|  |
| --- |
|  |
| Jahhan框架说明文档 |
|  |
|  |
| **吴宇静** |
| **2016/11/2 Wednesday** |

|  |
| --- |
| Java服务端框架的说明文档 |

[1 更新说明 4](#_Toc466185453)

[1.1 版本说明 4](#_Toc466185454)

[1.2 更新内容 4](#_Toc466185455)

[1.2.1 1.0.0版本 4](#_Toc466185456)

[2 术语及专用名词说明 4](#_Toc466185457)

[3 服务端框架设计说明 4](#_Toc466185458)

[3.1 框架整体说明 5](#_Toc466185461)

[3.1.1 架构图 5](#_Toc466185462)

[3.1.2 框架特点概述 5](#_Toc466185463)

[3.1.3 框架包依赖关系 6](#_Toc466185464)

[3.1.4 框架包说明 7](#_Toc466185465)

[3.2 base-util概述 8](#_Toc466185466)

[3.2.1 概述 8](#_Toc466185467)

[3.2.2 包结构及类命名规范 8](#_Toc466185468)

[3.3 启动机制 9](#_Toc466185469)

[3.3.1 启动机制概述 9](#_Toc466185471)

[3.3.2 启动机制说明 10](#_Toc466185472)

[3.3.3 系统初始化器详解 10](#_Toc466185473)

[3.4 接口初始化及调用机制 11](#_Toc466185474)

[**3.4.1** ApplicationActionIniter初始化流程及接口调用流程 11](#_Toc466185476)

[3.4.2 接口调用职责链详解 13](#_Toc466185477)

[3.5 关系数据库操作机制 13](#_Toc466185478)

[**3.5.1** Dao层包详解 13](#_Toc466185480)

[3.5.2 dao自动生成的代码详解 14](#_Toc466185481)

[3.5.3 事务操作流程 15](#_Toc466185482)

[3.6 dblogistics机制 16](#_Toc466185483)

[3.6.1 dblogistics使用思想 16](#_Toc466185485)

[3.7 soa机制 17](#_Toc466185486)

[3.7.1 dubbo流程图 17](#_Toc466185488)

[3.8 分布式消息队列机制 17](#_Toc466185489)

[3.8.1 基于redis的消息队列流程图 17](#_Toc466185491)

[3.9 前后端通讯机制 19](#_Toc466185492)

[3.9.1 前后端通讯机制流程图 19](#_Toc466185494)

[3.9.2 前后端通讯协议 20](#_Toc466185495)

[3.10 权限管理机制 21](#_Toc466185496)

[3.10.1 权限管理的表结构 21](#_Toc466185498)

[3.11 工作流机制 21](#_Toc466185499)

[3.11.1 工作流的表结构 21](#_Toc466185501)

[4 开发说明及例子 23](#_Toc466185502)

[4.1 启动模块 23](#_Toc466185504)

[4.1.1 说明 23](#_Toc466185507)

[4.1.2 实例 23](#_Toc466185508)

[4.2 action开发 25](#_Toc466185509)

[4.2.1 注解参数说明 25](#_Toc466185511)

[4.2.2 验证参数说明 26](#_Toc466185512)

[4.2.3 实例 27](#_Toc466185513)

[4.3 biz开发 30](#_Toc466185514)

[4.3.1 说明 30](#_Toc466185516)

[4.4 dao开发 31](#_Toc466185517)

[4.4.1 说明 31](#_Toc466185519)

[4.4.2 实例 31](#_Toc466185520)

[4.5 DBLogistics的repository开发 32](#_Toc466185521)

[4.5.1 说明 32](#_Toc466185523)

[4.5.2 实例 33](#_Toc466185524)

[4.6 定时器开发 36](#_Toc466185525)

[4.6.1 说明 36](#_Toc466185527)

[4.7 soa开发 38](#_Toc466185528)

[4.7.1 说明 38](#_Toc466185530)

[4.7.2 实例 38](#_Toc466185531)

[4.8 消息队列开发 39](#_Toc466185532)

[4.8.1 实例与说明 39](#_Toc466185534)

[4.9 框架提供的部分方法说明 40](#_Toc466185535)

[4.9.1 http连接及回调 40](#_Toc466185537)

[4.10 白盒测试开发 41](#_Toc466185538)

[4.10.1 说明 41](#_Toc466185540)

[4.10.2 实例 42](#_Toc466185541)

[4.10.3 excel配置方式测试 43](#_Toc466185542)

[5 部署说明 43](#_Toc466185543)

[5.1 tomcat部署附加说明 43](#_Toc466185545)

[5.2 redis启动附加说明 44](#_Toc466185546)

[6 系统接口说明 44](#_Toc466185547)

[6.1 fw\_getDomainIp获取服务ip 44](#_Toc466185549)

[6.2 fw\_consumeGetMessage获取接口响应时间信息 45](#_Toc466185550)

[7 配置及参数说明 45](#_Toc466185551)

[7.1 系统错误码 45](#_Toc466185553)

[7.2 配置文件说明 46](#_Toc466185554)

[7.2.1 sys\_baseconf.properties 46](#_Toc466185558)

[7.2.2 system.properties 46](#_Toc466185559)

[**7.2.3** redis.properties 47](#_Toc466185560)

[**7.2.4** domainIp.properties 47](#_Toc466185561)

[**7.2.5** dfs\_client.conf 47](#_Toc466185562)

[**7.2.6** dubbo.properties 48](#_Toc466185563)

[**7.2.7** jdbc.properties 48](#_Toc466185564)

# 更新说明

## 版本说明

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 版本号 | 拟制人/  修改人 | 拟制/修改日期 | 更改理由 | 主要更改内容  （写要点即可） |
| 1.0.0 | 吴宇静 | 2016/11/02 |  | 初版 |
|  |  |  |  |  |
|  |  |  |  |  |

## 更新内容

### 1.0.0版本

初版，包含内容如下：

框架基础层：包含：dubboHessian包，session包，util包

框架组件层：包含：redisCache包，job包，dubboExtention包，DBLogistics包

# 术语及专用名词说明

文档中涉及到术语缩写及专用名词：

|  |  |  |
| --- | --- | --- |
| BMP | Business Process Management | 工作流管理 |
| PLM | Product Lifecycle Management | 产品生命周期管理 |
| DBLogistics | Database logistics | 数据库逻辑 |
| ZK | Zookeeper | Zookeeper |

