天网服务端架构学习文档

目 录

[1 概述 1](#_Toc18826)

[1.1 项目简介 1](#_Toc4632)

[1.2 编写目的 1](#_Toc13695)

[1.3 文档结构 1](#_Toc23237)

[1.4 工程代码位置 1](#_Toc26740)

[2 天网服务端架构简述 2](#_Toc786)

[2.1 架构层次 2](#_Toc29604)

[2.2 架构功能定位 3](#_Toc32347)

[2.2.1 主接口层 3](#_Toc12254)

[2.2.2 RE层 3](#_Toc12682)

[2.2.3 AE层 4](#_Toc2136)

[2.2.4 BE层 4](#_Toc21804)

[2.3 部署方式 4](#_Toc3309)

[3 主接口架构分析 6](#_Toc5411)

[4 RE层架构分析 7](#_Toc3970)

[4.1 业务处理逻辑 7](#_Toc17164)

[4.1.1 服务初始化 7](#_Toc28013)

[4.1.2 请求处理流程 7](#_Toc22659)

[4.1.3 与其他层次的交互 9](#_Toc27904)

[4.2 重点模块介绍 9](#_Toc12535)

[4.2.1 model模块 9](#_Toc6330)

[4.2.2 common模块 9](#_Toc24234)

[4.2.3 engines 9](#_Toc3236)

[4.3 问题讨论 10](#_Toc23814)

[4.3.1 预处理与后处理 10](#_Toc4064)

[4.3.2 操作引擎 10](#_Toc12930)

[4.3.3 层次整合 10](#_Toc25468)

[4.4 重构后的RE 11](#_Toc26268)

[5 AE层架构分析 12](#_Toc14593)

[5.1 业务处理逻辑 12](#_Toc5303)

[5.2 重点模块介绍 12](#_Toc5719)

[5.2.1 规则引擎 12](#_Toc27360)

[5.2.2 分布式画像库 12](#_Toc23031)

[5.2.3 后处理过程 13](#_Toc7112)

[5.3 问题讨论 13](#_Toc27012)

[6 BE层架构分析 14](#_Toc17777)

[6.1 业务处理逻辑 14](#_Toc9787)

[6.2 问题讨论 14](#_Toc6751)

[7 小结 15](#_Toc24029)

# 概述

## 项目简介

天网产品线是数美的核心产品之一，主要提供账号安全和设备安全相关的服务，从架构层次上来说从上到下主要分为mainif、RE、AE和BE等层次，而目前本人接触到的re-fraud-account和re-postevent工程属于其中的RE层次。

各个层次的架构以及具体功能将在正文中进行介绍。

## 编写目的

由于目前项目文档较为匮乏，加之工程代码注释较少，不熟悉业务的新同事可能在阅读代码时较为迷茫和吃力，因此拟撰写本文档，作为对天网服务端的整体功能描述和业务流程讲解，供后续的开发和维护人员参考。

因本人也是初次接触项目代码，且目前也只阅读了RE层的几个工程，阐述过程中难免有疏漏之处，如有不当描述，还请大家及时指正，不胜感谢。

## 文档结构

本文的主要结构如下。

首先会对天网产品的整体架构进行大致的描述，并对各个层次的主要功能进行说明；

其次会详细分析各个层次的项目，结合代码分析具体的业务逻辑和流程，并对一些重点模块进行介绍，同时讨论一些值得商榷的问题，目前仅仅加入了RE层的详细分析；

最后会对本文档进行总结。

若之后接触其他层次的项目，也拟将其代码走读总结加入本文档。

## 工程代码位置

re-fraud-account项目： [git@code.aliyun.com:nextdata/re-fraud-account.git](mailto:git@code.aliyun.com:nextdata/re-fraud-account.git)

re-postevent项目： [git@code.aliyun.com:nextdata/re-postevent.git](mailto:git@code.aliyun.com:nextdata/re-fraud-account.git)

