**高性能高可用RPC框架**

**一、项目概述**

在现代分布式系统中，高可用性和高可靠性是至关重要的特性。为了满足不同服务之间的通信需求，保证系统在面对部分节点故障时仍能提供连续服务，本项目设计并实现了一个高可用性PRC（Procedure Remote Call）框架。该框架旨在通过分布式注册中心和高效的通信机制，实现服务的注册、发现、负载均衡及故障恢复，确保系统的稳定运行。

**1、目标**

高可用性: 实现一个能够在节点故障时仍然正常运行的PRC框架。

高性能: 提供低延迟、高吞吐量的RPC通信能力。

可扩展性: 支持大规模服务部署，确保系统在扩展时性能线性增长。

健壮性: 提供完善的服务注册与发现机制，确保服务之间的调用可靠。

**2、核心组件**

非对称协程库: 在Linux环境下实现的简易非对称协程库，用于提高并发处理能力和资源利用率。

TCP服务器: 基于IO多路复用和Reactor模式，实现高效的TCP服务器，提供可靠的网络通信能力。

分布式注册中心: 基于Raft一致性算法，实现分布式注册中心集群，提供服务注册、发现、订阅和通知功能。

**3、服务功能**

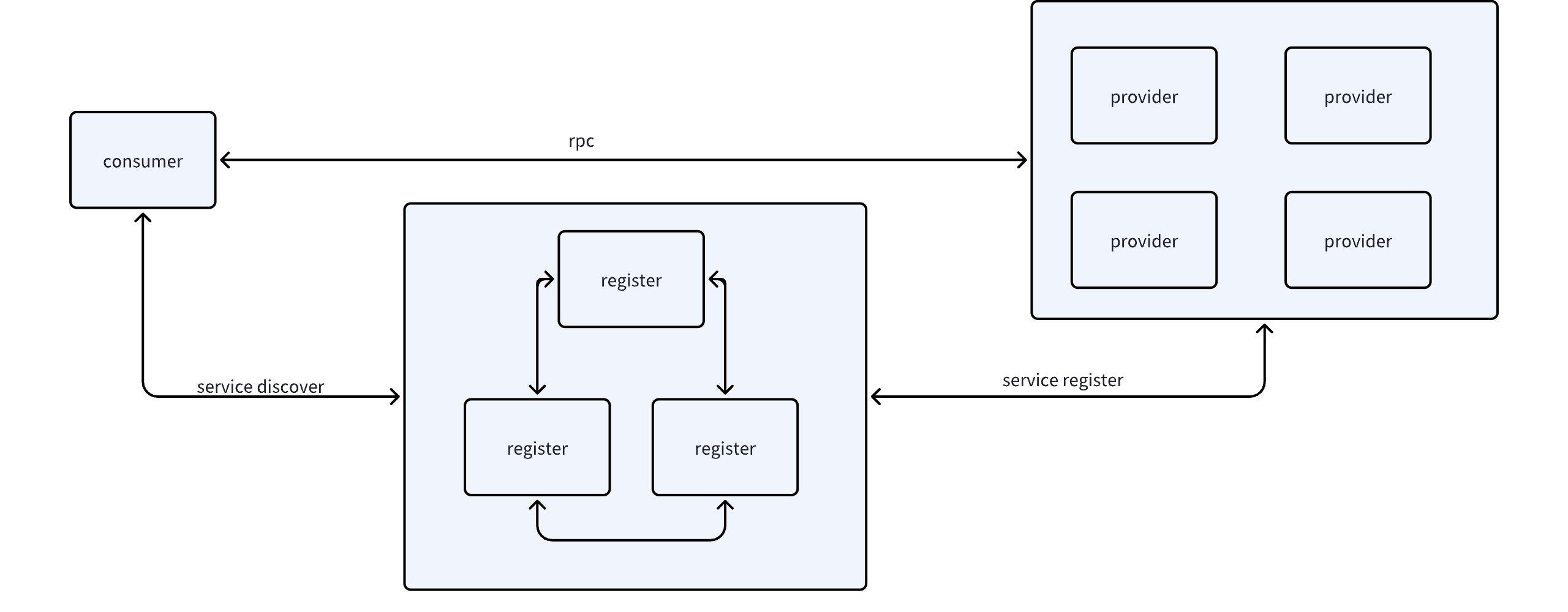
连接复用: 通过连接池和复用技术，提高网络通信的效率和稳定性。

服务注册与发现: 实现自动化的服务注册和发现机制，确保服务的动态扩展和灵活调用。

负载均衡: 基于多种策略（如IP一致性哈希、最小连接数（注册中心返回连接较少的一些服务提供者，消费者通过加权轮询）等），实现高效的负载均衡，优化资源使用。

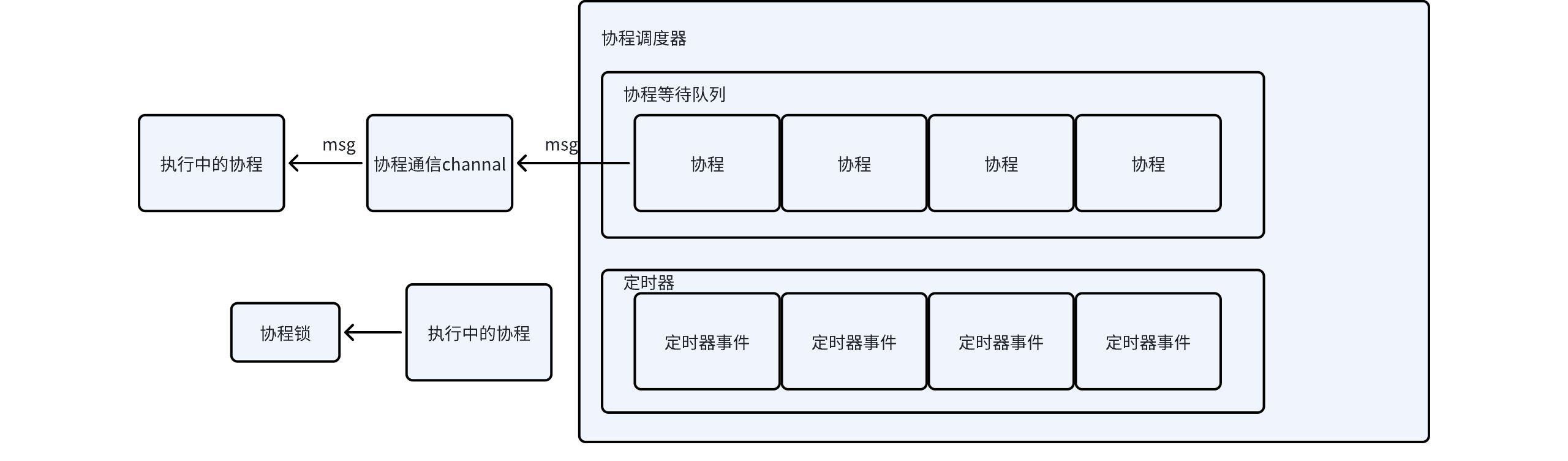
健康检查: 定期检查服务节点的健康状态，确保仅将健康节点纳入负载均衡池，提升系统可靠性。

**4、项目整体架构**



**一、协程库实现**

协程库整体设计如下，包括协程调度器、定时器、协程同步与互斥等。协程调度器从协程等待队列和定时器中调度协程执行。



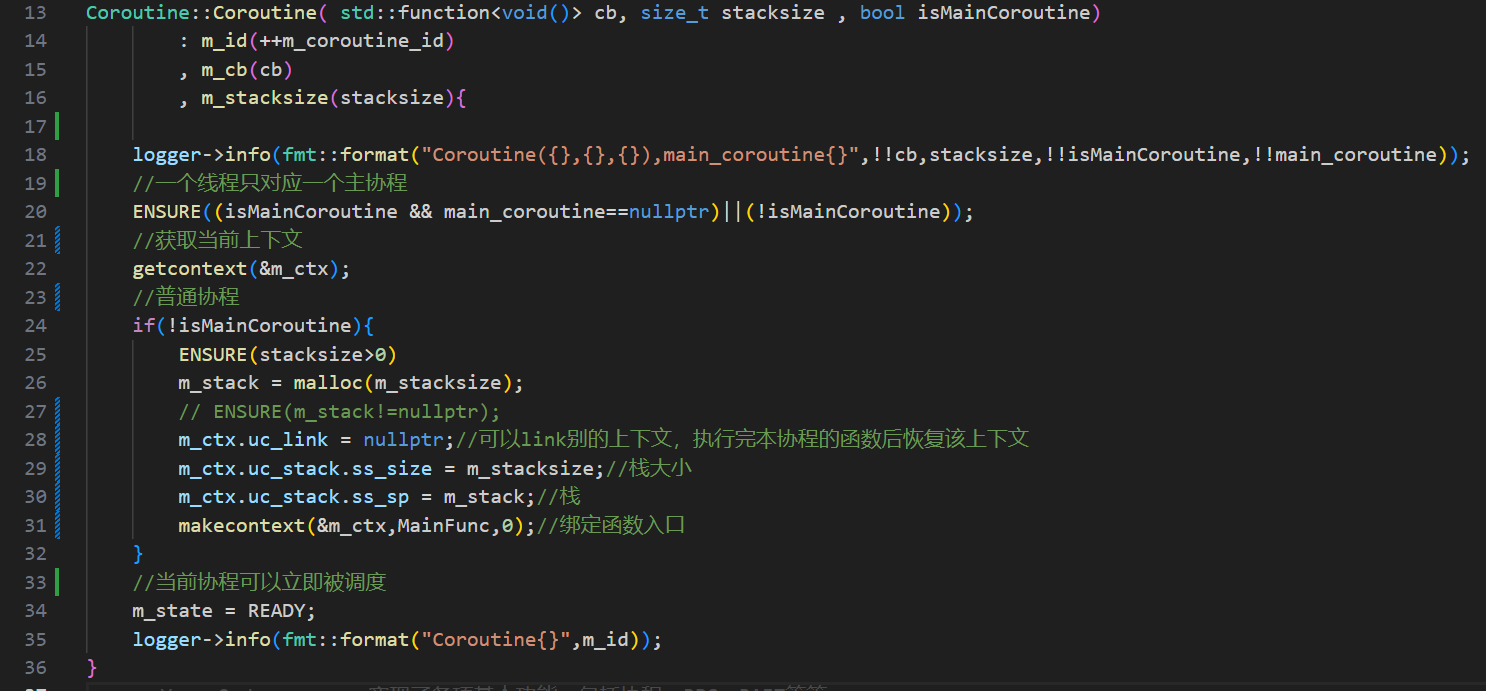
**1、协程实现**

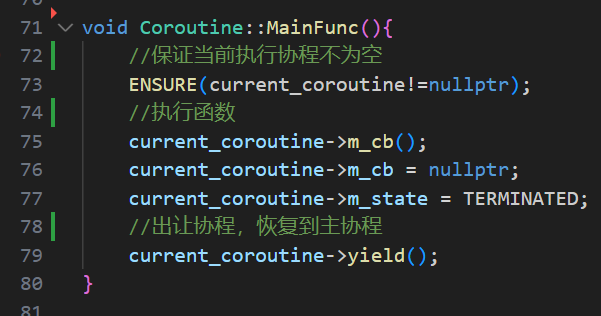
主要封装了linux下的ucontext库，ucontext.h是GNU C库的一个头文件，主要用于用户态下的上下文切换

