武汉理工大学毕业设计（论文）

**基于Thrift RPC的微服务自测平台**

学院（系）：计算机科学与技术学院

专业班级： 软件工程zy1501班

学生姓名： 卫亚峰

指导教师： 邹承明

学位论文原创性声明

本人郑重声明：所呈交的论文是本人在导师的指导下独立进行研究所取得的研究成果。除了文中特别加以标注引用的内容外，本论文不包括任何其他个人或集体已经发表或撰写的成果作品。本人完全意识到本声明的法律后果由本人承担。

作者签名：

年 月 日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保障、使用学位论文的规定，同意学校保留并向有关学位论文管理部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权省级优秀学士论文评选机构将本学位论文的全部或部分内容编入有关数据进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本学位论文属于1、保密囗，在 年解密后适用本授权书

2、不保密囗 。

（请在以上相应方框内打“√”）

作者签名：

年 月 日

导师签名：

年 月 日

摘 要

处于对开发高效性、部署便利性、系统稳定性、系统扩展性等因素的多重考虑，现业内大型互联网公司后端系统大多基于微服务架构。主流的微服务间通信方式有Thrift RPC、gRPC两种方式。目前这种基于RPC、微服务之间相互依赖的架构，使代码开发完成之后的测试变成一件十分繁琐的事情。为提高基Thrift RPC通信的微服务的开发人员的开发效率，本文设计了一个自动化的自测系统。

论文主要研究工作如下：

1. 基于特定服务的Thrift IDL文件与一些元信息，自动化的生成加载运行特定服务的Thrift客户端或者服务器。
2. 基于用户提供的测试数据，自动化的触发特定服务的特定接口，按预期调用下游服务和进行当前服务的逻辑处理之后，返回结果，供用户验证结果是否符合预期。
3. 在测试服务、测试接口、测试数据三个维度，为用户提供数据的维护与管理。

本系统的最终目标是提供一个自动化的自测系统，帮助开发人员在自测阶段搭建测试环境和维护测试用例等，以提高其工作效率。

关键词：微服务；Thrift RPC；beego；自测

**Abstract**

Considering the development efficiency, deployment convenience, system stability, system scalability and other factors, most of the back-end systems of large Internet companies in the industry are based on micro-service architecture. There are two main modes of inter-service communication: Thrift RPC and gRPC. At present, this architecture based on the interdependence between RPC and micro-services makes testing after code development become a very tedious task. In order to improve the development efficiency of micro-service developers based on Thrift RPC communication, an automatic self-test system is designed in this paper.

In the process of implementing this application, this article has done the following work:

1. Based on Thrift IDL files and some meta-information for specific services, the system automatically generates and loads Thrift clients or servers for specific services.

2. Based on the test data provided by users, specific interfaces of specific services are triggered automatically. After calling downstream services as expected and processing current services logically, the results are returned for users to verify whether the results are in line with expectations.

3. Provide users with data maintenance and management in three dimensions: test service, test interface and test data.

The final goal of this system is to provide an automated self-test system to help developers build test environment and maintain test cases during the test phase, so as to improve their work efficiency.

**Key Words：**Micro service; Thrift RPC; beego; Self-test

目 录

[第1章 绪论 1](#_Toc9616563)

[1.1 背景 1](#_Toc9616564)

[1.2 业内现状 1](#_Toc9616565)

[1.3 目的及意义 2](#_Toc9616566)

[1.4 预期目标 2](#_Toc9616567)

[第2章 关键技术分析研究 3](#_Toc9616568)

[2.1 Thrift RPC 3](#_Toc9616569)

[2.1.1 Thrift RPC 概述 3](#_Toc9616570)

[2.1.2 Thrift RPC 软件栈 3](#_Toc9616571)

[2.2 beego框架 4](#_Toc9616572)

[2.2.1 beego 框架架构 4](#_Toc9616573)

[2.2.2 HTTP请求处理流程 5](#_Toc9616574)

[2.3动态链接机制 6](#_Toc9616575)

[2.3.1 编译与链接 6](#_Toc9616576)

[2.3.2 静态链接库与动态链接库 7](#_Toc9616577)

[2.3.3 动态链接库的使用 7](#_Toc9616578)

[第3章 系统设计 8](#_Toc9616579)

[3.1 系统总体架构设计 8](#_Toc9616580)

[3.2 kernel模块设计 9](#_Toc9616581)

[3.2.1 GetServiceList接口 9](#_Toc9616582)

[3.2.2 AddService接口 9](#_Toc9616583)

[3.2.3 FillData接口 10](#_Toc9616584)

[3.2.4 RequestTrigger接口 11](#_Toc9616585)

[3.3 CGI模块设计 11](#_Toc9616586)

[3.5.1数据库设计 11](#_Toc9616587)

[3.5.2 CGI模块接口设计 13](#_Toc9616588)

[3.5.3 CGI架构设计 13](#_Toc9616589)

[3.4 本章小结 14](#_Toc9616590)

[第4章 系统关键功能实现 15](#_Toc9616591)

[4.1添加服务功能实现 15](#_Toc9616592)

[4.1.1 何时添加服务 15](#_Toc9616593)

[4.1.2 添加服务流程概述 15](#_Toc9616594)

[4.1.3 添加服务要点详述 16](#_Toc9616595)

[4.2 填充数据功能实现 18](#_Toc9616596)

[4.2.1 TJSONProtocl协议 18](#_Toc9616597)

[4.2.2 填充数据流程 18](#_Toc9616598)

[4.2.3 FillData接口实现逻辑 19](#_Toc9616599)

[4.3 本章小结 20](#_Toc9616600)

[第5章 系统测试 21](#_Toc9616601)

[5.1 kernel模块测试 21](#_Toc9616602)

[5.1.1 AddService接口 22](#_Toc9616603)

[5.1.2 GetServiceList接口 23](#_Toc9616604)

[5.1.3 FillData接口 24](#_Toc9616605)

[5.1.4 RequetTrigger接口 25](#_Toc9616606)

[5.2 CGI模块测试 25](#_Toc9616607)

[5.2.1 添加服务模式 25](#_Toc9616608)

[5.2.2 添加接口模式 26](#_Toc9616609)

[5.2.3 添加测试用例 27](#_Toc9616610)

[5.2.4 触发测试用例 28](#_Toc9616611)

[5.3 本章小结 29](#_Toc9616612)

[第6章 结论 30](#_Toc9616613)

[6.1总结 30](#_Toc9616614)

[致谢 2](#_Toc9616615)

# 第1章 绪论

## 1.1 背景

过去的十年中，互联网对我们的生活产生了翻天覆地的变化，而互联网产品通常有两大特点：需求变化快与用户基数大。在这种业务场景下，现代大型互联网公司后端系统大多采用微服务架构。

微服务是一个软件架构层面的理念。这种架构理念提倡当需要实现某种用户需求时，采取通过多个独立的小体量的服务相互协调的方案，而不是将所有的需求任务使用一个单一大体量的程序完成。其中多个独立的小体量的服务常被称为微服务。微服务与微服务之间常通过RPC机制进行通信。

作为一种主流的微服务通信机制解决方案，Thrift RPC是一种完备的、跨语言的、可伸缩的RPC解决方案[1]。在系统总体架构为微服务架构的前提下，通常，针对某个微服务，常通过实现一个Thrift RPC服务器，向上游提供服务；上游仅需要实现一个针对当前微服务的客户端，实现对当前微服务的调用。相对的，当前微服务也具有一个或多个下游服务，并通过实现对各个下游微服务对应的客户端的方法，调用下游依赖提供的服务。

工业环境下，针对某个需求的开发流程大致是：开发；自测；集成测试；上线。其中，集成测试一般由QA部门进行统一测试，并具有完善的全套的测试环境。自测指开发人员开发完成后，由开发人员进行针对微服务新特性的测试，所以一般并不会对每个开发人员配备完善的测试环境。而在上述通过Thrift RPC实现的微服务架构场景下，各个微服务之间常存在复杂的相互依赖关系，这使得针对特定服务的特定业务场景的测试成为一个业界难题。

## 1.2 业内现状

当某个开发人员完成对特定的微服务的新特性开发后，通过需要模拟出能够触发当前微服务的新特性的业务场景，测试当前微服务的新特性是否能按预期运行。这一过程在业内没有统一的解决方案，每个开发人员都有个人的解决方案。虽然各个解决方案实现起来有较大差异，但针对这个问题解决方案的大致思路都是一致的，如下所述：

1. 实现一个当前微服务对应的Thrift客户端，用以调用当前微服务。
2. 针对每一个下游微服务，实现一个下游微服务的Thrift服务器，以供当前微服务调用。
3. 针对每一个下游微服务的特定接口，模拟出特定的响应数据。
4. 针对当前微服务代测试的某个接口，模拟出特定的请求数据，并触发这个特定接口。
5. 针对当前测试用例（包涵当前微服务的请求数据与下游服务的响应数据），结合被测试接口的响应数据与当前微服务的打印日志，判断当前微服务新特性时候按预期运行。

总体来讲，这个解决问题的思路至少有如下两个问题：

1. 针对每一个所开发的微服务，开发人员同时也要维护当前微服务所有的上下游服务，这会大大增大开发人员的开发量。
2. 当业务场景越来越多时，对于测试用例的管理也会给开发人员带来很大的负担。

## 1.3 目的及意义

依上文所述，当针对某个微服务的新特性进行测试时，存在上述问题。为解决上述用户需求，本课题设计了本文所述的自测系统。

本系统所要实现的功能，主要是帮助开发人员进行自动化的生成Thrift客户端与服务器，并帮助用户进行测试数据的管理与维护，以达到提高开发人员工作效率的目的。

## 1.4 预期目标

本课题的预期目标如下：

1. 基于特定服务的Thrift IDL文件与一些元信息，自动化的生成加载运行特定服务的Thrift客户端或者服务器
2. 基于用户提供的测试数据，自动化的触发特定服务的特定接口，按预期调用下游服务和进行当前服务的逻辑处理之后，返回结果，供用户验证结果是否符合预期
3. 在测试服务、测试接口、测试数据三个维度，为用户提供数据的维护与管理。

