

[简介](#)

[架构图](#)

[工作时序图](#)

[部署架构](#)

[源码打包部署](#)

[serving-server的部署](#)

[serving-server.properties的配置详解](#)

[serving-proxy的部署](#)

[serving-proxy路由表route_table.json配置](#)

[serving-proxy路由表案例](#)

[serving-admin的部署](#)

[如何加载模型](#)

[FATE-Flow的配置（1.4.x 版本）](#)

[1.未启用注册中心](#)

[2.启用注册中心](#)

[3.模型发布](#)

[4.模型绑定](#)

[常见问题](#)

[推理接口](#)

[单笔接口](#)

[http请求](#)

[http请求地址](#)

[http请求类型](#)

[请求内容](#)

[请求示例](#)

[响应内容](#)

[java版SDK](#)

[操作步骤](#)

[示例](#)

[批量接口](#)

[http请求](#)

[请求地址](#)

[请求类型](#)

[请求内容](#)

[请求示例](#)

[响应内容](#)

[java版SDK](#)

[操作步骤](#)

[示例](#)

[错误码表](#)

[HOST如何获取特征](#)

[自定义Adapter开发](#)

[fate-serving-extension](#)

[预设Adapter](#)

[MockAdapter](#)

[MockBatchAdapter](#)

[BatchTestFileAdapter](#)

TestFileAdapter

日志

serving-server的访问日志格式

serving-proxy的访问日志格式

可选打印内容

fate-serving-client命令行工具

fate-serving 的服务治理

zookeeper中的数据结构

订阅与注册的可靠性保证

客户端订阅服务信息的缓存与持久化

客户端的负载均衡

服务端的优雅停机

serving-admin介绍

介绍

前端部分

后端部分

Swagger 支持

功能介绍

节点管理

模型管理

服务管理

统计/监控

流量控制

用户管理

配置详解

application.properties

代码导读

算法相关

基础类

HeteroSecureBoost 组件

HeteroSecureBoost

HeteroSecureBoostingTreeGuest and HeteroSecureBoostingTreeHost

HeteroLR 在线推理

HeteroLR

HeteroLRGuest and HeteroLRHost

特征工程组件介绍

纵向特征分箱 (Hetero Feature Binning)

文件结构

参数和方法说明

纵向特征选择 (Hetero Feature Selection)

文件结构

参数和方法说明

One-Hot组件

参数和方法说明

Scale组件

参数和方法说明

Imputer组件

参数和方法说明

Outlier组件

简介

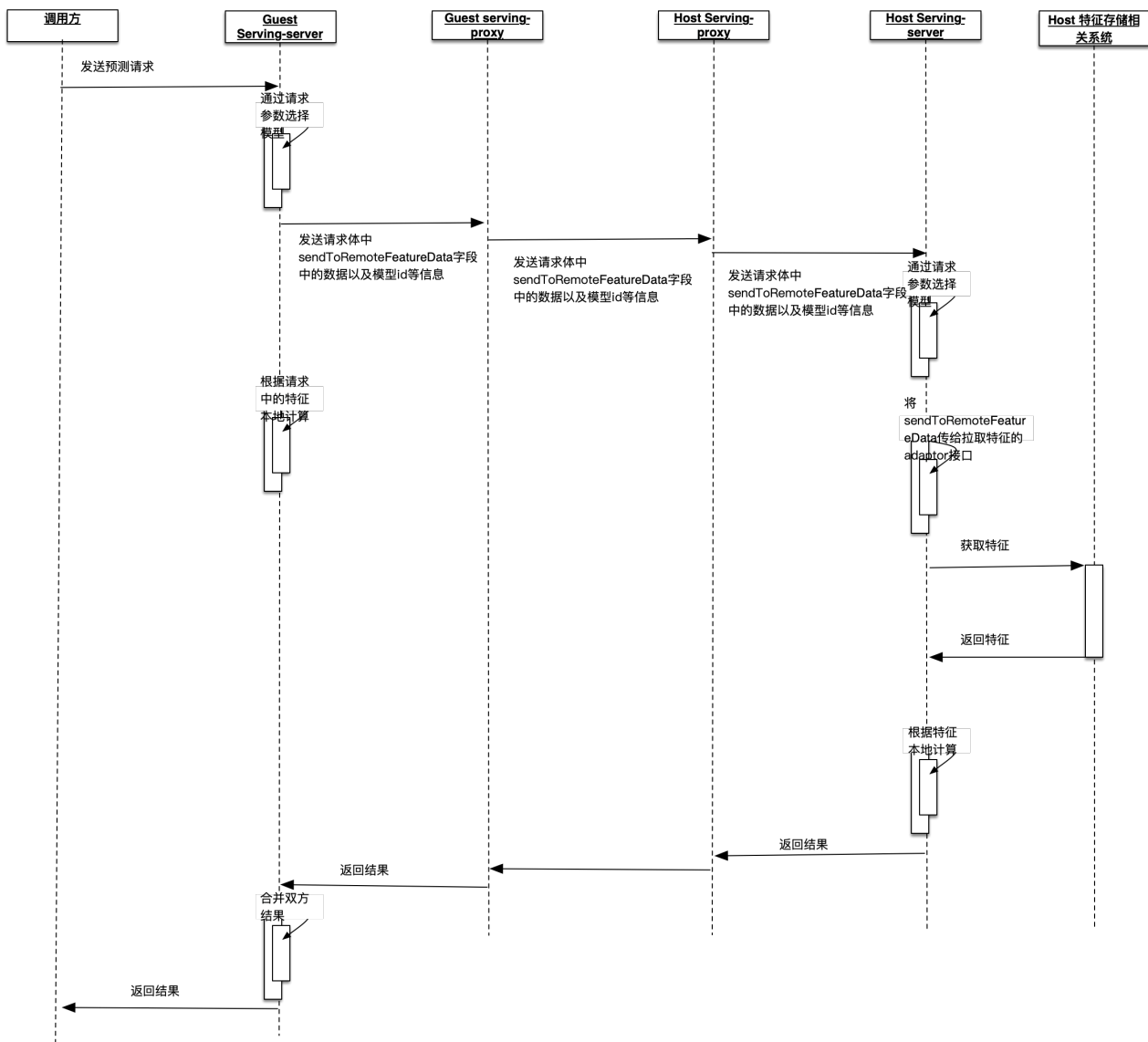
fate-serving是FATE的在线部分，在使用FATE进行联邦建模完成之后，可以使用fate-serving进行在线联合预测。**目前的文档只适用于2.0以上版本** FATE-SERVING 2.0 支持的特性：

- 1.单笔预测，2.0.*版本guest方与host方将并行计算，从而降低了耗时。
- 2.批量预测，2.0.*版本开始引入的新的特性，一次请求可以批量提交一批需要预测的数据，极大地提高了吞吐量。
- 3.并行计算，在1.3.*版本中guest方的预测与host方的预测是串行，从2.0版本开始guest方与host方将采用并行预测的方式，各方预测时可以根据特征数量拆分成子任务再并行计算。
- 4.可视化的集群操作界面，引入新的组件serving-admin，它将提供集群的可视化操作界面，包括模型的管理、流量的监控、配置的查看、服务的管理等操作。
- 5.新的模型持久化/恢复方式，在serving-server重启时1.3.*版本采用了回放推送模型的请求的方式来实例重启时恢复模型，2.0.*版本使用了直接恢复内存数据的方式恢复模型。
- 6.java版的SDK，使用该SDK可以使用FATE-SERVING带有的服务治理相关的功能，如服务自动发现、路由等功能。
- 7.新的扩展模块，将需要用户自定义开发的部分（如：host方的特征获取adaptor接口开发）放到该模块，从而与核心源码分离。
- 8.支持多种缓存方式，FATE-SERVING在1.3.*版本强依赖redis，从2.0.*版本开始，不再强依赖redis。可以选择不使用缓存、使用本地内存缓存、使用redis。
- 9.更改内部预测流程，重构了核心代码，去除调了前后处理等组件，使用了统一的异常处理。算法组件不再跟rpc接口紧耦合。

架构图

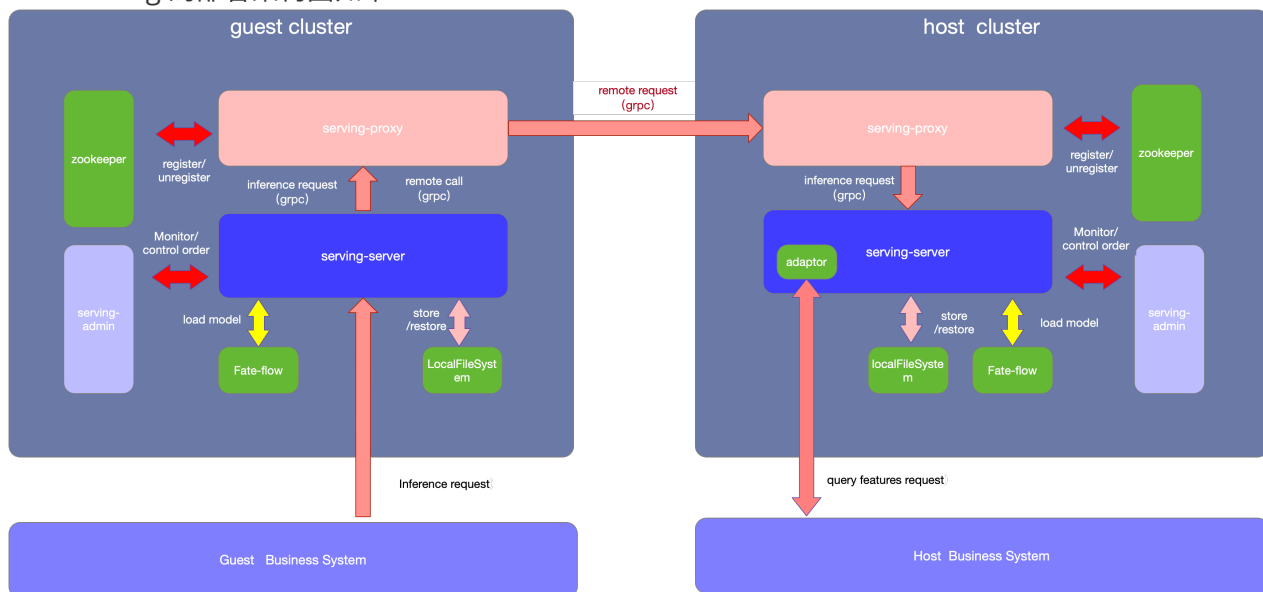


工作时序图



部署架构

fate-serving 的部署架构图如下



如上图所示，整个集群需要有几个组件

- serving-server
- serving-proxy
- serving-admin(非必选)
- [zookeeper](#) 3.5.5 以上版本

源码打包部署

fate-serving 使用mvn作为jar包管理以及打包的工具，在打包前需要检查以下条件是否满足：

- 安装jdk8+
- 有良好的网络
- 已安装maven

serving-server的部署

serving-server用于实时处理在线预测请求,serving-server需要从fate-flow加载模型成功之后才能对外提供服务。在FATE中建好模型之后，通过fate-flow的推送模型脚本可以将模型推送至serving-server。推送成功之后，serving-server会将该模型相关的预测接口注册进zookeeper，外部系统可以通过服务发现获取接口地址并调用。同时本地文件持久化该模型,以便在serving-server实例在集群中某些组件不可用的情况下，仍然能够从本地文件中恢复模型。

部署serving-server时，可以根据实际业务请求量来动态扩缩容。

部署步骤

1. 从github上克隆代码git clone <https://github.com/FederatedAI/FATE-Serving.git>
2. 执行 cd FATE-Serving，进入源码的根目录。
3. 执行 mvn clean package -Dmaven.test.skip=true命令
4. 拷贝 serving-server/target/fate-serving-server-{version}-release.zip 到想要部署的路径下，并解压。（version为当前版本）
5. 根据需要修改或者不修改部署目录下conf/serving-server.properties文件