代码实现在source/coroutine/coroutine.h，source/coroutine/coroutine.cpp

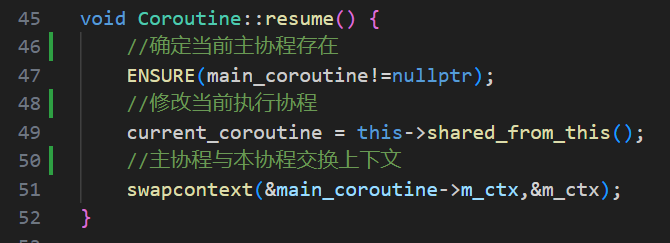
* 创建协程

一个协程只能创建一个主协程（主协程负责调度普通协程执行，工作线程的状态转化为（主协程->普通协程->主协程))

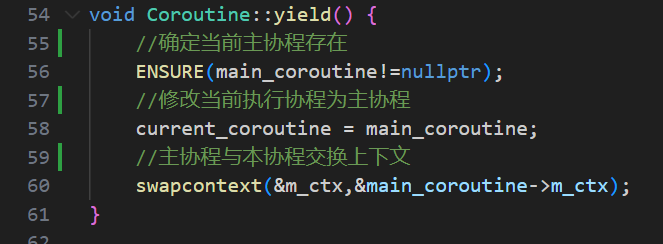




* 执行协程



* 挂起协程



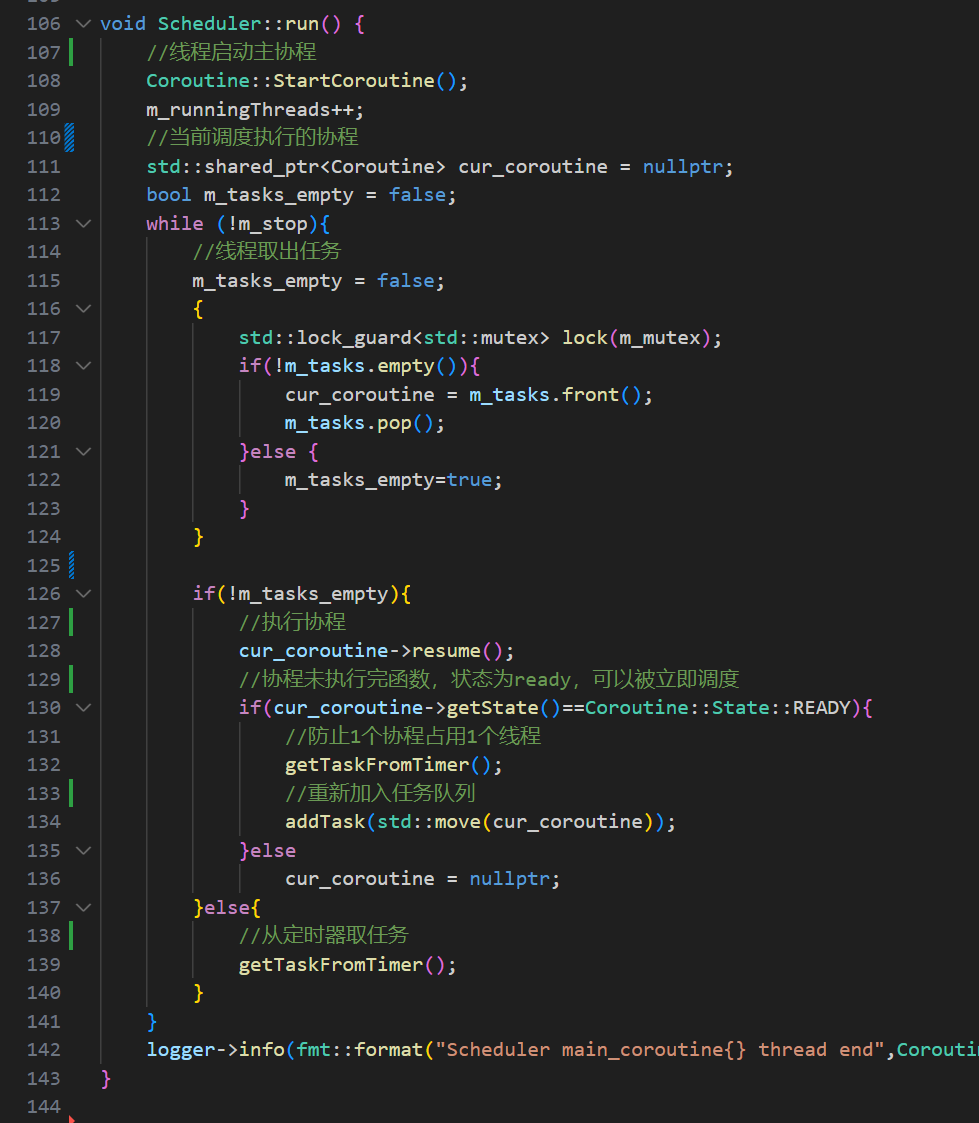
**2、协程调度器实现**

协程调度器的实现逻辑为：采用非对称协程设计，主协程负责普通协程的调度。

调度器线程创建主协程，主协程负责从协程等待队列中提取协程执行，协程执行结束或挂起后恢复主协程，由此循环往复。

代码实现在source/coroutine/scheduler.h，source/coroutine/scheduler.cpp

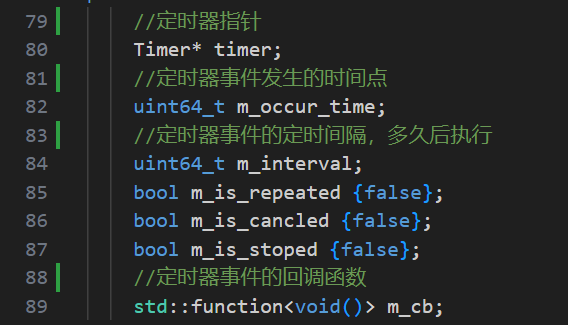
以下为调度器线程执行函数体



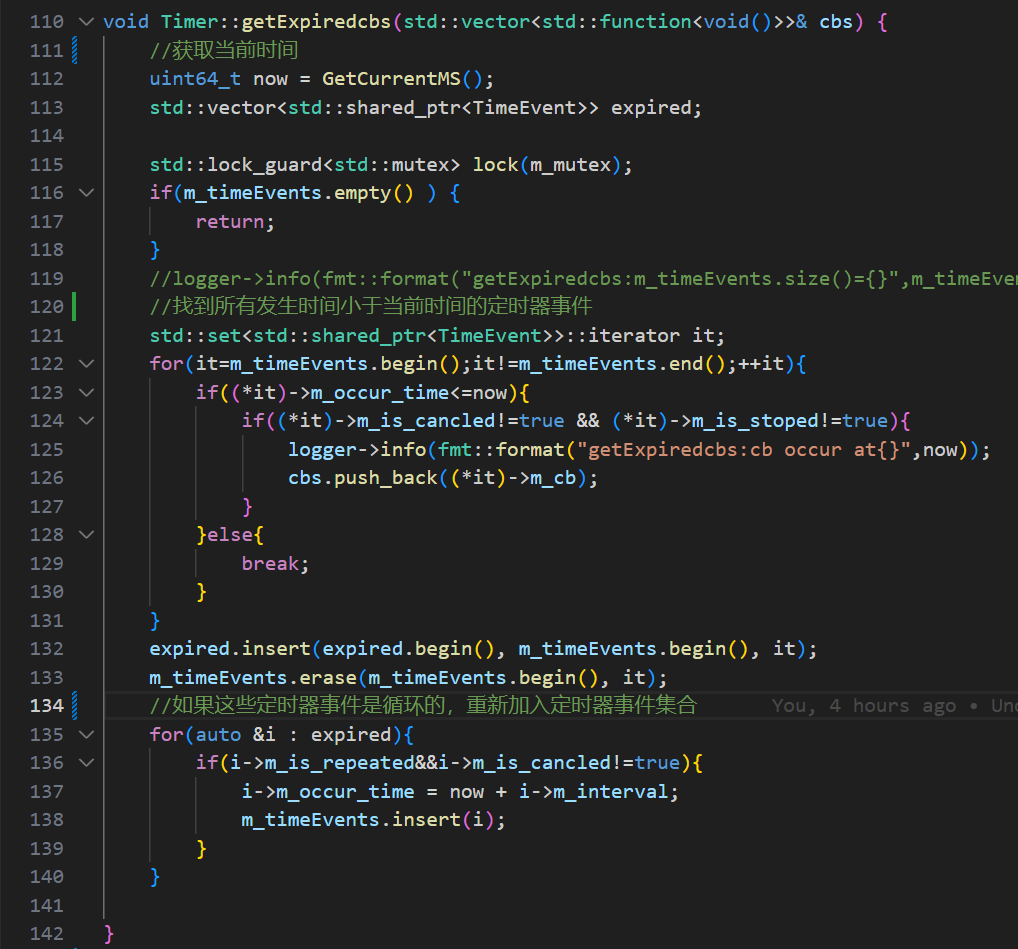
**3、定时器实现**

代码实现在source/coroutine/timer.h

实现了定时器事件类，保存执行的回调函数以及时间等



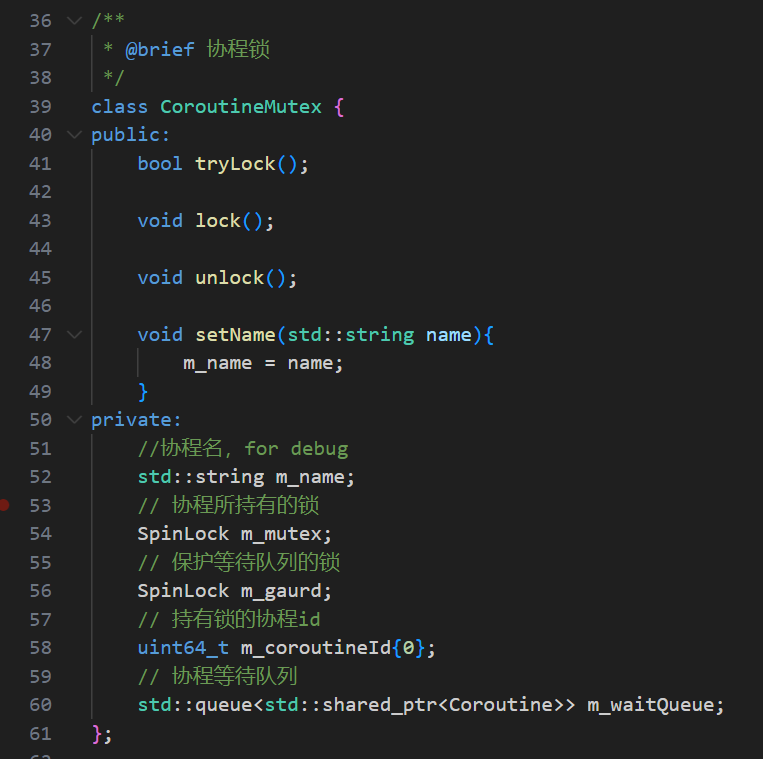