# 服务端框架设计说明



## 框架整体说明

### 架构图



### 框架特点概述

框架基于http请求与客户端进行交互，可支持websocket；

框架使用guice做为注入框架（不使用spring）；

框架使用dubbo做为soa治理框架（spring依赖部分已经去除）；

框架使用zk做为服务发现程序；

框架的util包为接口包，也包含接口的一些基本实现，其他实现在组件层，组件层的实现可替换；

框架使用redis做为分布式缓存，框架同样使用redis实现了简单的消息队列，如果不使用jahhan提供的实现，实现util包的相关接口就可以了；

框架使用mysql做为基本的数据库，dblogistics为对neo4j及tokumx的混合存储逻辑，做为对关系型数据库的扩展存储；

框架提供了部分代码生成工具，方便快速开发；

框架提供了接口测试工具；

### 框架包依赖关系



部分代码未整理完毕，未上传

### 框架包说明

整体工程可以分为框架层及业务层两大层，框架层又可以分成基础框架层（base），框架组件层（component），框架公共服务层（service），框架部署层（deploy）四层。

公共服务层和部署层可做为业务层代码的样例参考。

框架层中：

基础框架层为框架的基础包，包含的包有：

base-dubboHessian为dubbo优化后的hessian，序列化工具，框架的内部传输及对redis的数据（除session，session采用java序列化方式）保存，均使用hessian方式序列化；

base-util为框架接口及工具以及基础模块包，为其他层框架包的父工程包；

base-session为tomcat把session存至redis的功能包，一般不为框架工程依赖，部署tomcat实例时需打包存在在tomcat的lib包里；

框架组件层为框架实现第三方功能模块的功能包，包含的包有：

component-redisCache为框架redis功能实现包

component-fastDFS为fastDFS文件服务器连接的功能包

component-job为定时器任务的功能包

component-dubboExtension为dubbo功能扩展包

component-DBLogistics为多数据库存储的逻辑功能包

框架公共服务层为公共服务的服务包，包含的包有：

service-base为基础功能包，包含基础数据的dao方法，需被其他功能包依赖

service-soaService放置框架层soa接口与传输的bean

service-weixin为微信对接功能包

service-authority为权限相关功能包

service-BPM为流程管理接口的实现包

框架部署层为可以直接部署运行的公共工程包，包含的包有：

deploy-managerAction为框架管理端的web包

deploy-BMPProvider为流程管理的提供端

deploy-RelationProvider为用户关系的提供端

另外：

业务层中：

Dao层为数据操作层

Biz层为服务层，同时依赖dubbo包，也可单独部署成dubbo提供端。

Action层为接口层，如果有提供对微信的对接接口，可依赖service-weixin包。

Job为定时器任务以及需单独启动消息队列监听器进程

WhiteBoxTest为白盒测试包

SOAService为业务层的dubbo接口包

SOAProvider为业务层的dubbo实现包

## base-util概述

### 概述

util包作为所有框架包的父包，主要用于存放下层框架包及业务代码需实现的接口，公用的实现逻辑及方法等。

### 包结构及类命名规范

单词含义：



图中蓝色方块代表具体功能，橘色方块代表相应的文件命名规范。

包名及用途含义

|  |  |
| --- | --- |
| 包名 | 用途及解释 |
| annotation | 注解 |
| api | 框架对业务层提供的接口 |
| constant | 常量 |
| context | 上下文 |
| db | 框架的数据层逻辑块 |
| demand | Util对下层框架提供的需下层框架实现的接口 |
| exception | 异常 |
| factory | Util对一些实现的工厂方法 |
| guiceutil | 需要用guice方式使用的util方法 |
| handler | Handler接口及实现（部分handler接口在自己的逻辑块内） |
| init | 启动器的逻辑块 |
| main | 以线程启动的框架入口 |
| mq | 消息队列的逻辑块 |
| service | 框架为业务层提供的接口 |
| thread | Util的一些线程方法 |
| utils | 用static方式为业务层提供的方法 |
| version | 版本兼容的逻辑块 |
| web | 从tomcat启动的框架入口及相关逻辑代码，包含接口拦截处理以及字段校验等逻辑 |

## 启动机制



### 启动机制概述

框架基于guice依赖注入，启动过程中先加载guice的model模块，后初加载框架初始化器进行其他初始化加载，加载过程中如遇到需要读取配置文件，则自动去初始化PropertiesUtil类，自动扫描加载properties文件。



### 启动机制说明

系统初始化模块位于net.jahhan.init.initer|module包底下，业务层使用初始化器应把初始化器置于公司名称+init.initer|module包底下，如：com.xxx.init.initer|module.

module模块用来guice绑定，initer为其他模块启动器。

module模块需继承AbstractModule类，initer需实现BootstrapInit接口，两种初始化器都需有@InitAnnocation注解。

@InitAnnocation注解中，isLazy参数为是否懒加载，默认为false;initSequence为启动顺序，默认为9999，isLazy为false是生效，启动顺序不能重复。onlyWeb为true时，soa启动时不加载，默认为false，web工程不管true和false都加载。

web工程进入启动机制的入口为ApplicationContextListener类。

main方法启动可以运行net.jahhan.main.Start里的main方法。

### 系统初始化器详解

module模块：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 模块类名 | 说明 | 启动顺序 | 所属包 | OnlyWeb |
| BaseModule | 为Dao注入实现类 | 1300 | redisCache | False |
| Servlet3Module | 扫描绑定servlet与url | 9000 | util | True |
| DemandModule | Demand接口的cache实现 | 1400 | redisCache | True |
| RedisMQModule | Redis消息队列实现方法绑定 | 1100 | redisCache | False |
| AuthorityServiceModule | 权限相关service注入实现类 | 2200 | service-authority | True |
| AssamblyModule | 流程组件实现方法的绑定 | 3100 | service-BPM | False |
| ComponentModule | 流程构件实现方法的绑定 | 3200 | service-BPM | False |
| WeixinDaoModule | 微信Dao注入实现类 | 4100 | service-weixin | False |
| SerializerModule | 序列化接口注入实现类 | 1000 | redisCache | False |
| ServiceModule | Service接口注入 | 2000 | redisCache | False |
| QuarzModule | Job相关的接口注入 | 5100 | job | False |

