理解tokio核心(2): task

本篇是介绍tokio核心的第二篇，理解tokio中的task。

何为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。

解释了何为绿色线程后，回到tokio的task概念。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。

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。

task::spawn()

这个很简单，就是直接在当前的runtime中生成一个异步任务。

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

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

fnmain()  {
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));
}
```

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。

```
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");
```

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。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。

```
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。

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

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

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。

```
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。

task::unconstrained()

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()`
。

task::spawn_local()

关于`spawn_local()`

，后面介绍LocalSet的时候再一起介绍。

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。

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

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

        rt.block_on(async {
let task = tokio::task::spawn(async {
                    time::sleep(time::Duration::from_secs(10)).await;
                });

// 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: JoinError = task.await.unwrap_err();
println!("{}", abort_err.is_cancelled());
        })
}
```

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。

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。

例如：

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

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

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

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

fnmain() {
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: 2021-11-02 16:51:36
doing two: 2021-11-02 16:51:36
do two done: 2021-11-02 16:51:37
do one done: 2021-11-02 16:51:38
all done: 2021-11-02 16:51:38
```

下面是官方文档中`try_join!`的示例：

```
asyncfndo_stuff_async() -> Result<(), &'staticstr> {
// async work
}

asyncfnmore_async_work() -> Result<(), &'staticstr> {
// more here
}

#[tokio::main]
asyncfnmain() {
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);
                }
        }
}
```

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。

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

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

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

    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;
```

```
    });
}
```

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：

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

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

fnmain() {
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awaitLocalSet的时候，它们会等待调度并运行。

```
use chrono::Local;
```

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

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

fnmain() {
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: 2021-10-2620:19:11
local task1
local task3
local task2
local task1 done 2021-10-2620:19:13
local task3 done: 2021-10-2620:19:14
```

```
re enter localset context: 2021-10-2620:19:29
local task2 done, 2021-10-2620:19:29
re enter localset context done: 2021-10-2620:19:30
all local tasks done: 2021-10-2620:19:30
```

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。

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

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

fnmain() {
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: 2021-10-26 21:23:18
local task2
local task1
local task2 done: 2021-10-26 21:23:21
local task3
local task1 done 2021-10-26 21:23:31
local task3 done: 2021-10-26 21:23:31
all local tasks done: 2021-10-26 21:23:31
```

tokio::select!宏

在Golang中有一个select关键字，tokio中则类似地提供了一个名为`select!`的宏。`tokio::select!`

宏使用场景非常普遍，因此有必要理解该宏的工作流程。

`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! {
    = <async expression 1> (, if )? => ,        // branch 1
    = <async expression 2> (, if )? => ,        // branch 2
    ...
    (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! {
    = <async expression 1> => ,        // branch 1
    = <async expression 2> => ,        // 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：

```
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：

```
#[tokio::main]
asyncfnmain()  {
letmut  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;  assert_eq!(count,  1);  }
                        _  =  async  {},  if  count  <  2  =>  {  count  +=  1;  assert_eq!(count,  2);  }
                        _  =  async  {},  if  count  <  3  =>  {  count  +=  1;  assert_eq!(count,  3);  }
                        _  =  async  {},  if  count  <  4  =>  {  count  +=  1;  assert_eq!(count,  4);  }
else  =>  {  break;  }
                };
        }
}
```

另外，上面的例子中将`select!`
放进了一个loop循环中，这是很常见的用法。对于上面的例子来说，如果注释掉`biased`
，那么在第一轮循环中，由于`select!`
中的4个if前置条件均为true，因此按照随机的顺序推进这4个异步任务。由于上面示例中的异步任务表达式不做任何事，因此第一个被推进的异步任务会先完成，`selcet!`
将取消剩余3个任务，假如先完成任务的分支的断言通过，那么`select!`
返回后将进入下一轮loop循环，重新调用一次`select!`
宏，重新评估if条件，这次将只有3个分支通过检测，不通过的那个分支将被禁用，`select!`
将按照随机顺序推进这3个分支。