以下为协程调度器从定时器获取任务执行的代码，思路是从已经排好序的定时器事件集合中提取时间戳小于当前系统时间的定时器事件，如果该事件没有被取消或暂停，则将其返回给协程调度器，然后协程调度器再进行调度执行；如果该事件为循环事件，重新将其加入定时器事件集合。



**4、协程同步与互斥**

代码实现在source/coroutine/coroutinesync.h，source/coroutine/coroutinesync.cpp，source/coroutine/coroutinechannal.h

* 协程锁



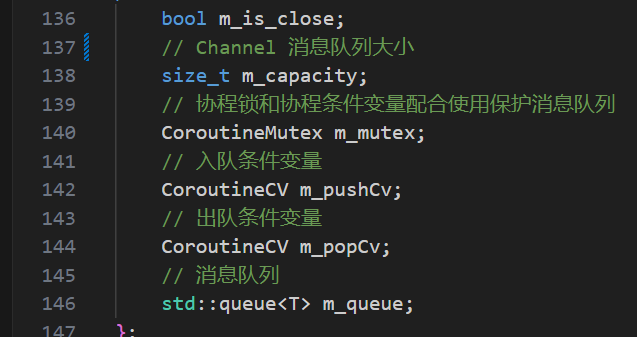
协程锁的核心是协程等待队列，当协程获取锁时，如果获取失败，则将自己加入协程等待队列并挂起；

当协程释放锁时，从协程等待队列中提取一个协程加入协程调度器的协程等待队列，然后协程调度器调度该协程再次获取锁，具体代码不再赘述。

* 协程条件变量

协程条件变量的实现思路和协程锁类似，wait()类似lock()的实现，notify()类似unlock()的实现，核心都是协程等待队列

* 协程通信channal



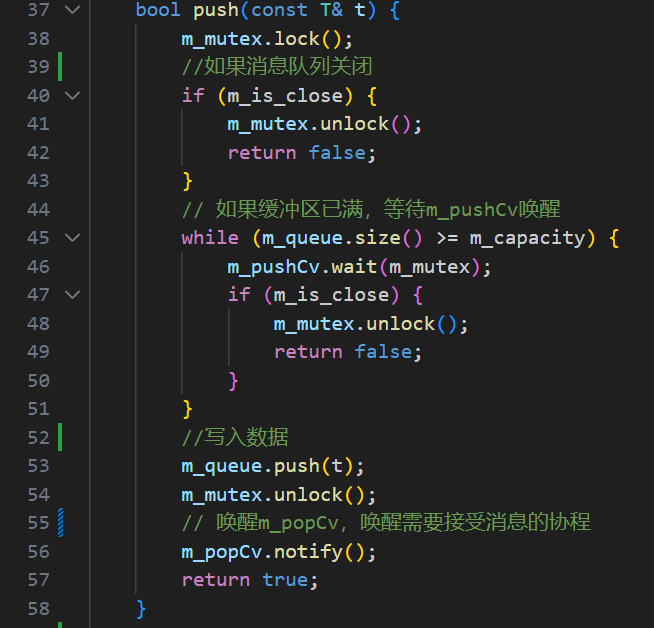
协程通信channal的核心是协程锁和协程条件变量

协程对消息队列进行操作时必须拿到协程锁，考虑到线程1执行协程A，操作消息队列的途中挂起，然后线程1转而执行协程B，然后操作改消息队列，如果锁的粒度是线程级别的话，就会冲突。

