

Proto文件中若含有 service, 那么如果 不想使用 grpc连接器链接编译则: proto文件中需要把生成服务的Option打开: `option cc_generic_services = true; // 生成服务的Option要打开`

```
service UserServiceRpc { rpc Login(LoginRequest) returns(LoginResponse); rpc Register(RegisterRequest)
returns(RegisterResponse); }
```

服务生成后: UserServiceRpc\_Stub类 给调用方使用 UserServiceRpc类 给服务器 提供方使用

Caller:

```
#include<iostream>
#include"UserService.pb.h"
#include<mprpcchannel.h> // 使用 mprpc的管道设计

int main(int argc,char* argv[])
{
    std::string ip = "127.0.0.1";
    short port = 8888;

    auto stub = fixbug::UserServiceRpc_Stub(new MprpcChannel(ip, port, false));

    fixbug::LoginRequest request;
    request.set_name("zhang san");
    request.set_pwd("123456");

    fixbug::LoginResponse response;

    // 这里直接调用我想要调用的函数即可 就和调用自己本身的函数方法一模一样
    stub.Login(nullptr, &request, &response, nullptr);

    if(0==response.result().errcode()){
        std::cout << "rpc login response success:" << response.success() << std::endl;
    }else{
        std::cout << "rpc login response error:" << response.result().errmsg() << std::endl;
    }

    return 0;
}
```

Stub需要使用管道 (内部的传输层设置 1、自定义协议 仿照mprpcchannel去写, 自定义协议 2、使用 mRpc框架里的 channel设置) stub.Login 表明要调用的: `UserServiceRpc`服务中的 Login方法 (stub会和服务进行绑定, 然后stub内部可以显示, 此服务中可调用的方法)调用方法时, 第一个参数可以传入 controller, 然后 RPC的调用结果, 可以通过 controller中的状态看出来

Callee:

第一步：重写 UserServiceRpc 类中对应的服务函数

```
#include<iostream>
#include"UserService.pb.h"
#include<mprpcchannel.h>
#include"rpcprovider.h"
#include<string>

class Callee : public fixbug::UserServiceRpc{
public:
    // 提供方自身的 业务处理函数
    bool Login(std::string name, std::string pwd){
        std::cout<<"doint local service :Login"<<std::endl;
        std::cout << "name:" << name << "pwd:" << pwd << std::endl;
        return false;
    }
    // RPC 通信：接收请求，处理请求，调用业务处理函数；处理结果，设置结果，返回。
    void Login(::google::protobuf::RpcController* controller,
              const ::fixbug::LoginRequest* request,
              ::fixbug::LoginResponse* response,
              ::google::protobuf::Closure* done){
        std::string name = request->name();
        std::string pwd = request->pwd();

        // 调用业务处理函数
        bool login_result = Login(name, pwd);

        fixbug::ResultCode* code = response->mutable_result();
        code->set_errcode(0);

        code->set_errmsg("");

        response->set_success(login_result);

        done->Run();
    }
};

int main(int argc, char* argv[]){
    std::string ip="127.0.0.1";
    short port = 8888;

    auto stub = fixbug::UserServiceRpc_Stub(new MprpcChannel(ip, port, false));

    RpcProvider provide;

    provide.NotifyService(new Callee());

    provide.Run(1, 8888);

    return 0;
}
```

第二步：创建提供对象 RpcProvider NotifyService 表明自己提供的服务是哪个 provide.Run(1, 8888) 是表明，该节点的 Index 是 1，port 端口是 8888，内部绑定的 ip，是在 Run 函数内部 通过函数自动获取本地 ip 后进行绑定的

