意线程切换带来的性能损耗，那么多线程使用起来会简单、方便的多！最后再简单总结下:
若大家使用 tokio，**那 CPU 密集的任务尤其需要用线程的方式去处理**，例如使用 spawn_blocking 创建一个阻塞的线程取完成相应 CPU 密集任务。

至于具体的原因，不仅是上文说到的那些，还有一个是：tokio 是协作式地调度器，如果某个 CPU 密集的异步任务是通过 tokio 创建的，那理论上来说，该异步任务需要跟其它的异步任务交错执行，最终大家都得到了执行，皆大欢喜。但实际情况是，CPU 密集的任务很可能会一直霸着着 CPU，此时 tokio 的调度方式决定了该任务会一直被执行，这意味着，其它的异步任务无法得到执行的机会，最终这些任务都会因为得不到资源而饿死。

而使用 **spawn_blocking** 后，会创建一个单独的 OS 线程，该线程并不会被 tokio 所调度(被 OS 所调度)，因此它所执行的 CPU 密集任务也不会导致 tokio 调度的那些异步任务被饿死

- 有大量 IO 任务需要并发运行时，选 async 模型
- 有部分 IO 任务需要并发运行时，选多线程，如果想要降低线程创建和销毁的开销，可以使用线程池
- 有大量 CPU 密集任务需要并行运行时，例如并行计算，选多线程模型，且让线程数等于或者稍大于 CPU 核心数
- 无所谓时，统一选多线程（即是能用多线程解决尽量使用多线程）

1.4.4 async 和多线程的性能对比

操作	async	线程
创建	0.3 微秒	17 微秒
线程切换	0.2 微秒	1.7 微秒

可以看出，async 在线程切换的开销显著低于多线程，对于 IO 密集的场景，这种性能开销累计下来会非常可怕!

1.4.5 语言和库的支持

async 的底层实现非常复杂，且会导致编译后文件体积显著增加，因此 Rust 没有选择像 Go 语言那样内置了完整的特性和运行时，而是选择了通过 Rust 语言提供了必要的特性支持，再通过社区来提供 async 运行时的支持。 因此要完整的使用 async 异步编程，你需要依赖以下特性和外部库:

- 所必须的特征(例如 **Future**)、类型和函数，由标准库提供实现
- 关键字 async/await 由 Rust 语言提供，并进行了编译器层面的支持
- 众多实用的类型、宏和函数由官方开发的 futures 包提供(不是标准库)，它们可以用于任何 async 应用中。
- async 代码的执行、IO 操作、任务创建和调度等等复杂功能由社区的 async 运行时提供，**例如 tokio 和 async-std**

1.5 async/.await 简单入门

async/.await 是 Rust 内置的语言特性，可以让我们用同步的方式去编写异步的代码。
通过 async 标记的语法块会被转换成实现了Future特征的状态机。 与同步调用阻塞当前线程不同，当Future执行并遇到阻塞时，它会让出当前线程的控制权，这样其它的Future就可以在该线程中运行，这种方式完全不会导致当前线程的阻塞。
下面我们来通过例子学习 async/.await 关键字该如何使用，在开始之前，需要先引入 futures 包。编辑 Cargo.toml 文件并添加以下内容:

```
[dependencies]
features ="0.10.0"
```

1.5.1 使用 async

首先，使用 async fn 语法来创建一个异步函数:

```
async fn do_something() {
    println!("go go go !");
}
```

需要注意，**异步函数的返回值是一个 Future**，若直接调用该函数，不**会输出任何结果，因为 Future 还未被执行**:
ex1_5_1_1

```
async fn do_something() {
    println!("go go go !");
}

fn main() {
    do_something();
}
```

运行后，go go go并没有打印，同时编译器给予一个提示：**warning: unused implementer of Future that must be used**，告诉我们 Future 未被使用，那么到底该如何使用？答案是使用一个执行器(executor):
ex1_5_1_2

```
// `block_on`会阻塞当前线程直到指定的`Future`执行完成，这种阻塞当前线程以等待任务完成的方式较为简单、粗暴，
// 好在其它运行时的执行器(executor)会提供更加复杂的行为，例如将多个`future`调度到同一个线程上执行。
use futures::executor::block_on;

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

fn main() {
    let future = hello_world(); // 返回一个Future，因此不会打印任何输出
    block_on(future); // 执行`Future`并等待其运行完成，此时"hello, world!"会被打印输出
}
```

1.5.2 使用.await

在上述代码的main函数中，我们使用block_on这个执行器等待**Future的完成**，让代码看上去非常像是同步代码，但是如果你要在一个async fn函数中去调用**另一个async fn**并等待其完成后再执行后续的代码，该如何做？例如:
ex1_5_2_1

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

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

async fn hello_cat() {
    println!("hello, cat!");
```

```
}
fn main() {
    let future = hello_world();
    block_on(future);
}
```

这里，我们在hello_world异步函数中先调用了另一个异步函数hello_cat，然后再输出hello, world!，看看运行结果：

```
warning: unused implementer of `futures::Future` that must be used
 --> src/main.rs:6:5
  |
6 |     hello_cat();
  |     ^^^^^^^^^^^^
= note: futures do nothing unless you `.await` or poll them
...
hello, world!
```

不出所料，main函数中的future我们通过block_on函数进行了运行，但这里hello_cat返回的Future却没有任何人去执行它，不过好在编译器友善的给出了提示：futures do nothing unless you `.await` or poll them，两种解决方法：使用.await语法或者对Future进行轮询(poll)。
后者较为复杂，暂且不表，先来使用**.await**试试：

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

async fn hello_world() {
    hello_cat().await;  // 线程有切换去执行其他函数
    println!("hello, world!");
}

async fn hello_cat() {
    println!("hello, cat!");
}
fn main() {
    let future = hello_world();
    block_on(future);
}
```

为hello_cat()添加上.await后，结果立刻大为不同:
hello, cat!

hello, world!
输出的顺序跟代码定义的顺序完全符合，因此，我们在上面代码中**使用同步的代码顺序实现了异步的执行效果**，非常简单、高效，而且很好理解。

总之，在async fn函数中使用.await可以等待另一个异步调用的完成。**但是与block_on不同，.await并不会阻塞当前的线程**，而是异步的等待Future A的完成，在等待的过程中，该线程还可以继续执行其它的Future B，最终实现了并发处理的效果。

1.5.3 一个例子

考虑一个载歌载舞的例子，如果不用.await，我们可能会有如下实现：
ex1_5_3_1

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

struct Song {
    author: String,
    name: String,
}

async fn learn_song() -> Song {
    Song {
        author: "王菲".to_string(),
        name: String::from("《我的歌声里》"),
    }
}

async fn sing_song(song: Song) {
    println!(
        "给大家献上一首{}的{} ~ {}",
        song.author, song.name, "你存在我深深的脑海里~ ~"
    );
}

async fn dance() {
    println!("唱到情深处，身体不由自主的动了起来~ ~");
}

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

当然，以上代码运行结果无疑是正确的，但。。。它的性能何在？**需要通过连续三次阻塞去等待三个任务的完成**，一次只能做一件事，实际上我们完全可以载歌载舞啊:

ex1_5_3_2

Cargo.toml 加入依赖库

```
[dependencies]
futures = "0.3"
async-std = { version = "1", features = ["attributes"] }
```

```
use futures::executor::block_on;
use std::time::Duration;

use async_std::task;
struct Song {
    author: String,
    name: String,
```

```
}

async fn learn_song() -> Song {
    task::sleep(Duration::from_secs(1)).await;  // 故意休眠一小会
    Song {
        author: "王菲".to_string(),
        name: String::from("《我的歌声里》"),
    }

}

async fn sing_song(song: Song) {
    println!(
        "给大家献上一首{}的{} ~ {}",
        song.author, song.name, "你存在我深深的脑海里~ ~"
    );
}

async fn dance() {
    println!("跳舞 跳舞~");
}

async fn learn_and_sing() {
    // 这里使用`.await`来等待学歌的完成，但是并不会阻塞当前线程，该线程在学歌的任务`.await`后，完全可以去执行跳舞的任务
    println!("学歌曲");
    let song = learn_song().await;
    // 唱歌必须要在学歌之后
    println!("唱歌曲");
    sing_song(song).await;
}

async fn async_main() {  // 本身他也是一个异步任务
    let f1 = learn_and_sing();
    let f2 = dance();

    // `join!`可以并发的处理和等待多个`Future`，若`learn_and_sing Future`被阻塞，那`dance Future`可以拿过线程的所有权继续执行。若`dance`也变成阻塞状态，那`learn_and_sing`又可以再次拿回线程所有权，继续执行。
    // 若两个都被阻塞，那么`async main`会变成阻塞状态，然后让出线程所有权，并将其交给`main`函数中的`block_on`执行器
    futures::join!(f1, f2);
}

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

上面代码中，**学歌和唱歌**具有明显的先后顺序，但是这两者都可以跟跳舞一同存在，也就是你可以在跳舞的时候学歌，也可以在跳舞的时候唱歌。**如果上面代码不使用.await，而是使用block_on(learn_song())，** 那在学歌时，当前线程就会阻塞，不再可以做其它任何事，包括跳舞。

因此**.await对于实现异步编程至关重要**，它允许我们在同一个线程内并发的运行多个任务，而不是一个一个先后完成。若大家看到这里还是不太明白，强烈建议回头再仔细看一遍，同时亲自上手修改代码试试效果。

1.6 异步编程总结

对于 Async Rust，最最重要的莫过于底层的异步运行时，它提供了执行器、任务调度、异步API等核心服务。简单来说，使用 Rust 提供的 `async/.await` 特性编写的异步代码要运行起来，就必须依赖于异步运行时，否则这些代码将毫无用处。

如何选择异步运行时

Rust 语言本身只提供了异步编程所需的基本特性，例如 `async/.await` 关键字，标准库中的 `Future` 特征，官方提供的 `futures` 实用库，这些特性单独使用没有任何用处，因此我们需要一个运行时来将这些特性实现的代码运行起来。

异步运行时是由 Rust 社区提供的，它们的核心是一个 `reactor` 和一个或多个 `executor`(执行器):

- `reactor` 用于提供外部事件的订阅机制，例如 `I/O` 、进程间通信、定时器等
- `executor` 在上一章我们有过深入介绍，它用于调度和执行相应的任务(`Future`)

目前最受欢迎的几个运行时有:

- `tokio`，目前最受欢迎的异步运行时，功能强大，还提供了异步所需的各种工具(例如 tracing)、网络协议框架(例如 HTTP，gRPC)等等
- `async-std`，最大的优点就是跟标准库兼容性较强
- `smol`, 一个小巧的异步运行时

但是，大浪淘沙，留下的才是金子，随着时间的流逝，`tokio`越来越亮眼，无论是性能、功能还是社区、文档，它在各个方面都异常优秀，时至今日，可以说已成为事实上的标准。

2 Rust tokio核心-runtime和task

对于 Async Rust，最最重要的莫过于底层的异步运行时，它提供了执行器、任务调度、异步API等核心服务。简单来说，使用 Rust 提供的 `async/.await` 特性编写的异步代码要运行起来，就必须依赖于异步运行时，否则这些代码将毫无用处。

tokio两大核心：**runtime和task**

2.1 tokio核心: runtime

tokio提供了两种工作模式的**runtime**:

- 1.单一线程的runtime(single thread runtime，也称为**current thread runtime**)
- 2.多线程(线程池)的runtime(multi thread runtime)

注: 这里的所说的线程是Rust线程，而每一个Rust线程都是一个**OS线程（比如linux线程）**。

IO并发类任务较多时，单一线程的runtime性能不如多线程的runtime，但因为多线程runtime使用了多线程，使得线程间的通信变得更为复杂，也加重了线程间切换的开销，使得有些情况下的性能不如使用单线程runtime。因此，在要求极限性能的时候，建议测试两种工作模式的性能差距来选择更优解。

2.1.1 创建tokio Runtime

创建单一线程ex2_1_1_1

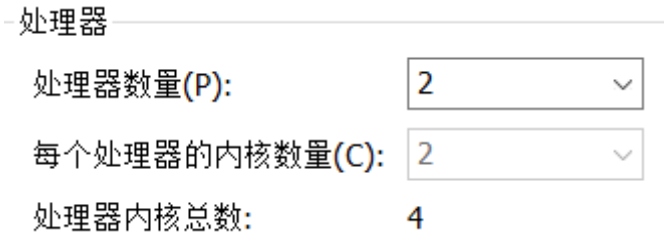
```
// 创建单一线程的runtime
let rt = tokio::runtime::Builder::new_current_thread().build().unwrap();
```

创建一个多线程的runtime， ex2_1_1_2

```
let rt = tokio::runtime::Runtime::new().unwrap();
```

在另一个终端查看线程数

```
 ps -eLf | grep 'ex2_1_1_2'
```



总共5个OS线程，其中**4个worker thread**(我的虚拟机是2核4线程的)，外加一个main thread。

2.1.2 async main

对于main函数，tokio提供了简化的异步运行时创建方式：

```
use tokio;

#[tokio::main]
async fn main() {}
```

通过#[tokio::main]**注解(annotation)**，使得async main自身成为一个async runtime。

#[tokio::main]创建的是多线程runtime，还有以下几种方式创建多线程runtime：

```
#[tokio::main(flavor = "multi_thread"] // 等价于#[tokio::main]
#[tokio::main(flavor = "multi_thread", worker_threads = 10))]
#[tokio::main(worker_threads = 10))]
```

等价于如下没有使用#[tokio::main]的代码：

```
fn main(){
  tokio::runtime::Builder::new_multi_thread()
        .worker_threads(N)
        .enable_all()
        .build()
        .unwrap()
        .block_on(async { ... });
}
```

#[tokio::main]也可以创建单一线程的main runtime：

```
#[tokio::main(flavor = "current_thread")]
```

等价于：

```
fn main() {
    tokio::runtime::Builder::new_current_thread()
        .enable_all()
        .build()
        .unwrap()
        .block_on(async { ... })
}
```

2.1.3 多个runtime共存

可手动创建线程，并在不同线程内创建互相独立的runtime。
ex2_1_3

```
use std::thread;
use std::time::Duration;
use tokio::runtime::Runtime;

fn main() {
  // 在第一个线程内创建一个多线程的runtime
  let t1 = thread::spawn(||{
    let rt = Runtime::new().unwrap();
    thread::sleep(Duration::from_secs(10));
  });

  // 在第二个线程内创建一个多线程的runtime
  let t2 = thread::spawn(||{
    let rt = Runtime::new().unwrap();
    thread::sleep(Duration::from_secs(10));
  });

  t1.join().unwrap();
  t2.join().unwrap();
}
```

注意：worker-thread，spawn-thread， main-thread的区别。

runtime实现了Send和Sync这两个Trait，因此可以将**runtime包在Arc里，然后跨线程使用同一个runtime。**

2.1.4 进入runtime: 阻塞的block_on

多数时候，异步任务是一些带有网络IO操作的任务，比如异步的http请求。
比如**ex2_1_4** (需要在Cargo.toml引入chrono = "0.4")

```
use tokio::runtime::Runtime;
use chrono::Local;

fn main() {
    let rt = Runtime::new().unwrap(); // run time
    rt.block_on(async {
        println!("before sleep: {}", Local::now().format("%F %T.%3f"));
        tokio::time::sleep(tokio::time::Duration::from_secs(2)).await;
        println!("after sleep: {}", Local::now().format("%F %T.%3f"));
    });
}
```

block_on要求一个Future作为参数，可以直接使用一个async {}来定义一个Future。每一个Future都是一个已经定义好但尚未执行的异步任务，**每一个异步任务中可能会包含其它子任务。**

这些异步任务不会直接执行，需要先将它们放入到runtime环境，然后在合适的地方通过Future的**await**来执行它们。**await可以将已经定义好的异步任务立即加入到runtime的任务队列中等待调度执行，于此同时，await会等待该异步任务完成才返回。**例如：

```
rt.block_on(async {
    // 只是定义了Future，此时尚未执行
    let task = tokio::time::sleep(tokio::time::Duration::from_secs(2));
    // ...不会执行...
    // ...
    // 开始执行task任务，并等待它执行完成
    task.await;

    // 上面的任务完成之后，才会继续执行下面的代码
});
```

block_on会阻塞当前线程(例如阻塞住上面的main函数所在的主线程)，直到其指定的**异步任务树(可能有子任务)**全部完成。
注：**block_on是等待异步任务完成**，而不是等待runtime中的所有任务都完成。
block_on也有返回值，其返回值为其所执行异步任务的返回值。例如：

```
use tokio::{time, runtime::Runtime};

fn main() {
    let rt = Runtime::new().unwrap();
    let res: i32 = rt.block_on(async{
      time::sleep(time::Duration::from_secs(2)).await;
      3
    });
    println!("{}", res);  // 3
}
```

2.1.5 spawn: 向runtime中添加新的异步任务

有时候，**定义要执行的异步任务时，并未身处runtime内部。**例如定义一个异步函数，此时可以使用**tokio::spawn()**来生成异步任务。
ex2_1_5

```
use std::thread;
```

```
use chrono::Local;
use tokio::{self, runtime::Runtime, time};

fn now() -> String {
    Local::now().format("%F %T").to_string()
}

// 在runtime外部定义一个异步任务，且该函数返回值不是Future类型
fn async_task() {
  println!("create an async task: {}", now());
  tokio::spawn(async {
    time::sleep(time::Duration::from_secs(10)).await;
    println!("async task over: {}", now());
  });
}

fn main() {
    let rt1 = Runtime::new().unwrap();
    rt1.block_on(async {
      // 调用函数，该函数内创建了一个异步任务，将在当前runtime内执行
      async_task();
    });
}
```

除了**tokio::spawn()**，runtime自身也能spawn，因此，也可以传递runtime(注意，要传递runtime的引用)，然后使用runtime的spawn()。

