

电子科技大学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

学士学位论文

BACHELOR THESIS



论文题目 基于 Thrift 框架的调用链信息埋入的设计与实现

学 院 信息与软件工程学院
专 业 软件工程（数字动漫）
学 号 2018091606029
作 者 姓 名 史瑞琪
指 导 教 师 白忠健

摘 要

RPC (Remote Procedure Call) 即远程过程调用，是一个会话层协议，在工业界被广泛应用。在工业中，RPC 调用结构复杂，一个 RPC 请求会触发大量连锁 RPC 调用，调用链中信息埋点的方式与效率十分重要。本论文将通过重写 THRIFT 框架的二进制传输协议以及编写统一的客户端类和服务端类等途径，提供将调用链信息的编码与解码方法，使调用链信息可以随 RPC 请求携带的信息在客户端和服务端间传递；并在客户端或服务端的本地基于线程调用期完成对 RPC 调用信息的埋点，提供查看接口，令使用 THRIFT 框架构建 RPC 服务的开发者可以清晰地查询并回溯 RPC 调用链信息；同时，在调用链的信息传递中添加对调用方身份和资格的鉴权以及对信息完整性的 CRC 校验，当调用发生错误时，可以对异常进行处理，自动进行回滚，并能够对错误信息进行记录。

通过以上工作，可以使 THRIFT 框架中 RPC 调用链信息的埋入在框架层面实现，减少业务方的参与，并减少使用追踪 RPC 调用链服务的消耗。

关键词：RPC 调用链，信息埋入，THRIFT 框架，错误回溯，CRC 校验

ABSTRACT

Remote Procedure Call (RPC), a session layer protocol, is widely used in the industry. In industry, RPC call structure is complex, one RPC request will trigger a large number of chain RPC calls, and the way and efficiency of information burying point in the call chain are very important. By rewriting the binary transfer protocol of THRIFT framework and writing uniform client class and server class, this thesis provides the encoding and decoding method of calling chain information, so that the calling chain information can be transmitted between client and server along with the information carried by RPC request. In the local thread-based call period of client or server, the buried point of RPC call information is completed, and an interface for viewing is provided, so that the developers who build RPC services using THRIFT framework can clearly query and trace the RPC call chain information. At the same time, the authentication of the caller's identity and qualification and CRC check of information integrity are added to the information transmission of the call chain. When errors occur in the call, exceptions can be processed, automatic rollback can be performed, and error information can be recorded.

Through the above work, the embedding of RPC call chain information in THRIFT framework can be realized at the framework level, reducing the participation of business parties and reducing the consumption of RPC call chain tracing service.

Keywords: RPC call chain, Information embedding, THRIFT framework, Error backtracking, CRC check

目 录

第一章 绪论	1
1. 1 研究现状及发展态势	1
1. 1. 1 研究现状	1
1. 1. 2 现有问题	1
1. 2 选题依据及意义	2
1. 2. 1 选题依据	2
1. 2. 2 选题意义	2
1. 3 复杂工程问题与拟完成目标	2
1. 3. 1 拟解决复杂工程问题	2
1. 3. 2 可行性论述	3
1. 3. 3 拟完成目标	3
1. 3. 4 论文创新点	4
第二章 开发环境与关键技术	5
2. 1 技术选型	5
2. 2 开发环境搭建	5
2. 2. 1 Linux 环境搭建与远程开发工具配置	5
2. 2. 2 Thrift 环境搭建	9
2. 3 RPC 调用应用原理	9
2. 3. 1 远程过程调用信息传递过程	9
2. 3. 2 Thrift 框架支持的 RPC 服务的优势	10
2. 4 Thrift 应用原理	10
2. 4. 1 Thrift 架构	10
2. 4. 2 Thrift 传输协议	11
2. 5 线程应用原理	12
2. 5. 1 线程存储期	12
2. 5. 2 线程阻塞	13
2. 6 Cyclic Redundancy Check 应用原理	13
2. 7 SPDLogSink 组件应用原理	15
第三章 设计与实现	16
3. 1 设计整体框架	16

3.1.1 客户端设计框架	16
3.1.2 服务端设计框架	17
3.1.3 调用链设计框架	18
3.1.4 鉴权系统设计框架	19
3.2 设计数据结构	20
3.2.1 信息头结构 CommonHeader	20
3.2.2 线程存储期数据存储块 Block	20
3.2.3 数据存储期结构 ThreadContext 类	22
3.2.4 服务信息存储结构 ServerInfo 类	23
3.2.5 线程存储期读写辅助存储块 RpcContext 类	23
3.3 新协议 HeaderBinaryProtocol 的实现	24
3.3.1 官方协议 TBinaryProtocol 的研究	24
3.3.2 协议的调用流程	27
3.3.3 CRC 校验接口编写	29
3.3.4 信息头的写入	30
3.3.5 CRC 校验码的写入	32
3.3.6 信息头的读出	32
3.3.7 CRC 校验码的校验	33
3.4 鉴权的实现	35
3.4.1 密钥编码解码	35
3.4.2 设置白名单	37
3.4.3 鉴权	37
3.5 客户端的实现	38
3.5.1 客户端类继承结构	38
3.5.2 客户端在主函数中的实例化	39
3.5.3 客户端在 EchoClient 层的实现	41
3.5.4 客户端在 ThriftClient 层的实现	43
3.5.5 客户端在基类 Client 层的定义	47
3.6 服务端的实现	48
3.6.1 服务端在主函数中的实例化	48
3.6.2 服务处理类 EchoServiceHandler 的实现	52
第四章 系统测试	53
4.1 Thrift 测试 demo 的编写	53

4.2 调用链的设计与实现	57
4.3 正常流程测试结果	59
4.4 异常流程测试	61
4.4.1 Token 鉴权失败回滚.....	61
4.4.2 No more data to read 错误模拟	62
4.4.3 Broken Pipe 错误模拟.....	62
4.5 测试结论	63
第五章 结论	64
5.1 项目总结	64
5.1.1 完成情况	64
5.1.2 未来展望	64
5.2 对工程伦理的理解	65
5.3 对职业素养的理解	66
致 谢	67
参考文献	68
外文资料原文	69
外文资料译文	72

第一章 绪论

RPC (Remote Procedure Call) 即远程过程调用，是一个会话层协议，在工业界被广泛应用。工业中，RPC 调用结构复杂，一个 RPC 请求会触发大量连锁 RPC 调用，调用链中信息埋点的方式与效率十分重要。

1.1 研究现状及发展态势

1.1.1 研究现状

RPC 调用结构复杂的情况下，一个 RPC 请求会触发大量连锁 RPC 调用。对于贯穿 RPC 调用链路的信息使用业务逻辑传输，会使信息在大量的调用中被入参，然后再传递到下一个接口中。在这个过程中，可能会造成以下消耗：

- 1) 降低了传输效率
- 2) 增加了出错可能性

同时，在递归的调用链中，需要进行埋点记录，以此增加调用链的可回溯性以及对于异常的可控性。现有的埋点服务及其缺点如下：

- 1) cat：基于 Java 开发的实时应用监控平台。基于代码埋点实现监控，对代码的干扰很大，集成成本高，风险大。
- 2) Zipkin：一个分布式的跟踪系统，集成方便，但是功能简单。

1.1.2 现有问题

针对调用链中信息的埋入问题，如今的解决与发展方向有以下问题：

- 1) 重复造轮子

工业对调用链信息埋点的处理具有差异性，为了避免业务层感知，不同公司会重复开发轮子，造成人员浪费。

- 2) 消耗巨大

以某互联网公司为例，为进行调用链信息跟踪，公司内部面向所有服务提供了一个统一的标准化服务，所有的 RPC 调用链都会依赖这个服务进行记录与鉴权。此服务减少了业务代码层对调用链的感知，但是由于公司规模巨大，本服务的 QPS(Queries per second) 已经达到了亿的数量级，消耗巨大。同时，单独开发服务进行调用链的信息埋入，会增加系统整体的复杂性，降低原服务的稳定性。

- 3) 代码侵入性强

现有的 RPC 调用链追踪系统一般作为业务组件存在，或多或少会入侵其他业务系统，依赖于应用开发者的主动配合，增加开发人员的负担，同时可能因为开发者本身的错误代码导致与负担。

1.2 选题依据及意义

1.2.1 选题依据

依据上文对现有解决方式的分析，可知当前在应用层另起服务的方式无法高效地在 RPC 调用链中进行信息传递以及日志记录。Thrift 是一个使用较多的开源 RPC 框架，因此可以通过分析其的代码逻辑，了解客户端信息封装方式以及服务端信息处理方式，学习 RPC 框架底层原理，并且在此基础上对架构层信息读写的部分进行重写，可以达到更高效建立信息埋点的目的。

1.2.2 选题意义

本选题会尝试在 Thrift 架构层进行改写，在架构层对信息的读写时完成调用链信息的埋入，并基于线程存储期存储调用链信息，提高信息在 RPC 调用链中存储传播的效率和安全性。同时，可以减少服务器开销与重复造轮子的人力浪费。

1.3 复杂工程问题与拟完成目标

1.3.1 拟解决复杂工程问题

1) 定位改写位置

研究现有 Thrift 源码结构，了解客户端信息封装方式以及服务端处理方式，学习 RPC 框架底层原理。定位改动量小，影响逻辑少，冗余小的改写位置。

2) 数据结构设计

选择适当的埋入信息的数据结构，减少调用埋点的性能消耗。

3) 确定信息埋入的实现方式

在之前确定的位置使用选择的技术模型进行代码编写。

4) 保证设计的可扩展性

除固定类型信息外，还应支持其他信息的埋入。不同类型的参数信息均可通过改进后的 Thrift 框架进行埋入。

5) 保证信息在客户端与服务端之间的可靠传递

令调用链信息可以在客户端与服务端之间进行传输，并进行鉴权和信息完整性检测。

6) 保证调用链异常时的回滚

当调用链异常，需要记录出错信息，并且回滚，重新进行调用。

1.3.2 可行性论述

1. 社会方面

本项目可以增加基于 RPC 调用的分布式系统的稳定性。当前社会大部分的服务都是基于 RPC 调用的，很多服务的稳定性会影响社会的稳定，例如核酸-安全码维护系统，医保结算系统，交通控制计算系统等。提高这些重要系统的稳定性和安全性，对社会稳定发展具有重大意义。

2. 经济方面

工业对调用链信息埋点的处理具有差异性，为了避免业务层感知，不同公司会重复开发轮子，造成人员浪费，因此本项目可以减少人力成本的浪费。

同时，公司内部面向所有服务提供了一个统一的标准化服务，所有的 RPC 调用链都会依赖这个服务进行记录与鉴权。当公司规模变大，服务器和电力的经济成本的消耗巨大。

其次，单独开发服务进行调用链的信息埋入，会增加系统整体的复杂性，降低原服务的稳定性，对系统稳定的维护也需要有相应的人力成本支出。

综上所述，本项目对节省经济成本有很大的意义。

3. 技术方面

本项目是基于开源项目 Thrift 框架，对于其修改不造成对其公司的侵权。相反，本项目也将以开源的方式进行分发，有助于促进开源运动的发展。开源对于系统的发展具有很重要的意义，即我可以从 Thrift 源码中学习到很多技术，同时我也可以对其进行改进，推动技术的发展。

1.3.3 拟完成目标

能够正确地在一个模拟 RPC 调用链中完成信息的埋入，打出正确、清晰、易于分析的调用链日志。

能够做到业务无感知的调用链信息埋入，减少业务方的必须操作的同时，以免业务方出错导致整个调用链不可靠。

可以做到一定的自我修复能力，能够在 Thrift 中途请求出错时，自动回滚调用链并打出相应的错误日志的同时，不妨碍正常的调用进行。

1.3.4 论文创新点

本课题是从工业界当前需求出发进行思考，总结出现有的调用链中数据冗余传输，调用链跟踪困难，异常难以记录等问题。与现有的在 RPC 调用层上添加一层数据处理层的思路不同，本设计将从 Thrift 本身进行改动，通过对现有框架进行学习和创新，达到优化和解决问题的目的。将埋点设置在调用链的请求节点中，通过线程上下文进行调用链追踪，对调用链异常进行处理，便于回滚与重新发起请求。

第二章 开发环境与关键技术

2.1 技术选型

1. 选择 RPC 框架

考虑到 Thrift 是开源的，用途广泛，而且是企业级的，我们选择 Thrift 作为我们的 RPC 框架。Thrift 是一个可扩展的跨语言服务开发框架。作为一个强大的 RPC 框架，它可以满足企业对高并发的需求。同时，Thrift 的编程模型比较简单，也可以提高服务开发的效率。因此，Thrift 被选为改进的对象。

2. 选择语言

Thrift 框架语言为 C++，选择 C++ 为主要语言。作为系统语言，C++ 具有很高的灵活性，提供了一系列良好的内置库和计算机底层进程的控制方式。语言本身可以很好适应服务器部署环境。

3. 系统选型

考虑到 RPC 服务在工业中一般会被部署到服务器上，现有的大部分服务器集群均为 Linux 系统。同时，由于 C++ 的特殊性，尽管 Unix/Windows 系统中有相同的库，其用法和表现都与传统服务器系统 Linux 有一定的差异。为了后续的服务应用，服务稳定性和减少维护成本，本项目直接在 Linux 平台进行开发。

4. 总体思路

- 1) 在服务本地使用线程存储期，将信息写入线程，在服务线程中维护信息。
- 2) 在客户端/服务端读写信息时加入对信息头的读写，随请求传递调用链信息。
- 3) 再服务端和客户端添加虚拟层，控制异常处理，在异常出现时进行记录回滚，中断本线程调用链并重新建立线程。
- 4) 通过鉴权保证连接安全，通过信息校验保证数据传输完整性。

2.2 开发环境搭建

2.2.1 Linux 环境搭建与远程开发工具配置

由于本次课题与 RPC 框架 Thrift 有关，本框架经常被部署于 Linux 服务器上，因此我们需要搭建 Linux 开发环境以便我们后续进行实验及开发。

Linux 是一个免费和开源的类 UNIX 操作系统。通常，Linux 被打包为 PC 和服务器的 Linux 发行版。在本实验中，使用的是 Debian 版本。

在本实验中，我们将使用腾讯云服务器并通过 SSH 登录。Secure Shell (SSH) 是一种网络协议，允许通过安全连接在两台计算机之间交换数据。加密可确保数据的机密性和完整性。SSH^[1] 使用公钥加密对远程主机进行身份验证，并允许远程主机对用户进行身份验证（如有必要）。在本机生成公钥，存储在`~/.ssh/id_rsa.pub`，生成公钥指令如代码 2-1 所示。

代码 2-1 生成公钥指令

```
$ ssh-keygen -t rsa
```

登陆服务器后，将生成的公钥存储到服务器的`~/.ssh/authorized_keys`。更改 ssh 配置，将 `PubkeyAuthentication` 中的 no 改为 yes，允许公钥登陆。完成后可通过公钥登陆服务器，进行开发。登陆服务器 Linux 开发环境终端如图 2-1 所示

```
Last login: Mon Apr 18 20:37:40 on ttys000
[~] ~ ssh root@043.128.110.76
Linux VM-0-2-debian 4.19.0-11-amd64 #1 SMP Debian 4.19.146-1 (2020-09-17) x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Fri Apr 15 15:52:31 2022 from 113.54.231.118

# root @ VM-0-2-debian in ~ [17:14:50]
$ uname -a
Linux VM-0-2-debian 4.19.0-11-amd64 #1 SMP Debian 4.19.146-1 (2020-09-17) x86_64 GNU/Linux
```

图 2-1 登陆服务器 Linux 开发环境（终端）

在环境中下载所需要的库，本实验目前用到的库与工具有：libthrift, libthriftz, libthriftnb, libboost_filesystem, libboost_regex, libisal, libz, libevent, pthread, gdb 库获取的主要途径为 apt-get 包管理器，如包管理器中无此库或版本不合适，需要使用 wget 获得源码并自行编译安装，以 libevent 为例，库 libevent 源码的下载编译如代码 2-2 所示：

代码 2-2 库 libevent 源码的下载编译

```
$ wget https://github.com/libevent/libevent/releases/download/release-2.1.12-stable/libevent-2.1.12-
stable.tar.gz
$ tar -zvxf libevent-2.1.12-stable.tar.gz
$ cd libevent-2.1.12-stable
$ ./configure
$ make
$ make install
```

编译安装后的 libevent 相关库如图 2-2 所示：

```
# root @ VM-0-2-debian in /usr/local/lib [17:16:45]
[$ find . -name "*libevent.*"
./pkgconfig/libevent.pc
./libevent.a
./libevent.so
./libevent.la
```

图 2-2 编译安装后的 libevent 库

为后续进行开发与 debug，需配置 Clion 远程开发工具。在 Clion 的 Preference 标签中的 Build,Execution,Deployment 中进行配置，首先设置 Toolchain，配置远程服务器的地址，cmake 和 gdb 工具，创建新 Toolchain 并配置如图 2-3 所示：

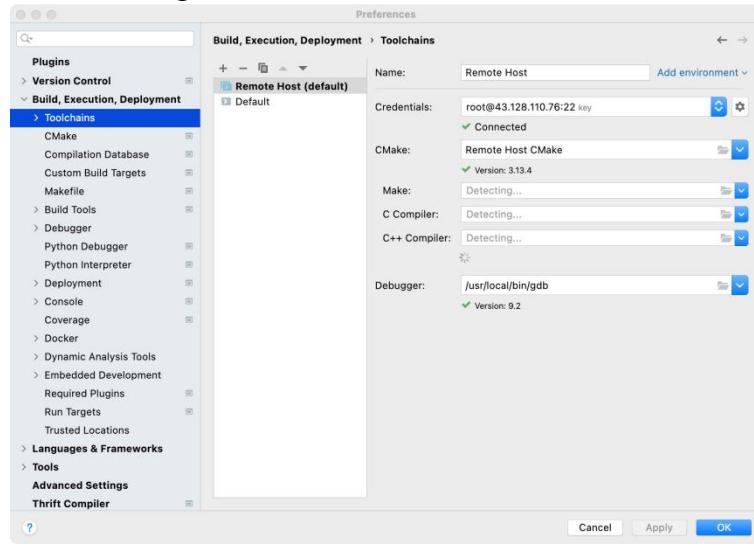


图 2-3 创建新 Toolchain 并配置

配置 CMake，并设置编译参数如图 2-4 所示。

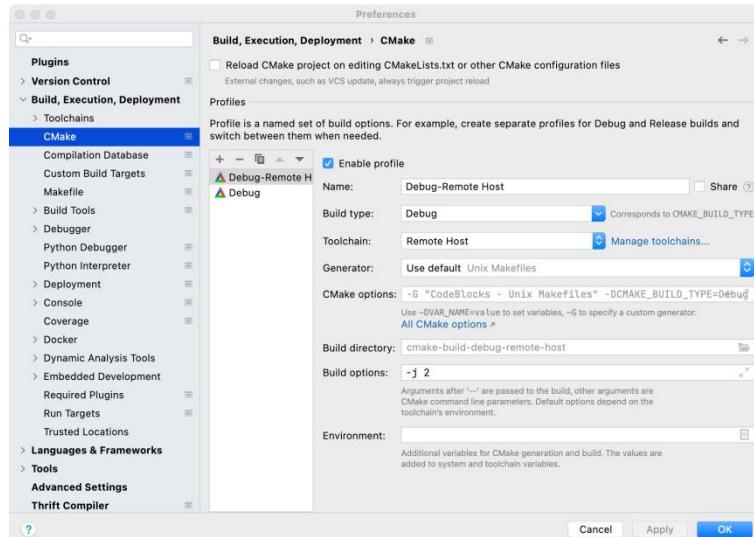


图 2-4 设置 CMake

配置 SFTP 远程引用协议并映射本地文件和服务器文件路径。SFTP 代表 SSH 文件传输协议，也称为秘密文件传输协议。它是一种网络协议，允许通过任何受信任的数据流访问、传输和管理文件。SFTP 不是在 SSH 上运行的 FTP，而是由 IETF SECSH 工作组从头设计的一种新协议。它有时与简单文件传输协议混淆 [3]。

对 SSH 链接的配置如图 2-5 所示，设置本地与远程的文件目录映射如图 2-6 所示：

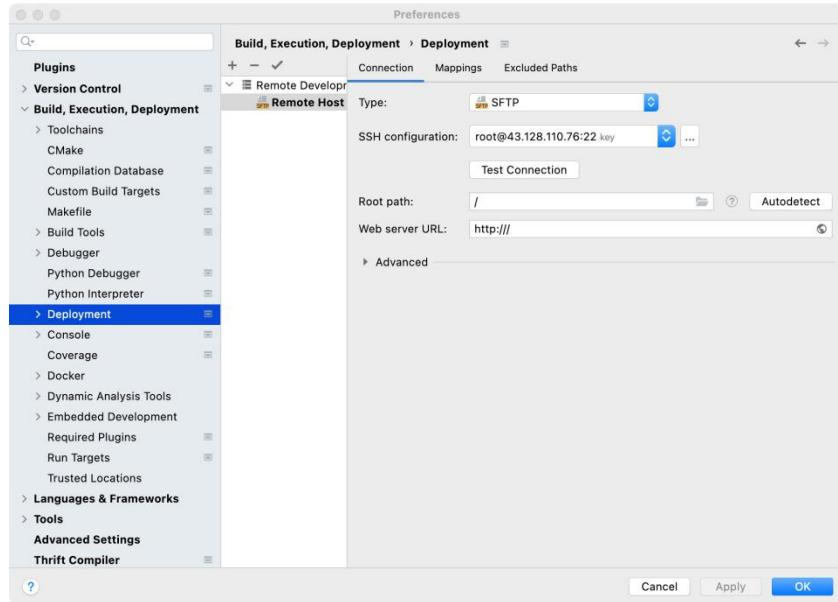


图 2-5 设置 ssh 链接

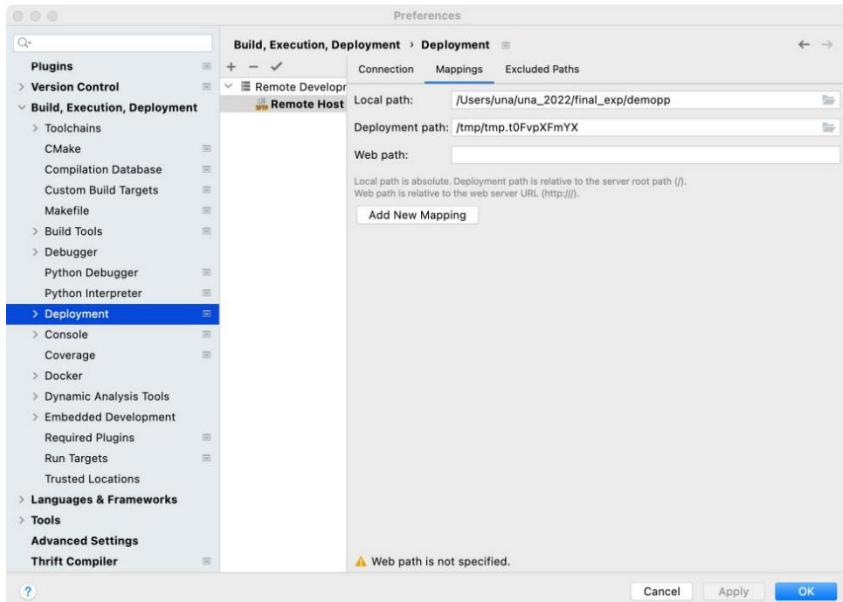


图 2-6 设置本地与远程的文件目录映射

2.2.2 Thrift 环境搭建

Thrift 包括用于构建客户端和服务器程序的完整堆栈。顶层部分是 thrift 定义生成的代码。该服务是从此文件客户端和处理程序代码生成的。通过代码生成引擎组合软件堆栈，以创建不同规模的高效跨平台服务^[2]。这里我们先搭建一个 Thrift 的环境。

