# **01-Rust std::net 简介**

Rust 的标准库 std::net 提供了对 TCP/IP 协议栈使用的封装。但 std::net 是同步的，如果需要异步高性能网络，可以使用 tokio::net，而且两者的对外 api 几乎一模一样。

std::net 的内容：

TCP：TcpListener / TcpStream 分别用来处理服务器的监听和客户端的连接；

UDP：UdpSocket 用来处理基于 UDP 的 Socket；

IpAddr 用来处理 IPv4 / IPv6 地址封装；SocketAddr 用来封装 IP + port 信息。

# **02-处理网络链接的一般方法**

TcpListener / TcpStream 的使用方法如下：

/// server 端// 监听特定端口let listener = TcpListener::bind("127.0.0.1:9527").await?;loop { // 等待客户端连接 let (stream, addr) = listener.accept().await?; // 通常做法是创建线程来处理客户端连接，避免长时间阻塞主线程}/// client 端// 链接到特定端口let stream = TcpStream::connect("127.0.0.1:9527").await?;// 之后就可以与 server 收发消息了

复制代码

从上面 api 可以看出，Rust 得益于所有权原则和 Drop trait，不需要像 Java 一样，写大量释放资源的样板代码。

## **02.1-**如何处理大量连接？****

按照上面使用新线程处理客户端连接的方式，系统存在一个瓶颈，即所谓的”C10K”，当连接数达到万这个级别，系统会遭遇到资源和算力的双重瓶颈。从资源角度，Rust 中栈帧默认为 2M，10k 个连接需要 20G；从算力角度，太多线程切换上下文消耗太大。

处理大量连接（超过 C10K，到达 C10M），需要通过用户态协程。Rust 中支持异步处理的无栈协程。

## **02.2-如何处理共享信息？**

对于需要线程间共享的数据，如果是只读数据，可以使用 Arc<T>；如果是需要修改的数据，需要使用 Arc<RwLock<T>>。

使用锁时，被锁的资源会影响系统地吞吐量。一种解决思路是，降低锁力度。这在其他编程语言中也能看到类似思路，例如 Java HashMap 中的分段锁。另一种思路是，改变访问共享资源的方式，使其只被一个特定的线程访问；其他线程或者协程只能通过给其发消息的方式与之交互。Rust 下的 channel 都有非常棒的实现。

# **03-处理网络数据的一般方法**

处理客户端 / 服务端通信时，如果使用 HTTP 协议，JSON 通常作为首选数据结构。Rust 下，对 JSON 序列化和反序列化可以通过第三方库 [serde](https://xie.infoq.cn/link?target=https://serde.rs/" \o ")。

#[derive(Serialize, Deserialize)]#[serde(crate = "rocket::serde")]struct Hello { name: String,}

复制代码

某些情况可能需要自定义客户端 / 服务端通信使用的数据结构，此时通常会使用 protobuf。由于 protobuf 在传输过程中是二进制流，接收消息是需要某种方式来界定消息帧。可以借助 tokio 提供的 length\_delimited codec 搭配 Framed 结构使用。

// 服务端接受let (stream, addr) = listener.accept().await?; // LengthDelimitedCodec 默认 4 字节长度let mut stream = Framed::new(stream, LengthDelimitedCodec::new());  
// 客户端发送let stream = TcpStream::connect("127.0.0.1:9527").await?; let mut stream = Framed::new(stream, LengthDelimitedCodec::new());