在 `Callee.cpp` 文件中，服务端的 IP 和端口绑定是通过 `RpcProvider::Run` 方法实现的。

`RpcProvider` 内部的 `Run` 方法处理了网络服务器的创建和绑定。具体来说，这些步骤是在 `Run` 方法中完成的：

1. **获取本机 IP**：通过 `gethostname` 和 `gethostbyname` 函数获取本机的 IP 地址。
2. **绑定 IP 和端口**：创建一个 `muduo::net::InetAddress` 对象，并使用它来创建 `muduo::net::TcpServer` 对象。`TcpServer` 对象绑定到指定的 IP 和端口。
3. **启动服务器**：调用 `TcpServer` 对象的 `start` 方法启动网络服务，并进入事件循环。

```
void RpcProvider::Run(int nodeIndex, short port) {
    // 获取本机 IP 地址
    char *ipC;
    char hname[128];
    struct hostent *hent;
    gethostname(hname, sizeof(hname));
    hent = gethostbyname(hname);
    for (int i = 0; hent->h_addr_list[i]; i++) {
        ipC = inet_ntoa(*(struct in_addr *)(hent->h_addr_list[i])); // IP 地址
    }
    std::string ip = std::string(ipC);

    // 写入配置文件
    std::string node = "node" + std::to_string(nodeIndex);
    std::ofstream outfile;
    outfile.open("test.conf", std::ios::app); // 打开文件并追加写入
    if (!outfile.is_open()) {
        std::cout << "打开文件失败！" << std::endl;
        exit(EXIT_FAILURE);
    }
    outfile << node + "ip=" + ip << std::endl;
    outfile << node + "port=" + std::to_string(port) << std::endl;
    outfile.close();
}
```

```
// 设置回调
m_muduo_server->setConnectionCallback(std::bind(&RpcProvider::OnConnection, this, std::placeholders::_1, std::placeholders::_2));
m_muduo_server->setMessageCallback(std::bind(&RpcProvider::OnMessage, this, std::placeholders::_1, std::placeholders::_2));

// 启动服务器
m_muduo_server->start();
m_eventLoop->loop();
}
```

那么 前面定义 `std::string ip = "127.0.0.1";short port = 7788;` 有啥哟？ 并且 `Run`第一个参数是2的话会怎么样？ `Run`内部的ip和端口信息从何而来，为什么需要写入配置文件？ 配置文件不是在外面提前写好的吗？ 配置文件里面的信息是啥？

---

## 1. `std::string ip = "127.0.0.1"; short port = 7788;` 的作用

在服务端的代码中，定义 IP 和端口（如 `std::string ip = "127.0.0.1"; short port = 7788;`）是为了初始化 `RpcProvider` 对象的服务发布设置。这些变量用于：

- **IP 地址**：指定服务监听的网络接口地址。如果你使用 `127.0.0.1`，服务将只接受来自本机的请求。
- **端口**：指定服务监听的端口号。端口号是通信的关键部分，确保客户端和服务端能够正确地建立连接。

## 2. `Run` 方法中的第一个参数（`nodeIndex`）

`Run` 方法的第一个参数（`nodeIndex`）通常用于标识不同的服务节点。在多节点分布式系统中，每个节点可能会有不同的配置。`nodeIndex` 可以用来：

- **标识不同的节点**：使得每个节点有不同的标识符。
- **生成唯一的配置文件**：确保每个节点的信息在配置文件中是唯一的。

在实际使用中，如果 `Run` 方法中的 `nodeIndex` 设置为 `2`，这意味着：

- **配置文件**：`test.conf` 文件中会包含 `node2ip` 和 `node2port` 的条目。
- **服务标识**：该服务将会有 `node2` 的标识符，在配置文件中记录服务的 IP 和端口信息时会有所区别。

## 3. IP 和端口信息从何而来

在 `Run` 方法中，IP 和端口信息通过以下步骤获取和设置：

- **IP 地址**：通过 `gethostname` 和 `gethostbyname` 获取本机的 IP 地址。这是服务监听的地址。
- **端口号**：作为参数传递给 `Run` 方法的端口号，指定服务监听的端口。

## 4. 写入配置文件的目的是

写入配置文件的目的是：

- **服务发现**：配置文件可以帮助其他节点或系统找到和连接到这个服务节点。配置文件通常包含节点的 IP 和端口信息。
- **管理和监控**：在复杂的系统中，配置文件可以用于集中管理和监控服务节点的信息。

配置文件的内容：

在你提供的代码中，`test.conf` 文件会包含如下内容：

```
makefile
```

[复制代码](#)

```
node1ip=127.0.0.1
node1port=7788
```

或者当 `nodeIndex` 是 2 时:

```
makefile
```

[复制代码](#)

```
node2ip=127.0.0.1
node2port=7788
```

这些条目表示:

- `nodeXip`: 服务节点的 IP 地址。
- `nodeXport`: 服务节点的端口号。

Server 的服务类的重写, 基本都包括: RPC通信+实际的本地业务处理逻辑函数

```
class FriendService : public fixbug::FiendServiceRpc {
public:
    std::vector<std::string> GetFriendsList(uint32_t userid) {
        std::cout << "local do GetFriendsList service! userid:" << userid << std::endl;
        std::vector<std::string> vec;
        vec.push_back("gao yang");
        vec.push_back("liu hong");
        vec.push_back("wang shuo");
        return vec;
    }

    // 重写基类方法
    void GetFriendsList(::google::protobuf::RpcController *controller, const ::fixbug::GetFriendsListRequest *request,
        ::fixbug::GetFriendsListResponse *response, ::google::protobuf::Closure *done) {
        uint32_t userid = request->userid();
        std::vector<std::string> friendsList = GetFriendsList(userid);
        response->mutable_result()->set_errcode(0);
        response->mutable_result()->set_errmsg("");
        for (std::string &name : friendsList) {
            std::string *p = response->add_friends();
            *p = name; //先开辟数组中 地址空间, 然后向空间中填充 GetFriendList的响应结果 friendList: string
        }
        done->Run();
    }
};
```

服务器用到了 `rpcProvider` 类, 里面用到了 `muduo` 库进行 TCP 链接设置, 通信设置, 数据的格式, 序列化格式等。

是的, 按照你提供的 `RpcProvider` 类的实现, `Callee` 确实使用了自定义的通信协议, 主要通过 `muduo` 库来进行 TCP 连接和事件循环的管理。而 `Caller` 如果使用的是默认的 RPC 机制, 那么它可能依赖的是常见的 RPC 框架自带的传输层协议, 如 HTTP/2 (gRPC 的默认传输层协议) 或其他标准的传输协议。

因此, `Callee` 使用了定制的通信协议, 而 `Caller` 使用默认的传输层协议, 这两者的差异可能体现在消息格式、序列化方式和连接管理等方面。

muduo监听是异步的：

是的，Callee 利用 `muduo` 库实现的监听是异步的。

`muduo` 库本身是一个高性能的异步网络库，采用了 Reactor 模型进行事件驱动。通过 `EventLoop` 和 `TcpServer` 等核心组件，`muduo` 可以在单线程或多线程环境下异步地处理网络 I/O 事件。

在 `RpcProvider` 中，启动的 `TcpServer` 会通过 `EventLoop` 循环监听网络事件，而不是阻塞地等待连接或消息。这意味着 Callee 的网络通信和处理是异步的，能够在多个客户端同时发起 RPC 请求时高效地处理每个请求，不会因为某一个请求而阻塞整个服务。

cpp

复制代码

```
m_muduo_server->start();  
m_eventLoop.loop(); // 进入事件循环，异步处理连接和消息
```

这一部分代码启动了 `TcpServer`，并且 `EventLoop` 的循环负责处理所有的网络事件和回调，包括连接建立、数据到达等。因此，Callee 的监听和通信是异步进行的。



## 异步事件驱动模型的工作机制

`muduo` 基于 Reactor 模型，这意味着服务器通过 `EventLoop`（事件循环）不断地监听所有网络事件（如连接、读写等）。当有事件（如新的连接、数据到达等）发生时，事件循环会触发对应的回调函数来处理。

以下是多个客户端（Clerk）同时发起 RPC 请求时的处理流程：

### 1. 多个 Clerk 发起请求：

- 当多个客户端（Clerk）同时发起 RPC 请求时，服务器并不会依次阻塞等待每个请求处理完毕。每当有客户端发起请求时，`muduo` 库的 `EventLoop` 会检测到新的网络事件，并将该请求的处理委派给回调函数 `OnMessage`。