下载环境相关的依赖指令如代码 2-3 所示：

代码 2-3 下载相关依赖

```
$ sudo apt-get install automake bison flex g++ git libboost-all-dev libevent-dev libssl-dev libtool make  
pkg-config
```

我主要的构建语言为 C++，因此额外下载 boost，对 boost 的下载如代码 2-4 所示：

代码 2-4 boost 依赖下载

```
$ wget http://sourceforge.net/projects/boost/files/boost/1.60.0/boost_1_60_0.tar.gz  
$ tar xvf boost_1_60_0.tar.gz  
$ cd boost_1_60_0  
$ ./bootstrap.sh  
$ sudo ./b2 install
```

从 github 下载 Thrift 源码并进行编译安装（编译源码安装方法类似 1.1.1 中对库 libevent 的下载安装），安装指令如代码 2-5 所示：

代码 2-5 编译安装 Thrift 源码

```
$ git clone https://github.com/apache/thrift.git  
$ cd thrift
```

2.3 RPC 调用应用原理

2.3.1 远程过程调用信息传递过程

远程过程调用是一个分布式计算的客户端-服务器（Client/Server）的例子，它简单而又广受欢迎。远程过程调用总是由客户端对服务器发出一个执行若干过程请求，并用客户端提供的参数。执行结果将返回给客户端。由于存在各式各样的变体和细节差异，对应地派生了各式远程过程调用协议，而且它们并不互相兼容。它的流程如下：

1. 客户端调用客户端 stub（client stub）。这个调用是在本地，并将调用参数 push 到栈（stack）中。

2. 客户端 stub (client stub) 将这些参数包装，并通过系统调用发送到服务端机器。打包的过程叫 marshalling。（常见方式：XML、JSON、二进制编码）
3. 客户端本地操作系统发送信息至服务器。（可通过自定义 TCP 协议或 HTTP 传输）
4. 服务器系统将信息传递至服务端 stub (server stub)。
5. 服务端 stub (server stub) 解析信息。
6. 服务端 stub (server stub) 调用程序，并通过类似的方式返回给客户端。

2.3.2 Thrift 框架支持的 RPC 服务的优势

首先，RPC 主要用于公司内部的服务调用，性能消耗低，传输效率高，服务治理方便。

Thrift 框架跨语言，所以使得在使用不同编程语言的程序，可以很容易的传输共享数据和进行远程过程调用。

其次，它是一个轻量级的，独立于语言的开发框架，在点对点 RPC 数据传输，数据的序列化上为我们提供了比较简洁的抽象和实现。

同样的，由于 Thrift 在传输数据时，采用的是二进制，相对于在 HTTP 协议中使用的 xml 和 json 占用体积更小，在很多高并发、数据量级大的系统中更加有优势。

2.4 Thrift 应用原理

2.4.1 Thrift 架构

Thrift 是一种二进制通信接口描述语言和协议，用于定义和构建跨语言服务。它用作 RPC (Remote Procedure Call) 框架，由 Facebook 为“广泛开发跨语言服务”而开发，它通过一个代码生成引擎联合了一个软件栈，构建无缝的、不同层次的跨平台高效服务。^[4]

Thrift 包含一套完整的栈来创建客户端和服务端程序。顶层部分^[5]是由 Thrift 定义生成的代码。该服务是从此文件客户端和处理程序代码生成的。内置类型以外的数据结构在生成的代码中创建并作为结果发送。协议和传输层是运行时库的一部分。

使用 Thrift，可以在不重新编译代码的情况下定义服务或更改通信和传输协议。除了客户端之外，Thrift 还包括一个服务器基础架构来集成协议和传输，例如阻塞、非阻塞和多线程服务器。构成 I/O 基础的堆栈部分对于不同的语言有不同的实现。

Thrift 提供众多的服务器，包括：

TNonblockingServer – 使用非阻塞 I/O (Java 实现使用 NIO 通道) 的多线程服务器。 TFramedTransport 必须与此服务器一起使用。

TSimpleServer – 使用标准 I/O 阻塞的单线程服务器。测试时很有用。

TThreadPoolServer – 使用标准块 I/O 的多线程服务器。

Thrift 一些已经明确的优点包括：

1. 语言之间的序列化成本较低，因为它使用二进制格式。
2. 拥有干净的库、编码框架和 XML 配置文件。
3. 应用层的通信格式与序列化层的通信格式完全分离。所有都可以独立编辑。
4. 预定义的序列化格式包括二进制格式、HTTP 兼容格式和紧凑二进制格式。它还充当多种语言的文件的序列化。
5. 没有构建依赖项、构建依赖项和非标准软件。没有不兼容的软件混用情况。

2.4.2 Thrift 传输协议

数据在网络上是以二进制方式传输的。

如果客户端通过网络将其发送到服务器，客户端必须在将其写入网络 IO 之前将其转换为二进制。当服务器从网络 IO 接收数据时，必须将二进制数据转换为对象才能进行操作。^[6]

在 Thrift 中，编解码操作又叫传输协议，由抽象类 TProtocol 来定义，它的方法主要分为两大类：写入方法，读取方法。

本实验的目的是添加一块 Header 区域，携带调用链信息并传递，因此我们需要在原有的二进制协议(TBinaryProtocol)基础上添加对 Header 的读写。同时，添加 crc 校验，增加可靠性。

Thrift 中常用的传输协议有以下几种：

TBinaryProtocol: 数据传输的二进制编码格式

TCompactProtocol: 用于数据传输的高效且密集的二进制编码格式

TJSONProtocol: 使用 JSON 文本数据编码协议进行数据传输

TSimpleJSONProtocol: 仅提供适用于脚本语言解析的 JSON 只写协议

TMultiplexedProtocol: 同时处理多个服务的复合协议

TProtocol 抽象类为每种数据类型提供了读写启动和进入方法。这里的读写方法应该是针对网络 IO 读写的，但真正实现网络读写的并不是这里的方法。这里的方法主要处理数据，例如更改数据格式。网络 IO 读写的实际实现是由 TTransport 相关类实现的。（TTransport 还控制它的传输方式，例如是否压缩。）TProtocol 将

特定的数据类型定义为写和读方法。由于消息必须通过网络传输，因此还定义了消息的发送和读取方式。同时还定义了一些常见的特性，比如跳过不读取某些结构，调整最终数据格式，大小，主机字节序和网络字节序之间的转换。

所有协议类都直接和间接继承自 TProtocol 类，它为上层提供了一个集成的访问接口，每个协议类都有对应的生产对象工厂（protocol factory）。类之间的继承更加复杂，并在抽象类 TProtocol 和具体协议类之间增加了第二层继承，以提供更好的可扩展性：

第一层是 TProtocolDefault。它直接继承自 TProtocol 并实现所有虚拟方法，但所有实现都是空实现。

它主要重写了抽象类 TProtocol 的非虚拟化的方法，这些方法都抛出一个方法为实现（TProtocolException::NOT_IMPLEMENTED）的异常。这样做的主要目的为了类 TVirtualProtocol 提供默认的继承基类，从而防止无限递归调用。

第二层是 TVirtualProtocol 类，它是一个模板类，带有两个模板参数，一个表示数据传输的真正协议，一个是用来继承的，它本身没有对协议具体内容做实现，所以说它是一个虚的协议类。

2.5 线程应用原理

2.5.1 线程存储期

thread_local 是一个存储类说明符，其作用类似于命名空间，并指定变量名称的存储期和链接方法。类似的关键字是：

auto: 自动存储期；

register: 自动存储期，提示编译器将此变量放入寄存器；

static: 静态或线程存储期，提示是内部链接；

extern: 静态或线程存储期，提示是外部链接；

thread_local: 线程存储期；

mutable: 不影响存储期或链接。

对于 thread_local，官方解释是：thread_local 关键词只对声明于命名空间作用域的对象、声明于块作用域的对象及静态数据成员允许。对象的存储在线程开始时分配，而在线程结束时解分配。每个线程拥有其自身的对象实例。唯有声明为 thread_local 的对象拥有此存储期。

2.5.2 线程阻塞

本论文中的就是基于线程存储期声明来完成对于调用链信息的追溯的。

.join()函数的解释：该.join 函数在线程执行完成时返回。这使该 join 函数返回的时刻与线程中所有操作的完成同步：这将阻止调用该函数的线程的执行，直到构造调用的函数返回。^[7]

本线程操纵函数可以用来做 server 服务的构建，当 server 服务线程未返回时，server 的主函数处于阻塞状态，当 server 返回，server 的主函数正常进行，调用结束函数。

.join()的使用范例如下：

代码 2-6 .join()的使用范例

```
int main()
{
    cout << "主线程开始运行\n";
    std::thread d2(download2);
    download1();
    d2.join();
    process();
}
```

线程对象 d2 调用 join()函数，所以在 d2.join()函数返回之前，必须等待 d2 下载任务完成。由于 d2 在主线程的上下文中调用 join()函数，所以主线程必须等待 d2 的线程工作完成。否则，主线程将一直被阻塞。实际任务（下载）d2 在单独的线程中执行。d2 调用 join()函数的动作是在主线程环境中进行的。

2.6 Cyclic Redundancy Check 应用原理

循环冗余查核^[8] (Cyclic redundancy check, 通称“CRC”) 是一种根据网络数据数据包或电脑文件等数据产生简短固定位数校验码的一种散列函数，它主要用于检测或验证发送或存储数据后可能发生的任何错误。传输或存储之前计算生成的数字并将其添加到数据中，并且接收者检查数据是否已更改。通常，循环冗余校验值是一个 32 位整数。

CRC 为校验和的一种，是两个字节数据流采用二进制除法（没有进位，使用 XOR 来代替减法）相除所得到的余数。其中被除数是需要计算校验和的信息数据流的二进制表示；除数是一个长度为 $n + 1$ 的预定义（短）的二进制数，通常用多项式的系数来表示。在做除法之前，要在信息数据之后先加上 n 个 0.

CRCa 是基于有限域 **GF(2)** (即除以 2 的同余) 的多项式环。简单的来说，就是所有系数都为 0 或 1 (又叫做二进制) 的多项式系数的集合，并且集合对于所有的代数操作都是封闭的。例如：

$$(x^3 + x) + (x + 1) = x^3 + 2x + 1 \equiv x^3 + 1 \quad (2.1)$$

2 会变成 0，因为对系数的加法运算都会再取 2 的模数。乘法也是类似的：

$$(x^2 + x)(x + 1) = x^3 + 2x^2 + x \equiv x^3 + x \quad (2.2)$$

我们同样可以对多项式作除法并且得到商和余数。例如，如果我们用 $x^3 + x^2 + x$ 除以 $x + 1$ 。我们会得到：

$$\frac{(x^3 + x^2 + x)}{(x+1)} = (x^2 + 1) - \frac{1}{(x+1)} \quad (2.3)$$

或者：

$$(x^3 + x^2 + x) = (x^2 + 1)(x + 1) - 1 \quad (2.4)$$

等价于：

$$(x^2 + x + 1)x = (x^2 + 1)(x + 1) - 1 \quad (2.5)$$

这里除法得到了商 $x^2 + 1$ 和余数 -1，因为是奇数所以最后一位是 1。

字符串的每一位实际上对应于这类多项式的系数。为了得到 CRC，我们首先将其乘以 x^n ，这里 n 是一个固定多项式的阶数，然后再将其除以这个固定的多项式，余数的系数就是 CRC。

在上面的等式中， $x^2 + x + 1$ 表示了本来的信息位是 111， $x + 1$ 是所谓的钥匙，而余数 1 (也就是 x^0) 就是 CRC. key 的最高次为 1，所以我们将原来的信息乘上 x^1 来得到 $x^3 + x^2 + x$ ，也可视为原来的信息位补 1 个零成为 1110。

一般来说，其形式为：

$$M(x) \cdot x^n = Q(x) \cdot K(x) - R(x) \quad (2.6)$$

这里 $M(x)$ 是原始的信息多项式。 $K(x)$ 是 n 阶的“钥匙”多项式。 $M(x) \cdot x^n$ 表示了将原始信息后面加上 n 个 0。 $R(x)$ 是余数多项式，即是 CRC“校验和”。在通信中，发送者在原始的信息数据 M 后附加上 n 位的 R (替换本来附加的 0) 再发送。接收者收到 M 和 R 后，检查 $M(x) \cdot x^n + R(x)$ 是否能被 $K(x)$ 整除。如果是，那么接收者认为该信息是正确的。值得注意的是 $M(x) \cdot x^n + R(x)$ 就是发送者所想要发送的数据。这个串又叫做 *codeword*.

CRCs 通常被称为“校验和”，但这样的陈述并不完全准确，因为技术上校验和是通过加法而不是除法计算的。

“错误纠正编码”(Error-Correcting Codes, 简称 ECC) 常常和 CRCs 紧密相关，其语序纠正正在传输过程中所产生的错误。这些编码方式常常和数学原理紧密相关。

2.7 SPDLogSink 组件应用原理

调用链信息的回溯需要配合 log 组件记录，本项目选用 SPDLogSink 实现。SPDLogSink 是 BlobStorage 项目使用的日志组件，其基于开源 spdlog 库二次开发。此 log 组件的使用较为便捷，其源码基于 C++11 标准，使用时只需添加 log.h 与 log.cc，现有代码的 CMakeList 无需更改。同时，本组件支持在同一个进程中通过自定义宏将日志输出到不同路径下。

流水日志初始化函数声明如代码 2-7 所示，其中传入参数 model_name 是流水日志文件的文件名。log_dir 是流水日志文件保存路径，buffer_size_in_Mbytes 是日志组件缓冲区预设大小，log_key 用于区分不同的 logger，可支持将日志输出到不同文件中：

代码 2-7 流水日志初始化

```
int InitNormalLog(const std::string& model_name,
                  const std::string& log_dir,
                  std::size_t buffer_size_MB = 64
                  const std::string& log_key = "global")
```

本组件支持设置日志记录的过滤等级，只有超出 level 等级的日志条目才会被记录输出，level 的取值与日志规范定义的日志等级一致(DEBUG、TRACE、INFO、WARN、FATAL)，string_level 为 level 的同名字符串("DEBUG"、"TRACE"、"INFO"、"WARN"、"FATAL")，对过滤登记的设置如代码 2-8 所示：

代码 2-8 设置日志文件过滤等级

```
int SetFilterLevel(LogLevel level) const;
int SetFilterLevel(const std::string& string_level) const;
```

设置切分日志文件大小如代码 2-9 所示：

代码 2-9 设置日志文件切分大小

```
int SetSplitSize(std::size_t max_file_size_in_Mbytes) const;
```

第三章 设计与实现

3.1 设计整体框架

3.1.1 客户端设计框架

客户端是请求的发起端，设计工作涉及三层，业务层，框架层和协议层（传输层不进行修改，不涉及）。设计在业务层提供客户端继承类，并提供类函数 WriteBlock() 和 ReadBlock()，并在协议层添加对信息头的写入和读取，对 CRC 码的校验。

Client 的整体设计框架如图 3-1 所示：

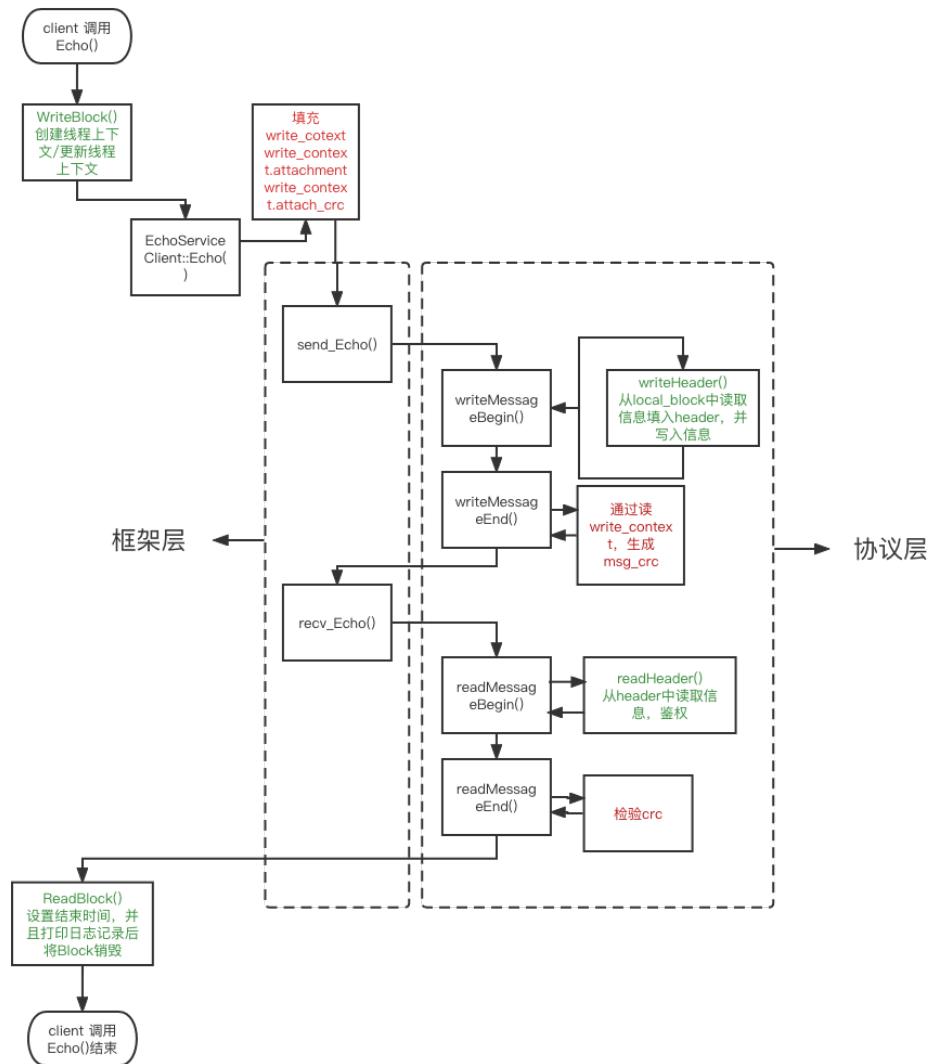


图 3-1 客户端整体设计

各模块设计简要介绍如下：

WriteBlock()和ReadBlock()设计为新建类ThriftClient的类函数，客户端将继承此类，设计这两个函数的第一个目的是将RPC流程入口整合到一个函数中，在便于进行错误捕捉以及回滚；第二个目的是如果客户端完成了符合预期的流程，将在WriteBlock()中填充调用链信息到Block中，便于协议层中读取Block信息对信息头进行填充，并在ReadBlock()函数中进行数据埋点，将Block中的数据进行存储，并清空Block。Block为我们自己设计的线程存储数据结构（具体设计见3.2.2线程存储期数据存储块Block），用于在线程上下文中存储调用链信息。

对于协议层的修改，将继承抽象类TProtocol实现（具体继承关系见3.3.1官方协议TBinaryProtocol的研究）。在协议层，Header为我们设计的消息头数据结构（具体设计见3.2.1信息头结构CommonHeader），它会携带调用链信息跟随请求/回复信息发送到服务端，便于服务端对调用链信息进行处理以及记录。writeHeader()为我们添加入writeMessageBegin()的函数，它将读取Block中信息，填充Header，编码后写入发送信息。ReaderHeader()将读取回复信息，并鉴权后，用新的调用链信息填充Block。

涉及CRC校验部分设计在框架图中由红色显示，由附件生成CRC校验码，并通过线程存储期write_context存储，并将其封装到发送信息中。使服务端可以通过本CRC码校验数据在传输中的完整性。CRC是一种功能强大的校验和类型，能够检测存储在计算机和/或在计算机之间传输的数据的损坏。在新协议中会加入对附件的CRC校验，需要编写CRC的编码和校验接口。

具体实现见3.5客户端的实现。

3.1.2 服务端设计框架

server对于协议层的调用不是显式的，而是通过处理器自动触发。线程使用工厂模式，可以多线程处理请求。

客户端和服务端对信息的读写顺序是相反的，使用同一个协议，它们都会通过Thrift框架被调用，因此协议的信息读写部分需要区分调用者的身份。比如客户端对调用链信息的线程存储数据结构Block（此数据结构的设计和作用见3.2.2线程存储期数据存储块Block）的填充和读取并不是在协议层调用的，而是在业务层被调用的。

server的整体框架设计如图3-2所示：

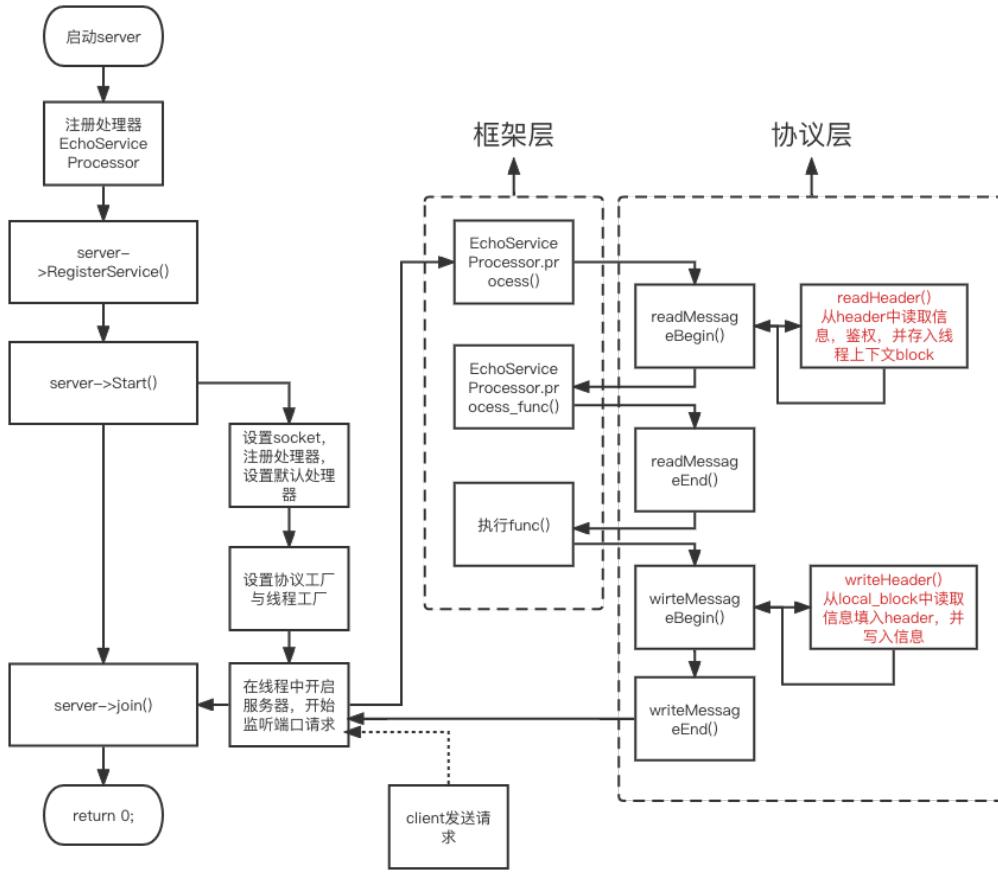


图 3-2 服务端架构设计

3.1.3 调用链设计框架

如图 3-3 为模拟调用链，在 server 进行 Echo() 服务时，调用 EchoPutAttach()，形成调用链。

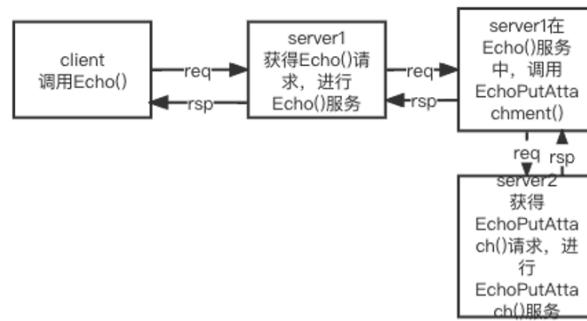


图 3-3 调用链设计

3.1.4 鉴权系统设计框架

设计在协议层完成鉴权，鉴权系统依赖于密钥，密钥白名单，以及密钥的加解密算法。主要数据流程如下，其中，`ServerInfo` 为存储相关服务器及鉴权信息的数据结构（具体设计见本章 3.2.4 服务信息存储结构 `ServerInfo` 类）。如图 3-4 为整体的鉴权设计框架：

