[计算机网络大作业]

[RPC 实验报告]

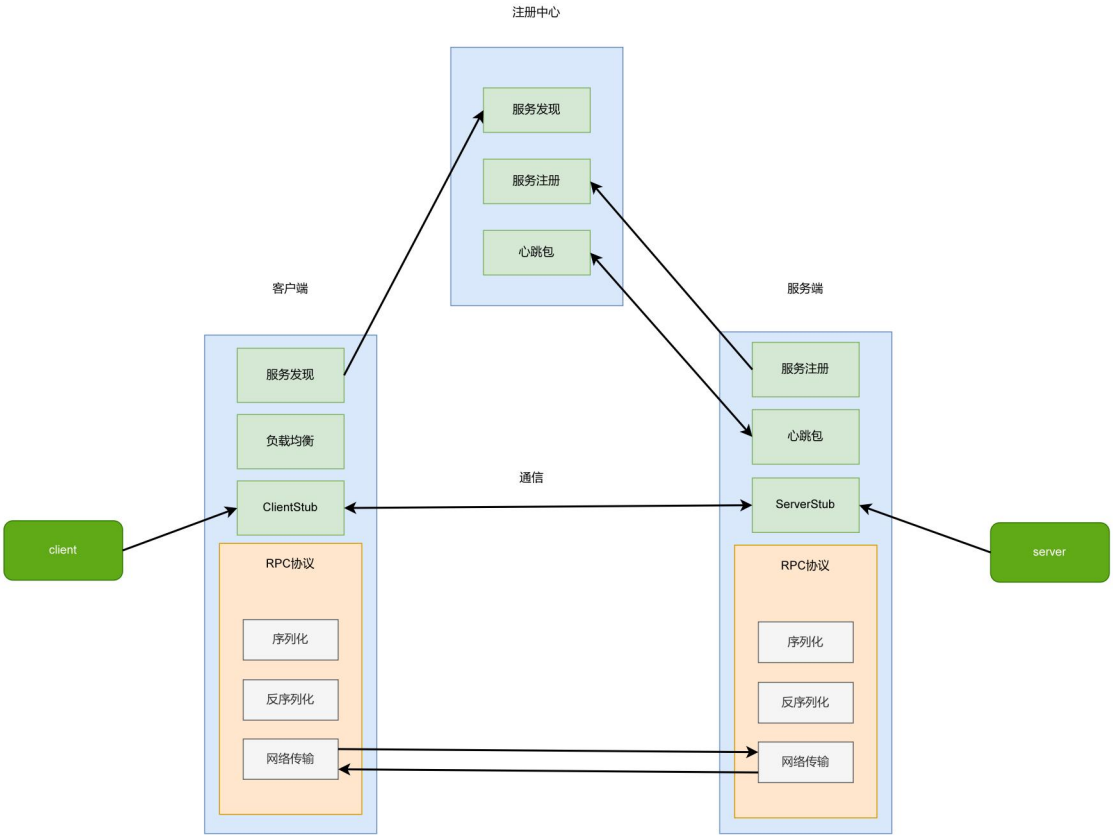
姓名：贺龙

学号：22330132

[2024.7.10]

目录

一、RPC 框架整体架构图



RPC 架构说明：

- **客户端**
  - **服务发现**：注册中心提供的接口，客户端能够通过调用此接口获取服务端是否支持其希望调用的服务。
  - **负载均衡**：为了减少服务端的负载，客户端可以通过服务发现模块获取支持该调用服务的服务器列表，根据随机选择策略（从服务器列表中随机选择一个服务器），选择通信的服务器。
  - **ClientStub**：类似客户端代理模块，客户端通过该模块的接口来实现远端服务器的调用近似为本地调用一样，同时客户端的服务发现、负载均衡模块均实现在此模块中，以此达到简化客户端的目的，客户端只需要像本地调用一样调用自己想要的服务就好了。
- **服务端**

- **服务注册：**注册中心提供的接口，服务端通过将本地服务注册到注册中心，此时就不要硬编码服务端的地址，客户端只需要通过注册中心来获取所需服务，服务端也只需要通过注册中心来注册和发布服务。
- **心跳包：**服务端定时向注册中心发送心跳包来表明自己的状态（active、dead）。
- **ServerStub：**类似服务端代理，服务端通过调用此模块的服务注册接口，即可快速注册服务，而与客户端的通信则只需要通过该代理模块实现，可以大大简化服务端的使用（只需要调用服务注册接口即可）。