### 2. 非阻塞的处理：

- `OnMessage` 回调函数负责解析 RPC 请求的数据头和内容，然后根据请求的服务名称、方法名称及参数，调用本地的业务逻辑方法（例如 `service->CallMethod()`）。这些操作在事件循环中是非阻塞的，这意味着服务器不会因为处理某个 Clerk 的 RPC 请求而阻塞整个系统。

### 3. 请求处理中的阻塞情况：

- 如果某个 RPC 请求的业务逻辑处理本身是耗时的（例如需要访问磁盘或数据库），而又没有使用异步的业务处理方式，确实可能会阻塞该线程。但是，由于 `muduo` 库支持多线程，通常会配置多个工作线程来处理不同的请求（`m_muduo_server->setThreadNum(4);`），所以即使一个线程在处理某个请求时阻塞了，其他线程仍然可以继续处理其他客户端的请求。

### 4. 异步处理的优势：

- 由于网络事件和回调都是由 `EventLoop` 异步驱动的，新的 RPC 请求会立即被分发到空闲的线程进行处理，而不会因为一个请求的处理时间过长而影响其他请求的响应速度。

## 例子

假设有 3 个 Clerk 同时发起 RPC 请求，以下是处理过程：

### 1. Clerk 1 发起请求：

- `EventLoop` 检测到请求，并调用 `OnMessage` 处理 Clerk 1 的请求。假设这个请求需要进行复杂的计算或磁盘操作，那么该线程可能会暂时阻塞。

### 2. Clerk 2 发起请求：

- `EventLoop` 检测到 Clerk 2 的请求，由于 `muduo` 使用多线程模型，`EventLoop` 会将这个请求分发到其他空闲线程处理，而不会等待 Clerk 1 的请求完成。

### 3. Clerk 3 发起请求：

- 同样，`EventLoop` 检测到 Clerk 3 的请求，会再次分发到另一个空闲线程处理。

## 总结

由于 `muduo` 使用异步非阻塞的方式处理网络事件，并且支持多线程，因此在一个 Clerk 的请求处理时，即使该请求阻塞了，其他客户端的请求仍然可以被服务器高效处理，不会因为某一个请求的阻塞



而影响整个系统的响应能力。这就是异步处理的优势所在。

Server服务结束后的 回调函数done:

```
// 给下面的method方法的调用, 绑定一个Closure的回调函数
// closure是执行完本地方法之后会发生的回调, 因此需要完成序列化
// 化和反向发送请求的操作
google::protobuf::Closure *done =
    google::protobuf::NewCallback<RpcProvider, const
muduo::net::TcpConnectionPtr &, google::protobuf::Message *>(
    this, &RpcProvider::SendRpcResponse, conn, response);

// 在框架上根据远端rpc请求, 调用当前rpc节点上发布的方法
// new UserService().Login(controller, request, response, done)
```

该回调函数和 Provider中的回调函数:

这里说的回调函数和服务提供的服务中的 回调 Closure有什么区别或联系吗? 该Closure回调函数的作用是把调用结果传给调用方Caller的吗, 如果是的话服务中的 Closure回调函数是需要我们自定义的吗? 那我们自定义的时候, 又是通过什么操作使得它能将结果传递给调用方Caller的呢?

```
void GetFriendsList(::google::protobuf::RpcController *controller,
const ::fixbug::GetFriendsListRequest *request,
                ::fixbug::GetFriendsListResponse *response,
::google::protobuf::Closure *done) {
    uint32_t userid = request->userid();
    std::vector<std::string> friendsList = GetFriendsList(userid);
    response->mutable_result()->set_errcode(0);
    response->mutable_result()->set_errmsg("");
    for (std::string &name : friendsList) {
        std::string *p = response->add_friends();
        *p = name;    //先开辟数组中 地址空间, 然后向空间中填充
        GetFriendList的响应结果 friendList: string
    }
    done->Run();
}
```

在你的 `GetFriendsList` 函数中，`Closure` 回调函数的确是用来将服务端的调用结果传递给客户端 (Caller) 的。`Closure` 是由 Protocol Buffers 提供的回调机制，确保当服务端完成对请求的处理后，可以通过回调函数来处理后续的操作，例如返回结果。