协程push时，如果队列已满，就会调用push\_cv.wait，挂起自己，然后等待别的协程调用push\_cv.notify来唤醒它

协程pop时同理

push的具体实现如下



**二、网络模块实现**

**1、序列化器**

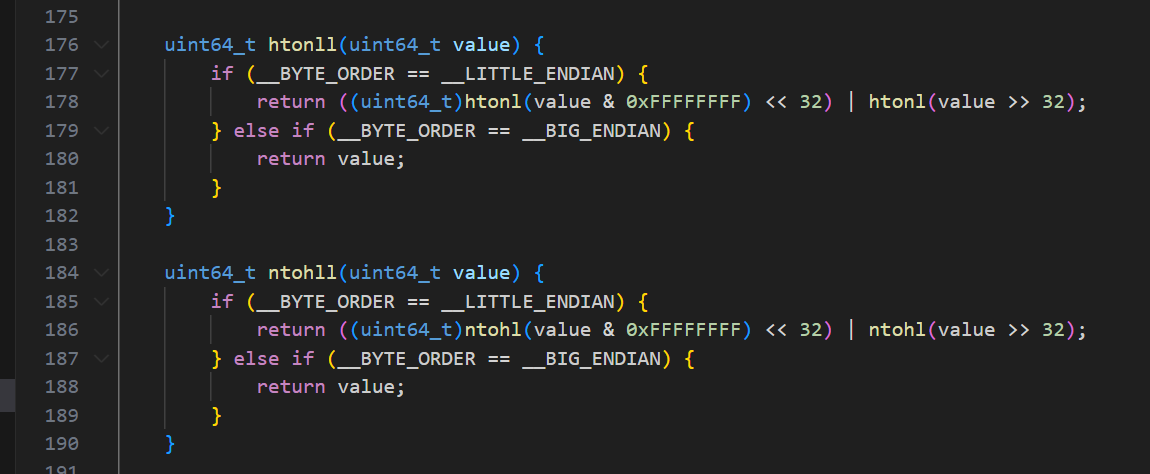
使用模板，实现了对于基本数据类型以及stl容器的序列化与反序列化

序列化时将主机序转化为网络序，反序列化将网络序转化为主机序

以下是部分实现，代码主要在source/util/serializer.h

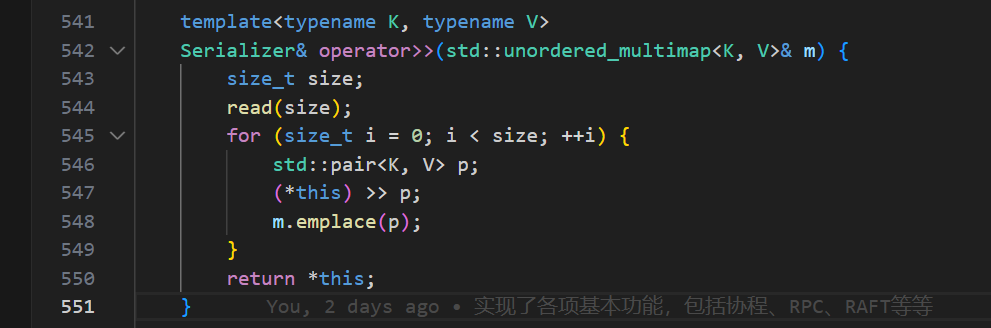
* buffer类

封装了std::vector<char>，写入、读出时转化字节序



* Serializer类

封装了buffer类



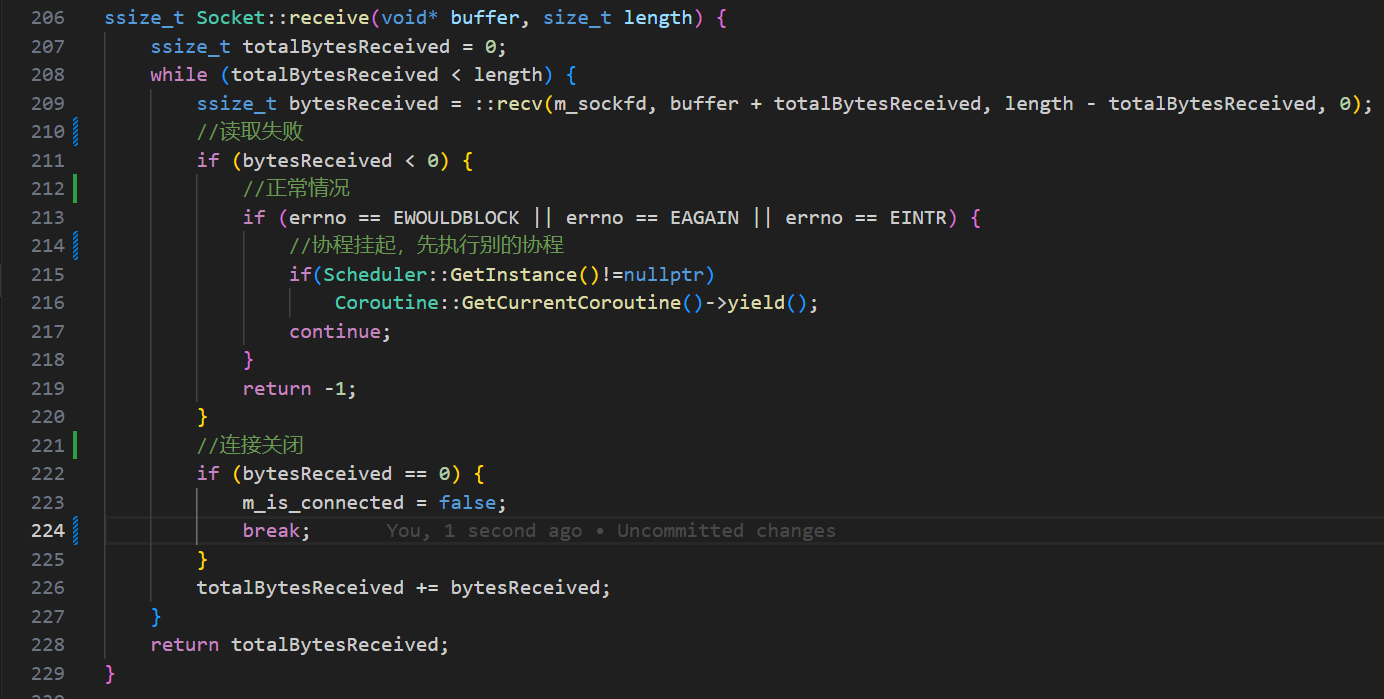


**2、非阻塞socket封装**

使用协程的一方面是为了以同步的方式实现异步编程，便利于后续开发；另一方面是为了提高并发的性能，如果当协程阻塞时，线程会挂起该协程调用另一个协程，减少了线程切换的开销。由此本项目采用非阻塞socket编程，避免协程执行时阻塞线程。

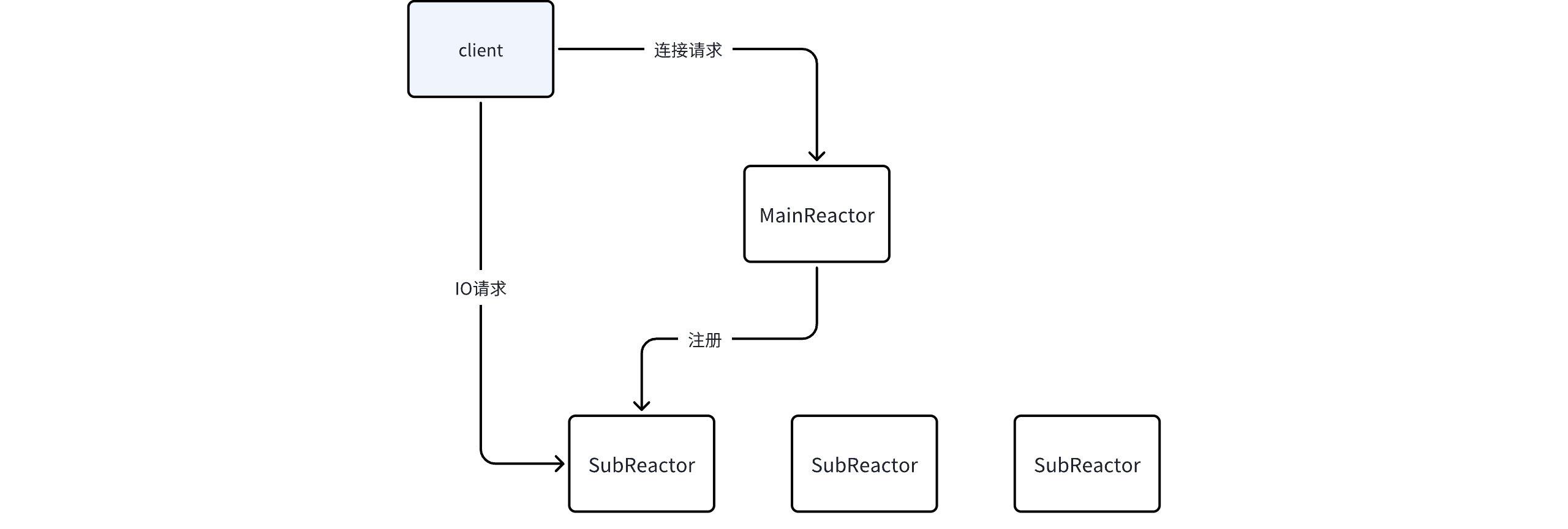
代码主要实现在source/socket/socket.h，source/socket/socket.cpp

例如，receive数据时，使用非阻塞socket，数据读取未成功时，挂起协程，先执行其它协程，避免线程阻塞、资源浪费



**3、TCP服务器**

代码主要实现在source/socket/tcpserver.h，source/socket/tcpserver.cpp

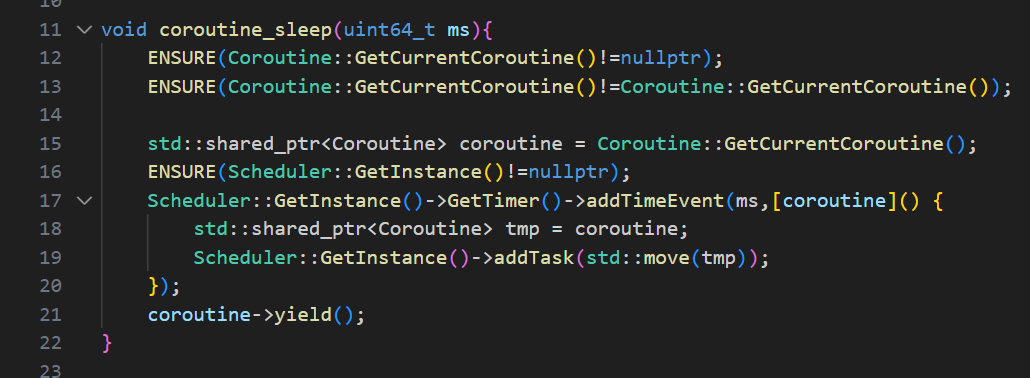


基本实现思想是结合IO多路复用和reactor模式，客户端连接由主协程处理，main协程建立好连接后将socketfd注册到sub协程上，由sub协程处理之后与客户端的数据交互

代码实现思路如下，由于epoll\_wait是阻塞函数，为了防止线程阻塞过久，我们需要设定epoll\_wait的执行时间，即最近的定时器事件发生时间和某个设定值相比较的较小值。

|  |
| --- |
| //mainReactor  while (true) {  int eventCount = epoll\_wait(m\_epollFD, events, MAX\_EVENTS, getLatestTEOccur()); // 等待事件发生  if (eventCount == -1) {  // 打印错误信息  break;   }  for (int i = 0; i < eventCount; ++i) { // 遍历发生的事件  if (events[i].data.fd == m\_listenSocket.getSocketFD()) { // 如果是监听套接字的事件  //accept连接  //将socketfd注册到sub协程上  }  }  }  //subReactor   while (true) {  int eventCount = epoll\_wait(m\_epollFD, events, MAX\_EVENTS, getLatestTEOccur()); // 等待事件发生  if (eventCount == -1) {  // 打印错误信息  break;   }  for (int i = 0; i < eventCount; ++i) { // 遍历发生的事件  auto it = m\_clientSockets.find(events[i].data.fd);  if (it != m\_clientSockets.end()) { // 如果找到客户端套接字  ssize\_t bytesReceived = it->second->receive(data, 4); // 接收数据  //处理数据  }  } |

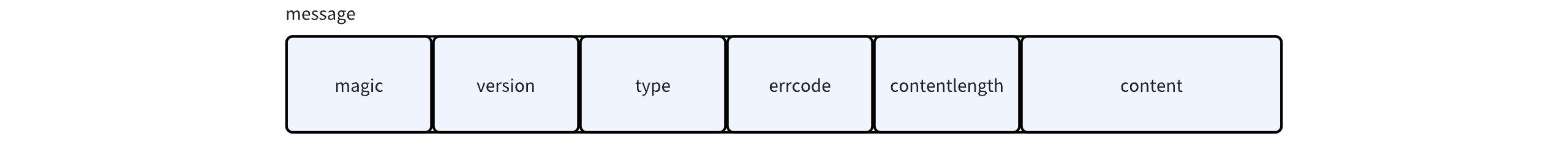
实际上，对于某些阻塞函数，我们可以实现相应的hook函数，使得协程阻塞而线程不阻塞，下面以sleep为例，我们设定一个协程恢复执行的定时器事件，然后将当前协程挂起就可以了。