# 第2章 关键技术分析研究

本章主要对本系统的实现过程中涉及到的一些核心技术进行分析与研究。

## 2.1 Thrift RPC

### 2.1.1 Thrift RPC 概述

Thrift RPC是一种开源的RPC框架，其包含两部分。一是用以支持Thrift底层通信协议的库文件，二是Thrift编译器。其主要特定如下：

1. 开发速度快：开发人员仅需根据微服务间通信协议编写IDL文件。Thrift编译器利用IDL文件即可自动生成服务端框架代码与客户端桩代码。这些代码结合Thrift提供的库文件，即可实现RPC机制。整个过程接口编解码、消息传输、服务器多线程模型等基础工作均有Thrift实现，用户仅需实现业务代码即可。
2. 接口维护简单高效：使用Thrift协议规定的IDL文件格式，即可作为微服务之间通信的接口使用文档。可以直接根据IDL文件生成接口代码，使得代码与文档可以保持一致性。而且IDL文件本身也提供了良好的版本控制方案。
3. 学习成本低：因为其来自Google Protocol Buffers开发团队，所以其IDL文件风格类似Google Protocol Buffers[7]，且更加易读易懂；特别是RPC服务接口的风格就像写一个一般的面向对象的Class一样简单[6]。

### 2.1.2 Thrift RPC 软件栈

Thrift RPC协议本身定义的软件栈提供了清晰的分层，下层协议向上层协议提供服务，各个协议层可以松散的耦合。Thrift 软件栈如下图2.1所示。

具体来讲，Thrift软件栈有如下层[12]：

1. Transport: 传输层，定义数据传输方式，即定义获取字节流的方式，可以通过文件获取字节流，通过内存获取字节流，通过网络获取字节流等。
2. protocol: 协议层, 定义数据传输格式，即定义字节流与Thrift所提供标准数据结构的编解码规则，可以为JSON、XML或二进制格式等。
3. Processor: 处理器层，此协议层由Thrift编译器根据用户定义的IDL文件生成，对通信协议的输入输出进行了封装，并托付给用户实现的处理器进行处理。
4. Server: 服务层, 整合上述组件, 提供网络模型(单线程/多线程/事件驱动), 最终形成真正的服务。

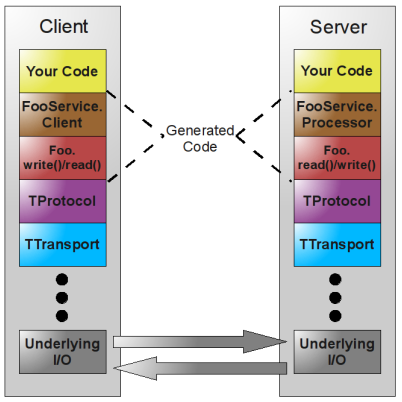


图2.1 Thrift RPC软件栈

## 2.2 beego框架

beego是一个基于Go语言的HTTP框架，它为后端服务在处理HTTP请求时常见的问题，提供了较为通用的解决方案**[14]**。

### 2.2.1 beego 框架架构

beego 是基于八大独立的模块构建的，是一个高度解耦的框架，如图2.2所示。

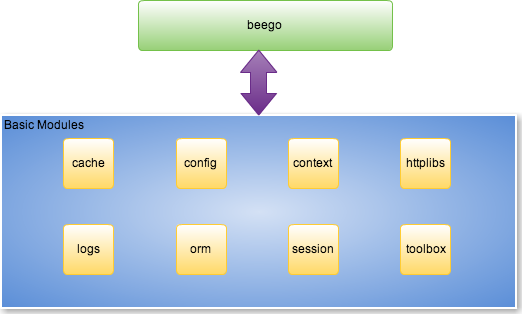


图2.2 beego框架架构

其中，每一个模块都独立负责解决Web开发中某一方面常见的问题。在本课设的设计中，用到的beego模块主要有如下几个：

1. session模块：负责进行会话管理，即维护用户的登录状态。
2. config模块：负责维护系统所需要的配置信息。
3. logs模块：负责系统的日志系统。
4. orm模块：提供了go语言中的Struct 与数据库中表的映射，基于这种映射封装了一些对于数据库的操作。

### 2.2.2 HTTP请求处理流程

beego作为一个成熟的Web框架，从请求处理的角度看，采用了典型了MVC架构，具体来讲，beego框架处理一个请求的流程如下图2.3所示：

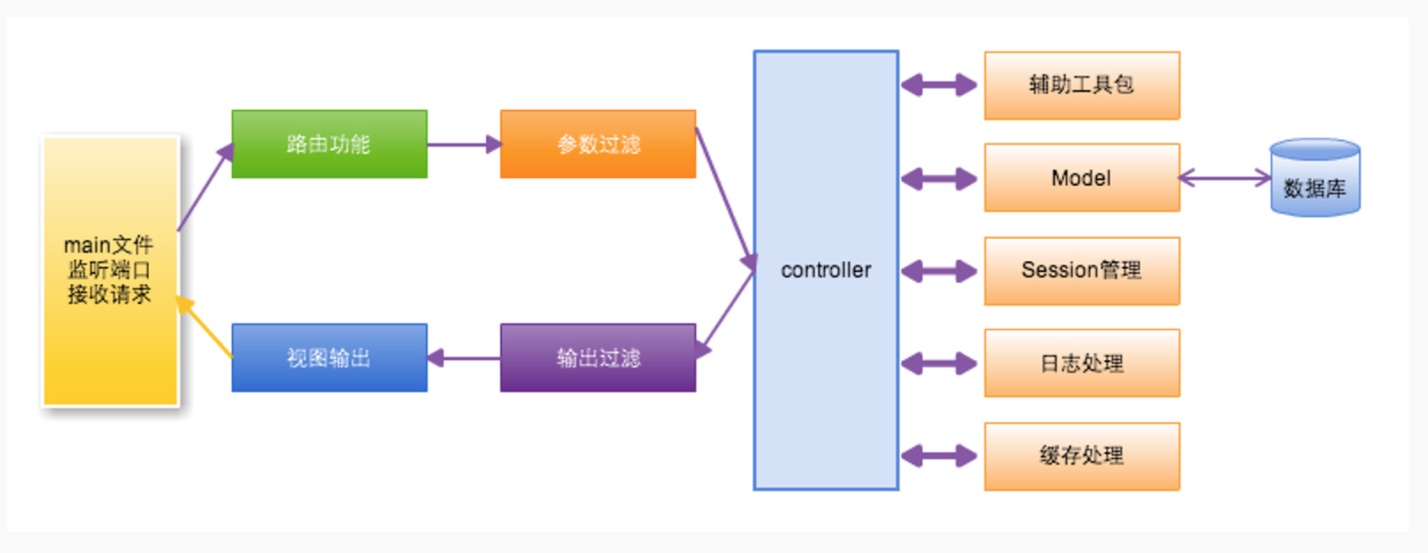


图2.3 beego请求处理流程

如上图所述，当beego框架接收到一个HTTP请求时，会根据路由规则，利用HTTP请求行中的URL结合一些用户添加的参数过滤规则，降不同的请求路由至不同的controller实现类，再根据HTTP请求的请求方法将请求映射到具体实现类的某个特定方法上。

controller 层接收到请求时，负责处理主要的业务逻辑，若业务逻辑涉及到对于数据库的操作，则会把对于数据的操作单独封装为model模块，这是为了将数据操作与业务处理分离，实现系统更好的模块化。在controller 层，也会调用其他的一些模块，类似于调用Session模块用于会话管理，调用日志模块用户处理日志等。

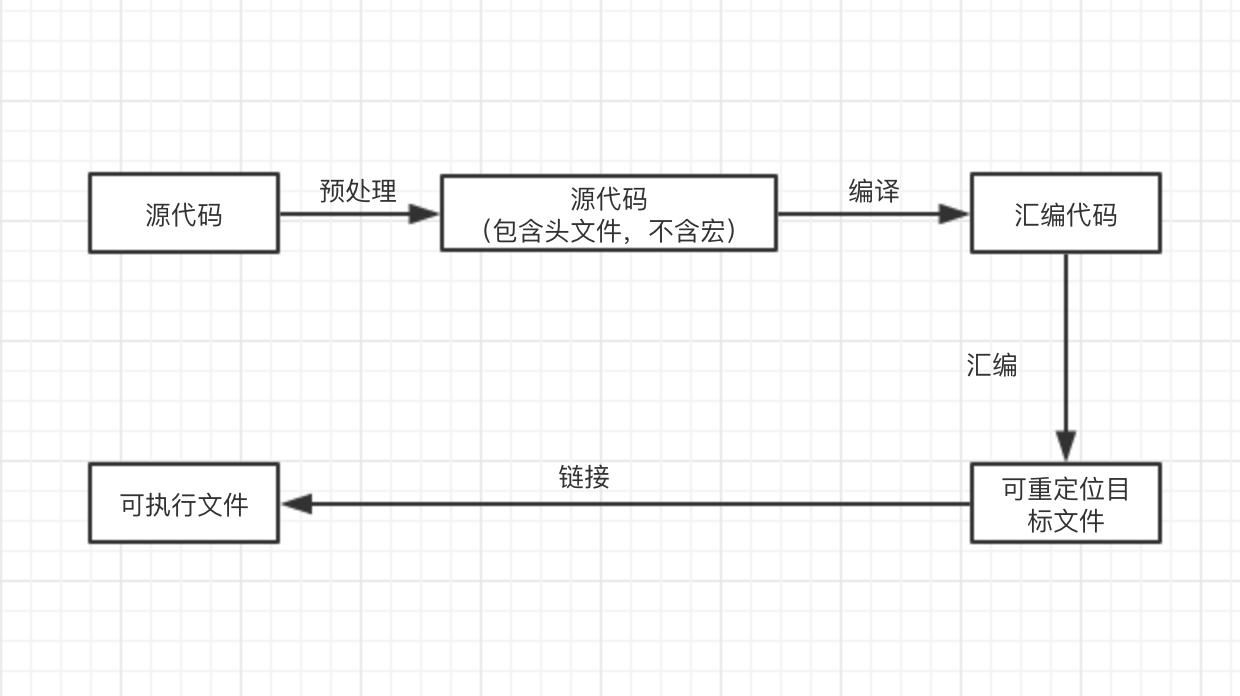
在controller层处理完业务逻辑后，一般会输出一些处理结果给前端展示，beego的视图层根据controller层的输出结果与用户定义的一些输出过滤规则，结合一些事先定义的视图模板，会展现给用户最终的页面效果。

## 2.3动态链接机制

此处的动态链接机制，指的是Linux系统下，C/C++ 程序的动态链接机制。在本系统的实现中，最核心模块的关键功能，依赖于对这一技术的灵活应用。因此，下文对动态链接机制，做概要性的叙述。