initer模块：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 模块类名 | 说明 | 启动顺序 | 所属包 | OnlyWeb |
| ApplicationActionIniter | 接口初始化 | 9000 | util | True |
| ActionInfoSyncIniter | 接口信息同步数据库 | 懒加载 | service-authority | True |
| ActionHandlerIniter | Action处理器初始化 | 8000 | util | True |
| ApplicationContextIniter | 上下文初始化 | 7000 | util | False |
| ConnectionManagerIniter | 数据库session及事件监听初始化 | 3000 | util | False |
| DBConnFactoryIniter | 数据库连接工厂初始化 | 2000 | util | False |
| DomainIpSetIniter | 域名ip配置初始化 | 4000 | util | True |
| CacheFactoryIniter | Redis工厂初始化 | 1000 | redisCache | False |
| FilsysIniter | 文件传输的初始化 | 5000 | fastDFS | False |
| DubboIniter | Dubbo初始化 | 6000 | DubboExtension | False |
| RedisMQIniter | Redis消息队列初始化 | 懒加载 | redisCache | False |
| CacheSetIniter | 框架本地缓存初始化 | 懒加载 | service-authority | True |
| AssemblyIniter | 组件初始化 | 懒加载 | service-BPM | True |
| WXHelperHolderIniter | HelperHolder初始化 | 9100 | service-weixin | True |

开发例子见：[启动模块](#_启动模块)

## 接口初始化及调用机制



### ApplicationActionIniter初始化流程及接口调用流程

servlet方式请求的初始化流程和调用流程：



workHandler为职责链处理器，初始化时按顺序由小往大加载，处理时，由大往小执行，执行往业务代码后再由小往大执行。

servlet若为异步可多次响应。

sip协议，参数json实例：

{"service\_id":"接口名","app\_type":客户端及服务端类型号,"ver\_name":"版本号" ,"content":{"user\_id":32," user\_name":"aaa" }}

返回参数为：

{"r\_msg":"success","r\_content":{"balance":"30.85"},"r\_code":0}

请求url为http://域名/工程路径/sip

建议使用sip协议

sic协议为请求参数都往一个content参数中设置，参数json实例：

{content：{"service\_id":"接口名","app\_type":客户端及服务端类型号,"ver\_name":"版本号" ,"content":{"user\_id":32," user\_name":"aaa" }}}

返回参数为：

{"r\_msg":"success","r\_content":{"balance":"30.85"},"r\_code":0}

请求url为http://域名/工程路径/sic

siu协议参数json实例：

{user\_id:32,user\_name:"aaa"}

请求url为http://域名/工程路径/post/接口名

siu协议主要用于支付宝回调

xml协议为请求参数往一个xml文件流里面设置，参数实例为：

<xml>  
   <appid>wx2421b1c4370ec43b</appid>  
   <attach>支付测试</attach>  
 </xml>

请求url为http://域名/工程路径/xml/接口名

xml协议主要用于微信支付回调

websocket通讯流程：



注：接口都为单例

workHandler需添加@HandlerAnnocation注解，value值为加载顺序，从小到大排序。

### 接口调用职责链详解

|  |  |  |
| --- | --- | --- |
| Workhandler | 用途 | 加载顺序 |
| ActionWorkHandler | 解耦用，拦截业务层抛出的异常 | 最内层，action加载前加载 |
| AuthorityValidateWorkHandler | 权限校验 | 90 |
| DBConnHandler | 数据库连接管理 | 100 |
| DecryptWorkHandler | Act请求解密 | 700 |
| SessionValidateWorkHandler | Session验证 | 200 |
| SignValidateWorkHandler | 签名验证 | 400 |
| SimpleMapDataJsonParseWorkHanlder | 请求参数存在验证 | 600 |
| SimpleMapParameterValidateWorkHandler | 请求参数合法性验证 | 500 |
| WriteWorkHandler | 响应处理 | 999 |
| FastBackHandler | 快速返回 | 900 |

## 关系数据库操作机制



### Dao层包详解

java目录：

dao 业务接口

|-abstr 框架自动生成代码的接口的实现类

|-abstrDao 框架自动生成代码的接口

|- cache 业务代码带redis缓存的实现

|- impl 业务代码实现

|- listen redis缓存操作类

pojo 框架自动生成的bean

|-page 框架自动生成的查询类

resource目录：

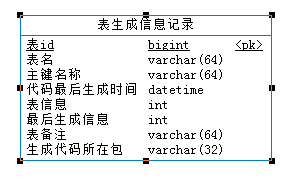
mapper 框架自动生成的mybatis的xml文件

mapper2 为业务实现需要的mybatis的xml文件

注：abstr,abstrDao,listen,pojo ,page, mapper这些包中文件不建议修改，因为这些为自动生成的代码，自动生成时会替换掉。

自动生成的代码根据t\_sys\_table\_info的配置不同可生成不同的代码。

t\_sys\_table\_info表结构：



表信息及最后生成信息字段用2进制来表示表的各种配置

表信息(1:停用,2:不自增4:不开启redis缓存8:不开启线程缓存16扩展的dao方法)

最后生成信息(1:有修改,2:不自增4:不开启redis缓存8:不开启线程缓存16扩展的dao方法)

在表生成后，执行tool-codeTool，会在该表插入一条记录。

若表结构有修改，将最后生成信息的字段加1，然后执行tool-codeTool，则会更新自动生成的代码。

注：如果表信息和最后生成信息字段相同，则不会更新自动生成的代码。

注：一般来说mapper2不允许存在新增，修改及删除的方法，若一定要存在，则需关闭4和8的缓存（即12）

注：一般来说表都需要单个主键，无主键及多主键的表不能用代码字段生成。

### dao自动生成的代码详解

若没有扩展查询方法，自动生成的dao包含如下接口：

|  |  |
| --- | --- |
| 接口名 | 用途 |
| add${tableInfo.className} | 新增 |
| addBatch${tableInfo.className} | 批量增加 |
| resetAll${tableInfo.className} | 根据对象主键更新对象所有字段 |
| updatePart${tableInfo.className} | 根据对象主键部分更新字段 |
| updatePartByIds | 根据主键按条件部分更新 |
| del${tableInfo.className} | 删除 |
| query${tableInfo.className} | 根据主键查询单条 |
| list${tableInfo.className}ByIds | 根据主键Id列出对象，不保证记录的顺序 |
| listIds | 加载指定条件的所有id |
| pagedResultOfList${tableInfo.className} | 带分页的精确查询 |
| delByIds | 批量删除 |
| getMaxSequence() | 获取最大主键 |