```
use tokio::{Runtime, time}
fn async_task(rt: &Runtime) {
  rt.spawn(async {
    time::sleep(time::Duration::from_secs(10)).await;
  });
}

fn main(){
  let rt = Runtime::new().unwrap();
  rt.block_on(async {
    async_task(&rt);
  });
}
```

2.1.6 进入runtime: 非阻塞的enter()

block_on()进入runtime时，会阻塞当前线程，enter()进入runtime时，不会阻塞当前线程，它会返回一个EnterGuard。EnterGuard没有其它作用，它仅仅只是声明从它开始的所有异步任务都将在runtime上下文中执行，直到删除该EnterGuard。

删除EnterGuard并不会删除runtime，只是释放之前的runtime上下文声明。因此，删除EnterGuard之后，可以声明另一个EnterGuard，这可以再次进入runtime的上下文环境。
ex2_1_6

```
use tokio::{self, runtime::Runtime, time};
use chrono::Local;
use std::thread;

fn now() -> String {
```

```
  Local::now().format("%F %T").to_string()
}

fn main() {
    let rt = Runtime::new().unwrap();

    // 进入runtime，但不阻塞当前线程
    let guard1 = rt.enter();

    // 生成的异步任务将放入当前的runtime上下文中执行
    tokio::spawn(async {
      time::sleep(time::Duration::from_secs(5)).await;
      println!("task1 sleep over: {}", now());
    });

    // 释放runtime上下文，这并不会删除runtime
    drop(guard1);

    // 可以再次进入runtime
    let guard2 = rt.enter();
    tokio::spawn(async {
      time::sleep(time::Duration::from_secs(4)).await;
      println!("task2 sleep over: {}", now());
    });

    drop(guard2);
    println!("sleep wait end");
    // 阻塞当前线程，等待异步任务的完成
    thread::sleep(std::time::Duration::from_secs(10));
}
```

2.1.7 理解runtime和异步调度

异步Runtime提供了异步IO驱动、异步计时器等异步API，还提供了任务的调度器(scheduler)和Reactor事件循环(Event Loop)。
每当创建一个Runtime时，就在这个Runtime中创建好了一个Reactor和一个Scheduler，同时还创建了一个任务队列。
PS: 异步运行时和操作系统的进程调度方式是类似的，只不过现代操作系统的进程调度逻辑要比异步运行时的调度逻辑复杂的多。

当一个异步任务需要运行，这个任务要被放入到可运行的任务队列(就绪队列)，然后等待被调度，当一个异步任务需要阻塞时(对应那些在同步环境下会阻塞的操作)，它被放进阻塞队列。
阻塞队列中的每一个被阻塞的任务，都需要等待Reactor收到对应的事件通知(比如IO完成的通知、睡眠完成的通知等)来唤醒它。当该任务被唤醒后，它将被放入就绪队列，等待调度器的调度。
就绪队列中的每一个任务都是可运行的任务，可随时被调度器调度选中。调度时会选择哪一个任务，是调度器根据调度算法去决定的。某个任务被调度选中后，调度器将分配一个线程去执行该任务。

以上是通用知识，用于理解何为异步调度系统，每个调度系统都有自己的特性。例如，Rust tokio并不完全按照上面所描述的方式进行调度。tokio的作者，非常友好地提供了一篇他实现tokio调度器的思路，里面详细介绍了调度器的基本知识和tokio调度器的调度策略，参考[Making the Tokio scheduler 10x faster]。

2.1.8 tokio的两种线程：worker thread和blocking thread

tokio提供了两种功能的线程：

- **用于异步任务的工作线程(worker thread)**
- **用于同步任务的阻塞线程(blocking thread)**

单个线程或多个线程的runtime，指的都是工作线程，即只用于执行异步任务的线程，这些任务主要是IO密集型的任务。tokio默认会将每一个工作线程均匀地绑定到每一个CPU核心上。

有些必要的任务可能会长时间计算而占用线程，甚至任务可能是同步的，它会直接阻塞整个线程(比如thread::time::sleep())，这类任务如果计算时间或阻塞时间较短，勉强可以考虑留在异步队列中，但如果任务计算时间或阻塞时间可能会较长，它们将不适合放在异步队列中，因为它们会破坏异步调度，使得同线程中的其它异步任务处于长时间等待状态，也就是说，这些异步任务可能会很长一段时间得不到执行。

直接在runtime中执行阻塞线程的操作，由于这类阻塞操作不在tokio系统内，tokio无法识别这类线程阻塞的操作，tokio只能等待该线程阻塞操作的结束，才能重新获得那个线程的管理权。换句话说，worker thread被线程阻塞的时候，它已经脱离了tokio的控制，在一定程度上破坏了tokio的调度系统。

```
rt.block_on(async{
  // 在runtime中，让整个线程进入睡眠，注意不是tokio::time::sleep()
  std::thread::sleep(std::time::Duration::from_secs(10));
});
```

因此，tokio提供了这两类不同的线程。worker thread只用于执行那些异步任务，异步任务指的是不会阻塞线程的任务。而一旦遇到本该阻塞但却不会阻塞的操作(如使用tokio::time::sleep()而不是std::thread::sleep())，会直接放弃CPU，将线程交还给调度器，使该线程能够再次被调度器分配到其它异步任务。blocking thread则用于那些长时间计算的或阻塞整个线程的任务。

blocking thread默认是不存在的，只有在调用了spawn_blocking()时才会创建一个对应的blocking thread。

blocking thread不用于执行异步任务，因此runtime不会去调度管理这类线程，它们在本质上相当于一个独立的thread::spawn()创建的线程，它也不会像block_on()一样会阻塞当前线程。它和独立线程的唯一区别，是blocking thread是在runtime内的，可以在runtime内对它们使用一些异步操作，例如await。
ex2_1_8

```
use std::thread;
use chrono::Local;
use tokio::{self, runtime::Runtime, time};

fn now() -> String {
    Local::now().format("%F %T").to_string()
}

fn main() {
    let rt1 = Runtime::new().unwrap();
    // 创建一个blocking thread，可立即执行(由操作系统调度系统决定何时执行)
    // 注意，不阻塞当前线程
    let task = rt1.spawn_blocking(|| {
      println!("in task: {}", now());
      // 注意，是线程的睡眠，不是tokio的睡眠，因此会阻塞整个线程
      thread::sleep(std::time::Duration::from_secs(10))
    });
```

```
    // 小睡1毫秒，让上面的blocking thread先运行起来
    std::thread::sleep(std::time::Duration::from_millis(1));
    println!("not blocking: {}", now());

    // 可在runtime内等待blocking_thread的完成
    rt1.block_on(async {
      task.await.unwrap();
      println!("after blocking task: {}", now());
    });
}
```

```
in task: 2022-05-06 22:02:39
not blocking: 2022-05-06 22:02:39
after blocking task: 2022-05-06 22:02:49
```

需注意，blocking thread生成的任务虽然绑定了runtime，**但是它不是异步任务，不受tokio调度系统控制**。因此，如果在block_on()中生成了blocking thread或普通的线程，block_on()不会等待这些线程的完成。

```
rt.block_on(async{
  // 生成一个blocking thread和一个独立的thread
  // block on不会阻塞等待两个线程终止，因此block_on在这里会立即返回
  rt.spawn_blocking(|| std::thread::sleep(std::time::Duration::from_secs(10)));
  thread::spawn(||std::thread::sleep(std::time::Duration::from_secs(10)));
});
```

tokio允许的blocking thread队列很长(默认512个)，且可以在runtime build时通过max_blocking_threads()配置最大长度。如果超出了最大队列长度，新的任务将放在一个等待队列中进行等待(比如当前已经有512个正在运行的任务，下一个任务将等待，直到有某个blocking thread空闲)。
blocking thread执行完对应任务后，并不会立即释放，而是继续保持活动状态一段时间，此时它们的状态是空闲状态。当空闲时长超出一定时间后(可在runtime build时通过thread_keep_alive()配置空闲的超时时长)，该空闲线程将被释放。
blocking thread有时候是非常友好的，它像独立线程一样，但又和runtime绑定，它不受tokio的调度系统调度，tokio不会把其它任务放进该线程，也不会把该线程内的任务转移到其它线程。**换言之，它有机会完完整整地发挥单个线程的全部能力，而不像worker线程一样，可能会被调度器打断。**

2.1.9 关闭Runtime

正常关闭

超时关闭

由于异步任务完全依赖于Runtime，而Runtime又是程序的一部分，它可以轻易地被删除(drop)，这时Runtime会被关闭(shutdown)。

```
let rt = Runtime::new().unwrap();
...
drop(rt);
```

这里的变量rt，官方手册将其称为runtime的句柄(runtime handle)。
关闭Runtime时，将使得该Runtime中的所有**「异步任务」**被移除。完整的关闭过程如下：

1. 先移除整个任务队列，保证不再产生也不再调度新任务
2. 移除当前正在执行但尚未完成的**「异步任务」**，即终止所有的*worker thread*
3. 移除*Reactor*，禁止接收事件通知

注意，这种删除runtime句柄的方式只会立即关闭未被阻塞的worker thread，那些已经运行起来的**blocking thread以及已经阻塞整个线程的worker thread仍然会执行。**但是，**删除runtime又要等待runtime中的所有异步和非异步任务(会阻塞线程的任务)都完成，因此删除操作会阻塞当前线程。**
ex2_1_9

```
use std::thread;
use chrono::Local;
use tokio::{self, runtime::Runtime, time};

fn now() -> String {
    Local::now().format("%F %T").to_string()
}

fn main() {
    let rt = Runtime::new().unwrap();
    // 一个运行5秒的blocking thread
    // 删除rt时，该任务将继续运行，直到自己终止
    rt.spawn_blocking(|| {
      thread::sleep(std::time::Duration::from_secs(5));
      println!("blocking thread task over: {}", now());
    });

    // 进入runtime，并生成一个运行3秒的异步任务，
    // 删除rt时，该任务直接被终止
    let _guard = rt.enter();
    rt.spawn(async {
      time::sleep(time::Duration::from_secs(3)).await;
      println!("worker thread task over 1: {}", now());
    });

    // 进入runtime，并生成一个运行4秒的阻塞整个线程的任务
    // 删除rt时，该任务继续运行，直到自己终止
    rt.spawn(async {
      std::thread::sleep(std::time::Duration::from_secs(4));
      println!("worker thread task over 2: {}", now());
    });

    // 先让所有任务运行起来
    std::thread::sleep(std::time::Duration::from_millis(3));

    // 删除runtime句柄，将直接移除那个3秒的异步任务，
    // 且阻塞5秒，直到所有已经阻塞的thread完成
    drop(rt);
    println!("runtime droped: {}", now());
}
```

输出结果

```
worker thread task over 2: 2022-05-06 22:09:04
blocking thread task over: 2022-05-06 22:09:05
runtime droped: 2022-05-06 22:09:05
```

关闭runtime可能会被阻塞，因此，如果是在某个runtime中关闭另一个runtime，将会导致当前的runtime的某个worker thread被阻塞，甚至可能会阻塞很长时间，这是异步环境不允许的。

tokio提供了另外两个关闭runtime的方式：**shutdown_timeout()和shutdown_background()。**前者会等待指定的时间，如果正在超时时间内还未完成关闭，shutdown_timeout**将强行终止runtime中的所有线程。后者是shutdown_background立即强行关闭runtime。**

```
use std::thread;
use chrono::Local;
use tokio::{self, runtime::Runtime, time};

fn now() -> String {
    Local::now().format("%F %T").to_string()
}

fn main() {
    let rt = Runtime::new().unwrap();

    rt.spawn_blocking(|| {
      thread::sleep(std::time::Duration::from_secs(5));
      println!("blocking thread task over: {}", now());
    });

    let _guard = rt.enter();
    rt.spawn(async {
      time::sleep(time::Duration::from_secs(3)).await;
      println!("worker thread task over 1: {}", now());
    });

    rt.spawn(async {
      std::thread::sleep(std::time::Duration::from_secs(4));
      println!("worker thread task over 2: {}", now());
    });

    // 先让所有任务运行起来
    std::thread::sleep(std::time::Duration::from_millis(3));

    // 1秒后强行关闭Runtime
    rt.shutdown_timeout(std::time::Duration::from_secs(1));
    println!("runtime droped: {}", now());
}
```

需要注意的是，强行关闭Runtime，可能会使得尚未完成的任务的资源泄露。因此，应小心使用强行关闭Runtime的操作。

2.1.10 untime Handle

tokio提供了一个称为runtime Handle的东西，它实际上是runtime的一个引用，可以随意被clone。它可以spawn()生成异步任务，这些异步任务将绑定在其所引用的runtime中，还可以block_on()或enter()进入其所引用的runtime，此外，还可以生成blocking thread。

```
let rt = Runtime::new().unwrap();
let handle = rt.handle();
handle.spawn(...)
handle.spawn_blocking(...)
handle.block_on(...)
handle.enter()
```

需注意，如果runtime已被关闭，handle也将失效，此后再使用handle，将panic。

2.1.11 理解多进程、多线程、多协程的并发能力

大家都说，多进程效率不如多线程，多线程效率又不如多协程。但这里面并不是如此简单的一句话就能描述准确的，还需要理解其中的真相。

如果有很多IO任务要执行，为了让这些IO操作不阻塞程序，可以使用多进程的方式将这些IO操作丢到【后台】去等待，然后通过各种进程间通信的方式来传递数据。但是进程间的上下文切换会带来较大的开销。因此，当程序使用多进程方式，且工作进程数量较多时，因为不断地进行进程间切换和内存拷贝，效率会明显下降。
比多进程更好一些的是多线程方式，线程是进程内部的执行单元，线程间的上下文切换的开销要远小于进程间切换的开销。因此，大概可以认为，多线程要优于多进程，如果单个进程内的线程数量较多，可以考虑引入多进程，然后在某些进程内使用多线程。

比多线程更好一些的是多协程方式，协程是线程内部的执行单元，协程的上下文切换开销，又要远小于线程间切换的开销。因此，大概可以认为，多协程要优于多线程，如果单个线程内的协程数量较多，可以考虑引入多线程，然后在某些线程内使用多协程。

但是，多进程效率并不真的差，多线程的效率也并不真的比多协程效率差。高并发能力的高低，完全取决于程序是否出现了等待、是否浪费了可调度单元(即进程、线程、协程)、是否浪费了更多的CPU。

一个简单的例子，假如要发送10W个HTTP请求，用多协程是最好的。为什么呢？因为HTTP请求是一个非常简单的IO任务，它只需要发送请求，然后等待。如果用多线程的并发模式，每个线程负责发送一个HTTP请求，那么每一个线程都将长时间处于等待状态，什么也不做，这是对线程的浪费，加之线程数量太多，在这么多的线程之间进行切换也会浪费大量CPU。因此，在这种情况下，多协程优于多线程。

另一方面，如果是要计算10W个密钥，应当去使用一定数量的多进程或多线程(少于或等于CPU核数)，以保证能尽量多地利用多核CPU。用多协程可能会很不好，因为协程调度会打断计算进度，当然这取决于协程调度器的调度逻辑。

从这两个简单又极端的示例可以大概理解，如果要执行的任务越简单(这里的简单表示的是计算密集程度低)，越IO密集，越应该使用粒度更小的可调度单元(即协程)。计算任务越重，越应该利用多核CPU。
更多时候，**一个任务里会同时带有IO和计算，无法简单地判断它是IO密集还是CPU密集的任务。这时候需要进行测试。**

2.1.12 选择单一线程runtime还是多线程runtime?

tokio提供了单一线程的runtime和多线程的runtime，虽然官方文档里时不时地提到【多数时候是多线程的runtime】，但并不意味着多线程的runtime优于单一线程的runtime，这取决于异步任务的工作类型。

简单来说，**「每一个异步任务都是一个线程内的【协程】，单一线程的runtime是在单个线程内调度管理这些任务，多线程runtime则是在多个线程内不断地分配和跨线程传递这些任务」。**

单一线程的runtime的优势在于它的任务调度开销低，因为它不需要进行开销更重的线程间切换，更不需要不断地在线程间传递数据。因此，对于计算程度不重的需求，它的高并发性能会很好。

单一线程的runtime的劣势在于这个runtime只能利用单核CPU，它无法利用多核CPU，也就无法发挥多核CPU的优势。

> 注：也可以认为，单一线程的runtime，和Python、Ruby等语言的并发是类似的，都是充分利用单核CPU。但却比它们更高效一些，一方面是语言本身的性能，另一方面是**tokio的worker thread都是绑定CPU的**，不会在不同的CPU核心之间进行切换，降低了切换开销。

但是，可以手动在多个线程内创建互相独立的单一线程runtime，这样也可以利用多核CPU。
ex2_1_12_1