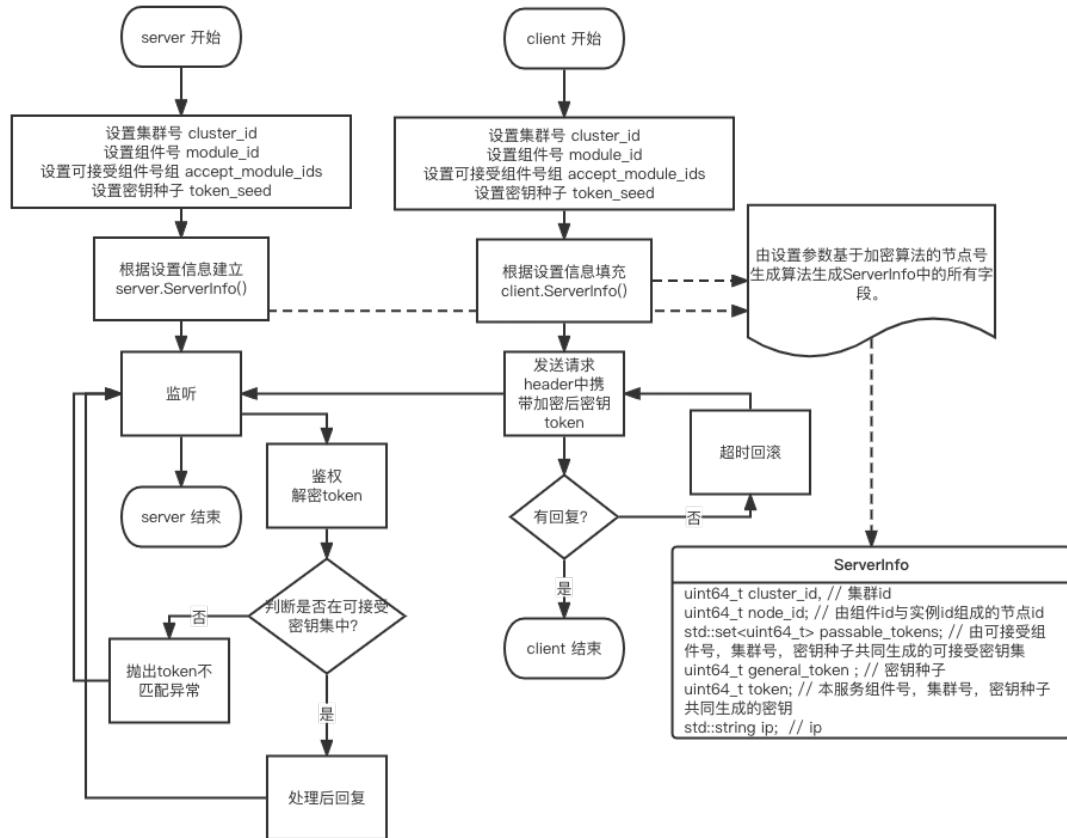


图 3-4 鉴权设计框架

首先，需要通过可配置文件，编译参数传入或代码中嵌入等方式配置集群号，组件号，组件白名单和密钥种子，其中服务端和客户端的密钥种子和密钥加密方式应该相同，保证客户端携带的密钥在服务端的白名单中。

服务端在进行请求处理之前，需要将从请求头中读取的密钥信息进行解码，与白名单进行比对，如果鉴权通过则正常进行请求处理，如果不通过则抛出异常。客户端应该在这种情况下进行回滚与信息埋点。

密钥结构设计如图 3-5，通过位运算进行加解密（具体实现见本章 3.4 节 鉴权的实现）：

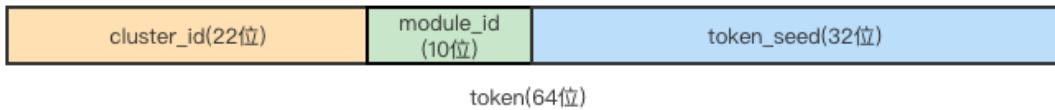


图 3-5 密钥结构

3.2 设计数据结构

3.2.1 信息头结构 CommonHeader

构建信息头的结构。在.thrift 文件中创建存储调用链信息的数据结构 CommonHeader，其成员变量如图 3-1 所示：其中，version 指版本 id，node_id 指节点 id，token 为鉴权信息，trace_id 为调用链 id，block

_id 为线程上下文数据存储块的 id，parent_block_id 为父节点的线程上下文数据存储块的 id，client_ip 为调用方的 ip，rpc_try_time 为 rpc 请求重试时间。基于新的.thrift 重新生成框架代码 gen-cpp。

代码 3-1 信息头结构设计

```
// project/thrift/echo.thrift
struct CommonHeader {
    1:i8    version
    2:i64   node_id
    3:i64   token
    4:i64   trace_id
    5:i64   block_id
    6:i64   parent_block_id
    7:i32   client_ip
    8:optional i32 segment_size
    9:optional i32 rpc_try_time
}
```

3.2.2 线程存储期数据存储块 Block

构建线程存储期数据存储块结构。Block 为我们自己设计的线程存储数据结构，用于在线程上下文中存储调用链信息。创建 Block 类。设置私有类变量，如代码 3-2 所示，并设置对应的公有函数操作这些私有变量。其中 end_time_ 为请求/服务结束时间，is_callee_ 标示本线程为服务端线程还是客户端线程，annotation_ 为备注信息，tags_ 为标签信息。parent_block_ 为指向父节点的指针。

代码 3-2 线程存储数据结构类 Block 设计

```
// project/include/block.h
// 对线程存储数据结构类 Block 的设计
Class Block{
```

```

private:
    uint64_t trace_id_;
    uint64_t block_id_;
    uint64_t parent_block_id_;
    uint64_t node_id_;
    int64_t start_time_;
    int64_t end_time_;
    bool is_callee_;
    std::string annotation_;
    std::map<uint64_t, std::string> annotation_map_;
    std::map<std::string, std::string> tags_;
    std::shared_ptr<Block> parent_block_ = nullptr;
};

```

在 Block 类中设置访问私有成员变量的公有函数。如代码 3-3 为部分函数实现：

代码 3-3 线程存储数据结构类 Block 成员函数设计

```

// project/include/block.h
// member functions of Block
Class Block{
public:
    Block(uint64_t node_id, bool is_callee, const std::string& annotation = "");
    Block(const Block& s);
    uint64_t trace_id() const {return trace_id_;}
    void set_trace_id(uint64_t trace_id) {trace_id_ = trace_id; }
    uint64_t block_id() const {return block_id_;}
    void set_block_id(uint64_t block_id) {block_id_ = block_id; }
    uint64_t parent_block_id() const {return parent_block_id_;}
    // other set&get private members' functions ...
};

```

Block 类中设置信息埋点函数，记录调用链信息。如代码 3-4 为线程存储数据结构类 Block 中设置信息埋点函数：

代码 3-4 线程存储数据结构类 Block 中设置信息埋点函数

```

// project/include/block.h
// Set the information buried function
namespace UTILS {
Class Block{
void Submit() {
    LOGF(TRACE, " trace_id {} block_id {} parent_block_id {} start_time {} end_time {} is_callee {} annotation {}",
        trace_id_, block_id_, parent_block_id_, start_time_, end_time_, is_callee_, annotation_);
}

void PrintInfo() {
    std::string annotations_json = "{";auto annotation_iter = annotation_map_.begin();
    while (annotation_iter != annotation_map_.end()) {
        annotations_json += std::to_string(annotation_iter->first) + ":" +
            "\\" + annotation_iter->second + "\\";
        if (++annotation_iter != annotation_map_.end()) {
            annotations_json += ",";
        }
    }
}

```

```

}
annotations_json += "}";

std::string tag_json = "{}";
auto tag_iter = tags_.begin();
while (tag_iter != tags_.end()) {
    tag_json += "\"" + tag_iter->first + "\"" + ":" + "\"" + tag_iter->second + "\"";
    if (++tag_iter != tags_.end()) {
        tag_json += ",";
    }
}
tag_json += "}";

LOGF(INFO, "[TraceLog] trace_id={} block_id={} parent_block_id={} start_time={} end_time={}
is_callee={}"
    " annotation={} tags={}",
    trace_id_, block_id_, parent_block_id_, start_time_, end_time_, is_callee_,
    annotations_json, tag_json);
}
};
}

```

3.2.3 数据存储期结构 ThreadContext 类

设置线程存储期类型 ThreadContext 类。其中，应该包括线程存储数据块 Block，rpc 信息发送的本地 host 和 rpc 的远程 host。并设定对于本结构体的符号重载，初始化，以及清空操作。Block 和 Thread_context 的定义和操作都放到了 UTILS 的命名空间，便于清晰管理。如代码 3-5 设置数据存储期结构 ThreadContext 类。

代码 3-5 数据存储期结构 ThreadContext 类

```

// project/include/Thread_context.h
namespace UTILS {
struct ThreadContext {
    std::shared_ptr<Block> lock;
    std::string rpc_local_host;
    std::string rpc_remote_host;
    ThreadContext& operator=(const ThreadContext& ctx) {
        if (ctx.block != nullptr) { block = std::make_shared<UTILS::Block>(*(ctx.block)); }
        rpc_local_host = ctx.rpc_local_host;
        rpc_remote_host = ctx.rpc_remote_host;
        return *this;
    }
    ThreadContext(const ThreadContext& ctx) {
        if (ctx.block == nullptr) { block.reset(); } else {
            block = std::make_shared<UTILS::Block>(*(ctx.block));
        }
        rpc_local_host = ctx.rpc_local_host;
        rpc_remote_host = ctx.rpc_remote_host;
    }
    void Clear() {
        block.reset();
        rpc_local_host = "";
        rpc_remote_host = "";
    }
};

```

```
std::unique_ptr<ThreadContext>& GetThreadContext();
void ClearThreadContext();
}
```

3.2.4 服务信息存储结构 ServerInfo 类

设置服务的基本信息以及鉴权白名单，用于生成密钥和可接受密钥集。存储类结构如代码 3-6 所示：

代码 3-6 服务信息存储结构 ServerInfo 类设计

```
//project/include/thrift/api/operation.h
struct ServerInfo {
    uint64_t cluster_id;
    uint64_t node_id;
    std::set<uint64_t> passable_tokens;
    uint64_t general_token;
    uint64_t token;
    std::string ip; // host which module locate
};
```

各变量介绍如下：

`cluster_id`: 集群 id， 默认跨集群服务无法访问。

`node_id`: 节点 id， 其生成需要依赖于组件 id 和实例 id。

`passable_tokens`: 密钥白名单， 服务依赖本名单进行鉴权， 本服务只会为在密钥白名单上的请求提供服务。

`general_token` : 密钥种子。

`token`: 密钥， 由密钥种子， 集群 id， 组件 id 加密生成。

`Ip`: 组件所在服务器 ip。

其中 `node_id` (节点 id) 可与 `module_id` (组件) 互相推导， 其中需要用到 `instance_id` (实例 id)， 推导关系如代码 3-7 所示：

代码 3-7 节点 id 与组件 id 互相推导关系

```
//project/include/thrift/api/operation.h
#define MODULE_ID(node_id)((node_id & 0xffff0000000000ULL) >> 54)
#define NODE_ID(module_id, instance_id)((uint64_t)module_id << 54) + instance_id
```

3.2.5 线程存储期读写辅助存储块 RpcContext 类

本存储块类似于线程存储期数据存储块 Block， 都是线程存储期类型变量。区别在于 Block 类主要存储调用链信息， 用于信息埋点； 本 RpcContext 类主要用于辅助记录 CRC 校验信息以及 RPC 重试次数与时间。其数据结构如代码 3-8 所示：

代码 3-8 线程存储期读写辅助存储块 RpcContext 类设计

```
// project/include/thrift/api/operation.h
struct RpcContext {
    std::map<std::string, std::string> header;
    std::string attachment;
    uint32_t attachment_crc = 0;
    uint64_t connection_create_time_us = 0;
    int32_t rpc_try_time = 1;
    std::vector<uint32_t> attachment_segment_crcs;//for read context
};
```

header: 字典存储区，存储一些辅助信息。

attachment: 附件。

attachment_crc: 附件的 CRC 编码。

connection_create_time_us: 链接创建时间（美国标准时间）

rpc_try_time: rpc 重试次数，控制回滚次数。

attachment_segment_crcs: 分段计算 CRC 编码并存储。

3.3 新协议 HeaderBinaryProtocol 的实现

3.3.1 官方协议 TBinaryProtocol 的研究

先研究 TBinaryProtocol 的实现：

首先它继承了 TVirtualProtocol，这是一个可继承的协议类。Thrift 开发团队为了以后的协议可扩展性添加了这个继承层，允许其他组织、团队或个人实现自己的数据（主要是数据格式封装）协议。其实 TVirtualProtocol 是一个静态绑定模板，继承自默认实现类 TProtocolDefaults，TProtocolDefaults 继承自 TProtocol 类。之所以存在这样复杂的继承关系，是为了令 TVirtualProtocol 实现多态，但是避免虚拟继承。虚函数调用相比于普通函数调用有性能损失，性能损失来自几个方面：

1. 虚函数无法内联优化
2. 虚函数需要重新查虚函数表等更多的流程
3. 虚函数导致指令不紧凑，有很大的可能 cache miss，在缓存中找不到需要调用的函数代码
4. 影响 CPU 流水线

TVirtualProtocol 采用 CRTP (Curiously Recurring Template Pattern: 奇特的递归模板模式) 的方式静态绑定模板来解决这个问题。CRTP(Curiously Recurring Template Pattern)，是一种实现静态多态的 C++ 模板编程技巧。其基本做法是将派

生类作为模板参数传递给它自己的基类。虚函数重写多态是动态绑定（运行时绑定），而 CRTP 函数覆盖是静态绑定（编译时绑定）。

CRTP 为了防止产生递归，需要子类覆盖父类方法，`TProtocolDefaults` 类防止了递归的出现，继承关系的部分实现代码如代码 3-9 所示：

代码 3-9 Thrift 协议类之间的集成关系

```

class TProtocol
public:
    virtual uint32_t writeByte_virt(const int8_t byte) = 0;
    virtual uint32_t writeI32_virt(const int32_t i32) = 0;
    uint32_t writeI32(const int32_t i32) {return writeI32_virt(i32);}
    // other code ...
};

// A default implementation of TProtocol that overrides the functions of the base class without recursion
// problems
class TProtocolDefaults : public TProtocol
{
public:
    uint32_t writeByte(const int8_t byte) {throw
TProtocolException(TProtocolException::NOT_IMPLEMENTED, "this protocol does not support
writing (yet).");}
    uint32_t writeI32(const int32_t i32) {throw
TProtocolException(TProtocolException::NOT_IMPLEMENTED, "this protocol does not support
writing (yet).");}
};

// A complete CRTP base class, with no virtual functions and no recursion
template <class Protocol_, class Super_ = TProtocolDefaults>
class TVirtualProtocol : public Super_
{public: uint32_t writeByte_virt(const int8_t byte) override {
    return static_cast<Protocol_*>(this)->writeByte(byte); }
    uint32_t writeI32_virt(const int32_t i32) override {
        return static_cast<Protocol_*>(this)->writeI32(byte); }
}

```

用图表现类的集成关系如图 3-6 所示，`TBinaryProtocol` 为协议类的一种具体实现：

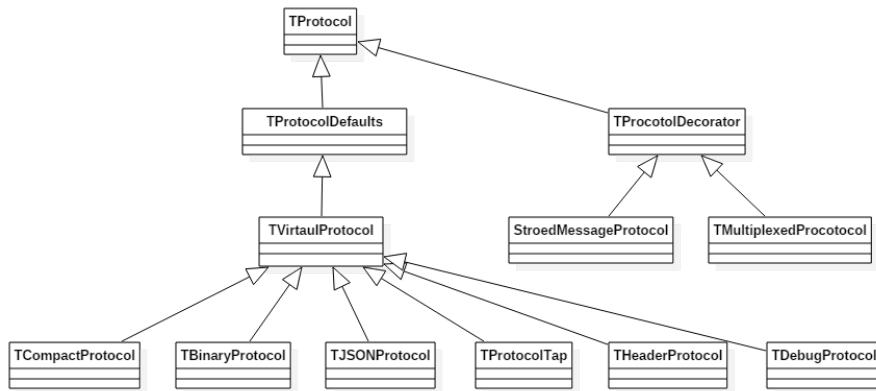


图 3-6 Thrift 协议类集成关系图

对于 TBinaryProtocol 中的具体实现，首先它根据初始化传入的信息类型与数量，设置了其初始化方式，如代码 3-10 所示：

代码 3-10 TBinaryProtocol 类初始化函数

```
// thrift/lib/cpp/src/thrift/protocol/TBinaryProtocol.h
template <class Transport_, class ByteOrder_ = TNetworkBigEndian>
class TBinaryProtocolT : public TVirtualProtocol<TBinaryProtocolT<Transport_, ByteOrder_>> {
public:
    static const int32_t VERSION_MASK = ((int32_t)0xfffff0000);
    static const int32_t VERSION_1 = ((int32_t)0x80010000);
    // VERSION 2 (0x80020000) was taken by TDenseProtocol (which has since been removed)
    TBinaryProtocolT(std::shared_ptr<Transport_> trans)
        : TVirtualProtocol<TBinaryProtocolT<Transport_, ByteOrder_>>(trans), trans_(trans.get()),
          string_limit_(0), container_limit_(0), strict_read_(false), strict_write_(true) {}
    TBinaryProtocolT(std::shared_ptr<Transport_> trans, int32_t string_limit, int32_t container_limit,
                    bool strict_read, bool strict_write)
        : TVirtualProtocol<TBinaryProtocolT<Transport_, ByteOrder_>>(trans), trans_(trans.get()),
          string_limit_(string_limit), container_limit_(container_limit), strict_read_(strict_read),
          strict_write_(strict_write) {}
```

其次，设置信息读取写入的长度限制的方法函数，如代码 3-11 所示：

代码 3-11 TBinaryProtocol 类对信息读取写入的长度限制的方法函数

```
// thrift/lib/cpp/src/thrift/protocol/TBinaryProtocol.h
void setStringSizeLimit(int32_t string_limit) {}
void setContainerSizeLimit(int32_t container_limit) {}
void setStrict(bool strict_read, bool strict_write) {}
```

协议类实现了对信息的写入和对信息的读取函数，以及对不同数据结构信息的析构编码方式，代码 3-12 为部分方法函数列表：

代码 3-12 TBinaryProtocol 类部分对不同数据结构信息的析构编码方法函数

```
// thrift/lib/cpp/src/thrift/protocol/TBinaryProtocol.h
/* Writing functions */
/*ol*/ uint32_t writeMessageBegin(const std::string& name,
                                  const TMessageType messageType,
                                  const int32_t seqid);
/*ol*/ uint32_t writeMessageEnd();
inline uint32_t writeStructBegin(const char* name);
inline uint32_t writeStructEnd();
inline uint32_t writeFieldBegin(const char* name, const TType fieldType, const int16_t fieldId);
inline uint32_t writeFieldEnd();
inline uint32_t writeFieldStop();
inline uint32_t writeMapBegin(const TType keyType, const TType valType, const uint32_t size);
inline uint32_t writeMapEnd();
// other writing functions ...
/* Reading functions */
/*ol*/ uint32_t readMessageBegin(std::string& name, TMessageType& messageType, int32_t& seqid);
/*ol*/ uint32_t readMessageEnd();
inline uint32_t readStructBegin(std::string& name);
inline uint32_t readStructEnd();
inline uint32_t readFieldBegin(std::string& name, TType& fieldType, int16_t& fieldId);
```

```

inline uint32_t readFieldEnd();
inline uint32_t readMapBegin(TType& keyType, TType& valType, uint32_t& size);
inline uint32_t readMapEnd();
// other reading functions ...

