理解tokio核心(1): runtime

在使用tokio之前，应当先理解tokio的核心概念：runtime和task。只有理解了这两个核心概念，才能正确地、合理地使用tokio。本文先详细介绍runtime这个核心概念，还会介绍一些基本的调度知识，这些都是理解异步理解tokio的必要知识，后面再专门介绍task。

创建tokio Runtime

要使用tokio，需要先创建它提供的异步运行时环境(Runtime)，然后在这个Runtime中执行异步任务。

使用`tokio::runtime`

创建Runtime：

```
use tokio;

fnmain() {
// 创建runtime
let rt = tokio::runtime::Runtime::new().unwrap();
}
```

也可以使用Runtime Builder来配置并创建runtime：

```
use tokio;

fnmain() {
// 创建带有线程池的runtime
let rt = tokio::runtime::Builder::new_multi_thread()
        .worker_threads(8)    // 8个工作线程
        .enable_io()                // 可在runtime中使用异步IO
        .enable_time()            // 可在runtime中使用异步计时器(timer)
        .build()                        // 创建runtime
        .unwrap();
}
```

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。

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。在后面深入了一些tokio后，我会再花一个小节来解释单一线程的runtime和多线程的runtime的调度区别以及如何选择合适的runtime。

默认情况下(比如以上两种方式)，创建出来的runtime都是多线程runtime，且没有指定工作线程数量时，默认的工作线程数量将和CPU核数(虚拟核，即CPU线程数)相同。

只有明确指定，才能创建出单一线程的runtime。例如：

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

例如，创建一个多线程的runtime，查看其线程数：

```
use  tokio;

fnmain(){
let  rt  =  tokio::runtime::Runtime::new().unwrap();
    std::thread::sleep(std::time::Duration::from_secs(10));
}
```

在另一个终端查看线程数：

```
$  ps  -eLf  |  grep  'targe[t]'
longshu+  15759      62  15759    6    9  20:42  pts/0    00:00:00  target/debug/async  main
longshu+  15759      62  15761    0    9  20:42  pts/0    00:00:00  target/debug/async  main
longshu+  15759      62  15762    0    9  20:42  pts/0    00:00:00  target/debug/async  main
longshu+  15759      62  15763    0    9  20:42  pts/0    00:00:00  target/debug/async  main
longshu+  15759      62  15764    0    9  20:42  pts/0    00:00:00  target/debug/async  main
longshu+  15759      62  15765    0    9  20:42  pts/0    00:00:00  target/debug/async  main
longshu+  15759      62  15766    0    9  20:42  pts/0    00:00:00  target/debug/async  main
longshu+  15759      62  15767    0    9  20:42  pts/0    00:00:00  target/debug/async  main
longshu+  15759      62  15768    0    9  20:42  pts/0    00:00:00  target/debug/async  main
```

总共9个OS线程，其中8个worker thread(我的电脑是4核8线程的)，外加一个main thread。

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]
asyncfnmain()  {}
```

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))]
```

它们等价于如下没有使用`#[tokio::main]`

的代码：

```
fnmain(){
```

```
    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 { ... })
}
```

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。

例如：

```
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();
}
```

对于4核8线程的电脑，此时总共有19个OS线程：16个worker-thread，2个spawn-thread，一个main-thread。

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。

进入runtime: 在异步runtime中执行异步任务

提供了Runtime后，可在Runtime中执行异步任务。

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。但是介绍tokio用法时，不需要那么复杂，只需使用tokio的异步timer即可解释清楚，如`tokio::time::sleep()`

。

> 注：`std::time`
> 也提供了sleep()，但它会阻塞整个线程，而`tokio::time`
> 中的sleep()则只是让它所在的任务放弃CPU并进入调度队列等待被唤醒，它不会阻塞任何线程，它所在的线程仍然可被用来执行其它异步任务。因此，在tokio runtime中，应当使用`tokio::time`
> 中的sleep()。

例如：

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

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

输出：

```
before sleep: 2021-10-24 11:53:38.496
after sleep: 2021-10-24 11:53:40.497
```