```
use std::thread;
use tokio;
async fn hello_world(hi:&str) {
    println!("hello {}", hi);
}
fn main() {
    let t1 = thread::spawn(|| {
        let rt = tokio::runtime::Builder::new_current_thread().build().unwrap();
        let future = hello_world("t1");
        rt.block_on(future);
    });

    let t2 = thread::spawn(|| {
        let rt = tokio::runtime::Builder::new_current_thread().build().unwrap();
        let future = hello_world("t2");
        rt.block_on(future);
    });

    t1.join().unwrap();
    t2.join().unwrap();
}
```

这种手动创建多个单线程runtime的方式，可以利用多核CPU，但是这几个线程是不受控制的，完全由操作系统决定如何调度它们。这种方式是多线程runtime的雏形。它和多线程runtime的区别在于，多线程runtime会调度管理这些线程，会尽量以高效的方式来分配任务(比如从其它线程中偷任务)。**但是有了多线程，就有了额外的切换开销，就有了CPU利用率的浪费。**
因此，可以这样认为，**「单线程runtime对单个线程(单核CPU)的利用率，是高于多线程runtime的」**。
如果并发任务不重，单核CPU都无法跑满，显然单线程runtime要更优。如果并发任务中有较重的计算任务，则还需要再测试何种方式更优。

2.2 tokio核心: task

2.2.1 什么是tokio task

tokio官方手册tokio::task中用了一句话介绍task：Asynchronous green-threads(异步的绿色线程)。

- Rust中的原生线程(std::thread)是OS线程，每一个原生线程，都对应一个操作系统的线程。操作系统线程在内核层，由操作系统负责调度，缺点是涉及相关的系统调用，它有更重的线程上下文切换开销。
- **green thread则是用户空间的线程，由程序自身提供的调度器负责调度**，由于不涉及系统调用，同一个OS线程内的多个绿色线程之间的上下文切换的开销非常小，因此非常的轻量级。可以认为，它们就是一种特殊的协程。

什么是task呢?
每定义一个Future(例如一个async语句块就是一个Future)，**就定义了一个静止的尚未执行的task**，当它在runtime中开始运行的时候，它就是真正的task，一个真正的异步任务。

要注意，在tokio runtime中执行的并不都是异步任务，绑定在runtime中的可能是同步任务(例如一个数值计算就是一个同步任务，只是速度非常快，可忽略不计)，可能会长时间计算，可能会阻塞整个线程，这一点在前一篇介绍runtime时详细说明过。tokio严格区分异步任务和同步任务，只有异步任务才算是tokio task。tokio推荐的做法是将同步任务放入blocking thread中运行。
从官方手册将task描述为绿色线程也能理解，tokio::task只能是完全受tokio调度管理的异步任务，而不是脱离tokio调度控制的同步任务。

2.2.2 tokio::task

tokio::task模块本身提供了几个函数：

- **spawn：向runtime中添加新异步任务**
- spawn_blocking：生成一个blocking thread并执行指定的任务
- block_in_place：在某个worker thread中执行同步任务，但是会将同线程中的其它异步任务转移走，使得异步任务不会被同步任务饥饿
- **yield_now**: 立即放弃CPU，将线程交还给调度器，自己则进入就绪队列等待下一轮的调度
- unconstrained: 将指定的异步任务声明未不受限的异步任务，它将不受tokio的协作式调度，它将一直霸占当前线程直到任务完成，不会受到tokio调度器的管理
- **spawn_local**: 生成一个在当前线程内运行，一定不会被偷到其它线程中运行的异步任务

这里的三个spawn类的方法都返回JoinHandle类型，JoinHandle类型可以通过await来等待异步任务的完成，也可以通过abort()来中断异步任务，异步任务被中断后返回JoinError类型。

2.2.2.1 task::spawn()

就是直接在当前的runtime中生成一个异步任务。
ex2_2_2_1

```
use chrono::Local;
use std::thread;
use tokio::{self, task, runtime::Runtime, time};

fn now() -> String {
    Local::now().format("%F %T").to_string()
}

fn main() {
    let rt = Runtime::new().unwrap();
    let _guard = rt.enter();
    task::spawn(async {
        time::sleep(time::Duration::from_secs(3)).await;
        println!("task over: {}", now());
    });

    thread::sleep(time::Duration::from_secs(4));
}
```

2.2.2.2 task::spawn_blocking()

生成一个blocking thread来执行指定的任务。在前一篇介绍runtime的文章中已经解释清楚，这里不再解释。

```rust
let join = task::spawn_blocking(|| {
    // do some compute-heavy work or call synchronous code
    "blocking completed"
});

let result = join.await?;
assert_eq!(result, "blocking completed");
```

2.2.2.3 task::block_in_place()

block_in_place()的目的和spawn_blocking()类似。区别在于spawn_blocking()会新生成一个blocking thread来执行指定的任务，而block_in_place()是在当前**worker thread中执行指定的可能会长时间运行或长时间阻塞线程的任务**，但是它会先将该worker thread中已经存在的异步任务转移到其它worker thread，使得这些异步任务不会被饥饿。

1. **spawn_blocking 新建一个线程**
2. **block_in_place使用原来的worker thread，占用了worker thread，其他的异步任务放入到其他的work thread执行。**

显然，block_in_place()只应该在多线程runtime环境中运行，如果是单线程runtime，block_in_place会阻塞唯一的那个worker thread。

```rust
use tokio::task;

task::block_in_place(move || {
    // do some compute-heavy work or call synchronous code
});
```

在block_in_place内部，可以使用block_on()或enter()重新进入runtime环境。

```rust
use tokio::task;
use tokio::runtime::Handle;

task::block_in_place(move || {
    Handle::current().block_on(async move {
        // do something async
    });
});
```

2.2.2.4 task::yield_now

让当前任务立即放弃CPU，将worker thread交还给调度器，**任务自身则进入调度器（任务的调度器，不是*linux*线程的调度器）的就绪队列等待下次被轮询**调度。类似于其它异步系统中的next_tick行为。
需注意，调用yield_now()后还需await才立即放弃CPU，**因为yield_now本身是一个异步任务。**

```rust
use tokio::task;

async {
    task::spawn(async {
        // ...
        println!("spawned task done!")
    });

    // Yield, allowing the newly-spawned task to execute first.
    task::yield_now().await;
    println!("main task done!");
}
```

注意，yield后，任务调度的顺序是未知的。有可能任务在发出yield后，**紧跟着的下一轮调度会再次调度该任务。**

2.2.2.5 task::unconstrained()

不建议使用。

tokio的异步任务都是受tokio调度控制的，tokio采用协作式调度策略来调度它所管理的异步任务。当异步任务中的执行到了某个本该阻塞的操作时(即使用了tokio提供的那些原本会阻塞的API，例如tokio版本的sleep())，将不会阻塞当前线程，而是进入等待队列，等待Reactor接收事件通知来唤醒该异步任务，这样当前线程会被释放给调度器，使得调度器能够继续分配其它异步任务到该线程上执行。

task::unconstrained()则是创建一个不受限制不受调度器管理的异步任务，它将不会参与调度器的协作式调度，可以认为是将这个异步任务暂时脱离了调度管理。**这样一来，即便该任务中遇到了本该阻塞而放弃线程的操作，也不会去放弃，而是直接阻塞该线程。**

因此，unconstrained()创建的异步任务将会使得同线程的其它异步任务被饥饿。如果确实有这样的需求，建议使用block_in_place()或spawn_blocking()。

2.2.2.6 task::spawn_local()

关于spawn_local()，后面介绍LocalSet的时候再一起介绍。

2.2.3 取消任务

正在执行的异步任务可以随时被abort()取消，取消之后的任务返回JoinError类型。
ex2_2_3

```rust
use tokio::{self, runtime::Runtime, time};

fn main() {
    let rt = Runtime::new().unwrap();

    rt.block_on(async {
        let task = tokio::task::spawn(async {  // tokio::task::spawn加入任务
            println!("tokio::task::spawn sleep 10s");
            time::sleep(time::Duration::from_secs(10)).await; // 休眠的时候能偶被调度才能取消
            println!("tokio::task::spawn sleep finish");  // 这里没有继续执行
        });

        // 让上面的异步任务跑起来
```

```
        time::sleep(time::Duration::from_millis(1)).await;
        task.abort();  // 取消任务
        // 取消任务之后，可以取得JoinError
        let abort_err = task.await.unwrap_err(); // let abort_err: JoinError =
task.await.unwrap_err();
        println!("{}", abort_err.is_cancelled());
    })
}
```

打印：
tokio::task::spawn sleep 10s
true

如果异步任务已经完成，再对该任务执行abort()操作将没有任何效果。也就是说，没有JoinError，task.await.unwrap_err()将报错，而task.await.unwrap()则正常。

2.2.4 tokio::join!宏和tokio::try_join!宏

可以使用await去等待某个异步任务的完成，无论这个异步任务是正常完成还是被取消。
tokio提供了两个宏tokio::join!和tokio::try_join!。它们可以用于等待多个异步任务全部完成：

- join!必须等待所有任务完成
- try_join!要么等待所有异步任务正常完成，**要么等待第一个返回Result Err的任务出现**

另外，这两个宏都需要Future参数，它们将提供的各参数代表的任务封装成为一个大的task。

ex2_2_4

```
use chrono::Local;
use tokio::{self, runtime::Runtime, time};

fn now() -> String {
    Local::now().format("%F %T").to_string()
}

async fn do_one() {
  println!("doing one: {}", now());
  time::sleep(time::Duration::from_secs(2)).await;
  println!("do one done: {}", now());
}

async fn do_two() {
  println!("doing two: {}", now());
  time::sleep(time::Duration::from_secs(1)).await;
  println!("do two done: {}", now());
}

fn main() {
    let rt = Runtime::new().unwrap();
    rt.block_on(async {
      tokio::join!(do_one(), do_two());// 等待两个任务均完成，才继续向下执行代码
      println!("all done: {}", now());
    });
}
```

输出：

```
doing one: 2022-05-07 14:47:59
doing two: 2022-05-07 14:47:59
do two done: 2022-05-07 14:48:00
do one done: 2022-05-07 14:48:01
all done: 2022-05-07 14:48:01
```

下面是官方文档中try_join!的示例：

```
async fn do_stuff_async() -> Result<(), &'static str> {
    // async work
}

async fn more_async_work() -> Result<(), &'static str> {
    // more here
}

#[tokio::main]
async fn main() {
    let res = tokio::try_join!(do_stuff_async(), more_async_work());

    match res {
         Ok((first, second)) => {
             // do something with the values
         }
         Err(err) => {
            println!("processing failed; error = {}", err);
         }
    }
}
```

2.2.5 固定在线程内的本地异步任务: tokio::task::LocalSet

当使用多线程runtime时，tokio会协作式调度它管理的所有worker thread上的所有异步任务。例如某个worker thread空闲后可能会从其它worker thread中偷取一些异步任务来执行，或者tokio会主动将某些异步任务转移到不同的线程上执行。这意味着，异步任务可能会不受预料地被跨线程执行。

有时候并不想要跨线程执行。例如，那些没有实现Send的异步任务，它们不能跨线程，只能在一个固定的线程上执行。
tokio提供了让**某些任务固定在某一个线程中运行的功能，叫做LocalSet**，这些异步任务被放在一个独立的本地任务队列中，它们不会跨线程执行。

要使用**tokio::task::LocalSet**，需使用LocalSet::new()先创建好一个LocalSet实例，它将生成一个独立的任务队列用来存放本地异步任务。
之后，便可以使用LocalSet的spawn_local()向该队列中添加异步任务。但是，添加的异步任务不会直接执行，只有对LocalSet调用await或调用LocalSet::run_until()或LocalSet::block_on()的时候，才会开始运行本地队列中的异步任务。调用后两个方法会进入LocalSet的上下文环境。
例如，使用await来运行本地异步任务。

ex2_2_5_1

```
use chrono::Local;
use tokio::{self, runtime::Runtime, time};

fn now() -> String {
    Local::now().format("%F %T").to_string()
```

```
}

fn main() {
    let rt = Runtime::new().unwrap();
    let local_tasks = tokio::task::LocalSet::new();

    // 向本地任务队列中添加新的异步任务，但现在不会执行
    println!("add task1");
    local_tasks.spawn_local(async {
        println!("local task1");
        time::sleep(time::Duration::from_secs(5)).await;
        println!("local task1 done");
    });
    println!("add task2");
    local_tasks.spawn_local(async {
        println!("local task2");
        time::sleep(time::Duration::from_secs(5)).await;
        println!("local task2 done");
    });

    println!("before local tasks running: {}", now());
    rt.block_on(async {
        // 开始执行本地任务队列中的所有异步任务，并等待它们全部完成
        local_tasks.await;
    });
}
```

输出
add task1
add task2
before local tasks running: 2022-05-07 14:50:57
local task1
local task2
local task1 done
local task2 done

除了LocalSet::spawn_local()可以生成新的本地异步任务，**tokio::task::spawn_local()也可以生成新的本地异步任务，但是它的使用有个限制，必须在LocalSet上下文内部才能调用。**
例如ex2_2_5_2：

```
use chrono::Local;
use tokio::{self, runtime::Runtime, time};

fn now() -> String {
    Local::now().format("%F %T").to_string()
}

fn main() {
    let rt = Runtime::new().unwrap();
    let local_tasks = tokio::task::LocalSet::new();

    local_tasks.spawn_local(async {
        println!("local task1");
        time::sleep(time::Duration::from_secs(2)).await;
        println!("local task1 done");
    });
```

```
    local_tasks.spawn_local(async {
        println!("local task2");
        time::sleep(time::Duration::from_secs(3)).await;
        println!("local task2 done");
    });

    println!("before local tasks running: {}", now());
    // LocalSet::block_on进入LocalSet上下文
    local_tasks.block_on(&rt, async {
        tokio::task::spawn_local(async {
            println!("local task3");
            time::sleep(time::Duration::from_secs(4)).await;
            println!("local task3 done");
        }).await.unwrap();
    });
    println!("all local tasks done: {}", now());
}
```

需要注意的是，调用LocalSet::block_on()和LocalSet::run_until()时均需指定一个异步任务(Future)作为其参数，**它们都会立即开始执行该异步任务以及本地任务队列中已存在的任务**，但是这两个函数均只等待其参数对应的异步任务执行完成就返回。这意味着，它们返回的时候，**可能还有正在执行中的本地异步任务，它们会继续保留在本地任务队列中。当再次进入LocalSet上下文或await LocalSet的时候，它们会等待调度并运行。**

ex2_2_5_3

```
use chrono::Local;
use tokio::{self, runtime::Runtime, time};
use std::thread;
fn now() -> String {
    Local::now().format("%F %T").to_string()
}

fn main() {
    let rt = Runtime::new().unwrap();
    let local_tasks = tokio::task::LocalSet::new();

    local_tasks.spawn_local(async {
        println!("local task1");
        time::sleep(time::Duration::from_secs(2)).await;
        println!("local task1 done {}", now());
    });

    // task2要睡眠10秒，它将被第一次local_tasks.block_on在3秒后中断
    local_tasks.spawn_local(async {
        println!("local task2");
        time::sleep(time::Duration::from_secs(10)).await;
        println!("local task2 done, {}", now());
    });

    println!("before local tasks running: {}", now());
    local_tasks.block_on(&rt, async {
        tokio::task::spawn_local(async {
            println!("local task3");
            time::sleep(time::Duration::from_secs(3)).await;
            println!("local task3 done: {}", now());
```

```
        }).await.unwrap();
    });

    // 线程阻塞15秒，此时task2睡眠10秒的时间已经过去了，
    // 当再次进入LocalSet时，task2将可以直接被唤醒
    thread::sleep(std::time::Duration::from_secs(15));

    // 再次进入LocalSet
    local_tasks.block_on(&rt, async {
        // 先执行该任务，当遇到睡眠1秒的任务时，将出现任务切换，
        // 此时，调度器将调度task2，而此时task2已经睡眠完成
        println!("re enter localset context: {}", now());
        time::sleep(time::Duration::from_secs(1)).await;
        println!("re enter localset context done: {}", now());
    });
    println!("all local tasks done: {}", now());
}
```

输出结果：

```
before local tasks running: 2022-05-07 14:56:51
local task1
local task3
local task2
local task1 done 2022-05-07 14:56:53
local task3 done: 2022-05-07 14:56:54
re enter localset context: 2022-05-07 14:57:09
local task2 done, 2022-05-07 14:57:09
re enter localset context done: 2022-05-07 14:57:10
all local tasks done: 2022-05-07 14:57:10
```

需要注意的是，再次运行本地异步任务时，之前被中断的异步任务所等待的事件可能已经出现了，因此它们可能会被直接唤醒重新进入就绪队列等待下次轮询调度。正如上面需要睡眠10秒的task2，它会被第一次block_on中断，虽然task2已经不再执行，但是15秒之后它的睡眠完成事件已经出现，它可以在下次调度本地任务时直接被唤醒。但注意，唤醒的任务不是直接就可以被执行的，而是放入就绪队列等待调度。
这意味着，再次进入上下文时，所指定的Future中必须至少存在一个会引起调度切换的任务，否则该Future以同步的方式运行直到结束都不会给已经被唤醒的任务任何执行的机会。
例如，将上面示例中的第二个block_on中的Future参数换成下面的async代码块，task2将不会被调度执行：

```
local_tasks.block_on(&rt, async {
  println!("re-enter localset context, and exit context");
  println!("task2 will not be scheduled");
})
```

下面是使用run_until()两次进入LocalSet上下文的示例，和block_on()类似，区别仅在于它只能在Runtime::block_on()内或[tokio::main]注解的main函数内部被调用。
ex2_2_5_4

```
use chrono::Local;
use tokio::{self, runtime::Runtime, time};
use std::thread;
fn now() -> String {
    Local::now().format("%F %T").to_string()
```

```
}

fn main() {
    let rt = Runtime::new().unwrap();
    let local_tasks = tokio::task::LocalSet::new();

    local_tasks.spawn_local(async {
        println!("local task1");
        time::sleep(time::Duration::from_secs(5)).await;
        println!("local task1 done {}", now());
    });

    println!("before local tasks running: {}", now());
    rt.block_on(async {
        local_tasks
            .run_until(async {
                println!("local task2");
                time::sleep(time::Duration::from_secs(3)).await;
                println!("local task2 done: {}", now());
            })
            .await;
    });

    thread::sleep(std::time::Duration::from_secs(10));
    rt.block_on(async {
        local_tasks
            .run_until(async {
                println!("local task3");
                tokio::task::yield_now().await;
                println!("local task3 done: {}", now());
            })
            .await;
    });
    println!("all local tasks done: {}", now());
}
```

输出结果：