可调用的方法如下

|  |  |
| --- | --- |
| 方法名 | 用途 |
| list${tableInfo.className} | 根据条件精确查询 |
| count${tableInfo.className} | 精确查询的数量 |

page包里的实体类具有分页查询，order by，group by的功能。

若使用扩展dao，则多包含如下接口：

|  |  |
| --- | --- |
| 接口名 | 用途 |
| pagedResultOfLike${tableInfo.className} | 带分页的模糊查询（若字段为string类型，则用like方式查询） |

新增的方法如下

|  |  |
| --- | --- |
| 方法名 | 用途 |
| likeList${tableInfo.className} | 根据条件模糊查询（若字段为string类型，则用like方式查询） |
| countLike${tableInfo.className} | 模糊查询的数量 |

page包里的实体类若字段为数值类型或时间类型，则额外增加{字段}\_Begin, {字段}\_End, {字段}\_RangeType等字段用于查询范围确认。begin为开始，end为结束，RangeType为区间类型。close为闭区间，open为开区间。

page包里的实体类字段都额外增加{字段}\_List字段，用于列表查询。

pojo包里的实体类若为数值类型则额外增加{字段}\_Increase字段，用于保存操作时做数值增加的操作。

dao的开发例子见：[dao开发](#_dao开发)

### 事务操作流程

写操作流程：



读操作流程：



## dblogistics机制



### dblogistics使用思想

dblogistics为多数据库操作逻辑。现支持的数据库为：neo4j3.0版本，tokumx1.2版本，redis。

注：neo4j及tokumx都可支持事务管理，所以dblogistics目前也同样支持事务管理。

使用思想：

将个体与个体间的关系使用图数据库保存，但是图数据库不时候保存单个个体的大量的详细数据；将个体的大量数据保存至文档数据库；将这两个分开的数据整合后缓存至redis库，以提升搜索速度。

dblogistics适合用于多个用户不会交叉操作单个节点的场合，如多个用户需要交互操作同一份数据，则最好将数据保存至关系库。

## soa机制



### dubbo流程图

dubbo的基本流程图：



框架的dubbo扩展为去spring化后的扩展，所有关于spring的配置都与框架不兼容，不建议使用。

其他用法及详细解释可参考dubbo官网：

<http://dubbo.io/User+Guide-zh.htm>

开发实例及说明：

[soa开发](#_soa开发)

## 分布式消息队列机制



### 基于redis的消息队列流程图



框架实现的redis消息队列有3种模式：广播者监听者模式，生产者消费者模式，事务安全的生产者消费者模式

广播者监听者模式为广播者发送消息，所有监听者都能收到消息。如果没有监听者，消息会丢失。

生产者消费者模式为生产者发送消息，只有一个消费者能收到消息。如果没有消费者，消息不会丢失。如果消费者获取消息后消费失败，消息也会丢失。

事务安全的生产者模式为队列中只保存消息id信息，redis中再保存一份完整的键值对信息，消费者消费成功后删除该键值对。如果消费失败，消息队列中的id丢失，信息会一直保存在redis中待人工处理。

消息队列的开发例子见：[消息队列开发](#_消息队列开发)

redis的监听类为xxxRegister。

发送类为xxxSender

## 前后端通讯机制



### 前后端通讯机制流程图



注：客户端所有请求在有cookie的时候都必须发送cookie，由服务端判断cookie是否过期，即便重新登陆的情况下，客户端的cookie没过期，服务端也不会返回新的cookie给客户端。cookie应该保存在客户端本地。在重新打开app或者浏览器后也不应重新登陆。

每次请求服务端都会返回cookie新的过期时间。

cookie就是服务端的session id。

### 前后端通讯协议

接口请求字段协议

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **序号** | **名称** | **必填** | **类型** | **描述** |
| 1 | service\_id | Y | String | 服务ID |
| 2 | content | Y | String | json参数或其加密串 |
| 3 | app\_type | Y | Integer | 由3位数值组成，第一位为客户端类型，后两位为APP类型，第一位含义（1：web访问，2：微信，3：安卓，4：IOS）后两位含义 (1管理员端 2车主端 等)  如安卓的车主端app\_type应为302 |
| 4 | ver\_name | N | String | 应用版本 |
| 5 | sign | Y | String | Content的校验码 |

**响应参数（服务发起）：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **序号** | **名称** | **必填** | **类型** | **描述** |
| 1 | r\_code | Y | String | 应用工程返回码 |
| 2 | r\_content | Y | String | 应用工厂返回参数集合 |
| 3 | r\_msg | Y | String | 应用工程返回信息提示 |

## 权限管理机制



### 权限管理的表结构



表大致包括：

用户表，角色表，资源表（绿色表），权限配置表（黄色表），关联表（灰色表）。

用户及角色的权限从权限配置表中获取，不从菜单与其他资源的关联表中获取。

角色，菜单，页面及页面元素和操作有分使用端类型，不同的使用端类型对应不同的数据。

## 工作流机制



### 工作流的表结构

表结构总览：



黄色表为模板相关模块：



一个模板应包含多个构件，每个构件有自己的类型，即组件，包含：开始，选择器，结束，生命周期状态器，与，或等等。选择器构件由权限组控制操作，一个权限组包含多个权限。选择器构件的下一个构件的指向由选择器控制，选择器指向一个指向器，每个指向器包含多个指向器指向。

除选择器外的构件的下一个构件指向由指向器控制。

一个模板可能被多个不同类型的实体使用，使用模板凹槽来等待不同实体填充，达到不同的运行效果。

绿色模块为任务流：



一个任务流程底下有多个任务，执行记录表用于保存执行记录。

红色外框的为生命周期：



每个模板都应该包含一个生命周期模板，一个生命周期模板有多个生命周期状态组成。流程运行时会改变自身的生命周期状态，记录在生命周期记录表中。

橘色外框的为实体信息：



# 开发说明及例子



## 启动模块



### 说明

框架允许在公司名称+init.initer|module包底下添加启动器。

module模块需继承AbstractModule类，initer需实现BootstrapInit接口，两种初始化器都需有@InitAnnocation注解。

@InitAnnocation注解中，isLazy参数为是否懒加载，默认为false，如果为true，将在同类启动器都完成后执行;initSequence为启动顺序，默认为9999，isLazy为false是生效，启动顺序不能重复。onlyWeb为true时，soa或job启动时不加载，默认为false，web工程不管true和false都加载。业务层启动顺序后两位不能都为0。