# 天网服务端架构简述

## 架构层次

天网产品以web服务(http / https)的形式向用户提供服务，其大致服务端架构主要分为mainif、RE、AE、BE等4个层次，如图2.1所示。

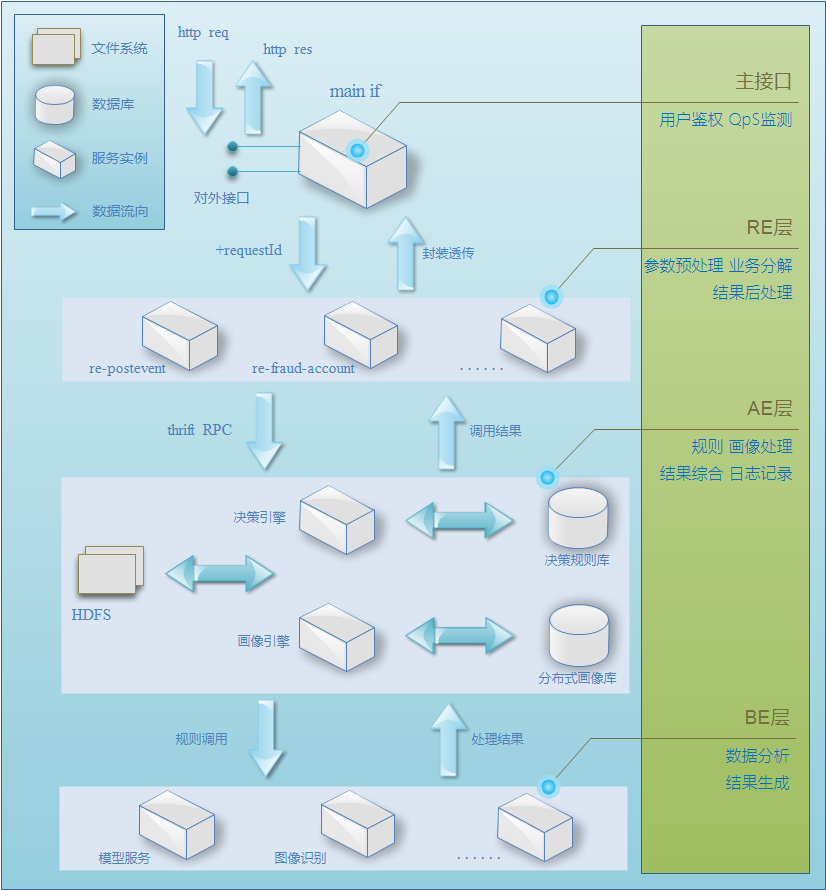


图2.1 天网服务端架构层次

用户以http请求的形式调用服务端对外提供的mainif(主接口)，一个请求的大致处理流程如下。

1. mainif层接收命令后，进行公共的用户鉴权、QpS监测等操作，对请求进行过滤，根据AccessKey等参数确定请求来源的organization，筛选除符合条件的请求，并生成requestId写入http请求的header中，随后将交给RE层处理。requestId为一条请求的唯一性标识；
2. RE层接受mainif下发的请求，并利用httpHandler将不同的URL绑定至具体的操作函数，以分解不同业务类型的处理逻辑，若有必要则进行参数的预处理(如图片的下载)，并构造相应的请求参数。经过上述预处理之后，使用thrift RPC将请求下发至AE；参数的ServiceId标识需要处理的业务类型。
3. AE层进行主要的规则处理，根据不同的业务类型使用规则引擎定义的规则，按照一定的逻辑顺序调用相应的BE；
4. BE是最基础的数据处理单元，以一定的规则分析上层来的数据，并生成一个结果(如score、通过标识等)，并返回给AE层；
5. AE层再通过一定的规则将BE返回的结果进行综合处理，作为此次RPC调用的调用结果回传给RE层。与此同时记录结果日志，并由另外的服务同步至HDFS等文件系统中；
6. RE层将结果转换封装，若有必要还需进行结果的后处理(如视频鉴定结果的综合)，之后将结果透传给mainif，最终回传给用户。

## 架构功能定位

### 主接口层

主接口层负责与用户的直接交互，处于整个服务端架构的最上层。其中集成了所有业务请求均需要进行的用户鉴权、QpS监测等操作。

### RE层

RE层是从主接口层中分离出来的层次，从性质上来说可定位为一个适配层，其通过http服务接收上层的请求，主要功能是参数的预处理、业务逻辑的分解和结果的后处理。

参数的预处理是指将用户下发的数据转换成下层服务(AE、BE)能够直接处理的数据。简而言之，就是解析http请求中的参数，然后封装成下层需要的结构下发或者存库(video)。