```
before local tasks running: 2022-05-07 14:58:19
local task2
local task1
local task2 done: 2022-05-07 14:58:22
local task3
local task1 done 2022-05-07 14:58:32
local task3 done: 2022-05-07 14:58:32
all local tasks done: 2022-05-07 14:58:32
```

2.2.6 tokio::select!宏

select!宏的作用是轮询指定的多个异步任务，每个异步任务都是select!的一个分支，当某个分支已完成，则执行该分支对应的代码，同时取消其它分支。简单来说，**select!的作用是等待第一个完成的异步任务并执行对应任务完成后的操作。**
它的使用语法参考如下：

```
tokio::select! {
  <pattern1> = <async expression 1> (, if <precondition1>)? => <handler1>,    //
branch 1
  <pattern2> = <async expression 2> (, if <precondition2>)? => <handler2>,    //
branch 2
  ...
  (else => <handler_else>)?
};
```

else分支是可选的，每个分支的if前置条件是可选的。因此，简化的语法为：

```
tokio::select! {
  <pattern1> = <async expression 1> => <handler1>,    // branch 1
  <pattern2> = <async expression 2> => <handler2>,    // branch 2
  ...
};
```

即，每个分支都有一个异步任务，并对异步任务完成后的返回结果进行模式匹配，如果匹配成功，则执行对应的handler。
一个简单的示例： ex2_2_6_1

```
use tokio::{self, runtime::Runtime, time::{self, Duration}};

async fn sleep(n: u64) -> u64 {
    time::sleep(Duration::from_secs(n)).await;
    n
}

fn main() {
    let rt = Runtime::new().unwrap();
    rt.block_on(async {
        tokio::select! {
          v = sleep(5) => println!("sleep 5 secs, branch 1 done: {}", v),
          v = sleep(3) => println!("sleep 3 secs, branch 2 done: {}", v),
        };

        println!("select! done");
    });
}
```

输出结果：

```
sleep 3 secs, branch 2 done: 3
select! done
```

注意，select!本身是【阻塞】的，只有select!执行完，它后面的代码才会继续执行。

每个分支可以有一个if前置条件，当if前置条件为false时，对应的分支将被select!忽略(禁用)，但该分支的异步任务仍然会执行，只不过select!不再轮询它(即不再推进异步任务的执行)。
下面是官方手册对select!工作流程的描述：

1. 评估所有分支中存在的if前置条件，如果某个分支的前置条件返回false，则禁用该分支。注意，循环(如loop)时，每一轮执行的select!都会清除分支的禁用标记

2. 收集所有分支中的异步表达式(包括已被禁用的分支)，并在**「同一个线程」**中推进所有未被禁用的异步任务执行，然后等待
3. 当某个分支的异步任务完成，将该异步任务的返回值与对应分支的模式进行匹配，如果匹配成功，则执行对应分支的handler，如果匹配失败，则禁用该分支，本次select!调用不会再考虑该分支。如果匹配失败，则重新等待下一个异步任务的完成
4. 如果所有分支最终都被禁用，则执行else分支，如果不存在else分支，则panic

默认情况下，select!会伪随机公平地轮询每一个分支，如果确实需要让select!按照任务书写顺序去轮询，可以在select!中使用biased。
例如，官方手册提供了一个很好的例子：ex2_2_6_2

```
#[tokio::main]
async fn main() {
    let mut count = 0u8;
    loop {
        tokio::select! {
            // 如果取消biased，挑选的任务顺序将随机，可能会导致分支中的断言失败
            biased;
            _ = async {}, if count < 1 => { count += 1; println!("< 1");
assert_eq!(count, 1); }
            _ = async {}, if count < 2 => { count += 1; println!("<
2");assert_eq!(count, 2); }
            _ = async {}, if count < 3 => { count += 1; println!("<
3");assert_eq!(count, 3); }
            _ = async {}, if count < 4 => { count += 1; println!("<
4");assert_eq!(count, 4); }
            else => { break; }
        };
    }
}
```

打印

```
< 1
< 2
< 3
< 4
```

3 理解计时器timer

每一个异步框架都应该具备计时器功能，tokio的计时器功能在开启了time特性后可用。

```
tokio = {version = "1", features = ["rt", "rt-multi-thread", "time"]}
```

tokio的time模块包含几个功能：

- Duration类型：是对std::time::Duration的重新导出，两者等价。它用于描述持续时长，例如睡眠3秒的3秒是一个时长，每隔3秒的3秒也是一个时长
- Instant类型：从程序运行开始就单调递增的时间点，仅结合Duration一起使用。例如，此刻是处在某个时间点A，下一次(例如某个时长过后)，处在另一个时间点B，时间点B一定不会早于时间点A，即便修改了操作系统的时钟或硬件时钟，它也不会时光倒流的现象
- Sleep类型：是一个Future，通过调用sleep()或sleep_until()返回，该Future本身不做任何事，它只在到达某个时间点(Instant)时完成

- Interval类型：是一个流式的间隔计时器，通过调用interval()或interval_at()返回。Interval使用Duration来初始化，表示每隔一段时间(即指定的Duration时长)后就产生一个值
- Timeout类型：封装异步任务，并为异步任务设置超时时长，通过调用timeout()或timeout_at()返回。如果异步任务在指定时长内仍未完成，则异步任务被强制取消并返回Error

3.1 时长: tokio::time::Duration

tokio::time::Duration是对std::time::Duration的Re-exports，它两完全等价，因此可在tokio上下文中使用任何一种Duration。
Duration类型描述了一种时长，该结构包含两部分：秒和纳秒。

```
pub struct Duration {
    secs: u64,
    nanos: u32,
}
```

可使用Duration::new(Sec, Nano_sec)来构建Duration。例如，Duration::new(5, 30)构建了一个5秒30纳秒的时长，即总共5_000_000_030纳秒。
如果Nano_sec部分超出了纳秒范围(1秒等于10亿纳秒)，将进位到秒单位上，例如第二个参数指定为500亿纳秒，那么会向秒部分加50秒。

注意，构建时长时，这两部分的值可能会超出范围，例如计算后的秒部分的值超出了u64的范围，或者计算得到了负数。对此，Duration提供了几种不同的处理方式。

特殊地，如果两个参数都指定为0，那么表示时长为0，可用is_zero()来检测某个Duration是否是0时长。0时长可用于上下文切换(yield)，例如sleep睡眠0秒，表示不用睡眠，但会交出CPU使得发生上下文切换。
还可以使用如下几种简便的方式构建各种单位的时长：

- Duration::from_secs(3)：3秒时长
- Duration::from_millis(300)：300毫秒时长
- Duration::from_micros(300)：300微秒时长
- Duration::from_nanos(300)：300纳秒时长
- Duration::from_secs_f32(2.3)：2.3秒时长
- Duration::from_secs_f64(2.3)：2.3秒时长

对于构建好的Duration实例dur = Duration::from_secs_f32(2.3)，可以使用如下几种方法方便地提取、转换它的秒、毫秒、微秒、纳秒。

- dur.as_secs()：转换为秒的表示方式，2
- dur.as_millis(): 转换为毫秒表示方式，2300
- dur.as_micros(): 转换为微秒表示方式，2_300_000
- dur.as_nanos(): 转换为纳秒表示方式，2_300_000_000
- dur.as_secs_f32(): 小数秒表示方式，2.3
- dur.as_secs_f64(): 小数秒表示方式，2.3
- dur.subsec_millis(): 小数部分转换为毫秒精度的表示方式，300
- dur.subsec_micros(): 小数部分转换为微秒精度的表示方式，300_000
- dur.subsec_nanos(): 小数部分转换为纳秒精度的表示方式，300_000_000

Duration实例可以直接进行大小比较以及加减乘除运算：

- checked_add(): 时长的加法运算，超出Duration范围时返回None
- checked_sub(): 时长的减法运算，超出Duration范围时返回None
- checked_mul(): 时长的乘法运算，超出Duration范围时返回None
- checked_div(): 时长的除法运算，超出Duration范围时(即分母为0)返回None
- saturating_add()：饱和式的加法运算，超出范围时返回Duration支持的最大时长

- saturating_mul()：饱和式的乘法运算，超出范围时返回Duration支持的最大时长
- saturating_sub()：饱和式的减法运算，超出范围时返回0时长
- mul_f32()：时长乘以小数，得到的结果如果超出范围或无效，则panic
- mul_f64()：时长乘以小数，得到的结果如果超出范围或无效，则panic
- div_f32()：时长除以小数，得到的结果如果超出范围或无效，则panic
- div_f64()：时长除以小数，得到的结果如果超出范围或无效，则panic

3.2 时间点: tokio::time::Instant

Instant用于表示时间点，主要用于两个时间点的比较和相关运算。
tokio::time::Instant是对std::time::Instant的封装，添加了一些对齐功能，使其能够适用于tokio runtime。
Instant是严格单调递增的，绝不会出现时光倒流的现象，即之后的时间点一定晚于之前创建的时间点。但是，tokio time提供了pause()函数可暂停时间点，还提供了advance()函数用于向后跳转到某个时间点。

tokio::time::Instant::now()用于创建代表此时此刻的时间点。Instant可以直接进行大小比较，还能执行+、-操作。
ex3_2

```
use tokio;
use tokio::time::Instant;
use tokio::time::Duration;

#[tokio::main]
async fn main() {
    // 创建代表此时此刻的时间点
    let now = Instant::now();

    // Instant 加一个Duration，得到另一个Instant
    let next_3_sec = now + Duration::from_secs(3);
    // Instant之间的大小比较
    println!("{}", now < next_3_sec);  // true

    // Instant减Duration，得到另一个Instant
    let new_instant = next_3_sec - Duration::from_secs(2);

    // Instant减另一个Instant，得到Duration
    // 注意，Duration有它的有效范围，因此必须是大的Instant减小的Instant，反之将panic
    let duration = next_3_sec - new_instant;
}
```

打印

```
true
```

此外tokio::time::Instant还有以下几个方法：

- from_std(): 将std::time::Instant转换为tokio::time::Instant
- into_std(): 将tokio::time::Instant转换为std::time::Instant
- elapsed(): 指定的时间点实例，距离此时此刻的时间点，已经过去了多久(返回Duration)
- duration_since(): 两个Instant实例之间相差的时长，要求B.duration_since(A)中的B必须晚于A，否则panic
- checked_duration_since(): 两个时间点之间的时长差，如果计算返回的Duration无效，则返回None

- saturating_duration_since(): 两个时间点之间的时长差，如果计算返回的Duration无效，则返回0时长的Duration实例
- checked_add(): 为时间点加上某个时长，如果加上时长后是无效的Instant，则返回None
- checked_sub(): 为时间点减去某个时长，如果减去时长后是无效的Instant，则返回None

tokio顶层也提供了一个tokio::resume()方法，功能类似于tokio::time::from_std()，都是将std::time::Instant::now()保存为tokio::time::Instant。不同的是，后者用于创建tokio time Instant时间点，而resume()是让tokio的Instant的计时系统与系统的计时系统进行一次同步更新。

3.3 睡眠: tokio::time::Sleep

tokio::time::sleep()和tokio::time::sleep_until()提供tokio版本的睡眠任务：
ex3_3_1

```
use tokio::{self, runtime::Runtime, time};

fn main(){
    let rt = Runtime::new().unwrap();
    rt.block_on(async {
        // 睡眠2秒
        time::sleep(time::Duration::from_secs(2)).await;

        // 一直睡眠，睡到2秒后醒来
        time::sleep_until(time::Instant::now() +
time::Duration::from_secs(2)).await;
    });
}
```

注意，std::thread::sleep()会阻塞当前线程，而tokio的睡眠不会阻塞当前线程，实际上tokio的睡眠在进入睡眠后不做任何事，仅仅只是立即放弃CPU，并进入任务轮询队列，等待睡眠时间终点到了之后被Reactor唤醒，然后进入就绪队列等待被调度。
可以简单理解异步睡眠：**调用睡眠后，记录睡眠的终点时间点，之后在轮询到该任务时，比较当前时间点是否已经超过睡眠终点，如果超过了，则唤醒该睡眠任务，如果未超过终点，则不管。**
注意，tokio的sleep的睡眠精度是毫秒，因此无法保证也不应睡眠更低精度的时间。例如不要睡眠100微秒或100纳秒，这时无法保证睡眠的时长。
下面是一个睡眠10微秒的例子，多次执行，会发现基本上都要1毫秒多，去掉执行指令的时间，实际的睡眠时长大概是1毫秒。另外，将睡眠10微秒改成睡眠100微秒或1纳秒，结果也是接近的。
ex3_3_2

```
use tokio::{self, runtime::Runtime, time};

fn main() {
    let rt = Runtime::new().unwrap();
    rt.block_on(async {
        let start = time::Instant::now();
        // time::sleep(time::Duration::from_nanos(100)).await;
        // time::sleep(time::Duration::from_micros(100)).await;
        time::sleep(time::Duration::from_micros(10)).await;
        println!("sleep {}",
time::Instant::now().duration_since(start).as_nanos());
    });
}
```

执行的多次，输出结果：
sleep 2668939
sleep 1456944
sleep 1161200
sleep 1393200
sleep 1306400
sleep 1285300

sleep()或sleep_until()都返回time::Sleep类型，它有3个方法可调用：

- deadline(): 返回Instant，表示该睡眠任务的睡眠终点
- is_elapsed(): 可判断此时此刻是否已经超过了该sleep任务的睡眠终点
- reset()：可用于重置睡眠任务。如果睡眠任务未完成，则直接修改睡眠终点，如果睡眠任务已经完成，则再次创建睡眠任务，等待新的终点

需要注意的是，reset()要求修改睡眠终点，因此Sleep实例需要是mut的，但这样会消费掉Sleep实例，更友好的方式是使用tokio::pin!(sleep)将sleep给pin在当前栈中，这样就可以调用as_mut()方法获取它的可修改版本。
ex3_3_3

```
use chrono::Local;
use tokio::{self, runtime::Runtime, time};

#[allow(dead_code)]
fn now() -> String {
    Local::now().format("%F %T").to_string()
}

fn main() {
    let rt = Runtime::new().unwrap();
    rt.block_on(async {
        println!("start: {}", now());
        let slp = time::sleep(time::Duration::from_secs(1));
        tokio::pin!(slp);

        slp.as_mut().reset(time::Instant::now() + time::Duration::from_secs(2));

        slp.await;
        println!("end: {}", now());
    });
}
```

输出：
start: 2022-05-07 15:12:06
end: 2022-05-07 15:12:08

重置已完成的睡眠实例：
ex3_3_4

```
use chrono::Local;
use tokio::{self, runtime::Runtime, time};

#[allow(dead_code)]
fn now() -> String {
    Local::now().format("%F %T").to_string()
}
```

```rust
fn main() {
    let rt = Runtime::new().unwrap();
    rt.block_on(async {
        println!("start: {}", now());
        let slp = time::sleep(time::Duration::from_secs(1));
        tokio::pin!(slp);

        //注意调用slp.as_mut().await，而不是slp.await，后者会move消费掉slp
        slp.as_mut().await;
        println!("end 1: {}", now());

        slp.as_mut().reset(time::Instant::now() + time::Duration::from_secs(2));

        slp.await;
        println!("end 2: {}", now());
    });
}
```

输出结果：
start: 2022-05-07 15:12:57
end 1: 2022-05-07 15:12:58
end 2: 2022-05-07 15:13:00

3.4 任务超时: tokio::time::Timeout

tokio::time::timeout()或tokio::time::timeout_at()可设置一个异步任务的完成超时时间，前者接收一个Duration和一个Future作为参数，后者接收一个Instant和一个Future作为参数。这两个函数封装异步任务之后，返回time::Timeout，它也是一个Future。
如果在指定的超时时间内该异步任务已完成，则返回该异步任务的返回值，如果未完成，则异步任务被撤销并返回Err。
ex3_4

```rust
use chrono::Local;
use tokio::{self, runtime::Runtime, time};

fn now() -> String {
    Local::now().format("%F %T").to_string()
}

fn main() {
    let rt = Runtime::new().unwrap();
    rt.block_on(async {
        let res = time::timeout(time::Duration::from_secs(5), async {
            println!("sleeping: {}", now());
            time::sleep(time::Duration::from_secs(6)).await;
            33
        });

        match res.await {
            Err(_) => println!("task timeout: {}", now()),
            Ok(data) => println!("get the res '{}': {}", data, now()),
        };
    });
}
```

得到结果：
sleeping: 2022-05-07 15:14:54
task timeout: 2022-05-07 15:14:59

如果将睡眠6秒改为睡眠4秒，那么将得到结果：
sleeping: 2022-05-07 15:14:54
get the res '33': 2022-05-07 15:14:58

得到time::Timeout实例res后，可以通过res.get_ref()或者res.get_mut()获得Timeout所封装的Future的可变和不可变引用，使用res.into_inner()获得所封装的Future，这会消费掉该Future。
如果要取消Timeout的计时等待，直接删除掉Timeout实例即可。

3.5 间隔任务: tokio::time::Interval

tokio::time::interval()和tokio::time::interval_at()用于设置间隔性的任务。

- interval_at(): 接收一个Instant参数和一个Duration参数，Instant参数表示间隔计时器的开始计时点，Duration参数表示间隔的时长
- interval(): 接收一个Duration参数，它在第一次被调用的时候立即开始计时

注意，这两个函数只是定义了间隔计时器的起始计时点和间隔的时长，要真正开始让它开始计时，还需要调用它的tick()方法生成一个Future任务，并调用await来执行并等待该任务的完成。
例如，5秒后开始每隔1秒执行一次输出操作：
ex3_5 _1

```
use chrono::Local;
use tokio::{self, runtime::Runtime, time::{self, Duration, Instant}};

fn now() -> String {
    Local::now().format("%F %T").to_string()
}

fn main() {
    let rt = Runtime::new().unwrap();
    rt.block_on(async {
        println!("before: {}", now());