### 实例

Module实例：

@InitAnnocation(isLazy = **false**, initSequence = 2010)

**public** **class** BizModule **extends** AbstractModule {

**private** **static** Logger *logger* = LoggerFactory.*getLogger*(BizModule.**class**);

@Override

**protected** **void** configure() {

bind(ISmsAgent.**class**).annotatedWith(Names.*named*(SmsAgentType.***MAIN***.name())).to(RuanweiSms.**class**);

bind(ISmsAgent.**class**).annotatedWith(Names.*named*(SmsAgentType.***EXTRA***.name())).to(QixinggangSms.**class**);

}

@Singleton

@Named("PushExecutor")

@Provides

ScheduledExecutorService providePushExecutor() {

**final** ScheduledExecutorService pushExecutor = Executors.*newScheduledThreadPool*(1);

Runtime.*getRuntime*().addShutdownHook(**new** Thread() {

@Override

**public** **void** run() {

**try** {

pushExecutor.shutdown();

**if** (!pushExecutor.awaitTermination(5, TimeUnit.***SECONDS***)) {

pushExecutor.shutdownNow();

}

} **catch** (Exception e) {

*logger*.error("error when shutdown PushExecutor", e);

Thread.*currentThread*().interrupt();

}

}

});

**return** pushExecutor;

}

}

Initer实例：

@InitAnnocation(isLazy = **false**, initSequence = 1011)

**public** **class** XxxIniter **implements** BootstrapInit {

**public** **void** execute() {

//初始化内容

}

}

## action开发



### 注解参数说明

注解说明：

|  |  |  |  |
| --- | --- | --- | --- |
| 注解参数 | 说明 | 默认值 | 有效可选值 |
| act | 接口名 |  |  |
| importantParameters | 必传参数 |  |  |
| minorParameters | 选传参数 |  |  |
| requireLogin | 是否登陆验证 | ***NO*** | ***NO YES*** |
| requestEncrypt | 请求加密方式 | ***PLAIN*** | ***PLAIN***  ***AES***  ***LOGIN*** |
| signEncrypt | 是否参数md5校验 | ***PLAIN*** | ***PLAIN SIGN*** |
| responseEncrypt | 响应返回加密方式 | ***PLAIN*** | ***PLAIN***  ***AES***  ***LOGIN*** |
| returnParameters | 返回参数 |  |  |
| responseType | 返回值方式 | ***JSON*** | ***JSON*** ***CUSTOM***  ***FILE*** ***REDIRECT***  ***NONE*** |
| description | 接口描述 |  |  |
| validateParameters | 请求参数验证方式 |  |  |
| dbConnType | 数据库连接方式 | ***READ*** | ***READ*** ***NONE***  ***WEAK\_WRITE***  ***BATCH\_WRITE***  ***DISTRIBUTED\_WEAK\_WRITE*** |
| requestMethod | 请求方式 | ***JSON*** | ***JSON*** ***POST***  ***XML*** ***WS***  ***BPMRECESSFILL*** |
| referenceObject | 对象类型验证方式使用的验证类 |  |  |
| async | 开启异步 | ***false*** | ***true*** ***false*** |
| version | 代码版本 |  |  |
| fastBack | 快速返回 | ***false*** | ***true*** ***false*** |
| blockTime | 多长时间内快速返回 | ***1*** |  |
| fastBackType | 快速返回粒度 | ***SESSION*** | ***SESSION*** ***USERID*** ***ALL*** |
| fastBackFail | 快速返回失败 | ***true*** | ***true*** ***false*** |

请求加密方式与响应返回加密方式中***PLAIN***为明文，***AES***和***LOGIN***为AES加密，不同的是***AES***用的是sessionId作为加密秘钥，***LOGIN***为全局秘钥。

***WS***全部都是用sessionId作为加密秘钥，其他方式无效。

返回值方式中，***JSON***为正常框架封装返回值，***CUSTOM***为自定义返回值，***FILE***为返回一个文件（***JSON*** ***POST*** ***XML***方式有效），***REDIRECT***为返回一个302跳转（***JSON*** ***POST*** ***XML***方式有效）， ***NONE***为无返回值，在websocket连接时或自定义返回时可用。***BPMRECESSFILL***为bpm的凹槽方法回调。

如果使用对象的验证方式，则validateParameters内的方式无效。校验的类如果为内部类，需为静态内部类。校验类需有@Validation注解。

如接口需调用soa接口，action需添加@Singleton注解。

注：在***POST***请求方式中，若ResponseMessage的lastModify和cache都不为空，则会返回304未修改。

长时间请求可开启异步。

代码版本用于兼容旧版本代码。

若开启快速返回，在指定时间内，若发起相同请求，请求将不再经过action，直接从缓存中返回相同的返回信息或者返回失败。

注：接口为单例模式，不得在类中定义值可变参数。

### 验证参数说明

@Field字段说明

|  |  |
| --- | --- |
| 字段 | 说明 |
| fieldName | 字段名称（类校验方式可以为空） |
| fieldType | 校验方式 |
| defaultValue | 默认值（暂时无效） |
| description | 描述 |
| referenceObject | 使用类校验方式校验，在类校验方式中无法循环使用 |
| importantParameter | 字段是否必填 |

