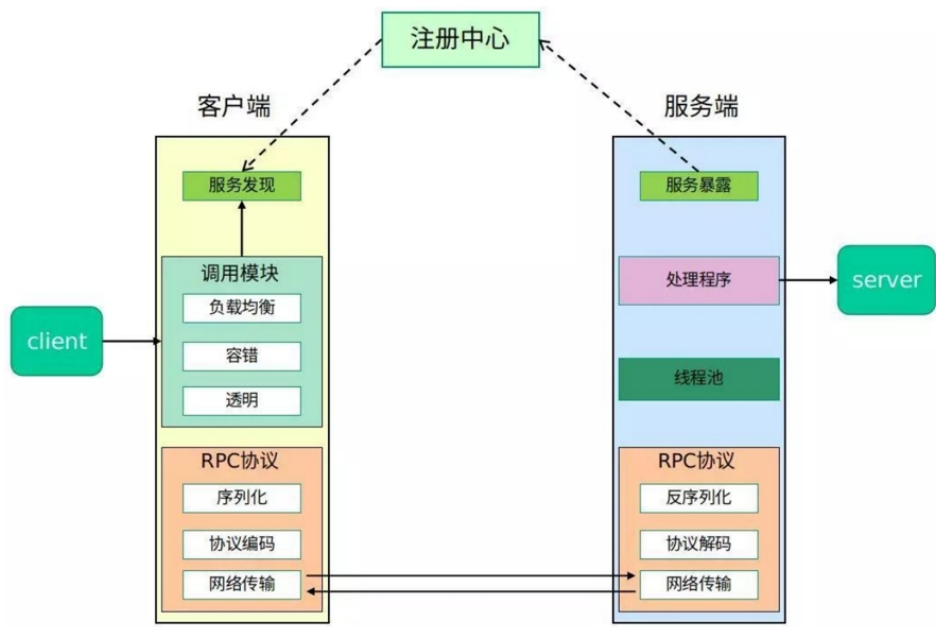


RPC实验文档

一、rpc框架设计思路

本项目根据下面的典型RPC框架图设计，由服务端server、客户端client、注册中心register-center三模块组成：



其中，预期各模块具备的功能如下：

服务端Server:

- 对客户端client：
 - 能接收、解码、处理来自客户端的遵从规定的请求数据格式的序列化数据，并返回处理结果；
 - 具有处理并发请求的能力；
 - 具有应对客户端连接中断等异常的处理能力；
- 对注册中心register-center：
 - 能向注册中心注册服务，并定期向其发送心跳表示服务活性；
 - 具有应对注册中心断连、服务端服务中断等异常的处理能力；
- 对自身：
 - 能优雅地主动/被动结束服务（注册中心正常服务正常时/注册中心断连服务正常时/注册中心正常服务断连时/注册中心断连服务断连时）

客户端Client:

- 对服务端server:

- 能按规定的请求数据格式序列化请求数据并发送至服务端，能接收、解码来自服务端的处理结果；
- 具有应对如服务端连接异常的处理能力；
- 对注册中心register-center：
 - 能从注册中心发现服务，设置本地服务缓存，定期轮询注册中心更新本地服务缓存；
 - 具有应对注册中心断连等异常的处理能力；
- 对自身：
 - 能采用某种负载均衡策略，从获取到的服务列表中选取此次调用使用的服务端；
 - 能在调用结束后优雅地清理RPCClient用到的资源；

注册中心server:

- 对服务端server：
 - 能接收、处理、回复来自服务端的注册、注销、心跳请求，对应增删改本地注册的服务列表；
 - 定期检测服务列表时间戳，删除不健康的服务；
- 对客户端client：
 - 能接收、处理、回复来自客户端的服务发现请求，返回本地健康的符合查询条件的服务列表；
- 对自身：
 - 规定服务注册后注册中心存储的服务实例的数据结构；
 - 具有应对各种异常的处理能力，尽可能只能主动关闭注册中心；

二、RPC框架设计实现

2.1 整体项目目录结构

项目目录结构如下:

```
E:\PYPROJECTS\RPC
|
|  config.ini                # 配置文件，存放项目的一些配置参数
|  docker-compose.yml       # 与下面Dockerfile一起负责构建docker测试环境
|  Dockerfile
|  README.md                # 项目文档
|
|—client
|   client.py               # RPC客户端代码
|
|—registry
|   registry.py             # 注册中心代码
|
|—server
|   server.py               # RPC服务器代码
```