```

最后构造二进制协议处理程序，并进行类型定义，代码 3-13 为对二进制协议别名对定义：

代码 3-13 定义二进制协议别名

```

typedef TBinaryProtocolT<TTransport> TBinaryProtocol;
typedef TBinaryProtocolT<TTransport, TNetworkLittleEndian> TLEBinaryProtocol;
< /**
 * Constructs binary protocol handlers
 */
template <class Transport_, class ByteOrder_ = TNetworkBigEndian>
class TBinaryProtocolFactoryT : public TProtocolFactory {
    // other reading functions ...
};

typedef TBinaryProtocolFactoryT<TTransport> TBinaryProtocolFactory;
typedef TBinaryProtocolFactoryT<TTransport, TNetworkLittleEndian> TLEBinaryProtocolFactory;
}

```

经过对 TBinaryProtocol 的继承关系和其方法函数的研究，可以大体了解 Thrift 协议的编写模式。在构造新的协议时，同样需要继承模板类 TVirtualProtocol。在复用 TBinaryProtocol 中的部分信息写入与读出函数的基础上，添加对于信息的 CRC 校验，增加信息的可靠性。

为实现将调用链信息随请求/回复传递的目的，添加读取信息头与写入信息头的函数，并将此函数的调用分别加入 writeMessageBegin() 和 readMessageBegin()。

为实现将调用链信息基于线程存储期存储，需要在服务端线程/客户端线程中维护线程存储期，这个过程将主要在协议层完成（部分在服务端/客户端业务层完成）。

3.3.2 协议的调用流程

TVirtualProtocol 是一个可继承的协议类，Thrift 开发团队为了后期协议的可扩展性和允许其他组织、团队或个人实现自己的数据（主要是数据格式的封装）协议，增加了这层继承关系。通过继承 TVirtualProtocol，可以实现满足需求的协议 HeaderBinaryProtocol，无感知地完成调用链信息的埋入。

协议层的函数对于客户端和服务端是共用的，但是调用的顺序是不同的。以下为客户端和服务端各自协议端调用流程图，有部分框架层的函数出现，便于了解协议层在整条调用链中的位置。了解协议调用流程对于设计新的协议非常重要。如图 3-7 为客户端调用协议的流程图，图 3-8 为服务端调用协议的流程图：

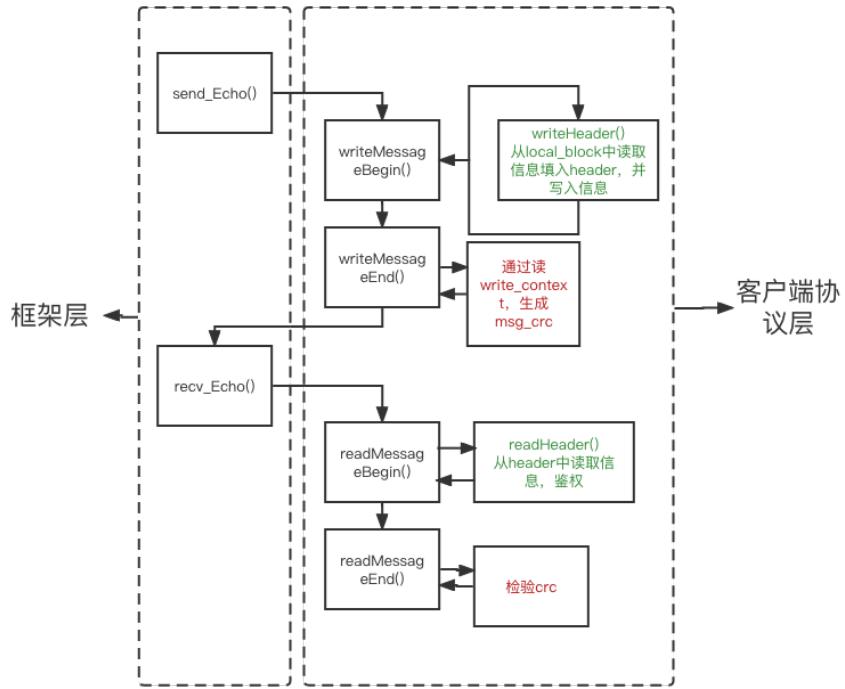


图 3-7 客户端的协议层调用流程图

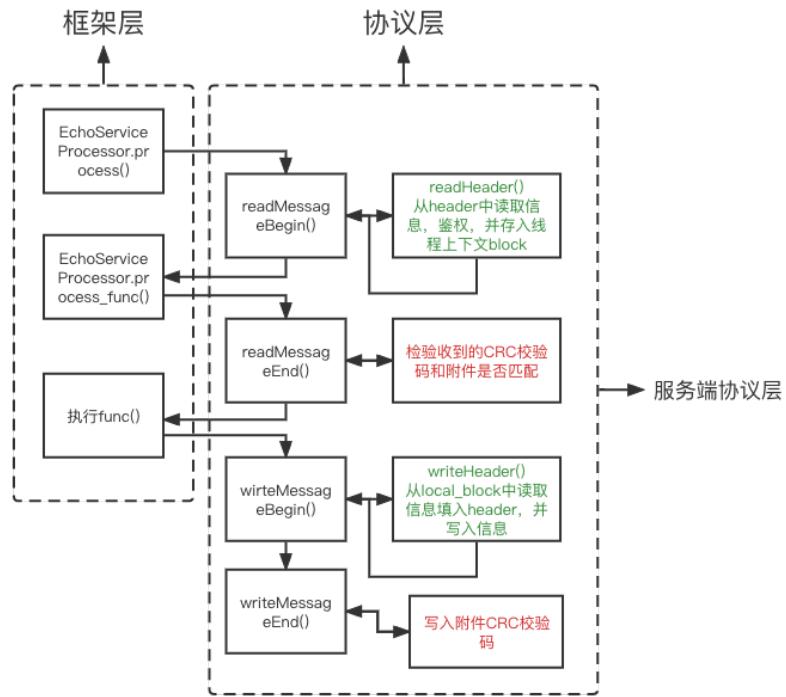


图 3-8 服务端协议层函数调用流程图

其中，带颜色的节点为新协议主要的增加部分，其具体实现将在本章说明。

3.3.3 CRC 校验接口编写

CRC 是一种功能强大的校验和类型，能够检测存储在计算机和/或在计算机之间传输的数据的损坏。在新协议中会加入对附件的 CRC 校验，需要编写 CRC 的编码和校验接口。

在写信息函数 `writeMessageEnd()` 中，会基于附件生成校验和 CRC 信息，并且将其加入传输信息，在读信息函数 `readMessageEnd()` 中，会基于附件生成校验和 CRC 信息，并与传输读到的 CRC 信息进行比对。如果两者有差异，说明附件信息在传输的过程中有损耗，抛出 CRC 不匹配的错误。

如下代码 3-14 为计算 CRC 校验和使用的组件。其中 `Value()` 可以计算数据为 `data`，长度为 `n` 的数据的校验和。`Extend` 可以在已有 CRC 校验和的基础上计算加入数据 `data` 后新的校验和，`Combine` 类似于 `Extend(Value(data1, len1), data2, len2)`；同时计算两个数据的校验和。

代码 3-14 CRC 编码接口的定义

```
// project/include/tools/crc/crc.h
#include <cstdint> //uint8_t
#include <cstddef>
namespace UTILS {
namespace crc32 {
uint32_t Extend(uint32_t crc, const char *data, size_t len);
inline uint32_t Value(const char *data, size_t n) {return Extend(0, data, n);}
uint32_t Combine(uint32_t crc1, uint32_t crc2, size_t len2); // : Extend(Value(data1, len1), data2, len2);
} //end namespace crc32
}
```

具体实现如代码 3-15 所示，其中 `crc32::Extend` 基于过库 `isa-l/crc.h` 中 `crc32_gzip_refl()` 实现，`crc32::Combine` 基于库 `zlib.h` 中 `crc32_combine()` 实现：

代码 3-15 CRC 编码接口的实现

```
// project/include/tools/crc/crc.cpp
#include "crc.h"
#include <zlib.h>
#include <isa-l/crc.h>
using namespace UTILS::crc32;
uint32_t UTILS::crc32::Extend(uint32_t crc, const char* data, size_t len) {
    return crc32_gzip_refl(crc, (unsigned char*)data, len);}
uint32_t UTILS::crc32::Combine(uint32_t crc1, uint32_t crc2, size_t len2) {
    return crc32_combine(crc1, crc2, len2);}
```

3.3.4 信息头的写入

信息头填充主要由函数 writeHeader()完成，其返回值为二进制编码后的信息头，编码也会被写入传输层，本函数会在 writeMessageBegin()中被调用。设计流程如图 3-9 所示：

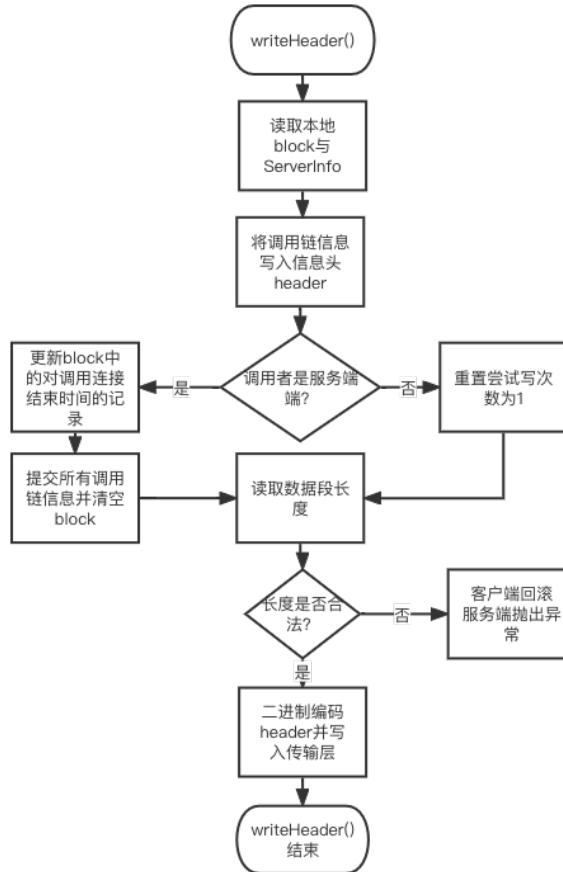


图 3-9 writeHeader 流程图设计

对于客户端，会在发出请求前将调用链信息写入信息头，进行二进制编码后发送至服务端；对于服务端，会在发送回复时将调用链信息写入信息头，进行二进制编码后发送至客户端，需要对两种情况分别处理。

调用者身份分支部分代码如代码 3-16 所示：

代码 3-16 writeHeader() 调用者分支代码

```

// project/include/thrift/protocol/header_binary_protocol.tcc
if (messageType == ::apache::thrift::protocol::T_REPLY) {
    // means service server response
    if (local_block == nullptr) {
        throw ::apache::thrift::TApplicationException(
            ::apache::thrift::TApplicationException::MISSING_RESULT,
  
```

```

    "send reply but has no block");}
local_block->set_end_time(UTILS::TimeUtils::GetCurrentTimeMs());
local_block->Submit();
// destroy current block
assert(local_block->parent_block() == nullptr);
UTILS::GetThreadContext()->block.reset();
} else if (messageType == ::apache::thrift::protocol::T_CALL) {
    // means client server request
    auto rpc_try_time = getWriteRpcTryTime();
    if (rpc_try_time > 1) {
        LOG_F(TRACE, "RpcTryTime={} ", rpc_try_time);
        header._set_rpc_try_time(rpc_try_time);
        setWriteRpcTryTime(1); // Reset
    }
}

```

其中 `getWriteRpcTryTime()` 是从 `RpcContext` 类数据结构 `write_context` 中读取 RPC 写入的尝试次数 (`RpcContext` 类数据结构设计见 3.2.5 线程存储期读写辅助存储块 `RpcContext` 类)。如果多次的话，将尝试次数记录到信息头中，并将本字段清空。

在一些情况下，需要对发送的信息进行分段发送，段长根据网络情况进行设置，需要对设置的段长合法性进行判断。段长也存储在 `write_context` 中，以 `string` 的格式存储。可以遇见的段长错误有多种，如数值为负，`string` 类型转 `int` 类型可能出现的无意义转换错误，超出 `int32` 位数值的超范围错误。

如代码 3-17 所示，为判断段长合法性的部分代码：

代码 3-17 `writeHeader()` 中对段长设置合法性检验的实现

```

// project/include/thrift/protocol/header_binary_protocol.tcc
std::string segment_size_s = getWriteHeader("segment_size");
if (!segment_size_s.empty()) {
    int32_t segment_size;
    try {
        segment_size = std::stoi(segment_size_s);
        if (segment_size < 0) {
            throw ::apache::thrift::TApplicationException(
                ::apache::thrift::TApplicationException::UNKNOWN,
                "invalid segment size ");
        }
        header._set_segment_size(segment_size);
    } catch (std::invalid_argument const &e) {
        throw ::apache::thrift::TApplicationException(::apache::thrift::TApplicationException::UNKNOWN,
                                                       "invalid segment size ");
    } catch (std::out_of_range const &e) {
        throw ::apache::thrift::TApplicationException(::apache::thrift::TApplicationException::UNKNOWN,
                                                       "segment size out of range");
    }
}

```

3.3.5 CRC 校验码的写入

需要在将附件的 CRC 校验码随消息发送到连接的服务器，便于对方校验附件的完整性。计算，写入 CRC 校验码的功能在函数 writeMessageEnd()中实现，是封装信息的最后一步。

CRC 是一种功能强大的校验和类型，能够检测存储在计算机和/或在计算机之间传输的数据的损坏（它的编码原理和工程应用见 2.5 Cyclic Redundancy Check 应用原理）。

在业务层，其实已经将附件的 CRC 校验码计算过了，在协议层，只需要更新一下协议类的类变量 message_crc_ 就可以了。之所以将 CRC 编码函数和更新类变量 message_crc_ 到函数分开，是因为附件的 CRC 校验码可能在不同的业务代码中被分别编码，在最后进行编码发送的应该是最终的结果，因此为了减少耦合，两者应该区分开。

之后需要将附件和计算获得附件 CRC 校验码都进行二进制编码，封装到二进制信息中，进行数据埋点后，信息写入流程全部结束。二进制信息包将会被写入传输层，向目标服务器发送。如代码 3-18 所示为 writeMessageEnd()实现：

代码 3-18 writeMessageEnd()的实现代码

```
// project/include/thrift/protocol/header_binary_protocol.tcc
template<class Transport_, class ByteOrder_>
uint32_t HeaderBinaryProtocolT<Transport_, ByteOrder_>::writeMessageEnd() {
    uint32_t wsize = 0;
    // 1. write attachment, attachment crc use calcaulated before
    std::string attachment;
    // get write_context.attachment
    uint32_t attach_crc = getWriteAttachment(&attachment);
    wsize += writeBinary(attachment, false);
    if (attachment.size() > 0) {updateMessageCrc(attach_crc, attachment.size());}
    // 2. write meaasge crc
    wsize += writeI32(message_crc_, false);
    // all done
    return wsize;
}
```

3.3.6 信息头的读出

信息头读出主要由函数 readHeader()完成，主要目的是一是从信息头中读取信息写入 Block，二是从信息头中读取密钥并鉴权，这两部分功能只有服务端需要，因为对于客户端，Block 的读写是放在 RPC 请求框架之外完成的，这样设计的原因是我们需要对客户端进行错误回滚，具体的实现见 3.5 客户端的实现。如图 3-10 为 readHeader()的流程图：

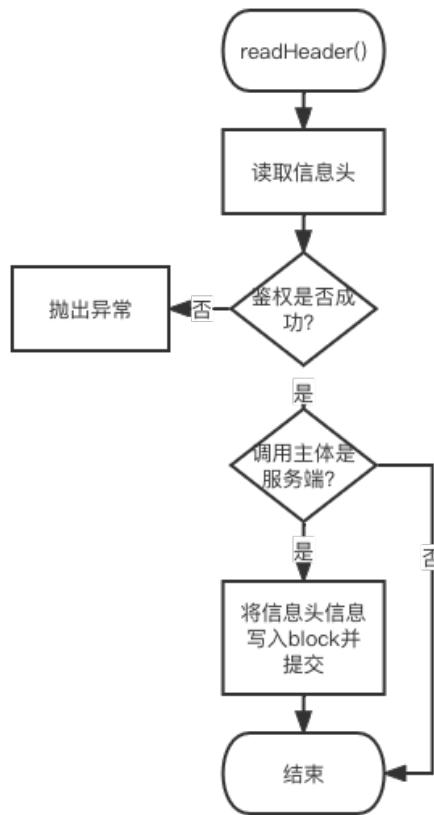


图 3-10 函数 readHeader() 设计流程图

3.3.7 CRC 校验码的校验

CRC 信息校验将在 readMessageEnd() 函数中完成，这是读信息的最后一步。首先从传输层读出附件内容，通过附件计算其 CRC 编码，与从传输层读出的 CRC 编码进行比对。如果两者相同则说明附件传输正常，否则需要抛出异常。

其中需要注意的是，如果设置了段长，则应该将信息分段计算 CRC 校验码。分段计算校验码的函数设计如代码 3-19 所示：

代码 3-19 分段计算 CRC 校验码的实现代码

```

// project/source/thrift/api/operation.cc
uint32_t setReadAttachment(const std::string &attachment, int32_t segment_size) {
    read_context.attachment = attachment;
    read_context.attachment_segment_crcs.clear();
    read_context.attachment_crc = 0;
    uint32_t attachment_len = attachment.length();
    for (uint32_t offset = 0; offset < attachment_len; offset += segment_size) {
        uint32_t segment_crc = UTILS::crc32::Value(attachment.data() + offset,
                                                    std::min(attachment_len - offset, (uint32_t)segment_size));
        read_context.attachment_segment_crcs.emplace_back(segment_crc);
        read_context.attachment_crc = UTILS::crc32::Combine(read_context.attachment_crc,
  
```

```

    segment_crc::min(attachment_len - offset, (uint32_t) segment_size));
    return read_context.attachment_crc;
}

```

其中，将段长作为偏移量对附件进行遍历，计算段 CRC 校验码后，与整体校验码进行合并，输出最终的校验码。

如果未设置段长，则直接计算 CRC 校验码，计算 CRC 校验码的实现代码如代码 3-20 所示：

代码 3-20 计算 CRC 校验码的实现代码

```

// project/source/thrift/api/operation.cc
uint32_t setReadAttachment(const std::string &attachment) {
    read_context.attachment = attachment;
    read_context.attachment_crc = UTILS::crc32::Value(read_context.attachment.data(),
                                                       read_context.attachment.size());
    return read_context.attachment_crc;
}

```

如果段长不合法，则抛出异常。可以遇见的段长错误有多种，如数值为负，`string` 类型转 `int` 类型可能出现的无意义转换错误，超出 `int32` 位数值的超范围错误。本部分流程实现如代码 3-21 所示：

代码 3-21 段长错误处理

```

// project/include/thrift/protocol/header_binary_protocol.tcc
if (!segment_size_s.empty()) {
    int32_t segment_size = 0;
    try { segment_size = std::stoi(segment_size_s); }
    catch (std::invalid_argument const &e) {
        throw ::apache::thrift::TApplicationException(
            ::apache::thrift::TApplicationException::UNKNOWN,
            "invalid segment size");
    }
    catch (std::out_of_range const &e) {
        throw ::apache::thrift::TApplicationException(
            ::apache::thrift::TApplicationException::UNKNOWN,
            "segment size out of range");
    }
    if (segment_size < 0) {
        throw ::apache::thrift::TApplicationException(
            ::apache::thrift::TApplicationException::UNKNOWN,
            "invalid segment size");
    } else if (segment_size == 0) { attach_crc = setReadAttachment(attachment);
    } else { attach_crc = setReadAttachment(attachment, segment_size);
    }
} else { attach_crc = setReadAttachment(attachment);
}

```

获得计算出的 CRC 校验码后，应与从传输层读出的 CRC 校验码进行比对，如果不同，则抛出异常，协议类中对信息的读取结束。比较计算所得的 CRC 校验码和传输层读取的 CRC 校验码的代码如代码 3-22 所示：

代码 3-22 比较计算所得的 CRC 校验码和传输层读取的 CRC 校验码

```
// project/include/thrift/protocol/header_binary_protocol.tcc
int32_t check_crc = 0;
result += readI32(check_crc, false);
if ((uint32_t)check_crc != message_crc_) {throw ::apache::thrift::TApplicationException(
    ::apache::thrift::TApplicationException::MISSING_RESULT, "crc not matched");}
```

3.4 鉴权的实现

一般情况下，客户端与服务端之间的鉴权由业务方完成。本课题设计将在协议层进行鉴权。鉴权系统依赖于密钥，密钥白名单，以及密钥的加解密算法。主要数据流程如下。设计 ServerInfo 为存储相关服务器及鉴权信息的数据结构。整体的框架设计如图 3-11 所示：

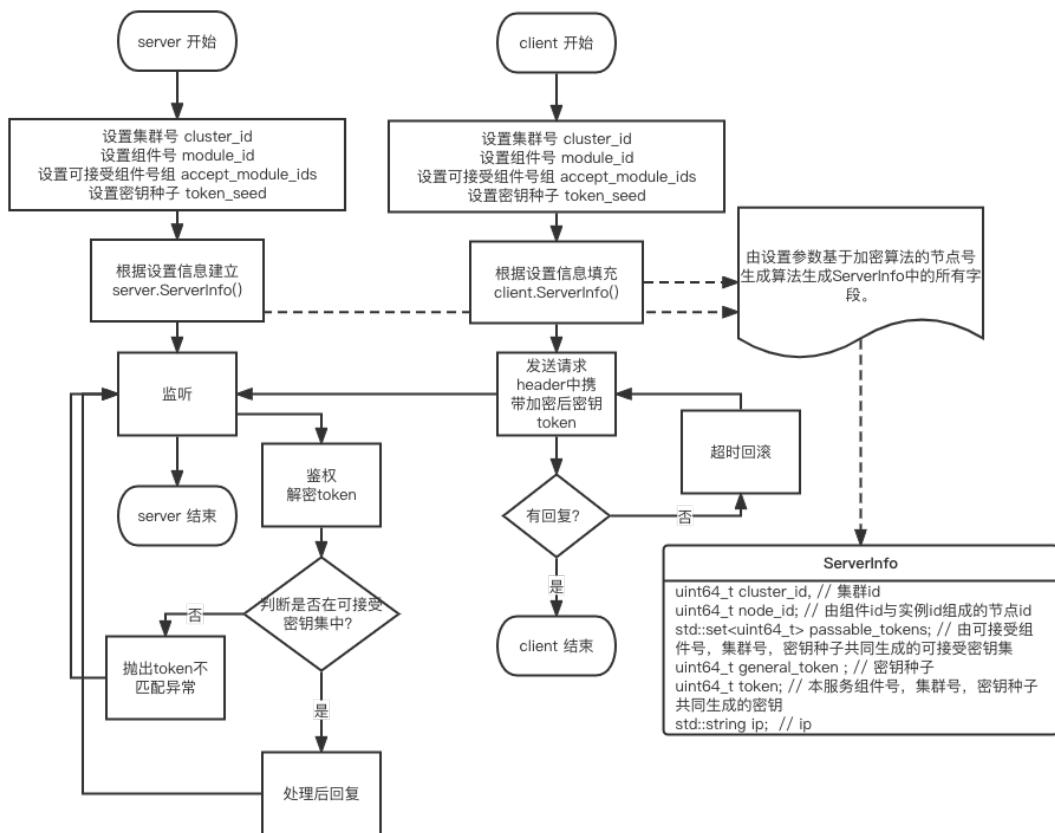


图 3-11 鉴权系统整体框架

3.4.1 密钥编码解码

密钥由集群号，组件号，密钥种子通过位计算编码生成。一共 64 位，其中高位 22 位为集群 id，中位 10 位为组件 id，低位 32 位为密钥种子，设置 64 位目的为便于十六进制存储，二进制传输。Token 的结构设计如图 3-12 所示：

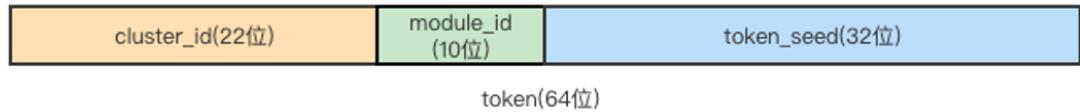


图 3-12 token 结构

编码依赖位运算实现，编码函数如代码 3-23 所示：

代码 3-23 密钥编码函数

```
// project/source/thrift/api/token_utils.h
uint64_t MakeTokenByPack(uint64_t cluster_id, uint32_t module_id, uint32_t token_seed) {
    return (((uint64_t)cluster_id & 0x000000000003ffffULL) << 42)
        | (((uint64_t)module_id & 0x0000000000000003ffULL) << 32)
        | (uint64_t)token_seed;
}
```

