C++简单实现RPC网络通讯的示例详解

C/C++ 108 0 0 2023-05-17

登录后可点赞和收藏
c++中的stack和dequeue解析
上一篇
C++多线程传参的实现方法
下一篇

目录

- 3.2 加入网络连接（socket）
- 3.3 加强并发性
- 3.4 加入容错机制（修改客户端部分）

RPC是远程调用系统简称，它允许程序调用运行在另一台计算机上的过程，就像调用本地的过程一样。RPC 实现了网络编程的"过程调用"模型，让程序员可以像调用本地函数一样调用远程函数。最近在做的也是远程调用过程，所以通过重新梳理RPC来整理总结一下。

项目来源：

GitHub - qicosmos/rest_rpc: modern C++(C++11), simple, easy to use rpc framework

一、RPC简介

1.1 简介

RPC指的是计算机A的进程调用另外一台计算机B的进程，A上的进程被挂起，B上被调用的进程开始执行，当B执行完毕后将执行结果返回给A，A的进程继续执行。调用方可以通过使用参数将信息传送给被调用方，然后通过传回的结果得到信息。这些传递的信息都是被加密过或者其他方式处理。这个过程对开发人员是透明的，因此RPC可以看作是本地过程调用的一种扩展，使被调用过程不必与调用过程位于同一物理机中。

RPC可以用于构建基于B/S模式的分布式应用程序：请求服务是一个客户端、而服务提供程序是一台服务器。和常规和本地的调用过程一样，远程过程调用是同步操作，在结果返回之前，需要暂时中止请求程序。

RPC的优点：

- 支持面向过程和面向线程的模型；

- 内部消息传递机制对用户隐藏；
- 基于 RPC 模式的开发可以减少代码重写；
- 可以在本地环境和分布式环境中运行；

1.2 本地调用和远程调用的区别

以ARM环境为例，我们拆解本地调用的过程，以下面代码为例：

```
int selfIncrement(int a)
{
    return a + 1;
}
int a = 10;
```

当执行到selfIncrement(a)时，首先把a存入寄存器R0，之后转到函数地址selfIncrement，执行函数内的指令 ADD R0,#1。跳转到函数的地址偏移量在编译时确定。