整个项目由配置文件、Docker相关文件、客户端、注册中心和服务端代码构成。

2.2 消息序列化和反序列化方式与消息格式定义

本项目使用 json 作为消息的序列化和反序列化方式，消息格式定义如下：

- 请求方法调用的消息格式定义：

```
{
  "method_name": "请求方法名",
  "method_args": "请求方法参数",
  "method_kwargs": "请求方法关键字参数"
}
```

- 响应方法调用的消息格式定义：

```
{
  "res": "方法调用结果"
}
```

2.2 rpc服务端的实现

server.py用到的库：

```
import argparse          # 用于编写启动参数
import configparser       # 用于读取配置文件
import http.client        # 用于发送HTTP请求
import inspect            # 用于响应用户服务发现时提取存储函数的参数
import json               # 用于处理JSON数据
import math               # 用于数学运算，注册函数编写时用到
import os                 # 用于操作系统相关的功能（如日志文件路径）
import socket             # 用于TCP通信
import threading          # 用于多线程处理
import time               # 用于时间相关的操作
from datetime import datetime
```

server.py代码结构：

```

> InstanceMeta
> Logger
> ServerStub
> RegistryClient
> TCPServer
> RPCServer(TCPServer)
  add(a, b, c=10)
  subtract(a, b)
  multiply(a, b)
  divide(a, b)
  area_of_circle(radius)
  square(number)
  to_uppercase(string)
  reverse_string(string)
  hi(user)
  hello(name)

```

其中：

- **InstanceMeta**: 与注册中心通信，注册与保活服务时约定的服务实例数据结构，序列化方式采用 json：

```

{
    "protocol": "json", // 实例使用的序列化协议
    "host": "192.168.1.1", // 实例监听的 IP 地址
    "port": 8080, // 实例监听的端口号
    "status": "true", // 实例的注册状态
    "parameters": {} // 额外信息的字典，用于存储自定义参数
}

```

- **Logger**: 用于输出与存储日志信息，默认不存储仅输出，分为info与error两个级别。
- **ServerStub**: 作为服务端代理，负责处理服务注册与服务调用，并支持客户端进行服务发现：

代码结构：

```

ServerStub
  __init__(self, logger)
  register_services(self, method, name=None)
  call_method(self, req, client_addr)
  logger
  services

```

register_services：实现服务注册，函数签名与实现思路：

```

def register_services(self, method, name=None):
    """
    处理方法注册，把注册的方法以方法名为键，函数为值（python中的函数是第一类对象（first-class
    objects），可以像其他对象一样被传递、赋值、存储在如列表、字典等数据结构中）的方式存于成员变量
    services中
    :param method: function 要注册的方法
    :param name: string 要注册方法的名称，为空则默认为注册方法函数名
    """

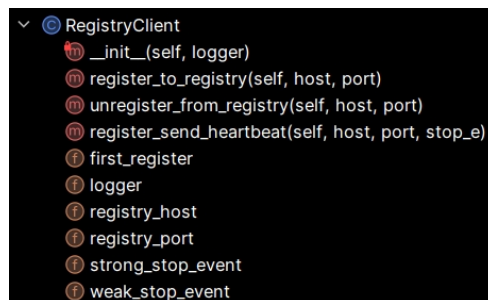
```

call_method: 实现服务调用与支持客户端进行服务发现, 函数签名与实现思路:

```
def call_method(self, req, client_addr):
    """
    处理方法的调用, 解析请求,
    若请求的方法名为 'all_your_methods', 表示这是服务发现请求, 服务端返回已注册的函数列表
    (函数名与输入参数)
    若不是, 则为服务调用请求, 服务端寻找请求的注册方法, 返回调用成功或失败的回复消息
    :param req: 以json格式序列化后的请求方法调用消息
    :param client_addr: 调用方的 ip 地址, 运行日志记录需要
    :return: reply: 序列化后的调用结果信息 (调用成功/调用不存在方法/调用方法参数错误/其余
    方法处理时发生错误)
    """
```

- **RegistryClient:** 负责注册中心相关的功能, 向注册中心注册、注销服务, 并能定期向其发送心跳保持服务活性:

代码结构:



register_to_registry: 实现向注册中心注册服务, 并用于服务保活, 函数签名与实现思路:

```
def register_to_registry(self, host, port):
    """
    通过发送HTTP POST请求, 向注册中心注册服务, 得到注册请求的结果
    也是服务向注册中心发送心跳的方式, 通过成员变量 self.first_register 判断是注册请求还是心跳发
    送请求

    :param host: 注册服务的IP地址
    :param port: 注册服务的端口
    """
```

unregister_from_registry: 实现向注册中心注销服务, 函数签名与实现思路:

```
def unregister_from_registry(self, host, port):
    """
    通过发送HTTP POST请求，向注册中心注销服务，得到注销请求的结果

    :param host: 注册服务的IP地址
    :param port: 注册服务的端口
    """
```

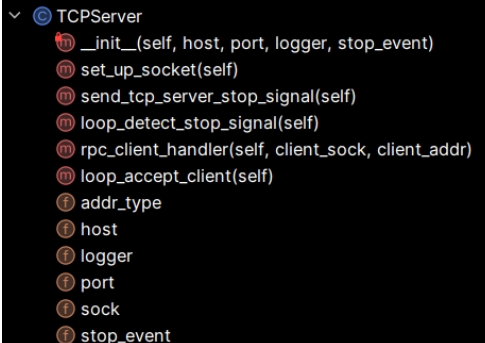
register_send_heartbeat: 实现服务保活，，函数签名与实现思路：

```
def register_send_heartbeat(self, host, port, stop_e):
    """
    注册服务并定期再次注册服务来表示发送心跳到注册中心，以实现服务保活，直到接收到停止信号。

    :param host: 注册服务的IP地址
    :param port: 注册服务的端口
    :param stop_e: 赋值 self.strong_stop_event 停止事件，用于控制心跳发送的停止
    """
```

- **TCPServer:** 此类负责封装TCP连接相关功能，使rpc服务端能并发处理客户端请求，并能够在收到停止信号时优雅关闭，RPCServer类通过继承此类实现与客户端的通信：

代码结构：



```
class TCPServer:
    def __init__(self, host, port, logger, stop_event):
    def set_up_socket(self):
    def send_tcp_server_stop_signal(self):
    def loop_detect_stop_signal(self):
    def rpc_client_handler(self, client_sock, client_addr):
    def loop_accept_client(self):
    addr_type
    host
    logger
    port
    sock
    stop_event
```

set_up_socket: 配置Socket选项，根据host确定使用IPv4还是IPv6，指定连接请求的最大等待队列长度>10以满足至少可以支持并发处理 10 个客户端的请求的要求；

send_tcp_server_stop_signal&loop_detect_stop_signal: 用于接受停止信号停止tcp服务器，函数签名与实现思路：

```
def send_tcp_server_stop_signal(self):
    """
    发送TCP服务器停止信号
    通过本地创建一个TCP客户端连接到服务器并马上关闭以触发服务器的accept方法
    解决accept不设timeout会无限期阻塞，无法进入下一次循环导致无法接收到停止信号的问题
    """

def loop_detect_stop_signal(self):
    """
    循环检测停止事件，如果检测到停止事件被设置，则发送服务器停止信号
    """
```

rpc_client_handler&loop_accept_client: **支持并发的实现**，函数签名与实现思路：

```
def rpc_client_handler(self, client_sock, client_addr):
    """
    处理每个客户端请求的handler，需要由继承的RPCServer实现具体处理逻辑

    :param client_sock: 客户端的Socket
    :param client_addr: 客户端的地址
    """
    pass

def loop_accept_client(self):
    """
    循环接受客户端连接，并为每个连接创建一个新的线程来处理请求以支持并发请求
    """
    while not 停止事件未被设置:
        client_sock, client_addr = self.sock.accept()
        # 开新线程处理此客户的请求，不影响server继续接受下一位客户
        t = threading.Thread(target=self.rpc_client_handler, args=(client_sock,
client_addr))
        t.start()
    self.sock.close() # 最后关闭自身socket
```