### 2.3.1 编译与链接

在Linux系统下，编译器将一个C/C++程序，从源代码转换为一个可执行程序的过程，总体上分为下图所示四个过程。

图2.4 C/C++程序编译流程

在上述四个过程中，前三个过程常被宽泛的统称为编译。编译过程是编译器针对单个源代码文件进行的，负责将单个以文本形式存储的源代码文件，翻译成以二进制形式存储的可重定位目标文件。每个可重定位目标文件，本质上是原程序对应最终可执行文件的部分指令集合，再加上一些用以描述这些指令的段表、符号表等信息构成的。

链接过程是编译器针对多个可重定位目标文件进行的，实质是根据每个可重定位目标文件中的符号表信息，将多个可重定位目标文件链接成一个可执行文件的过程[2]。因此，理解链接过程的核心，是理解可重定位目标文件的符号表信息。

符号，即是对函数或者全局变量的引用。针对链接过程，单个可重定位目标文件的符号表可以理解为由两部分组成，一是当前文件对外提供的符号集合，二是当前文件依赖的外部符号集合。链接过程，即可理解为链接器解析所有的文件，为每个文件的每个依赖的外部符号，在其他文件所对外提供的符号集合中，寻找到一个正确的映射。当程序被链接成单个可执行文件时，每个符号在程序中的相对地址都已确定，再更新可执行文件中每个符号引用为相对地址即可。至此，链接结束，生成最终可执行文件。

### 2.3.2 静态链接库与动态链接库

静态链接库与动态链接库，是代码复用的两种方式。其本质是库开发者开发完源代码后，将源代码编译成许多可重定位目标文件，再将这些文件打包成单个文件的产物[9]。它们的区别在于将库文件链接进可执行文件的过程是不同的。

静态链接库在链接阶段，把所有的库文件中的二进制指令全部链接进可执行文件中。

对于动态链接库，在编译的时候不直接拷贝至可执行代码，而是在将库文件中二进制指令加载到执行共享内存中，并将对动态链接库的引用重定向到对应的地址。

### 2.3.3 动态链接库的使用

动态链接库的使用方法有两种，一种是编译时添加动态链接库，一种是运行时添加链接库。

对于编译时添加链接库，需要在生成可执行程序的链接阶段，在链接命令中为每个链接库添加 –L 参数，用以指明链接器寻找链接库的目录，添加 –l 参数，用以指明动态链接库名。在可执行文件运行前，需要重置 LD\_LIBRARY\_PATH，指明操作系统寻找链接库的目录。在以这种方式使用动态链接时，程序开始运行，动态链接库就会被加载到执行共享内存中。

对于运行时添加动态链接库，则是在运行时动态的将动态链接库加载到执行共享内存中的一种机制。这种方式需要借助Linux操作系统所提供的一组系统调用接口来实现。具体来讲，主要有如下系统调用：

1. dlopen： 根据动态链接库文件名，将动态链接库加载到执行共享内存中，并返回指向这个动态库的引用。
2. dlsym: 根据指向动态库的引用与符号名，获取函数指针或指向全局变量的指针。
3. dlerror: 获取错误信息。

另外，dlsym 系统调用需要的符号名，指的是C语言中的函数名或者全局变量名，而不是符号表中的真实的符号名，因此，若在C++项目中，想要使用这种机制获取某个动态链接库对外暴露的函数接口，那么这个函数接口需要使用extern “C” 关键字修饰[15]。

# 第3章 系统设计

本章首先对系统总体架构设计简单概述，进而对每个模块所采取的设计方案进行了进一步阐述。

## 3.1 系统总体架构设计

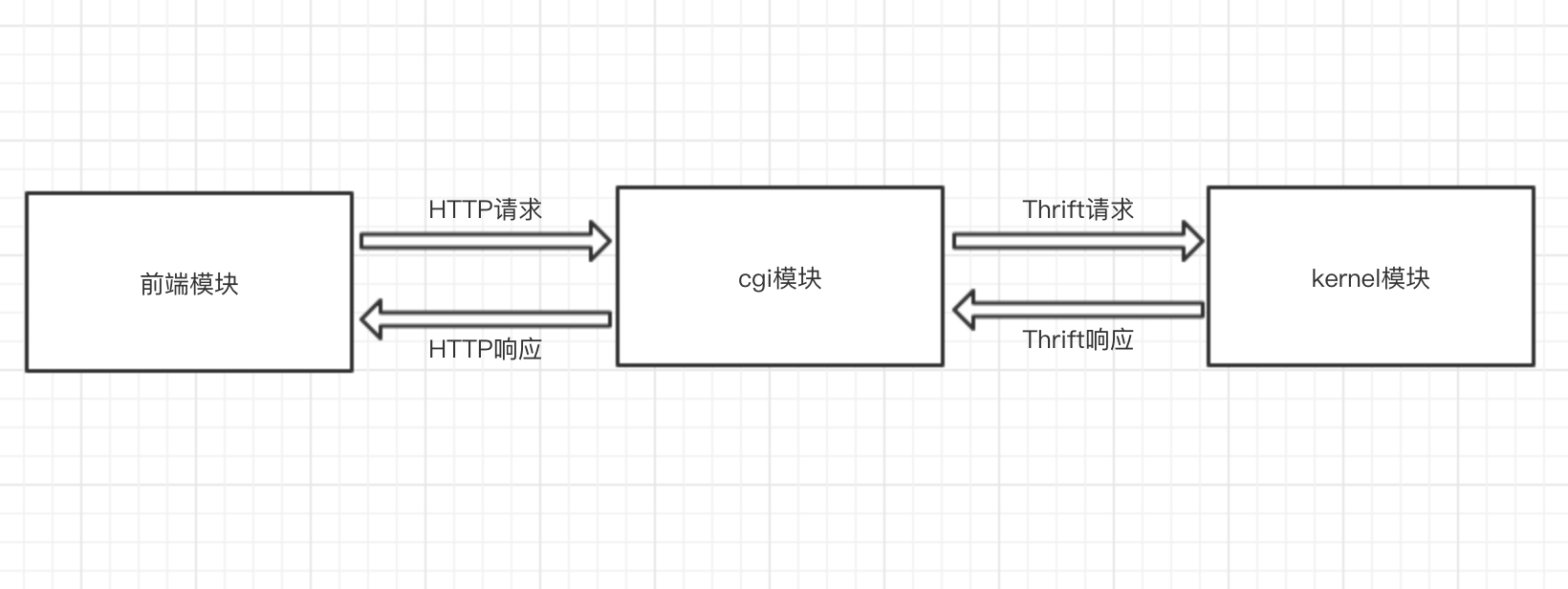
本系统基于C/S架构，在前后端分离的基础上，进一步将后端服务拆分成两个模块，分别负责系统逻辑上较为独立的两个部分。总体架构图如下：

图3.1 系统总体架构图

本系统总体上被划分为三个模块，分别为前端模块、CGI模块、kernel模块。其中，前端模块与CGI模块通过HTTP协议进行通信。kernel实现为一个微服务，使用Thrift RPC向上提供服务，因此，CGI模块与kernel模块通过Thrift协议进行通信。对于本系统的总体架构，每个模块所负责的功能大致如下：

1. 前端模块：主要负责面向用户的页面展示，包括了数据展示与提供给用户本系统的操作接口，并负责一部分与用户的交互逻辑[11]。前端模块仅负责少量的逻辑处理，主要的业务逻辑处理由后端服务负责[13]。
2. CGI模块：CGI模块负责处理来自前端的HTTP请求，并实现系统中主要的业务逻辑。包括用户session管理、服务管理、接口管理、测试用例管理等。其中，对于服务及测试用例的管理，仅涉及对其描述性信息的管理。同时，CGI模块也负责提供系统持久化相关的功能。
3. kernel模块：kernel模块主要负责三个核心的功能实现：一是解析Thrift IDL文件，动态生成加载Thrift微服务的客户端或者服务器；二是向特定服务的某个接口添加请求或者响应数据；三是触发某个Thrift客户端的某个接口，并获取响应。

## 3.2 kernel模块设计

kernel模块基于C++语言与Thrift RPC，被实现为一个向上提供Thrift RPC接口的微服务。因此，kernel模块的设计，即是对于所提供接口输入输出及所应当实现的功能的定义。kernel模块共提供下述四个接口：

1. service ThriftTestKernalService {
2. GetServiceListResponse GetServiceList (1: GetServiceListRequest request);
3. AddServiceResponse AddService (1:AddServiceRequest request);
4. FillDataResponse FillData (1:FillDataRequest request);
5. RequestTriggerResponse RequestTrigger (1:RequestTriggerRequest request);
6. }

下文，依次阐述各个接口功能定义。

### 3.2.1 GetServiceList接口

GetServiceList接口请求与响应参数：

1. **struct** GetServiceListRequest {
2. 1: required ServiceType type;  //0: Client, 1: Server
3. }
5. **struct** GetServiceListResponse {
6. 1: required i32 errorCode;                  //0: succ, 1: kernal internal error
7. 2: required list<string> instanceList;
8. }

在本系统中，不管是Thrift客户端或者是服务器，都统称为服务（即Service），GetServiceList获取本系统中已经存在服务。根据请求参数中的type字段判断请求获取Thrift客户端还是Thrift服务器。正确返回后list中每一个值代指一个服务。

### 3.2.2 AddService接口

AddService接口请求与响应参数：

1. **struct** AddServiceRequest {
2. 1: required ServiceType type;
3. 2: required string serviceName;
4. 3: required string version;
5. 4: required i32 port;
6. 5: required string transport;
7. 6: required string protocol;
8. 7: required string IDLAbsFileName;
9. 7: optional string ip;
10. }
12. **struct** AddServiceResponse  {
13. 1: required i32 errorCode;
14. }

AddService 接口负责在系统中添加一个服务。请求中的type字段指明添加的是Thrift客户端还是服务器。无论添加客户端还是服务器，都需要用服务名、版本、端口、传输层协议、协议层协议与IDL文件来唯一标识一个服务。若添加的是客户端，还需要用IP这个字段表示添加的客户端所对应服务器的地址。

在添加服务成功的前提下，若添加的是服务器，系统则会实例化一个对应的Thrift服务器并启动服务器监听对应的接口；若添加的是客户端，系统则会实例化一个对应的Thrift客户端，并对于客户端的每一个接口，系统都会维护一个服务名与接口名到对应客户端对应接口函数指针的映射，以供后续触发调用。