但是如果这是一个远程调用，selfIncrement函数存在于其他机器，为了实现远程调用，请求方和服务方需要提供需要解决以下问题：

1. 网络传输。

本地调用的参数存放在寄存器或栈中，在同一块内存中，可以直接访问到。远程过程调用需要借助网络来传递参数和需要调用的函数 ID。

2. 编解码

请求方需要将参数转化为字节流，服务提供方需要将字节流转化为参数。

3. 函数映射表

服务提供方的函数需要有唯一的 ID 标识，请求方通过 ID 标识告知服务提供方需要调用哪个函数。

以上三个功能即为 RPC 的基本框架所必须包含的功能。

1.3 RPC运行的流程

一次 RPC 调用的运行流程大致分为如下七步，具体如下图所示。

1.客户端调用客户端存根程序，将参数传入；

2.客户端存根程序将参数转化为标准格式，并编组进消息；

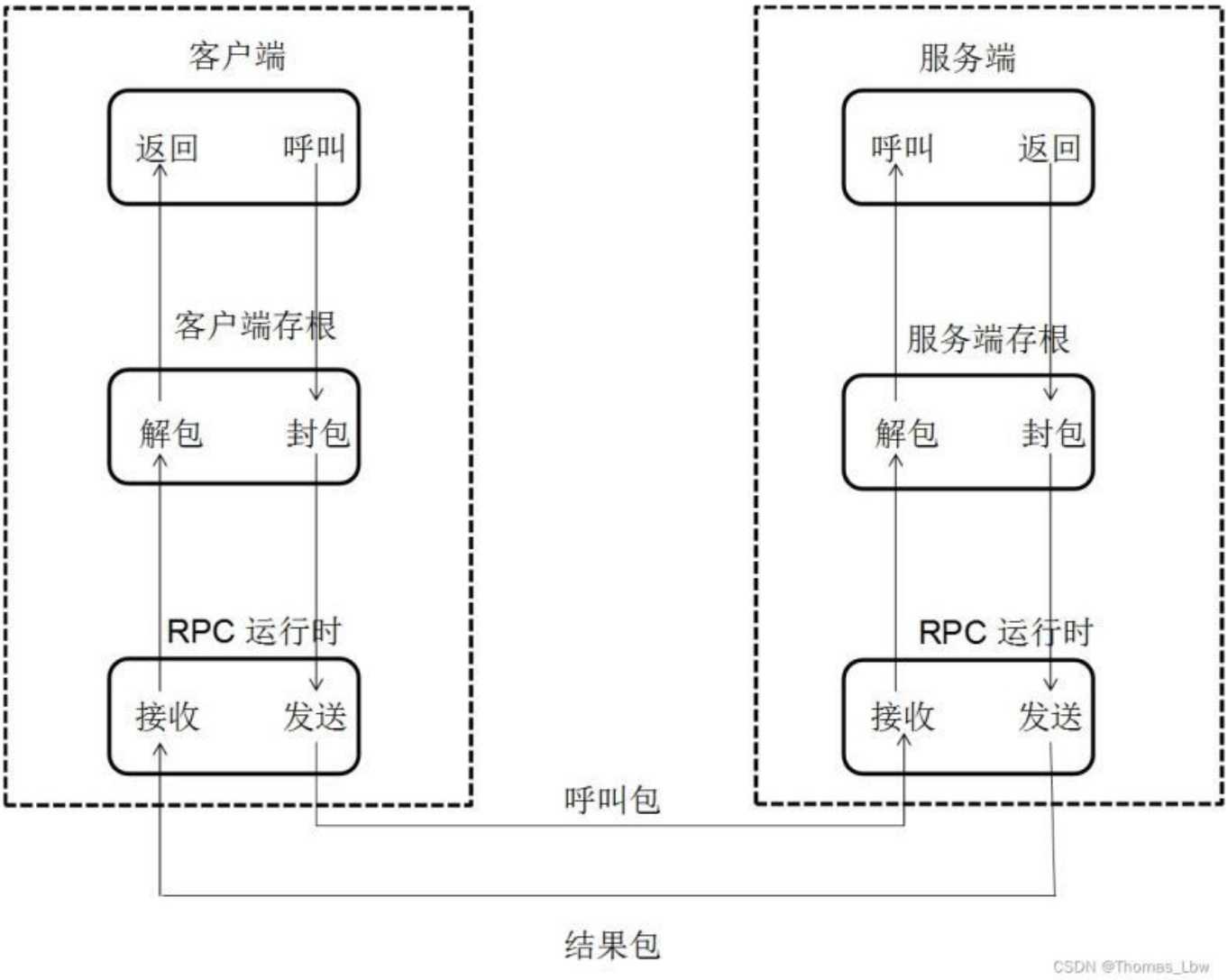
3.客户端存根程序将消息发送到传输层，传输层将消息传送至远程服务器；

4.服务器的传输层将消息传递到服务器存根程序，存根程序对阐述进行解包，并使用本地调用的机制调用所需的函数；

5.运算完成之后，将结果返回给服务器存根，存根将结果编组为消息，之后发送给传输层;

6.服务器传输层将结果消息发送给客户端传输层；

7.客户端存根对返回消息解包，并返回给调用方。



服务端存根和客户端存根可以看做是被封装起来的细节，这些细节对于开发人员来说是透明的，但是在客户端层面看到的是 “本地” 调用了 selfIncrement() 方法，在服务端层面，则需要封装、网络传输、解封装等等操作。因此 RPC 可以看作是传统本地过程调用的一种扩展，其使得被调用过程不必与调用过程位于同一物理机中。

1.4 小结

RPC 的目标是做到在远程机器上调用函数与本地调用函数一样的体验。 为了达到这个目的，需要实现网络传输、序列化与反序列化、函数映射表等功能，其中网络传输可以使用socket或其他，序列化和反序列化可以使用protobuf，函数映射表可以使用std::function。

lambda与std::function内容可以看：

C++11 匿名函数lambda的使用

C++11 std::function 基础用法

lambda 表达式和 std::function 的功能是类似的，lambda 表达式可以转换为 std::function，一般情况下，更多使用 lambda 表达式，只有在需要回调函数的情况下才会使用 std::function。