### 回调函数与 `Closure` 的区别和联系：

- **普通回调函数**：服务器使用的异步网络库（如 Muduo）中常用的回调函数，通常用来处理网络事件，如新连接到达、数据到达等。这些回调函数会在特定的事件发生时被触发，并由事件循环调用，主要用于处理网络和 IO 事件。
- **`Closure` 回调函数**：这是专门为 RPC 处理设计的回调机制。`Closure` 是 Protocol Buffers 框架的一部分，它的主要目的是在服务端完成方法调用后执行一些后续操作（如将处理结果返回给客户端）。

### `Closure` 回调函数的作用：

- 在服务端调用结束后，`Closure::Run()` 被调用。此时，`done->Run();` 会触发回调函数的执行，将 `response` 对象的序列化数据通过网络发送回调方。
- `Closure` 回调函数的作用是让开发者可以指定 RPC 调用完成后的处理方式，通常用于发送响应结果。

### 自定义 `Closure` 的操作：

在 `GetFriendsList` 方法中，`done->Run()` 的目的是通知框架这次 RPC 调用已经处理完毕。你不需要手动定义 `Closure` 的具体行为，因为在 RPC 框架内，`Closure` 的执行会自动将 `response` 序列化，并通过底层网络将结果传递给客户端。你只需要在适当的时机调用 `done->Run()`。

通过这种机制，当服务端处理完请求后，结果会被自动封装和发送到客户端。

### 自定义 `Closure`

我们可以自己绑定一个 `Closure` 对象吗，执行自定义的服务结束函数



是的，你可以绑定和执行自定义的 `Closure` 对象来处理服务结束后的逻辑。这允许你在 RPC 调用结束时执行一些自定义操作，比如记录日志、清理资源、甚至执行异步操作。你可以通过继承 `google::protobuf::Closure` 类来创建自定义的回调函数，然后将其绑定到你的服务中。

#### 自定义 `Closure` 的实现步骤：

1. **继承 `Closure` 类**：创建一个类，继承自 `google::protobuf::Closure`，并实现 `Run()` 方法。在这个方法里，你可以定义服务调用完成后的自定义逻辑。
2. **将自定义 `Closure` 绑定到 RPC 调用中**：在服务中，你可以在处理完请求后，使用你自定义的 `Closure`，并调用其 `Run()` 方法。

```

#include <google/protobuf/stubs/common.h>
#include <iostream>

class CustomClosure : public google::protobuf::Closure {
public:
    CustomClosure(std::string task) : task_(task) {}

    // 重写 Run 方法, 定义自定义行为
    void Run() override {
        // 自定义的服务结束后的逻辑
        std::cout << "Task finished: " << task_ << std::endl;
        // 这里可以加入更多自定义逻辑, 比如记录日志, 执行其他异步操作等
    }

private:
    std::string task_;
};

// 定义服务中的方法
void GetFriendsList(::google::protobuf::RpcController *controller, const
::fixbug::GetFriendsListRequest *request,
                    ::fixbug::GetFriendsListResponse *response,
                    ::google::protobuf::Closure *done) {
    uint32_t userid = request->userid();
    std::vector<std::string> friendsList = GetFriendsList(userid);
    response->mutable_result()->set_errcode(0);
    response->mutable_result()->set_errmsg("");
    for (std::string &name : friendsList) {
        std::string *p = response->add_friends();
        *p = name;
    }

    // 调用自定义的 Closure
    CustomClosure customDone("Sending response to client");
    customDone.Run(); // 自定义行为: 输出 Task finished: Sending response to
client

    done->Run(); // 继续 RPC 框架的默认行为, 将响应发送回客户端
}

```

### 解释:

- `CustomClosure` 类继承自 `google::protobuf::Closure`，并实现了 `Run()` 方法。在 `Run()` 中，可以自定义任何你想在服务结束时执行的操作。
- 在 `GetFriendsList` 方法中，我们实例化了一个 `CustomClosure` 对象，并在处理完成后调用 `customDone.Run()` 来执行自定义的逻辑。
- 最后，`done->Run()` 依然被调用，这个是 RPC 框架默认的行为，它负责将处理结果发送回客户端。

通过这种方式，你可以在 `Closure` 回调机制中引入自己的自定义行为，来扩展服务的功能。