- **RPCServer**: 继承自TCPServer，并组合Logger、ServerStub和RegistryClient实现了完整的RPC服务功能：

代码结构：

```
 />
```

成员变量解释：除继承和组合的实现，此类设置了三个线程来管理服务的开启、运行与停止，详细由init函数注释解释：

```
def __init__(self, host, port):
    self.logger = Logger() # 运行日志创建
    self.stub = ServerStub(self.logger) # 设置服务端代理，负责处理服务端方法的注册与调用请求
    self.registry_client = RegistryClient(self.logger) # 设置注册中心客户端，负责与注册中心通信，注册和保活服务
    self.stop_event = threading.Event() # 停止事件，用于控制RPCServer的停止
    super().__init__(host, port, self.logger, self.stop_event) # 初始化父类TCPServer，传入要监听的ip与端口号
    # 创建三个线程，分别用于处理停止信号、接受TCP连接和向注册中心注册与发送心跳。
    self.loop_detect_stop_signal_thread =
threading.Thread(target=self.loop_detect_stop_signal)
    self.tcp_serve_thread = threading.Thread(target=self.loop_accept_client)
    self.register_and_send_hb_thread =
threading.Thread(target=self.registry_client.register_send_heartbeat,
                  args=(self.host, self.port,
self.stop_event))
```

rpc_client_handler：实现父类的处理每个客户端请求的handler，函数签名与实现思路：

```
def rpc_client_handler(self, client_sock, client_addr):
    """
    实现父类的处理每个客户端请求的handler
    在stop_event () 停止信号没有被设置时，循环接收客户端请求消息，
    用stub的方法处理每个客户端的RPC请求，并返回调用结果，最终关闭客户端socket
    :param client_sock: 客户端的Socket
    :param client_addr: 客户端的地址
    """
```

serve：是本项目rpcserver被创建后服务启动的函数，设计为主线程负责管理服务端与客户端通信，与注册中心通信和停止服务端这三个作业线程，监听外部中断来发送停止信号，并在所有线程停止后终止程序，函数签名与实现思路：

```
def serve(self):
    """
    启动RPC服务器，开始监听并处理客户端连接，
    启动检测停止信号、处理TCP连接和注册中心心跳的线程。
    """
    self.loop_detect_stop_signal_thread.start()...三个线程开启
    try:
        while True: # 主线程开启监听
            time.sleep(100)
```



```
except KeyboardInterrupt: # 检测到外部中断，发送停止信号
    ...
finally: # 在所有线程停止后终止服务
    ...
    exit(0)
```

- 结构中剩余的10个函数为测试服务端功能时编写的注册的方法:

```
"""要注册的函数们"""
def add(a, b, c=10):
    return a + b + c
""".....略"""
```

2.3 rpc客户端的实现

本项目rpc客户端分成已知服务端地址，直接与服务端通信调用服务模式与通过注册中心发现服务端调用服务模式，后者在服务发现时需先通过注册中心发现服务端。

client.py用到的库:

```
import argparse          # 用于编写参数
import configparser      # 用于读取配置文件
import http.client       # 用于发送HTTP请求
import json              # 用于处理JSON数据
import os                # 用于操作系统相关的功能（如日志文件路径）
import socket            # 用于TCP通信
import random            # 用于随机选择负载均衡服务器
import threading         # 用于多线程处理
import time              # 用于时间相关的操作
from datetime import datetime
```

client.py代码结构:

```
> Ⓢ LoadBalance
> Ⓢ Logger
> Ⓢ RegistryClient
> Ⓢ TCPClient
> Ⓢ RPCClient
> Ⓢ test_sync_calls(client)
> Ⓢ test_async_calls(client)
```

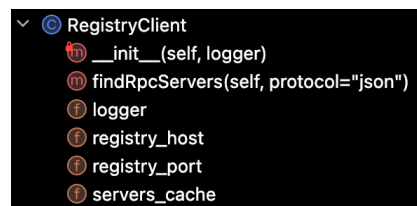
其中:

- **LoadBalance**: 负载均衡类，实现负载均衡功能，以静态方法方式提供负载均衡算法，本项目暂时只实现了随机负载均衡算法 `random`，后续可继续拓展:

```
class LoadBalance:
    @staticmethod
    def random(servers):
        s = random.choice(servers)
        return s
```