业务逻辑的分解是指根据用户请求的不同URL进行对应业务类型的参数校验、解析和请求的下发操作。

结果的后处理是指将AE层返回的结果进行综合，生成返回给用户的最终结果，然后封装进http响应结构中回传给用户。

RE层按照业务性质分为了多个集群进行部署，具体机制将在详细分析中进行介绍。

### AE层

AE层是逻辑处理的核心，其以lua脚本的形式将数据处理分为了5个过程：preprocess，postprocess，profilefinal，ruleengine，mysqlrecord，涉及数据预处理、读写画像、历史记录等操作。AE主服务根据ServiceId穿插调用这些过程脚本与BE，通过clips配置的策略生成结果模型，并通过结果模板和规则生成最终返回给RE的结果。

### BE层

BE层作为微服务，根据AE下发的参数进行功能性较强的操作，如写库读库，数据分析等，并返回操作的结果供上层进行综合。

BE的调用顺序完全由AE中针对不同的ServiceId配置的逻辑决定。

## 部署方式

各个层次的服务使用zookeeper进行集群部署，并根据请求大类分为了不同的集群提供服务，如re-fraud-account主要用于各种风险信息的识别，而re-postevent则主要用于事件信息的上报或数据的提取。

zookeeper的注册地址通常记录在配置文件中，在服务启动时读取，如图2.2所示。

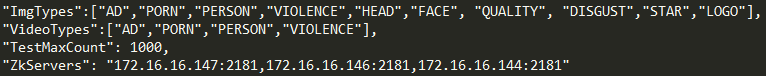


图2.2 配置文件中zookeeper的配置块

如图2.3所示为zookeeper服务集群的简要架构。

zookeeper主要通过选举机制保证一致性，通过监视和心跳机制保证可用性。具体内容可参考<https://www.w3cschool.cn/zookeeper/zookeeper_overview.html>。

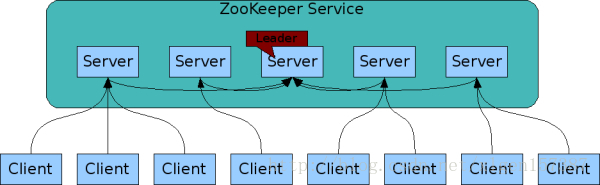


图2.3 zookeeper服务集群的简要架构

# 主接口架构分析

由于主接口的结构相对简单，因此不进行过多说明，若有遗漏，之后补充。

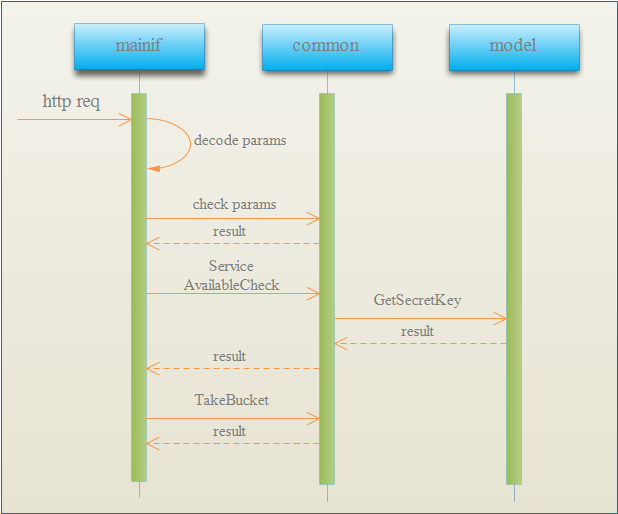


图3.1 主接口的调用时序图

如图3.1所示为主接口的调用时序图，其大致逻辑如下。

1. 解析并校验参数，主要面向AccessKey；
2. 进行用户鉴权操作，数据来源为数据库，通过AccessKey和Organization信息鉴定该用户是否有权限下发请求；
3. 进行QpS检测操作，数据来源为配置文件，通过计算该组织的QpS，并与配置文件中的数值进行比对，鉴定该用户的请求频率是否过高；
4. 上述过程中若出现异常，则立即返回相应错误，若均校验无误，则继续下发至RE层进行后续的逻辑处理。