默认使用端口为8000，以及默认使用zookeeper，一般情况下需要检查并修改zookeeper地址

6. sh service.sh restart 启动应用（windows 脚本暂时不 支持，如有需要可自行编写）
7. 检查日志与端口看启动是否正常

serving-server.properties的配置详解

源码中的配置文件没有罗列出所有配置，只保留了必需的配置，其他配置都采用了默认值。如果需要可以根据以下表格来在配置文件中新增条目。

配置项	配置项含义	默认值
port	服务监听端口	8000
remoteModelInferenceResultCacheSwitch	预测结果的缓存开关，false代表不使用缓存，true代表使用缓存，该配置跟cache.type 配合使用	false
cache.type	缓存类型，可选local/redis,其中local为进程中维持的LRU内存，不建议在生产上使用local	local

local.cache.expire	内置缓存过期时间，单位：秒，该配置在cache.type=local时生效	30
local.cache.interval	内置缓存过期处理间隔，单位：秒，该配置在cache.type=local时生效	3
local.cache.maxsize	内置缓存最大存储数量，该配置在cache.type=local时生效	10000
redis.ip	redis ip地址，该配置在cache.type=redis时生效	127.0.0.1
redis.port	redis端口，该配置在cache.type=redis时生效	3306
redis.cluster.nodes	redis集群节点，配置则开启集群模式，该配置在cache.type=redis时生效	空
redis.password	redis密码，该配置在cache.type=redis时生效	空
redis.expire	redis过期时间，该配置在cache.type=redis时生效	3000
redis.timeout	redis链接超时时间，该配置在cache.type=redis时生效	2000
redis.maxIdle	redis连接池最大空闲链接，该配置在cache.type=redis时生效	2
redis.maxTotal	redis连接池最大数量，该配置在cache.type=redis时生效	20
serving.core.pool.size	grpc服务线程池核心线程数	cpu核心数
serving.max.pool.size	grpc服务线程池最大线程数	cpu核心数 * 2
serving.pool.alive.time	grpc服务线程池超时时间	1000
serving.pool.queue.size	grpc服务线程池队列数量	100
single.inference.rpc.timeout	单次预测超时时间	3000
batch.inference.max	单次批量预测数量	300
batch.inference.rpc.timeout	批量预测超时时间	3000
batch.split.size	批量任务拆分数量，在批量预测时会根据该参数大小将批量任务拆分成多个子任务并行计算，比如如果该配置为100，那300条预测的批量任务会拆分成3个100条子任务并行计算	100
lr.use.parallel	lr模型是否启用并行计算	false
lr.split.size	LR多任务拆分数量，该配置在lr.use.parallel=true时生效	500
feature.batch.adaptor	批量特征处理器，Host方需要配置，用于批量获取Host方特征信息，用户可根据业务情况，实现AbstractBatchFeatureDataAdapter接口	com.webank.ai.fate.serving.adaptor.dataaccess.MockBatchAdapter
feature.single.adaptor	单次特征处理器，Host方需要配置，用于获取Host方特征信息，用户可根据业务情况，实现AbstractSingleFeatureDataAdapter接口	com.webank.ai.fate.serving.adaptor.dataaccess.MockAdapter
model.cache.path	模型缓存地址，对于内存中存在的模型，serving-server会持久化到本地以便在重启时恢复	服务部署目录
model.transfer.url	fateflow模型拉取接口地址，优先使用注册中心中的fateflow地址，若注册中心中没有找到fateflow地址，则会使用该配置地址	http://127.0.0.1:9380/v1/model/transfer
proxy	离线路由proxy地址，建议通过zookeeper来获取地址，不建议直接	127.0.0.1:8000

	配置	
zk.url	zookeeper集群地址	localhost:2181,localhost:2182,localhost:2183
useRegister	使用注册中心，开启后会将serving-server中的接口注册至zookeeper	true
useZkRouter	使用zk路由，开启后rpc调用时会使用注册中心中的地址进行路由	true
acl.enable	是否使用zookeeper acl鉴权	false
acl.username	acl 用户名	默认空
acl.password	acl 密码	默认空

serving-proxy的部署

serving-proxy 是serving-server的代理，对外提供了grpc接口以及http的接口，主要用于联邦预测请求的路由转发以及鉴权。在离线的联邦建模时，每一个参与方都会分配一个唯一的partId。serving-proxy 维护了一个各参与方partId的路由表，并通过路由表中的信息来转发请求。

serving-proxy的路由是根据接口中的

1. 从github上克隆代码git clone <https://github.com/FederatedAI/FATE-Serving.git> (若已执行过，则不需要再次执行)
2. 执行 cd FATE-Serving ，进入源码的根目录。
3. 执行 mvn clean package -Dmaven.test.skip=true命令 (若已执行过，则不需要再次执行)
4. 拷贝 fate-serving-proxy/target/fate-serving-proxy-{version}-release.zip 到想要部署的路径下，并解压。（version为当前版本）
5. 修改部署目录下 config/application.properties文件，具体的配置项解释见
6. router_table.json文件，具体的配置项解释见
7. sh service.sh restart 启动应用（windows 脚本暂时不 支持，如有需要可自行编写）
8. 检查日志与端口看启动是否正常

serving-proxy路由表route_table.json配置

下面的json代码大致说明了router_table.json的填写规则，之后我们会根据具体案例来配置一次。

```
{
  "route_table": {
    "default": {
      "default": [
        // 此处用于配置serving-proxy默认转发地址，切记不能配置成serving-proxy自己的ip
        // 端口，会形成回环
        {
          "ip": "127.0.0.1",
```



```

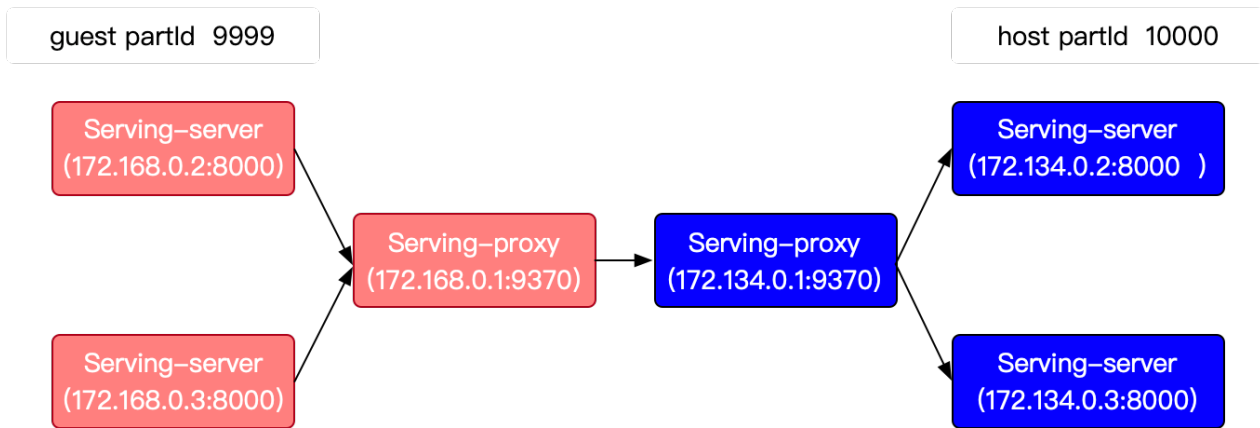
        "port": 9999
    }
}
// guest方使用上面的default配置就能满足大部分需求。
,
//serving-proxy 在收到grpc unaryCall接口的请求后，会根据请求中的目的partyId尝试
匹配。比如请求中目的partId为10000，则会在路由表中查找是否存在10000的配置
//此处的10000表示目的partId 为10000时的路由，匹配到10000之后，再根据请求中的角色信息
role ，比如请求中role 为serving则会继续匹配下面是否有serving的配置
"10000": {
    // 在未找到对应role的路由地址时，会使用default的配置
    "default": [
        {
            "ip": "127.0.0.1",
            "port": 8889
        }
    ],

    "serving": [
        // 当已经匹配到role为serving，则代表请求为发给serving-server的请求，这时检查是
        否启用了zk为注册中心，如果已启用zk则优先从zk中获取目标地址，未找到时使用以下地址
        // 此处配置已端对应serving服务地址列表，ip和port对应serving-server所启动的grpc
        服务地址
        {
            "ip": "127.0.0.1",
            "port": 8080
        }
    ]
}
},
// 此处配置当前路由表规则开启/关闭
"permission": {
    "default_allow": true
}
}

```

serving-proxy路由表案例

serving-proxy路由的配置由以下例子来说明：由下图所示：guest 的serving-proxy 的ip为172.168.0.1，请求将由它转发至host的serving-proxy，ip为172.134.0.1



guest 的配置:

```
{
  "route_table": {
    "default": {
      "default": [{
        "ip": "172.134.0.1",
        "port": 9370
      }]
    },
    "permission": {
      "default_allow": true
    }
  }
}
```

由于guest的请求只会向外发送，所以只需要配置出口ip端口就好，如以上代码所示只需要配置default转发规则，则会将所有请求转发至出口ip

host 的配置:

```
{
  "route_table": {
    "default": {
      "default": [{
        "ip": "172.134.0.2",
        "port": 8000
      }]
    },
    "10000": {
      "serving": [{
        "ip": "172.134.0.2",
        "port": 8000
      }],
      {
        "ip": "172.134.0.3",
        "port": 8000
      }
    }
  }
}
```

```
    }
  ]
}
},
"permission": {
  "default_allow": true
}
}
```

serving-admin的部署

serving-admin提供了集群的可视化操作界面，可以展示集群中各实例的配置、状态、模型、流量等信息，并可以执行模型的卸载、服务接口的权重调整等操作

建议安装serving-admin，通过使用serving-admin可以更方便地查看并操作模型等信息，能更方便地监控集群。

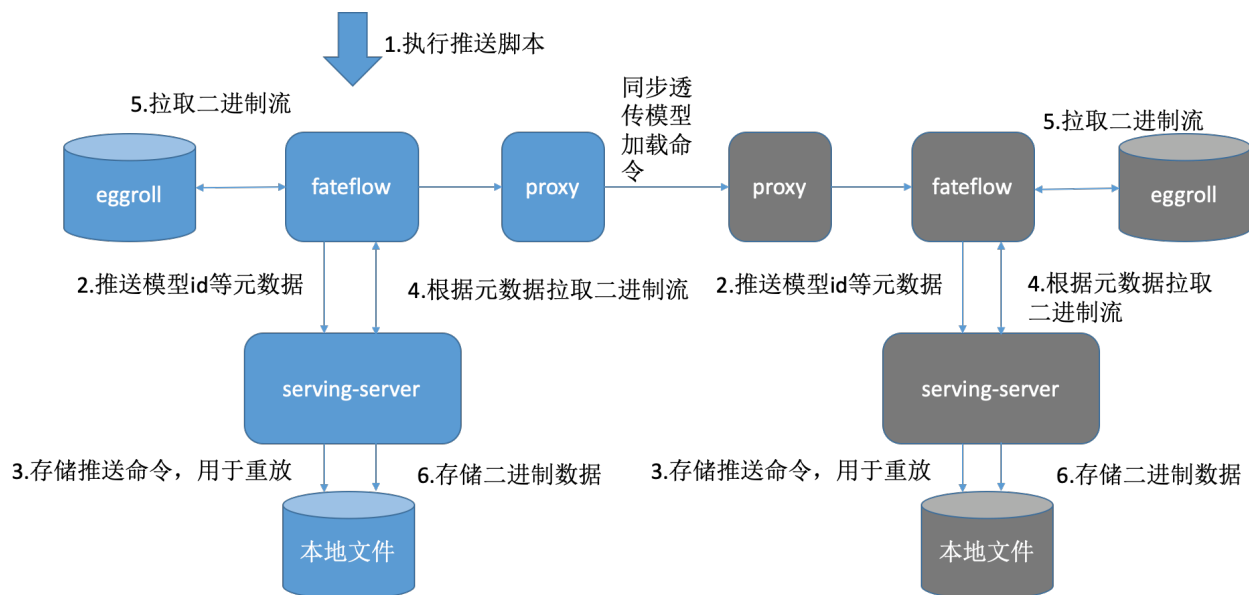
1. 从github上克隆代码git clone <https://github.com/FederatedAI/FATE-Serving.git> (若已执行过，则不需要再次执行)
2. 执行 cd FATE-Serving，进入源码的根目录。
3. 执行 mvn clean package -Dmaven.test.skip=true命令 (若已执行过，则不需要再次执行)
4. 拷贝 fate-serving-admin/target/fate-serving-admin-{version}-release.zip 到想要部署的路径下，并解压。（version为当前版本）
5. 修改部署目录下 application.properties文件，具体的配置项解释见
6. sh service.sh restart 启动应用（windows脚本暂时不支持，如有需要可自行编写）
7. 通过浏览器访问admin页面，默认端口8350

