### 1. Asio如何实现高并发，IO上下文起到了什么作用？

Asio的核心异步I/O模型允许我通过**事件驱动**的方式来处理大量并发连接，避免了传统的阻塞I/O操作。为了实现这一点，我使用了IO上下文，它在Asio中负责调度异步任务。

具体来说，我创建了一个io\_context池，根据CPU的核数创建多个线程，实现真正的**并行**，每个线程都运行一个io\_context的run()函数，这样每个线程可以独立处理多个异步I/O任务，保证高并发处理能力。

此外，为了进一步提升性能，我对I/O任务的调度进行了优化，例如合理分配任务到不同的io\_context，从而提高了系统的整体吞吐量。”

### 2. gRPC如何实现服务间高效通信？同时，如何处理gRPC的异步调用？

1. gRPC实现服务间高效通信:

在message.proto文件中定义了多个服务,如验证服务、状态服务、聊天服务等。

通过protoc编译器生成了对应的C++代码,如message.grpc.pb.h和message.grpc.pb.cc。  
在各个服务器上，使用grpc连接池来管理客户端的连接  
使用HTTP2作为传输协议，HTTP2提供了多路复用、头部压缩和二进制帧等功能，这使得多**个请求可以共享同一个TCP连接**，减少了开销并提高了传输效率。（共享一个TCP连接就是用grpc连接池创建多个stub存根，每个stub对应一个channel）

### 3 为什么用protobuf，不用头文件里写个结构体？

因为proto支持多语言，和高并发，提供了高效的二进制序列化，

与grpc紧密结合，提供了强大的grpc框架

### 4 Grpc实现了什么功能

1.服务发现和负载均衡：

StatusServer 使用 gRPC 提供了获取可用 ChatServer 的服务。

//GetChatServerRsp StatusGrpcClient::GetChatServer(int uid)

2.跨服务器通信：

不同的 ChatServer 实例之间使用 gRPC 进行通信，例如在处理好友请求、聊天消息转发等场景中。

// AddFriendRsp ChatGrpcClient::NotifyAddFriend(std::string *server\_ip*, const AddFriendReq& *req*)

3. 用户认证：

用户登录时，允许客户端在聊天服务上进行身份验证。

LoginRsp StatusGrpcClient::Login(int *uid*, std::string *token*)

 4.连接池管理：

为了提高性能和资源利用率，项目实现了 gRPC 连接池，管理与其他服务器的连接。

服务器之间的通讯为什么使用Grpc而不使用http

为什么需要做到序列化和反序列化？

1.数据传输：网络通信通常只能传输字节流。复杂的数据结构（如对象、数组等）需要转换为字节流才能在网络上传输。

2. 跨语言通信：序列化可以将数据转换为一种通用的格式，使得不同语言的程序可以相互通信。

3. 数据持久化：序列化后的数据可以**方便地存储**到文件或数据库中，以后可以通过反序列化恢复原始数据结构。

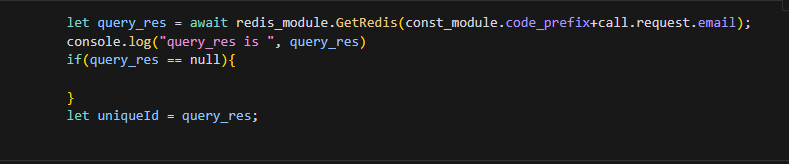
4.效率：序列化可以压缩数据，**减少网络传输的数据量**，提高通信效率。

5.安全性：某些序列化格式（如Protocol Buffers）可以提供一定程度的数据校验和安全性。

当客户端发送请求或服务器发送响应时，这些对象会被自动序列化为二进制格式，然后在接收端被反序列化回对象。这个过程是由gRPC和Protocol Buffers自动处理的，使得开发者可以专注于业务逻辑，而不必关心底层的数据传输细节。

### 5. Redis实现验证码//Nodejs参与了哪些工作，怎么叙述一下？

首先检查Redis中是否已有验证码（生成唯一的uuid，截取前4位作为验证码）,如果没有则生成新的验证码。然后,它将验证码存储在Redis中并通过邮件发送给用户。"



5.错误处理:

如果过程中Redis操作失败,我们会返回相应的错误码,确保系统的稳定性和可靠性。

**拓展：**如何防止恶意请求或暴力破解验证码。

比如，您可以通过限制同一邮箱的请求频率，或者在Redis中增加失败计数器来记录错误的验证码尝试次数，从而在多次错误输入后暂时冻结验证码请求。

Redis操作在高并发场景下，可能会遇到多个请求同时写入验证码的情况。您可以提到通过**Redis的原子操作（如SETNX**）来保证在同一时间只允许一个操作更新验证码，这样能够避免数据不一致的问题。