# RE层架构分析

## 业务处理逻辑

### 服务初始化

RE层服务初始化主要加载的模块如图4.1所示。



图4.1 RE层服务的初始化流程

### 请求处理流程

RE层的请求处理流程主要分为两种模式：与AE层交互和与数据库交互。其中前者适用于普遍的实时下发+实时返回的场景，如图片处理、文本处理、账户登录等，而后者则适用于异步返回或者直接拉取数据的场景，如视频处理和风险账号信息获取等。

如图4.2所示为与AE层交互场景的请求处理流程。

1. 解析入参，并调用common模块中封装的接口进行入参校验；
2. 构造当前类型业务对应的处理引擎并构造参数调用执行方法；
3. 引擎向AE层下发请求，接收返回的结果；
4. 将结果记录至Redis数据库中，并返回http响应；

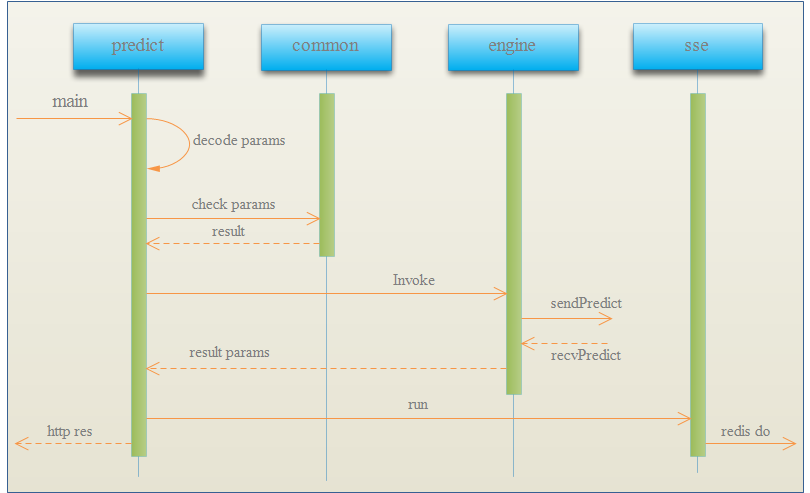


图4.2 与AE层交互的请求处理流程

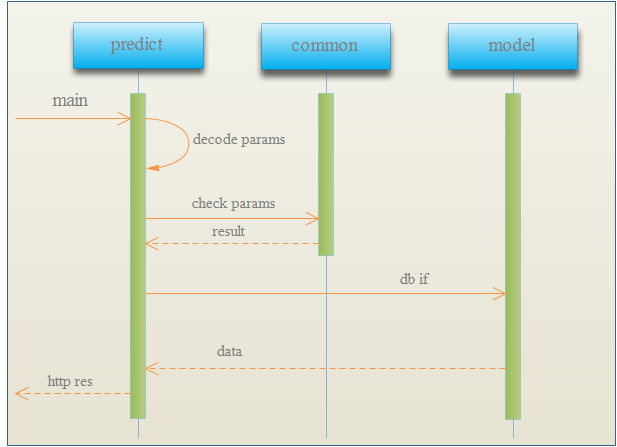


图4.3 与数据库交互的请求处理流程

如图4.3所示为与数据库交互的请求的处理流程。

1. 解析入参，并调用common模块中封装的接口进行入参校验；
2. 调用model模块中封装的数据库接口，写入或拉取数据；
3. 返回http响应。

### 与其他层次的交互

结合4.1.2中的说明，RE层与上层的交互过程，主要通过http请求接受命令，通过http响应回复数据；与下层的交互过程，通过thrift RPC向AE层下发命令并接受数据，通过数据库接口与数据库进行数据查询或写入。

## 重点模块介绍

### model模块

model模块中集成了几乎所有的数据库接口，或者说，所有数据库接口都应封装在model模块中。

当需要封装一个数据库接口时，一般先定义一个入参结构体，并在其上定义相应方法，使用sql语句完成需要的操作。若为查询接口，还需要定义一个出参结构体，与前述结构体为组合关系。