fieldType校验方式及类型转换对应说明

|  |  |  |
| --- | --- | --- |
| fieldType | 转换类型 | 说明 |
| ***TYINT*** | byte |  |
| ***INT*** | Int |  |
| ***SMALL\_INT*** | Short |  |
| ***BIG\_INT*** | Long |  |
| ***BOOLEAN*** | Boolean |  |
| ***DOUBLE*** | Double |  |
| ***CHAR8*** | String | 不超过8个字符 |
| ***CHAR11*** | String | 等于11个字符 |
| ***CHAR24*** | String | 不超过24个字符 |
| ***CHAR32*** | String | 不超过32个字符 |
| ***CHAR36*** | String | 不超过36个字符 |
| ***CHAR64*** | String | 不超过64个字符 |
| ***CHAR128*** | String | 不超过128个字符 |
| ***CHAR512*** | String | 不超过512个字符 |
| ***CHAR1024*** | String | 不超过1024个字符 |
| ***CHAR2048*** | String | 不超过2048个字符 |
| ***CHAR65535*** | String | 不超过65535个字符 |
| ***DATE*** | String（类校验中可直接转Date） | 传入格式为"yyyy-MM-dd" |
| ***DATETIME*** | String（类校验中可直接转Date） | 传入格式为"yyyy-MM-dd HH:mm:ss" |
| ***DATE\_OR\_EMPTY*** | String（类校验中可直接转Date） | 传入格式为"yyyy-MM-dd"或空 |
| ***DATETIME\_OR\_EMPTY*** | String（类校验中可直接转Date） | 传入格式为"yyyy-MM-dd HH:mm:ss"或空 |
| ***TIME*** | String | 传入格式为"HH:mm:ss" |
| ***LIST*** | List |  |
| ***MAP*** | Map |  |
| ***OBJECT*** | 对象 | 类校验中不可用 |
| ***BIGDECIMAL*** | BigDecimal |  |

注解校验方式取值：

Map<String, Object> mapVersion = **new** HashMap<String, Object>();

Integer appType = (Integer) requestMap.get("app\_type");

类校验方式取值：

ParkCoupons parkCoupons=(ParkCoupons)requestMessage.getDefaultObject();

类校验方式同样可以使用注解方式取值。

### 实例

类校验方式类实例：

@Validation

**static** **class** ParkCoupons {

@Field(fieldName="is\_monthly",description = "是否包月")

**public** Long isMonthly;

@Field(fieldName="is\_monthly2",description = "是否包月")

**public** Integer isMonthly2;

@Field(fieldName="is\_monthly3",description = "是否包月")

**public** Date isMonthly3;

@Field(fieldName="is\_monthly4",description = "是否包月")

**public** BigDecimal isMonthly4;

@Field(fieldName="is\_monthly5",description = "是否包月")

**public** List isMonthly5;

@Field(fieldName="is\_monthly6",description = "是否包月")

**public** Map isMonthly6;

}

对应注解写法：

@ActionService(act = "test", description = "测试接口",referenceObject=ParkCoupons.**class**)

注解方式校验实例：

@Field(fieldName = "app\_type", fieldType = FieldTypeEnum.***INT***, description = "app类型"),

@Field(fieldName = "system\_type", fieldType = FieldTypeEnum.***INT***, description = "系统类型")

302跳转实例：

@ActionService(act = "fw\_test\_post", description = "post测试接口",

dbConnType = DBConnectionType.***NONE***,responseType=ResponseTypeEnum.***REDIRECT***,

validateParameters = { })

**public** **class** PostTestAction **implements** Action {

**protected** **static** Logger *logger* = LoggerFactory

.*getLogger*(PostTestAction.**class**);

**protected** **static** Date *last* = **new** Date(System.*currentTimeMillis*()-30\*1000);

@Override

**public** **void** execute(RequestMessage requestMessage,

ResponseMessage responseMessage) {

responseMessage.setRedirect("");

}

}

登陆接口实例：

@ActionService(act = "fw\_test\_login", description = "登陆测试接口",

dbConnType = DBConnectionType.***NONE***, returnParameters={"sessionid"},

validateParameters = { @Field(fieldName = "telephone", fieldType = FieldTypeEnum.***CHAR11***, description = "手机号")},

requireLogin = LoginEnum.***NO***, requestEncrypt = CryptEnum.***LOGIN***,

responseEncrypt = CryptEnum.***LOGIN***,signEncrypt=CryptEnum.***SIGN***)

**public** **class** LoginTestAction **implements** Action {

**protected** **static** Logger *logger* = LoggerFactory

.*getLogger*(LoginTestAction.**class**);

@Override

**public** **void** execute(RequestMessage requestMessage,

ResponseMessage responseMessage) {

ApplicationContext applicationContext = ApplicationContext.*CTX*;

HashMap<String, Object> requestMap = requestMessage.getRequestMap();

String telephone = (String) requestMap.get("telephone");

//业务代码

//

WxUserEntity userEntity = **new** WxUserEntity();

userEntity.setUserName("xxx");

applicationContext.setUserSession(userEntity);

responseMessage.setResponseMap(**new** HashMap<>());

}

}

注：业务接口使用session需先实现一个UserEntity的继承类。

UserEntity继承类实例：

**public** **class** WxUserEntity **extends** UserEntity {

**private** **static** **final** **long** ***serialVersionUID*** = 2939366693509111738L;

**private** String openId;

**private** **boolean** isAttention;

**public** String getOpenId() {

**return** openId;

}

**public** **void** setOpenId(String openId) {

**this**.openId = openId;

}

**public** **boolean** isAttention() {

**return** isAttention;

}

**public** **void** setAttention(**boolean** isAttention) {

**this**.isAttention = isAttention;

}

}

接口调用soa实例：

@ActionService(act = "fw\_test2", description = "测试接口", dbConnType = DBConnectionType.***NONE***, importantParameters = {}, returnParameters = {

"key1" }, validateParameters = {}, requestMethod = RequestMethodEnum.***JSON***, requireLogin = LoginEnum.***NO***)

**public** **class** TestAction **implements** Action {

**protected** **static** Logger *logger* = LoggerFactory.*getLogger*(TestAction.**class**);

@Reference(version = "1.0.1")

**private** TestService testService;

@Override

**public** **void** execute(RequestMessage requestMessage, ResponseMessage responseMessage) {

*logger*.debug("test in");

Map<String, Object> result = **new** HashMap<String, Object>();

testService.call("a");

result.put("key1", "value1");

responseMessage.setResponseMap(result);

}

}

## biz开发



### 说明

在biz层开发多端公共代码中，可以通过注解开关数据库连接。

实例如下：

@Singleton

@Biz

**public** **class** ControllerCache {

@DbConn(DBConnectionType.***READ***)

**public** **void** reset() {

Set<String> unSet = authorityService.getPagesWithoutAuthority();

Map<String, String> pageUrlMap = authorityService.getPageUrlMap();

Map<String, String> urlPageMap = **new** HashMap<>();