### 6.整个项目里边redis是怎么使用的？

根据项目代码，Redis在这个系统中主要用于以下几个方面：

 1.用户认证和会话管理：

当用户登录时，我们会在Redis中存储一个键值对，键是用户ID，值是一个包含token和用户基本信息的JSON字符串，缓存用户信息，提高访问速度

 2.服务器负载均衡：

 Redis的哈希结构来记录每个ChatServer的当前连接数。用于简单的负载均衡决策

 3.缓存数据：

 将频繁访问的用户数据缓存在Redis中。当需要用户信息时，先检查Redis，如果没有再去查询数据库，然后将结果存入Redis。这大大减少了数据库的压力。

 4.临时数据存储：对于验证码这类临时数据，我们也使用Redis存储，并设置一个合适的过期时间。这避免了这些临时数据占用永久存储空间。

 5.服务器心跳检测：

 Redis连接池会定期向Redis发送PING命令，确保连接的活跃性。如果发现连接断开，会自动重连。这增强了系统的稳定性。

Mysql每隔60s就心跳一下

### 7.在设计用户表结构时，如何确保查询的高效性？有没有使用索引或者其他优化手段？Mysql

主键索引：我们为用户表设置了**自增**的uid作为主键，可以大大提高按uid查询的速度。

唯一索引：我们为用户名（name）和邮箱（email）字段创建了**唯一索引**。这不仅确保了这些字段的唯一性，还提高了通过这些字段进行查询的效率。

合理的字段设计：我们仔细考虑了每个字段的数据类型和长度，以确保存储效率和查询性能的平衡。例如，对于uid我们使用了INT类型，对于name和email使用了VARCHAR类型并设置了合适的长度限制。

缓存策略：虽然不是直接的表结构优化，但我们实现了Redis缓存来减少对数据库的直接查询。这在代码中可以看到，例如：我们首先尝试从Redis缓存中获取用户信息，只有在缓存中没有找到时才会查询MySQL数据库。

查询优化：在查询MySQL时，我们使用了预处理语句（就是有？占位符的那种）来防止SQL注入并提高查询效率。预处理语句只需要编译一次,之后可以多次执行,减少了SQL解析和优化的开销。

**拓展：**可以提到在高并发环境下，B树索引如何保持查询的平衡，以及如何避免过多的索引导致插入和更新操作的性能下降。

可以补充说明是否有使用复合索引（联合索引）。例如，如果系统中涉及用户的登录场景，可以为(email, password)字段设置复合索引，这样能够进一步加快多条件查询。

如果数据库用户量非常大，您可以提到分库分表或者分区表来优化查询性能。分区表能够根据某个字段（如用户的uid或创建时间）将数据划分到不同的表中，提升大数据集的查询速度。

在您的项目中，您提到使用了多线程模式来提升并发性能，特别是封装了io\_context池以提升并发处理能力。

### 8.多线程高并发的？在处理线程安全问题时，您采取了哪些措施？

多线程模式来提升并发性能,主要通过以下几个方面实现:

1.我们封装了一个AsioIOServicePool类,管理多个IO上下文对象。每个IO上下文在独立的线程中运行,这样可以充分利用多核CPU的优势。

2. 任务分配机制:我们使用轮询方式将任务分配到不同的io\_context,确保负载均衡。

3. 我们大量使用Boost.Asio的异步操作,这允许我们高效处理并发请求而不阻塞线程。

4. 连接池的实现:

每个服务端都有用连接池通过预先创建多个连接对象，并在需要时从池中获取连接，使用完后再归还到池中，

管理与其他服务的gRPC连接,我们实现了连接池,减少了创建连接的开销,提高了并发性能。

在处理**线程安全**问题时,我们采取了以下措施:

1.使用线程安全的数据结构:

例如,使用std::atomic来管理共享计数器,确保在多线程环境下的安全访问。尽量减少锁的使用从而减小性能开销。

2. 我们尽量避免在不同线程间共享可变状态,每个io\_context在自己的线程中运行,减少了线程间的竞争。

3.在必须共享资源的地方,我们使用std::mutex来保护关键区域,确保线程安全。

4.智能指针的应用:

广泛使用std::shared\_ptr和std::unique\_ptr来管理资源,避免内存泄漏并提高线程安全性。

5. 使用线程安全的设计模式:

使用单例模式，确保在多线程环境下也只有一个实例被创建。

**扩展：**

* 1. 可以进一步说明为什么选择了轮询（Round-Robin）是否考虑过其他调度策略（如基于负载或任务复杂度的调度）？

各个任务的复杂度相对均衡,轮询已经能满足需求。此外,轮询的开销很小,不会给系统带来额外的负担。

* 1. 任务调度与负载均衡：  
     可以进一步说明如何确保任务在多个线程之间的均匀分配，特别是在高并发的场景下，如何避免某些io\_context线程过载的问题。
  2. 线程安全的细节：  
     使用场景问题：std::atomic如何保证原子性操作的线程安全，而std::mutex在处理哪些场景时是不可避免的？尤其是如何权衡锁的使用，以避免性能瓶颈。  
     std::atomic主要用于**计数器**等简单的共享变量,如\_nextIOService。它保证了在多线程环境下对该变量的操作是原子的,无需额外的锁。std::mutex则用于保护更复杂的共享资源,如连接池
  3. 异步与并发的权衡：  
     您提到了异步操作，进一步补充可以说明异步与多线程并发的结合点在哪里。例如，异步操作如何减轻线程的负担，如何避免线程被阻塞。

异步操作和多线程并发在我们的系统中是相辅相成的。异步操作主要用于I/O密集型任务,如网络通信,这可以在不阻塞线程的情况下处理大量并发连接。

*\_acceptor.async\_accept(new\_session->GetSocket(), std::bind(&CServer::HandleAccept, this, new\_session, placeholders::\_1));*

这里的async\_accept是一个异步操作,它不会阻塞线程,而是在有新连接时触发回调。这样,即使在处理大量并发连接时,也能保持较少的线程数,提高系统的整体效率。

### 9.描述一下分层架构是如何设计的，以及如何通过这种架构来提高系统的可维护性和扩展性？

1.网络层：

主要由Cserver（服务器连接类）和Csession（客户端会话管理类）类实现，负责处理底层的网络通信。

**CServer**: 负责处理网络连接。它的主要职责是监听端口，接受新的客户端连接，并为每个连接创建一个新的会话

**CSession**: 管理单个客户端会话。每个客户端连接都会创建一个CSession实例

AsioIOServicePool（异步IO任务池）类用于管理Boost.Asio的io\_context，提供了高效的异步I/O处理。我们确保网络层只专注于通信，所有业务逻辑交给逻辑层处理。

 2.逻辑层：

LogicSystem（逻辑处理类） 处理业务逻辑，负责处理从客户端接收到的消息，并根据消息类型调用相应的处理函数。它通过注册回调函数的方式实现消息处理的灵活性。