二、RPC简单实现

2.1 客户端实现代码

复制

```
#include <iostream>
#include <memory>
#include <thread>
#include <functional>
#include <cstring>

class RPCClient
{
public:
    using RPCCallback = std::function<void(const std::string&)>;
    RPCClient(const std::string& server_address) : server_address_(server_address)
    ~RPCClient() {}

    void Call(const std::string& method, const std::string& request, RPCCallback c
    {
        // 序列化请求数据
        std::string data = Serialize(method, request);
        // 发送请求
        SendRequest(data);
        // 开启线程接收响应
        std::thread t([this, callback]() {
            std::string response = RecvResponse();
            // 反序列化响应数据
            std::string result = Deserialize(response);
            callback(result);
        });
        t.detach();
    }

private:
    std::string Serialize(const std::string& method, const std::string& request)
    {
        // 省略序列化实现
    }

    void SendRequest(const std::string& data)
    {
        // 省略网络发送实现
    }

    std::string RecvResponse()
```

```
    {
        // 省略网络接收实现
    }

    std::string Deserialize(const std::string& response)
    {
        // 省略反序列化实现
    }

private:
    std::string server_address_;
};

int main()
{
    std::shared_ptr<RPCClient> client(new RPCClient("127.0.0.1:8000"));
    client->Call("Add", "1,2", [](const std::string& result) {
        std::cout << "Result: " << result << std::endl;
    });
    return 0;
}
```

这段代码定义了RPCClient类来处理客户端的请求任务，用到了lambda和std::function来处理函数调用，在Call中使用多线程技术。main中使用智能指针管理Rpcclient类，并调用了客户端的Add函数。

127.0.0.1为本地地址，对开发来说需要使用本地地址自测，端口号为8000，需要选择一个空闲端口来通信。

2.2 服务端代码

下面是服务端的实现

```
#include <iostream>
#include <map>
#include <functional>
#include <memory>
#include <thread>
#include <mutex>
// 使用第三方库实现序列化和反序列化
#include <boost/serialization/serialization.hpp>
#include <boost/serialization/map.hpp>
using namespace std;
// 定义RPC函数类型
using RPCCallback = std::function<std::string(const std::string&)>;
class RPCHandler {
public:
```

```
    void registerCallback(const std::string& name, RPCCallback callback) {
        std::unique_lock<std::mutex> lock(mtx_);
        callbacks_[name] = callback;
    }
    std::string handleRequest(const std::string& request) {
        // 反序列化请求
        std::map<std::string, std::string> requestMap;
        std::istringstream is(request);
        boost::archive::text_iarchive ia(is);
        ia >> requestMap;
        // 查找并调用对应的回调函数
        std::string name = requestMap["name"];
        std::string args = requestMap["args"];
        std::unique_lock<std::mutex> lock(mtx_);
        auto it = callbacks_.find(name);
        if (it == callbacks_.end()) {
            return "Error: Unknown function";
        }
        RPCCallback callback = it->second;
        return callback(args);
    }
private:
    std::map<std::string, RPCCallback> callbacks_;
    std::mutex mtx_;
};
int main() {
    RPCHandler rpcHandler;
    // 注册回调函数
    rpcHandler.registerCallback("add", [](const std::string& args) {
        std::istringstream is(args);
        int a, b;
        is >> a >> b;
        int result = a + b;
        std::ostringstream os;
        os << result;
        return os.str();
    });
    rpcHandler.registerCallback("sub", [](const std::string& args) {
        std::istringstream is(args);
        int a, b;
        is >> a >> b;
        int result = a - b;
        std::ostringstream os;
        os << result;
        return os.str
    });
    // 创建处理请求的线程
    std::thread requestThread([&]() {
        while (true) {
```

```
            std::string request;
            std::cin >> request;
            std::string response = rpcHandler.handleRequest(request);
            std::cout << response << std::endl;
        }
    });
    requestThread.join();
    return 0;
}
```