Iterator<String> keyIt = pageUrlMap.keySet().iterator();

**while** (keyIt.hasNext()) {

String key = keyIt.next();

urlPageMap.put(pageUrlMap.get(key), key);

}

unauthoritySet = unSet;

pageMap = pageUrlMap;

urlMap = urlPageMap;

}

}

类必须用@Biz注解，方法用@DbConn注解，@DbConn的value值为连接类型，若action已经开过连接，则不再开连接。transaction默认为true，若连接为写类型连接，transaction为false，则只开连接，不开事务。事务可以在里面一次的方法用@transaction注解来开，里面一层的方法的类同样需要用@Biz注解。

## dao开发



### 说明

dao接口继承对应AbstrXxxDao接口

impl接口实现继承对应AbstractXxxDao抽象类

cache接口实现继承对应XxxImpl类，重新使用redis缓存的方法。

mapper为系统自动生成的xml，mapper2放置用户自己写的xml。

### 实例

dao接口实例：

**public** **interface** CarBrandDao **extends** AbstrCarBrandDao{

/\*\*

\* 根据车辆品牌查询品牌信息列表(排序)

\*/

List<CarBrand> queryCarBrandOrderByBrandName(String brandName);

}

impl实例：

**public** **class** CarBrandDaoImpl **extends** AbstractCarBrandDao {

@Override

**protected** **boolean** isCachable() {

**return** **false**;

}

@Override

**public** List<CarBrand> queryCarBrandOrderByBrandName(String brandName) {

CarBrandPage cPage = **new** CarBrandPage();

cPage.setBrandName(brandName);

cPage.setOrder\_by("INITIAL");

List<CarBrand> list = listCarBrand(cPage);

**if** (**null** != list && list.size() > 0)

**return** list;

**return** **null**;

}

}

xml实例：

<!-- 查询-->

<select id=*"queryCityId"* parameterType=*"String"* resultType=*"Long"*>

SELECT city\_id as cityId FROM sys\_city where city\_name like "%"#{cityName}"%"

</select>

## DBLogistics的repository开发



### 说明

node为图库的节点，需继承SuperEntity类，需要有@NodeEntity注解。

relation为图库的关系，需继承SuperEntity类，需要有@RelationshipEntity注解，注解参数type值为关系类型。

节点和关系都有一个ooid做为唯一id，此id做为图库与文档库同一份数据的关联。

repository为图库的数据层接口，需继承SuperRepository接口，需要有@DBRepository注解。无需写实现。

subentity为节点或关系的子文档数据，数据存于文档库。

node节点开发说明：

节点的关系需要有@Relationship注解，type参数为关系类型，direction参数为关系指向方向，可选参数有（***INCOMING进，OUTGOING出，UNDIRECTED无向***）

若节点的数据不需要存在图库中，只需要存在文档库里，则加@NeoTransient和@Transient注解。

关系开发说明：

关系的开始节点需要有@StartNode注解，结束节点需要有@EndNode注解，若关系的数据不需要存在图库中，只需要存在文档库里，则加@NeoTransient和@Transient注解。

repository开发说明：

repository定义的接口有两种注解：@NeoQuery和@DocQuery，@NeoQuery用于图库的搜索，@DocQuery用于文档库的搜索。

@NeoQuery的value值为cypher搜索语句，useDoc值默认为true，若为false，则不对存在文档库的字段进行补充搜索。

@DocQuery的value值为文档库搜索语句；projection值为返回过滤条件；sort为排序条件；skip为跳过多少条记录；limit为返回条数限制；useNeo值默认为true，若为false，则不对存在图库的字段进行补充搜索；writeQuery默认为false，为true时则表示此语句是个文档库写语句，当为写语句时，对图库的搜索无效；aggregate为对文档库的聚合搜索。

注：文档库搜索一般为json格式。

注：图库搜索语句以及文档图搜索语句皆可使用freemarker进行模板字符串替换。

### 实例

node实例：

@NodeEntity

**public** **class** UserNode **extends** SuperEntity {

@Relationship(type = RelationShipConst.***ACT***)

**private** List<RoleNode> roles;

@NeoTransient

@Transient

**private** Date createTime;

@NeoTransient

@Transient

**private** List<Notify> notifies;

**public** List<Notify> getNotifies() {

**return** notifies;

}

**public** **void** setNotifies(List<Notify> notifies) {

**this**.notifies = notifies;

}

**public** Date getCreateTime() {

**return** createTime;

}

**public** **void** setCreateTime(Date createTime) {

**this**.createTime = createTime;

}

**public** List<RoleNode> getRoles() {

**return** roles;

}

**public** **void** setRoles(List<RoleNode> roles) {

**this**.roles = roles;

}

}

relation实例：

@RelationshipEntity(type = RelationShipConst.***FRIEND***)

**public** **class** FriendRelation **extends** SuperEntity {

@StartNode

**private** RoleNode role1;

@EndNode

**private** RoleNode role2;

@NeoTransient

@Transient

**private** List<Message> messages;

@NeoTransient

@Transient

**private** Date createTime;

**public** Date getCreateTime() {

**return** createTime;

}

**public** **void** setCreateTime(Date createTime) {

**this**.createTime = createTime;

}

**public** List<Message> getMessages() {

**return** messages;

}

**public** **void** setMessages(List<Message> messages) {

**this**.messages = messages;

}

**public** RoleNode getRole1() {

**return** role1;

}

**public** **void** setRole1(RoleNode role1) {

**this**.role1 = role1;

}

**public** RoleNode getRole2() {

**return** role2;

}

**public** **void** setRole2(RoleNode role2) {

**this**.role2 = role2;

}

}

Repository实例：

@DBRepository

**public** **interface** UserRepository **extends** SuperRepository<UserNode> {

@NeoQuery(value = "MATCH (n:RoleNode { ooid: ${roleId} }) <-[:" + RelationShipConst.***ACT***

+ "]-(m:UserNode) RETURN m", useDoc = **false**)

**public** UserNode getRoleUser(Long roleId);

@DocQuery(value = "{'ooid':${userId}}", projection = "{'\_id':0,notifies:{$elemMatch:{read:false}}}", useNeo = **false**)

**public** UserNode getAllUnReadNotifies(Long userId);

@DocQuery(value = "{'ooid':${userId}}", projection = "{'\_id':0,notifies:{$elemMatch:{notifyId:${notifyId}}}}", useNeo = **false**)