解码函数也是通过位运算完成，集群 id 的解码函数如 3-24 所示：

代码 3-24 集群 id 的解码函数

```
// project/source/thrift/api/token_utils.h
uint64_t ParseClusterIdFromToken(uint64_t token) {
    return token >> 42;
}
```

组件 id 的解码函数如代码 3-25 所示：

代码 3-25 组件 id 的解码函数

```
// project/source/thrift/api/token_utils.h
uint32_t ParseModuleIdFromToken(uint64_t token) {
    return (token >> 32) & (0x3ffULL);
}
```

密钥种子的解码函数如代码 3-26 所示：

代码 3-26 密钥种子的解码函数

```
// project/source/thrift/api/token_utils.h
uint32_t ParseTokenSeedFromToken(uint64_t token) {
    return token & (0xffffffffULL);
}
```

本设计的基础为默认不同集群不进行交流。当不需要依据集群分隔服务，可以将集群信息移出密钥。集群信息的剔除函数如代码 3-27 所示：

代码 3-27 集群信息的剔除

```
// project/source/thrift/api/token_utils.h
uint64_t MaskOutClusterId(uint64_t token) {
    return token & 0x000003ffffffffULL;
}
```

3.4.2 设置白名单

密钥由集群号，组件号，密钥种子通过位计算编码生成。设置服务时需要设置信任组件列表，当请求来自同一模块的可信任组件，并拥有相同的密钥种子时，则说明本请求来自白名单，可以通过鉴权。将可信任密钥存储在 ServerInfo 的 passable_token 中，生成本列表的操作如代码 3-28 所示：

代码 3-28 可通过密钥列表生成

```
// project/source/thrift/api/operation.cc
for (auto &accept_module_id: accept_modules) {
    uint64_t passable_token = MakeTokenByPack(cluster_id, accept_module_id, token_seed);
    std::unique_lock<std::shared_mutex> lock(g_token_mutex);
    g_server_info.passable_tokens.insert(passable_token);
    LOGF(INFO, "accept cluster_id {}, token_seed {}, module_id {}, passable_token {}", cluster_id,
          token_seed, accept_module_id, passable_token);
}
```

其中，g_token_mutex 是读写锁，用以保护对 passable_token 更改的线程安全。读/写锁，也称为“共享互斥锁”，是用于控制计算机程序并发的同步机制。通过 std::shared_mutex^[9] 实现。

3.4.3 鉴权

在协议层进行鉴权，设置在函数读取信息头的函数中。如代码 3-29 为协议层鉴权部分操作：

代码 3-29 协议层鉴权

```
// project/include/thrift/protocol/header_binary_protocol.tcc
template<class Transport_, class ByteOrder_>
uint32_t HeaderBinaryProtocolT<Transport_, ByteOrder_>::readHeader(const TMessageType
messageType) {
    // 1. read header from rpc message
    /* ... read header operations. */
    header.token,
    // 2. check authority
    uint64_t token = header.token;
    if (messageType == ::apache::thrift::protocol::T_CALL &&
        (uint64_t)header.token != getServerInfo().general_token) {
        // if token(packed) doesn't exist in passable_token
        if (!RpcController::ValidateToken(token)) {
            LOG_F(WARN,
                  "header token not matched : token {}(cluster_id {}, token_seed {}, module_id {})
trace_id {} client_ip {}",
                token, ParseClusterIdFromToken(token), ParseTokenSeedFromToken(token),
                ParseModuleIdFromToken(token),
                (uint64_t)header.trace_id,
                UTILS::NetUtils::InetUtoa(header.client_ip));
            throw ::apache::thrift::TApplicationException(
```

```

        ::apache::thrift::TApplicationException::MISSING_RESULT,
        "token not matched");
    }
}

// 3. ... other operations
}

```

需要注意的是触发鉴权的条件是当信息类型为请求，且进行请求读操作。即当本服务为服务端服务时，会进行鉴权。同时设定了边缘情况简化，如当集群号和信任组件号都未进行设置时，只会比较密钥种子部分，减少对信任密钥列表的读操作。

同样使用读写锁保证线程安全，使用迭代器确认密钥是否在可信任密钥列表中。当鉴权失败，进行信息埋点，服务端抛出异常。其中，密钥的合法性鉴定函数是 ValidateToken()，它的实现如代码 3-30 所示：

代码 3-30 密钥合法性鉴定函数

```

// project/source/thrift/api/operation.cc
bool RpcController::ValidateToken(uint64_t token) {
    // used to make g_server_info.passable_tokens thread-safe
    std::shared_lock lock(g_token_mutex);
    return g_server_info.passable_tokens.find(token) != g_server_info.passable_tokens.end();
}

```

3.5 客户端的实现

客户端的设计是基于 C++ 的面向对象特性设计的，3.5.1 节将总体介绍客户端的类继承结构，剩余的章节将自顶向下介绍每一层的设计，设计的目的，以及它们实现的功能。

3.5.1 客户端类继承结构

客户端类的继承关系如图，其中 Client 类为基类，定义了客户端的基础功能：连接函数，断开链接函数，判断连接是否正常函数。它们都是虚函数，将在 ThriftClient 中被实现。ThriftClient 类是比较重要的类，在这里加入了我们设计的一些客户端泛用的功能，其中最重要的是模版函数 RequestSchema()。RPC 请求的入口函数将被作为参数传入 RequestSchema()，我们可以将此入口函数封装到 try-catch 框架中去，进而实现错误的捕捉和回滚。同时，需要在这一层实现对客户端存储调用链信息的 Block 的写入和读出。与客户端不同的是，服务端对 Block 的读写在协议层，具体设计和原因见 3.3.6 信息头的读出。

EchoClient() 是实现类，会实现 Thrift 定义的 RPC 函数。EchoClient() 类将在主函数中被实例化，通过 client 实例进行请求的发送和回复的处理。如图 3-13 为客户端类的继承关系：

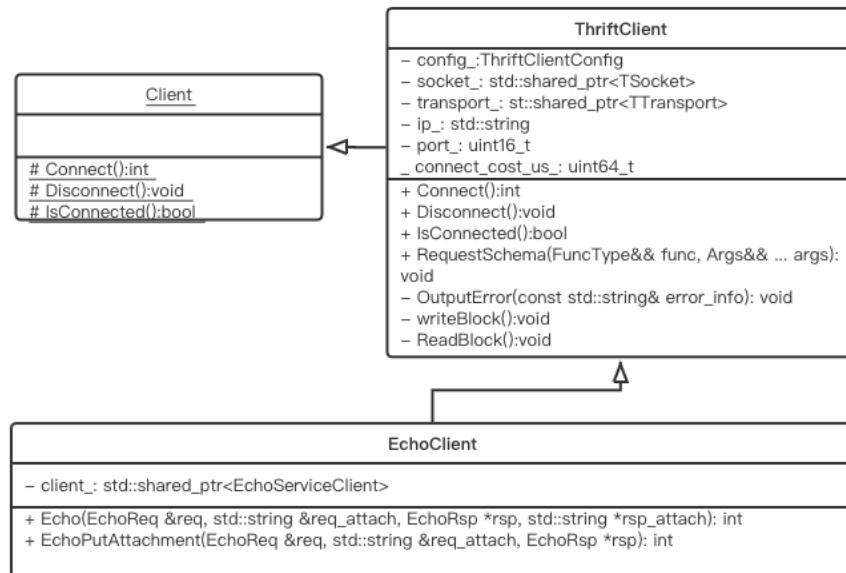


图 3-13 客户端类的继承关系

3.5.2 客户端在主函数中的实例化

客户端主函数设计流程图如图 3-14 所示：

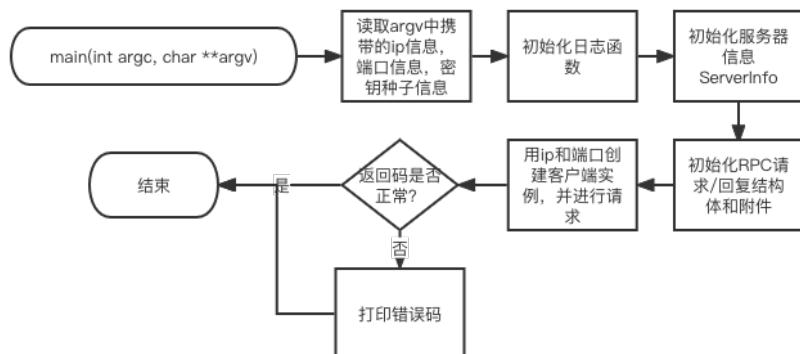


图 3-14 客户端主函数流程图

1. 读取 main 函数参数 argv 中信息

argv 参数的读取操作如代码 3-31 所示，将 ip 信息，port 信息，密钥种子信息在编译时作为参数传入，存入 main 函数参数 argv 中。

代码 3-31 argv 参数的读取

```

// project/source/client/client.cc
// deal with config
if(argc < 4) {
    std::cout << argv[0] << " + ip " << " + port " << " + token " << std::endl;
    return -1;
}

```

```

}
std::string ip = argv[1];
int port = atoi(argv[2]);
uint32_t token = atoi(argv[3]);

```

2. 初始化日志组件

初始化日志组件操作见代码 3-32，使用 SPDLogSink 组件进行日志管理。组件具体使用情况，见 2.7 SPDLogSink 组件应用原理。

代码 3-32 初始化日志组件代码

```

// project/source/client/client.cc
LOGGER::LogInstance log_instance;
log_instance.InitNormalLog("server1", "server1_log");
log_instance.SetSplitSize(1024 * 1024 * 1024);
log_instance.SetFilterLevel("TRACE");

```

3. 初始化服务器信息，请求/回复的消息体和附件

在这里设置了客户端所在服务器的集群 id，客户端服务的组件 id，可接受的组件 id 和密钥种子 token。这些信息会被存储到 ServerInfo 中，并在 setServerInfo() 函数中被编码为密钥，在协议层被写入信息头，随请求发送至服务端服务器，由服务端进行鉴权操作（生成密钥过程与鉴权细节见 3.4 鉴权的实现）。初始化注册客户端需要的数据结构的操作如代码 3-33 所示：

代码 3-33 初始化注册客户端需要的数据结构

```

// project/source/client/client.cc
uint64_t cluster_id = 0;
std::vector<uint32_t> accept_modules={3};
::apache::thrift::protocol::setServerInfo(cluster_id, 1, accept_modules, token);
EchoRequest request;
request.content = "helloworld";
EchoResponse response;
std::string req_attachment(1024 * 1024, 'a');
std::string rsp_attachment;
auto start = std::chrono::high_resolution_clock::now();
EchoClient client(ip, port);

```

4. 创建客户端实例发送 RPC 请求并处理返回码

根据设置的 ip 信息和端口信息进行客户端的实例化，并调用 RPC 请求。如果返回码出错，进行打印并断言 false，这一步的作用是为了防止 gcc 的-Wparentheses 提出的错误信息被覆盖。如果返回码不出错，则客户端流程结束。注册客户端操纵如代码 3-34 所示：

代码 3-34 注册客户端

```

// project/source/client/client.cc
EchoClient client(ip, port);
int ret = client.Echo(request, req_attachment, &response, &rsp_attachment);

```

```

if(ret != 0) {
    LOG_F(ERROR, "echo failed ret{}.", ret);
    std::cout << "echo failed ret " << ret << std::endl;
    assert(false);
}
return 0;

```

3.5.3 客户端在 EchoClient 层的实现

类 EchoClient 的名字是根据我们设定的 RPC 接口设定的，在本类中，需要实现我们设定的 RPC 接口函数。以本项目为例，在 echo.thrift 中我们设定了三个函数，如代码 3-35 所示：

代码 3-35 .thrift 设定函数

```

// project/source/client/client.cc
service EchoService {
    EchoResponse Echo(1:EchoRequest request);
    EchoResponse EchoPutAttachment(1:EchoRequest request);
    EchoResponse EchoGetAttachment(1:EchoRequest request);
}

```

基于这个.thfit 文件，通过 Thrift 工具已经生成了对应的框架代码，这三个函数的具体实现则要在 EchoClient 类中实现，以 Echo() 函数为例，实现如下：

1. 封装 CRC 校验码

函数 PackRpcMessage() 可以将附件信息编码为 CRC 校验码，在协议层将写入传输层，随请求发送至服务端服务器，服务端也将计算所收到附件的 CRC 校验码，与收到的 CRC 校验码进行比对，判断附件内容是否在传输过程中受损。对 CRC 校验码的封装如代码 3-36 所示：

代码 3-36 对 CRC 校验码的封装

```

// project/source/client/client.cc
class EchoClient : public UTILS::ThriftClient {
public:
    int Echo(const EchoRequest &request,
// 1. pack request message
        uint32_t attachment_crc = apache::thrift::protocol::PackRpcMessage((void *) &request,
        request_attachment);
// ... other operations
}

```

函数 PackRpcMessage() 实现如代码 3-37 所示：

代码 3-37 CRC 校验码封装函数的定义

```

// project/source/thrift/api/operation.cc
uint32_t PackRpcMessage(void *message, const std::string &attachment) {
    (void) message;
    write_context.attachment = attachment;
}

```

```

    write_context.attachment_crc = UTILS::crc32::Value(write_context.attachment.data(),
                                                       write_context.attachment.size());
    return write_context.attachment_crc;
}

```

2. 发送请求

发送请求的函数如代码 3-38 所示：

代码 3-38 客户端对请求的发送

```

// project/source/client/client.cc
class EchoClient : public UTILS::ThriftClient {
public:
int Echo(const EchoRequest &request,
// ... other operations
// 2. send request
auto func = std::bind(&EchoServiceClient::Echo, client_, std::ref(*response), std:: cref(request));
int ret = RequestSchema(func) < 0 ? -1 : response->code;
if (ret != 0) {
    return -1;
// ... other operations
}

```

这里的函数 RequestSchema(func) 的参数 func 是一个函数，它被绑定为框架函数 EchoServiceClient::Echo()，即 RPC 请求的入口，它的实现如代码 3-39 所示：

代码 3-39 RPC 请求入口实现

```

void EchoServiceClient::Echo(EchoResponse& _return, const EchoRequest& request)
{
    send_Echo(request);
    recv_Echo(_return);
}

```

其中，send_Echo(request) 实现如代码 3-40 所示。这里调用了协议层的 writeMessageBegin() 和 writeMessageEnd()，并将消息写入了传输层：

代码 3-40 发送请求函数 send_Echo 的实现

```

void EchoServiceClient::send_Echo(const EchoRequest& request)
{
    int32_t cseqid = 0;
    oprot_->writeMessageBegin("Echo", ::apache::thrift::protocol::T_CALL, cseqid);
    EchoService_Echo_pargs args;
    args.request = &request;
    args.write(oprot_);
    oprot_->writeMessageEnd();
    oprot_->getTransport()->writeEnd();
    oprot_->getTransport()->flush();
}

```

3. 读取 CRC 校验码

进行到这一步时，客户端已经收到并处理了回复，完成了整个 RPC 请求流程。使用函数 ParseRpcMessage() 将读取 CRC 校验码并进行存储，一切顺利则返回状态

码 0，表明客户端的请求正常收到了回复，并顺利处理完成。读取 CRC 校验码的实现如代码 3-41 所示：

代码 3-41 接受回复后读取服务端回复的 CRC 校验码

```
// project/source/client/client.cc
class EchoClient : public UTILS::ThriftClient {
public:
    int Echo(const EchoRequest &request,
    // ... other operations
    // 3. parse response message
    uint32_t recv_attachment_crc = apache::thrift::protocol::ParseRpcMessage(response,
    response_attachment);
    return 0;
}
```

3.5.4 客户端在 ThriftClient 层的实现

在 ThriftClient 类中，主要需要实现几个成员函数：

1. Connect() 连接函数

ThriftClient 类的成员函数 Connect() 实现如代码 3-42 所示：

代码 3-42 ThriftClient 类的成员函数 Connect() 实现

```
// project/source/rpc/thrift_client.cc
int ThriftClient::Connect() {
    int ret = 0;
    UTILS::Timer t;
    try {transport_->open();} catch (TException& e) {
        OutputError(std::string("Connect failed, exception = ") + e.what());
        Disconnect();
        ret = -1;
    }
    const uint64_t kTimeInterval100Ms = 100*1000;
    connect_cost_us_ = t.GetIntervalUs();
    if (connect_cost_us_ > kTimeInterval100Ms) {
        LOG_F(WARN, "[Alarm:RpcConnectHugeLatency] Cost={} Ret={} ip={} port={}",
t.GetIntervalUs(), ret, ip_, port_);
    }
    return ret;
}
```

需要打开传输层连接，如果出错则进行断连并返回错误状态码。同时需要记录连接时间，如连接时间过长则需要发出警告。使用时间类 UTILS::Timer，本类的设计如图 3-15 所示：

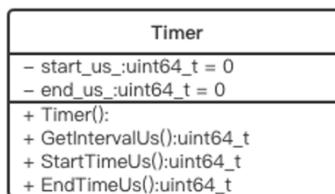


图 3-15 时间操作类

在时间类初始化时会调用初始化函数 Timer(), 将类变量开始时间 start_us_ 进行设定。当连接完成, GetIntervalUs() 可以获得从开始时间到结束时间的差值。

2. Disconnect() 断连函数与 IsConnected() 判断连接状态函数

代码 3-43 ThriftClient 类的成员函数 Disconnect() 和 IsConnected() 的实现

```
// project/source/rpc/thrift_client.cc
void ThriftClient::Disconnect() {transport_->close();}
bool ThriftClient::IsConnected() const {return transport_->isOpen();}
```

断连和判断是否连接直接调用传输层 API 即可。

3. RequestSchema() 请求模板函数

在本函数中实现回滚和重试次数的功能, 设计流程图如下:

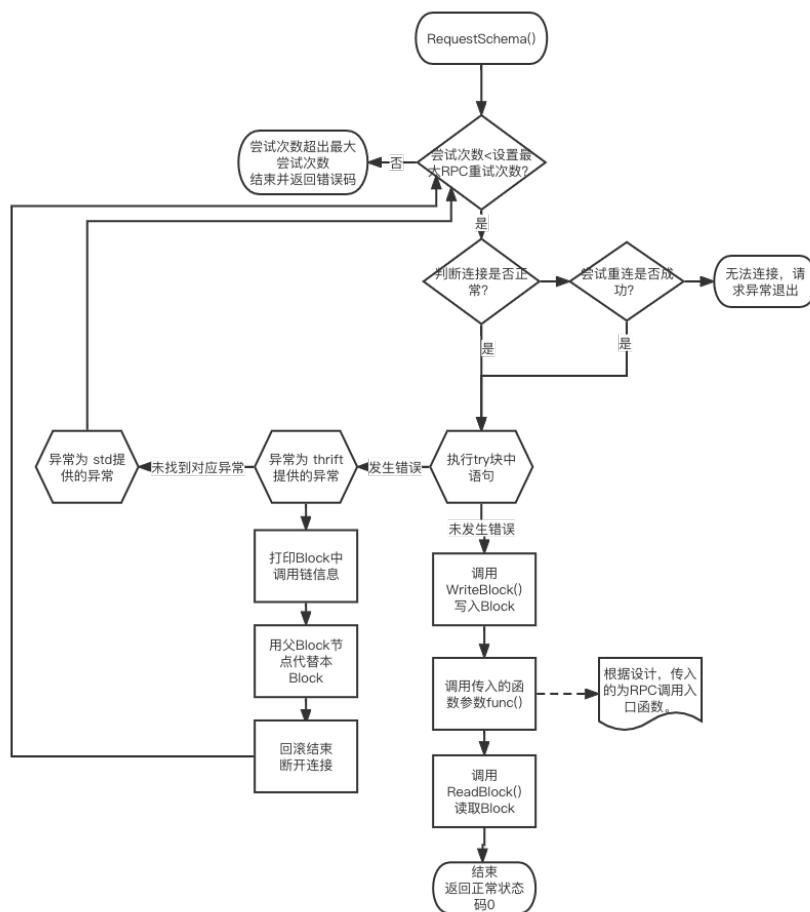


图 3-16 RequestSchema() 请求函数模板

经过查阅资料和研究^[10], 我们发现, 当一个客户端请求发出时, 如果服务器断开连接那么他就不会执行 ReadBlock 的内容, 导致了一个 Block 无法被闭合, 乃至不能自动回滚聚合到一次(发送->请求)的链路。同时这个 Block 会一直留在线程的上下文里, 被放入线程池。在下次发出请求拿到这个线程的时候, 我们会重新拿到

这个垃圾 Block，以为他是正常的闭合 Block，拓展子节点写入新的 Block。于是，垃圾 Block 就被留在了线程的上下文中永远得不到回滚，导致日志输出不符合预期因此我们需要进行回滚操作。

首先设置循环控制重试次数，由变量 `max_rpc_retries` 控制，对于 Thrift 错误，会回滚信息后，断开连接并尝试重连，对其他错误，不会断开连接，直接进行循环。尝试重连时，如果失败，会直接结束请求，返回异常退出错误码。这两种重试是不一样的，前者由 `ThriftClient` 类代码控制，后者由传输层控制，如代码 3-44 所示：

代码 3-44 重试次数上限变量 `max_rpc_retries` 的声明

```
// project/include/rpc/thrift_client.h
struct ThriftClientConfig {uint32_t max_rpc_retries{2};};
```

初始化配置实例，并在初始化客户端的时候传入，设计重试次数上限变量的声明如代码 3-45 所示：

代码 3-45 重试次数上限变量 `max_rpc_retries` 的声明

```
// project/source/client/client.cc
const UTILS::ThriftClientConfig kThriftConfig = {}(const char *output) { LOG_F(INFO,
    "ThriftOutput: {}", output); }, 3, 3, 1000, 1000, 10000, false};
EchoClient(const std::string &ip, int port)
    : ThriftClient(ip, port, g_echo_constants.service_name, kThriftConfig),
      client_(std::make_shared<EchoServiceClient>(protocol_)) {}
```

在初始化客户端时，会调用 `SetConfig` 函数，将按照传入的配置进行设置。`SetConfig` 函数如代码 3-46 所示，它也是 `ThriftClient` 的成员函数。

代码 3-46 `SetConfig` 函数

```
// project/include/rpc/thrift_client.cc
void ThriftClient::SetConfig(const ThriftClientConfig& config) {
    if (config.output_func != nullptr) {
        GlobalOutput.setOutputFunction(config.output_func);
    }
    socket_->setMaxRecvRetries(config.max_recv_retries);
    socket_->setRecvTimeout(config.recv_timeout_ms);
    socket_->setSendTimeout(config.send_timeout_ms);
    socket_->setConnTimeout(config.conn_timeout_ms);
    socket_->setKeepAlive(config.keep_alive);
    output_func_ = config.output_func;
    config_ = config;
}
```

在 try 代码块中，`func` 为在框架函数 `EchoServiceClient::Echo()`，它是 rpc 请求的入口，我们将在调用它之前调用写 Block 函数，在 rpc 请求结束后，调用读 Block 函数。如果在这个过程中出现错误，将被 catch。如果是 `TException` 中定义的错误，将进行回滚，并将 Block 指针指回父 Block 节点，并同时断开连接，传输层将重新

