Tokio

配置

Cargo.toml

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

```

Tokio main

> See main in tokio - Rust (docs.rs)

```rust
#[tokio::main]
async fn main() {
    println!("Hello world");
}

```

未使用`#[tokio::main]`的等效代码

```rust
fn main() {
    tokio::runtime::Builder::new_multi_thread()
        .enable_all()
        .build()
        .unwrap()
        .block_on(async {
            println!("Hello world");
        })
}

```

Pin

```rust
use tokio::pin;

async fn my_async_fn() {
    // async logic here
}

#[tokio::main]
async fn main() {
    let future = my_async_fn();
    pin!(future);

    (&mut future).await;
}

```

支持一次性执行这两项操作的宏的变体：

```
pin! {
    let future1 = my_async_fn();
    let future2 = my_async_fn();
}
```

生成任务

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

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

    loop {
        let (socket, _) = listener.accept().await.unwrap();
        // 为每一条连接都生成一个新的任务，
        // `socket` 的所有权将被移动到新的任务中，并在那里进行处理
        // 返回一个 JoinHandle 类型的句柄
        tokio::task::spawn(async move {
            process(socket).await;
        });

        // 允许阻塞
        // tokio::task::spawn_blocking()
    }
}
```

Send约束

> See 创建异步任务 - Rust语言圣经(Rust Course)

一个任务要实现 `Send` 特征，那它在 `.await` 调用的过程中所持有的全部数据都必须实现 `Send` 特征。当 `.await` 调用发生阻塞时，任务会让出当前线程所有权给调度器，然后当任务准备好后，调度器会从上一次暂停的位置继续执行该任务。该流程能正确的工作，任务必须将`.await`之后使用的所有状态保存起来，这样才能在中断后恢复现场并继续执行。若这些状态实现了 `Send` 特征(可以在线程间安全地移动)，那任务自然也就可以在线程间安全地移动。

例如以下代码可以工作:

```
use tokio::task::yield_now;
use std::rc::Rc;

#[tokio::main]
async fn main() {
    tokio::spawn(async {
        // 语句块的使用强制了 `rc` 会在 `.await` 被调用前就被释放，
        // 因此 `rc` 并不会影响 `.await`的安全性
        {
            let rc = Rc::new("hello");
            println!("{}", rc);
        }

```

```
        // `rc` 的作用范围已经失效，因此当任务让出所有权给当前线程时，它无需作为状态被保存起来
        yield_now().await;
    });
}
```

但是下面代码就不行：

```
use tokio::task::yield_now;
use std::rc::Rc;

#[tokio::main]
async fn main() {
    tokio::spawn(async {
        let rc = Rc::new("hello");

        // `rc` 在 `.await` 后还被继续使用，因此它必须被作为任务的状态保存起来
        yield_now().await;

        // 事实上，注释掉下面一行代码，依然会报错
        // 原因是：是否保存，不取决于 `rc` 是否被使用，而是取决于 `.await`在调用时是否仍然处于 `rc` 的作用域中
        println!("{}", rc);

        // rc 作用域在这里结束
    });
}
```

迭代

目前， Rust 语言还不支持异步的 `for` 循环，因此我们需要 `while let` 循环和 `StreamExt::next()` 一起使用来实现迭代的目的：

```
use tokio_stream::StreamExt;

#[tokio::main]
async fn main() {
    let mut stream = tokio_stream::iter(&[1, 2, 3]);

    while let Some(v) = stream.next().await {
        println!("GOT = {:?}", v);
    }
}
```