- **注册中心**
  - **服务发现：**客户端通过发送自己希望调用的服务名，注册中心根据本地的服务表返回支持该服务的服务器列表。
  - **服务注册：**服务端通过发送自己希望注册的服务名，注册中心进行注册服务（缓存到本地的服务表中），并向服务端返回注册情况。
  - **心跳包：**注册中心收到服务端发送的心跳包后，检查与上一次接收到的心跳包的时间之差是否大于其设定的超时时间，并做相关的处理（删除该服务器的服务、发送重新注册响应）。

二、消息格式定义和消息的序列化/反序列化

1. 消息格式定义

```
1. syntax = "proto3";
2.
3. package rpc;
4.
5. message Server {
6.     string host = 1;
```

```
    int32 port = 2;
}

message AddRequest {
    int32 a = 1;
    int32 b = 2;
}

message AddResponse {
    int32 sum = 1;
}

message Request {
    string type = 1;
    string service_name = 2;
    string time = 3;
    Server server = 4;
    AddRequest add = 5;
}

message Response {
    string type = 1;
    string content = 2;
    string time = 3;
    repeated Server servers = 4;
    AddResponse add = 5;
}

service Calculator {
    rpc add (AddRequest) returns (AddResponse);
```

```
37. }
```

以上是消息定义的 .proto 文件，以下是简要说明：

- **message:** 表示自定义的消息结构，举 ”AddRequest” 为例，包含两个字段 a 和 b，int32 表示其类型，和结构体非常类似。
- **service：**因为这个 service 需要用到 grpc（代码中并未使用），所以这里只是解释一下其含义，”Calculator ” 表示一个服务，其中有一个支持 rpc 的 add 方法，参数为 “AddRequest”，返回值为 “AddResponse”。

其他思考：

- 为什么不直接采用类似下面的消息格式定义？虽然通过直接在代码中采 JSON 格式的消息定义，比如服务调用请求，定义一个服务名和参数（python 中还直接使用 *args 这样的列表当作传入参数），好像这样会使消息定义变得更简单，但是却存在以下问题：
  - 客户端的用户可能不知道该参数表示的类型。
  - 在消息定义放在 rpc 代码之中，而不是统一到一个单独的文件之中，有时候会造成混乱。
  - .proto 文件相当于是客户端和服务端两者共有的文件，因此客户端能够知道如何调用服务端的一些服务，而以上方式对客户端而言是不可见的。

```
1. message {
2.   "service_name": "method",
3.   "args": [arg1, arg2, ...],
4.   "kwargs": {"key": value, ...}
5. }
```

- 虽然 .proto 文件的方式是参考 grpc 的一些实现方式，但是 .proto 文件却有些复杂化问题了。例如，当服务变多时，需要定义的消息类型就会变多，在一定程度上不利于管理和实现。

注：得到上述 .proto 文件后，要使用以下命令将其转化为 .py 文件（实际上也

可以自己定义一个类似的类文件）。

```
protoc --python_out=. rpc.proto
```

2. 消息的序列化/反序列化

本项目因为使用了上述的 .proto 文件，因此也采用与其相关的 Protobuf 序列化方法，同时该方法和其他序列化方法（json、xml）相比，优点如下：

- **高效性**：Protobuf 使用二进制编码，相比于 XML 和 JSON 等文本格式，其序列化后的数据体积更小。
- **速度快**：Protobuf 的序列化和反序列化速度非常快，通常是 XML 和 JSON 的几十倍到上百倍。

三、服务注册

1. 服务注册流程

① 服务端使用 ServerStub 中服务注册接口向注册中心发送服务注册请求，请求消息定义如下。

```
message Request {
    string type = 'register';
    string service_name = 'service_name';
    Server server = (host=, port=);
}
```

② 注册中心接收到服务注册请求后，进行服务注册，储存在本地的服务表之中。

③ 注册中心根据注册结果返回相应的响应（当然响应也有不同类型，这里不在一一例举）。

2. 服务组织的数据结构

注册中心在本地通过维护一个**字典**，以服务名称为键，服务器的地址为值，从而实现服务的组织。这样，可以方便地通过服务名称快速得到对应的服务器地址。当然也可以通过其他方式，如现有的框架如 ZooKeeper、Eureka、Consu 来实现。