进行连接，直到达到 `max_recv_retries`。如果是其余的错误，只会结束本次循环，并重新尝试发送 RPC 请求，直到重试达到 `max_rpc_retries`。代码实现如 3-47 所示：

代码 3-47 回滚重连实现

```
//project/include/rpc/thrift_client.h
template<class FuncType, class... Args>
int RequestSchema(FuncType&& func, Args&& ... args) {
    for (uint32_t n_tries = 0; n_tries < config::max_rpc_retries + 1; ++n_tries) {
        if (!IsConnected()) {if (Connect() < 0) {return -2; }x}
        try {
            WriteBlock();
            func(std::forward<Args>(args)...);
            ReadBlock();
            return 0;
        } catch (TException &e) {
            OutputError(std::string("Thrift exception caught, exception = ") + e.what());
            // catch exception, rollback block.
            std::cout << "Thrift exception caught, exception = " << e.what() << "t_id:" << UTILS::GetThreadContext()->block->trace_id() << std::endl;
            UTILS::GetThreadContext()->block =
UTILS::GetThreadContext()->block->parent_block();
            puts("rollback");
            Disconnect();
        } catch (std::exception &e) {
            OutputError(std::string("Std exception caught, exception = ") + e.what());
        }
    }
    return -1;
}
```

4. WriteBlock()写入调用链信息函数

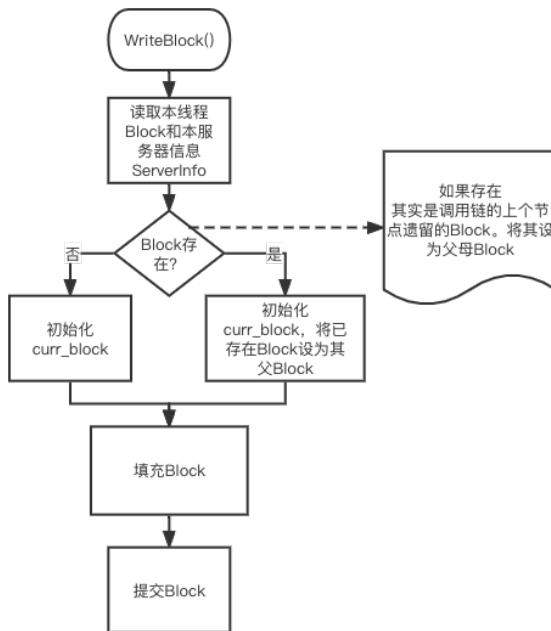


图 3-17 写入 Block 函数流程

写入 Block 的函数流程如图 3-17 所示，在初始化本 RPC 节点的 Block 之前应先查看本线程是否已有 Block 存在，如果存在则说明其为本节点的父节点。用服务器信息类以及 RpcContext 类中的信息填充 Block 后提交。需要填充的字段有 trace_id, block_id, parent_block_id 等。

5. ReadBlock()读出调用链信息函数

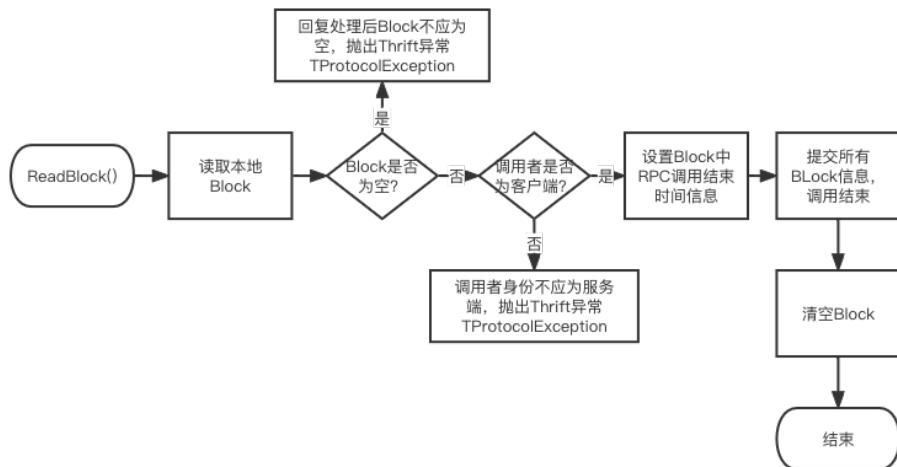


图 3-18 读出 Block 函数流程

读出 Block 信息的流程如图 3-18，需要判断 Block 是否正常，如果异常则抛出异常，正常则填入调用结束，并提交 Block 信息。提交之后清空 Block，客户端调用流程结束。

3.5.5 客户端在基类 Client 层的定义

代码 3-48 客户端基类定义

```
// project/include/rpc/client.h
class Client {
public:
    virtual ~Client() = default;
    virtual int Connect() = 0;
    virtual void Disconnect() = 0;
    virtual bool IsConnected() const = 0;
};
```

在 Client 基类定义 Client 的基础函数，如代码 3-48 所示，将其定义为虚函数，提醒开发者针对不同的传输层实现这几个函数，本项目中的这几个函数在 ThrifitClient 类中实现。

3.6 服务端的实现

3.6.1 服务端在主函数中的实例化

主函数的实例化流程图如下：

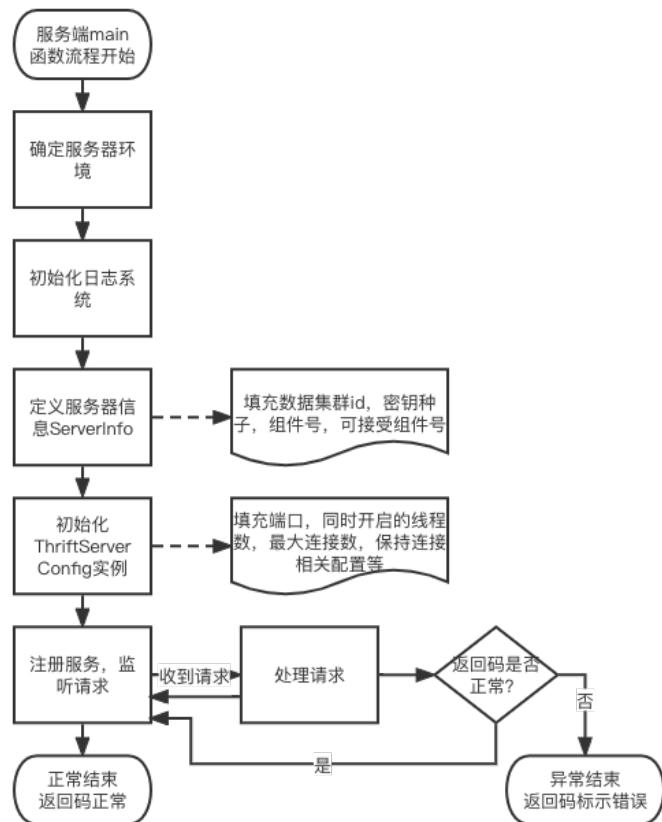


图 3-19 服务端主流程

1. 服务器系统判定

判定系统有助于出现错误时检查系统原因，通过访问系统宏定义实现，如代码 3-49 所示：

代码 3-49 服务端记录系统

```
// project/source/server/server.cc
std::cout << "OS is: "
#ifdef __linux__
<< "Linux"
#elif __APPLE__
<< "MAC"
#endif
<< std::endl;
std::cout << "Starting the server..." << std::endl;
```

2. 初始化日志组件

使用 SPDLogSink 组件进行日志管理，组件具体使用情况，见 2.7 SPDLogSink 组件应用原理。初始化日志组件如代码 3-50 所示：

代码 3-50 服务端初始化日志组件

```
// project/source/server/server.cc
LOGGER::LogInstance log_instance;
log_instance.InitNormalLog("server2", "server2_log");
log_instance.SetSplitSize(1024 * 1024 * 1024);
log_instance.SetFilterLevel("TRACE");
```

3. 填充 ServerInfo 与 ThriftServerConfig

前者保存服务器信息，主要有集群 ID，密钥种子，组件 ID，可接受组件列表等信息（具体的数据结构见 3.2.4 服务信息存储结构 ServerInfo 类）。后者存储服务器的一些有关 Thrift 架构的配置。其结构如代码 3-51 所示：

代码 3-51 ThriftServerConfig 类成员变量

```
// project/include/rpc/thrift_server.h
struct ThriftServerConfig {
    std::string ip{"0.0.0.0"};
    uint16_t port{0};
    OutputFunc output_func=nullptr;
    size_t io_thread_count{1};
    size_t work_thread_count{16};
    size_t max_connection_count{0};
    uint32_t recv_timeout_ms{0}; // setsockopt : SO_RCVTIMEO
    uint32_t send_timeout_ms{0}; // setsockopt : SO_SNDFTIMEO
    uint32_t tcp_recv_buffer{0}; // setsockopt : SO_RCVBUF
    uint32_t tcp_send_buffer{0}; // setsockopt : SO_SNDBUF
    bool keep_alive{true}; // setsockopt : SO_KEEPALIVE
    int keep_alive_time = 10; // setsockopt : TCP_KEEPIDLE
    int keep_alive_interval = 10; // setsockopt : TCP_KEEPINTVL
    int keep_alive_probes = 5; // setsockopt : TCP_KEEP_CNT
};
```

其中，除了 port 需要在使用本结构体时定义，其余的变量可以使用默认值。output_func 定义输出到什么变量里，如设置输出到 stdout 或 stderr。io_thread_count 为 io 线程数，如果需要传输大量的数据，应多开几个线程。work_thread_count 为处理线程数，最好不要超过服务器 cpu 核数。max_connection_count 为最大连接数，如果超过了，则新的连接将被关闭。剩余的设定选项为 Linux 系统的 socket 设定选项。

在实验中，这两个数据结构的实例化如代码 3-52 所示：

代码 3-52 ServerInfo 和 ThriftServerConfig 的实例化

```
// project/source/server/server.cc
uint64_t cluster_id = 0;
int32_t token_seed = 6666;
```

```

uint32_t module_id = 2;
std::vector<uint32_t> accept_module_ids={1,2};
::apache::thrift::protocol::setServerInfo(cluster_id, module_id, accept_module_ids, token_seed);
// token seed used for making encrypted token
int port = 9090;
UTILS::ThriftServerConfig thrift_server_config;
thrift_server_config.io_thread_count = 3;
thrift_server_config.port = port;

```

4. 注册服务

服务端的注册如代码 3-53 所示：

代码 3-53 服务端注册服务

```

// project/source/server/server.cc
auto server = std::make_shared<UTILS::ThriftServer>(thrift_server_config);
auto echo_processor =
std::make_shared<EchoServiceProcessor>(std::make_shared<EchoServiceHandler>());
server->RegisterService(std::make_shared<UTILS::ThriftService>(g_echo_constants.service_name,
echo_processor));

```

首先需要用填充的 `thrift_server_config` 声明类 `ThriftServer`。`ThriftServer` 类的结构如图 3-20 所示，主要定义了 Server 的初始化，注册，开始，停止等函数：

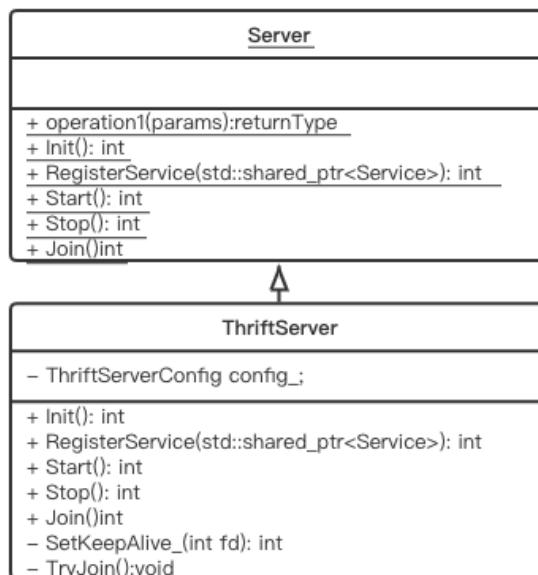


图 3-20 服务器类继承关系

之后，需要声明服务处理器，这部分主要是用类 `EchoServiceHandler` 继承框架代码定义的类 `EchoServiceIf`，实现框架定义的 RPC 函数。继承关系如图 3-21 所示：

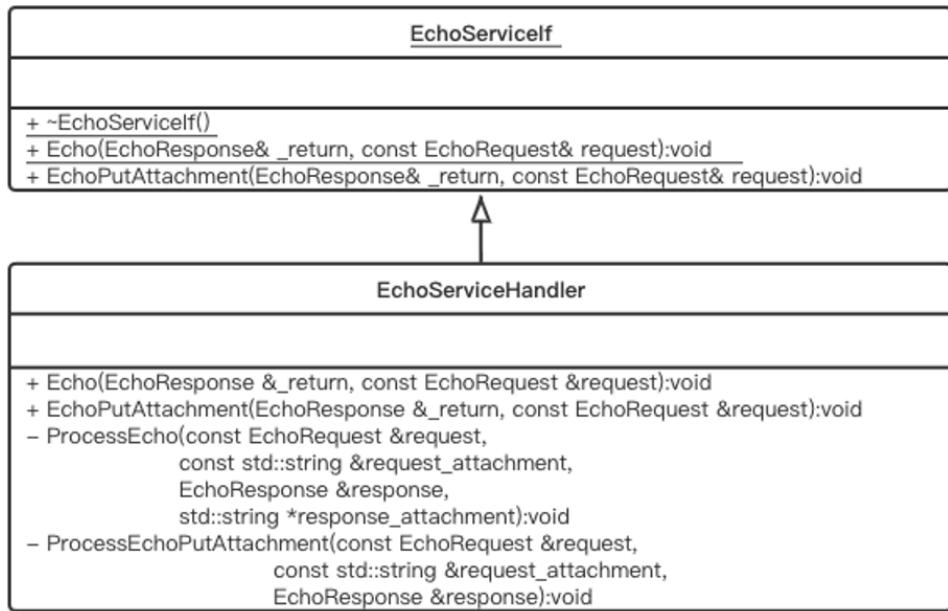


图 3-21 服务处理器类继承关系

用服务器注册服务，服务类 `ThriftService` 的实例化需要用到服务处理器。服务类 `ThriftService` 的继承关系如图 3-22 所示：

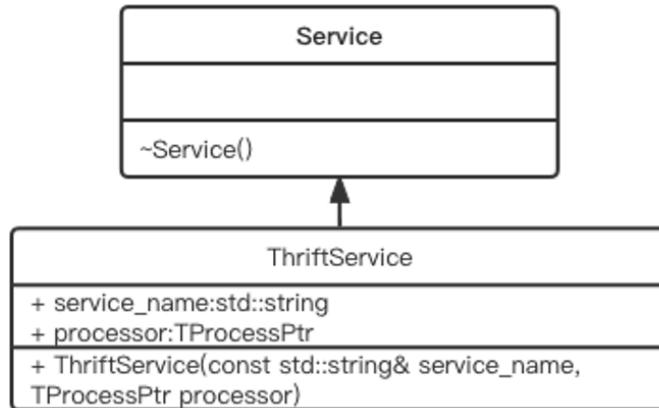


图 3-22 服务类继承关系

注册后，服务将开始阻塞主线程并监听请求。服务结束时，返回码正常则结束，返回码异常则返回错误码结束。服务器处理异常退出如代码 3-54 所示：

代码 3-54 服务端处理异常退出

```
// project/source/server/server.cc
int ret = server->Start();
if (ret != 0) {LOG_F(WARN, "server start failed ret {}", ret); return ret;}
server->Join();
return 0;
```

3.6.2 服务处理类 EchoServiceHandler 的实现

EchoServiceHandler 类继承了框架类 EchoServiceIf，我们需要在 EchoServiceHandler 中实现服务端定义的函数 Echo() 和 EchoPutAttachment()。同时，将具体的数据处理函数作为私有函数，可以减少耦合性。类的具体结构如图 3-23：

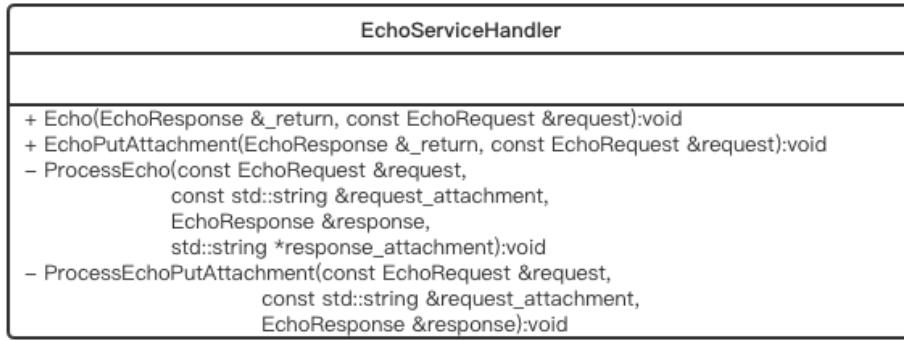


图 3-23 EchoServiceHandler 类的定义

以 EchoPutAttachment 为例，介绍其实现。本服务的目的是将传入的字符串加上特定的字符串。同时，会传入附件，当进行到业务层的时候，其实已经校验过校验码了。在这里需要计算新的 CRC 校验码，并且传入回复中。函数 EchoPutAttachment 的实现见代码 3-55：

代码 3-55 EchoPutAttachment()的实现

```

// project/include/rpc/service.h
void EchoPutAttachment(EchoPutAttachmentResponse &_return, const EchoPutAttachmentRequest &request) {
    RPCLOG_F(INFO, "=====EchoPutAttachment in server begin====req content:{}",
             request.content);
    std::string request_attachment, response_attachment;
    RpcController cntl;
    request_attachment = cntl.GetRequestAttachment();
    uint32_t cal_attachment_crc = UTILS::crc32::Value(request_attachment.data(),
                                                       request_attachment.size());
    ProcessEchoPutAttachment(request, request_attachment, _return);
    RPCLOG_F(INFO, "=====EchoPutAttachment in server end====rsp.content:{} , code:{}",
             _return.content, _return.code );
}

```

这里处理数据，生成回复的函数 ProcessEchoPutAttachment 实现如代码 3-56：

代码 3-56 ProcessEchoPutAttachment()的实现

```

void ProcessEchoPutAttachment(const EchoPutAttachmentRequest &request,
                             const std::string &request_attachment,
                             EchoPutAttachmentResponse &response) {
    response.content = request.content + " PutAttachment";
}

```

第四章 系统测试

4.1 Thrift 测试 demo 的编写

首先编写一个基于一般 Thrift 框架的服务 demo，测试 Thrift 框架在服务器环境的运行情况。在本服务中，客户端将发送一个包含一串字符串的请求，服务端将接受本字符串并返回本字符串的长度以及一个服务状态码。

编写.thrift 文件，这是一个为 Thrift 类型和服务建立的定义接口。在本文件中，需要定义服务的名称，请求与返回的结构，服务的内容。.thrift 文件的定义如代码 4-1 所示：

代码 4-1 .thrift 文件定义

```
// demo/echodemo.thrift
const string service_name = "echo";
struct EchoRequest {
    1: string content;
}
struct EchoResponse {
    1: i32 code;
    2: i32 content;
    3: string err;
}
service EchoService {
    EchoResponse Echo(1:EchoRequest request);
}
```

基于.thrift 文件，自动生成框架代码。生成过程结果如图 4-1 所示：

```
# root @ VM-0-2-debian in /tmp/tmp.t0FvpXFmYX/demo [16:17:55]
$ ls
CMakeLists.txt bin build echodemo.thrift src
# root @ VM-0-2-debian in /tmp/tmp.t0FvpXFmYX/demo [16:17:55]
$ thrift -r --gen cpp echodemo.thrift
# root @ VM-0-2-debian in /tmp/tmp.t0FvpXFmYX/demo [16:18:28]
$ ls
CMakeLists.txt bin build echodemo.thrift gen-cpp src
# root @ VM-0-2-debian in /tmp/tmp.t0FvpXFmYX/demo [16:18:30]
$ cd gen-cpp
# root @ VM-0-2-debian in /tmp/tmp.t0FvpXFmYX/demo/gen-cpp [16:18:58]
$ ls
EchoService.cpp EchoService.h EchoService_server.skeleton.cpp echodemo_constants.cpp echodemo_constants.h echodemo_types.cpp echodemo_types.h
```

图 4-1 自动生成框架代码

分别编写服务端和客户端的代码 server.cc 和 client.cc。首先编写客户端代码：设置套接字，传输协议，之后进行请求逻辑的编写。客户端代码的实现代码 4-2：

代码 4-2 客户端代码的实现

```
// demo/src/client.cc
#include <iostream>
#include <thrift/protocol/TBinaryProtocol.h>
```

```

#include <thrift/transport/TSocket.h>
#include <thrift/transport/TTransportUtils.h>
#include "../gen-cpp/EchoService.h"
using namespace std;
using namespace apache::thrift;
using namespace apache::thrift::protocol;
using namespace apache::thrift::transport;
int main(int argc, char** argv) {
    std::shared_ptr<TTransport> socket(new TSocket("localhost", 9090));
    std::shared_ptr<TTransport> transport(new TBufferedTransport(socket));
    std::shared_ptr<TProtocol> protocol(new TBinaryProtocol(transport));
    EchoServiceClient client(protocol);
    try {
        transport->open();
        EchoResponse rsp;
        EchoRequest req;
        req.content = argv[1];
        cout << "client request is:" << argv[1] << endl;
        client.Echo(rsp, req);
        cout << "response code is:" << rsp.code << endl;
        if(rsp.code == 0){
            cout << "response err is:" << rsp.err << endl;
        }else{
            cout << "response content is:" << rsp.content << endl;
        }
        transport->close();
    } catch (TException& e) {
        cout << "ERROR: " << e.what() << endl;
    }
}

```

之后进行服务端的编写，首先需要编写服务的 handler 代码，实现服务函数，如代码 4-3 所示：

代码 4-3 服务端处理器的实现

```

// demo/src/server.cc
class EchoServiceHandler : public EchoServiceIf {
public:
    EchoServiceHandler() = default;
    void Echo(EchoResponse& rsp, const EchoRequest& req) override {
        try{ std::cout << "server receive request is:" << req.content << std::endl;
            rsp.content = req.content.length();
            rsp.code = 1;
            std::cout << "server sends response content is:" << rsp.content << std::endl;
            std::cout << "server sends response code is:" << rsp.code << std::endl;
        }catch(std::exception& e) {
            std::cout << "server receive request is:" << req << std::endl;
            std::cout << e.what() << std::endl;
            rsp.err = e.what();
            rsp.code = 0;
            std::cout << "server sends response:" << rsp << std::endl;
        }
    }
};

```

之后，需要进行服务端工厂模式与具体处理程序实例声明的编写。Factory 对于连通服务侧的传输，是非常有用。它对于创建每个连接状态也很有用。没有这个 Factory 的话，所有连接都将共享相同的处理程序实例。服务端处理器工厂的实现如代码 4-4 所示：