如何加载模型

前面介绍了如何安装各个组件，在各组件都成功安装后，接下来需要将模型推送至serving-server。推送模型的一般流程是

1. 通过FATE建模
2. 分别部署guest方 Fate-serving 与host方Fate-serving
3. 分别配置好guest方FATE-FLOW与guest方Fate-serving、host方FATE-FLOW 与host方Fate-serving，请参考[FATE-Flow配置](#)
4. 推送模型
5. 将模型绑定serviceId

如果是为了验证以及学习用，可以简化部署。代码中提供了简单的LR模型，用于进行简单测试，将 [example/model_cache_example.zip](#) 解压至guest与host双方serving-server实例部署目录下的 `.fate` 目录下，重启即可自动加载模型并绑定到 `lr-test` (Service ID)



FATE-Flow的配置（1.4.x 版本）

在执行模型的推送和绑定操作之前，需要先配置FATE-Flow,并重启FATE-FLOW：

注意多方都需要修改成各自FATE-Serving的实际部署地址

1.未启用注册中心

当FATE-Serving集群未启用注册中心时：

修改服务配置

- 修改arch/conf/server_conf.json，填入FATE-Serving集群实际部署serving-server服务的ip:port，如：

```
"servings": [
  "192.168.1.1:8000",
  "192.168.1.2:8000"
]
```

- 重启FATE-Flow服务

2.启用注册中心

当FATE-Serving集群启用注册中心时：

修改服务配置

- 修改部署目录下arch/conf/base_conf.yaml

```
use_registry: true
zookeeper:
  hosts:
    - '192.168.1.1:2181'
    - '192.168.1.1:2182'
```

其中 `ZOOKEEPER_HOSTS` 填入FATE-Serving集群实际部署Zookeeper的ip:port

- 若zookeeper开启了ACL，则需要添加以下配置，否则略过此步骤。修改部署目录下 `arch/conf/base_conf.yaml`

```
zookeeper:
  use_acl: true
  user: fate
  password: fate
```

其中 `use` 与 `password` 填入FATE-Serving群实际部署Zookeeper的用户名与密码

服务生效

- 重启FATE-Flow服务

当上述两种情况都配置了的情况下，会默认使用zookeeper模式。

3.模型发布

配置路径: `$pythonpath/fate_flow/examples/publish_load_model.json`

修改内容: 将实际任务配置 (`initiator`, `role`, `job_parameters`) 进行修改, 请确保 `model_id` 及 `model_version` 与离线训练的模型的相同。

配置格式:

```
{
  "initiator": {
    "party_id": "10000",
    "role": "guest"
  },
  "role": {
    "guest": ["10000"],
    "host": ["10000"],
    "arbiter": ["10000"]
  },
  "job_parameters": {
    "work_mode": 1,
    "model_id": "arbiter-10000#guest-9999#host-10000#model",
    "model_version": "202006122116502527621"
  }
}
```

4.模型绑定

配置路径: `$pythonpath/fate_flow/examples/bind_model_service.json`

修改内容：自定义 `service_id`，后续将利用该 `service_id` 将模型绑定到模型服务中。将实际任务配置（`initiator`, `role`, `job_parameters`）进行修改，请确保 `model_id` 及 `model_version` 与离线训练的模型的相同。

配置格式：

```
{
  "service_id": "",
  "initiator": {
    "party_id": "10000",
    "role": "guest"
  },
  "role": {
    "guest": ["10000"],
    "host": ["10000"],
    "arbiter": ["10000"]
  },
  "job_parameters": {
    "work_mode": 1,
    "model_id": "arbiter-10000#guest-10000#host-10000#model",
    "model_version": "202006122116502527621"
  },
  "servings": [
  ]
}
```

常见问题

常见问题	可能原因	解决方案
加载（load）/绑定（bind）模型提示“Please configure servings address”	用户未在server_conf中配置FATE-Serving的ip地址及端口	正确修改server_conf中的serving组件配置，重启fate_flow_server后进行重试。
加载（load）模型提示“failed”	FATE rollsite或serving组件未启动	检查\$pythonpath/logs/fate_flow/fate_flow_stat.log，确定是否有组件未启动，如果有，请正确启动组件后重试。
绑定（bind）模型提示“no service id”	bind任务配置文件中未指定service_id	修改bind任务配置文件，自定义指定service_id。

更多FATE-Flow问题请查看[FATE-Flow](#)

推理接口

FATE-Serving的在线推理功能，需要通过Guest和Host双方对模型数据样本进行联合预测，入口则是Guest方提供的inference接口。目前可以通过serving-proxy提供的http接口访问，由serving-proxy转发至serving-server，或者使用提供的sdk（目前提供了java版）直接访问serving-server 推理接口目前支持单笔以及批量接口

目前有以下几方式调用推理接口，具体采用哪种方式可以根据实际情况选择。

- 访问serving-proxy的http接口，由serving-proxy转发请求至serving-server。优点：接入简单，可在serving-proxy前面增加nginx作为反向代理。
- 使用源码中自带的SDK访问serving-server。优点：省去了中间serving-proxy作为转发节点，提高通信效率。并将使用fate-serving的服务注册以及发现等功能，直接调用serving-server的grpc接口
- 自行开发并直接调用serving-server提供的grpc接口。优点：目前sdk部分只提供了java版，若是其他未支持的语言，可以自行开发并调用相关接口。部署时可采用 nginx前置部署，用于反向代理grpc请求。

单笔接口

http请求

http请求地址

请求路径：http://{ip}:{port}/federation/v1/inference

`{ip}:{port}` 为Guest方serving-proxy的地址，依据实际部署架构而定

http请求类型

- POST
- content-application/json

请求内容

- 请求体（Request Body）

参数名	是否必填	类型	描述
head	是	json object	系统所需参数
body	否	json object	模型预测需要用到的数据，一般是包括ID，比如手机号、设备号等

请求示例

```
{
  "head": {
    "serviceId": "test-lr"
  },
}
```

```

"body": {
  "featureData": {
    "x0": 1.88669,
    "x1": -1.359293,
    "x2": 2.303601,
    "x3": 2.00137,
    "x4": 1.307686
  },
  "sendToRemoteFeatureData": {
    "device_id": "8",
    "phone_num": "122222222"
  }
}
}

```

`head` 中填入系统参数，`featureData` 中为模型所需特征数据，不会传递给Host方，只有 `sendToRemoteFeatureData` 中的才会传递给Host方，一般 `sendToRemoteFeatureData` 需要包含host方用于匹配样本的id，例如设备号或者手机号

响应内容

字段名	类型	描述
retcode	int	错误码，0 表示请求成功
retmsg	string	错误提示信息
data.score	double	预测得分
data.modelId	string	模型ID
data.modelVersion	string	模型版本
data.timestamp	long	模型发布时间戳
flag	int	保留字段

```

{
  "retcode": 0,
  "retmsg": "",
  "data": {
    "score": 0.5386207970765767,
    "modelId": "guest#9999#arbiter-10000#guest-9999#host-10000#model",
    "modelVersion": "202007281411525000000",
    "timestamp": 1595916929427
  },
  "flag": 0
}

```


目前提供了java版sdk，可以通过使用java版SDK来对接serving-server，sdk提供服务发现以及路由功能。需要使用在目标工程pom文件中加入相关依赖

操作步骤

```
1.cd fate-serving源码根目录, 执行 mvn clean install -pl
com.webank.ai.fate:fate-serving-core,com.webank.ai.fate:fate-serving-
sdk,com.webank.ai.fate:fate-serving-register,com.webank.ai.fate:fate-serving
2.在目标工程文件中加入依赖
<dependency>
    <groupId>com.webank.ai.fate</groupId>
    <artifactId>fate-serving-sdk</artifactId>
    <version>2.0.0</version>
</dependency>
```

示例

```
import java.io.IOException;
import java.io.OutputStream;
import java.net.Socket;
import java.util.Map;

import com.google.common.collect.Maps;
import com.webank.ai.fate.serving.core.bean.InferenceRequest;
import com.webank.ai.fate.serving.core.bean.ReturnResult;
import com.webank.ai.fate.serving.sdk.client.Client;

public class ClientTest {

    static Client client =
Client.getClient("localhost:2181,localhost:2182");

    static InferenceRequest buildInferenceRequest(){
        InferenceRequest inferenceRequest = new InferenceRequest();
        inferenceRequest.setServiceId("mytest");
        Map<String,Object> featureData = Maps.newHashMap();
        featureData.put("x1",0.1);
        inferenceRequest.setFeatureData(featureData);
        Map<String,Object> sendToRemote = Maps.newHashMap();
        sendToRemote.put("phone_num","1233333");
        inferenceRequest.setSendToRemoteFeatureData(sendToRemote);
        return inferenceRequest;
    }

    public static void main(String[] args) throws IOException,
InterruptedException {
        InferenceRequest inferenceRequest = buildInferenceRequest();
```

```
        try {
            ReturnResult returnResult =
client.singleInference(inferenceRequest);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

批量接口

FATE-Serving的在线批量推理功能，在单次推理基础上提升吞吐量，提供了支持批量推理接口，入口则是Guest方提供的batchInference接口。

http请求

请求地址

请求路径：http://{ip}:{port}/federation/v1/batchInference

`{ip}:{port}` 为Guest方serving-proxy的地址，依据实际部署架构而定

请求类型

- POST
- content-application/json

请求内容

- 请求体（Request Body）

参数名	是否必填	类型	描述
head	是	json object	系统所需参数
body	否	json object	模型预测需要用到的数据，一般是包括ID，比如手机号、设备号等

请求示例

```
{
  "head": {
    "serviceId": "test-lr"
  },
  "body": {
    "batchDataList": [
      {
```

```

        "index": 0,
        "featureData": {
            "x0": -0.161357,
            "x1": 0.822813,
            "x2": -0.031609,
            "x3": -0.248363,
            "x4": 1.662757,
            "x5": 1.81831
        },
        "sendToRemoteFeatureData": {
            "device_id": "8",
            "phone_num": 644
        }
    },
    {
        "index": 1,
        "featureData": {
            "x0": -0.161357,
            "x1": 0.822813,
            "x2": -0.031609,
            "x3": -0.248363,
            "x4": 1.662757,
            "x5": 1.81831
        },
        "sendToRemoteFeatureData": {
            "device_id": "8",
            "phone_num": 644
        }
    }
]
}
}

```

head 中填入系统参数，body 中 batchDataList 为参数集合，featureData 中为模型所需特征数据，不会传递给Host方，只有 sendToRemoteFeatureData 中的才会传递给Host方，一般 sendToRemoteFeatureData 需要包含host方用于匹配样本的id，例如设备号或者手机号

响应内容

字段名	类型	描述
retcode	int	错误码, 0 表示请求成功
retmsg	string	错误提示信息
flag	int	保留字段
data.modelId	string	模型ID
data.modelVersion	string	模型版本
data.timestamp	long	模型发布时间戳
batchDataList	object array	批量预测结果合集
singleInferenceResultMap	map	批量预测结果下标映射Map

