今天我们继续高并发的话题，在上次的博客中我们有提到，Rust的Future机制非常有助于程序员按照更为自然、简洁的逻辑去设计系统，我们必须要知道高并发系统的关键在于立交桥的分流与导流构造而非信号灯的限流。因此把精力放在设计锁、互斥系这些信号系统上是非常事倍功半的。

从机制上来讲Rust从函数式语言借鉴而来的Future机制是先进的，而且从亲身教小孩编程的时候笔者意外发现，对于没有任何编程经验的人来说，他们学习async/await的成本，要比理解层层回调的机制要低得多。程序员在学习Future的难度大，其实完全是因为之前的历史包袱太重了。

为什么说Future更像自然语言

在以下这段代码中，网络连接socket、请求发送request、响应接收response三个对象全部都是future类型的，也就是在代码执行之后不会被执行也没有值仅有占位的意义，当未来执行后才会有值返回，and\_then方法其实是在future对象执行成功后才会被调用的方法，比如read\_to\_end这行代码就是在request对象执行成功后，调用read\_to\_end方法对读取结果。

use futures::Future;

use tokio\_core::reactor::Core;

use tokio\_core::net::TcpStream;

fn main() {

let mut core = Core::new().unwrap();

let addr = "127.0.0.1:8080".to\_socket\_addrs().unwrap().next().unwrap();

let socket = TcpStream::connect(&addr, &core.handle());

let request = socket.and\_then(|socket|{

tokio\_core::io::write\_all(socket, "Hello World".as\_bytes())

});

let response = request.and\_then(|(socket, \_)| {

tokio\_core::io::read\_to\_end(socket, Vec::new())

});

let (\_, data) = core.run(response).unwrap();

println!("{}", String::from\_utf8\_lossy(&data));

}

而想象一下如果是传统编程所采用的方式，需要在网络连接完成后调用请求发送的回调函数，然后再请求发送的响应处理方法中再注册接收请求的回调函数，复杂不说还容易出错。

而future机制精髓之处在于，整个过程是通过core.run(response).unwrap();这行代码运行起来的，也就是说开发人员只需要关心最终的结果就可以了。从建立网络连接开始的调用链交给计算机去帮你完成，最终的效率反而还会更高。

并发中的poll模式到底是什么意思？

笔者看到不少博主在介绍Rust的Future等异步编程框架时都提到了Rust的Future采用poll模式，不过到底什么是poll模式却大多语焉不详。

笔者还是这样的观点，程序员群体之所以觉得future机制难以理解，其关键在于思维模式被计算机的各种回调机制给束缚住了，而忘记了最简单直接的方式。在解决这个问题之前我们先来问一个问题，假如让我们自己设计一个类似于goroutine之类事件高度管理器，应该如何入手？

最直接也是最容易想到的方案就是事件循环，定期遍历整个事件队列，把状态是ready的事件通知给对应的处理程序，这也是之前mfc和linux的select的方案，这实际上也就是select方案；另外一种做法是在事件中断处理程序中直接拿到处理程序的句柄，不再遍历整个事件队列，而是直接在中断处理响应中把通知发给对应处理进程，这就是Poll模式。

多路复用是另一种机制，这种机制可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。笔者在前文《这位创造了Github冠军项目的老男人，堪称10倍程序员本尊》中曾经介绍过Tdengine的定时器，其中就有这种多路复用的思想。由于操作系统timer的处理程序还不支持epoll的多路复用，因此每注册一个timer就必须要启动一个线程进行处理，资源浪费严重，因此Tdengine自己实现了一个多路复用的timer，可以做到一个线程同时处理多个timer，这些细节上的精巧设计也是Tdengine封神的原因之一。

Epoll的代价-少量连接场景不适用

当然epoll还有一个性能提升的关键点，那就是使用红黑树做为事件队列的存储模型，我们在上文《用了十年竟然都不对，Java、Rust、Go主流编程语言的哈希表比较》中曾经提到过，红黑树是一种解决哈希碰撞时比较好的退化选择，不过这也给epoll机制带来了一些适用场景的限制，如果连接总数本身就不高的情况下，那么epoll可能还不如select高效。其原因同时也在《用了十年竟然都不对，Java、Rust、Go主流编程语言的哈希表比较》中说明了，由于红黑树在内存中也是散列的状态，这就会造成连续存储的数据在总长度较小的情况下获得比红黑树更好的性能，具体这里就不加赘述了。

ET还是LT如何触发又是个选择

Epoll的触发又分为水平触发和垂直触发两种模式，具体介绍如下：

LT（level triggered）水平触发，是缺省的工作方式，顾名思义，也就是即使状态不变也可能模式通知的模式，同时支持阻塞和非阻塞两种方案.在这种做法中，内核通知注册的进程一个有任务已经就绪，不过这种模式下就算进程不作任何操作，内核还是会继续通知，所以这种模式属于唐僧式的模式，虽然唠叨但出BUG的可能性要小一点。

ET （edge-triggered），垂直触发，也就是当且仅当有任务状态发生变化时才会被触发，属于高速工作方式。在ET模式下仅当有事件从未就绪变为就绪时，内核才会触发通知。但是内核的通知只会发出一次，也就是说如果事件一直没有进程处理，内核也不会发送第二次通知。

其实从代码来看ET和LT的差别不多，具体如下：

if (epi->event.events & EPOLLONESHOT)

epi->event.events &= EP\_PRIVATE\_BITS;

else if (!(epi->event.events & EPOLLET)) { //如果是是LT模式，当前事件会被重新放到epoll的就绪队列。

list\_add\_tail(&epi->rdllink, &ep->rdllist);

ep\_pm\_stay\_awake(epi);

}

可以看到LT模式从不会丢弃事件，只要队列里还有数据能够读到，就会不断的发起通知，属于链式反应的一种，效率低点但不容易出错，而ET只在则只在新事件到来时才会发起通知，效率高但也容易出BUG。当然如果socketfd事件与处理线程之间是一对多的关系，也就是说一个socketfd只对应一个线程，那倒也还好说。但由于在很多高并发的场景下，很多socketfd是由多个进程同时监控的，因此这又会造成一个惊群的问题。

正如前文所说，多路复用机制也允许多个进程(线程)在等待同一个事件的到来，当这个 fd(socket)的事件发生的时候，这些睡眠的进程(线程)就会被同时唤醒，去处理这个事件，这和一大群鱼，争抢一个鱼食的现象非常类似，因此也就被称为"惊群"现象。

由于大量的进程计算资源被浪费在被抢食的过程中，实际上却没做任何有意义的工作，因此"惊群"效率低下，而且在鱼群抢食的过程中，会造成系统短暂的吞吐能力下降。对于流量分布极不均衡的系统来说，惊群的影响很大。

不过在LT模式下，通知是链式的，因此惊群难以避免，ET模式下效率虽多，但如果有一个进程出现问题，则很有可能造成难以察觉的BUG，高并发系统绝对是个说起来容易，做起来难的设计。

————————————————

版权声明：本文为CSDN博主「beyondma」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/BEYONDMA/article/details/119480735