 3.数据访问层：

Mysql管理类和Redis管理类用于管理数据库连接和基本操作：插入，查询和更新等。

 4.配置管理：

ConfigMgr类用于管理配置信息，使得配置更改不影响核心代码。

 5.服务间通信：

使用gRPC实现服务间的通信，如ChatGrpcClient和StatusGrpcClient。

这种分层架构提高了系统的可维护性和扩展性：

 解耦：网络层和逻辑层的分离使得修改一层不会直接影响另一层。

 模块化：每个层都有明确的职责，使得代码更容易理解和维护。

 可测试性：逻辑层可以独立于网络层进行单元测试。

 可扩展性：可以轻松添加新的功能模块或替换现有模块，而不影响整体架构。

 并发处理：使用AsioIOServicePool和消息队列实现了高效的并发处理。

 配置灵活：通过ConfigMgr集中管理配置，使得系统更容易适应不同环境。

 服务解耦：使用gRPC进行服务间通信，使得各个服务可以独立开发和部署。

### 10.为什么要使用单例模板类//手动封装了单例模板类管理不同的场景，然后用智能指针去管理

项目中定义了一个通用的单例模板类 Singleton<T>，可以在 Singleton.h 文件中看到：

在单例类中，使用 std::shared\_ptr 来管理单例实例：

在我的项目里边有很多类继承来自这个单例模板类，需要用到的时候我就用getInstance方法来获取实例。

配置管理类，redis管理类，asio io服务池，chatgrpc客户端，状态grpc客户端。都使用了连接池来管理 gRPC 连接，提高性能和资源利用率。

确保全局只有一个实例，避免资源浪费和冲突。

提供一个全局访问点，方便其他组件获取服务。

在整个应用中共享状态，如配置信息、连接池等。

**用智能指针管理的：**确保了单例对象在多线程环境下的安全初始化和访问。

可以在发生异常时自动释放资源，通过返回 std::shared\_ptr<T>为所有使用单例的代码提供了一致的接口，简化了使用方式

**智能指针可以更精确地控制对象的生命周期，特别是当单例对象需要在特定时间被销毁时。**

### 11..项目里为什么使用懒汉式

 这个项目里用的就是懒汉式的，

1.聊天服务器可能需要管理多个资源组件（如数据库连接等）。懒汉式允许这些组件只在真正需要时才初始化，避免在启动时就占用大量资源。

2.尤其是在分布式系统里边，根据运行时条件决定需不需要这些组件

3.懒汉式可以加快服务器的启动速度，因为不需要在启动时就初始化所有组件。看起来像是更快地就绪了。

4.如果组件初始化失败了，能处理，而不是程序启动的时候就崩溃。

5.可以更友好的进行单元测试。

6.支持动态配置更新，更容易实现组件的重新初始化

 懒汉式：实例在第一次被请求时才创建（延迟初始化）。

 饿汉式：实例在程序启动时就创建。

为什么需要单例模式/好处是什么

1. 全局状态管理：确保关键组件在整个应用中只有一个实例，维护状态一致性。  
   比如说逻辑处理层，：用来管理消息队列，回调函数的映射这些全局的状态，保证这些状态在整个应用程序是一致的。
2. 资源共享和优化：避免重复创建资源密集型对象，提高资源利用率。  
   比如说对于基于asio的io服务池，他管理了很多用于处理异步io的对象，那么所有进行异步io操作的组件可以共享同一个io服务池。
3. 简化访问：提供全局访问点，configmgr配置管理类：允许在应用程序任何地方方便的访问配置信息，
4. 生命周期管理：确保关键组件在整个应用生命周期内可用，并程序结束的时候进行正确清理。
5. 可以通过线程安全的方式，确保只有一个实例被创建和访问，避免多线程竞争导致数据不一致。
6. 如果用懒汉式可以采用延迟加载，提高系统启动速度，在需要时才分配资源，。

### 12.为什么要用nodejs，原理是什么

 非阻塞 I/O：Node.js 使用单线程，事件驱动、非阻塞 I/O 模型，使其轻量且高效。

b. 统一的语言：前后端都可以使用 JavaScript，降低了开发难度。

c. 丰富的生态系统：npm（Node Package Manager）提供了大量的第三方库。

d. 高性能：对于 I/O 密集型应用，Node.js 表现出色。

e. 适合微服务：轻量级特性使其适合构建微服务架构。

垃圾回收机制Defer类  
 对象生命周期控制：当创建一个 Defer 对象时，将一个函数（通过 std::function<void()> 来表示）传递给 Defer 的构造函数。这个函数会在 Defer 对象被销毁时执行。

 析构函数的作用：Defer 类的析构函数 ~Defer() 在对象销毁时自动调用。在析构函数中，会执行之前传入的函数 func\_()，这样就能实现延迟执行的效果。

### 13.槽函数的原理/信号与槽的关系

当信号（Signal）发出时，绑定到该信号的槽函数会被调用

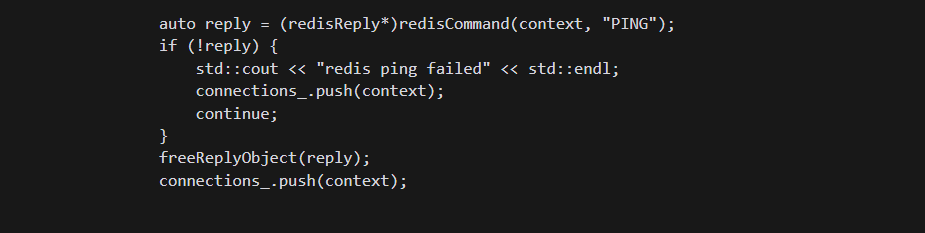
信号与槽通过 connect() 函数关联在一起。当信号被发出时，QT框架会自动调用与该信号关联的槽函数。