```
{
  "retcode": 0,
  "retmsg": "",
  "data": {
    "modelId": "guest#9999#arbiter-10000#guest-9999#host-10000#model",
    "modelVersion": "2020072814115256400000",
    "timestamp": 1595916929427
  },
  "flag": 0,
  "batchDataList": [
    {
      "index": 0,
      "retcode": 0,
      "retmsg": "",
      "data": {
        "score": 0.5386207970765767
      }
    },
    {
      "index": 1,
      "retcode": 0,
      "retmsg": "",
      "data": {
        "score": 0.5386207970765767
      }
    }
  ],
  "singleInferenceResultMap": {
    "0": {
      "index": 0,
      "retcode": 0,
      "retmsg": "",

```

```

        "data": {
            "score": 0.5386207970765767
        }
    },
    "1": {
        "index": 1,
        "retcode": 0,
        "retmsg": "",
        "data": {
            "score": 0.5386207970765767
        }
    }
}
}
}

```

java版SDK

目前提供了java版sdk，可以通过使用java版SDK来对接serving-server，sdk提供服务发现以及路由功能。需要使用在目标工程pom文件中加入相关依赖

操作步骤

- 1.cd fate-serving源码根目录，执行 `mvn clean install -pl com.webank.ai.fate:fate-serving-core,com.webank.ai.fate:fate-serving-sdk,com.webank.ai.fate:fate-serving-register,com.webank.ai.fate:fate-serving`
- 2.在目标工程文件中加入依赖

```

<dependency>
    <groupId>com.webank.ai.fate</groupId>
    <artifactId>fate-serving-sdk</artifactId>
    <version>2.0.0</version>
</dependency>

```

示例

```

import java.io.IOException;
import java.io.OutputStream;
import java.net.Socket;
import java.util.Map;

import com.google.common.collect.Maps;
import com.webank.ai.fate.serving.core.bean.InferenceRequest;
import com.webank.ai.fate.serving.core.bean.ReturnResult;
import com.webank.ai.fate.serving.sdk.client.Client;

public class ClientTest {

```

```

static Client client =
Client.getClient("localhost:2181,localhost:2182");

static InferenceRequest buildInferenceRequest(){
    InferenceRequest inferenceRequest = new InferenceRequest();
    inferenceRequest.setServiceId("mytest");
    Map<String,Object> featureData = Maps.newHashMap();
    featureData.put("x1",0.1);
    inferenceRequest.setFeatureData(featureData);
    Map<String,Object> sendToRemote = Maps.newHashMap();
    sendToRemote.put("phone_num","1233333");
    inferenceRequest.setSendToRemoteFeatureData(sendToRemote);
    return inferenceRequest;
}

public static void main(String[] args) throws IOException,
InterruptedException {
    InferenceRequest inferenceRequest = buildInferenceRequest();
    try {
        ReturnResult returnResult =
client.singleInference(inferenceRequest);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

错误码表

错误码目前为4位，最高位1代表guest，2代表host

错误码低三位	描述
0	成功
100	参数解析错误
102	Adaptor 特征错误
104	模型不存在
105	网络错误
106	路由错误
107	加载模型失败
108	合并预测结果失败
109	绑定模型失败
110	系统错误
111	服务已关闭
113	特征不存在
116	Adaptor 返回失败
120	请求参数错误
121	鉴权失败
122	未找到服务
123	模型初始化失败
124	请求负载错误
125	不支持的指令
126	取消注册错误
127	无效的口令
128	serving-proxy路由错误
129	serving-proxy鉴权失败

HOST如何获取特征

FATE-Serving在调用在线预测接口时，需要数据使用方（Guest）、数据提供方（Host）双方联合预测，Guest方对模型和特征数据进行业务处理后，Guest方接口参数中的 `sendToRemoteFeatureData` 会发往Host端，Host方则是通过自定义的Adaptor跟己方业务系统交互（eg：通过访问远程rpc接口、或者通过访问存储）来获取特征数据，并将获取的特征交给算法模块进行计算，最终得出合并后的预测

结果并返回给Guest。源码中提供了两个抽象类AbstractSingleFeatureDataAdapter、AbstractBatchFeatureDataAdapter用于可用于继承并实现自定义的接口类。

默认的情况使用系统自带的MockAdaptor，仅返回固定数据用于简单测试，实际生产环境中需要使用者需要自行开发并对接自己的业务系统。

自定义Adapter开发

实现自定义Adapter，只需要分别继承AbstractSingleFeatureDataAdapter或AbstractBatchFeatureDataAdapter并重写父类抽象方法，AbstractSingleFeatureDataAdapter用于在线单次预测业务，AbstractBatchFeatureDataAdapter用于在线批量预测业务。

init方法中可直接使用environment获取serving-server.properties配置中参数

```
public class CustomAdapter extends AbstractSingleFeatureDataAdapter {

    @Override
    public void init() {
        // init() 方法中可以直接使用environment对象
        // environment.getProperty("port");
    }

    @Override
    public ReturnResult getData(Context context, Map<String, Object> featureIds)
    {
        // ...
    }
}

public class CustomBatchAdapter extends AbstractBatchFeatureDataAdapter {

    @Override
    public void init() {

    }

    @Override
    public BatchHostFeatureAdaptorResult getFeatures(Context context,
        List<BatchHostFederatedParams.SingleInferenceData> featureIdList) {
        // ...
    }
}
```

Context为上下文信息，用于传递请求所需参数，featureIds用于传递数据提供方传递过来的特征ID (eg:手机号、设备号)


```
# 在host方的配置文件serving-server.properties中将其配置成自定义的类的全路径，如下所示
feature.single.adaptor=com.webank.ai.fate.serving.adaptor.dataaccess.CustomAdapter
feature.batch.adaptor=com.webank.ai.fate.serving.adaptor.dataaccess.CustomBatchAdapter
```

可以根据需要实现Adapter中的逻辑，并修改 `serving-server.properties` 中 `feature.single.adaptor` 或 `feature.batch.adaptor` 配置项为新增Adapter的全类名即可。可以参考源码中的MockAdapter

fate-serving-extension

为了更好的代码解耦合，代码中将自定义adapter分离到 `fate-serving-extension` 模块中。用户可在此模块中开发自定义的adapter。

```
mvn clean package -pl fate-serving-extension -am
```

单独打包 `fate-serving-extension`，将 `target/fate-serving-extension-{version}.jar` 拷贝到 `serving-server` 部署目录下 `extension` 中覆盖，重启服务即可生效。

`serving-server` 部署目录下 `extension` 已加载到类路径

预设Adapter

fate-serving-extension中预设了6种Adapter的简单实现

MockAdapter

固定返回mock特征数据

MockBatchAdapter

用于批量预测，原理同上

BatchTestFileAdapter

用于批量预测，原理同上

TestFileAdapter

从host_data.csv种读取特征数据，每次调用返回值为csv种所有内容, host_data.csv需上传至Host方 serving-server实例部署根目录下，

```
x0:-0.320167,x1:0.58883,x2:-0.18408,x3:-0.384207,x4:2.201839,x5:1.68401,x6:1.219096,x7:1.150692,x8:1.9656,x9:1.572462,x10:-0.35685
x0:1,x1:5,x2:13,x3:58,x4:95,x5:352,x6:418,x7:833,x8:888,x9:937,x10:32776
```

日志

目前fate-serving 使用log4j2作为日志组件，使用log4j2.xml来作为配置文件。

```
<Configuration status="ERROR" monitorInterval="60">
  <Properties>
    <Property name="logdir">logs</Property>
    <Property name="project">fate</Property>
    <Property name="module">serving-server</Property>
  </Properties>
  <Appenders>
    <Console name="console" target="SYSTEM_OUT">
      <PatternLayout charset="UTF-8"
        pattern="%d{yyyy-MM-dd HH:mm:ss,SSS} [%-5p] %c{1.} (%F:%L) - %m%n"/>
    </Console>
    <RollingFile name="info" fileName="${logdir}/${project}-${module}.log"
      filePattern="${logdir}/%d{yyyy-MM-dd}/${project}-${module}.log.%d{yyyy-MM-dd}">
      <PatternLayout charset="UTF-8" pattern="%d{yyyy-MM-dd HH:mm:ss,SSS} [%-5p] %c{1.} (%F:%L) - %m%n"/>
      <Policies>
        <TimeBasedTriggeringPolicy/>
      </Policies>
      <DefaultRolloverStrategy max="24"/>
    </RollingFile>

    <RollingFile name="flow" fileName="${logdir}/flow.log"
      filePattern="${logdir}/%d{yyyy-MM-dd}/flow.log.%d{yyyy-MM-dd}.log">
      <PatternLayout charset="UTF-8" pattern="%d{yyyy-MM-dd HH:mm:ss,SSS}|%m%n"/>
      <Policies>
        <TimeBasedTriggeringPolicy/>
      </Policies>
      <DefaultRolloverStrategy max="24"/>
    </RollingFile>

    <RollingFile name="error" fileName="${logdir}/${project}-${module}-error.log"
      filePattern="${logdir}/${project}-${module}-error.log.%d{yyyy-MM-dd}.log">
      <PatternLayout charset="UTF-8" pattern="%d{yyyy-MM-dd HH:mm:ss,SSS} [%-5p] %c{1.} (%F:%L) - %m%n"/>
      <Policies>
        <TimeBasedTriggeringPolicy/>
      </Policies>
      <DefaultRolloverStrategy max="24"/>
    </RollingFile>
  </Appenders>
</Configuration>
```

```

</RollingFile>

    <RollingFile name="debug" fileName="${logdir}/${project}-${module}-
debug.log"
                filePattern="${logdir}/${project}-${module}-
debug.log.%d{yyyy-MM-dd}.log">
        <PatternLayout charset="UTF-8" pattern="%d{yyyy-MM-dd
HH:mm:ss,SSS} [%-5p] %c{1.}(%F:%L) - %m%n"/>
        <Policies>
            <TimeBasedTriggeringPolicy/>
        </Policies>
        <DefaultRolloverStrategy max="24"/>
    </RollingFile>

</Appenders>

<Loggers>
    <logger name="org.apache.zookeeper" level="WARN"></logger>

    <AsyncLogger name="flow" level="info" includeLocation="true"
additivity="true">
        <AppenderRef ref="flow"/>
    </AsyncLogger>

    <AsyncRoot level="info" includeLocation="true">
        <AppenderRef ref="console" level="info"/>
        <AppenderRef ref="info" level="info"/>
        <AppenderRef ref="error" level="error"/>
    </AsyncRoot>
</Loggers>
</Configuration>

```

默认日志路径会打在启动目录的 `logs` 文件夹中 日志文件有四个：

- fate-`${module}`.log
该日志为INFO日志，用于记录INFO级别日志,同时ERROR级别日志也会在其中体现
- fate-`${module}`-error.log
该日志为错误日志，用于记录error级别日志
- fate-`${module}`-debug.log
该日志为调试日志，用于记录debug级别日志，该日志需要手动开启
- flow.log
该日志为访问日志，用于记录每一笔请求的到达时间、耗时、返回码、请求参数等

serving-server的访问日志格式

```
2020-08-10
12:47:49,708|6578868d962047c996c0505821cae830|2113|6|5|singleInference|172.16.
153.105:8879||
```

时间|CaselId|返回码|总耗时|下游通信耗时|服务名称|路由信息|请求参数|返回值

serving-proxy的访问日志格式

```
2020-08-10
14:08:39,430|127.0.0.1|1597039719427|9999|10000|0|3|3|unaryCall|172.16.153.136
:8869||
```

时间|目标地址|CaselId|GuestAppld|HostAppld|返回码|总耗时|下游通信耗时|服务名称|路由信息|请求参数|返回值

可选打印内容

工程的配置文件中现提供两个可选参数，启动即可在flow日志中打印请求的参数和返回值，可以根据自己需要修改参数并重启系统

```
print.input.data=true    // flow日志打印参数
print.output.data=true   // flow日志打印返回值
```

fate-serving-client命令行工具

FATE-Serving提供了fate-serving-client工具进行便捷调用。下载系统对应版本fate-serving-client：

- linux版本：[fate-serving-client-2.0.0-linux.tar.gz](#)
- mac版本：[fate-serving-client-2.0.0-darwin.tar.gz](#)