代码 4-4 服务端处理器工厂的实现

```
// demo/src/server.cc
class EchoServiceFactory : virtual public EchoServiceIfFactory {
public:
    ~EchoServiceFactory() override = default;
    EchoServiceIf* getHandler(const ::apache::thrift::TConnectionInfo& connInfo) override
    {
        std::shared_ptr<TSocket> sock = std::dynamic_pointer_cast<TSocket>(connInfo.transport);
        cout << "Incoming connection\n";
        cout << "\tSocketInfo: " << sock->getSocketInfo() << "\n";
        cout << "\tPeerHost: " << sock->getPeerHost() << "\n";
        cout << "\tPeerAddress: " << sock->getPeerAddress() << "\n";
        cout << "\tPeerPort: " << sock->getPeerPort() << "\n";
        return new EchoServiceHandler;
    }
    void releaseHandler( EchoServiceIf* handler) override {delete handler; }
};
```

最后，初始化服务端监听请求，注册服务器如代码 4-5 所示：

代码 4-5 服务端注册

```
// demo/src/server.cc
int main()
{
    TThreadedServer server(
        std::make_shared<EchoServiceProcessorFactory>(std::make_shared<EchoServiceFactory>()),
        std::make_shared<TServerSocket>(9090), //port
        std::make_shared<TBufferedTransportFactory>(),
        std::make_shared<TBinaryProtocolFactory>());
    cout << "Starting the server..." << endl;
    server.serve();
    cout << "Done." << endl;
    return 0;
}
```

编写 CMakeList.txt，设置编译配置。CMakeList.txt 如代码 4-6 所示：

代码 4-6 CMakeList.txt 编写

```
// demo/CMakeList.txt
cmake_minimum_required(VERSION 2.8)
set(CMAKE_CXX_STANDARD 17)
project(demo)
include_directories(/usr/include /usr/local/include .)
set(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
link_directories(/usr/local/lib /usr/lib)
add_executable(
```

```

client
src/client.cc
gen-cpp/echodemo_constants.cpp
gen-cpp/echodemo_types.cpp
gen-cpp/EchoService.cpp
)
target_link_libraries(client libthrift.a)
target_link_libraries(client pthread)
add_executable(
    server
    src/server.cc
    gen-cpp/echodemo_constants.cpp
    gen-cpp/echodemo_types.cpp
    gen-cpp/EchoService.cpp
)
target_link_libraries(server libthrift.a)
target_link_libraries(server pthread)

```

根据 CMakeList.txt 生成 Makefile 后进行编译, 编译过程及结果如图 4-2 所示:

```

# root @ VM-0-2-debian in /tmp/tmp.t0FvpXFmYX/demo/build [17:34:42]
[$ cmake ..
-- The C compiler identification is GNU 8.3.0
-- The CXX compiler identification is GNU 8.3.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /tmp/tmp.t0FvpXFmYX/demo/build

# root @ VM-0-2-debian in /tmp/tmp.t0FvpXFmYX/demo/build [17:34:47]
[$ make
Scanning dependencies of target client
[ 10%] Building CXX object CMakeFiles/client.dir/src/client.cc.o
[ 20%] Building CXX object CMakeFiles/client.dir/gen-cpp/echodemo_constants.cpp.o
[ 30%] Building CXX object CMakeFiles/client.dir/gen-cpp/echodemo_types.cpp.o
[ 40%] Building CXX object CMakeFiles/client.dir/gen-cpp/EchoService.cpp.o
[ 50%] Linking CXX executable ../../bin/client
[ 50%] Built target client
Scanning dependencies of target server
[ 60%] Building CXX object CMakeFiles/server.dir/src/server.cc.o
[ 70%] Building CXX object CMakeFiles/server.dir/gen-cpp/echodemo_constants.cpp.o
[ 80%] Building CXX object CMakeFiles/server.dir/gen-cpp/echodemo_types.cpp.o
[ 90%] Building CXX object CMakeFiles/server.dir/gen-cpp/EchoService.cpp.o
[100%] Linking CXX executable ../../bin/server
[100%] Built target server

```

图 4-2 编译 Thrift_demo

运行编译成功的可执行文件, 服务端调用见图 4-3, 客户端调用见图 4-4:

```

# root @ VM-0-2-debian in /tmp/tmp.t0FvpXFmYX/demo/bin [17:49:34]
[$ ./server
Starting the server...
Incoming connection
    SocketInfo: <Host: 127.0.0.1 Port: 36562>
    PeerHost: localhost.localdomain
    PeerAddress: 127.0.0.1
    PeerPort: 36562
server receive request is:una
server sends response content is:3
server sends response code is:1

```

图 4-3 服务端调用

```
# root @ VM-0-2-debian in /tmp/tmp.t0FvpXFmYX/demo/bin [17:49:54] C:139
$ ./client una
client request is:una
response code is:1
response content is:3
```

图 4-4 客户端调用

Thrift demo 搭建完毕，文件结构如图 4-5：

```
# root @ VM-0-2-debian in /tmp/tmp.t0FvpXFmYX/demo [17:52:06]
$ tree -L 2
.
├── bin
│   └── client
│       └── server
└── build
    ├── CMakeCache.txt
    ├── CMakeFiles
    │   ├── cmake_install.cmake
    │   └── Makefile
    ├── CMakeLists.txt
    └── echodemo.thrift
        ├── gen-cpp
        │   ├── echodemo_constants.cpp
        │   ├── echodemo_constants.h
        │   ├── echodemo_types.cpp
        │   ├── echodemo_types.h
        │   ├── EchoService.cpp
        │   ├── EchoService.h
        │   └── EchoService_server.skeleton.cpp
        └── src
            └── client.cc
            └── server.cc
5 directories, 16 files
```

图 4-5 Thrift demo 的文件结构

4.2 调用链的设计与实现

为模拟 RPC 调用链的应用，设计调用链，如图 4-6 所示：

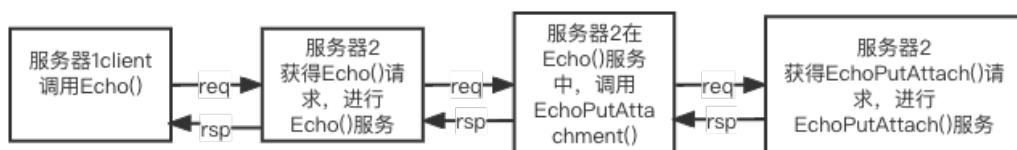


图 4-6 简单调用链的设计

其中 `EchoPutAttachment` 的作用是接受一个字符串，并将其加上字符串“`PutAttachment`”，`Echo` 服务的作用是接受一个字符串，然后发送到 `EchoPutAttachment` 服务中处理字符串后，将处理后的结果字符串再进行处理，加上字符串“`Echo`”。`EchoPutAttachment` 服务将带上一个附件，进行 CRC 校验。调用链信息和普通的日志信息将输出到不同的记录文件中去。

调用链的流程图见图 4-7。这里忽略了具体实现，只表示调用关系，更详细设计实现见第三章设计与实现

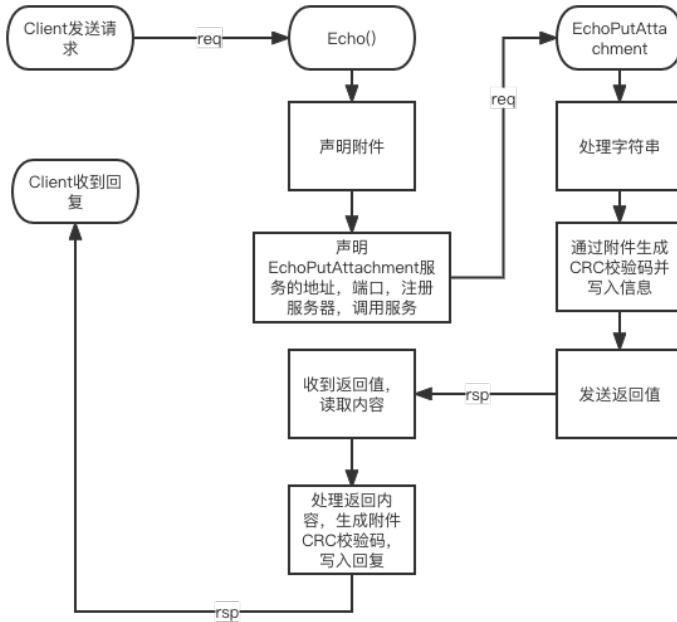


图 4-7 调用链调用关系设计

以 Echo 服务为例，它请求 EchoPutAttachment 的代码实现如代码 4-7 所示：

代码 4-7 Echo 的服务端作为客户端请求 EchoPutAttachment 服务

```

void Echo(EchoResponse &_return, const EchoRequest &request) {
    auto &local_block = UTILS::GetThreadContext()->block;
    // EchoPutAttachment
    auto ip = "127.0.0.1";
    auto port = 9090;
    EchoPutAttachmentRequest request_;
    request_.content = request.content;
    EchoPutAttachmentResponse response;
    std::string req_attachment(1024 * 1024, 'a');
    std::string rsp_attachment;
    EchoClient client(ip, port);
    int ret = client.EchoPutAttachment(request_, req_attachment, &response);
    std::cout << response.content << ":" << response.code << std::endl;
    if (ret != 0) {std::cout << "Put failed ret " << ret << std::endl; }
    // Echo
    // ...
}

```

使用返回值进行下一步操作，读取请求 EchoPutAttachment 服务的回复，将其进行处理，如代码 4-8 所示：

代码 4-8 Echo 的服务端作为客户端处理 EchoPutAttachment 服务的回复

```

void Echo(EchoResponse &_return, const EchoRequest &request) {
    // EchoPutAttachment
    std::string request_attachment, response_attachment;
    uint32_t attachment_crc = apache::thrift::protocol::ParseRpcMessage((void *) &request,

```

```

&request_attachment);
EchoRequest processed_request;
processed_request.content = response.content;
ProcessEcho(processed_request, request_attachment, _return, &response_attachment);
attachment_crc = apache::thrift::protocol::PackRpcMessage((void *) &_return, response_attachment);
std::cout << "end" << std::endl;
}

```

其中具体的处理数据以及包装回复的函数 ProcessEcho 的定义如代码 4-9 所示：

代码 4-9 处理函数 ProcessEcho 定义

```

void ProcessEcho(const EchoRequest &request,
                 const std::string &request_attachment,
                 EchoResponse &response,
                 std::string *response_attachment) {
    response.content = request.content + " Echo";
    *response_attachment = request_attachment;
}

```

4.3 正常流程测试结果

基于所有的设计和实现，测试正常流程下的数据埋入后的调用链信息记录。

客户端的表现如图 4-8 所示：

```

05-20 15:39:06.706642 INFO 2896 log_instance.h:320:InitNormalLog trace_id:NONE Log rotate
enabled result: false
05-20 15:39:06.708024 INFO 2896 client.cc:175:main trace_id:NONE ===Echo request begin.===
05-20 15:39:06.709287 TRACE 2896 thrift_client.h:201:WriteBlock trace_id:62776338759861 Client: WriteBlock()
05-20 15:39:06.709308 TRACE 2896 block.h:89:Submit trace_id:62776338759861 block_id 88
688750488396 parent_block_id 0 start_time 1653032346709 end_time 0 is_callee false annotation
05-20 15:39:06.727217 TRACE 2896 thrift_client.h:229:ReadBlock trace_id:62776338759861 Client: ReadBlock()
05-20 15:39:06.727237 TRACE 2896 block.h:89:Submit trace_id:62776338759861 block_id 88
688750488396 parent_block_id 0 start_time 1653032346709 end_time 1653032346727 is_callee false annotation
05-20 15:39:06.727254 INFO 2896 client.cc:175:main trace_id:NONE ===Echo end normally.===
05-20 15:39:11.593721 INFO 2925 log_instance.h:320:InitNormalLog trace_id:NONE Log rotate
enabled result: false
05-20 15:39:11.595173 INFO 2925 client.cc:175:main trace_id:NONE ===Echo request begin.===
05-20 15:39:11.596434 TRACE 2925 thrift_client.h:201:WriteBlock trace_id:70442234419803 Client: WriteBlock()
05-20 15:39:11.596455 TRACE 2925 block.h:89:Submit trace_id:70442234419803 block_id 52
093094783902 parent_block_id 0 start_time 1653032351596 end_time 0 is_callee false annotation
05-20 15:39:11.615300 TRACE 2925 thrift_client.h:229:ReadBlock trace_id:70442234419803 Client: ReadBlock()
05-20 15:39:11.615320 TRACE 2925 block.h:89:Submit trace_id:70442234419803 block_id 52
093094783902 parent_block_id 0 start_time 1653032351596 end_time 1653032351615 is_callee false annotation
05-20 15:39:11.615338 INFO 2925 client.cc:175:main trace_id:NONE ===Echo end normally.===

```

图 4-8 RPC 调用客户端发出请求

在测试中，调用了两次 Echo 请求，在客户端，这两次请求的 RPC 调用链 trace_id 是不一样的。RPC 调用中，一对客户端-服务端我们称之为一个节点，因此，在发起请求的客户端调用链信息中可见，父节点 id 为 0，只有自己的节点 id。is_callee 字段是 bool 类型，记录当前调用链信息打印者身份，服务端为真，客户端为假。字段 start_time 和 end_time 记录了调用的开始时间和结束时间。

服务端表现如图 4-9 所示，这里展示了两次回应 Echo 的服务端日志。其中分别圈出了 EchoPutAttachment 服务的客户端和服务端部分。由 is_callee 位展示是否是服务端。Echo 服务的 block_id 是 EchoPutAttachment 服务的 parent_block_id，表明 Echo 服务是 EchoPutAttachment 服务的父节点。但两个服务在同一条 RPC 调用链上，因此它们的 trace_id 是一样的。

```

05-20 15:38:24.570044 INFO 2773 log_instance.h:320:InitNormalLog_ trace_id:NONE Log rotate enabled result: false
05-20 15:39:06.713019 TRACE 2775 header_binary_protocol.tcc:412:readHeader trace_id:62776338759861 Server:readHeader
05-20 15:39:06.713040 TRACE 2775 block.h:89:Submit trace_id:62776338759861 block_id 88688750488396 parent_block_id 0 start_time 1653032346712 end_time 0 is_callee true annotation Echo服务成为EchoPutAttachment服务的父节点
05-20 15:39:06.715020 INFO 2775 service.h:112:Echo trace_id:62776338759861 ===Echo in server begin==req content:helloworld
05-20 15:39:06.717466 TRACE 2775 thrift_client.h:201:WriteBlock trace_id:62776338759861 Client: WriteBlock()
05-20 15:39:06.717673 TRACE 2775 block.h:89:Submit trace_id:62776338759861 block_id 78775057025900 parent_block_id 88688750488396 start_time 1653032346717 end_time 0 is_callee false annotation EchoPutAttachment客户端
05-20 15:39:06.720694 TRACE 2776 header_binary_protocol.tcc:412:readHeader trace_id:62776338759861 Server:readHeader
05-20 15:39:06.720744 TRACE 2776 block.h:89:Submit trace_id:62776338759861 block_id 78775057025900 parent_block_id 88688750488396 start_time 1653032346720 end_time 0 is_callee true annotation
05-20 15:39:06.723324 INFO 2776 service.h:153:EchoPutAttachment trace_id:62776338759861 ===EchoPutAttachment in server begin==req content:helloworld
05-20 15:39:06.725246 INFO 2776 service.h:186:EchoPutAttachment trace_id:62776338759861 ===EchoPutAttachment in server end==rsp.content:helloworld PutAttachment, code:0
05-20 15:39:06.725520 TRACE 2776 header_binary_protocol.tcc:58:writeHeader trace_id:62776338759861 Server: writeHeader
05-20 15:39:06.725657 TRACE 2776 block.h:89:Submit trace_id:62776338759861 block_id 78775057025900 parent_block_id 88688750488396 start_time 1653032346725 is_callee true annotation
05-20 15:39:06.725995 TRACE 2775 thrift_client.h:229:ReadBlock trace_id:62776338759861 Client: ReadBlock()
05-20 15:39:06.726135 TRACE 2775 block.h:89:Submit trace_id:62776338759861 block_id 78775057025900 parent_block_id 88688750488396 start_time 1653032346725 is_callee false annotation
05-20 15:39:06.726388 INFO 2775 service.h:147:Echo trace_id:62776338759861 ===Echo in server end==, rsp.content:helloworld PutAttachment Echo, code:0
05-20 15:39:06.726766 TRACE 2775 header_binary_protocol.tcc:58:writeHeader trace_id:62776338759861 Server: writeHeader
05-20 15:39:06.726909 TRACE 2775 block.h:89:Submit trace_id:62776338759861 block_id 88688750488396 parent_block_id 0 start_time 1653032346726 end_time 1653032346726 is_callee true annotation
05-20 15:39:11.600064 TRACE 2777 header_binary_protocol.tcc:412:readHeader trace_id:70442234419803 Server:readHeader
05-20 15:39:11.600104 TRACE 2777 block.h:89:Submit trace_id:70442234419803 block_id 52093094783902 parent_block_id 0 start_time 1653032351600 end_time 0 is_callee true annotation
05-20 15:39:11.602520 INFO 2777 service.h:112:Echo trace_id:70442234419803 ===Echo in server begin==req content:helloworld
05-20 15:39:11.604871 TRACE 2777 thrift_client.h:201:WriteBlock trace_id:70442234419803 Client: WriteBlock()
05-20 15:39:11.605026 TRACE 2777 block.h:89:Submit trace_id:70442234419803 block_id 57965088562518 parent_block_id 52093094783902 start_time 1653032351604 end_time 0 is_callee false annotation
05-20 15:39:11.608145 TRACE 2778 header_binary_protocol.tcc:412:readHeader trace_id:70442234419803 Server:readHeader
05-20 15:39:11.608165 TRACE 2778 block.h:89:Submit trace_id:70442234419803 block_id 57965088562518 parent_block_id 52093094783902 start_time 1653032351608 end_time 0 is_callee true annotation
05-20 15:39:11.611749 INFO 2778 service.h:153:EchoPutAttachment trace_id:70442234419803 ===EchoPutAttachment in server begin==req content:helloworld
05-20 15:39:11.613424 INFO 2778 service.h:186:EchoPutAttachment trace_id:70442234419803 ===EchoPutAttachment in server end==rsp.content:helloworld PutAttachment, code:0
05-20 15:39:11.613680 TRACE 2778 header_binary_protocol.tcc:58:writeHeader trace_id:70442234419803 Server: writeHeader
05-20 15:39:11.613829 TRACE 2778 block.h:89:Submit trace_id:70442234419803 block_id 57965088562518 parent_block_id 52093094783902 start_time 1653032351608 end_time 1653032351613 is_callee true annotation
05-20 15:39:11.614170 TRACE 2777 thrift_client.h:229:ReadBlock trace_id:70442234419803 Client: ReadBlock()
05-20 15:39:11.614299 TRACE 2777 block.h:89:Submit trace_id:70442234419803 block_id 57965088562518 parent_block_id 52093094783902 start_time 1653032351604 end_time 1653032351614 is_callee false annotation
05-20 15:39:11.614621 INFO 2777 service.h:147:Echo trace_id:70442234419803 ===Echo in server end==, rsp.content:helloworld PutAttachment Echo, code:0
05-20 15:39:11.614855 TRACE 2777 header_binary_protocol.tcc:58:writeHeader trace_id:70442234419803 Server: writeHeader
05-20 15:39:11.614969 TRACE 2777 block.h:89:Submit trace_id:70442234419803 block_id 52093094783902 parent_block_id 0 start_time 1653032351600 end_time 1653032351614 is_callee true annotation

```

图 4-9 RPC 服务端处理请求

在测试中，只打印了 RPC 调用链上的框架性信息，业务逻辑日志记录在其他的文件中。在实际的服务中，可以加入其他的数据埋点，更好地追踪 RPC 调用链。

由测试可知，我们可以在不同的服务器上追踪同一条 RPC 调用链，并且可以根据 block_id 和 parent_block_id 获得 RPC 调用链上的调用顺序，便于出错的时候进行定位。

除函数名称，请求和回复内容的打印以外，其余的日志打印均是无感知的。即在使用 Thrift 时，在协议层选用本项目重写的协议 HeaderBinary Protocol，并继承本项目建立的客户端/服务端封装类，便可以追踪 RPC 调用链。

4.4 异常流程测试

4.4.1 Token 鉴权失败回滚

密钥鉴权失败有两种情况，一是密钥种子不一致，二是客户端不在服务端的可信任名单上。

鉴权失败时，客户端表现见图 4-10：

```
05-20 16:22:19.140699 INFO 9291 log_instance.h:320:InitNormalLog_ trace_id:None Log rotate enabled result: false
05-20 16:22:19.142066 INFO 9291 client.cc:167:main trace_id:None ===Echo request begin.===
05-20 16:22:19.143810 TRACE 9291 thrift_client.h:201:WriteBlock trace_id:97471411035566 Client: WriteBlock()
05-20 16:22:19.143831 TRACE 9291 block.h:89:Submit trace_id:97471411035566 block_id 15825218288189 parent_block_id
0 start_time 1653034939143 end_time 0 is_called false annotation
05-20 16:22:20.180912 INFO 9291 thrift_client.h:103:RequestSchema trace_id:97471411035566 Thrift exception caught
exception: THRIFT_EAGAIN (timed out) Trace id: 97471411035566
05-20 16:22:20.181013 TRACE 9291 thrift_client.h:104:RequestSchema trace_id:97471411035566 ===Try to rollback and
build another trace.===
05-20 16:22:20.181381 TRACE 9291 thrift_client.h:201:WriteBlock trace_id:97146820453075 Client: WriteBlock()
05-20 16:22:20.181471 TRACE 9291 block.h:89:Submit trace_id:97146820453075 block_id 96976747757587 parent_block_id
0 start_time 1653034940181 end_time 0 is_called false annotation
05-20 16:22:21.204805 INFO 9291 thrift_client.h:103:RequestSchema trace_id:97146820453075 Thrift exception caught
exception: THRIFT_EAGAIN (timed out) Trace id: 97146820453075
05-20 16:22:21.204966 TRACE 9291 thrift_client.h:104:RequestSchema trace_id:97146820453075 ===Try to rollback and
build another trace.===
05-20 16:22:21.205230 TRACE 9291 thrift_client.h:201:WriteBlock trace_id:65220688773546 Client: WriteBlock()
05-20 16:22:21.205312 TRACE 9291 block.h:89:Submit trace_id:65220688773546 block_id 65745714614119 parent_block_id
0 start_time 1653034941265 end_time 0 is_called false annotation
05-20 16:22:22.228806 INFO 9291 thrift_client.h:103:RequestSchema trace_id:65220688773546 Thrift exception caught
exception: THRIFT_EAGAIN (timed out) Trace id: 65220688773546
05-20 16:22:22.228921 TRACE 9291 thrift_client.h:104:RequestSchema trace_id:65220688773546 ===Try to rollback and
build another trace.===
05-20 16:22:22.229240 TRACE 9291 thrift_client.h:201:WriteBlock trace_id:15367458787797 Client: WriteBlock()
05-20 16:22:22.229322 TRACE 9291 block.h:89:Submit trace_id:15367458787797 block_id 36335189816740 parent_block_id
0 start_time 1653034942229 end_time 0 is_called false annotation
05-20 16:22:23.252815 INFO 9291 thrift_client.h:103:RequestSchema trace_id:15367458787797 Thrift exception caught
exception: THRIFT_EAGAIN (timed out) Trace id: 15367458787797
05-20 16:22:23.252920 TRACE 9291 thrift_client.h:104:RequestSchema trace_id:15367458787797 ===Try to rollback and
build another trace.===
05-20 16:22:23.253140 INFO 9291 client.cc:172:main trace_id:None ===Echo end failly==
```