**三、服务提供者实现**

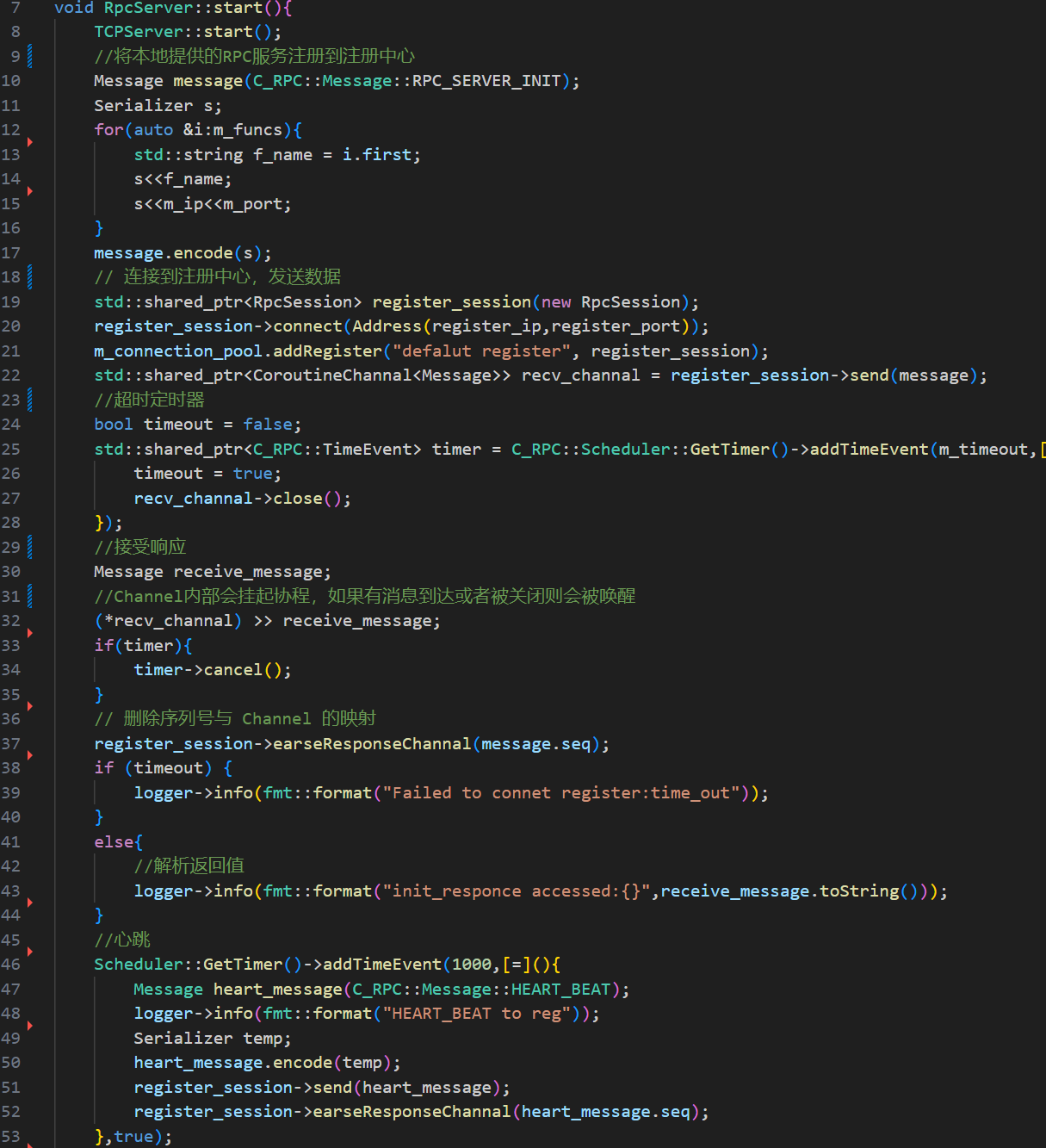
服务提供者接受服务消费者的PRC调用请求，并且返回执行结果

首先，我们要定义1个用于本项目的网络通信协议，如下所示，具体代码实现在source/rpc/rpc\_protocol.h



服务提供者主要代码在source/rpc/rpc\_server

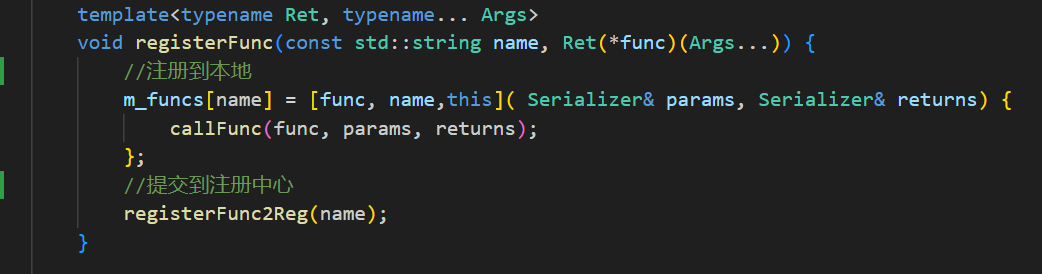
服务提供者初始化的时候绑定本地端口，并且连接注册中心，将本地的需要注册的服务提交给注册中心，然后启动心跳机制，定时发送心跳到注册中心，具体实现如下

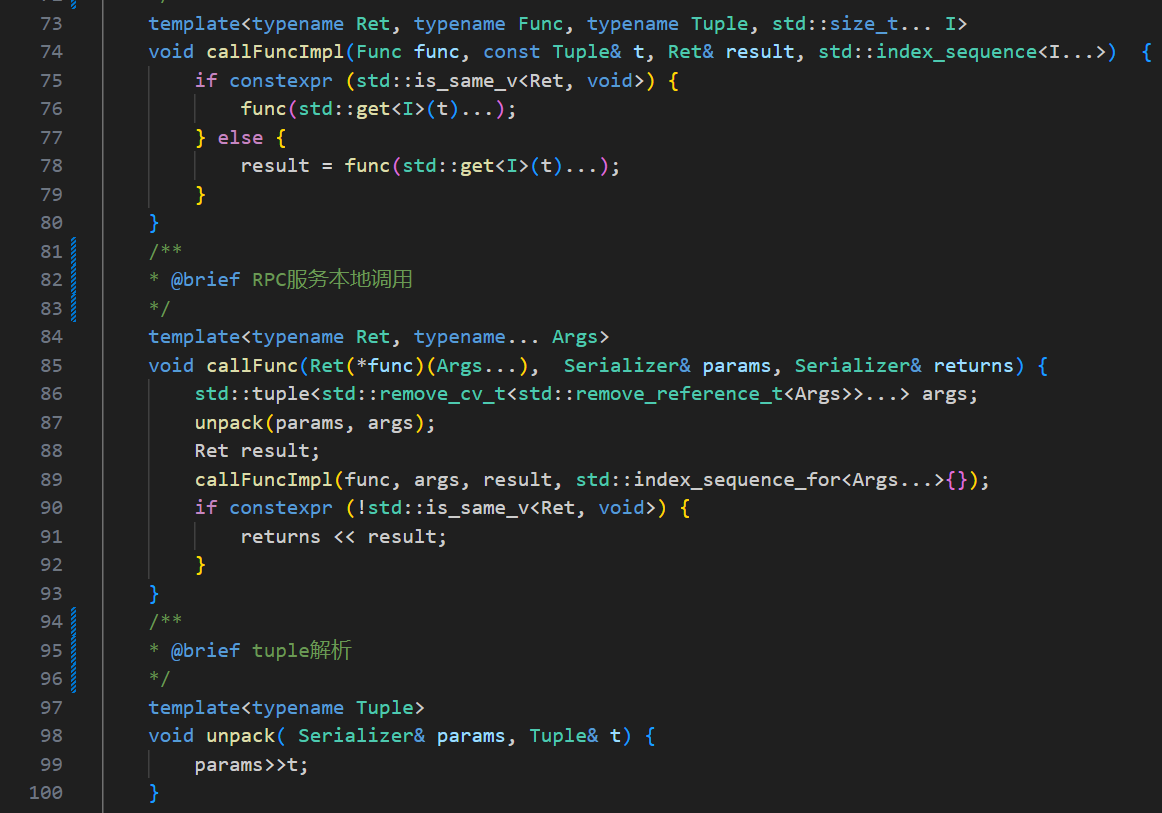


下面介绍服务提供者注册服务到响应RPC调用的过程

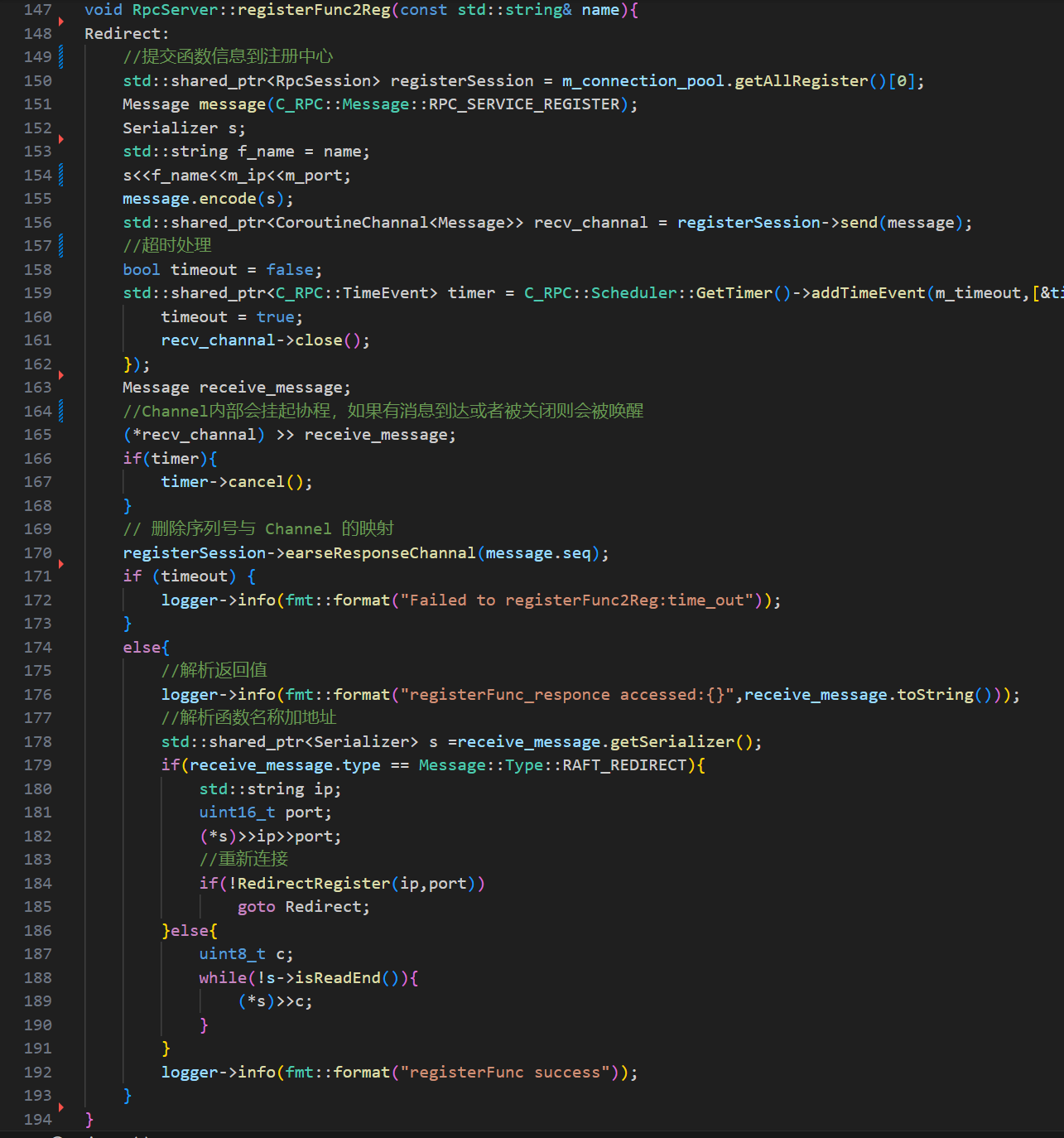
首先，服务提供者注册RPC函数到本地，并提交相关信息到注册中心。

注册到本地时，为了处理的方便，我们会将原来的函数封装成为统一形式的void(Serializer& params, Serializer& returns)函数，使用tuple来处理函数的参数。