在终端中使用 `./fate-serving-client` 启动，默认情况下，client连接localhost:8000，使用参数可以指定目标地址 `./fate-serving-client [-h host] [-p port]`

```
./fate-serving-client 127.0.0.1 8000
```

提供了以下几种指令：

- showconfig 查看服务配置
- showmodel 查看已发布的模型信息
- inference 在线单笔预测，参数为参数文件路径，如：inference /data/projects/request.json 参数格式 { "servicelId": "lr-test", "featureData": { "x0": 0.100016, "x1": 1.21, "x2": 2.321, "x3": 3.432, "x4": 4.543, "x5": 5.654, "x6": 5.654, "x7": 0.102345 }, "sendToRemoteFeatureData": {

```
"device_id": "8" } }
```

- batchInference 在线批量预测，参数为参数文件路径，如：inference /data/projects/request.json **参数格式** { "serviceId": "lr-test", "batchDataList": [{ "index": 0, "featureData": { "x0": 0.4853, "x1": 1.1996, "x2": -1.574, "x3": -0.8811, "x4": -0.6176, "x5": 0.5997, "x6": -0.5361, "x7": -0.1189, "x8": -1.5728 }, "sendToRemoteFeatureData": { "device_id": "299", "phone_num": 585 } }] }
- help 查看帮助信息
- quit 关闭连接

fate-serving 的服务治理

在线预测部分主要有一下几个模块会进行rpc调用：

- serving-server
- serving-proxy
- fateflow

其中serving-server 和serving-proxy对外提供了预测接口， 承载业务流量， 所以有弹性扩缩容的需求， 同时要求高可用。 针对以上的几点， fate-serving使用了zookeeper作为注册中心， 管理各模块的服务注册以及发现。可参考代码中register模块。目前源码默认使用zookeeper。可参考 serving-server 配置详解， 以及 serving-proxy配置详解。

zookeeper中的数据结构

zookeeper中使用的数据结构如下

```
/FATE-SERVICES/{模块名}/{ID}/{接口名}/provider/{服务提供者信息}
```

接下来说明zookeeper各级路径的含义

- 第一级路径：永久节点，固定为FATE-SERVICES
- 第二级路径：永久节点，各自模块的名字，比如serving-proxy的模块名为proxy， serving-server的模块名为serving
- 第三级路径：永久节点，ID， 根据接口的不同ID的值不一样。与模型版本强相关的接口，如inference接口， ID为fate-flow推送模型时产生的serviceId， 这样可以使得不同版本模型注册路径不一样。其他接口为字符串online
- 第四级路径：永久节点，固定字符串provider
- 第五级路径：临时节点，详细描述注册信息 eg: grpc://172.168.1.1:8000

订阅与注册的可靠性保证

目前订阅/取消订阅、 注册/取消注册操作都使用了定时重试的机制来保证操作的最终成功。

客户端订阅务信息的缓存与持久化

默认在当前用户目录下生成.fate 的文件夹，所有的持久化信息都会放入该文件夹。使用服务治理管理的模块启动之后，首先会从本地缓存文件加载之前订阅的接口，然后再从注册中心拉取并更新本地文件。在极端情况下，如注册中心宕机，本地的持久化文件将继续服务，不会影响业务流量。

客户端的负载均衡

目前支持

- 随机(默认)
- 加权随机

服务端的优雅停机

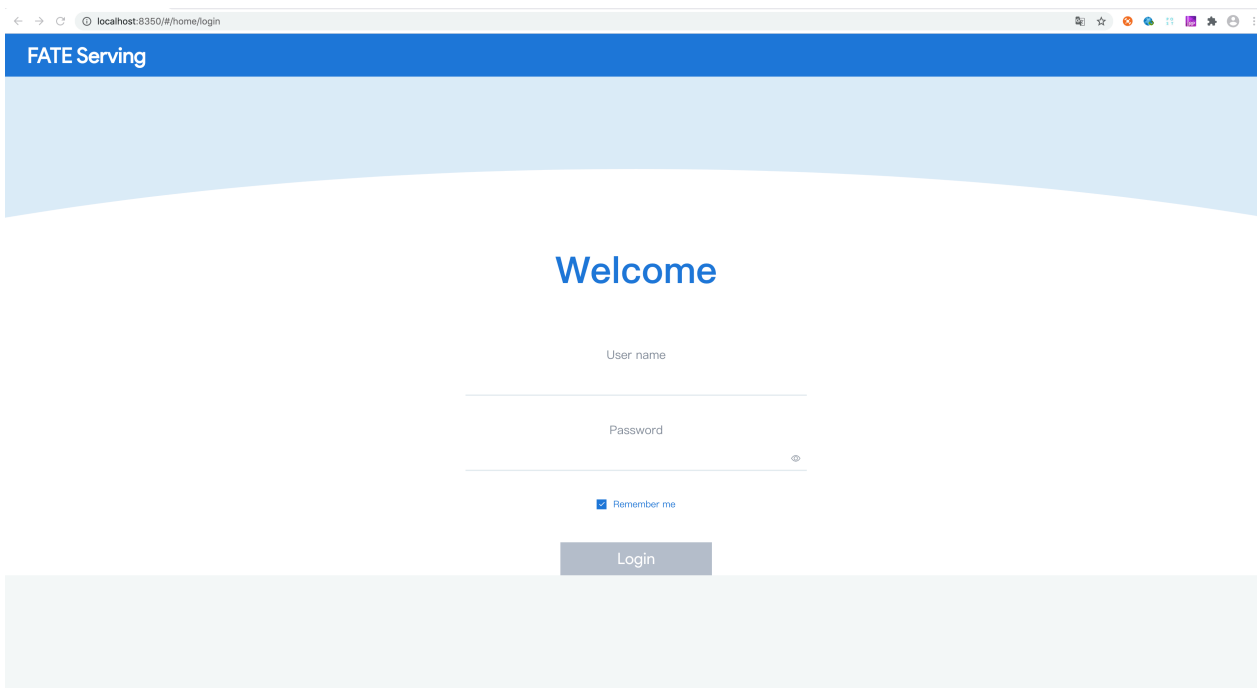
服务端会在jvm 退出时，主动取消注册在注册中心的接口，拒绝新的请求，并等待当前正在处理的请求退出。需要注意的是，不能使用kill -9 命令退出，这样不会触发jvm退出前的动作，如果进程意外退出，zookeeper会判断心跳超时，所创建的临时节点会消失，但是在心跳还未超时的这段时间里，业务流量会被路由到当前已被kill的实例上来，造成风险。建议使用kill

serving-admin介绍

介绍

serving-admin提供了FATE-Serving集群的可视化操作界面，依赖zookeeper注册中心，可以展示集群中各实例的配置、状态、模型、监控等信息，并可以执行模型的卸载、服务接口的权重调整、流量控制等操作。默认端口8350，可以根据自己需要修改端口。

在安装serving-admin并启动之后，可以通过浏览器进行访问,默认用户名 admin 密码admin



前端部分

- 使用Vue.js作为javascript框架
- fate-serving-admin-ui/README.md中有更详细的介绍

后端部分

- fate-serving-admin为前端提供api接口

Swagger 支持

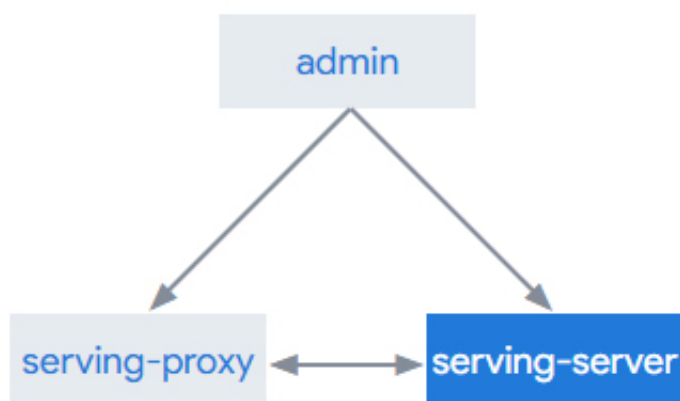
部署完成后，可以访问 <http://localhost:8350/swagger-ui.html> 来查看所有的restful api

功能介绍

节点管理

- 节点展示：提供集群组件展示，可切换实例进行操作

Cluster



serving-server

Click IP to view the instance details

[127.0.0.1:8100](#)

2020-08-03 17:39:31

127.0.0.1:8200

2020-08-05 14:59:57

- 参数配置：展示实例的参数配置及元数据，参数解释请参考[配置详解](#)

Basic	Models	Traffic Monitor	JVM	Q Keyword
key		value		
acl.enable		false		
batch.inference.max		300		
batch.inference.rpc.timeout		3000		
batch.split.size		100		
cache.type		local		
currentVersion		200		
feature.batch.adaptor		com.webank.ai.fate.serving.adaptor.dataaccess.MockBatchAdapter		
feature.single.adaptor		com.webank.ai.fate.serving.adaptor.dataaccess.MockAdapter		
local.cache.expire		30		
local.cache.interval		3		
local.cache.maxsize		10000		
lr.split.size		500		
model.cache.path		/home/app		
model.transfer.uri		http://127.0.0.1:9380/v1/model/transfer		

模型管理

- 模型查询：展示实例从fateflow载入成功的模型，仅显示单方模型，数据使用方会显示绑定的服务ID

Basic

































Models

Traffic Monitor

JVM

Q

Service ID

Model ID	Model Version	Service ID ⇅	Role & Party ID	Timestamp ⇅	Operation
guest#9999#guest-9999#host-10000#model	2020052210090267601672	optimal_binning	guest-9999, host-10000	2020-07-07 18:27:40	 FlowControl  Unload  Unbind
guest#9999#guest-9999#host-10000#model	2020052620094026481582	optimal_binning_3	guest-9999, host-10000	2020-07-07 18:27:41	 FlowControl  Unload  Unbind
host#9999#arbiter-9999#guest-10000#host-...	20200521201025132228190	—	host-9999	2020-07-07 18:27:40	 FlowControl  Unload
guest#9999#guest-9999#host-10000#model	20200616202805301413191	sbt_3w	guest-9999, host-10000	2020-07-07 18:27:42	 FlowControl  Unload  Unbind
guest#9999#guest-9999#host-10000#model	20200617185607637290225	dataio_1	guest-9999, host-10000	2020-07-07 18:27:42	 FlowControl  Unload  Unbind
guest#9999#arbiter-10000#guest-9999#host-...	20200617191614128627228	lr_1000_0	guest-9999, host-10000	2020-07-07 18:27:42	 FlowControl  Unload  Unbind
guest#9999#arbiter-10000#guest-9999#host-...	2020060310164559473015	lr_1	guest-9999, host-10000	2020-07-07 18:27:41	 FlowControl  Unload  Unbind
guest#9999#arbiter-10000#guest-9999#host-...	20200617162628134791204	lr_3w_2	guest-9999, host-10000	2020-07-07 18:27:42	 FlowControl  Unload  Unbind
guest#9999#arbiter-10000#guest-9999#host-...	2020060321225422175141	cwj_sbt_3	guest-9999, host-10000	2020-07-07 18:27:42	 FlowControl  Unload  Unbind
guest#9999#arbiter-10000#guest-9999#host-...	202003041043164787405	—	guest-9999, host-10000	2020-07-07 18:27:40	 FlowControl  Unload
guest#9999#arbiter-10000#guest-9999#host-...	202003041039226809873	—	guest-9999, host-10000	2020-07-07 18:27:40	 FlowControl  Unload
guest#9999#arbiter-10000#guest-9999#host-...	20200616142113301011186	—	guest-9999, host-10000	2020-07-07 18:27:42	 FlowControl  Unload