图 4-10 鉴权失败客户端表现

当服务端鉴权失败，客户端会 catch 到 THRIFT_EAGAIN 错误，会断开连接后重新连接（建立新的 RPC 调用链）。在这个过程中，客户端的信息会进行回滚，比如 block 信息，开始时间，父节点等字段。

鉴权失败时，服务端表现如图 4-11：

```
05-20 16:20:59.475552 INFO 2779 header_binary_protocol.tcc:365:readHeader trace_id:None header token not matched
: token 8589935258(cluster_id 0, token_seed 666, module_id 2) trace_id 14229030888053 client_ip 172.17.0.1
05-20 16:21:00.503890 INFO 2780 header_binary_protocol.tcc:365:readHeader trace_id:None header token not matched
: token 8589935258(cluster_id 0, token_seed 666, module_id 2) trace_id 68702428949987 client_ip 172.17.0.1
05-20 16:21:01.527072 INFO 2781 header_binary_protocol.tcc:365:readHeader trace_id:None header token not matched
: token 8589935258(cluster_id 0, token_seed 666, module_id 2) trace_id 22604420649038 client_ip 172.17.0.1
05-20 16:21:02.551118 INFO 2782 header_binary_protocol.tcc:365:readHeader trace_id:None header token not matched
: token 8589935258(cluster_id 0, token_seed 666, module_id 2) trace_id 95192248913202 client_ip 172.17.0.1
05-20 16:22:19.149457 INFO 2783 header_binary_protocol.tcc:365:readHeader trace_id:None header token not matched
: token 8589935258(cluster_id 0, token_seed 666, module_id 2) trace_id 97471411035566 client_ip 172.17.0.1
05-20 16:22:20.184091 INFO 2784 header_binary_protocol.tcc:365:readHeader trace_id:None header token not matched
: token 8589935258(cluster_id 0, token_seed 666, module_id 2) trace_id 97146820453075 client_ip 172.17.0.1
05-20 16:22:21.206927 INFO 2785 header_binary_protocol.tcc:365:readHeader trace_id:None header token not matched
: token 8589935258(cluster_id 0, token_seed 666, module_id 2) trace_id 65220688773546 client_ip 172.17.0.1
05-20 16:22:22.231240 INFO 2786 header_binary_protocol.tcc:365:readHeader trace_id:None header token not matched
: token 8589935258(cluster_id 0, token_seed 666, module_id 2) trace_id 15367458787797 client_ip 172.17.0.1
```

图 4-11 密钥不匹配服务端表现

服务端会在读信息头时进行鉴权，如果不通过会打印客户端信息，并提示 token 不匹配。

4.4.2 No more data to read 错误模拟

“No more data to read.” 是最常见的 Apache Thrift 报错，该报错的根本原因是“连接被对端关闭”，出现原因可能是：

1. 如果是长连接，连接闲置时间超过服务端的接收超时，则服务端就会将连接关闭，此后再使用该连接发送数据则会出现“No more data to read.” 的报错；
2. 此外，服务端连接进行 recv 时如果被系统中断打断，也会触发服务端关闭连接，此时客户端再对连接进行操作，同样会出现“No more data to read.” 的报错；
3. 并发压力比较大的时候，client 端 connect 成功，但 server 端由于并发压力过大并没有真正的 accept，client 端此时再使用这个连接进行通信，同样会出现“No more data to read.” 的报错。

我们通过关闭通信中的服务端来模拟这种错误。错误发生时，客户端的回滚如图 4-12 所示：

```
04-30 17:11:08.598927 INFO 6984 log_instance.h:320:InitNormalLog_ trace_id:None Log rotate enabled result: false
04-30 17:11:08.592333 TRACE 6984 client.cc:167:main trace_id:None =====Echo request begin=====
04-30 17:11:08.593758 TRACE 6984 thrift_client.h:202:WriteBlock trace_id:75313657158594 Client: WriteBlock()
04-30 17:11:08.593779 TRACE 6984 block.h:89:Submit trace_id:75313657158594 block_id 72621301477664 parent_block_id 0 start_time 1651309868593 end_time
0 is_caller false annotation
04-30 17:11:09.623237 INFO 6984 thrift_client.h:104:RequestSchema trace_id:75313657150594 Thrift exception caught, exception: THRIFT_EAGAIN (timed out)
) Trace_id: 75313657150594
04-30 17:11:09.623296 TRACE 6984 thrift_client.h:105:RequestSchema trace_id:75313657150594 =====Try to rollback and build another trace=====
04-30 17:11:09.623637 TRACE 6984 thrift_client.h:202:WriteBlock trace_id:6954993555554 Client: WriteBlock()
04-30 17:11:09.623710 TRACE 6984 block.h:89:Submit trace_id:6954993555554 block_id 53474459523795 parent_block_id 0 start_time 1651309869623 end_time 0 is_caller false annotation
04-30 17:11:10.301027 INFO 6984 thrift_client.h:104:RequestSchema trace_id:6954993555554 Thrift exception caught, exception: No more data to read. Trace_id: 6954993555554
04-30 17:11:10.301169 TRACE 6984 thrift_client.h:105:RequestSchema trace_id:6954993555554 =====Try to rollback and build another trace=====
04-30 17:11:10.301553 TRACE 6984 thrift_client.h:85:RequestSchema trace_id:None =====Connection broken, don't rollback=====
```

图 4-12 No more data to read 错误模拟

4.4.3 Broken Pipe 错误模拟

broken pipe 错误产生的原因通常是因为管道的当前读取端没有读取，而管道的写入端仍在由线程写入。相应的，是 thrift 客户端将数据写入关闭的 socket，引发异常。

我们通过在客户端写入数据时关闭服务端来模拟此错误。错误发生后，客户端的回滚如图 4-13 所示：

```
04-30 17:25:39.352110 INFO 9320 log_instance.h:320:InitNormalLog_ trace_id:None Log rotate enabled result: false
04-30 17:25:39.353542 TRACE 9320 client.cc:167:main trace_id:None =====Echo request begin=====
04-30 17:25:39.354983 TRACE 9320 thrift_client.h:202:WriteBlock trace_id:41250319449964 Client: WriteBlock()
04-30 17:25:39.356003 TRACE 9320 block.h:89:Submit trace_id:41250319449964 block_id 62242319952586 parent_block_id 0 start_time 1651310739354 end_time 0 is_caller false annotation
04-30 17:25:44.362183 INFO 9320 thrift_client.h:104:RequestSchema trace_id:41250319449964 Thrift exception caught, exception: write() send(): Broken pipe
Trace_id: 41250319449964
04-30 17:25:44.362289 TRACE 9320 thrift_client.h:105:RequestSchema trace_id:41250319449964 =====Try to rollback and build another trace=====
04-30 17:25:44.362720 TRACE 9320 thrift_client.h:85:RequestSchema trace_id:None =====Connection broken, don't rollback=====
```

图 4-13 Broken Pipe 错误模拟

4.5 测试结论

测试结论表明，系统基本达到设计目标。

1. 正常测试结论

经过测试，证明系统在正常流程下，能够正确地在 RPC 调用链中完成信息的埋入，打出正确、清晰、易于分析的调用链日志。

同时，系统能够做到业务无感知的调用链信息埋入。在编写 RPC 客户端和服务端时，只需要继承本项目设计实现的客户端类/服务端类，并且在注册时声明使用新的 HeaderBinaryProtocol 协议，即可完成对 Thrift 框架下 RPC 调用链的信息埋点。

2. 异常测试结论

经过测试，证明系统具有一定的健壮性，对附件的传递使用 CRC 校验，如附件出现损坏，进行提醒并回滚。

系统具有鉴权功能，引入密钥系统，可保证服务的安全性，如鉴权失败，进行提醒并回滚。

系统具有自我修复能力，能够在 Thrift 中途请求出错时，自动回滚调用链并打出相应的错误日志的同时，不妨碍正常的调用进行。

第五章 结论

5.1 项目总结

5.1.1 完成情况

本项目通过设计并编写客户端类，服务端类，和新的 Thrift 传输协议，完成了对基于 Thrift 框架的 RPC 调用链信息埋入的设计和实现。能够正确地在 RPC 调用链中完成信息的埋入，打出正确、清晰、易于分析的调用链日志。

同时，能够做到业务无感知的调用链信息埋入，在编写 RPC 客户端和服务端时，只需要继承本项目设计实现的客户端类/服务端类，并且在注册时声明使用新的 HeaderBinaryProtocol 协议，即可完成对 Thrift 框架下 RPC 调用链的信息埋点。减少业务方的对系统的侵入，以免业务方出错导致整个调用链不可靠。

本系统具有一定的健壮性，对附件的传递使用 CRC 校验，可保证附件的传输的完整性；具有鉴权功能，引入密钥系统，可保证服务的安全性；具有自我修复能力，能够在 Thrift 中途请求出错时，自动回滚调用链并打出相应的错误日志的同时，不妨碍正常的调用进行。

总体来说，完成了初期对本系统的设计和目标，在一定程度上解决了工业界 RPC 调用链信息埋入消耗大，代码侵入性强，操作复杂的问题。成功建立了一个离线的调用链信息埋入系统。

5.1.2 未来展望

本系统的发展方向基本符合预期，完成度依旧有可以进一步增高的空间，未来的发展方向主要向以下几个方面发展：

1. 优化客户端/服务端的业务函数传入方式

请求/处理请求的函数代码依旧需要在基类中实现，要求业务方在特定的位置嵌入业务代码。未来会进一步提高对客户端/服务端基类封装的程度，如提供客户端/服务端初始化的接口，需要将业务函数作为函数指针传入，这样可以进一步增加封装的程度。

2. 采用更高效的分发方式

本调用链信息埋入系统的构建分发依旧是基于源码实现的，分发方式的优化有以下两个方向：一、在完成未来展望的第一条的基础上，将本系统生成库函数，在库中提供客户端/服务端初始化的接口。二、分发 shell 脚本或者分发可执行文件，用户可以使用脚本/可执行文件生成框架代码，借此使用本系统。

3. 完善信息管理

本调用链信息埋入系统现在已经可以完成对于调用链信息的记录和对于调用链过程的回溯。之后可以对其他维度的信息进行记录，比如成功次数，概率，拥塞情况，调用链压力统计等其他统计学信息进行整理输出。如果进一步发展，可以发展 UI 面板，输出数据统计图形，增加系统的可视化程度。

4. 可配置化

本系统对于系统信息的配置化程度较低，只能通过编译参数/代码嵌入的方式进行配置，在未来可以优化配置方式，必要时加入配置的动态加载。

本系统作为一个基于 Thrift 框架的开源离线 RPC 调用链信息埋入系统，希望可以在未来得到进一步优化，并为个人和工业带去更多的便利。

5.2 对工程伦理的理解

随着软件的规模不断扩张，其专业技术水平和对工程人员的合作要求都在不断增高，软件工程也成了一个具有巨大社会效应与社会影响力的领域。它和科学一样，为人类带来的福祉，同时也带来了众多风险和挑战。

工程师最重要的责任就是对工程本身负有的责任。软件工程需要严格遵守严密的设计流程，即问题定义，可行性研究，需求分析，总体设计，详细设计，编码与单测，综合测试，后续维护等工作，这是对工作和工程结果的负责。

同时，工程师还应对工程伦理中的道德层面负责：

一、在设计中遵循道德观念，不能违反社会良知，即不能通过非法手段获得利益，不能损害工程使用者的权利。

二、对不可测的工程研究结果负责，在设计时考虑到可能造成的不良后果，并能够提前进行应对，防止损害社会利益和他人利益。

三、是作为专业人士，应该产生对社会有价值的产品以及服务，推动公众对工程职业的信赖，借由服务社会，树立工程师的正面形象。

四、保持诚信，以客观的、诚实的方式进行工业设计，避免欺骗，以自己正直、可靠、道德以及合法地方式维护本职业的尊严、地位和荣誉。

作为一个年轻的软件工程师，在本次毕业论文中，我体会到了作为一个软件工程师的责任。我需要在维护系统稳定的基础上，尽可能多地考虑到满足社会和他人的利益，并且严格遵守设计规范，保护使用者的信息安全，使使用者可以得到好的使用体验。同时，我遵守开源的原则，为整个软件工程社区的开源生态贡献了力量。

在未来的工作中，我将严格遵守工程伦理，在维护自身和他人的利益的基础上，为集体作出贡献。

5.3 对职业素养的理解

在系统的设计，实施和论文的撰写过程中，我认识到了我作为一个软件工程师的责任和不足。

首先，软件工程师应有的职业素养有以下几点：

一、注重细节，程序设计严谨，测试充分，不放过任何一个异常，不抱有侥幸心理。

二、主动提交程序对应的流程图、说明文档、测试文档。

三、认真听取他人意见，敢于正视自己的错误。

四、主动提高自己的职业水准，保持学习的习惯，接纳新的发展和技术。

我的不足主要表现在我对较大规模工程的设计流程依旧不够熟练。设计对于边缘情况的考虑依旧不足，在实现中才发现的问题较多，使设计比较冗杂。同时，我对于软件开发知识的了解尚有短板，在他人指出我的问题之后才较为系统地学习了很多非常重要的技术知识。在之后的工作和学习中，我更加耐心、努力、细心地学习每一个技术要点，并且拥有攻坚克难的勇气，更好解决技术难点，为软件工程的发展作出贡献。

致 谢

本论文的工作是在我的导师白忠健老师悉心指导下完成的，我要感谢我的导师，谢谢他对我孜孜不倦的教诲，对我论文从选题到构思再到定稿中的每个环节给予的细心的指导，他给出的建议对论文的完成起到了关键性的作用。

其次，我想要感谢我的同学邓萌达同学，他对我的研究提出了很多建设性的意见，并且帮助我在遇到瓶颈时分析问题，给予帮助，感谢他成为我前进道路上的启明星。

同时，我想要感谢我的父母亲人们，感谢他们用辛勤的劳动，给予我一个安稳且富足的生活环境，并且一直非常支持我追求我想要的生活，没有他们的付出，也没有能够坐在大学安静的图书馆中，完成这篇论文的我。

最后，我想感谢我的学校电子科技大学及其每一位教师和职工，感谢他们在过去的四年里为我提供了一片学习的沃土，孜孜不倦地为我传授知识，勤勤恳恳地为我提供舒适的生活。

感谢每一个在我成长道路上遇见的人，青山不改，绿水长流，后会有期。

参考文献

- [1] Ylonen T, Lonvick C. The secure shell (SSH) protocol architecture[J]. 2006.
- [2] Barrett D J, Silverman R E, Byrnes R G. SSH, The Secure Shell: The Definitive Guide: The Definitive Guide[M]. " O'Reilly Media, Inc.", 2005.
- [3] <https://thrift.apache.org/>
- [4] Slee M, Agarwal A, Kwiatkowski M. Thrift: Scalable cross-language services implementation[J]. Facebook white paper, 2007, 5(8): 127.
- [5] <http://jnb.ociweb.com/jnb/jnbJun2009.html>
- [6] Sites R L, Chernoff A, Kirk M B, et al. Binary translation[J]. Communications of the ACM, 1993, 36(2): 69-81.
- [7] Martorell X, Ayguadé E, Navarro N, et al. Thread fork/join techniques for multi-level parallelism exploitation in NUMA multiprocessors[C]//Proceedings of the 13th international conference on Supercomputing. 1999: 294-301.
- [8] Peterson W W, Brown D T. Cyclic codes for error detection[J]. Proceedings of the IRE, 1961, 49(1): 228-235.
- [9] <https://web.archive.org/web/20160826091616/https://isocpp.org/blog/2013/04/n3659-shared-locking> C++ shared
- [10] Xu D. Performance study and dynamic optimization design for thread pool systems[R]. Ames Lab., Ames, IA (United States), 2004.

外文资料原文

Thrift: Scalable Cross-Language Services Implementation

Mark Slee, Aditya Agarwal and Marc Kwiatkowski Facebook, 156 University Ave, Palo Alto, CA

{meslee,aditya,marc}@facebook.com

Abstract

Thrift is a software library and set of code-generation tools developed at Facebook to expedite development and implementation of efficient and scalable backend services. Its primary goal is to enable efficient and reliable communication across programming languages by abstracting the portions of each language that tend to require the most customization into a common library that is implemented in each language. Specifically, Thrift allows developers to define datatypes and service interfaces in a single language-neutral file and generate all the necessary code to build RPC clients and servers.

This paper details the motivations and design choices we made in Thrift, as well as some of the more interesting implementation details. It is not intended to be taken as research, but rather it is an exposition on what we did and why.

1. Introduction

As Facebook's traffic and network structure have scaled, the resource demands of many operations on the site (i.e. search, ad selection and delivery, event logging) have presented technical requirements drastically outside the scope of the LAMP framework. In our implementation of these services, various programming languages have been selected to optimize for the right combination of performance, ease and speed of development, availability of existing libraries, etc. By and large, Facebook's engineering culture has tended towards choosing the best tools and implementations available over standardizing on any one programming language and begrudgingly accepting its inherent limitations.

Given this design choice, we were presented with the challenge of building a transparent, high-performance bridge across many programming languages. We found that most available solutions were either too limited, did not offer sufficient datatype freedom,

The solution that we have implemented combines a language-neutral software stack implemented across numerous programming languages and an associated code generation engine that transforms a simple interface and data definition language into client and server remote procedure call libraries. Choosing static code generation over a dynamic system allows us to create validated code that can be run without the need for any advanced introspective run-time type checking. It is also designed to be as simple as possible for the developer, who can typically define all the necessary data structures and interfaces for a complex service in a single short file.

Surprised that a robust open solution to these relatively common problems did not yet exist, we committed early on to making the Thrift implementation open source.

¹ See Appendix A for a discussion of alternative systems.

In evaluating the challenges of cross-language interaction in a networked environment, some key components were identified:

Types. A common type system must exist across programming languages without requiring that the application developer use custom Thrift datatypes or write their own serialization code. That is, a C++ programmer should be able to transparently exchange a strongly typed STL map for a dynamic Python dictionary. Neither programmer should be forced to write any code below the application layer to achieve this. Section 2 details the Thrift type system.

Transport. Each language must have a common interface to bidirectional raw data transport. The specifics of how a given transport is implemented should not matter to the service developer. The same application code should be able to run against TCP stream sockets, raw data in memory, or files on disk. Section 3 details the Thrift Transport layer.

Protocol. Datatypes must have some way of using the Transport layer to encode and decode themselves. Again, the application developer need not be concerned by this layer. Whether the service uses an XML or binary protocol is immaterial to the application code. All that matters is that the data can be read and written in a consistent, deterministic manner. Section 4 details the Thrift Protocol layer.

Versioning. For robust services, the involved datatypes must provide a mechanism for versioning themselves. Specifically, it should be possible to add or remove fields in an object or alter the argument list of a function without any interruption in service (or, worse yet, nasty segmentation faults). Section 5 details Thrift's versioning system.

Processors. Finally, we generate code capable of processing data streams to accomplish remote procedure calls. Section 6 details the generated code and TProcessor paradigm.

Section 7 discusses implementation details, and Section 8 describes our conclusions.

外文资料译文

Thrift：可扩展的跨语言服务实施

Mark Slee, Aditya Agarwal and Marc Kwiatkowski Facebook, 156 University Ave, Palo Alto,
CA {mcslee,aditya,marc}@facebook.com

摘要

Thrift 是 Facebook 开发的一个软件库和一组代码生成工具，用于加快高效和可扩展后端服务的开发和实施。它的主要目标是通过将每种语言中往往需要最多定制的部分抽象到一个用每种语言实现的公共库中，从而实现跨编程语言的高效和可靠的通信。具体来说，Thrift 允许开发人员在单个语言中立文件中定义数据类型和服务接口，并生成构建 RPC 客户端和服务器所需的所有代码。

本文详细介绍了我们在 Thrift 中所做的动机和设计选择，以及一些更有趣的实现细节。它不打算被视为研究，而是对我们所做的事情和原因的阐述。

一、介绍

随着 Facebook 的流量和网络结构的扩展，网站上许多操作（即搜索、广告选择和投放、事件记录）的资源需求已经提出了远远超出 LAMP 框架范围的技术要求。在我们实施这些服务的过程中，选择了各种编程语言来优化性能、开发的简易性和速度、现有库的可用性等的正确组合。总的来说，Facebook 的工程文化倾向于选择最好的工具和实现可以超越任何一种编程语言的标准化并勉强接受其固有的局限性。

鉴于这种设计选择，我们面临着构建跨多种编程语言的透明、高性能桥梁的挑战。我们发现大多数可用的解决方案要么太有限，要么没有提供足够的数据类型自由度，要么有低于预期的表现。

我们实施的解决方案结合了跨多种编程语言实现的语言中立的软件堆栈和相关的代码生成引擎，该引擎将简单的接口和数据定义语言转换为客户端和服务器远程过程调用库。在动态系统上选择静态代码生成允许我们创建经过验证的代码，无需任何高级内省运行时类型检查即可运行。它还被设计为对开发人员来说尽可能简单，他们通常可以在单个短文件中定义复杂服务的所有必要数据结构和接口。

对这些相对常见的问题尚不存在强大的开放解决方案感到惊讶，我们很早就承诺将 Thrift 实现开源。

在评估网络环境中跨语言交互的挑战时，确定了一些关键组件：

类型。通用类型系统必须跨编程语言存在，而不需要应用程序开发人员使用自定义 Thrift 数据类型或编写自己的序列化代码。也就是说，C++ 程序员应该能够透明地将强类型 STL 映射交换为动态 Python 字典。程序员都不应被迫在应用层以下编写任何代码来实现这一点。第 2 节详细介绍了 Thrift 类型系统。

运输。每种语言都必须有一个通用接口来双向传输原始数据。如何实现给定传输的细节对服务开发人员来说并不重要。相同的应用程序代码应该能够针对 TCP 流套接字、内存中的原始数据或磁盘上的文件运行。第 3 节详细介绍了 Thrift 传输层。

协议。数据类型必须有某种方式使用传输层对其自身进行编码和解码。同样，应用程序开发人员不需要关心这一层。服务使用 XML 还是二进制协议对应用程序代码无关紧要。重要的是数据可以以一致的、确定的方式读取和写入。第 4 节详细介绍了 Thrift 协议层。

版本控制。对于健壮的服务，所涉及的数据类型必须提供一种对自身进行版本控制的机制。具体来说，应该可以在对象中添加或删除字段或更改函数的参数列表，而不会中断服务（或者更糟糕的是，令人讨厌的分段错误）。第 5 节详细介绍了 Thrift 的版本控制系统。

处理器。最后，我们生成能够处理数据流以完成远程过程调用的代码。第 6 节详细介绍了生成的代码和 TProcessor 范例。