然后提交相关信息到注册中心，如果注册中心采用raft集群，则可能会重定向到主节点

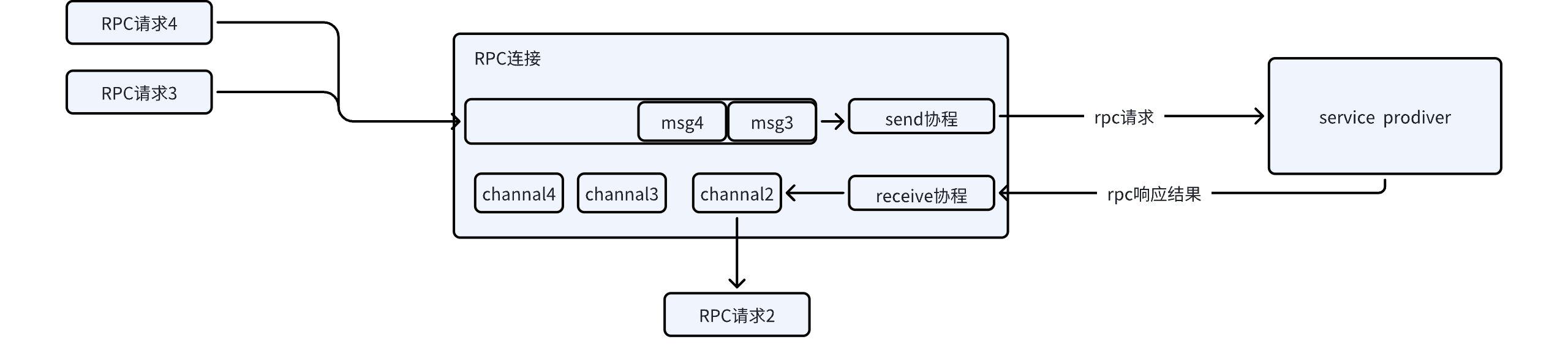


接受RPC请求时，解析RPC调用的函数以及参数，本地执行然后返回结果到客户端即可。

**四、服务消费者实现**

**1、RPCsession**

由于本项目使用连接复用来支持更高的并发，一条连接上可能有多个并发的请求，所以msg包含序列号以识别请求顺序。同时，为了防止socket读写时的并发冲突和可能的等待现象，本项目的每一个连接都实现了一个send协程和一个receive协程来处理网络IO。具体如下：服务消费者进行RPC调用时，将请求提交到对应的RPC连接的send协程，由send协程发送到服务提供者，send协程会提供对应的channal给服务消费者RPC请求协程，当服务提供者返回结果时，服务消费者RPC请求协程就可以从对应的channal处提取。

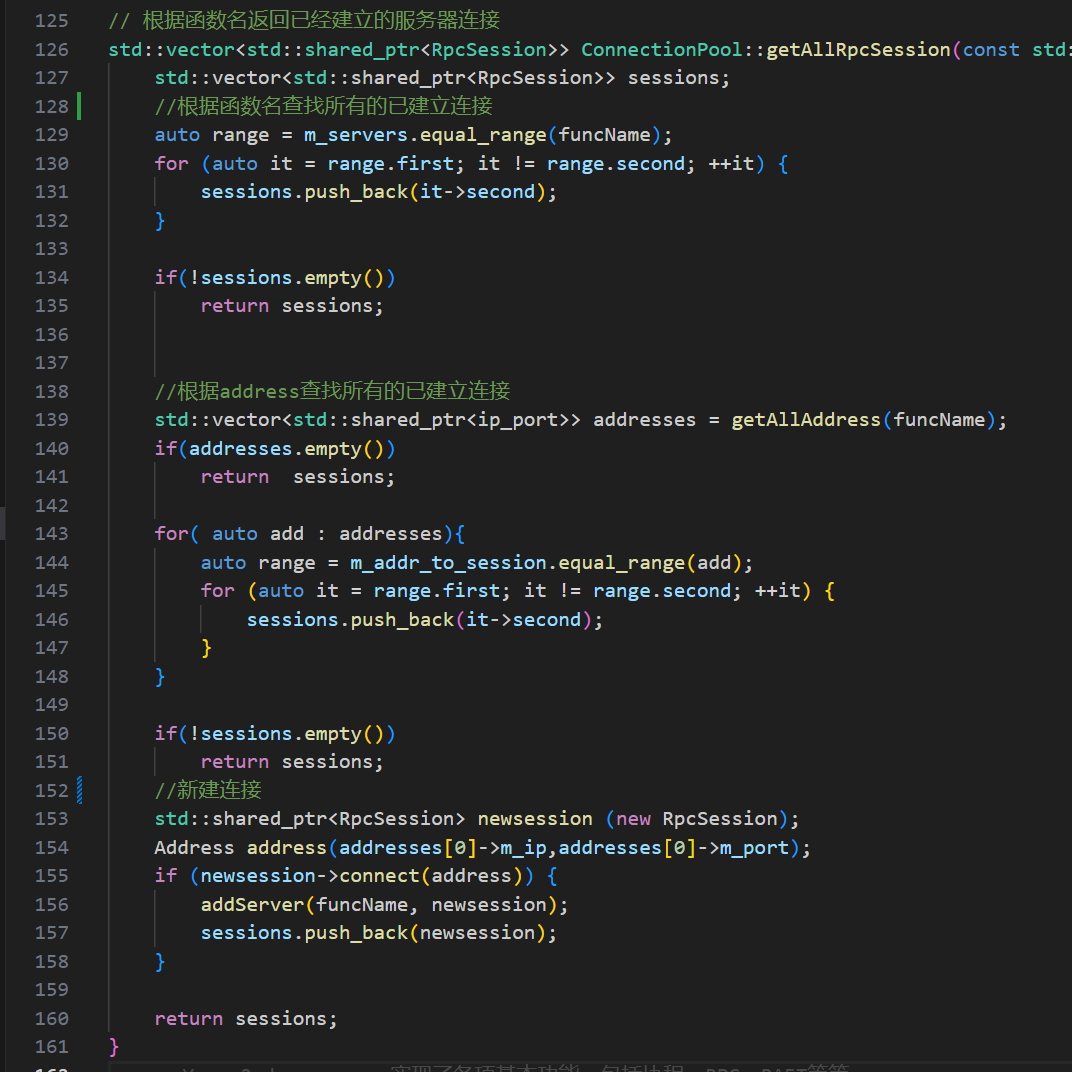


代码主要实现在source/rpc/rpc\_session.h

**2、rpc连接池**

用于管理RPCsession

例如，可以根据函数名称查找相应的rpc连接，如果没有找到对应的RPC连接，那么会根据该函数的远程地址查找对应的RPC连接，如果还是没有找到，就新建一个连接返回给消费者。

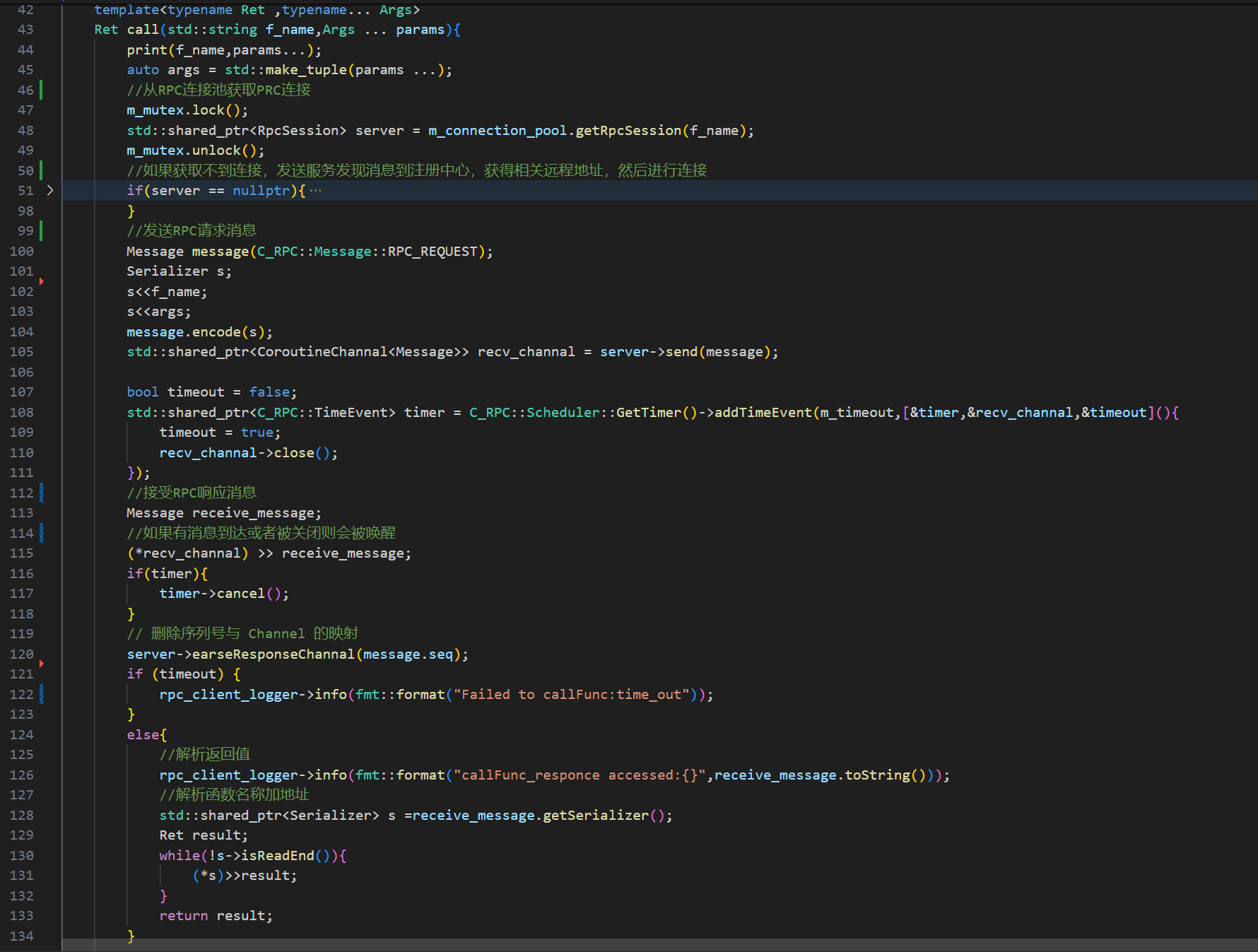


此外，我们得到相关连接后可以根据相关的负载均衡算法挑选合适的连接，比如IP一致性哈希、权重轮询（每次服务发现时注册中心会提供相关的权重）

**3、服务消费者实现**

主要代码实现在source/rpc/rpc\_client

这里主要介绍如何实现RPC调用，首先，通过调用函数名从连接池内取出相应连接，如果没有相应连接则通过服务发现机制从注册中心获取相关远程地址再连接，然后发送RPCrequest消息到服务提供方，通过channal接受消息（把当前协程挂起直至收到消息），然后解析返回值。如果我们不返回函数的调用返回值，而是返回对应的channal，则可以实现异步的RPC调用。