        // 计时器的起始计时点：此时此刻之后的5秒后
        let start = Instant::now() + Duration::from_secs(5);
        let interval = Duration::from_secs(1);
        let mut intv = time::interval_at(start, interval);

        // 该计时任务"阻塞"，直到5秒后被唤醒
        intv.tick().await;
        println!("task 1: {}", now());

        // 该计时任务"阻塞"，直到1秒后被唤醒
        intv.tick().await;
        println!("task 2: {}", now());

        // 该计时任务"阻塞"，直到1秒后被唤醒
        intv.tick().await;
        println!("task 3: {}", now());
    });
}
```

输出结果：
before: 2022-05-07 15:16:32
task 1: 2022-05-07 15:16:37
task 2: 2022-05-07 15:16:38
task 3: 2022-05-07 15:16:39

上面定义的计时器，有几点需要说明清楚：

1. interval_at()第一个参数定义的是计时器的开始时间，这样描述不准确，它表述的是最早都要等到这个时间点才开始计时。例如，定义计时器5秒之后开始计时，但在第一次tick()之前，先睡眠了10秒，那么该计时器将在10秒后才开始，但如果第一次tick之前只睡眠了3秒，那么还需再等待2秒该tick计时任务才会完成。
2. 定义计时器时，要将其句柄(即计时器变量)声明为mut，因为每次tick时，都需要修改计时器内部的下一次计时起点。
3. 不像其它语言中的间隔计时器，tokio的间隔计时器需要手动调用tick()方法来生成临时的异步任务。
4. 删除计时器句柄可取消间隔计时器。

再看下面的示例，定义5秒后开始的计时器，但在第一次开始计时前，先睡眠10秒。
ex3_5_2

```
use chrono::Local;
use tokio::{
    self,
    runtime::Runtime,
    time::{self, Duration, Instant},
};

fn now() -> String {
    Local::now().format("%F %T").to_string()
}

fn main() {
    let rt = Runtime::new().unwrap();
    rt.block_on(async {
        println!("before: {}", now());

        let start = Instant::now() + Duration::from_secs(5);
        let interval = Duration::from_secs(1);
        let mut intv = time::interval_at(start, interval);

        time::sleep(Duration::from_secs(10)).await;
        intv.tick().await;
        println!("task 1: {}", now());
        intv.tick().await;
        println!("task 2: {}", now());
    });
}
```

输出结果：
before: 2022-05-07 15:17:24
task 1: 2022-05-07 15:17:34
task 2: 2022-05-07 15:17:34

注意输出结果中的task 1和task 2的时间点是相同的，说明第一次tick之后，并没有等待1秒之后再执行紧跟着的tick，而是立即执行之。
简单解释一下上面示例中的计时器内部的工作流程，假设定义计时器的时间点是19:00:10：

- 定义5秒后开始的计时器intv，该计时器内部有一个字段记录着下一次开始tick的时间点，其值为19:00:15
- 睡眠10秒后，时间点到了19:00:20，此时第一次执行intv.tick()，它将生成一个异步任务，执行器执行时发现此时此刻的时间点已经超过该计时器内部记录的值，于是该异步任务立即完成并进入就绪队列等待调度，同时修改计时器内部的值为19:00:16
- 下一次执行tick的时候，此时此刻仍然是19:00:20，已经超过了该计时器内部的19:00:16，因此计时任务立即完成

这是tokio Interval在遇到计时延迟时的默认计时策略，叫做Burst。tokio支持三种延迟后的计时策略。可使用set_missed_tick_behavior(MissedTickBehavior)来修改计时策略。
「1.Burst策略，冲刺型的计时策略，当出现延迟后，将尽量快地完成接下来的tick，直到某个tick赶上它正常的计时时间点」。
例如，5秒后开始的每隔1秒的计时器，第一次开始tick前睡眠了10秒，那么10秒后将立即进行如下几次tick，或者说瞬间完成如下几次tick：

- 第一次tick，它本该在第五秒的时候被执行
- 第二次tick，它本该在第六秒的时候被执行
- 第三次tick，它本该在第七秒的时候被执行
- 第四次tick，它本该在第八秒的时候被执行
- 第五次tick，它本该在第九秒的时候被执行
- 第六次tick，它本该在第十秒的时候被执行

而第七次tick的时间点，将回归正常，即在第十一秒的时候开始被执行。
「2.Delay策略，延迟性的计时策略，当出现延迟后，仍然按部就班地每隔指定的时长计时」。在内部，这种策略是在每次执行tick之后，都修改下一次计时起点为Instant::now() + Duration。因此，这种策略下的任何相邻两次的tick，其中间间隔的时长都至少达到Duration。
例如：ex3_5_3

```
use chrono::Local;
use tokio::{self, runtime::Runtime};
use tokio::time::{self, Duration, Instant, MissedTickBehavior};

fn now() -> String {
    Local::now().format("%F %T").to_string()
}

fn main() {
    let rt = Runtime::new().unwrap();
    rt.block_on(async {
        println!("before: {}", now());

        let mut intv = time::interval_at(
            Instant::now() + Duration::from_secs(5),
            Duration::from_secs(2),
        );
        intv.set_missed_tick_behavior(MissedTickBehavior::Delay);

        time::sleep(Duration::from_secs(10)).await;

        println!("start: {}", now());
        intv.tick().await;
        println!("tick 1: {}", now());
```

```
        intv.tick().await;
        println!("tick 2: {}", now());
        intv.tick().await;
        println!("tick 3: {}", now());
    });
}
```

输出结果：
before: 2022-05-07 15:18:42
start: 2022-05-07 15:18:52
tick 1: 2022-05-07 15:18:52
tick 2: 2022-05-07 15:18:54
tick 3: 2022-05-07 15:18:56

「3.Skip策略，忽略型的计时策略，当出现延迟后，仍然所有已经被延迟的计时任务」。这种策略总是以定义计时器时的起点为基准，类似等差数量，每一次执行tick的时间点，一定符合Start + N * Duration。

ex3_5_4

```
use chrono::Local;
use tokio::{self, runtime::Runtime};
use tokio::time::{self, Duration, Instant, MissedTickBehavior};

fn now() -> String {
    Local::now().format("%F %T").to_string()
}

fn main() {
    let rt = Runtime::new().unwrap();
    rt.block_on(async {
        println!("before: {}", now());

        let mut intv = time::interval_at(
            Instant::now() + Duration::from_secs(5),
            Duration::from_secs(2),
        );
        intv.set_missed_tick_behavior(MissedTickBehavior::Skip);

        time::sleep(Duration::from_secs(10)).await;

        println!("start: {}", now());
        intv.tick().await;
        println!("tick 1: {}", now());
        intv.tick().await;
        println!("tick 2: {}", now());
        intv.tick().await;
        println!("tick 3: {}", now());
    });
}
```

输出结果：
before: 2022-05-07 15:19:41
start: 2022-05-07 15:19:51
tick 1: 2022-05-07 15:19:51

tick 2: 2022-05-07 15:19:52
tick 3: 2022-05-07 15:19:54

注意上面的输出结果中，第一次tick和第二次tick只相差1秒而不是相差2秒。
上面通过interval_at()解释清楚了tokio::time::Interval的三种计时策略。但在程序中，更大的可能是使用interval()来定义间隔计时器，它等价于interval_at(Instant::now() + Duration)，表示计时起点从现在开始的计时器。
此外，可以使用period()方法获取计时器的间隔时长，使用missed_tick_behavior()获取当前的计时策略。

4 tokio task的通信和同步

4.1 简介

通常来说，对于允许并发多执行分支的内核或引擎来说，都需要提供对应的通信机制和同步机制。
例如，多进程之间，有进程间通信方式，比如管道、套接字、共享内存、消息队列等，还有进程间的同步机制，例如信号量、文件锁、条件变量等。多线程之间，也有线程间通信方式，简单粗暴的是直接共享同进程内存，同步机制则有互斥锁、条件变量等。
tokio提供了异步多任务的并发能力，它也需要提供异步任务之间的通信方式和同步机制。
在介绍它们之前，需要先开启tokio的同步功能。
tokio = {version = "1", features = ["rt", "sync", "rt-multi-thread"]}

tokio = {version = "1", features = ["full"]}

4.1.1 sync模块功能简介

sync模块主要包含两部分功能：异步任务之间的通信模块以及异步任务之间的状态同步模块。

任务间通信

tokio的异步任务之间主要采用**消息传递**(message passing)的通信方式，即某个异步任务负责发消息，另一个异步任务收消息。这种通信方式的最大优点是避免并发任务之间的数据共享，消灭数据竞争，使得代码更加安全，更加容易维护。
消息传递通常使用通道(channel)来进行通信。tokio提供几种不同功能的通道：

- oneshot通道: 一对一发送的一次性通道，即该通道只能由一个发送者(Sender)发送最多一个数据，且只有一个接收者(Receiver)接收数据
- mpsc通道: 多对一发送，即该通道可以同时有多个发送者向该通道发数据，**但只有一个接收者接收数据** (Arc)
- broadcast通道: 多对多发送，即该通道可以同时有多个发送者向该通道发送数据，也可以有多个接收者接收数据
- watch通道: 一对多发送，即该通道只能有一个发送者向该通道发送数据，但可以有多个接收者接收数据

不同类型的通道，用于解决不同场景的需求。通常来说，**最常用的是mpsc类型的通道。**

任务间状态同步

在编写异步任务的并发代码时，很多时候需要去检测任务之间的状态。比如任务A需要等待异步任务B执行完某个操作后才允许向下执行。
比较原始的解决方式是直接用代码去轮询判断状态是否达成。但在异步编程过程中，这类状态检测的需求非常普遍，因此异步框架会提供一些内置在框架中的同步原语。同步原语封装了各种状态判断、状态等待的轮询操作，这使得编写任务状态同步的代码变得更加简单直接。
通常来说，有以下几种基本的同步原语，这些也是tokio所提供的：

- Mutex: 互斥锁，任务要执行某些操作时，必须先申请锁，只有申请到锁之后才能执行操作，否则就等待
- RwLock: 读写锁，类似于互斥锁，但粒度更细，区分读操作和写操作，可以同时存在多个读操作，但写操作必须独占锁资源
- Notify: 任务通知，用于唤醒正在等待的任务，使其进入就绪态等待调度
- Barrier: 屏障，**多个任务**在某个屏障处互相等待，只有这些任务都达到了那个屏障点，这些任务才都继续向下执行
- Semaphore: 信号量(信号灯)，限制同时执行的任务数量，例如限制最多只有20个线程(或tokio的异步任务)同时执行

4.2 通信

tokio使用通道在task之间进行通信，有四种类型通道：oneshot、mpsc、broadcast和watch。

4.2.1 oneshot通道

oneshot通道的特性是：单Sender、单Receiver以及单消息，简单来说就是**一次性的通道。**
oneshot通道的创建方式是使用oneshot::channel()方法：

```
pub fn channel<T>() -> (Sender<T>, Receiver<T>)
```

它返回该通道的写端sender和读端receiver，其中泛型T表示的是读写两端所传递的消息类型。
例如，创建一个可发送i32数据的一次性通道：

```
let (tx, rx) = oneshot::channel::<i32>();
```

返回的结果中，tx是发送者(sender)、rx是接收者(receiver)。
多数时候不需要去声明通道的类型，编译器可以根据发送数据时的类型自动推断出类型。

```
let (tx, rx) = oneshot::channel();
```

Sender

Sender通过send()方法发送数据，因为oneshot通道只能发送一次数据，所以send()发送数据的时候，tx直接被消费掉。Sender并不一定总能成功发送消息，比如，Sender发送消息之前，Receiver端就已经关闭了读端。因此send()返回Result结果：如果发送成功，则返回Ok(())，如果发送失败，则返回Err(T)。
因此，发送数据的时候，通常会做如下检测：

```
// 或 if tx.send(33).is_err() {}
// 或直接忽略错误 let _ = tx.send();
if let Err(_) = tx.send(33) {
  println!("receiver closed");
}
```

另外需注意，send()是非异步但却不阻塞的，它总是立即返回，**如果能发送数据，则发送数据，如果不能发送数据，就返回错误，**它不会等待Receiver启动读取操作。也因此，send()可以应用在同步代码中，也可以应用在异步代码中。
Sender可以通过is_closed()方法来判断Receiver端是否已经关闭。
Sender可以通过close()方法来等待Receiver端关闭。它可以结合select!宏使用：其中一个分支计算要发送的数据，另一个分支为closed()等待分支，如果先计算完成，则发送计算结果，而如果是先等到了对端closed的异步任务完成，则无需再计算浪费CPU去计算结果。例如：

```
#![allow(unused)]
fn main() {
tokio::spawn(async move {
  tokio::select! {
    _ = tx.closed() => {
      // 先等待到了对端关闭，不做任何事，select!会自动取消其它分支的任务
    }
    value = compute() => {
      // 先计算得到结果，则发送给对端
      // 但有可能刚计算完成，尚未发送时，对端刚好关闭，因此可能发送失败
      // 此处丢弃发送失败的错误
      let _ = tx.send(value);
    }
  }
});
}
```

Receiver

Receiver没有recv()方法，rx本身实现了Future Trait，它执行时对应的异步任务就是接收数据，因此只需await即可用来接收数据。
但是，接收数据并不一定会接收成功。例如，Sender端尚未发送任何数据就已经关闭了(被drop)，此时Receiver端会接收到error::RecvError错误。因此，接收数据的时候通常也会进行判断：

```
match rx.await {
  Ok(v) => println!("got = {:?}", v),
  Err(_) => println!("the sender dropped"),
  // Err(e: RecvError) => xxx,
}
```

既然通过rx.await来接收数据，那么已经隐含了一个信息，异步任务中接收数据时会进行等待。
Receiver端可以通过close()方法关闭自己这一端，当然也可以直接drop来关闭。关闭操作是幂等的，即，如果关闭的是已经关闭的Recv，则不产生任何影响。
关闭Recv端之后，可以保证Sender端无法再发送消息。但需要注意，有可能Recv端关闭完成之前，Sender端正好在这时发送了一个数据过来。因此，在关闭Recv端之后，尽可能地再调用一下try_recv()方法尝试接收一次数据。
try_recv()方法返回三种可能值：

- Ok(T): 表示成功接收到通道中的数据
- Err(TryRecvError::Empty): 表示通道为空
- Err(TryRecvError::Closed): 表示通道为空，且Sender端已关闭，即Sender未发送任何数据就关闭了

例如：

```rust
let (tx, mut rx) = oneshot::channel::<()>();

drop(tx);

match rx.try_recv() {
    // The channel will never receive a value.
    Err(TryRecvError::Closed) => {}
    _ => unreachable!(),
}
```

使用示例

一个完整但简单的示例：
ex4_2_1_1

```rust
use tokio::{self, runtime::Runtime, sync};

fn main() {
    let rt = Runtime::new().unwrap();
    rt.block_on(async {
        let (tx, rx) = sync::oneshot::channel();

        tokio::spawn(async move {
            if tx.send(33).is_err() {
                println!("receiver dropped");
            }
        });

        match rx.await {
            Ok(value) => println!("received: {:?}", value),
            Err(_) => println!("sender dropped"),
        };
    });
}
```

4.2.2 mpsc通道

mpsc通道的特性是可以有**多个发送者发送多个消息，且只有一个接收者。**mpsc通道是使用最频繁的通道类型。
mpsc通道分为两种：

- **bounded channel:** 有界通道，通道有容量限制，即通道中最多可以存放指定数量(至少为1)的消息，通过mpsc::channel()创建
- unbounded channel: 无界通道，**通道中可以无限存放消息，直到内存耗尽**，通过mpsc::unbounded_channel()创建

有界通道

通过mpsc::channel()创建有界通道，需传递一个大于1的usize值作为其参数。
例如，创建一个最多可以存放100个消息的有界通道。

```
// tx是Sender端，rx是Receiver端
// 接收端接收数据时需修改状态，因此声明为mut
let (tx, mut rx) = mpsc::channel(100); // 容纳100的数据
```

mpsc通道只能有一个Receiver端，但可以tx.clone()得到多个Sender端，clone得到的Sender都可以使用send()方法向该通道发送消息。
发送消息时，如果通道已满，发送消息的任务将等待直到通道中有空闲的位置。
发送消息时，如果Receiver端已经关闭，则发送消息的操作将返回SendError。
如果所有的Sender端都已经关闭，则Receiver端接收消息的方法recv()将返回None。
一个简单的示例：
ex4_2_2_1

```
use tokio::{ self, runtime::Runtime, sync };

fn main() {
    let rt = Runtime::new().unwrap();
    rt.block_on(async {
        let (tx, mut rx) = sync::mpsc::channel::<i32>(10);

        tokio::spawn(async move {
            for i in 1..=10 {
                // if let Err(_) = tx.send(i).await {}
                if tx.send(i).await.is_err() {
                    println!("receiver closed");
                    return;
                }
            }
        });

        while let Some(i) = rx.recv().await {
            println!("received: {}", i);
        }
    });
}
```

输出的结果：

```
received: 1
received: 2
received: 3
received: 4
received: 5
received: 6
received: 7
received: 8
received: 9
received: 10
```

上面的示例中，先生成了一个异步任务，该异步任务向通道中发送10个数据，Receiver端则在while循环中不断从通道中取数据。

将上面的示例改一下，**生成10个异步任务分别发送数据**：
ex4_2_2_2

```
use tokio::{ self, runtime::Runtime, sync };

fn main() {
    let rt = Runtime::new().unwrap();
    rt.block_on(async {
        let (tx, mut rx) = sync::mpsc::channel::<i32>(10);

        for i in 1..=10 {
            let tx = tx.clone(); // 引用
            tokio::spawn(async move {
                if tx.send(i).await.is_err() {
                    println!("receiver closed");
                }
            });
        }
        drop(tx);

        while let Some(i) = rx.recv().await {
            println!("received: {}", i);
        }
    });
}
```

输出的结果：