- **Logger:** 用于输出与存储日志信息，默认不存储仅输出，分为info与error两个级别，与server.py一致。
- **RegistryClient:** 负责与注册中心通信，能向注册中心请求**服务发现**获取可用的服务端列表并存至本地缓存的服务端列表：

代码结构：



成员变量解释：

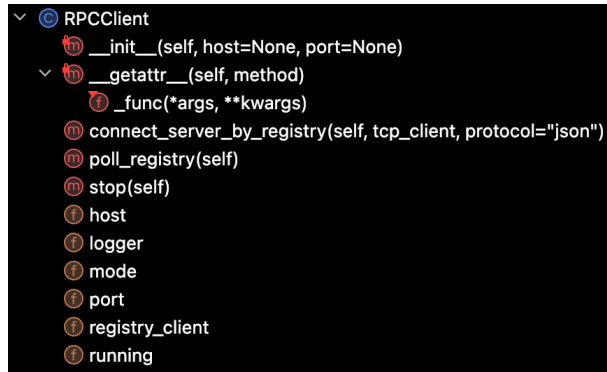
```
registry_host : string 配置文件中读入的注册中心的 IP
registry_port : int 配置文件中读入的注册中心的端口号
servers_cache = set() 本地缓存的服务端列表
logger: 运行日志
```

findRpcServers：根据客户端支持的消息序列化协议向注册中心请求实现此协议的服务端，默认json，或者也可以理解成实现了自己设置的rpc消息协议的服务端，比如此项目的protocol=json就指的是寻找实现了以json方式序列化的自定义消息数据格式的服务端，函数签名与实现思路：

```
def findRpcServers(self, protocol="json"):
    """
    http与注册中心通信，查询参数protocol为客户端使用的消息数据格式，
    将查询到的可用服务端列表与本地缓存的服务端列表比对，更新本地列表
    :return: tuple list 服务的 (host, port) 的元组 list
    """
```

- **TCPClient:** 基础的 TCP 客户端，封装了TCP通信socket的一些功能便于RPCClient的编写。
- **RPCClient:** 通过组合RegistryClient, Logger，在服务调用时使用TCPClient，实现了 RPC 客户端的功能，分成两种模式，使用注册中心进行服务发现然后调用，与不使用注册中心直接与服务端连接进行服务调用：

代码结构：



成员变量解释：

host：直接连接服务端模式时指定的服务端IP地址

port：直接连接服务端模式时指定的服务端端口号

logger：Logger 运行日志

registry_client：RegistryClient 客户端的RegistryClient实例，与注册中心通信

running：bool 运行状态标志，用于停止可能的轮询注册中心线程

mode：0(no registry) / 1(with registry) 如果使用注册中心，启动一个线程定期轮询注册中心

getattr&_func：客户端实现rpc服务调用的方法，函数签名与实现思路：

```
def __getattr__(self, method):
    """
    访问不存在属性时被调用的方法，动态创建一个代理函数_func，用于处理该方法调用，
    从而实现RPC远程调用；
    如 RPCClient调用add方法，即调用了对应的_func方法，
    将数据发送至Server端并返回远程调用返回的数据
    :param method: 试图访问的不存在的属性名
    :return: _func: 远程调用method后返回调用结果的函数
    """

    def _func(*args, **kwargs):
        """
        代理函数，用于调用Server端的方法；
        :param args: 远程调用位置参数
        :param kwargs: 远程调用关键字参数
        :return: 远程调用的结果
        """
        try:
            tcp_client = TCPClient(self.host, self.port)
            # 略模式选择
            ...
```

```

        # 按照规定的消息格式构建调用消息并发送
        dic = {'method_name': method, 'method_args': args, 'method_kwargs':
kwargs}

        tcp_client.send(json.dumps(dic).encode('utf-8'))
        # 接受调用结果，反序列化并取出结果
        response = self.recv(1024)
        result = json.loads(response.decode('utf-8'))
        result = result["res"]
        # 关闭本次请求socket
        tcp_client.close()
    except (json.JSONDecodeError, ConnectionError) as e:
        # 收结果异常处理
        return result

    setattr(self, method, _func)
    return _func