### 3.2.3 FillData接口

FillData接口请求与响应参数：

1. **struct** FillDataRequest {
2. 1: required ServiceType type;
3. 2: required string serviceName;
4. 3: required string methodName;
5. 4: required string data;
6. }
8. **struct** FillDataResponse {
9. 1: required i32 errorCode;
10. }

FillData负责系统中的某一个服务的某一个接口添加请求或者响应数据。请求中的type字段指明添加的是请求还是响应。而添加的数据格式是Thrift 中TJSONProtocol协议中要求的JSON格式。kernel模块要求，在触发一个请求时，必须调用此接口，为触发的接口添加请求数据，并为下游依赖的接口添加响应数据。

### 3.2.4 RequestTrigger接口

RequestTrigger接口请求与响应参数：

1. **struct** RequestTriggerRequest {
2. 1: required string serviceName;
3. 2: required string methodName;
4. }
6. **struct** RequestTriggerResponse {
7. 1: required i32 errorCode;
8. 2: required string responseJson;
9. }

RequestTrigger接口用于触发一个特定服务的客户端的特定接口，并将该接口响应按照TJSONProtocol协议规定的JSON格式格式化，作为返回值中的responseJson字段。

## 3.3 CGI模块设计

作为系统的中心模块，CGI模块一方面依据接收到的用户HTTP请求，实现用户交互逻辑，另一方面依托于数据管理进行系统主要的业务逻辑的处理。因此，对于CGI模块的设计，分别以以下三个方面介绍。

### 3.5.1数据库设计

在设计一个系统时，进行清晰明确的数据库设计是极为重要的一环。设计数据库时，需要对系统所提供的功能有清晰的认知，并基于这种认知与系统的大致实现逻辑，抽象出系统中的一些公共的抽象概念。然后明确哪些东西是需要持久化的，哪些东西是不需要持久化的。下文，依据数据库设计时设计的表结构，对以下需要持久化的概念进行介绍。

User : 系统依托User表进行用户管理。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 字段名 | 字段类型 | 字段长度 | 字段约束 | 是否主键 |
| userId | int | 11 | not null | 是 |
| userName | varchar | 30 | not null | 否 |
| password | varchar | 30 | not null | 否 |

ServiceMetaData：记录Thrift微服务的描述性信息，不作客户端与服务器的区分。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 字段名 | 字段类型 | 字段长度 | 字段约束 | 是否主键 |
| serviceId | int | 11 | not null | 是 |
| serviceName | varchar | 30 | not null | 否 |
| idl | varchar | 256 | not null | 否 |
| port | int | 11 | not null | 否 |
| transport | varchar | 30 | not null | 否 |
| bufftransport | varchar | 30 | not null | 否 |
| protocol | varchar | 30 | not null | 否 |
| ip | varchar | 30 | not null | 否 |

ServicePatterm：用以描述对于某一个测试项目中服务的映射关系。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 字段名 | 字段类型 | 字段长度 | 字段约束 | 是否主键 |
| servicePattermId | int | 11 | not null | 是 |
| userId | int | 11 | not null | 否 |
| clientServiceId | int | 11 | not null | 否 |
| serverServiceIds | varchar | 50 | not null | 否 |

InterfaceMetaData：用以记录某个服务的某个接口的描述性信息。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 字段名 | 字段类型 | 字段长度 | 字段约束 | 是否主键 |
| interfaceId | int | 11 | not null | 是 |
| serviceId | int | 11 | not null | 否 |
| interfaceName | varchar | 30 | not null | 否 |
| reqDesc | varchar | 1024 | not null | 否 |
| rspDesc | varchar | 1024 | not null | 否 |

InterfacePatterm：用以记录某个测试项目中接口的映射关系。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 字段名 | 字段类型 | 字段长度 | 字段约束 | 是否主键 |
| interfacePattermId | int | 11 | not null | 是 |
| servicePattermId | int | 11 | not null | 否 |
| clientInterfaceId | int | 11 | not null | 否 |
| serverInterfaceIds | varchar | 50 | not null | 否 |

TestCase：用以记录测试用例，某个测试用例是针对某个接口映射关系的。clientInterfaceData代表上游客户端的请求数据；serverInterfacesData代表下游服务器的响应数据。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 字段名 | 字段类型 | 字段长度 | 字段约束 | 是否主键 |
| testCaseId | int | 11 | not null | 是 |
| interfacePattermId | int | 11 | not null | 否 |
| clientInterfaceData | varchar | 500 | not null | 否 |
| serverInterfacesData | varchar | 500 | not null | 否 |

### 3.5.2 CGI模块接口设计

CGI模块接口设计即是定义系统与用户进行交互的HTTP接口，这本身也定义了系统所提供的功能。因为HTTP接口是暴露给用户供用户使用本系统的接口，其定义系统功能的同时，也定义了本系统中需要被用户认知的几个概念。因此，下文会结合本系统中几个核心的需要被用户认知的概念阐述HTTP接口的设计。

用户。与用户相关的接口包括登录、注册。

服务模式。对于某个被测微服务，下游常会依赖一个或多个微服务。这个被测微服务与下游微服务的映射关系被称为服务模式。与服务模式相关的接口有为当前用户添加服务模式，查询当前用户所有服务模式，更新某个服务模式，删除某个服务模式。

接口模式。对于特定的服务模式，针对被测微服务的每一个接口，被测微服务可能会调用下游服务中的不同接口。这种被测服务的某个接口与下游服务的多个接口的映射关系，被称为接口模式。与接口模式相关的接口有为某个服务模式添加接口，删除接口，更新接口，以及查询当前用户所有的接口模式。另外，因用户添加测试用例时需要知晓每个接口的请求或者响应的字段名和类型名，因此，当添加或更新接口模式时，需返回给用户对应所有接口的请求或响应的字段名和类型名。

测试用例。对于某个接口模式的一个测试用例，包含两部分数据。一是被测试微服务接口的请求数据，二是每个下游依赖服务接口的响应数据。与测试用例相关的接口有为某个接口模式添加测试用例，删除测试用例，更新测试用例，查询当前用户测试用例，以及使用某个测试用例触发某个接口模式并获取响应。

### 3.5.3 CGI架构设计

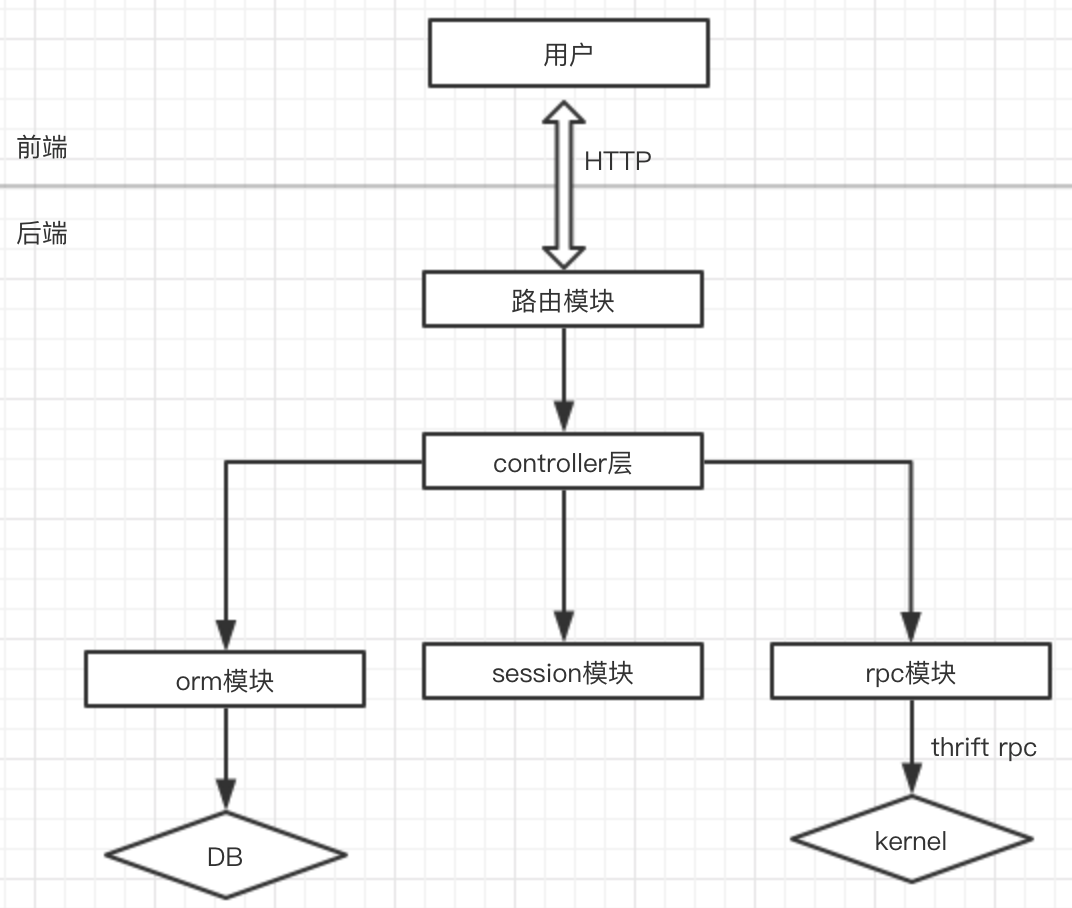
CGI模块为提供应有的功能，其整体架构如下图所示。

图3.2 CGI模块设计图

orm模块负责封装对数据库的操作。通过维护go语言中Struct与数据库表的映射，提供通过go语言操作数据库的接口。

session模块负责进行会话管理，维护用户的登录状态。

rpc模块负责封装远程过程调用操作，给予CGI模块调用kernel模块服务的接口。

路由负责对HTTP请求进行路由，将不同的请求路由给不同的控制器进行处理。

controller模块负责解析请求，处理逻辑，必要时调用orm模块、session模块及rpc模块提供的服务，并生成响应返回给用户。

## 3.4 本章小结

本章对本系统的总体架构设计进行了简要描述，并针对系统中较为重要的两个模块，即kernel模块与CGI模块，分析了其设计方案并明确了其所提供的系统功能。对于kernel模块，考虑到其作为一个Thrift微服务向CGI模块提供服务的事实，主要通过描述kernel模块对外暴露的接口输入输出与实现功能的角度阐述其设计方案。而对于CGI模块，则在数据库设计、HTTP接口设计、整体架构设计三个角度阐述了其设计方案。