### common模块

common模块中集成了各种可能复用的组件，如入参结构和入参校验接口、返回结构定义、解码编码接口等。

### engines

按照目前的代码结构，一般是一个或者多个业务类型对应一个操作引擎进行处理，操作引擎使用解析完毕后的参数通过thrift RPC将请求下发至AE层。

thrift是一个使用较为广泛的RPC框架，传输方式有阻塞式和非阻塞式，传输协议有文本和二进制两大类。

目前RE层实现的引擎主要使用TSocket阻塞式连接，使用thrift默认的TBinaryProtocol二进制传输协议。

特别的，下层服务的配置信息均存储在配置文件中，若请求失败，引擎中的逻辑允许请求在一定重试次数和一定重试时间内重试其他host，从而在一定程度上保证了可用性(对外)和容错性(对内)。

## 问题讨论

### 预处理与后处理

之前了解到RE层应具有参数预处理和结果后处理的功能。

预处理一个典型的例子是将根据图片的URL下载到真正的图片，但这样的操作明显是不适合由RE层负责的，实际上目前代码中也确实没有这样实现。

后处理一个典型的例子是将视频取关键帧之后分别调用下层服务进行检测，最后综合所有的结果，但没有上述预处理的过程，这种后处理也显然无从谈起。

图片或视频等一般都承载了较大的数据量，若在服务的最上层便执行下载操作并作为参数下发至下层服务，无疑会大大增加进程间通信的开销，造成资源的无谓浪费，因此在需要处理数据的地方获取数据，在需要处理结果的地方处理结果是最合适的。

如此说来，似乎RE层所谓的参数预处理和后处理操作便只剩下了http请求参数的解析、封装和和RPC返回结果的透传，其层次的功能性并没有得到很好的体现。

### 操作引擎

结合4.4.1中的分析，由于json结构的灵活性，解析后得到的参数是完全有可能全部复用一个结构体的，仅针对需要特化的字段使用map[string]interface{}结构进行填充即可。

因此目前来看，不同的业务逻辑共用一个操作引擎应该是可以实现的，没有必要像目前的代码中这样每种或者每几中请求便各自实现一个引擎，而只需要区分各自的serviceId即可。

### 层次整合

之前了解到，RE层和主接口是一个层次分离的结果，但综上所述，目前的层次感还不甚明显。

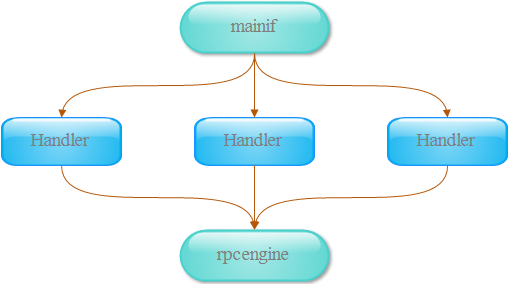


图4.4 RE层次整合

若将主接口的功能(用户鉴权、QpS监测)作为RE层的一个公共模块，所有RPC调用引擎也统一封装为一个公共模块，将主接口与RE层重新合并为一个层次，如图4.4所示，个人认为也并无不妥。

如此一来，整个RE层次的入口和出口均是公共模块提供的统一接口，而具体的业务逻辑分离均封装在中间的各个Handler中，整个层次的功能性应该会更明显一些。

## 重构后的RE

根据4.3中讨论的结果进行取舍和优化后，对re-postevent中的架构逻辑进行整合，并最终达到了如下效果：

1. 所有业务对外提供一个统一的接口，即/v2/event，不同的业务共用一个处理函数和一个操作引擎，根据下发参数中不同的eventId进行特化的入参校验和参数构造；
2. 需要新增业务时，仅需要新增eventId即可，不需要前端和AE层再添加新ServiceId的处理，也不需要给客户提供新的接口，扩展性较强；
3. 同一事件同时支持单个和批量的操作。

# AE层架构分析

## 业务处理逻辑

还未了解到，待之后补充。

## 重点模块介绍

### 规则引擎

AE层的规则使用clips规则引擎进行配置。

clips(C Language Integrated Production System)，是一种基于正向推理的控制策略语言，其使用事先声明的模板(template)定义一系列事实(fact)，在其基础上通过设定的规则(rule)推演新的事实，其中使用=>符号表达 if-then的逻辑句式。