3. 服务注册接口

服务注册接口如下（因为代码较长的原因，这里使用图片来代替代码块）：

```python
def __register_service(self, service_name):
    request = Request(
        type='register',
        service_name=service_name,
        server=Server(host=self.host, port=self.port),
    )
    response = Response()
    response = self.__connect(self.registry_host, self.registry_port, request, response)
    # 超时/出错（注册失败），重新连接一次
    if response.type == 'timeout' or response.type == 'error':
        response = self.__connect(self.registry_host, self.registry_port, request, response)
        # 还是失败的话，抛出错误
        if response.type == 'timeout' or response.type == 'error':
            raise Exception(response.content)
```

简单来说，先构造 request 和 response 的类型，然后服务端与注册中心通信即可，最后返回服务注册的响应。

四、服务发现

1. 服务发现接口

服务发现接口如下：

```python
def __discover(self, service_name):
    request = Request(type='discover', service_name=service_name, )
    response = Response()
    response = self.__connect(self.registry_host, self.registry_port, request, response)
    # 超时
    if response.type == 'timeout':
        response = self.__connect(self.registry_host, self.registry_port, request, response)
        # 还是失败的话，抛出错误
        if response.type == 'timeout':
            raise Exception(response.content)
    # 出错（不存在该服务）
    elif response.type == 'fail':
        raise Exception(response.content)
    return response.servers
```

依旧是先构造 Request 和 Response 类型，然后客户端与注册中心通信即可，最后返回服务发现的响应。

2. 服务端如何找到服务

当客户端发起服务调用后，服务端会收到客户端发来的调用请求，其中包含有相关的服务的信息，例如服务名称、所需参数，本项目通过在本地实现一个和客户中心相应的**服务表**（字典），从而快速找到所需服务，而不是每一次以 O(n)的时间复杂度来找到服务。当然，也可以通过一些**反射机制**来实现。

五、服务调用

1. 输入/输出结构

关于服务调用相关的输入/输出结构，其实也就是上述相关消息的定义（详情见“二、消息格式定义和消息的序列化/反序列化”部分）。消息结构形式如下：

```
// 输入
message Request {
    string type = 'call';
    string service_name = 'service_name';
    AddRequest add;
}

```

```
8. // 输出
9. message Response {
10.     string type = 'success';
11.     SubResponse sub;
12. }
```

2. 如何将结果组织到其结构

六、注册中心

1. 如何注册

关于如何注册到注册中心的问题，可见上文“四、服务注册”部分。

2. 是否永久注册

使用服务注册接口实现的注册并不是永久注册，原因如下：

- 服务器有时可能会因为某些原因而关闭，这时候注册在注册中心的服务就会失效。
- 服务器的服务并不是永远不变的，有时候可能会发生变化，如服务的增删，服务内容的改变等等。

3. 心跳检测

鉴于上述 2. 中的问题，本项目实现了相关的心跳检测来保证服务的有效性。

① 服务端通过定时发送表明自己还存活（active）的**心跳包**给注册中心。

② 注册中心通过接收该**心跳包**的时间和上一次接收时间相比，检查是否超过了预定的超时时间，如果超时则删除该服务端注册的相关服务，并发送响应让服务端进行再次注册。

七、并发

1. 如何实现并发

本项目通过多线程实现并发操作，以便服务端能够并行处理客户端的请求。**线程池——**一种基于池化技术的多线程管理工具，用于减少在创建和销毁线程过程中的开销，使得能够重用已创建的线程。本项目通过使用线程池替代一般情况下的 threading 模块，能够极大地减少资源的浪费。使用 concurrent.futures.ThreadPoolExecutor 来替换原来的多线程实现，以便更有效地管理线程池。

2. IO 多路复用

IO 多路复用是一种在单个线程中管理多个输入/输出通道的技术。它允许一个线程同时监听多个输入流（如网络套接字、文件描述符等），并在有数据可读或可写时进行相应的处理，而不需要为每个通道创建一个独立的线程。但是并项目并未使用 IO 多路服用的技术。

八、异常处理和超时处理

1. 存在的异常和超时情况

本项目中存在许多需要进行异常处理和超时处理的地方，例如：

- 与服务端建立连接时产生的异常/超时
- 调用映射服务的方法时，处理数据导致的异常/超时
- 等待服务端处理时，等待处理导致的异常/超时（比如服务端已挂死，迟迟不响应）
- 从服务端接收响应时，读数据导致的异常/超时
- ……