# 第4章 系统关键功能实现

本章聚焦于系统两个关键核心功能的实现机制。分别是添加服务功能实现与填充数据功能实现。

## 4.1添加服务功能实现

添加服务实际是指向系统添加Thrift服务器或Thrift客户端。二者服务生成逻辑大致相同而在文本处理阶段略有不同，下文详细阐述。

### 4.1.1 何时添加服务

考虑到对一个团队来讲，其所负责开发的微服务总数不会太大，而对于每一个微服务都启动一个服务器和客户端在性能上也是可以接受的，并且启动一个服务是一个用户可感知的很耗时的过程（大约几秒到十几秒），因此，kernel模块仅提供添加服务的接口呢，而不提供删除服务的接口。

由于系统需要尽快启动所有测试每个微服务所需要的Thrift客户端和服务器，因此系

统会在用户添加服务模式时就会添加服务，而不是等到用户触发某个请求时，真正需要相应的服务时再去启动服务。

当用户添加一个服务模式时，实际上提供了一个被测试微服务与多个下游依赖服务的映射关系。为完成这个测试，系统需要启动被测试微服务的Thrift客户端与多个下游依赖的Thrift服务器。

因此，CGI模块在接收到添加或者更改服务模式时，CGI会对用户提供的服务模式中的每个服务向kernel发起AddService请求，并传递给kernel启动一个服务所需要的服务类型、服务名、服务版本好、服务端口、通信协议、IDL文件等参数。

### 4.1.2 添加服务流程概述

对于kernel模块，从接收到AddService请求，到向系统添加服务完成，所作的处理流程大致如下图4.1所示。

由图4.1所述，当kernel模块接收到一个AddService的请求时，系统为完成添加一个服务的功能，需要大致分为四步[4]，依次如下。

1. 调用脚本使用thrift编译器编译IDL文件，生成服务代码。
2. 使用文本处理脚本基于生成代码构造完整的服务端代码或客户端代码。
3. 使用编译脚本调用gcc编译器将C++代码编译成动态链接库。
4. 在程序中将对应的动态链接库链接进当前进程。

当实现上述系统逻辑时，首先应当考虑的是关于如何在程序中执行Linux shell脚本的问题。在考察了多种执行shell脚本的方式后，最终选用了使用popen系统调用建立从shell进程到当前进程的方式[8]。选择此技术方案的主要原因是可以通过shell进程的标准输出与当前进程的标准输入之间的数据传输实现两进程之间的灵活通信。实际的做法是

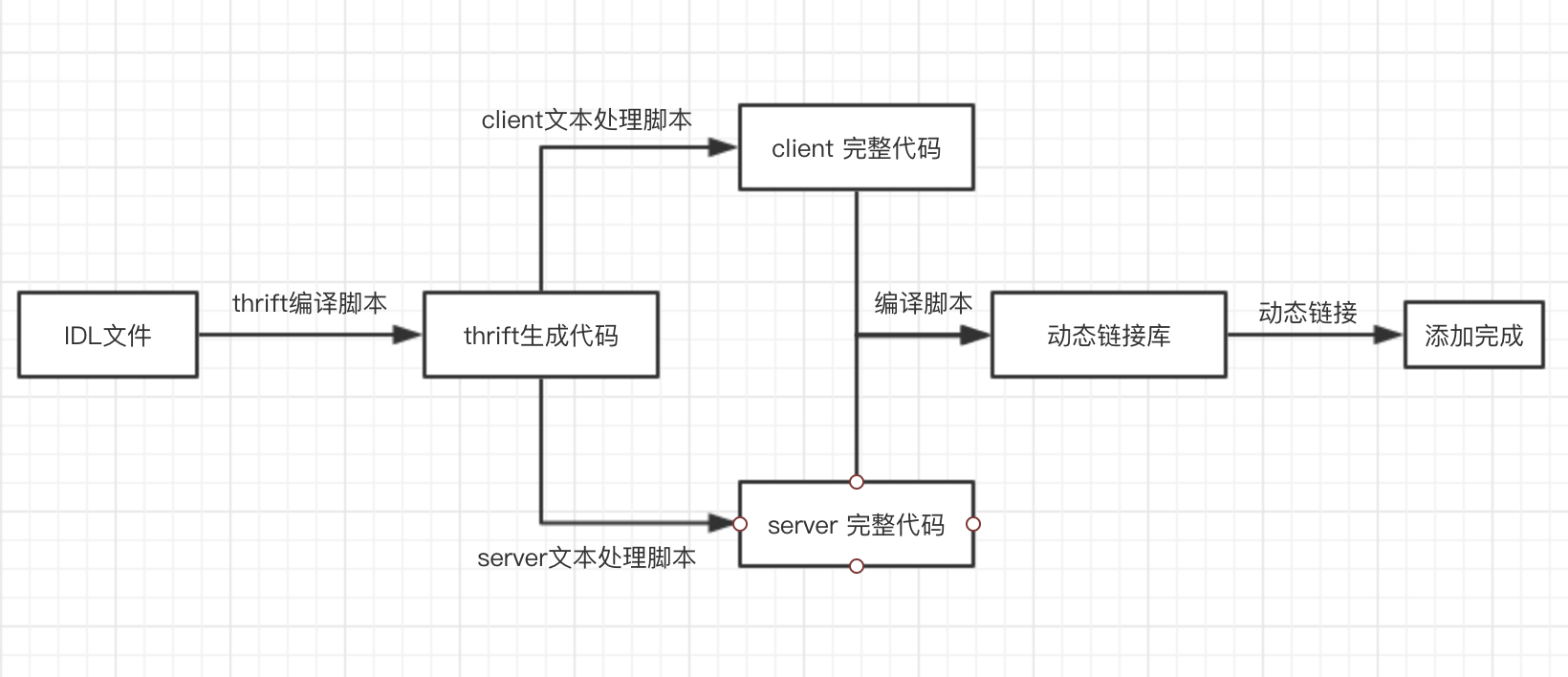


图4.1 添加服务流程图

若shell进程输出0，则代表shell进程成功执行并正确返回；否则，shell进程执行失败，并会通过标准输出输出一些错误信息。kernel进程通过标准输入获取到这些错误信息后可以显示给用户，也可以通过打日志记录错误信息。

### 4.1.3 添加服务要点详述

#### 4.1.3.1 文本处理阶段

文本处理阶段，系统会根据Thrift编译器的生成代码构造出完整的服务器代码或者客户端代码。

首先，应当阐明当前进程构造代码所使用的技术。在研究了不同服务的服务器或者客户端的代码实现之后，系统最终采用了shell脚本基于多个代码片段，进行文本处理，并最终整合成单个目标文件的技术方案。选择此技术方案的原因是认识到不管是对于多个服务的服务器还是客户端，其整体代码结构都是一致的，只有特定代码片段会根据服务相关特征的变化而变化这一事实。

其次，应当定义构造代码的结构与提供的功能。这一方面是明确完整的服务器代码或客户端代码除了生成代码之外的部分，另一方面是需要定义当当前被构造服务作为动态链接库链接进当前进程时，需要提供给当前进程的接口。

若代添加服务类型为服务器时，文本处理阶段需要构造出针对代添加服务的Handler类，并在Handler中对于带添加服务的每个方法，实现其方法体；除Handler之外，还需要按照请求中的协议类型、端口号等参数实现一个服务器。对于动态链接库暴露给当前进程的接口（即函数指针），被设计成如下格式：

1. typedef apache::thrift::server::TSimpleServer\* (\*Server\_Entry)(int port);

对于每一个生成的服务器，其Server\_Entry 的命名为 get\_SERVERNAME\_server。其中，SERVERNAME代指服务名。根据这个命名规范，系统进程可以根据服务名获取到这个接口的函数指针。而调用这个接口，可以获得一个代添加服务的服务器，系统获得这个服务器，即可完成启动代添加服务的服务器的目的。

若代添加服务类型为客户端时，需要为代添加服务每个方法都暴露给当前进程一个接口，以便系统根据此接口触发客户端方法。对于动态链接库暴露给当前进程的接口（即函数指针），被设计成如下格式：

1. typedef int (\*Client\_Entry)(std::string ip, int port);

对于生成的客户端的每一个接口，其Client\_Entry 的命名为 SERVICENAME\_METHODNAME\_PORT， 其中SERVICENAME代指服务名，METHODNAME代指方法名，PORT代指端口号。根据这个命名规范，系统进程可以根据服务名、方法名、端口号从动态链接库中获取到这个接口的函数指针，根据这个函数指针，即可完成触发对应接口的目的。

对于每个客户端接口实现的方法体，需要完成下述四步操作：

1. 根据请求中的协议类型、服务名等参数实现一个客户端。
2. 获取请求参数。
3. 利用客户端与请求参数获取响应。
4. 传递响应给当前进程。

#### 4.1.3.2 编译阶段

编译阶段的主要任务是，利用gcc编译器将某个服务的生成代码编译成一个动态链接库。因本系统支持对同一个服务同时添加服务器与客户端，因此上述“某个服务的生成代码”在不同的情况略有不同。若系统不含代添加服务的服务器或者客户端，上述生成代码代指Thrift编译器生成的代码与系统构造的代码。若在系统已经包含代添加服务的服务器情况下添加客户端，或者，在系统已经包含代添加服务的客户端情况下添加服务器，上述生成代码单指系统构造的代码，而不包含Thrift编译器生成的代码。此逻辑是为了避免因符号重定义引发链接错误。

#### 4.1.3.2 动态链接阶段

动态链接阶段系统将利用dlopen系统调用动态加载动态链接库，并利用dlsym系统调用获取动态链接库提供给系统的接口。

对于服务器，利用dlsym获取的接口提供的功能是获取一个代添加服务的服务器指针，利用这个服务器指针，系统可以实现启动对应服务器的目的。

对于客户端，利用dlsym获取的接口提供的功能是触发对应的客户端接口，并获取下游返回的响应。

## 4.2 填充数据功能实现

首先应当明确，系统填充数据功能是为保证当用户触发一个请求时，客户端可以正确读取用户指定的测试用例中的请求数据，下游服务器可以正确读取到用户指定的测试用例的响应数据。与上文添加服务功能实现逻辑仅涉及kernel模块一个模块不同，实现填充数据功能的逻辑在系统三个模块即前端模块、CGI模块与kernel模块均有分布。