Total 22

<

1

2

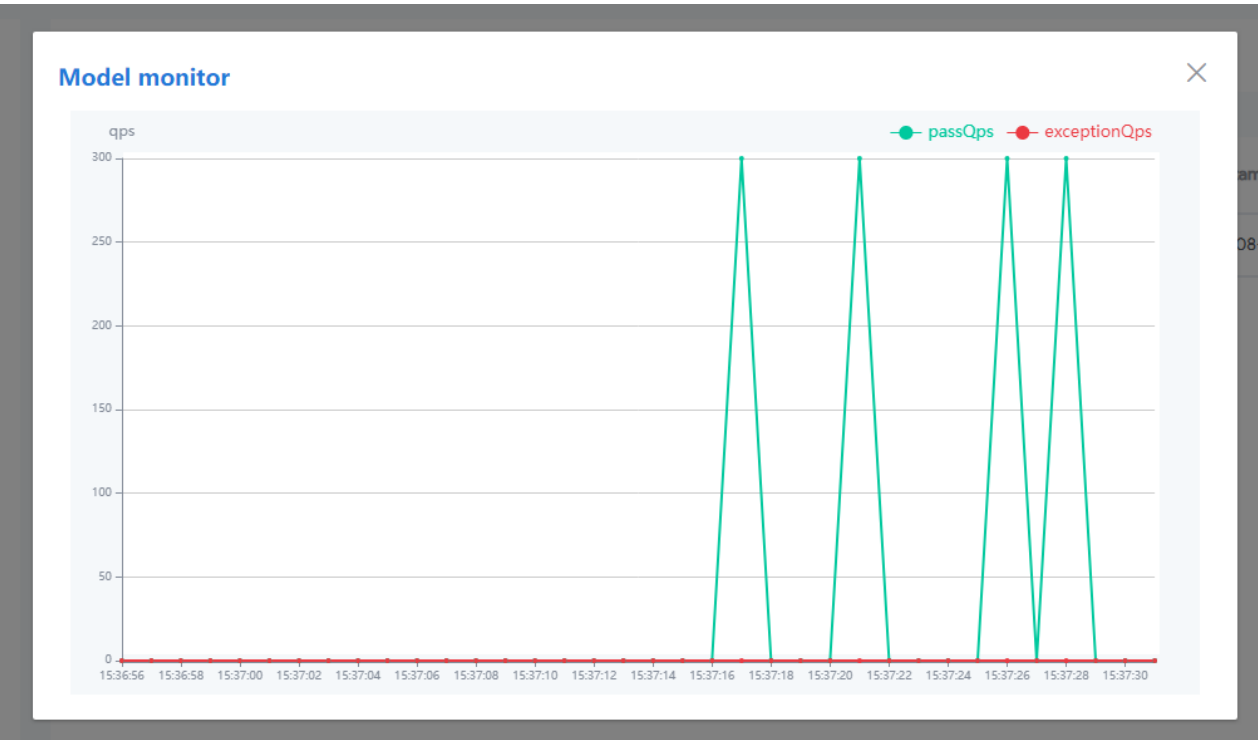
>

Go to 1

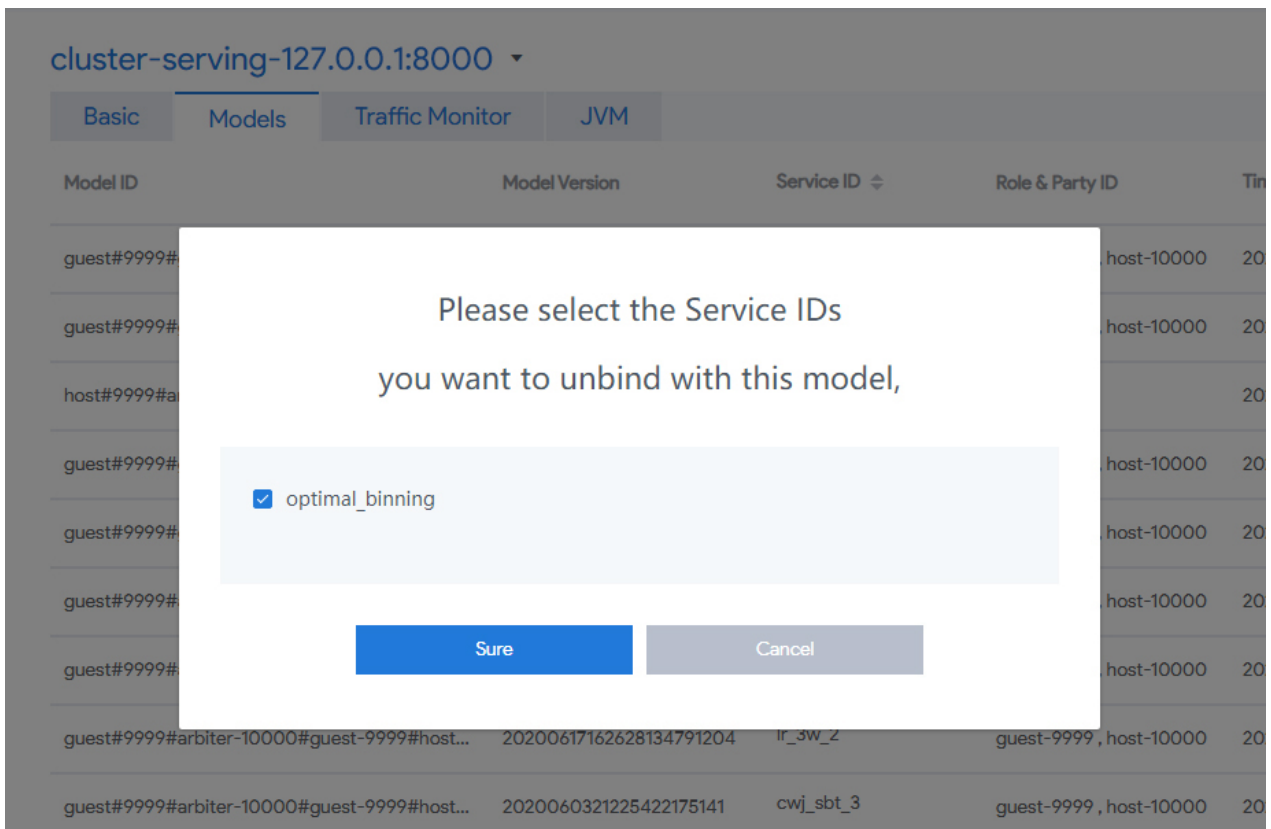
列表字段

字段名	备注
Model ID	模型ID
Model Version	模型版本
Service ID	服务ID
Role & Party ID	模型对应的角色和节点ID
Timestamp	模型发布时间

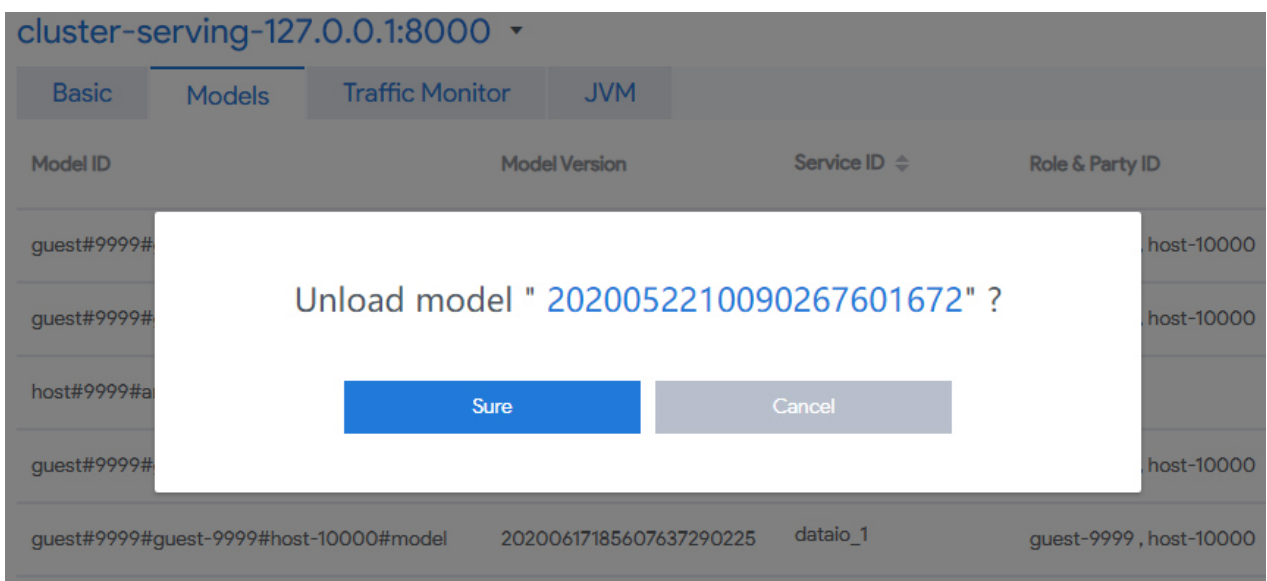
- 模型调用监控：模型列表右侧可展示模型调用监控，绿色线条为调用量，红色线条为异常次数。一次调用批量预测模型调用量为请求参数中 `batchDataList` 数量，



- 模型解绑：可以对模型绑定的服务ID进行解绑，并注销对应服务注册的服务接口



- 模型卸载：卸载实例已载入的模型，模型卸载会同时解绑服务ID，并注销注册的服务接口



模型管理仅在serving-server实例中显示

服务管理

- 服务接口：展示注册中心中各实例注册的服务接口

列表字段

字段名	备注
Project	服务项目，serving-server为serving，serving-proxy为proxy
Environment	环境，模型Bind操作绑定服务ID后，此字段会显示注册的服务接口的服务ID
Name	接口名称，{Project}/{Environment}/{ServiceName}
Host	服务接口对应grpc服务所在主机
Port	服务接口对应grpc服务监听端口
Weight	服务接口路由权重，用于调整服务节点之间的流量分配

- 接口加权：服务接口加权可以更轻松地在服务发现时实施负载均衡

Cluster						
Service						
Project	Environment	Name	Host	Port	Weight	Operation
serving	lr-001	serving/lr-001/unaryCall	127.0.0.1	8000	100	🔗
serving	lr-001	serving/lr-001/inference	127.0.0.1	8000	60	✓ ✕ Verify
serving	lr-001	serving/lr-001/batchInference	127.0.0.1	8000	100	🔗 Verify

- 接口验证：页面提供了接口的简单调用，提供参数即可对接口进行验证，当前可验证 `inference` 和 `batchInference` 接口

Cluster						
Service						
Project	Environment	Name	Host	Port	Weight	Operation
serving	lr-001	serving/lr-001/unaryCall	127.0.0.1	8000	100	🔗
serving	lr-001	serving/lr-001/inference	127.0.0.1	8000	60	✓ ✕ Verify
serving	lr-001	serving/lr-001/batchInference	127.0.0.1	8000	100	🔗 Verify

Request body

```
{  "serviceld": "lr-001",  "featureData": {    "x0": 1.88669,    "x1": -1.359293,    "x2": 2.303601,    "x3": 2.00137,    "x4": 1.307606  }  }
```