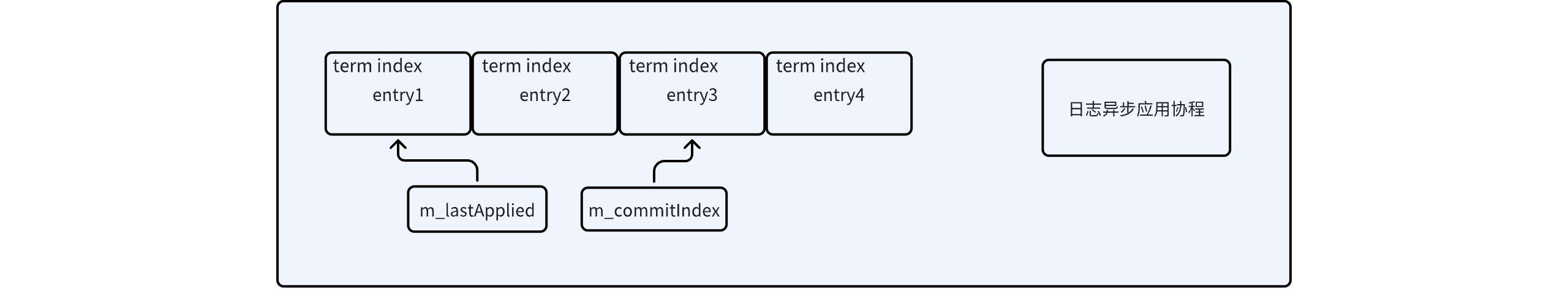


**五、服务注册中心实现**

主要代码实现在source/raft

**1、raft日志类**

具体实现如下图所示



核心为日志异步应用协程和PRC添加日志项函数

日志异步应用协程将index小于m\_commitIndnx的日志应用，并修改m\_lastApplied。当服务提供者注册函数时，只有leader节点日志被应用了，才返回注册结果。

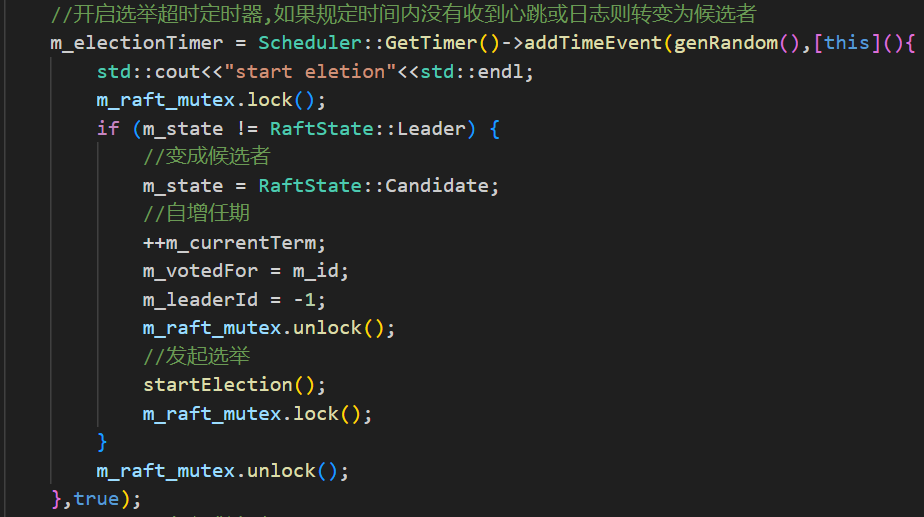
PRC添加日志项函数将尝试通过同步日志的prevLogIndex和prevLogTerm找到对应的日志位置进行添加。

**2、raft节点类**

关于raft实现的文章已经很多了，算法提出者的论文也讲述的比较详细，下面就列举几个关键函数的实现

* raft选举

首先，节点初始化为follower，然后开启一个选举超时定时器，如果规定时间内没有收到leader心跳或日志则转变为candidate，自增任期发起选举

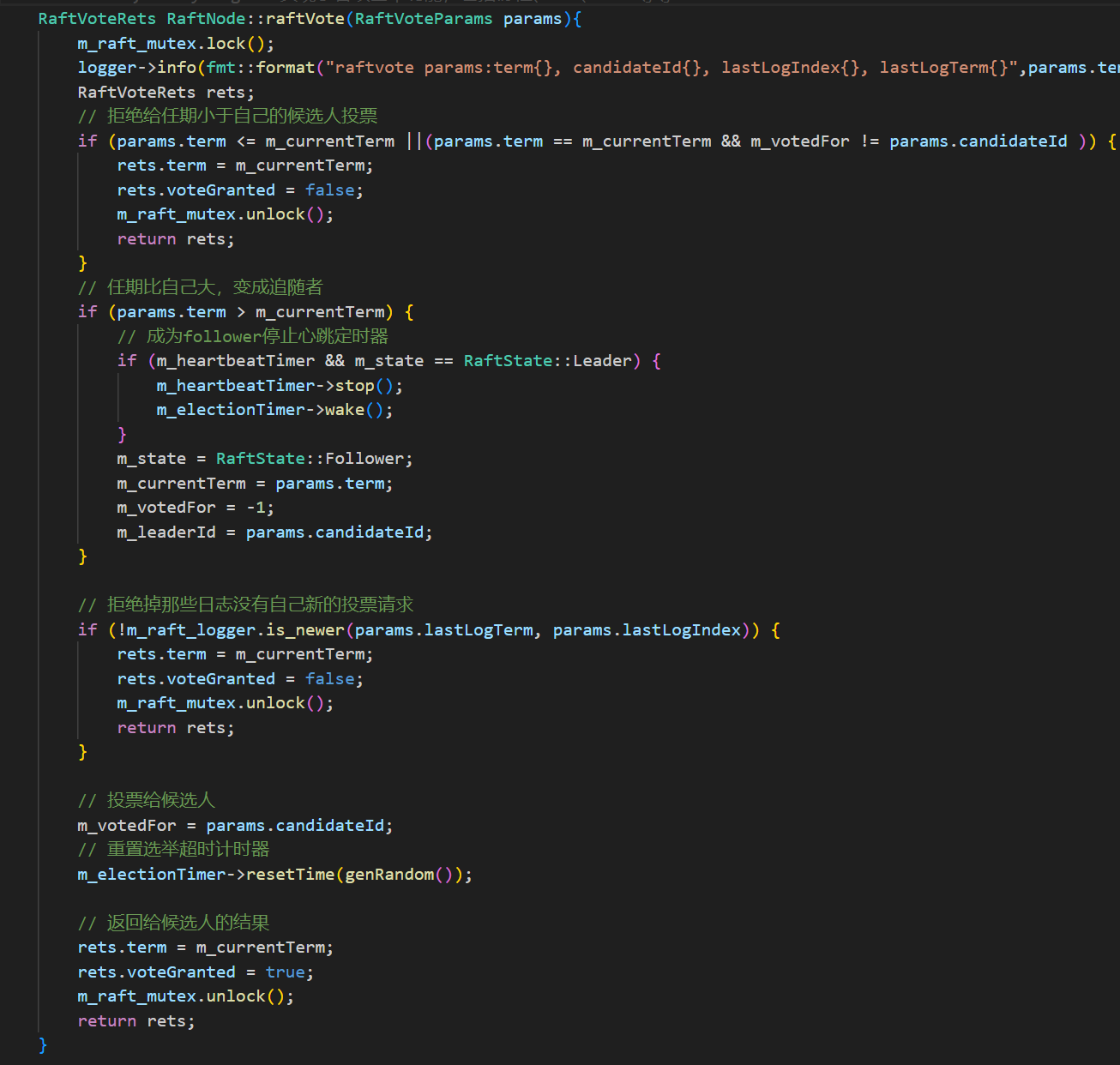


发起选举时，candidate发送当前任期、id、最新的日志任期及索引，使用RPC远程调用，使得其余节点进行投票，具体的RPC调用流程与之前的服务消费者类似。

当candidate接收到其余节点的投票结果时，如果票数超过一半，则成为leader，初始化相关参数，停止选举超时定时器，开启心跳定时，开始同步日志；如果其余节点的任期比自己高，则变成follower，等待leader的心跳或日志。具体实现如下