在阐述系统填充数据功能实现逻辑之前，需要现先明确Thrift中TJSONProtocl协议规定的Json数据格式。

### 4.2.1 TJSONProtocl协议

TJSONProtocl协议是Thrift协议层的一种协议。协议层下层是传输层，传输层定义了Thrift获取字节流的方式，而协议层定义了字节流与Thrift标准数据类型的编解码方式。而TJSONProtocl协议规定了一种特殊的Json格式，只要字节流按这种这种特殊的Json格式编码，Thrift就可以读取为Thrfit数据类型的值。这种编码方式具体如下文所述。

1. 所有Thrift整形与浮点型及Thrift字符串类型，都被编码为Json字符串。
2. Thrift二进制类型，经过base64编码后作为Json字符串。
3. Thrift结构体作为一个Json对象，对于每个Thrfit结构体属性，属性ID作为Json对象属性名，属性值是个单属性的Json对象，属性名是代表Thrift结构体属性数据类型的简单字符串，属性值是Thrift属性值。（Thrift数据类型与简单字符串的映射关系如下：tf对应bool；i8对应byte；i16对应16位整形；i32对应32位整形；i64对应64位整形；dbl对应双精度浮点型；rec对应结构体；map对应map；list对应list；set对应set。）
4. Thrift list或者set类型被编码为一个Json数组。第一个元素是数据类型；第二个元素是元素个数；剩下的元素依次是Thrift list或set的各个元素。
5. Thrift map类型被编码为一个Json数组。前两个元素表示map类型key与value的数据类型。第三个元素表示map类型元素个数。第四个元素是一个包含所有map映射关系的Json对象。

在填充数据功能的实现过程中，TJSONProtocl协议在多处被利用，主要利用了其字节流与Thrift结构体的相互转换机制，实现了Thrift微服务接口请求或响应的持久化与网络传输[10]。

### 4.2.2 填充数据流程

为完成当用户触发一个测试接口时，系统系统中已有的客户端或服务器能找到对饮的请求数据或者响应数据，用户与系统的交互过程与系统总体的处理流程如图4.2所示。

下文对上述流程作详细阐述。

1. 用户添加接口模式时，会通过前端模块将接口的映射关系传递给CGI模块。由于添加测试用例时前端模块需要知道Thrift结构体每个属性的属性名与类型名的映射关系，

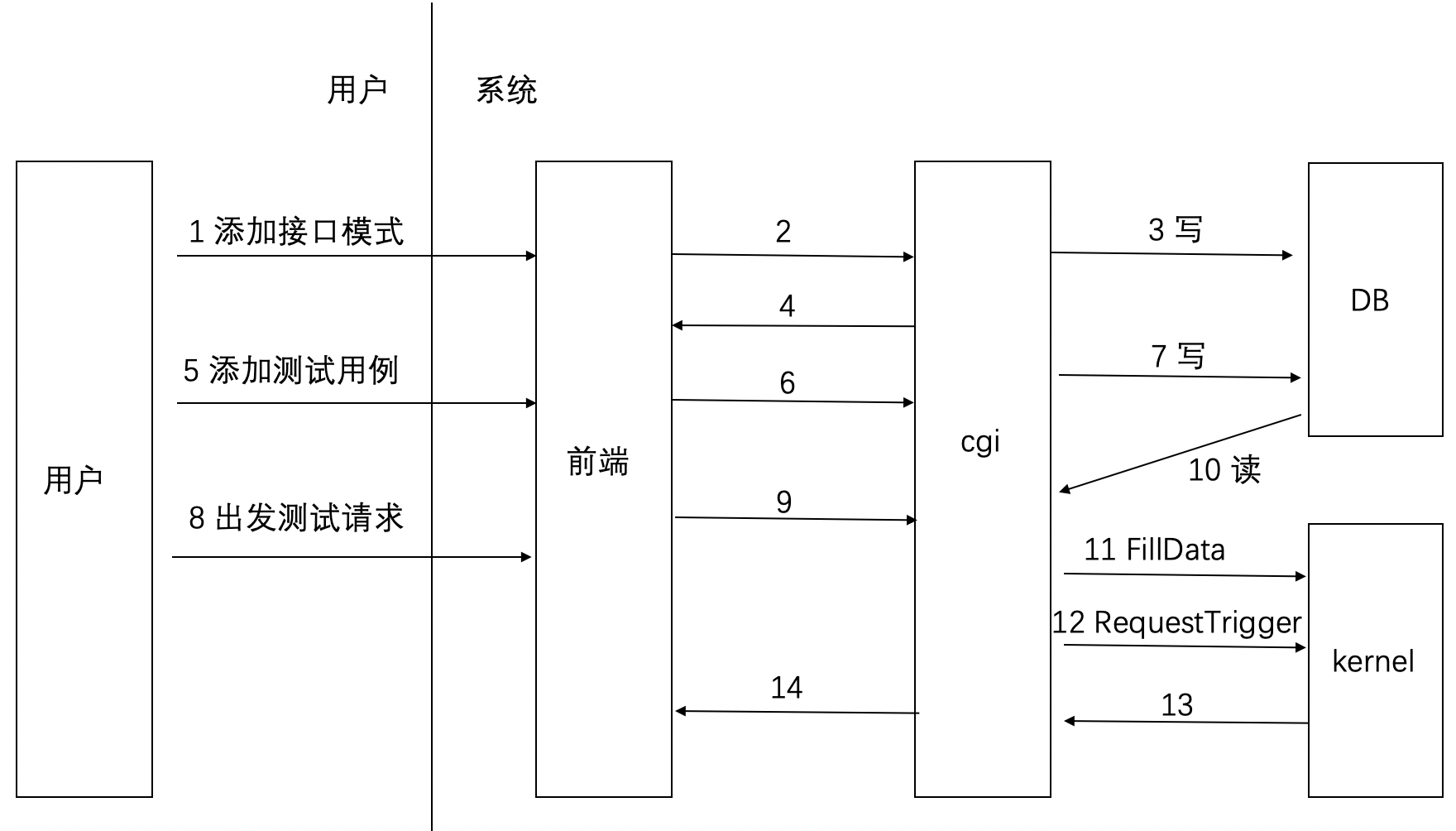


图4.2 填充数据流程图

因此CGI模块会根据不同的接口名，解析IDL文件，获取接口请求或响应的属性名与类型名的映射关系，并返回给前端模块。操作3写数据库的操作是将接口的请求或响应属性名及类型信息作持久化，以便后续为已有接口模式添加测试用例时无需再次解析IDL文件操作。（解析IDL文件是一个十分耗时的工作。）

2. 当用户添加测试用例时，实质上提供了上游被测接口的请求数据与下游依赖接口的响应数据。前端模块接收到数据后，会结合请求或响应的属性名与类型信息映射，手工构造一个符合TJSONProtocol规范的请求或响应的Json对象，并序列化该Json对象。然后将序列化出的文本作为一个测试用例持久化到数据库。

3. 当用户发出测试请求时，会指明实用的测试用例ID。CGI模块会根据测试用例ID读数据库获取具体的测试用例，并对每个请求或响应调用kernel模块的FillData接口。然后CGI模块会调用RequestTrigger接口获取被测接口的响应数据并返回给前端模块展示给用户。之所以不在用户添加测试用例时调用FillData接口是因为kernel模块不负责数据维护而应将对测试用例的维护逻辑放置在CGI模块上。

### 4.2.3 FillData接口实现逻辑

kernel模块一个核心设计理念是将服务与测试数据分离开来。因此每次触发请求前都要调用FillData接口。而之所以可以实现服务与测试数据的分离，原因在于当被测接口读取请求或者下游依赖接口读取响应时，使用了直接从指定文件读取的方式。以被测接口读取请求为例，其中读取请求的代码片段模版如下：

1. REQ\_TYPE REQ\_NAME;
2. boost::shared\_ptr<TTransport> REQ\_NAME\_itrans(**new** TSimpleFileTransport("REQ\_PATH", **true**, **true**));
3. boost::shared\_ptr<TProtocol> REQ\_NAME\_iprot(**new** TJSONProtocol(REQ\_NAME\_itrans));
4. REQ\_NAME.read(REQ\_NAME\_iprot.get());

其中，REQ\_TYPE代指请求的数据类型，REQ\_NAME代指请求名，REQ\_PATH代指读取特定请求的文件路径。

由上代码片段可知，系统在读取一个请求时，使用TsimpleFileTransport作为传输层，从文件中直接获取字节流而不做其他编解码工作。而协议层是用TJSONProtocol协议使得只要文件中的文本服务TJSONProtocol协议的规范，就可以正确读出请求值。

由上节可知，用户添加测试用例时，前端会手工构造一个符合TJSONProtocol规范的请求或响应的Json对象，而当系统调用kernel模块的FillData接口时，传递的参数正是这个对象的序列话结果。

综上所述，当kernel模块接收到一个FillData请求时，只需将其请求中的请求或响应参数，根据服务名与方法名存储在不同的文件中即可。

## 4.3 本章小结

对于系统的功能实现，本章并非采用对每个模块的每个接口描述其实现原理的描述方案，而是通过描述两个关键功能的实现原理，揭示了本系统核心部分的实现逻辑。对于添加服务功能，主要涉及到kernel模块，而实现这个功能包括四个主要流程：使用Thrift编译器编译IDL文件；文本处理；C++编译器编译代码生成动态链接库；动态链接进当前进程。对于填充数据的功能实现，首先介绍了TJSONProtocol协议定义的Json格式，其次根据用户触发接口的流程分析了系统进行数据填充的逻辑，最后具体分析了kernel模块FillData接口的实现逻辑。

# 第5章 系统测试

本章主要针对系统的两个核心模块，设计合适的测试方案，并展示一些关键的测试结果。其中的测试方案，不包含对单个函数或文件的单元测试，也不包含对系统的性能测试，而只涉及对模块整体的所做的功能性测试。

## 5.1 kernel模块测试

kernel模块是系统的核心模块，作为一个Thrift微服务，为保证kernel每个接口的正常工作，系统另外实现了一个针对kernel模块的Thrift客户端对其测试。下文对测试结果的展示中，客户端调用接口的命令如下，其中 path/to/json 指的是指向请求的Json文件路径。