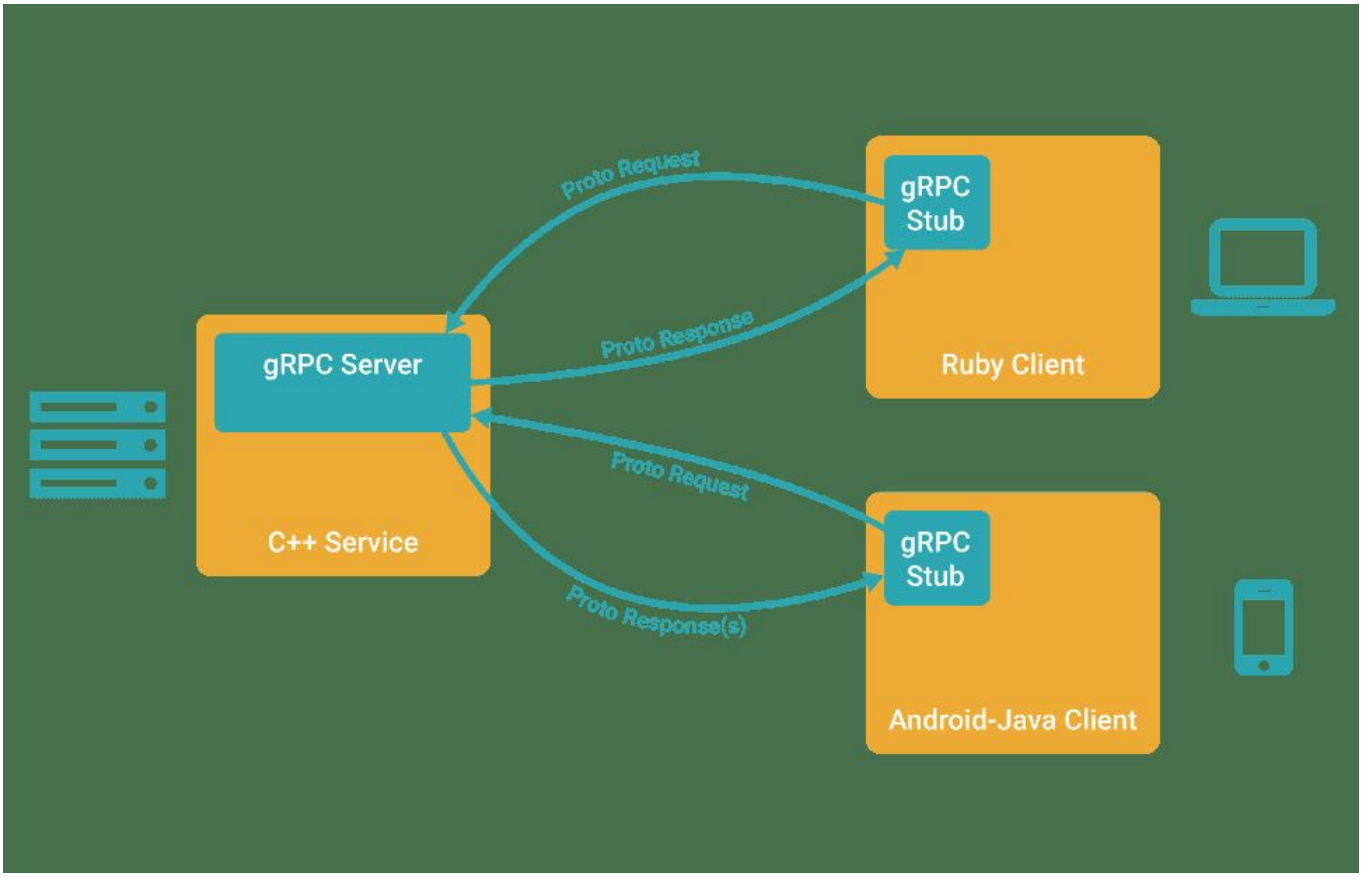
上面的代码实现了一个简单的C++ RPC服务端。主要实现了以下功能：

1.定义了RPC函数类型 RPCCallback，使用std::function<std::string(const std::string&)>表示。

2.RPCHandler类实现了注册函数和处理请求的功能。

3.在main函数中创建了一个RPCHandler对象，并注册了两个函数"add" 和 "sub"。这些函数通过lambda表达式实现，并在被调用时通过std::istringstream读取参数并返回结果。