Verify

Response

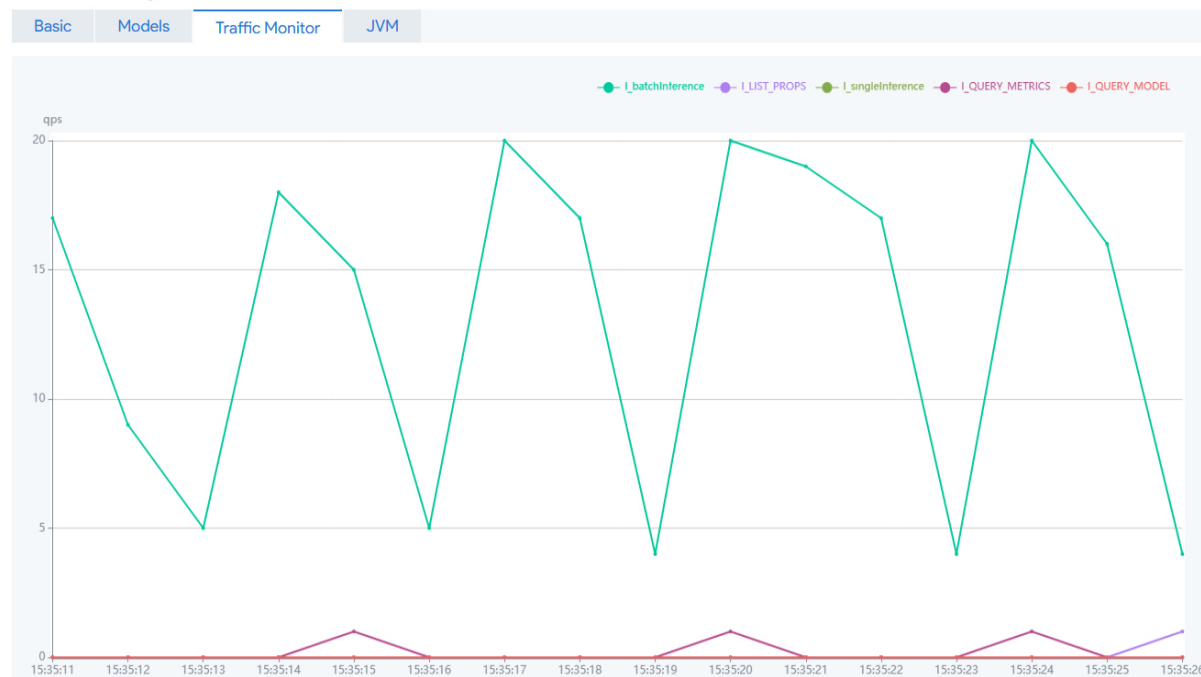
```
{  "retcode": 0,  "retmsg": "",  "caseid": "",  "data": {    "score": 0.6601084304494412,    "modelld": "guest#9999#arbitrator-10000#guest-9999#host-10000#model",    "modelVersion": "2020080416153238910324",    "timestamp": 1596595625179,    "log": {},    "warn": {},    "flag": 0  }  }
```

Close

统计/监控

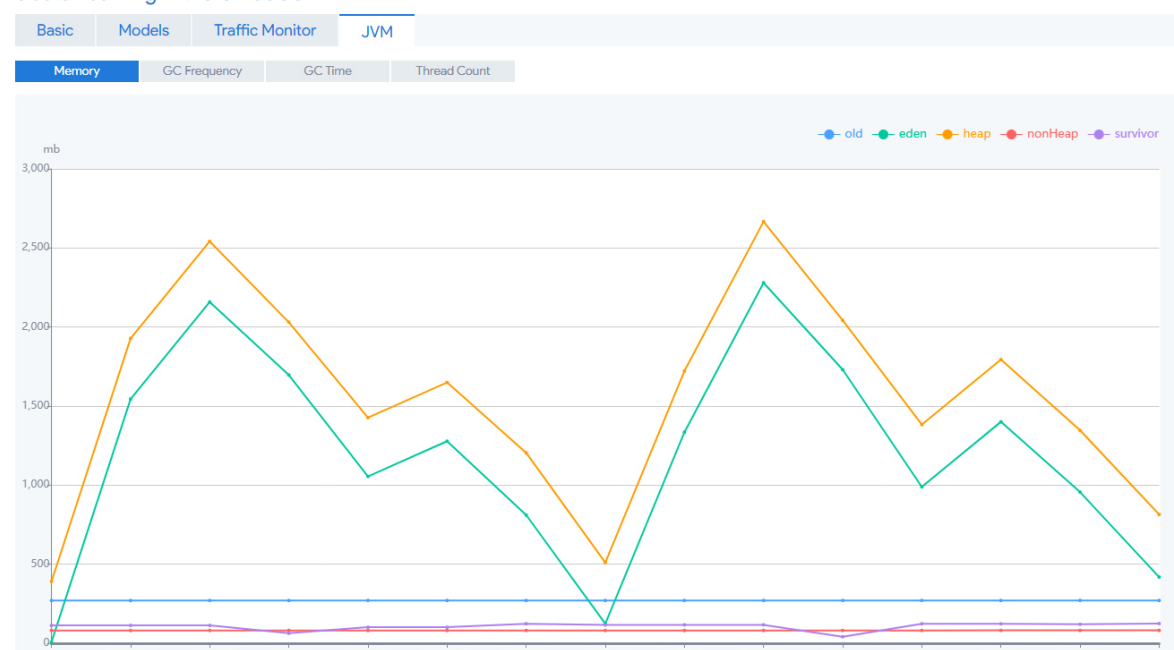
- 调用量统计：提供实例各接口调用量图表展示，此处仅显示接口调用量

cluster-serving-127.0.0.1:8000 ▾



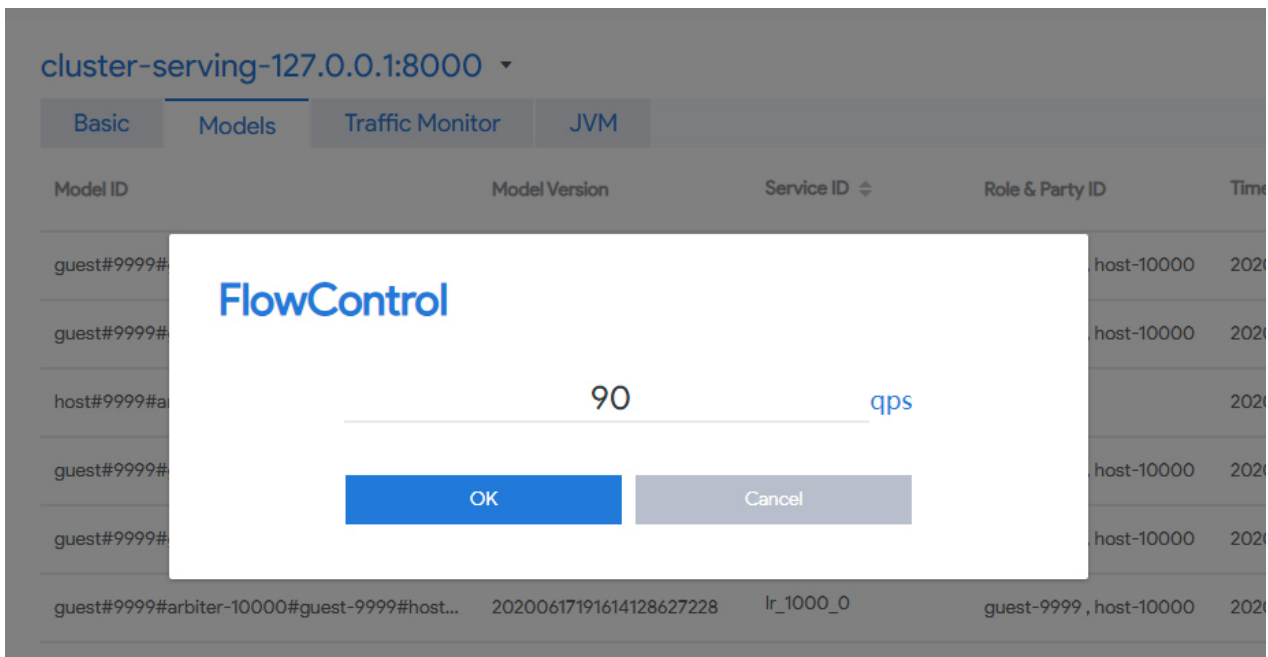
- JVM监控：提供JVM信息图表展示，包含Memory（内存情况）、GC Frequency（GC 频率）、GC Time（GC时间）、Thread Count（线程数量）

cluster-serving-127.0.0.1:8000 ▾



流量控制

- 模型调用流量控制：可通过限制模型调用QPS，可进行流量控制，超过限制的请求会被拦截



用户管理

- 默认用户：**admin**，默认密码：**admin**，用户可在 `application.properties` 中修改预设用户
- serving-admin仅实现简单的用户登录，用户可业务需求，自行实现登录逻辑，或接入第三方平台

配置详解

serving-admin的配置文件

- `application.properties`

源码中的配置文件没有罗列出所有配置，只保留了必需的配置，其他配置都采用了默认值。如果需要可以根据以下表格来在配置文件中新增条目。

`application.properties`

配置项	配置项含义	默认值
<code>server.port</code>	服务端口	8350
<code>cache.type</code>	缓存类型，可选local/redis	local
<code>local.cache.expire</code>	内置缓存过期时间，单位：秒	30
<code>zk.url</code>	zookeeper集群地址，serving-admin需开启注册中心	localhost:2181,localhost:2182,localhost:2183
<code>grpc.timeout</code>	grpc请求超时时间	serving
<code>admin.username</code>	预设用户名	admin
<code>admin.password</code>	预设密码	admin
<code>acl.enable</code>	是否使用zookeeper acl鉴权	false
<code>acl.username</code>	acl 用户名	默认空
<code>acl.password</code>	acl 密码	默认空
<code>print.input.data</code>	flow日志打印请求参数	false

代码导读

算法相关

基础类

PipelineModelProcessor类是模型处理逻辑的实现类，serving-server在收到推送模型的请求后，会在内存中初始化一个PipelineModelProcessor实例，该实例中包含了由模型中各算法组件组成的pipeline。pipeline处理逻辑分为两个阶段 1.本地模型处理（这一步在Guest方，以及Host方都会存在） 2.合并远端数据（这一步只在Guest方存在） BaseComponent 类是所有算法组件的基类，它实现了LocalInferenceAware接口

```
public interface LocalInferenceAware {  
  
    public Map<String, Object> localInference(Context context,  
    List<Map<String, Object>> input);  
  
}
```

PipelineModelProcessor中的pipeline 在检测到组件为LocalInferenceAware 时，会调用localInference方法来进行本地预测。另外一个重要的接口MergeInferenceAware,实现该接口的组件可以执行合并远端返回数据的逻辑

```
public interface MergeInferenceAware {  
  
    public Map<String, Object> mergeRemoteInference(Context context,  
    List<Map<String, Object>> localData,                                Map<String,  
    Object> remoteData);  
}
```

HeteroSecureBoost 组件

HeteroSecureBoost 为纵向联邦SecureBoost在线推理的实现过程，与离线不同的是，目前serving只支持单个host的预测。

它主要包含3个文件"HeteroSecureBoost", "HeteroSecureBoostingTreeGuest", "HeteroSecureBoostingTreeHost", 下面对这三个文件展开说明

HeteroSecureBoost

HeteroSecureBoost是HeteroSecureBoostingTreeGuest和HeteroSecureBoostingTreeHost的模型基类，该基类提供了模型的初始化、层次遍历等功能。

1. HeteroSecureBoost继承自BaseModel类，BaseModel类为所有模型的基类，所有算法模型必须继承该类和实现相关接口，用于统一的调度。
2. 模型初始化：InitModel函数，功能是对输入的Meta和Param两个序列化的模型文件进行反序列化，同时，初始化相关的类属性，初始化的内容包括：
treeNum: 树的数量
initScore: boost的初始化得分，具体可参考FATE离线建模文档
trees: 具体的树信息列表，可参考对应的DecisionTreeModelParam
numClasses: 多少类，二分类问题为2，多分类问题则为具体分类数，回归问题为0，通过该字段可以判断具体建模任务类型
classes: 类别标签，对于分类问题，用预测的结果下表去索引真正的分类标签
treeDim: boost的每轮树的数量，对于回归和二分类等于1，对于多分类，是类别数量，每轮每个分类都有一个对应的树
learningRate: 学习率和权重放缩因子，推理时每个树得到的权重都会乘以learning_rate。
3. 功能函数说明：
 - a. getSite: 离线的时候，每个树节点的域信息是role :partyid，如host:10000，通过该函数获取\$role
 - b. generateTag: 用来存储和读取每轮使用的数据，用法在下面介绍
 - c. gotoNextLevel: 输入当前的树、节点编号，特征值，输出树的下一层节点编号

HeteroSecureBoostingTreeGuest and HeteroSecureBoostingTreeHost

HeteroSecureBoostingTreeGuest 和 HeteroSecureBoostingTreeHost是 party guest 和host对应的实现代码，其中party guest收到请求后，会执行推理过程，与此同时，需要与host一起决策每个树的预测流程，下面给出具体说明。