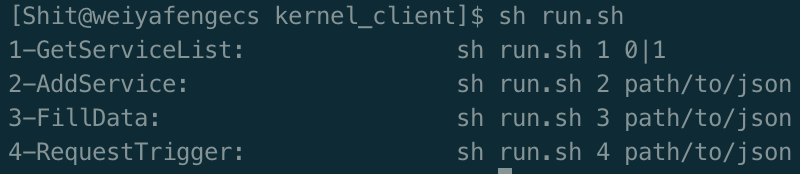


图5.1 kernel客户端调用命令

在下文的测试结果展示中，对kernel模块的测试方案是利用kernel模块的Thrift客户端，依次调用kernel模块的各个接口，完成分别添加calculate服务的客户端和服务器的操作，并调用calculate 服务的Calculate方法。

calculate 是个简单的Thrift微服务，其接口提供了简单的计算整数运算的操作，其IDL文件如下：

1. **namespace** cpp calculator
3. **enum** OPType{
4. ADD = 0;
5. SUB = 1;
6. MUL = 2;
7. DIV = 3;
8. }
10. **struct** CalculateRequest {
11. 1: required i32 left;
12. 2: required i32 right;
13. 3: required OPType op;
14. }
16. **struct** CalculateResponse {
17. 1: required i32 errorCode;
18. 2: required string errorMsg;
19. 3: required i32 result;
20. }
22. service CalculatorService {
23. CalculateResponse Calculate(1:CalculateRequest req);
24. }

### 5.1.1 AddService接口

请求参数：



图5.2 AddService接口请求

调用接口及响应：

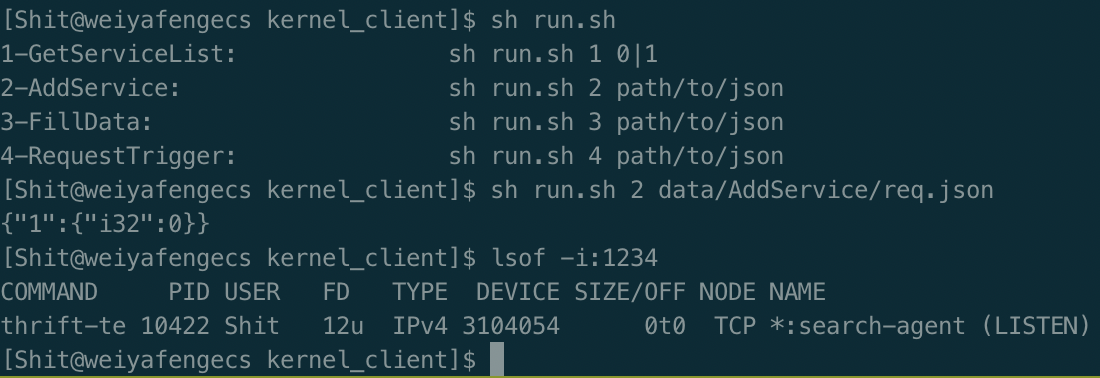
s

图5.3 AddService接口响应

请求参数中服务类型为1，代表调用AddService接口是在系统中添加一个Thrift服务器。由响应可知，接口调用成功，对应的Thrift服务器已经启动，并成功监听1234端口。是用同样的方法，亦成功的添加了calculate的客户端。

### 5.1.2 GetServiceList接口

使用kernel模块的客户端查看系统所有的Thrift服务器：

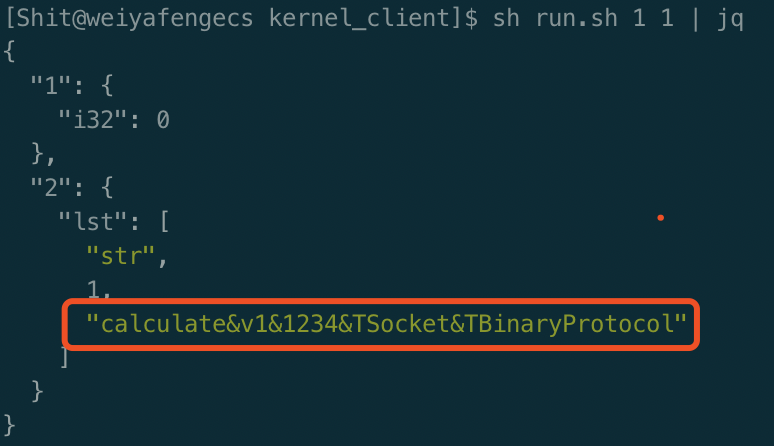


图5.4 查看系统中Thrift服务器

依上图，系统仅有一个calculate服务的服务器，符合预期。

使用kernel模块的客户端查看系统所有的Thrift客户端：

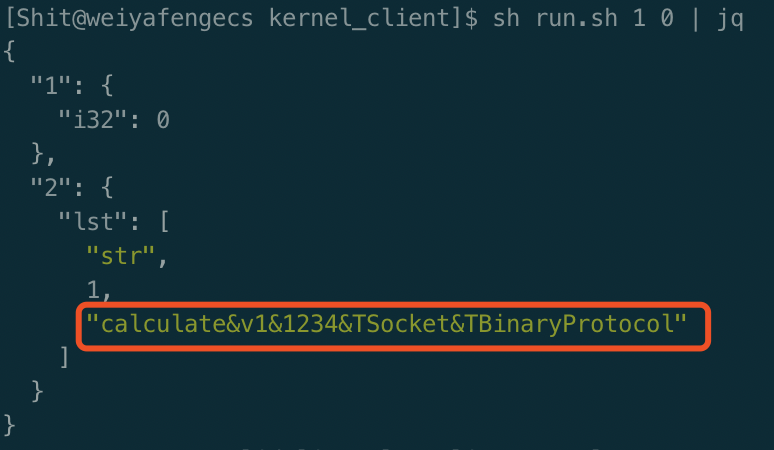


图5.5 查看系统中Thrift客户端

依上图，系统仅有一个calculate服务的客户端，符合预期。

### 5.1.3 FillData接口

为完成calculate服务客户端成功调用calculate服务服务器的目的，需要填充客户端Calculate接口的请求与服务器Calculate接口的响应。而无论填充的请求数据时什么，当触发calculate客户端的Calculate接口时，都可以返回填充的响应数据，即可证明系统运行符合预期。