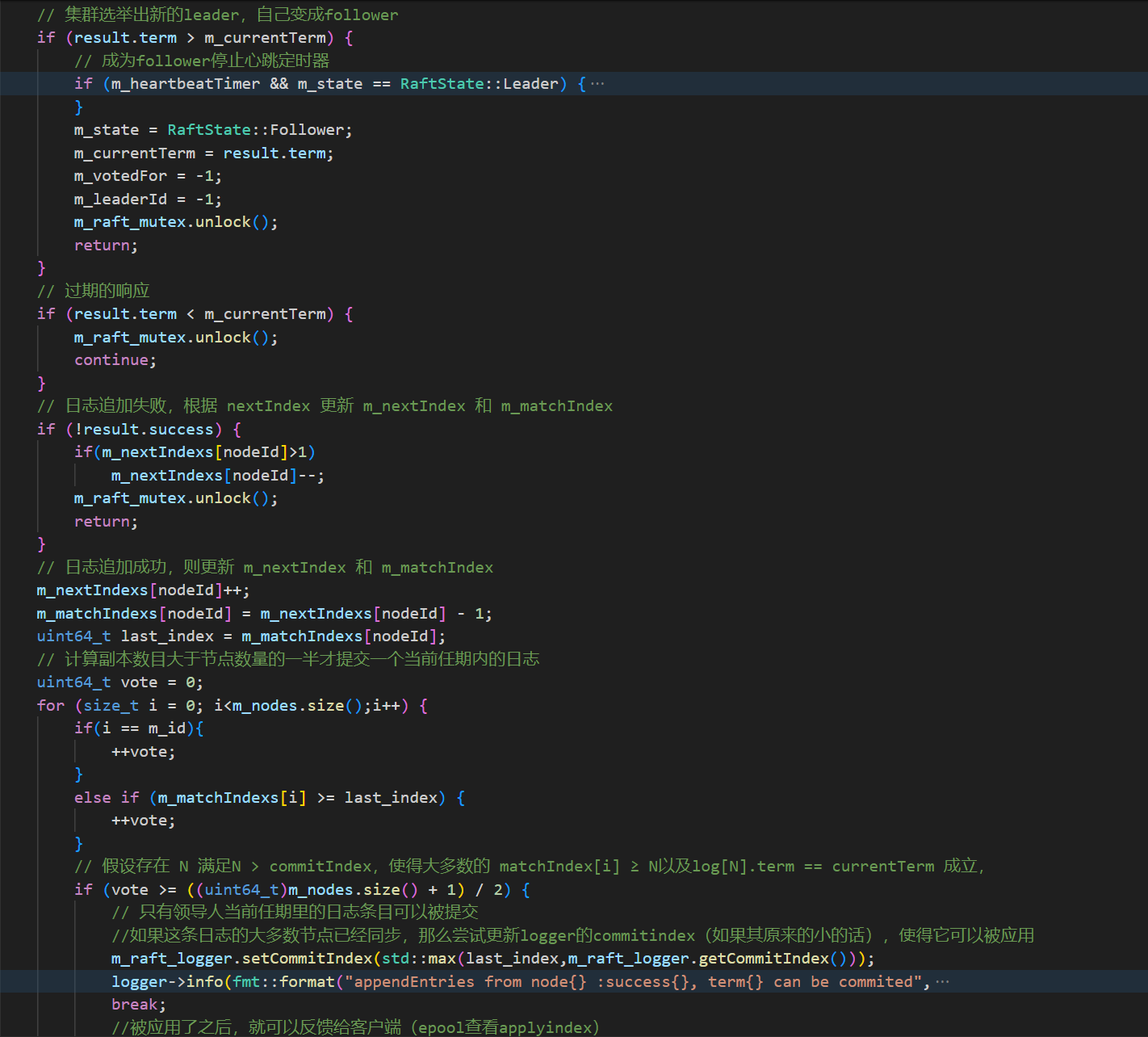
远程节点投票实现如下，如果候选人任期小于自己，投反对票；如果候选人任期比自己大，就变成追随者，改变任期；如果候选人的日志没有自己新，就投反对票，否则投赞成票，重置选举超时计时器。



* 日志同步

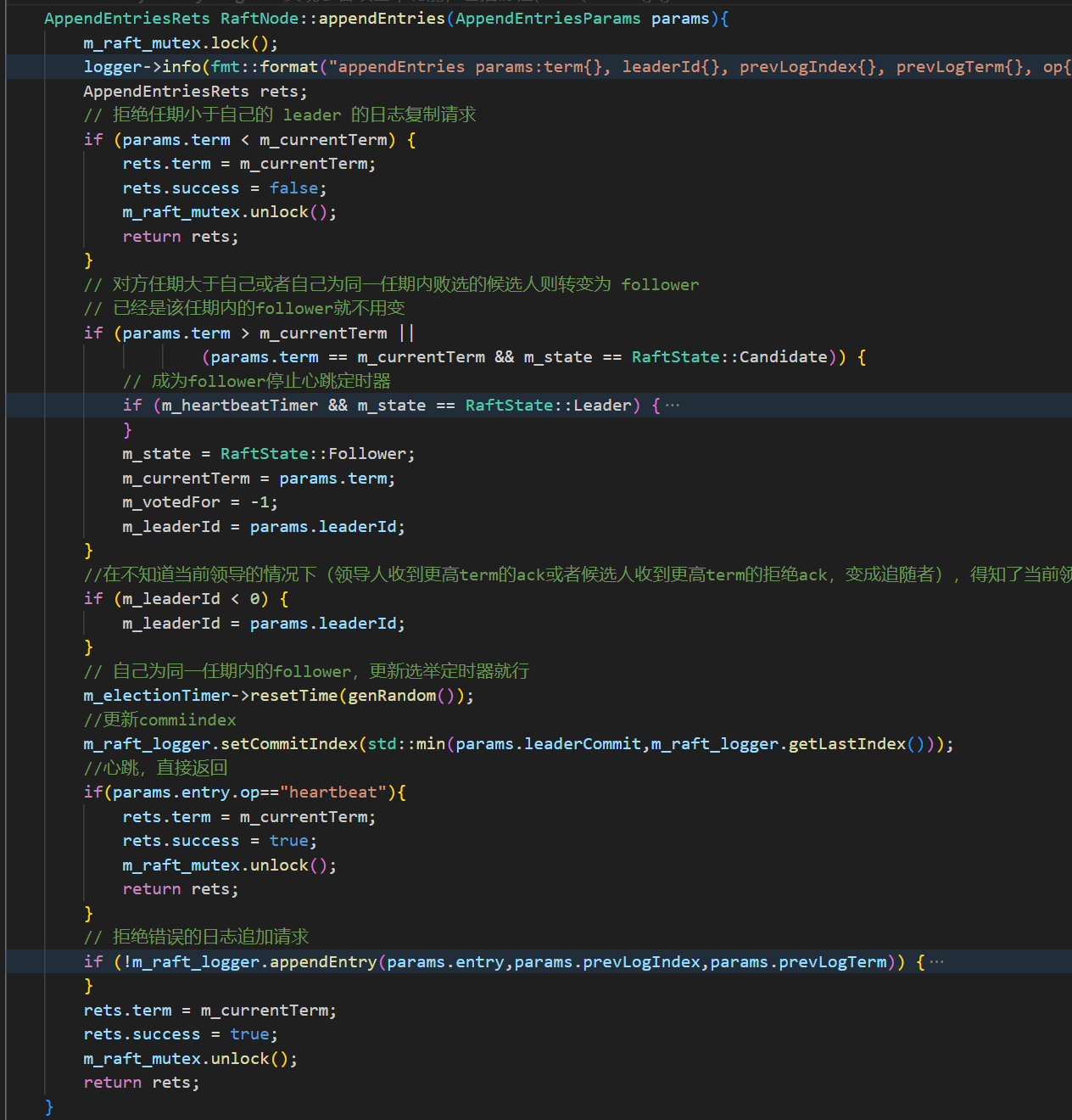
首先，leader尝试发送日志进行同步，首先初始化RPC调用的参数，包括任期、领导人id、领导人commitedindex、前一条日志的任期及索引、日志等。

接收到远程节点的返回值后，如果远程节点的任期比自己大，那么说明已经选举出了新的leader，那么自己变成追随者，等待新的leader发送消息即可；如果日志同步失败，那么尝试先同步更老的日志给对方；如果同步成功，则更新下一条发给该远程节点的日志索引，如果半数以上node同步成功，则将日志提交应用，然后告诉客户端执行成功。下一轮心跳或日志会通知所有node，这条日志可以提交应用了。



远程节点同步日志实现如下，

如果leader任期小于自己，拒绝同步；如果对方任期大于自己或者为本任期选举产生的新的leader，那么自己成为follower；由于接受到了有效消息，那么重置选举超时定时器，并且更新commitedIndex；如果消息为心跳，不同步日志；如果为日志，那么判断是否可以同步（可能为过于新的日志，但是自己还没有老的日志）。

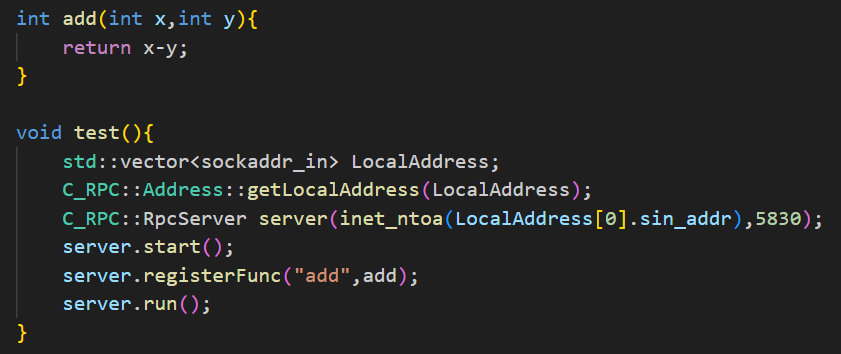


**3、服务注册中心功能**

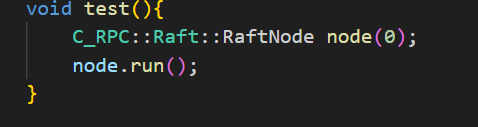
继承TCPserver类，类似服务提供者的实现，建立连接，接受消息，处理消息，返回结果，不再赘述。

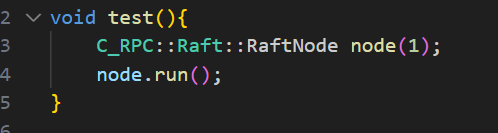
**六、RPC使用**

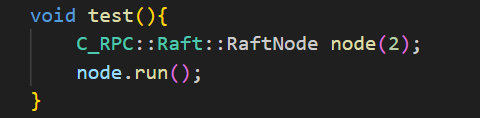
1、服务提供者



2、注册中心集群







3、服务消费者