1. HeteroSecureBoostingTreeGuest 发起推理指令给host
2. Host 接收到推理指令后，执行完整的推理调用逻辑（数据预处理->特征工程->HeteroSecureBoost），当首次调用HeteroSecureBoostingTreeHost的predict函数时，host方直接将上游组件的特征数据保存到数据库中。
3. HeteroSecureBoostingTreeGuest继续推理逻辑，首先对每个树进行遍历，等每个树遍历完成之后，如果treeLocation为空，进入5，否则进入4。遍历规则是：
 - a. 如果到达了叶子节点，则记录下节点编号，停止遍历。
 - b. 否则如果该点属于guest方，则使用自身特征数据判断下一层节点的走向，重复该过程直到碰到节点为host方或者走到了叶子节点。
 - c. 如果该点属于host方，则将(treeId, treeNodeId)保存至treeLocation
4. Guest给Host发送treeLocation，host接收到后，从数据库中取出特征数据，然后遍历树集合，将结果反馈给guest，guest得到结果后，返回3
5. Guest得到每个树的节点编号，利用节点编号索引出叶子权重，经过处理后得到预测结果，并将结果输出，推理流程完成。

当前的HeteroSecureBoost交互次数最坏可能达到树的深度，未来我们会进一步对此优化。

HeteroSecureBoost在线推理流程图如下：

HeteroLR 在线推理

HeteroLR为纵向联邦逻辑回归在线推理的实现过程。它主要包含3个文件"HeteroLRBoost", "HeteroLRGuest", "HeteroLRHost", 下面对这三个文件展开说明

HeteroLR

HeteroLR是HeteroLRGuest和HeteroLRHost的模型基类，该基类提供了模型的初始化、模型评分等功能。

1. HeteroLR继承自BaseModel类，BaseModel类为所有模型的基类，所有算法模型必须继承该类和实现相关接口，用于统一的调度。
2. 模型初始化: InitModel函数，功能是对输入的Meta和Param两个序列化的模型文件进行反序列化，同时，初始化相关的类属性，初始化的内容包括: a. weight: LR模型的权值 b. intercept: LR模型的偏置
3. 功能函数说明: a. forward: 计算 $\text{score} = \text{weight} * \text{value} + \text{intercept}$ ，若是host方，则intercept为0

HeteroLRGuest and HeteroLRHost

HeteroLRGuest 和 HeteroLRHost是 party guest和host对应的实现代码，其中party guest收到请求后，会执行推理过程，与此同时，需要与host也一起进行推理，下面给出具体说明。

1. 系统针对请求的id，同时给guest和host发起推理请求
2. Host接收到推理指令后，执行forward函数的前向计算流程，并将结果返回，由系统调度给guest
3. Guest接收到推理指令后，执行forward函数的前向计算流程，并且获取host的计算结果，组合一起，并计算sigmoid得到最终评分，完成完整的推理流程

特征工程组件介绍

目前FATE-serving提供了以下特征工程组件，下面将会一一介绍。

纵向特征分箱（Hetero Feature Binning）

该模块利用训练得到的模型，根据训练时输入的不同参数，将数据转化为数据所在分箱的index。

文件结构

该模块由三个文件组成，分别是HeteroFeatureBinning.java, HeteroFeatureBinningGuest.java以及HeteroFeatureBinningHost.java。其中HeteroFeatureBinning是HeteroFeatureBinningGuest和HeteroFeatureBinningHost的基类。而HeteroFeatureBinning继承了ModelBase。ModelBase是所有模型组件的基类。

参数和方法说明

1. initModel: 将模型的参数和结果，也就是Meta和param文件，反序列化从而对Serving模型初始化。其中从离线模型中继承的参数有：
 - need_run: 是否需要执行，如果为否，这个组件在后续预测时将被跳过
 - transformCols: 需要对哪些列做转化
 - featureBinningResult: 特征分箱后的结果。其中，包含每个特征的，分箱点，woe等
2. handlePredict: 进行转化功能，将数据和模型结果中的splitPoint比较，确定属于哪个分箱后，用分箱的index代替原值。

纵向特征选择（Hetero Feature Selection）

根据训练所得的模型，直接选取在模型中保留的特征，其余特征被过滤掉。

文件结构

该模块由三个文件组成，分别是FeatureSelection.java, HeteroFeatureSelectionGuest.java以及HeteroFeatureSelectionHost.java。其中FeatureSelection是HeteroFeatureSelectionGuest和HeteroFeatureSelectionHost的基类。而FeatureSelection继承了ModelBase。ModelBase是所有模型组件的基类。

参数和方法说明

1. initModel：将模型的参数和结果，也就是Meta和param文件，反序列化从而对Serving模型初始化。其中从离线模型中继承的参数有：
 - need_run：是否需要执行，如果为否，这个组件在后续预测时将被跳过
 - finalLeftCols：经过特征选择后，需要保留的列名。
2. handlePredict：进行转化功能，将输入数据中，属于最终需要保留的变量留下，其余变量被过滤掉。

One-Hot组件

利用训练得到的模型，将预测数据转成one-hot模式。请注意，该组件目前只支持整形数的输入，如果原始数据不是整数，可以配合Feature-binning组件使用。

该组件的文件为OneHotEncoder.java。

参数和方法说明

1. initModel：将模型的参数和结果，也就是Meta和param文件，反序列化从而对Serving模型初始化。其中从离线模型中继承的参数有：
 - need_run：是否需要执行，如果为否，这个组件在后续预测时将被跳过
 - cols：需要做转化的列名
 - colsMapMap：每个需要转化的列，各种可能的值对应的新列名
2. handlePredict：进行转化功能，对每个需要转化的输入数据，和colsMapMap中的key做对比，当相等时，将对应的新列名的值设定为1，其余值均设定为0.如果其中没有值与输入数据相等，则所有新列名对应的值均为0。

Scale组件

利用训练得到的模型，对预测数据进行归一化。目前之前的归一化包括min-max-scale和standard-scale 该组件的文件包括Scale.java, MinMaxScale.java, StandardScale.java

参数和方法说明

1. initModel：将模型的参数和结果，也就是Meta和param文件，反序列化从而对Serving模型初始化。其中从离线模型中继承的参数有：
 - need_run：布尔类型，是否需要执行，如果为否，这个组件在后续预测时将被跳过
 - method：字符串类型，离线归一化的方法，包括min-max-scale和standards-scale
 - mode：字符串类型，归一化对应的模式，包括"normal"和"cap"
 - area: 字符串类型，归一化的范围，"all"表示全部列都归一化，"col"表示只对参数"scale_column"对应的列进行归一化
 - scale_column: 字符串数组，参见"area"
 - feat_upper: 字符串数组，未归一化前，每一列特征数据的上限，当数据值超过上限，则用上限值代替原来的值

- feat_lower: 字符串数组，未归一化前，每一列特征数据的下限，当数据值低于下限，则用下限值代替原来的值
 - with_mean: 布尔类型，standard-scale方法对应的参数，当为True时候，原始数值会减去均值
 - with_std: 布尔类型，standard-scale方法对应的参数，当为True时候，数值会除以标准差
 - column_upper: 浮点数数组，表示每一列的上限，参见"feat_upper"
 - column_lower: 浮点数数组，表示每一列的下限，参见"feat_lower"
 - mean: 浮点数数组，表示每一列的均值，参见"with_mean"
 - std: 浮点数数组，表示每一列的标准差，参见"with_std"
2. transform: 利用离线训练产生的数据，对在线数据做同样处理。根据归一化方法的不同，处理方法分别对应min-max-scale和standard-scale

Imputer组件

缺失值在线处理模块，若离线建模，dataio有应用到缺失值处理，则在线推理也会经过相应处理，和dataio一样，缺失值模块会优先于其他在线推理组件。缺失值组件对应的文件为Imputer.java

参数和方法说明

1. Imputer: 初始化模型参数，包括：
 - missingValueSet: 离线训练时，包含异常值的每一列的列名，即变量名
 - missingReplaceValues: key-value格式，离线训练时，每一列的异常值和对应的替换值
2. transform: 异常值替换功能，具体逻辑为: a.搜索变量是否在离线训练时候进行过异常值处理 b.对进行过异常值处理的变量对应的值，搜索是否在missingReplaceValues中，若在，用替换值替代

Outlier组件

异常值在线处理模块，若离线建模，dataio有应用到异常值处理，则在线推理也会经过相应处理，和dataio一样，异常值模块会优先于其他在线推理组件，在缺失值组件后面。异常值对应的文件为Outlier.java。

参数和方法说明

1. Outlier: 初始化模型参数，包括：
 - outlierValueSet: 离线训练时，包含缺失值的每一列的列名，即变量名
 - outlierReplaceValues: key-value格式，离线训练时，每一列的缺失值和对应的替换值
2. transform: 缺失值替换功能，具体逻辑为: a.搜索变量是否在离线训练时候进行过缺失值处理 b.对进行过缺失值处理的变量对应的值，搜索是否在missingReplaceValues中，若在，用替换值替代

以上模块在在线推理阶段，皆是单边运行，不涉及多方交互的流程。

FAQ

此处会列出关于 FATE-Serving 的常见问题以及相应的注意事项和解决方案。

Q：请问集群模式中的host和guest是什么作用的，是表示角色权限吗？还是表示联邦学习中的多方？

A：可以理解为guest为数据使用方，host为数据提供方

Q：请问在线预测流程是怎么样子的？

A：双方进行联合建模，由数据使用方（Guest）发起Http/Grpc请求，双方同时对样本数据进行处理，再获取数据提供方（Host）对样本数据的处理结果，进行合并

Q：请问使用GRPC预测需要在host和guest分别执行python inference_request.py 192.168.1.1:8000吗？

A：只需要在guest执行就可以，数据会由guest处理后交给host端再处理，fateflow会负责联动双方，最后返回给客户端一个合并后的值

Q：请问在线预测接口中featureData里的特征两方各写各的吗？是在哪一方执行？

A：在线预测在guest方发起，请求参数中的featureData是提供给Guest方使用，参数中的sendToRemoteFeatureData将会发送到Host方

Q：请问serving-proxy组件的路由表应该要怎么样配置？

A：请参考[路由表配置](#)

Q：请问在生产环境使用，是只需要部署FATE-Serving吗，还是需要和FATE一起部署？

A：FATE属于离线部分，FATE-Serving属于在线部分，FATE-Serving依赖离线建模及FATE-Flow的模型操作（load/bind），需要一起部署，离线训练好模型后，发起load操作，FATE-Flow会根据配置的serving地址，将加载模型的请求发送到serving上，这一步需要手动触发，之后FATE-Serving会从FATE-Flow拉取并加载模型，然后再进行绑定操作，完成之后就可以接受在线请求。

Q：请问用FATE-Flow发布模型并绑定后，怎么向FATE-Serving发在线推理请求？

A：目前FATE-Serving 提供grpc/http接口对外，接口协议在fate-serving源码中inference_service.proto有提供，http接口仅1.2.0以上版本支持，具体可查看[ProxyController.java](#)源码

Q：请问我在在线推理过程中出现1104问题，应该如何去排查这个错误？

A：1104错误是Guest方模型未找到，首先确认下使用FATE-Flow发布模型是否成功，发布时显示状态码为0即为成功，其次发布的模型与绑定的服务ID是否是同一个离线模型，检查模型ID及模型版本是否相同，模型发布成功会在部署目录下 `.fate` 生成模型缓存文件

Q：请问你下预测请求中，featureIds的作用是什么吗？

A：featureIds参数的值为请求参数中的 `sendToRemoteFeatureData`，由Guest传递到Host端，Host根据参数中的特征ID处理后返回一组特征数据，计算后得到 `score` 返回到Guest端进行合并

Q：请问在在线推理的过程中，我在guest中load跟bind模型了，然后调用inference_request.py的时候可以得到一个score，但是我修改host端host_data.csv的特征值的时候score并没有改变，查看日志显示model is null，请问我是应该在host端也要load跟bind模型才行吗？

A：不需要，你在使用 fateflow推模型的时候，会Guest和Host联动推模型的