基于源对象系统工作的

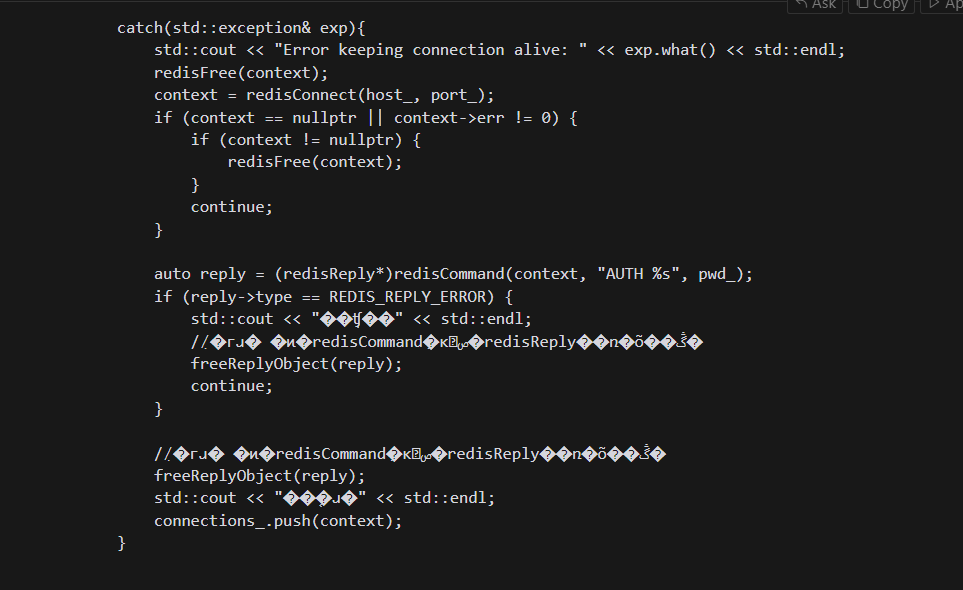
### 14.如果某个实例掉线了，怎么处理的呢

Redis 的 Hash 结构来存储每个服务器实例的状态和登录数

基于 Redis 的心跳机制。每个实例定期向 Redis 发送 PING 命令，更新自己的状态。



我们使用连接池来管理与 Redis 的连接，并实现了自动重连逻辑。如果检测到连接断开，系统会自动尝试重新建立连接。

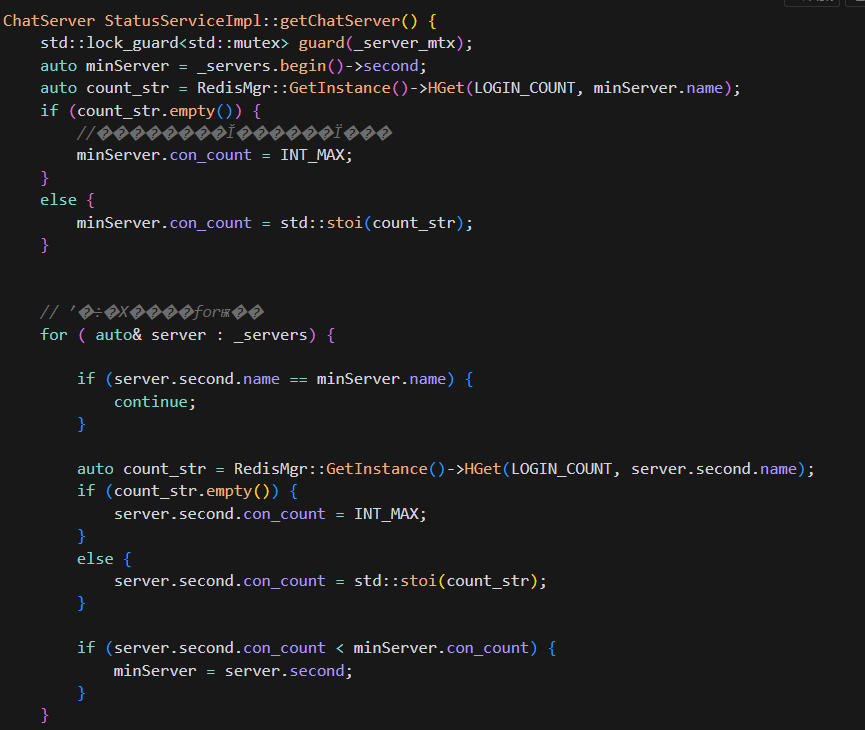


当服务器正常关闭时，我们会从 Redis 中删除该实例的状态信息，确保其他部分不会误认为这是一次意外掉线。

12.前端访问怎么拿到后端部署的实例？//前端怎么连接到后端

 1.负载均衡：您使用了一个StatusServer来实现简单的负载均衡。当客户端需要连接到ChatServer时，它首先会请求StatusServer来获取一个可用的ChatServer实例。