4.创建了一个新线程requestThread来处理请求。在这个线程中，通过std::cin读取请求，然后调用RPCHandler的handleRequest函数并使用std::cout输出响应。



注意，这套代码是最简单的RPC机制，只能调用本地的资源，他还存在以下缺点：

1.代码并没有处理错误处理，如果请求格式不正确或函数不存在，服务端将会返回“Error: Unknown function”。

2.没有使用网络库进行通信，所以只能在本机上使用。

3.没有提供高效的并发性能，所有请求都在单独的线程中处理。

4.没有考虑RPC服务的可用性和高可用性，如果服务端崩溃或不可用，客户端将无法继续使用服务。

5.没有考虑RPC服务的可扩展性，如果有大量请求需要处理，可能会导致性能问题。

6.使用了第三方库Boost.Serialization来实现序列化和反序列化，如果不想使用第三方库，可能需要自己实现序列化的功能。

下面我们一步一步完善它。

三、加强版RPC（以“RPC简单实现”为基础）

3.1 加入错误处理

下面是 RPCHandler 类中加入错误处理的代码示例:

```
class RPCHandler {
public:
    // 其他代码...
    std::string handleRequest(const std::string& request) {
        // 反序列化请求
        std::map<std::string, std::string> requestMap;
        std::istringstream is(request);
        boost::archive::text_iarchive ia(is);
        ia >> requestMap;
        // 查找并调用对应的回调函数
        std::string name = requestMap["name"];
        std::string args = requestMap["args"];
        std::unique_lock<std::mutex> lock(mtx_);
        auto it = callbacks_.find(name);
        if (it == callbacks_.end()) {
            return "Error: Unknown function";
        }
        RPCCallback callback = it->second;
        try {
            return callback(args);
        } catch (const std::exception& e) {
            return "Error: Exception occurred: " + std::string(e.what());
        } catch (...) {
            return "Error: Unknown exception occurred";
        }
    }
};
```

上面的代码在 RPCHandler 类的 handleRequest 函数中加入了错误处理的代码，它使用了 try-catch 语句来捕获可能发生的异常。如果找不到对应的函数或发生了异常，会返回错误信息。这样，如果请求格式不正确或函数不存在，服务端将会返回相应的错误信息。

3.2 加入网络连接（socket）

加入网络连接不需要动服务端的实现，只需要在main里创造套接字去链接就好：

```
int main()
{
    io_context ioc;
    ip::tcp::acceptor acceptor(ioc, ip::tcp::endpoint(ip::tcp::v4(), 8080));
    RPCHandler rpcHandler;
    // 注册函数
    rpcHandler.registerCallback("add", [](const std::string& args) {
        std::istringstream is(args);
        int a, b;
        is >> a >> b;
        int result = a + b;
        std::ostringstream os;
        os << result;
        return os.str();
    });
    rpcHandler.registerCallback("sub", [](const std::string& args) {
        std::istringstream is(args);
        int a, b;
        is >> a >> b;
        int result = a - b;
        std::ostringstream os;
        os << result;
        return os.str();
    });
    // 等待连接
    while (true) {
        ip::tcp::socket socket(ioc);
        acceptor.accept(socket);
        // 创建线程处理请求
        std::thread requestThread([&](ip::tcp::socket socket) {
            while (true) {
                // 读取请求
                boost::asio::streambuf buf;
                read_until(socket, buf, '\n');
                std::string request = boost::asio::buffer_cast<const char*>(buf.da
                request.pop_back();
                // 处理请求
                std::string response = rpcHandler.handleRequest(request);
                // 发送响应
                write(socket, buffer(response + '\n'));
            }
```

```
        }, std::move(socket));
        requestThread.detach();
    }
    return 0;
}
```