```
received: 2
received: 3
received: 1
received: 4
received: 6
received: 5
received: 10
received: 7
received: 8
received: 9
```

10个异步任务发送消息的顺序是未知的，因此接收到的消息无法保证顺序。

另外注意上面示例中的drop(tx)，因为生成的10个异步任务中都拥有clone后的Sender，clone出的Sender在每个异步任务完成时自动被drop，但原始任务中还有一个Sender，如果不关闭这个Sender，rx.recv()将不会返回None，而是一直等待。

如果通道已满，Sender通过send()发送消息时将等待。例如下面的示例中，通道容量为5，但要发送7个数据，前5个数据会立即发送，发送第6个消息的时候将等待，直到1秒后Receiver开始从通道中消费数据。
ex4_2_2_3

```
use chrono::Local;
use tokio::{self, sync, runtime::Runtime, time::{self, Duration}};

fn now() -> String {
```

```
    Local::now().format("%F %T").to_string()
}

fn main() {
    let rt = Runtime::new().unwrap();
    rt.block_on(async {
        let (tx, mut rx) = sync::mpsc::channel::<i32>(5); // 容量5的

        tokio::spawn(async move {
            for i in 1..=7 {    // 发送7个数据
              if tx.send(i).await.is_err() {
                println!("receiver closed");
                return;
              }
              println!("sended: {}, {}", i, now());
            }
        });

        time::sleep(Duration::from_secs(1)).await;
        while let Some(i) = rx.recv().await {
            println!("received: {}", i);
        }
    });
}
```

输出结果：

```
sended: 1, 2022-05-07 15:38:46
sended: 2, 2022-05-07 15:38:46
sended: 3, 2022-05-07 15:38:46
sended: 4, 2022-05-07 15:38:46
sended: 5, 2022-05-07 15:38:46
received: 1
received: 2
received: 3
received: 4
received: 5
sended: 6, 2022-05-07 15:38:47
received: 6
sended: 7, 2022-05-07 15:38:47
received: 7
```

Sender端和Receiver端有一些额外的方法需要解释一下它们的作用。
对于Sender端：

- capacity(): 获取当前通道的剩余容量(注意，不是初始化容量)
- closed(): 等待Receiver端关闭，当Receiver端关闭后该等待任务会立即完成
- is_closed(): 判断Receiver端是否已经关闭
- send(): 向通道中发送消息，通道已满时会等待通道中的空闲位置，如果对端已关闭，则返回错误
- send_timeout(): 向通道中发送消息，通道已满时只等待指定的时长
- try_send(): 向通道中发送消息，但不等待，如果发送不成功，则返回错误
- reserve(): 等待并申请一个通道中的空闲位置，返回一个Permit，申请的空闲位置被占位，且该位置只留给该Permit实例，之后该Permit可以直接向通道中发送消息，并释放其占位的位置。申请成功时，通道空闲容量减1，释放位置时，通道容量会加1
- try_reserve(): 尝试申请一个空闲位置且不等待，如果无法申请，则返回错误

- reserve_owned(): 与reserve()类似，它返回OwnedPermit，但会Move Sender
- try_reserve_owned(): reserve_owned()的不等待版本，尝试申请空闲位置失败时会立即返回错误
- blocking_send(): Sender可以在同步代码环境中使用该方法向异步环境发送消息

对于Receiver端：

- close(): 关闭Receiver端
- recv(): 接收消息，如果通道已空，则等待，如果对端已全部关闭，则返回None
- try_recv(): 尝试接收消息，不等待，如果无法接收消息(即通道为空或对端已关闭)，则返回错误
- blocking_recv(): Receiver可以在同步代码环境中使用该方法接收来自异步环境的消息

注意，在这些方法中，try_xxx()方法都是立即返回不等待的(可以认为是同步代码)，因此调用它们后无需await，只有调用那些可能需要等待的方法，调用后才需要await。例如rx.recv().await和rx.try_recv()。
下面是一些稍详细的用法说明和示例。
Sender端可通过send_timeout()来设置一个等待通道空闲位置的超时时间，它和send()返回值一样，此外还添加一种超时错误：超时后仍然没有发送成功时将返回错误。至于返回的是什么错误，对于发送端来说不重要，重要的是发送的消息是否成功。因此，对于Sender端的条件判断，通常也仅仅只是检测is_err()：

```
if tx.send_timeout(33, Duration::from_secs(1)).await.is_err() {
  println!("receiver closed or timeout");
}
```

需要特别注意的是，Receiver端调用close()方法关闭通道后，只是半关闭状态，Receiver端仍然可以继续读取可能已经缓冲在通道中的消息，close()只能保证Sender端无法再发送普通的消息，但Permit或OwnedPermit仍然可以向通道发送消息。只有通道已空且所有Sender端(包括Permit和OwnedPermit)都已经关闭的情况下，recv()才会返回None，此时代表通道完全关闭。
Receiver的try_recv()方法在无法立即接收消息时会立即返回错误。返回的错误分为两种:

- TryRecvError::Empty错误: 表示通道已空，但Sender端尚未全部关闭
- TryRecvError::Disconnected错误: 表示通道已空，且Sender端(包括Permit和OwnedPermit)已经全部关闭

关于reserve()和reserve_owned()，看官方示例即可轻松理解：
ex4_2_2_4

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

#[tokio::main]
async fn main() {
    // 创建容量为1的通道
    let (tx, mut rx) = mpsc::channel(1);
    // 申请并占有唯一的空闲位置
    let permit = tx.reserve().await.unwrap();
    // 唯一的位置已被permit占有，tx.send()无法发送消息
    assert!(tx.try_send(123).is_err());
    // Permit可以通过send()方法向它占有的那个位置发送消息
    permit.send(456);
    // Receiver端接收到消息
    assert_eq!(rx.recv().await.unwrap(), 456);


    // 创建容量为1的通道
    let (tx, mut rx) = mpsc::channel(1);
    // tx.reserve_owned()会消费掉tx
    let permit = tx.reserve_owned().await.unwrap();
```

```
    // 通过permit.send()发送消息，它又返回一个Sender
    let tx = permit.send(456);
    assert_eq!(rx.recv().await.unwrap(), 456);
    //可以继续使用返回的Sender发送消息
    tx.send(789).await.unwrap();
}
```

无界通道

理解了mpsc的有界通道之后，再理解无界通道会非常轻松。

```
let (tx, mut rx) = mpsc::unbounded_channel();
```

对于无界通道，它的通道中可以缓冲无限数量的消息，直到内存耗尽。这意味着，Sender端可以无需等待地不断向通道中发送消息，这也意味着无界通道的Sender既可以在同步环境中也可以在异步环境中向通道中发送消息。只有当Receiver端已经关闭，Sender端的发送才会返回错误。

使用无界通道的关键，在于**必须要保证不会无限度地缓冲消息而导致内存耗尽。**例如，让Receiver端消费消息的速度尽量快，或者采用一些复杂的限速机制让严重超前的Sender端等一等。

4.2.3 broadcast通道

broadcast通道是一种广播通道，**可以有多个Sender端以及多个Receiver端**，可以发送多个数据，且任何一个Sender发送的每一个数据都能被所有的Receiver端看到。

使用mpsc::broadcast()创建广播通道，要求指定一个通道容量作为参数。它返回Sender和Receiver。
Sender可以克隆得到多个Sender，可以调用Sender的subscribe()方法来创建新的Receiver。
例如，下面是官方文档提供的一个示例：
ex4_2_3_1

```
use tokio::sync::broadcast;

#[tokio::main]
async fn main() {
    // 最多存放16个消息
    // tx是Sender，rx1是Receiver
    let (tx, mut rx1) = broadcast::channel(16);

    // Sender的subscribe()方法可生成新的Receiver
    let mut rx2 = tx.subscribe();

    tokio::spawn(async move {
        assert_eq!(rx1.recv().await.unwrap(), 10);
        assert_eq!(rx1.recv().await.unwrap(), 20);
    });

    tokio::spawn(async move {
        assert_eq!(rx2.recv().await.unwrap(), 10);
        assert_eq!(rx2.recv().await.unwrap(), 20);
    });

    tx.send(10).unwrap();
    tx.send(20).unwrap();
```

```
}
```

Sender端通过send()发送消息的时候，如果所有的Receiver端都已关闭，则send()方法返回错误。Receiver端可通过recv()去接收消息，如果所有的Sender端都已经关闭，则该方法返回RecvError::Closed错误。该方法还可能返回RecvError::Lagged错误，该错误表示接收端已经落后于发送端。

虽然broadcast通道也指定容量，但是通道已满的情况下还可以继续写入新数据而不会等待(因此上面示例中的send()无需await)，**此时通道中最旧的(头部的)数据将被剔除，并且新数据添加在尾部。**就像是FIFO队列一样。出现这种情况时，就意味着接收端已经落后于发送端。

当接收端已经开始落后于发送端时，下一次的recv()操作将直接返回RecvError::Lagged错误。如果紧跟着再执行recv()且落后现象未再次发生，那么这次的recv()将取得队列头部的消息。
ex4_2_3_2

```rust
use tokio::sync::broadcast;

#[tokio::main]
async fn main() {
    // 通道容量2
    let (tx, mut rx) = broadcast::channel(2);

    // 写入3个数据，将出现接收端落后于发送端的情况，
    // 此时，第一个数据(10)将被剔除，剔除后，20将位于队列的头部
    tx.send(10).unwrap();
    tx.send(20).unwrap();
    tx.send(30).unwrap();

    // 落后于发送端之后的第一次recv()操作，返回RecvError::Lagged错误
    assert!(rx.recv().await.is_err());

    // 之后可正常获取通道中的数据
    assert_eq!(20, rx.recv().await.unwrap());
    assert_eq!(30, rx.recv().await.unwrap());
}
```

Receiver也可以使用try_recv()方法去无等待地接收消息，如果Sender都已关闭，则返回TryRecvError::Closed错误，如果接收端已落后，则返回TryRecvError::Lagged错误，如果通道为空，则返回TryRecvError::Empty错误。
另外，tokio::broadcast的任何一个Receiver都可以看到每一次发送的消息，且它们都可以去recv()同一个消息，tokio::broadcast对此的处理方式是消息克隆：每一个Receiver调用recv()去接收一个消息的时候，都会克隆通道中的该消息一次，直到所有存活的Receiver都克隆了该消息，该消息才会从通道中被移除，进而释放一个通道空闲位置。
这可能会导致一种现象：某个ReceiverA已经接收了通道中的第10个消息，但另一个ReceiverB可能尚未接收第一个消息，由于第一个消息还未被全部接收者所克隆，它仍会保留在通道中并占用通道的位置，假如该通道的最大容量为10，此时Sender再发送一个消息，那么第一个数据将被踢掉，ReceiverB接收到消息的时候将收到RecvError::Lagged错误并永远地错过第一个消息。

4.2.4 watch通道

watch通道的特性是：**只能有单个Sender，可以有多个Receiver，且通道永远只保存一个数据。**
Sender每次向通道中发送数据时，都会修改通道中的那个数据。
通道中的这个数据可以被Receiver进行引用读取。

一个简单的官方示例：
ex4_2_4_1

```
use tokio::sync::watch;
#[tokio::main]
async fn main() {
    // 创建watch通道时，需指定一个初始值存放在通道中
    let (tx, mut rx) = watch::channel("hello");

    // Recevier端，通过changed()来等待通道的数据发生变化
    // 通过borrow()引用通道中的数据
    tokio::spawn(async move {
        while rx.changed().await.is_ok() {
            println!("received = {:?}", *rx.borrow());
        }
    });

    // 向通道中发送数据，实际上是修改通道中的那个数据
    tx.send("world").unwrap();
}
```

watch通道的用法很简单，但是有些细节需要理解。
Sender端可通过subscribe()创建新的Receiver端。

当所有Receiver端均已关闭时，send()方法将返回错误。也就是说，send()必须要在有Receiver存活的情况下才能发送数据。

但是Sender端还有一个send_replace()方法，它可以在没有Receiver的情况下将数据写入通道，并且该方法会返回通道中原来保存的值。

无论是Sender端还是Receiver端，都可以通过borrow()方法取得通道中当前的值。由于可以有多个Receiver，为了避免读写时的数据不一致，watch内部使用了读写锁：**Sender端要发送数据修改通道中的数据时，需要申请写锁，论是Sender还是Receiver端，在调用borrow()或其它一些方式访问通道数据时，都需要申请读锁。**因此，访问通道数据时要尽快释放读锁，否则可能会长时间阻塞Sender端的发送操作。

如果Sender端未发送数据，或者隔较长时间才发送一次数据，那么通道中的数据在一段时间内将一直保持不变。如果Receiver在这段时间内去多次读取通道，得到的结果将完全相同。但有时候，可能更需要的是等待通道中的数据已经发生变化，然后再根据新的数据做进一步操作，而不是循环不断地去读取并判断当前读取到的值是否和之前读取的旧值相同。

watch通道已经提供了这种功能：Receiver端可以标记通道中的数据，记录该数据是否已经被读取过。Receiver端的changed()方法用于等待通道中的数据发生变化，其内部判断过程是：如果通道中的数据已经被标记为已读取过，那么changed()将等待数据更新，如果数据未标记过已读取，那么changed()认为当前数据就是新数据，changed()会立即返回。

Receiver端的borrow()方法不会标记数据已经读取，所以borrow()之后调用的changed()会立即返回。但是changed()等待到新值之后，会立即将该值标记为已读取，使得下次调用changed()时会进行等待。

此外，Receiver端还有一个borrow_and_update()方法，它会读取数据并标记数据已经被读取，因此随后调用chagned()将进入等待。
最后再强调一次，无论是Sender端还是Receiver端，访问数据的时候都会申请读锁，要尽量快地释放读锁，以免Sender长时间无法发送数据。

4.3 同步

tokio::sync模块提供了几种状态同步的机制：

- **Mutex: 互斥锁**
- RwLock: 读写锁
- Notify: 通知唤醒机制
- Barrier: 屏障
- Semaphore: 信号量

因为**tokio是跨线程执行任务的**，因此通常会使用Arc来封装这些同步原语，以使其能够跨线程。例如：

```
let mutex = Arc::new(Mutex::new());
let rwlock = Arc::new(Mutex::RwLock());
```

4.3.1 Mutex互斥锁

当多个并发任务(tokio task或线程)可能会修改同一个数据时，就会出现数据竞争现象(竞态)，具体表现为：某个任务对该数据的修改不生效或被覆盖。

互斥锁的作用，就是保护并发情况下可能会出现竞态的代码，这部分代码称为临界区。当某个任务要执行临界区中的代码时，必须先申请锁，申请成功，则可以执行这部分代码，执行完成这部分代码后释放锁。释放锁之前，其它任务无法再申请锁，它们必须等待锁被释放。
假如某个任务一直持有锁，其它任务将一直等待。因此，互斥锁应当尽量快地释放，这样可以提高并发量。

简单介绍完互斥锁之后，再看tokio提供的互斥锁。

tokio::sync::Mutex使用new()来创建互斥锁，使用lock()来申请锁，申请锁成功时将返回MutexGuard，并通过drop的方式来释放锁。
例如：
ex4_3_1_1

```
use std::sync::Arc;
use tokio::{self, sync, runtime::Runtime, time::{self, Duration}};

fn main() {
    let rt = Runtime::new().unwrap();
    rt.block_on(async {
        let mutex = Arc::new(sync::Mutex::new(0));

        for i in 0..10 {
            let lock = Arc::clone(&mutex);  // 复制
            tokio::spawn(async move {
                let mut data = lock.lock().await;
                *data += 1;
                println!("task: {}, data: {}", i, data);
            });
        }
```

```
        time::sleep(Duration::from_secs(1)).await;
    });
}
```

输出结果：

```
task: 0, data: 1
task: 1, data: 2
task: 2, data: 3
task: 3, data: 4
task: 4, data: 5
task: 5, data: 6
task: 6, data: 7
task: 7, data: 8
task: 8, data: 9
task: 9, data: 10
```

可以看到，任务的调度顺序是随机的，但是数据加1的操作是依次完成的。
需特别说明的是，tokio::sync::Mutex其内部使用了标准库的互斥锁，即std::sync::Mutex，而标准库的互斥锁是针对线程的，因此，使用tokio的互斥锁时也会锁住整个线程。此外，**tokio::sync::Mutex是对标准库的Mutex的封装，性能相对要更差一些。**也因此，官方文档中建议，如非必须，**应使用标准库的Mutex或性能更高的parking_lot提供的互斥锁，而不是tokio的Mutex。**
例如，将上面的示例该成标准库的Mutex锁。
ex4_3_1_2

```
use std::sync::Arc;
use tokio::{self, sync, runtime::Runtime, time::{self, Duration}};
fn main() {
    let rt = Runtime::new().unwrap();
    rt.block_on(async {
        let mutex = Arc::new(std::sync::Mutex::new(0));

        for i in 0..10 {
            let lock = mutex.clone();
            tokio::spawn(async move {
                let mut data = lock.lock().unwrap();
                *data += 1;
                println!("task: {}, data: {}", i, data);
            });
        }

        time::sleep(Duration::from_secs(1)).await;
    });
}
```

什么情况下可以选择使用tokio的Mutex？当跨await的时候，可以考虑使用tokio Mutex，因为这时使用标准库的Mutex将编译错误。当然，也有相应的解决方案。