 2.实例选择：StatusServer通过Redis中存储的登录数来选择负载最小的ChatServer实例。



3.返回实例信息：StatusServer选择好实例后，会将该实例的信息（如IP地址和端口）返回给客户端。

4.客户端连接：客户端（前端）收到ChatServer的信息后，直接与该ChatServer建立连接。

5. 连接管理：一旦连接建立，ChatServer会更新Redis中的登录计数：

6.断线重连：如果连接断开，客户端可以重新请求StatusServer来获取一个可用的ChatServer实例。

7. 心跳机制：为了保持连接的活跃状态和检测实例是否在线，您实现了一个基于Redis的心跳机制：

#### Tcp与http的区别与联系

**TCP 粘包与拆包**： 假设我们有两个数据包 A 和 B，发送顺序是 A -> B。如果 TCP 发送数据包时没有明确的边界标识，接收方可能会收到 A 和 B 粘在一起的数据包 A+B。在这种情况下，接收方需要解析数据包以识别数据包的分界。

**HTTP 回包处理**： 当你发送一个 HTTP 请求到服务器时，服务器会返回一个 HTTP 响应。响应中包括了 Content-Length 头部，它指示了消息体的长度。即使 TCP 中发生粘包或拆包，HTTP 协议仍然能够通过 Content-Length 或 chunked 传输编码来解析和提取完整的 HTTP 响应。

### 15.服务器和用户怎么通讯的/客户端和聊天服务怎么进行直接交互的。

需要仔细看代码

 1.连接建立：

客户端通过TCP连接到服务器。服务器使用Boost.Asio库监听连接请求，并为每个新连接创建一个Csession会话对象。

 2.消息接收：

Csession会话对象负责异步读取客户端发送的消息。它首先读取消息头，然后根据头部信息读取消息体。

 3.消息处理：

接收到的消息被封装成LogicNode对象，然后通过LogicSystem的PostMsgToQue方法加入处理队列。

 4.业务逻辑：

LogicSystem在单独的线程中处理消息队列。它根据消息ID调用相应的处理函数（如LoginHandler）。

 5.消息发送：

处理完毕后，服务器通过CSession的Send方法异步发送响应给客户端。还要看具体服务器的地址，看看需不需要调用grpc

6.断线重连机制

如果连接中断，按再次请求状态服务获取可用的聊天服务，重新建立连接。

GET和POST的区别？

1. 数据传输：GET 将参数附加在 URL 后，POST 将数据放在请求体中。
2. 数据量：GET 受 URL 长度限制，通常用于小量数据；POST 可以传输大量数据。

3. 安全性：GET 参数在 URL 中可见，不适合敏感信息；POST 相对更安全。

4. 用途：GET 主要用于获取资源，POST 用于提交数据。

5.缓存：GET 请求可以被缓存，POST 通常不缓存。

6. 幂等性：GET 是幂等的，多次请求结果相同；POST 可能导致多次操作。

你知道什么是断线重连吗？

 1.连接状态：检测客户端需要定期检查与服务器的连接状态。这可以通过以下方式实现：

 ①心跳机制：定期发送小型数据包，如果在一定时间内没有收到响应，则认为连接已断开。

 ②监听网络事件：使用操作系统提供的网络事件通知机制。

 2.重连逻辑：

当检测到连接断开时，客户端应该启动重连过程：

 设置重连间隔：通常采用递增的重连间隔，例如首次尝试立即重连，然后逐渐增加等待时间。

 最大重试次数：设置一个最大重试次数，避免无限重连。

3. 会话恢复：

重连成功后，可能需要恢复之前的会话状态：

 客户端保存重要的会话信息。

 服务器端的数据库中存储用户状态，可以基于用户ID或会话ID恢复状态。

 4.错误处理：

在重连过程中要妥善处理各种错误情况，如网络不可用、服务器无响应等。

 5. 用户界面反馈：

在重连过程中，应该向用户提供适当的反馈，如显示"正在重新连接"的提示。

### 16.两个client是怎么互相发消息的？

### 17.前端的的http服务和tcp服务都是用来干啥的

http：就是用户注册、登录、获取验证码的时候，（这些一次性的工作）

void HttpMgr::PostHttpReq  
设置URL、请求头和数据，发送post请求，收到返回的响应以后，触发回调函数