```

connect_server_by_registry: 通过注册中心连接服务端模式下连接服务端，函数签名与实现思路：

```

def connect_server_by_registry(self, tcp_client, protocol="json"):
    """
    通过注册中心连接服务端模式下连接服务端，此模式下轮询注册中心线程开启，
    优先使用本地服务端缓存，为空则调用registry_client的findRpcServers，若结果仍为空则抛
    出无可用服务端异常
    并在此处使用负载均衡类的负载均衡算法选出最终连接的服务端，进行连接
    :param tcp_client: TCPClient 与选出的server建立连接的tcp客户端
    :param protocol: 客户端使用的消息数据格式
    """

```

poll_registry: 轮询注册中心，定期从注册中心获取最新的服务器列表更新缓存。

stop: 停止客户端并关闭现有的socket连接。

- **test_sync_calls&test_async_calls**: 客户端同步调用测试与异步调用测试。

2.4 rpc注册中心的实现

registry.py用到的库：

```

import argparse
import json
import os
import socket
import threading
import time
from collections import defaultdict # 用于创建默认字典，存储不同协议的服务实例和实例时间戳
from datetime import datetime
from http.server import ThreadingHTTPServer, BaseHTTPRequestHandler # 用于创建多线程
HTTP服务器和处理HTTP请求
from typing import List # 用于类型注解，指定函数返回值为列表类型
from urllib.parse import urlparse, parse_qs # 用于处理请求时解析URL路径和查询字符串参数

```

registry.py代码结构：

```

> @ Logger
> @ InstanceMeta
> @ RegistryService
> @ RequestHandler(BaseHTTPRequestHandler)
❗ run(server_class=ThreadingHTTPServer, handler_class=RequestHandler, host='0.0.0.0', port=8081, registry_service=None, logger=None)

```

其中：

- **Logger:** 用于输出与存储日志信息，默认不存储仅输出，分为info与error两个级别，与server.py一致。
- **InstanceMeta:** 规定的服务实例数据结构，服务端进行服务注册注销时需要遵循此结构进行注册：

```

class InstanceMeta:
    """服务实例注册与发现使用的数据结构"""

    def __init__(self, protocol=None, host=None, port=None):
        self.protocol = protocol # 服务使用的序列化与反序列化的消息格式，如json
        self.host = host # 服务注册的ip地址
        self.port = port # 服务注册的端口号
        self.status = None # 服务注册状态，注销False，已注册状态True
        self.parameters = {} # 服务注册时附加参数，扩展可在参数上设条件细化对服务实例的管理
    """一些工具函数，于文档略..."""

```

- **RegistryService:** 负责处理服务的注册、注销、和健康检测：

代码结构：



成员变量解释:

```
logger = logger # 日志
_health_thread = threading.Thread(target=self.loop_check_health) # 心跳检测线程
_stop_event = threading.Event()
ins2timestamp = defaultdict(int) # 存各个服务实例的时间戳, 用于心跳检测
proto2instances = defaultdict(list) # 存不同消息协议对应的服务实例列表
```

register(self, ins: InstanceMeta): 处理服务实例注册, 若服务不存在于proto2instances[注册实例消息协议]里, 则服务实例为第一次注册, 将其存入proto2instances[注册实例消息协议]并获取当前时间戳, 将实例与其对应时间戳存入ins2timestamp[实例]中, 输出服务实例成功注册日志信息, 返回被成功注册的服务实例; 若服务已存在, 则更新ins2timestamp中对应实例的时间戳, 输出服务实例重复注册信息, 返回被重复注册的服务实例。

unregister(self, ins: InstanceMeta): 处理服务实例注销, 若服务不存在于proto2instances[注册实例消息协议]里, 则输出注销实例不存在的信息, 考虑可能是实例状态信息错误, 将实例状态status设置为False后返回服务实例; 若存在, 则从proto2instances[注册实例消息协议]中删去此实例, 并删除ins2timestamp[实例], 输出服务实例注销失败信息, 将实例状态status设置为False后返回服务实例。