什么是跨await？**每个await都代表一个异步任务，跨await即表示该异步任务中出现了至少一个子任务。而每个异步任务都可能会被tokio内部偷到不同的线程上执行，因此跨await时要求其父任务实现Send Trait，这是因为子任务中可能会引用父任务中的数据。**
例如，下面定义的async函数中使用了标准库的Mutex，且有子任务，这会编译错误：

ex4_3_1_3

```
use std::sync::{Arc, Mutex, MutexGuard};
use tokio::{self, runtime::Runtime, time::{self, Duration}};

async fn add_1(mutex: &Mutex<u64>) {
    let mut lock = mutex.lock().unwrap();
    *lock += 1;

    // 子任务，跨await，且引用了父任务中的数据
    time::sleep(Duration::from_millis(*lock)).await;
}

fn main() {
    let rt = Runtime::new().unwrap();
    rt.block_on(async {
        let mutex = Arc::new(Mutex::new(0));

        for i in 0..10 {
            let lock = mutex.clone();
            tokio::spawn(async move {
                add_1(&lock).await;
            });
        }

        time::sleep(Duration::from_secs(1)).await;
    });
}
```

报错：

```
   --> src/main.rs:19:13
    |
19  |             tokio::spawn(async move {
    |             ^^^^^^^^^^^^ future created by async block is not `Send`
```

std::sync::MutexGuard未实现Send，因此父任务async move{}语句块是非Send的，于是编译报错。但如果上面的示例中没有子任务sleep().await子任务，则编译无错，因为已经可以明确知道该Mutex所在的任务是在当前线程执行的。
对于上面的错误，可简单地使用tokio::sync::Mutex来修复。
ex4_3_1_4

```
use std::sync::Arc;
use tokio::{ self, runtime::Runtime, sync::{Mutex, MutexGuard}, time::{self,
Duration} };

async fn add_1(mutex: &Mutex<u64>) {
    let mut lock = mutex.lock().await;
    *lock += 1;
    time::sleep(Duration::from_millis(*lock)).await;
}

fn main() {
    let rt = Runtime::new().unwrap();
```

```
    rt.block_on(async {
        let mutex = Arc::new(Mutex::new(0));
        for i in 0..10 {
            let lock = mutex.clone();
            tokio::spawn(async move {
                add_1(&lock).await;
            });
        }

        time::sleep(Duration::from_secs(1)).await;
    });
}
```

前面已经说过，tokio的Mutex性能相对较差一些，因此可以不使用tokio Mutex的情况下，尽量不使用它。对于上面的需求，**仍然可以继续使用标准库的Mutex，但需要做一些调整。**
例如，可以在子任务await之前，把所有未实现Send的数据都drop掉，保证子任务无法引用父任务中的任何非Send数据。
ex4_3_1_5

```
use std::sync::{Arc, Mutex};
use tokio::{ self, runtime::Runtime, sync, time::{self, Duration}};

fn main() {
    use std::sync::{Arc, Mutex, MutexGuard};

    async fn add_1(mutex: &Mutex<u64>) {
        {
        let mut lock = mutex.lock().unwrap();
        *lock += 1;
        }
        // 子任务，跨await，不引用父任务中的数据
        time::sleep(Duration::from_millis(10)).await;
    }
}
```

这种方案的主要思想是让子任务和父任务不要出现不安全的数据交叉。如果可以的话，应尽量隔离子任务和非Send数据所在的任务。上面的例子已经实现了这一点，但更好的方式是将子任务sleep().await从这个函数中移走。
ex4_3_1_6

```
use std::sync::{Arc, Mutex};
#[allow(unused_imports)]
use tokio::{ self, runtime::Runtime, sync, time::{self, Duration}};

async fn add_1(mutex: &Mutex<u64>) -> u64 {
    let mut lock = mutex.lock().unwrap();
    *lock += 1;
    *lock
}  // 申请的互斥锁在此被释放

fn main() {
    let rt = Runtime::new().unwrap();
    rt.block_on(async {
        let mutex = Arc::new(Mutex::new(0));
```

```
        for i in 0..100 {
            let lock = mutex.clone();
            tokio::spawn(async move {
                let n = add_1(&lock).await;
                time::sleep(Duration::from_millis(n)).await;
            });
        }

        time::sleep(Duration::from_secs(1)).await;
        println!("data: {}", mutex.lock().unwrap());
    });
}
```

另外注意，标准库的Mutex存在毒锁问题。所谓毒锁，即某个持有互斥锁的线程panic了，那么这个锁有可能永远得不到释放(除非线程panic之前已经释放)，也称为被污染的锁。毒锁问题可能很严重，因为出现毒锁有可能意味着数据将从此开始不再准确，所以多数时候是直接让毒锁的panic向上传播或单独处理。但出现毒锁并不总是危险的，所以标准库也提供了对应的方案。
但tokio Mutex不存在毒锁问题，在持有Mutex的线程panic时，tokio的做法是直接释放锁。

4.3.2 RwLock读写锁

相比Mutex互斥锁，读写锁区分读操作和写操作，读写锁允许多个读锁共存，但写锁独占。因此，在并发能力上它比Mutex要更好一些。
下面是官方文档中的一个示例：
ex4_3_2_1

```
use tokio::sync::RwLock;

#[tokio::main]
async fn main() {
    let lock = RwLock::new(5);

    // 多个读锁共存
    {
        // read()返回RwLockReadGuard
        let r1 = lock.read().await;
        let r2 = lock.read().await;
        assert_eq!(*r1, 5);  // 对Guard解引用，即可得到其内部的值
        assert_eq!(*r2, 5);
    } // 读锁(r1, r2)在此释放

    // 只允许一个写锁存在
    {
        // write()返回RwLockWriteGuard
        let mut w = lock.write().await;
        *w += 1;
        assert_eq!(*w, 6);
    } // 写锁(w)被释放
}
```

需注意，读写锁有几种不同的设计方式：

- 读锁优先: 只要有读操作申请锁，优先将锁分配给读操作。这种方式可以提供非常好的并发能力，但是大量的读操作可能会长时间阻挡写操作
- 写锁优先: 只要有写操作申请锁，优先将锁分配给写操作。这种方式可以保证写操作不会被饿死，但会严重影响并发能力

tokio RwLock实现的是写锁优先，它的具体规则如下：

1. 每次申请锁时都将等待，申请锁的异步任务被切换，CPU交还给调度器
2. 如果申请的是读锁，并且此时没有写锁存在，则申请成功，对应的任务被唤醒
3. 如果申请的是读锁，但此时有写锁(包括写锁申请)的存在，那么将等待所有的写锁释放(因为写锁总是优先)
4. 如果申请的是写锁，如果此时没有读锁的存在，则申请成功
5. 如果申请的是写锁，但此时有读锁的存在，那么将等待当前正在持有的读锁释放

注意，RwLock的写锁优先会很容易产生死锁。例如，下面的代码会产生死锁：
ex4_3_2_2

```
use std::sync::Arc;
use tokio::{self, runtime::Runtime, sync::RwLock, time::{self, Duration}};

fn main() {
    let rt = Runtime::new().unwrap();
    rt.block_on(async {
        let lock = Arc::new(RwLock::new(0));

        let lock1 = lock.clone();
        tokio::spawn(async move {
            let n = lock1.read().await;

            time::sleep(Duration::from_secs(2)).await;
            let nn = lock1.read().await;
        });

        time::sleep(Duration::from_secs(1)).await;
        let mut wn = lock.write().await;
        *wn = 2;
    });
}
```

上面示例中，按照时间的流程，首先会在子任务中申请读锁，1秒后在当前任务中申请写锁，再1秒后子任务申请读锁。
申请第一把读锁时，因为此时无锁，所以读锁(即变量n)申请成功。1秒后申请写锁时，由于此时读锁n尚未释放，因此写锁申请失败，将等待。再1秒之后，继续在子任务中申请读锁，但是此时有写锁申请存在，因此第二次申请读锁将等待，于是读锁写锁互相等待，死锁出现了。
当要使用写锁时，**如果要避免死锁，一定要保证同一个任务中的任意两次锁申请之间，前面已经无锁，并且写锁尽早释放。**
对于上面的示例，同一个子任务中申请两次读锁，但是第二次申请读锁时，第一把读锁仍未释放，这就产生了死锁的可能。只需在第二次申请读锁前，将第一把读锁释放即可。更完整一点，在写锁写完数据后也手动释放写锁(上面的示例中写完就退出，写锁会自动释放，因此无需手动释放)。
ex4_3_2_3

```
use std::sync::Arc;
use tokio::{self, runtime::Runtime, sync::RwLock, time::{self, Duration}};
```

```
fn main() {
    let rt = Runtime::new().unwrap();
    rt.block_on(async {
        let lock = Arc::new(RwLock::new(0));

        let lock1 = lock.clone();
        tokio::spawn(async move {
            let n = lock1.read().await;
            drop(n);  // 在申请第二把读锁前，先释放第一把读锁

            time::sleep(Duration::from_secs(2)).await;
            let nn = lock1.read().await;
            drop(nn);
        });

        time::sleep(Duration::from_secs(1)).await;
        let mut wn = lock.write().await;
        *wn = 2;
        drop(wn);
    });
}
```

RwLock还有一些其它的方法，在理解了RwLock申请锁的规则之后，这些方法都很容易理解，可以自行去查看官方手册。

4.3.3 Notify通知唤醒

Notify提供了一种简单的通知唤醒功能，它类似于只有一个信号灯的信号量。
下面是官方文档中的示例：
ex4_3_3_1

```
use tokio::sync::Notify;
use std::sync::Arc;

#[tokio::main]
async fn main() {
    let notify = Arc::new(Notify::new());
    let notify2 = notify.clone();

    tokio::spawn(async move {
        notify2.notified().await;
        println!("received notification");
    });

    println!("sending notification");
    notify.notify_one();
}
```

Notify::new()创建Notify实例，Notify实例初始时没有permit位，permit可认为是执行权。
每当调用notified().await时，将判断此时是否有执行权，如果有，则可直接执行，否则将进入等待。因此，初始化之后立即调用notified().await将会等待。
每当调用notify_one()时，将产生一个执行权，但多次调用也最多只有一个执行权。因此，调用notify_one()之后再调用notified().await则并无需等待。
如果同时有多个等待执行权的等候者，释放一个执行权，在其它环境中可能会产生惊群现象，即大量等

候者被一次性同时唤醒去争抢一个资源，抢到的可以继续执行，而未抢到的等候者又重新被阻塞。好在，tokio Notify没有这种问题，tokio使用队列方式让等候者进行排队，先等待的总是先获取到执行权，因此不会一次性唤醒所有等候者，而是只唤醒队列头部的那个等候者。

Notify还有一个notify_waiters()方法，它不会释放执行权，但是它会一次性唤醒所有正在等待的等候者。严格来说，是让当前已经注册的等候者(即已经调用notified()，但是还未await)在下次等待的时候，可以直接通过。
官方手册给了一个示例：
ex4_3_3_2

```
use tokio::sync::Notify;
use std::sync::Arc;

#[tokio::main]
async fn main() {
    let notify = Arc::new(Notify::new());
    let notify2 = notify.clone();

    // 注册两个等候者
    let notified1 = notify.notified();
    let notified2 = notify.notified();

    let handle = tokio::spawn(async move {
        println!("sending notifications");
        notify2.notify_waiters();
    });

    // 两个等候者的await都会直接通过
    notified1.await;
    notified2.await;
    println!("received notifications");
}
```

4.3.4 Barrier屏障

Barrier是一种让多个并发任务在某种程度上保持进度同步的手段。
例如，一个任务分两步，有很多个这种任务并发执行，**但每个任务中的第二步都要求所有任务的第一步已经完成。**这时可以在第二步之前使用屏障，这样可以保证所有任务在开始第二步之前的进度是同步的。

当然，也不一定要等待所有任务的进度都同步，可以设置等待一部分任务的进度同步。也就是说，让并发任务的进度按批次进行同步。第一批的任务进度都同步后，这一批任务将通过屏障，但是该屏障依然会阻挡下一批任务，直到下一批任务的进度都同步之后才放行。

使用屏障时，一定要保证可以到达屏障点的并发任务数量是屏障宽度的整数倍，否则多出来的任务将一直等待。例如，**将屏障的宽度设置为10(即10个任务一批)**，**但是有15个并发任务**，**多出来的5个任务无法凑成完整的一批**，这5个任务将一直等待。
ex4_3_4_2

```
use std::sync::Arc;
use tokio::sync::Barrier;
use tokio::{ self, runtime::Runtime, time::{self, Duration} };

fn main() {
    let rt = Runtime::new().unwrap();
```

```
    rt.block_on(async {
        let barrier = Arc::new(Barrier::new(10));

        for i in 1..=15 {
            let b = barrier.clone();
            tokio::spawn(async move {
                println!("data before: {}", i);

                b.wait().await; // 15个任务中，多出5个任务将一直在此等待
                time::sleep(Duration::from_millis(10)).await;
                println!("data after: {}", i);
            });
        }
        time::sleep(Duration::from_secs(5)).await;
    });
}
```

4.3.5 Semaphore信号量

信号量可以保证在某一时刻最多运行指定数量的并发任务。

使用信号量时，需在初始化时指定信号灯(tokio中的SemaphorePermit)的数量，每当任务要执行时，将从中取走一个信号灯，当任务完成时(信号灯被drop)会归还信号灯。当某个任务要执行时，如果此时信号灯数量为0，则该任务将等待，直到有信号灯被归还。因此，信号量通常用来提供类似于限量的功能。

例如，信号灯数量为1，表示所有并发任务必须串行运行，这种模式和互斥锁是类似的。再例如，信号灯数量设置为2，表示最多只有两个任务可以并发执行，如果有第三个任务，则必须等前两个任务中的某一个先完成。
例如：
ex4_3_5_1

```
use chrono::Local;
use std::sync::Arc;
use tokio::{ self, runtime::Runtime, sync::Semaphore, time::{self, Duration}};

fn now() -> String {
    Local::now().format("%F %T").to_string()
}

fn main() {
    let rt = Runtime::new().unwrap();
    rt.block_on(async {
        // 只有3个信号灯的信号量
        let semaphore = Arc::new(Semaphore::new(3));

        // 5个并发任务，每个任务执行前都先获取信号灯
        // 因此，同一时刻最多只有3个任务进行并发
        for i in 1..=5 {
            let semaphore = semaphore.clone();
            tokio::spawn(async move {
                let _permit = semaphore.acquire().await.unwrap();
                println!("{}, {}", i, now());
                time::sleep(Duration::from_secs(1)).await;
            });
        }
```

```
        time::sleep(Duration::from_secs(3)).await;
    });
}
```

输出结果：

```
2, 2022-05-07 16:14:24
3, 2022-05-07 16:14:24
1, 2022-05-07 16:14:24
4, 2022-05-07 16:14:25
5, 2022-05-07 16:14:25
```

tokio::sync::Semaphore提供了以下一些方法:

- close(): 关闭信号量，关闭信号量时，将唤醒所有的信号灯等待者
- is_closed(): 检查信号量是否已经被关闭
- acquire(): 获取一个信号灯，如果信号量已经被关闭，则返回错误AcquireError
- acquire_many(): 获取指定数量的信号灯，如果信号灯数量不够则等待，如果信号量已经被关闭，则返回AcquireError
- add_permits(): 向信号量中额外添加N个信号灯
- available_permits(): 当前信号量中剩余的信号灯数量
- try_acquire(): 不等待地尝试获取一个信号灯，如果信号量已经关闭，则返回TryAcquireError::Closed，如果目前信号灯数量为0，则返回TryAcquireError::NoPermits
- try_acquire_many(): 尝试获取指定数量的信号灯
- acquire_owned(): 获取一个信号灯并消费掉信号量
- acquire_many_owned(): 获取指定数量的信号灯并消费掉信号量
- try_acquire_owned(): 尝试获取信号灯并消费掉信号量
- try_acquire_many_owned(): 尝试获取指定数量的信号灯并消费掉信号量

对于获取到的信号灯SemaphorePermit，有一个forget()方法，该方法可以将信号灯不归还给信号量，因此信号量中的信号灯将永久性地减少(当然，可使用add_permits()添加额外的信号灯)。
信号量的限量功能，也可以通过sync::mpsc通道来实现。大致逻辑为：设置通道宽度为允许的最大并发任务数量，并先填满通道，当执行一个任务时，先从通道取走一个消息，再执行任务，每次执行完任务后都重新向通道中回补一个消息。

5 使用tokio::net进行网络编程

okio提供了类似std::net所提供的基本设施以便进行异步网络编程，主要包括tcp、udp和unix domain三方面。
网络编程需要大量的网络编程知识，且和IO编程息息相关，因暂时还未介绍tokio::io，所以本文暂且仅介绍tokio::net的tcp编程相关的基础设施，不涉及具体的网络编程逻辑。(所以本文会比较枯燥，基本上是对官方文档的总结和引用)
要使用tokio::net，需在Cargo.toml文件中开启net特性：

```
tokio = {version = "1", features = ["rt", "net", "rt-multi-thread"]}
```

开启该特性之后，将可使用以下三个组件：

- TcpSocket: 创建和操作套接字的基础组件

- TcpListener: 对TcpSocket的一些封装，主要提供服务端套接字的相关操作
- TcpStream: 代表已建立的可直接传递数据的连接，对客户端来说代表已经被服务端接收，对服务端来说代表accept后的套接字

通常客户端可直接使用TcpStream，服务端可直接使用TcpListener和TcpStream，如果需要自定义修改套接字的选项或属性，则考虑使用TcpSocket。

5.1 IpAddr和SocketAddr

在开始介绍tokio::net之前，需先简单介绍一下与之相关的std::net::IpAddr和std::net::SocketAddr(注意它们来自标准库)。

IpAddr

IpAddr封装了IP地址，包括IP v4地址和IP v6地址：

```
pub enum IpAddr {
    V4(Ipv4Addr),
    V6(Ipv6Addr),
}
```