调用FillData接口添加被测接口请求值：

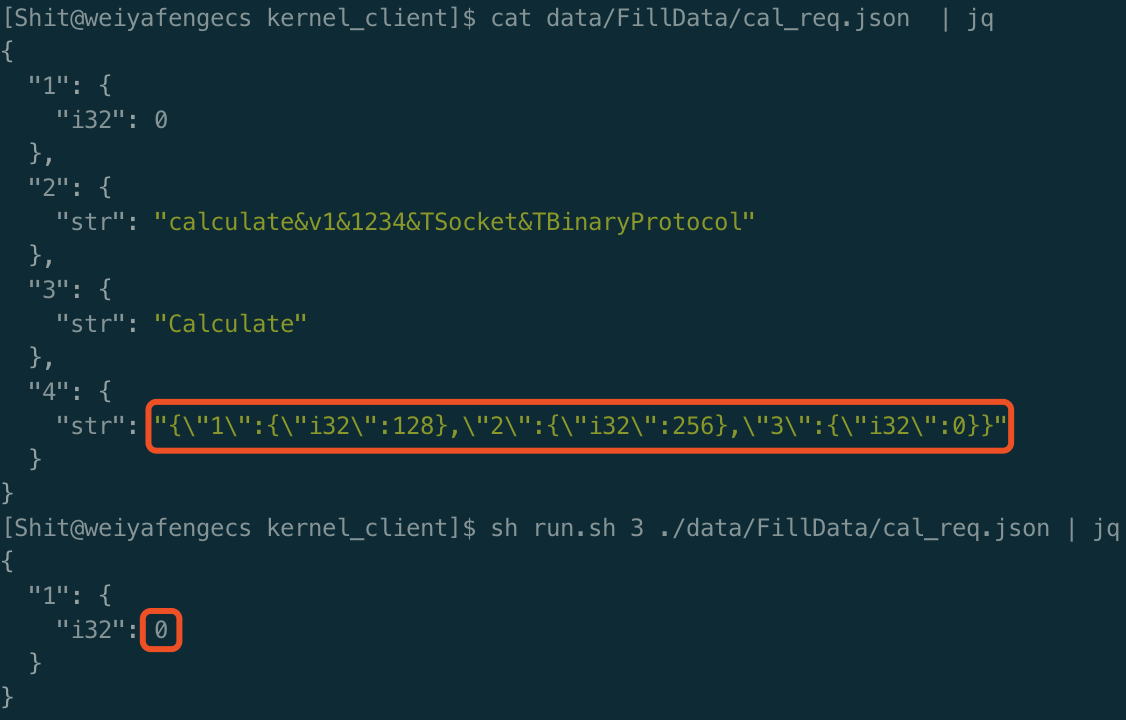


图5.6 FillData被测接口请求

由上图可知，FillData接口将被测接口的请求数据序列化成一个Json字符串，转义后作为其请求参数中的一个。由kernel模块返回值为0可知，符合预期。

调用FillData接口添加下游接口响应值：



图5.7 FillData下游接口响应

依上图，填充数据成功。因此，若触发待测接口，得到此处填充的响应的接口，则说明kernel模块FillData接口符合预期。

### 5.1.4 RequetTrigger接口

使用kernel模块客户端通过RequetTrigger接口触发calculate服务的Calculate接口，及响应结果如图：

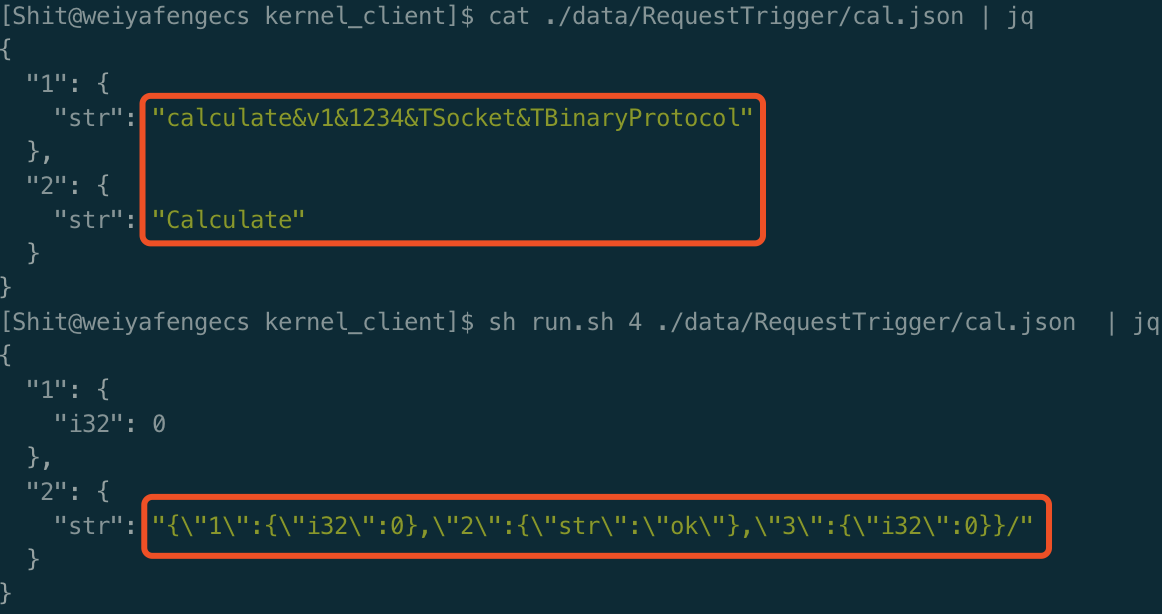


图5.8 RequetTrigger接口触发结果

如上图所述，在请求中指定触发calculate服务的Calculate接口，而获取的响应是上节中数据填充接口填充的响应的数据，符合预期，系统运行正常。

## 5.2 CGI模块测试

对CGI模块整体的功能性测试，即是对CGI模块对外暴露的HTTP接口功能的验证。因此为更加方便验证特定对应HTTP接口的请求的响应是否符合预期，使用了业界常用的HTTP API管理工具Postman。而鉴于篇幅所限，下列并不会列出所有的HTTP接口测试数据，而只对其中较为重要的接口测试结果做展示。

为实现对CGI模块的测试，测试方案采用单独启动一个简单的test\_service微服务，其提供的唯一接口功能仅仅只是调用了calculate微服务的Calculate接口。对CGI模块的测试方案关注于对test\_service微服务的测试上。

### 5.2.1 添加服务模式

添加服务HTTP请求体如图5.9所示：

在添加服务时，前端需要将服务的IDL文件传输给CGI模块，由于Json协议字符串不支持换行符，因此在传输时将换行符替换为@符号。

添加符号的HTTP请求响应如图5.10所示：



图 5.9 添加服务HTTP请求体



图5.10 添加服务HTTP响应

如图5.10上，添加服务成功，并返回服务模式ID。利用这个服务模式ID，用户可以对这个添加的服务模式添加接口模式。

### 5.2.2 添加接口模式

添加接口模式需要用户针对某个服务模式，指明被测微服务的接口与对于每一个下游依赖接口，其服务名与方法名的映射关系。

添加接口模式HTTP请求体如5.11所示：

添加接口模式HTTP响应如5.12所示：



图5.11 添加接口模式HTTP请求体



图5.12 添加接口模式响应

如图5.12所示，添加接口模式成功，并返回接口模式ID，利用这个接口模式ID，用户可以对这个接口模式添加测试用例。CGI模块亦返回请求或响应的结构体名，及结构体每个属性名和Thrift属性类型。

### 5.2.3 添加测试用例

添加测试用例是针对某个接口模式的，需提供被测试接口的请求数据与下游接口的响应数据。

添加测试用例HTTP请求体如图5.13：



图5.13 添加测试用例HTTP请求体

如图5.13，calculate服务Calculate接口的响应值被置为0，而test\_service服务的TestInterface接口所实现的功能仅仅将Calculate接口的响应值透穿出来，因此若触发test\_service服务的TestInterface接口，得到响应值为0则证明系统运行正常。

添加测试用例HTTP响应如下



图5.14 添加测试用例HTTP响应

如上，添加测试用例成功，并返回测试用例ID。当用户触发某个被测接口时，需要通过测试用例ID指定测试用例。

### 5.2.4 触发测试用例

触发测试用例HTTP请求体如图5.15所示。

当用户触发一个测试用例时，仅需指明测试用例ID。

触发测试用例HTTP响应如图5.16所示。

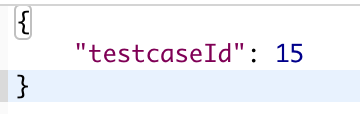


图5.15 触发测试用例HTTP请求体

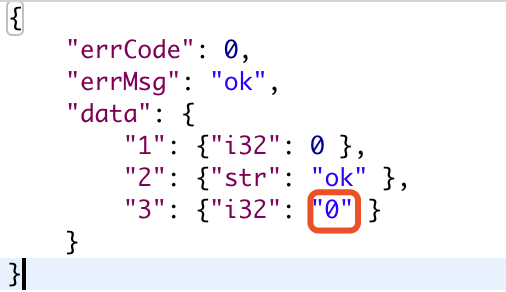


图5.16触发测试用例HTTP响应

由上图，触发测试用例成功，并且test\_service服务的TestInterface接口得到响应值为0，符合预期，系统运行正常。

## 5.3 本章小结

本章对kernel模块与CGI模块进行了总体上的功能测试及对部分测试结果进行了展示。对于kernel模块，使用另外单独实现一个kernel服务的Thrift客户端，依次调用各个接口的方式，验证每个接口是否能符合预期的运行。对于CGI模块，使用HTTP API 管理工具Postman对CGI模块进行了模拟调用，并依据响应验证CGI模块功能是否符合预期。

# 第6章 结论

## 6.1总结

对于基于Thrift RPC的微服务开发人员来讲，当完成对某个微服务的新特性的开发工作之后，如何根据微服务新特性，构造新的业务场景，并对微服务的某些新特性进行测试一直是个较为麻烦的问题。而在实际的生产环境下，为解决这一如何测试的问题开发人员也常付出许多努力用以完成许多与需求开发无关的工作。本系统的设计与实现，为业界这一问题的解决提供的一种较为可行的思路，即自动化的构造生成运行Thrift服务器与客户端，并利用开发人员构造的测试用例触发待测服务的特定接口。在开发人员完成特定微服务的开发工作后，仅需针对新的业务场景，构造测试用例，以验证微服务的新特性是否符合预期。综上，在本系统的帮助下，开发人员可以将测试工作的内容关注在针对新场景的测试用例的构造上，而不用浪费精力在与实际需求无关的开发工作上。

应当认识到的是，目前本系统仍存在一些问题。例如在被测试微服务的IDL文件中，暂不支持Thrift Struct类型的嵌套使用；暂不支持添加使用不同协议的相同服务；当系统调用脚本服务时，系统会给用户较长时间延迟，用户体验受损等问题。因此为解决以上问题，并使系统可以在生产环境中可以真正使用，本系统仍具有很大的发展空间。

参考文献

1. N. Alshuqayran, N. Ali and R. Evans, "A Systematic Mapping Study in Microservice Architecture," 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA), Macau, 2016, pp. 44-51.
2. Slee, Mark & Agarwal, Aditya & Kwiatkowski, Marc. (2019). Thrift: Scalable cross-language services implementation.
3. Thrift Document[EB]. Apache Thrift: https://thrift.apache.org/docs/, 2017年.
4. Protocol Buffer Tutorial[EB]. Tutorial|Protol Buffer: https://developers.google.com/protocol-buffers/docs/tutorials, 2018年11月6日.
5. Developer guide[EB]. Developer Guide|Protocol Buffer: https://developers.google.com/protocol-buffers/docs/overview, 2018年8月23号.
6. Thrift 源码[EB].apache/thrift: https://github.com/apache/thrift.
7. Proxygen 源码[EB]. Facebook/Proxygen: https://github.com/facebook/proxygen.
8. 游双. Linux高性能服务器编程[M].北京：机械工业出版社，2013.06：123-145.
9. W.Richard Stevens / Stephen A.Rago. Advanced Programming in the UNIX Environment[M].北京：人民邮电出版社， 2006年.
10. 侯捷.STL源码剖析[M].武汉：华中科技大学出版社，2002年6月.
11. Vue.js Guide[EB]. Introduction----Vue: https://cn.vuejs.org/v2/guide/.
12. BootStrap Document[EB]. Introduction-Bootstrap: https://getbootstrap.com/docs/4.3/getting-started/introduction/\#reboot.
13. Thrift C++ Tutorial[EB]. Apache Thrift: https://thrift.apache.org/tutorial/cpp, 2017年.
14. [美] Stanley B. Lippman / [美] Josée Lajoie / [美] Barbara E. Moo. C++ Primer, 5th Edition[M].北京：电子工业出版社， 2013年9月1日.
15. W. Richard Stevens. Unix Network Programming[M].北京：清华大学出版社，2006年1月.

# 致谢

写到这里，我的本科毕业论文结束了。这意味着，我的大学生活，也即将结束了。

四年大学，匆匆忙忙，虽谈不上丰富多彩，自觉也并非完全的乏味平庸。一路走来，好的坏的，都看过一些，开心与不开心，也都经历过一些。不足以谈及丰富了多少阅历，只是对自己的认识，确因这四年光阴深刻了一些。似乎也确没什么比清晰深刻的认识自己更重要的了。在这篇论文的最后，我要由衷的感谢一些人。

首先应当感谢我的导师，邹承明老师，没有他的指导与帮助，不会有这篇论文，更不会有这些文字。

感谢大学四年我曾经熟悉和现在熟悉的同学们，是你们使我已经过去的四年青春岁月中，点缀着一些美丽的记忆点。愿时光长流，它们不会被岁月偷去。

感谢大学四年我不曾熟悉的同学们。感谢你们没有用自己无处安放的时间，占据我生命的一部分，让我可以慢慢成长为目前的自己。

感谢我的父亲和母亲。为了孩子们，他们奉献了自己年轻的身体与灵魂。我所取得的每一点点称得上成就的东西，都应归功于他们的支持与奉献。

感谢李志、尧十三、张浅潜等同学，是你们使我平淡无聊的生活中出现一点点浪漫主义的颜色，在写本篇论文期间，给了我很大的安慰。

最后，感谢我的女朋友，在毕业前离开了我，让我可以专心完成本篇论文，顺利毕业。不过是抛弃一些幻想而无所谓什么，我感觉好多了。

道阻且长，长路当歌。感谢相遇的每个人，愿你们一路有伴。