find_instances_by_protocol(self, protocol="json"): 根据消息协议返回对应服务实例列表。

loop_check_health&handle_check_health&stop: 实现对注册的服务实例的定期健康检测, 移除不通过健康检测的服务实例, 并能在注册中心服务器停止后停止负责定期健康检测的线程, 代码思路:

```
def handle_check_health(self):
    """对服务实例进行健康检测"""
    cur_time = int(time.time()) # 获取当前时间戳
    threshold # 设置健康标准
    if not self.ins2timestamp: # 若服务实例列表为空
        pass or 输出日志信息
    else: # 不为空, 挨个检查
        for ins, timestamp in list(self.ins2timestamp.items()):
            if cur_time - timestamp > threshold: # 不满足健康标准
                self.unregister(ins) # 注销此实例

def stop(self):
    """停止心跳检测线程"""
```

```

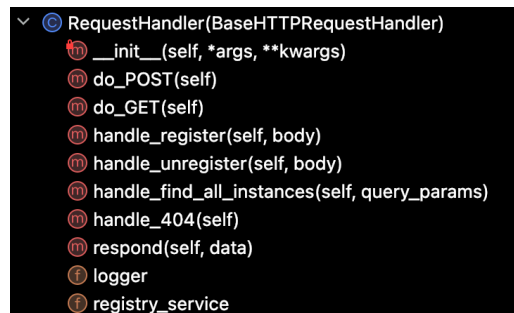
self._stop_event.set() # 设置停止事件
self._health_thread.join() # 等待线程结束

def loop_check_health(self):
    """定期健康检测，当停止事件未被设置时循环，每5s进行一次健康检查"""
    while not self._stop_event.is_set():
        self.handle_check_health()
        self._stop_event.wait(5)

```

- **RequestHandler**: 继承于**BaseHTTPRequestHandler**，负责处理HTTP请求，并根据不同的路径执行相应的注册中心功能：

代码结构：



```

RequestHandler(BaseHTTPRequestHandler)
  __init__(self, *args, **kwargs)
  do_POST(self)
  do_GET(self)
  handle_register(self, body)
  handle_unregister(self, body)
  handle_find_all_instances(self, query_params)
  handle_404(self)
  respond(self, data)
  logger
  registry_service

```

成员变量解释：

```

registry_service : RegistryService 实际处理服务实例
logger : Logger 运行日志

```

do_POST：处理服务注册与注销请求的路由 `'/myRegistry/register'` ， `'/myRegistry/unregister'` ，`，其他POST请求返回404。

do_GET：处理获取服务实例列表请求的路由 `'/myRegistry/findAllInstances'` ，其他GET请求返回404。

handle_register&handle_unregister&handle_find_all_instances&handle_404：分别处理服务注册、注销、发现及无效请求的逻辑。

respond：构造并发送HTTP响应。

- **run(...)**: 启动注册中心HTTP服务器：

```
def run(server_class=ThreadingHTTPServer, handler_class=RequestHandler,
        host='0.0.0.0', port=8081,
        registry_service=None, logger=None):
    """
    启动注册中心

    :param server_class: HTTP服务器类，默认为ThreadingHTTPServer，用于处理并发请求
    :param handler_class: 请求处理类，默认为RequestHandler，定义了注册中心各路由的处理方法
    :param host: 注册中心监听的IP地址，默认为'0.0.0.0'，即监听所有可用的网络接口
    :param port: 注册中心监听的端口号，默认为8081
    :param registry_service: 注册中心服务实例，用于管理注册和注销的服务实例
    :param logger: 日志记录实例，用于记录服务器运行状态和事件
    :return: None
    """
```

三、异常处理及超时处理

(1) 客户端处理异常/超时的地方：

1. 向注册中心请求服务端列表时产生的异常/超时处理：

```
# 在 client.py class RegistryClient 的 findRpcServers中进行处理
try:
    conn.request("GET", f"/myRegistry/findAllInstances?proto={protocol}")
    response = conn.getresponse()
    if response.status == 200:
        # 正常，返回找到的服务端列表
    else:
        # 不正常响应码，返回空列表，本次与服务端进行连接从本地缓存服务列表中选取
except (TimeoutError, ConnectionRefusedError) as e:
    # 捕捉超时与连接异常，本次与服务端进行连接从本地缓存服务列表中选取
finally: # 最终关闭http连接
    conn.close()
```