2. 处理

处理方法主要如下所示：

- **对超时的处理：**采取一次重传的机制。
- **对一些错误的处理：**采取打印出错信息，并抛出错误的处理。

```
    # 超时
    if response.type == 'timeout':
        response = self.__connect(self.registry_host, self.registry_port, request, response)
        # 还是失败的话，抛出错误
        if response.type == 'timeout':
            raise Exception(response.content)
    # 出错（调用函数出错）
    elif response.type == 'fail':
        raise Exception(response.content)
    return response
```

```
# 超时/出错（注册失败），重新连接一次
if response.type == 'timeout' or response.type == 'error':
    response = self.__connect(self.registry_host, self.registry_port, request, response)
    # 还是失败的话，抛出错误
    if response.type == 'timeout' or response.type == 'error':
        raise Exception(response.content)
```

九、负载均衡

负载均衡——把每个请求平均负载到每个服务器上，充分利用每个服务器的资源。负载均衡的实现方式有很多，比如随机选择策略、轮询算法、加权轮询、哈希/

一致性哈希策略等，本项目采用的是随机选择策略。

```
def __call(self, service_name, request, response):
    servers = self.__discover(service_name)
    if not servers:
        pass
    # 采用随机选择策略实现负载均衡
    server = random.choice(servers)
    response = self.__connect(server.host, server.port, request, response)
    # 超时
    if response.type == 'timeout':
        response = self.__connect(self.registry_host, self.registry_port, request, response)
        # 还是失败的话，抛出错误
        if response.type == 'timeout':
            raise Exception(response.content)
    # 出错（调用函数出错）
    elif response.type == 'fail':
        raise Exception(response.content)
    return response
```

随机选择策略虽然在一定程度上没有其他方法高效，但其简单的形式亦受开发者的青睐。

十、网络传输协议

本项目选择 TCP 协议作为网络传输层协议。原因如下：

- TCP 协议的特性：
  - **面向连接**：TCP 是一种面向连接的协议，在数据传输前需要建立连接，数据传输结束后需要释放连接。这种机制保证了数据传输的可靠性和顺序性。
  - **可靠性**：TCP 通过序列号、确认应答、重传机制等手段来保证数据的可靠传输。在 RPC 调用中，确保远程调用的结果能够准确无误地返回给调用方是至关重要的。
  - **拥塞控制**：TCP 具有拥塞控制机制，可以根据网络的拥塞程度动态调整发送数据的速率，避免网络拥塞导致的数据丢失和延迟增加。这对于分布式系统中的 RPC 调用来说，能够提升系统的稳定性和响应速度。
  - **流量控制**：TCP 通过滑动窗口机制来进行流量控制，确保发送方不会发送超出接收方处理能力的数据量。这对于 RPC 调用中的大量数据传输非常有用。

- RPC 对传送协议的要求：
  - **高效性**：RPC 调用需要高效的数据传输协议来支持，以减少网络延迟和提高系统性能。TCP 协议的高效性和可靠性使其成为 RPC 调用的理想选择。
  - **可靠性**：RPC 调用通常涉及远程服务的调用和返回结果，因此需要确保数据传输的可靠性。TCP 协议的可靠性机制能够满足这一需求。
  - **适应性**：分布式系统中的网络环境复杂多变，TCP 协议能够自适应不同的网络环境，确保 RPC 调用的稳定性和可靠性。

虽然 UDP 协议具有较低的传输延迟和较小的开销，但它不提供可靠性保障，适用于对数据丢失不敏感的实时应用。因此，本项目选择了 TCP 协议来确保 RPC 调用的可靠性和数据完整性。

十一、使用教程

1. 启动**注册中心**，命令如下：

```
python registry.py
```

注：因为注册中心的特殊性（一般不存在多个注册中心），这里不使用命令行形式，而是将注册中心的信息写入到配置文件 config.yaml 中，通过 python 中的 yaml 模块进行获取。

```
1. Registry:
2.   host: 127.0.0.1
3.   port: 54321
```

2. 启动**服务端**，命令如下：

```
python server.py -i 127.0.0.1 -p 50000
```

解释如下：

- **“-i”**：表示服务端要监听的 ip 地址，默认为 ”0.0.0.0”，可以为空

- **“-p”**：表示服务端要监听的端口号，默认为 50000，不能为空

3. 启动**客户端**，命令如下：

```
python client.py -i 127.0.0.1 -p 54321
```

解释如下：

- **“-i”**：表示客户端要监听注册中心的 ip 地址，默认为 ”127.0.0.1”，可以为空
- **“-p”**：表示客户端要监听注册中心的端口号，默认为 54321，可以为空

注：因为本项目实现了注册中心功能，所以客户端仅需要连接到注册中心即可，注册中心会提供相关服务端的信息（这里也可以直接从配置文件 config.yaml 中获取相关信息）。