规则引擎的引入主要是为了解决可能出现的频繁业务变化问题。当业务的复杂度越来越高时，单纯的硬编码或者仅仅设计模式上的优化往往很难适应灵活多变的需求，而即便使用配置文件代替硬编码，维护工作的时间成本也往往居高不下。

规则引擎的使用便可以很好地处理这种问题。开发人员仅需要通过统一的规范制定符合需求的规则范式，然后交给规则引擎进行自动化的处理分析运行。当需求改变或增加时，仅需要对规则进行修改，避免了开发人员无止境的编码工作。

目前clips主要用于策略的配置和结果的综合。AE中各个过程调用产生的数据通过配置的策略生成初步的模型，并由结果模板和规则综合为最终返回给RE的结果。

线上环境和测试环境规则的配置页面如下：

[https://admin.fengkongcloud.com/#/rule-engine/rule/config](https://admin.fengkongcloud.com/" \l "/rule-engine/rule/config)

[https://testadmin.fengkongcloud.com/#/rule-engine/rule/config](https://admin.fengkongcloud.com/" \l "/rule-engine/rule/config)

### 分布式画像库

AE层的画像库是一个分布式数据库，虽然主要通过be-profile进行读写操作但架构上仍暂时归为AE层，其主要使用一致性哈希支持快速高效的水平扩展。

数据节点的水平扩展应满足均衡性、单调性、分散性等要求。其中均衡性是指数据应尽可能平均分布到各个节点中，单调性是指在数据节点增加或减少时能最大限度地避免数据迁移，分散性是指尽量保证数据哈希映射的唯一性。

普通的哈希函数往往无法满足单调性的需求，而一致性哈希便被设计用于解决此类问题。其将数据和数据节点哈希映射到同一个哈希环上，并按照顺时针或逆时针原则将数据分配给相应的节点。如此一来，当节点增加或者减少时，需要进行迁移的便只有某段环上的数据，由此保证了整个系统的单调性。

但仅仅这样处理仍存在一个均衡性的隐患。当数据节点的数量相对较少时，无法保证节点在哈希环上的分布足够均衡，因此可能导致不同数据节点的负载相差很大。

为此，可通过引入虚拟节点，即将一个节点拆分为多个虚拟节点分布到哈希环中，如图5.1所示，利用数据量的增加使哈希函数的均衡性得以体现。

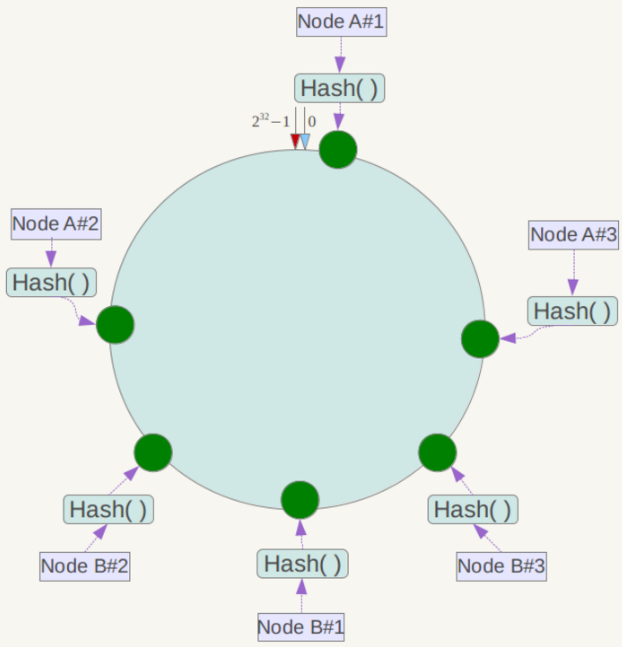


图5.1 一致性哈希

### 后处理过程

AE层通过穿插调用lua脚本的方式对各个过程生成的结果进行后处理操作，主要完成一些特殊场景的过滤和整合等功能。

## 问题讨论

# BE层架构分析

## 业务处理逻辑

还未了解到，待之后补充。

## 问题讨论

还未了解到，待之后补充。

# 小结