2. 与服务端建立连接时，发送请求到服务端时，等待服务端处理时，从服务端接收响应时产生的异常/超时处理：

```
# 在 client.py class RegistryClient 的 __getattr__ 的 _func 中进行处理
try:
    #...
    if 直接连接服务端模式:
        tcp_client.sock.settimeout(10) # 设置处理超时限制
        tcp_client.connect()
    else:
        # 通过注册中心连接服务端模式，connect_server_by_registry中未发现可用服务端或者与服务
        端连接出现异常时将抛出Exception
```



```

        self.connect_server_by_registry(tcp_client)
    #...
    tcp_client.send(json.dumps(dic).encode('utf-8')) # 发送请求, 出问题会被捕获
    response = tcp_client.recv(1024) # 接受消息, 出问题会被捕获
    result = json.loads(response.decode('utf-8'))["res"] # 反序列化出问题时也会被捕获
    #...
except Exception as e:
    输出异常信息日志, result=None返回

```

(2) 服务端处理异常/超时的地方:

1. 与注册中心通信的异常/超时处理: 位于RegistryClient的注销与心跳发送函数中:

```

except (TimeoutError, ConnectionRefusedError) as e:
    self.logger.error(f'与注册中心通信时发生错误: {e}, 停止与注册中心联系')

```

2. 与客户端连接、发送响应数据时发生的异常/超时的处理:

```

class RPCServer(TCPServer):
    def rpc_client_handler(self, client_sock, client_addr):
        try:
            while not self.stop_event.is_set():
                msg = client_sock.recv(1024)
                if not msg:
                    raise EOFError()
                response_data = self.stub.call_method(msg, client_addr)
                client_sock.sendall(response_data)
        except EOFError:
            self.logger.info(f'info on handle: 客户端{str(client_addr)}关闭了连接')
        except Exception as e:
            self.logger.error(f'except on handle: 客户端{str(client_addr)}异常地关闭了连接, {e}')
        finally:
            client_sock.close()

```

3. 调用映射服务的方法时, 处理数据导致的异常: 在 server.py class ServerStub call_method里, 捕获这几种异常并返回结果给客户端:

```

except KeyError:
    # 方法名不存在的情况
    res = f"No service found for: {method_name}"
except TypeError as e:
    # 方法参数错误的情况
    res = f"Argument error: {e}"
except Exception as e:
    # 其他调用错误的情况
    res = f"Error calling method: {e}"

```

(3) 注册中心处理异常/超时:

```

def run(server_class=ThreadingHTTPServer, handler_class=RequestHandler,
        try:
            # 启动服务器, 进入永远运行状态
            httpd.serve_forever()
        except KeyboardInterrupt:
            # 处理键盘中断 (Ctrl+C)
            logger.info("Main thread received KeyboardInterrupt, stopping...")
        except BaseException as e:
            # 处理其他异常
            logger.error(f"注册中心服务运行时出错: {e}, 停止运行")
        finally:
            # 停止注册中心服务
            registry_service.stop()
            logger.info("Registry service stopped.")
            exit(-1)

```

四、运行教程

根据运行环境是在本地还是docker，在client.py, server.py的RegistryClient初始化方法里选择相应的config:

```
config.read('docket_test_config.ini') # docker
# config.read('../config.ini') # local
```

服务端启动参数

以下是服务端启动参数的说明：

- `-l`, `--host`: 服务端监听的 IP 地址，支持 IPv4 和 IPv6，默认值为 `0.0.0.0`，即监听所有 IP 地址。
- `-p`, `--port`: 服务端监听的端口号，此参数为必填项。

示例命令：

```
python server.py -p 8089
```

客户端启动参数

以下是客户端启动参数的说明：

- `-i`, `--host`: 客户端需要发送的服务端 IP 地址，支持 IPv4 和 IPv6，此参数在 `server` 模式下为必填项。
- `-p`, `--port`: 客户端需要发送的服务端端口，此参数在 `server` 模式下为必填项。
- `-m`, `--mode`: 客户端运行模式，默认值为 `server`，可选值为 `registry` (通过注册中心发现服务)和 `server` (直接与服务端相连)。在 `registry` 模式下，无需指定 `host` 和 `port` 参数。

示例命令：

```
python client.py
python client.py -i 127.0.0.1 -p 8089 -m server
```

注册中心启动参数

以下是注册中心启动参数的说明：

- `-l`, `--host`: 注册中心监听的 IP 地址，支持 IPv4 和 IPv6，默认值为 `0.0.0.0`，即监听所有 IP 地址。
- `-p`, `--port`: 注册中心监听的端口号，此参数为必填项。

示例命令：

```
python registry.py -p 9999
```