IpAddr实现了FromStr，可直接将代表IP地址的字符串解析为IpAddr：

```
let localhsot: IpAddr = "127.0.0.1".parse().unwrap();
```

例如：

```
use std::net::{IpAddr, Ipv4Addr, Ipv6Addr};

let localhost = IpAddr::V4(Ipv4Addr::new(127, 0, 0, 1));
assert_eq!("127.0.0.1".parse(), Ok(localhost));
```

IpAddr还有一些方法，主要是一些布尔判断方法：

- is_ipv4()：是否是一个ipv4地址
- is_ipv6()：是否是一个ipv6地址
- is_loopack()：是否是一个loopback地址
- is_multicast()：是否是一个多播地址
- is_unspecified()：是否是一个0.0.0.0地址

IpAddr封装了ip v4地址或ip v6地址，以代表ip v4地址的Ipv4Addr为例。可使用new()并提供4个u8参数来创建ip v4地址：

```

use std::net::Ipv4Addr;

let localhost = Ipv4Addr::new(127, 0, 0, 1);
```

Ipv4Addr实现了FromStr，也可以很方便地直接将字符串解析为ip地址：

```
 let localhost = "127.0.0.1".parse().unwrap();
```

可使用octets()将一个IP地址转换为u8数组，即new()的反向操作：

```rust
use std::net::Ipv4Addr;

let addr = Ipv4Addr::new(127, 0, 0, 1);
assert_eq!(addr.octets(), [127, 0, 0, 1]);
```

Ipv4Addr还有其它一些方法，多数都是布尔判断方法:

- is_broadcast(): 是否是广播地址(255.255.255.255)
- is_multicast(): 是否是多播地址(224.0.0.0/4)
- is_private(): 是否是私有地址(10.0.0.0/8、172.16.0.0/12、192.168.0.0/16)
- is_link_local(): 是否是链路本地地址(169.254.0.0/16)
- is_loopback(): 是否是环回地址(127.0.0.0/8)
- is_unspecified(): 是否是0.0.0.0

此外，可直接对地址进行大小比较和等值比较。

SocketAddr

SocketAddr代表包含了**IP地址和端口号的套接字地址**，它封装了ipv4套接字地址和ipv6套接字地址：

```rust
pub enum SocketAddr {
    V4(SocketAddrV4),
    V6(SocketAddrV6),
}
```

SocketAddr实现了FromStr，因此可直接将代表套接字地址的字符串解析为SocketAddr:

```rust
use std::net::{IpAddr, Ipv4Addr, SocketAddr};

let socket: SocketAddr = "127.0.0.1:8080".parse().unwrap();
```

SocketAddr自身也提供了new()方法，需提供IpAddr和端口号(u16)作为参数：

```rust
use std::net::{IpAddr, Ipv4Addr, SocketAddr};

let ip = IpAddr::V4(Ipv4Addr::new(127, 0, 0, 1));
let socket = SocketAddr::new(ip, 8080);
```

此外，还有以下几个方法：

- is_ipv4(): 是否是ip v4套接字地址
- is_ipv6(): 是否是ip v6套接字地址
- ip(): 返回IP地址
- port(): 返回端口号
- set_ip(): 修改IP地址
- set_port(): 修改端口号

SocketAddr封装的代表ipv4套接字的SocketAddrV4也很简单直接，可由代表ipv4套接字的字符串解析得到，也可由new()方法创建，其也具有ip()、port()、set_ip()以及set_port()这几个方法。

```rust
use std::net::{Ipv4Addr, SocketAddrV4};

let socket = SocketAddrV4::new(Ipv4Addr::new(127, 0, 0, 1), 8080);

assert_eq!("127.0.0.1:8080".parse(), Ok(socket));
assert_eq!(socket.ip(), &Ipv4Addr::new(127, 0, 0, 1));
assert_eq!(socket.port(), 8080);
```

5.2 tokio::net::TcpListener

TcpListener代表服务端套接字，可使用bind()方法指定要绑定的地址，bind()之后再await，即可开始监听。

```rust
use tokio::net::TcpListener;

#[tokio::main]
async fn main(){
  let listener = TcpListener::bind("127.0.0.1:8888").await.unwrap();
}
```

这里的listener代表的是服务端负责监听的套接字。

注意，TcpListener::bind()默认会开启TCP的地址重用选项(SO_REUSEADDR)。如果想要修改该选项或设置其它TCP选项，应使用TcpSocket来创建套接字并设置选项，然后再调用bind()方法得到监听套接字。

得到监听套接字之后，可使用accept()去接收来自客户端的连接请求。accept()会阻塞(等待)，直到有新的客户端发起连接请求。
accept()成功，表示和客户端之间成功建立TCP连接(连接进入Established状态)，同时它会返回一个新的套接字(TcpStream)和代表客户端的套接字地址(SocketAddr)。可通过该TcpStream和客户端传输数据，可通过该SocketAddr获取客户端的地址和端口信息。如果要获取本地套接字地址相关的信息，可使用listener的local_addr()方法。

通常来说，会在一个无限循环中去accept()，这样可以保证多次接收客户端的连接请求。此外，一般也会为每一个accept()成功后返回的TcpStream去分配一个独立的线程或异步任务，这样可以异步地和每个客户端进行通信，且不影响监听套接字继续监听更多的客户端连接请求。

因此，tcp编程的服务端最基本的处理模式大致如下：

```rust
async fn main(){
    let listener = TcpListener::bind("127.0.0.1:8888").await.unwrap();

    loop {
        let (client, client_sock_addr) = listener.accept().await.unwrap();
        tokio::spawn(async move {
          // 该任务负责处理client
        });
    }
}
```

此外，tokio的监听套接字可和标准库的监听套接字(std::TcpListener)来回转换。由于tokio只提供了成品套接字，无法设置很多的套接字选项，因此如果需要修改或设置某些套接字选项，需要先构建标准库的套接字并设置选项，然后使用from_std()将标准库套接字转换为tokio的套接字。与from_std()对应的是into_std()。

5.3 tokio::net::TcpSocket

TcpSocket用于创建和设置套接字选项，它是未进行连接的套接字，可通过bind()和listen()操作得到服务端的监听套接字，可通过connect()得到客户端的套接字。
例如，创建监听套接字，下面的操作等价于TcpListener.bind()操作，它将监听127.0.0.1:8080端口：

```
use tokio::net::TcpSocket;

#[tokio::main]
async fn main() {
    let addr = "127.0.0.1:8080".parse().unwrap();
    let socket = TcpSocket::new_v4().unwrap();
    socket.set_reuseaddr(true).unwrap();
    socket.bind(addr).unwrap();

    let listener = socket.listen(1024).unwrap();
}
```

下面的操作等价于TcpStream::connect()操作，它将连接127.0.0.1:8080并返回该连接的TcpStream：

```
use tokio::net::TcpSocket;

#[tokio::main]
async fn main() {
    let addr = "127.0.0.1:8080".parse().unwrap();

    let socket = TcpSocket::new_v4().unwrap();
    let stream = socket.connect(addr).await.unwrap();
}
```

5.4 TcpStream

TcpStream代表客户端和服务端之间已经建立的可以进行数据通信的TCP连接。当然，TcpStream也提供了connect()方法来方便地建立和TCP服务端的连接。

```
let mut stream = TcpStream::connect("127.0.0.1:8080").await.unwrap();
```

TcpStream用于客户端和服务端的通信，因此可对其进行读和写。读操作表示接收来自对端发送过来的数据，写操作表示将数据通过TCP连接发送给对端。但是，通常会使用tokio::io::AsyncReadExt和tokio::io::AsyncWriteExt提供的读写API来读写TcpStream，因尚未介绍tokio::io，因此先跳过相关的读写操作。
TcpStream本身也提供了和读写相关的一些api：

- readable(): 等待TcpStream有数据可读
- writable(): 等待TcpStream可写入数据
- ready(): 类似Linux的select系统调用，注册可读、可写、读写关闭等事件后等待这些事件的出现

- try_read(): 尝试以不等待的方式读取TcpStream
- try_read_buf(): 尝试以不等待的方式读取TcpStream，并将读取成功的数据追加到给定的buf中
    - 和try_read()不同的是，try_read()每次读取数据后都会从前向后覆盖buf的字节，而try_read_buf()则是将读取的数据追加到buf的尾部
- try_read_vectored(): 尝试以不等待的方式读取TcpStream，并将读取成功的数据分别填充到给定的一个或多个buf中
    - 例如，给定了两个64K大小的buf，读取了100K数据，则前64K填充到第一个buf中，剩余的36K填充到第二个buf中
- try_write(): 尝试以不等待的方式写入TcpStream
- try_write_vectored(): 尝试以不等待的方式写入TcpStream，写入的数据源来自于给定的一个或多个buf
- peek(): 从TcpStream中读取数据，但不消费TcpStream中本次读取的数据。即，peek后还可以再次读取这部分数据
- split(): 将TcpStream的读和写进行分离，得到的读、写两端不可跨线程(或任务)
- into_split(): 将TcpStream的读和写进行分离，得到的读、写两端可跨线程(或任务)

稍后将简单介绍这些和读写相关的API的基本用法。

除了以上和IO相关的API，TcpSteam还提供了几个TCP连接选项设置的API：

- set_linger(): 修改TCP连接的SO_LINGER选项。在关闭连接时如果仍有未发送数据(比如仍然在缓冲等待着更多数据进入)，设置该选项决定是否要等待一段时间(期待后续会将缓冲的数据发送出去)才允许关闭TCP连接。若不设置该选项，则默认不等待
- linger(): 获取linger设置的值
- set_nodelay(): 修改TCP连接的TCP_NODELAY选项。设置该选项后，写入TcpStream的数据都将立即发送，而不会缓冲并等待凑够数据后才发送
- nodelay(): 是否设置了nodelay选项

再来介绍TcpStream提供的和读写相关的API。
通常，读相关的操作(try_read、peek等)会结合readable()来使用，写相关的操作(try_write)会结合writable()来使用。但是注意，即便readable()、writable()的返回分别代表了可读和可写，但这个可读、可写的就绪事件并不能确保真的可读可写，因此读、写时要做好判断。
例如，readable()结合try_read()：

```
use tokio::net::TcpStream;
use std::io;

#[tokio::main]
async fn main() {
    let stream = TcpStream::connect("127.0.0.1:8080").await.unwrap();
    let mut msg = vec![0; 1024];

    loop {
        // 等待可读事件的发生
        stream.readable().await.unwrap();

        // 即便readable()返回代表可读，但读取时仍然可能返回WouldBlock
        match stream.try_read(&mut msg) {
            Ok(n) => {    // 成功读取了n个字节的数据
                msg.truncate(n);
                break;
            }
```

```
            Err(ref e) if e.kind() == io::ErrorKind::WouldBlock => {
                continue;
            }
            Err(e) => {
                return;
            }
        }
    }

    println!("GOT = {:?}", msg);
}
```

当然，读写操作也可以结合ready()来使用，调用ready()时可注册感兴趣的事件，当注册的事件之一发生之后，ready()将返回Ready结构体，Ready结构体有一些布尔判断方法，用来判断某个事件是否发生。例如：

```
use tokio::io::Interest;
use tokio::net::TcpStream;
use std::io;

#[tokio::main]
async fn main() {
    let stream = TcpStream::connect("127.0.0.1:8080").await.unwrap();

    loop {
        // 注册可读和可写事件，并等待事件的发生
        let ready = stream.ready(Interest::READABLE |
Interest::WRITABLE).await.unwrap();

        // 如果注册的事件中，发生了可读事件，则执行如下代码
        if ready.is_readable() {
            let mut data = vec![0; 1024];
            match stream.try_read(&mut data) {
                Ok(n) => {
                    println!("read {} bytes", n);
                }
                Err(ref e) if e.kind() == io::ErrorKind::WouldBlock => {
                    continue;
                }
                Err(e) => {
                    return;
                }
            }
        }

        // 如果注册的事件中，发生了可写事件，则执行如下代码
        if ready.is_writable() {
            match stream.try_write(b"hello world") {
                Ok(n) => {
                    println!("write {} bytes", n);
                }
                Err(ref e) if e.kind() == io::ErrorKind::WouldBlock => {
                    continue
                }
                Err(e) => {
                    return;
```

```
                }
            }
        }
    }
}
```

peek()可读取TcpStream中的数据，但是和其它读取操作不同，peek()读取之后不会消费TcpStream中的数据。

```
use tokio::net::TcpStream;
use tokio::io::AsyncReadExt;

#[tokio::main]
async fn main() {
    let mut stream = TcpStream::connect("127.0.0.1:8080").await.unwrap();
    let mut b1 = [0; 10];
    let mut b2 = [0; 10];

    let n = stream.peek(&mut b1).await.unwrap();
    let n1 = stream.read(&mut b2[..n]).await.unwrap();
}
```

比较关键的是split()方法。TCP连接是全双工通信的，无论是TCP连接的客户端还是服务端，每一端都可以进行读操作和写操作。为了方便描述，此处将其称为读端和写端。即，客户端有读端和写端，服务端也有读端和写端。
通过TcpStream，可进行读操作，也可以进行写操作，正如前面几个示例代码所示。但是，通过TcpStream同时进行读写有时候会很麻烦，甚至无解。很多时候，需要将TcpStream的读端和写端进行分离，然后将分离的读、写两端放进独立的异步任务中去执行读或写操作(此时需跨线程)，即一个线程(或异步任务)负责读，另一个线程(或异步任务)负责写。
split()和into_split()正是用来分离TcpStream的读写两端的。
split()可将TcpStream分离为ReadHalf和WriteHalf，ReadHalf用于读，WriteHalf用于写。

```
let mut conn = TcpStream::connect("127.0.0.1:8888").await.unwrap();
let (mut read_half, mut write_half) = conn.split();
```

split()并没有真正将TcpStream的读写两端进行分离，仅仅只是引用TcpStream中的读端和写端。因此，split()得到的读写两端只能在当前任务中进行读写操作，不允许跨线程跨任务。
into_split()是split()的owned版，分离后可得到OwnedReadHalf和OwnedWriteHalf。它是真正地分离TcpStream的读写两端，它会消费掉TcpStream。OwnedReadHalf和OwnedWriteHalf可跨任务进行读写操作。

```
let conn = TcpStream::connect("127.0.0.1:8888").await.unwrap();
let (mut read_half, mut write_half) = conn.into_split();
```

请记住TcpStream的split()和into_split()方法，这两个方法在tokio网络编程时非常常用。

5.5 异步io示例二网络io

网络IO是最常见的IO方式之一，下面是一个非常简单的Client/Server两端通信中的服务端的示例。该示例中，Client/Server两端协议好以行为单位传输数据。

下面是服务端的代码：

```
use tokio::{
    io::{AsyncBufReadExt, AsyncWriteExt},
    net::{
        tcp::{OwnedReadHalf, OwnedWriteHalf},
        TcpListener, TcpStream,
    },
    sync::mpsc,
};

#[tokio::main]
async fn main() {
    let server = TcpListener::bind("127.0.0.1:8888").await.unwrap();
    while let Ok((client_stream, client_addr)) = server.accept().await {
        println!("accept client: {}", client_addr);
        // 每接入一个客户端的连接请求，都分配一个子任务，
        // 如果客户端的并发数量不大，为每个客户端都分配一个thread，
        // 然后在thread中创建tokio runtime，处理起来会更方便
        tokio::spawn(async move {
            process_client(client_stream).await;
        });
    }
}

async fn process_client(client_stream: TcpStream) {
    let (client_reader, client_writer) = client_stream.into_split();
    let (msg_tx, msg_rx) = mpsc::channel::<String>(100);

    // 从客户端读取的异步子任务
    let mut read_task = tokio::spawn(async move {
        read_from_client(client_reader, msg_tx).await;
    });

    // 向客户端写入的异步子任务
    let mut write_task = tokio::spawn(async move {
        write_to_client(client_writer, msg_rx).await;
    });

    // 无论是读任务还是写任务的终止，另一个任务都将没有继续存在的意义，因此都将另一个任务也终止
    if tokio::try_join!(&mut read_task, &mut write_task).is_err() {
        eprintln!("read_task/write_task terminated");
        read_task.abort();
        write_task.abort();
    };
}

/// 从客户端读取
async fn read_from_client(reader: OwnedReadHalf, msg_tx: mpsc::Sender<String>) {
    let mut buf_reader = tokio::io::BufReader::new(reader);
    let mut buf = String::new();
    loop {
        match buf_reader.read_line(&mut buf).await {
            Err(_e) => {
```

```
                eprintln!("read from client error");
                break;
            }
            // 遇到了EOF
            Ok(0) => {
                println!("client closed");
                break;
            }
            Ok(n) => {
                // read_line()读取时会包含换行符，因此去除行尾换行符
                // 将buf.drain(。。)会将buf清空，下一次read_line读取的内容将从头填充而不
是追加
                buf.pop();
                let content = buf.drain(..).as_str().to_string();
                println!("read {} bytes from client. content: {}", n, content);
                // 将内容发送给writer，让writer响应给客户端，
                // 如果无法发送给writer，继续从客户端读取内容将没有意义，因此break退出
                if msg_tx.send(content).await.is_err() {
                    eprintln!("receiver closed");
                    break;
                }
            }
        }
    }
}

/// 写给客户端
async fn write_to_client(writer: OwnedWriteHalf, mut msg_rx:
mpsc::Receiver<String>) {
    let mut buf_writer = tokio::io::BufWriter::new(writer);
    while let Some(mut str) = msg_rx.recv().await {
        str.push('\n');
        if let Err(e) = buf_writer.write_all(str.as_bytes()).await {
            eprintln!("write to client failed: {}", e);
            break;
        }
    }
}
```

参考：
https://course.rs/async-rust/async/getting-started.html