上面调用了runtime的`block_on()`

方法，`block_on`

要求一个`Future`

作为参数，可以像上面一样直接使用一个`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，后面介绍blocking thread时会再次说明block_on的阻塞问题。

`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};

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

spawn: 向runtime中添加新的异步任务

在上面的例子中，直接将`async {}`

作为`block_on()`

的参数，这个`async {}`

本质上是一个Future，即一个异步任务。在这个最外层的异步任务内部，还可以创建新的异步任务，它们都将在同一个runtime中执行。

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。

```
use std::thread;

use chrono::Local;
```

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

fnnow() -> 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
fnasync_task() {
println!("create an async task: {}", now());
    tokio::spawn(async {
        time::sleep(time::Duration::from_secs(10)).await;
println!("async task over: {}", now());
    });
}

fnmain() {
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}
fnasync_task(rt: &Runtime) {
    rt.spawn(async {
        time::sleep(time::Duration::from_secs(10)).await;
    });
}

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

进入runtime: 非阻塞的enter()

`block_on()`是进入runtime的主要方式。但还有另一种进入runtime的方式：`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。

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

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

fnmain() {
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);

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

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。

> 从这一点看来，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。

大方向上来说，有两种调度策略：抢占式调度和协作式调度。抢占式调度策略，调度器会在合适的时候(调度规则决定什么是合适的时候)强行切换当前正在执行的调度单元(例如进程、线程)，避免某个任务长时间霸占CPU从而导致其它任务出现饥饿。协作式调度策略则不会强行切断当前正在执行的单元，只有执行单元执行完任务或主动放弃CPU，才会将该执行单元重新排队等待下次调度，这可能会导致某个长时间计算的任务霸占CPU，但是可以让任务充分执行尽早完成，而不会被中断。

对于面向大众使用的操作系统(如Linux)通常采用抢占式调度策略来保证系统安全，避免恶意程序霸占CPU。而对于语言层面来说，通常采用协作式调度策略，这样既有底层OS的抢占式保底，又有协作式的高效。tokio的调度策略是协作式调度策略。

也可以简单粗暴地去理解异步调度：任务刚出生时，放进任务队列尾部，调度器总是从任务队列的头部选择任务去执行，执行任务时，如果任务有阻塞操作，则该任务总是会被放入到任务队列的尾部。如果任务队列的第一个任务都是阻塞的(即任务之前被阻塞但目前尚未完成)，则调度器直接将它重新放回队列的尾部。因此，调度器总是从前向后一次又一次地轮询这个任务队列。当然，调度算法通常会比这种简单的方式要复杂的多，它可能会采用多个任务队列，多种挑选标准，且队列不是简单的队列，而是更高效的数据结构。

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。

tokio的两种线程：worker thread和blocking thread

需要注意，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。

但是，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被饿很长一段时间。

例如，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。

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

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

fnmain()  {
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());
        });
}
```

输出：

```
in task: 2021-10-25 19:01:00
not blocking: 2021-10-25 19:01:00
after blocking task: 2021-10-25 19:01:10
```

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。

关闭Runtime

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。

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

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

fnmain() {
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(注意结果中没有异步任务中`println!()`输出的内容)：

```
worker thread task over 2: 2021-10-25 20:08:35
blocking thread task over: 2021-10-25 20:08:36
runtime droped: 2021-10-25 20:08:36
```

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，将强行终止runtime中的所有线程。后者是立即强行关闭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());
}
```

输出：

```
runtime droped: 2021-10-25 20:16:02
```

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。

runtime 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。

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。

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。

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

fn main(){
let t1 = thread::spawn(||{
let rt = tokio::runtime::Builder::new().new_current_thread()
                                .enable_all()
                                .build()
                                .unwrap();
        rt.block_on(...)
    });

let t2 = thread::spawn(||{
let rt = tokio::runtime::Builder::new().new_current_thread()
                                .enable_all()
                                .build()
                                .unwrap();
        rt.block_on(...)
    });
```

```
    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。