这是一个使用Boost.Asio库实现的RPC服务端代码示例。它使用了TCP协议监听8080端口，等待客户端的连接。当有客户端连接时，创建一个新线程来处理请求。请求和响应通过网络传输。

3.3 加强并发性

使用并发和异步机制，忽略重复代码，实现如下：

```
class RPCHandler {
public:
    // ...
    void handleConnection(ip::tcp::socket socket) {
        while (true) {
            // 读取请求
            boost::asio::streambuf buf;
            read_until(socket, buf, '\n');
            std::string request = boost::asio::buffer_cast<const char*>(buf.data()
            request.pop_back();
            // 使用并行执行处理请求
            std::vector<std::future<std::string>> futures;
            for (int i = 0; i < request.size(); i++) {
                futures.emplace_back(std::async(std::launch::async, &RPCHandler::h
            }
            // 等待所有请求处理完成并发送响应
            for (auto& f : futures) {
                std::string response = f.get();
                write(socket, buffer(response + '\n'));
            }
        }
    }
};
```

这样，请求会被分成多个部分并行处理，可以利用多核 CPU 的优势提高服务端的并发性能。

main () :

```
int main() {
    io_context ioc;
    ip::tcp::acceptor acceptor(ioc, ip::tcp::endpoint(ip::tcp::v4(), 8080));
    RPCHandler rpcHandler;
    // 注册函数
```

```
    rpcHandler.registerCallback("add", [](const std::string& args) {
        std::istringstream is(args);
        int a, b;
        is >> a >> b;
        int result = a + b;
        std::ostringstream os;
        os << result;
        return os.str();
    });
    rpcHandler.registerCallback("sub", [](const std::string& args) {
        std::istringstream is(args);
        int a, b;
        is >> a >> b;
        int result = a - b;
        std::ostringstream os;
        os << result;
        return os.str();
    });
    // 创建线程池
    boost::thread_pool::executor pool(10);
    // 等待连接
    while (true) {
        ip::tcp::socket socket(ioc);
        acceptor.accept(socket);
        // 将请求添加到线程池中处理
        pool.submit(boost::bind(&RPCHandler::handleConnection, &rpcHandler, std::m
    }
    return 0;
}
```

在 main 函数中可以使用 boost::thread_pool::executor 来管理线程池，在线程池中提交任务来处理请求。这里的线程池大小设置为10，可以根据实际情况调整。

3.4 加入容错机制（修改客户端部分）

在其中使用了重试机制来保证客户端能够重新连接服务端：

```
class RPCClient {
public:
    RPCClient(const std::string& address, int port) : address_(address), port_(por
        connect();
    }
    std::string call(const std::string& name, const std::string& args) {
        // 序列化请求
        std::ostringstream os;
        boost::archive::text_oarchive oa(os);
        std::map<std::string, std::string> request;
        request["name"] = name;
```

```
        request["args"] = args;
        oa << request;
        std::string requestStr = os.str();
        // 发送请求
        write(socket_, buffer(requestStr + '\n'));
        // 读取响应
        boost::asio::streambuf buf;
        read_until(socket_, buf, '\n');
        std::string response = boost::asio::buffer_cast<const char*>(buf.data());
        response.pop_back();
        return response;
    }
private:
    void connect() {
        bool connected = false;
        while (!connected) {
            try {
                socket_.connect(ip::tcp::endpoint(ip::address::from_string(address
                connected = true;
            } catch (const std::exception& e) {
                std::cerr << "Error connecting to server: " << e.what() << std::en
                std::this_thread::sleep_for(std::chrono::seconds(1));
            }
        }
    }
    std::string address_;
    int port_;
    io_context io_context_;
    ip::tcp::socket socket_;
};
```

在这个示例中，当连接服务端失败时，客户端会在一定的时间间隔后重试连接，直到成功连接上服务端为止。

c++中的stack和dequeue解析
上一篇

C++多线程传参的实现方法
下一篇