**public** UserNode getNotifiy(Long userId, Long notifyId);

@DocQuery(aggregate = { "{$match:{ooid:${userId}}}", "{$unwind:'$notifies'}", "{$match:{'notifies.read':false}}",

"{$limit:${showNO}}", "{$group:{'\_id': '$\_id',notifies:{$push: '$notifies'}}}" }, useNeo = **false**)

**public** UserNode getUnReadNotifies(Long userId, Integer showNO);

@DocQuery(value = "{'ooid':${userId}}", projection = "{$push:{notifies:${notify}}}", writeQuery = **true**)

**public** **void** addNotify(Long roleId, @SubDoc Notify notify);

@DocQuery(value = "{'ooid':${userId},'notifies.notifyId':${notifyId}}", projection = "{$set:{notifies.$.read:true,notifies.$.readTime:${readTime}}}", writeQuery = **true**)

**public** **void** readNotify(Long userId, Long notifyId, Date readTime);

@DocQuery(value = "{'ooid':${userId},'notifies.notifyId':${notifyId}}", projection = "{$set:{notifies.$.callbackValue:${callbackValue}}}", writeQuery = **true**)

**public** **void** setNotifyCallbackValue(Long userId, Long notifyId, String callbackValue);

}

Repository调用：

@Inject

**private** Provider<UserRepository> userRepository;

UserNode roleUser = userRepository.get().getRoleUser(message.getToId());

## 定时器开发



### 说明

定时器任务类说明：

定时器任务类需实现JobTask接口，添加@Job注解，@Job注解可以直接指定数据库连接类型。

在job.properties中配置定时器启动时间规则。

ParkingRewardJob.repeat=1000

ParkingRewardStatisticsJob.cron=50 0 0 1 \* ?

ParkingRewardJob.once=1

key中.之前的值为job类名，.之后的为启动时间类型

repeat为多长时间重复启动一次，单位为秒

cron为按照cron表达式方式启动

once为只启动一次，可用于测试

**实例**

定时器任务类例子：

@Job(DBConnectionType.***WEAK\_WRITE***)

**public** **class** AdvStatisticsJob **implements** JobTask {

@Inject

AdvStatisticsService advStatisticsService;

**public** **void** excute() {

**try** {

advStatisticsService.excute(1L);

} **catch** (Exception e) {

e.printStackTrace();

}

}

}

CRON表达式    含义   
"0 0 12 \* \* ?"    每天中午十二点触发   
"0 15 10 ? \* \*"    每天早上10：15触发   
"0 15 10 \* \* ?"    每天早上10：15触发   
"0 15 10 \* \* ? \*"    每天早上10：15触发   
"0 15 10 \* \* ? 2005"    2005年的每天早上10：15触发   
"0 \* 14 \* \* ?"    每天从下午2点开始到2点59分每分钟一次触发   
"0 0/5 14 \* \* ?"    每天从下午2点开始到2：55分结束每5分钟一次触发   
"0 0/5 14,18 \* \* ?"    每天的下午2点至2：55和6点至6点55分两个时间段内每5分钟一次触发   
"0 0-5 14 \* \* ?"    每天14:00至14:05每分钟一次触发   
"0 10,44 14 ? 3 WED"    三月的每周三的14：10和14：44触发   
"0 15 10 ? \* MON-FRI"    每个周一、周二、周三、周四、周五的10：15触发

## soa开发



### 说明

service包说明：

service 存放接口

bean 存放bean

接口中的方法可添加注解@DBConnect(DBConnectionType.***WEAK\_WRITE***)来表示是否开启事务及事务类型，默认开启空事务。

接口中的方法可添加注解@DubboMethod来控制dubbo的method具体配置，包括异步请求等，具体使用方式可参照[<dubbo:method/>](http://dubbo.io/User+Guide-zh.htm#UserGuide-zh-%253Cdubbo%253Amethod%252F%253E)

接口中的方法请求参数字段可添加注解@DubboArgument使用方式可参照[<dubbo:argument/>](http://dubbo.io/User+Guide-zh.htm#UserGuide-zh-%253Cdubbo%253Aargumentn%252F%253E)

接口类可添加@DubboInterface注解来做指定服务器的推送。clusterType = ClusterTypeEnum.DIRECT（默认为***FAILBACK***）, clusterTarget = "websocket"（指定推送的服务器集群类型），若指定服务器推送，则接口方法的第一个参数应为string类型，为指定推送服务器的host。

消费端如果不是action和job，消费类需要添加@Named注解

提供端实现类需添加@Service注解，该注解为dubbo注解。

### 实例

接口实例：

**public** **interface** TestService {

@DBConnect(DBConnectionType.***WEAK\_WRITE***)

**public** String call(String a);

}

@DubboInterface(clusterType = ClusterTypeEnum.***DIRECT***, clusterTarget = "websocket")

**public** **interface** ChartPushService {

@DubboMethod(async = **true**, isReturn = **false**)

**public** **void** pushMessage(String target, Message message);

}

提供端实例：

@Service(version="1.0.1")

**public** **class** TestServiceImpl **implements** TestService {

@Override

**public** String call(String a) {

System.***out***.println("ok!");

**return** "hello";

}

}

消费端实例：

@Named

@Singleton

**public** **class** Test {

@Reference(version="1.0.1")

**private** TestService testService;

**public** **void** call(){

System.***out***.println(testService.call("a"));

}

}

action的消费端实例见action的实例

## 消息队列开发



### 实例与说明

消息队列分***SYSTEM***, ***MESSAGE***, ***CHART***, ***HANDLE*** 四大类型。

对应用途分别为：

***SYSTEM*** // 系统处理消息，不发往客户端

***MESSAGE*** // 系统消息，发往客户端显示

***CHART*** // 用户聊天消息

***HANDLE***// 需客户端处理的消息

具体话题再根据使用场景进行细分。

消息发送端实例：

@Inject

@MqProCon

MQSender mQSender;

Message message = **new** Message();

message.setToType(toType);

message.setTo(toId);

message.setTimestamp(System.*currentTimeMillis*());

message.setContent(messageType);

mQSender.send(Topic.***SYSTEM***, "authorityChange", message);

注解@MqProCon为生产者消费者模式

注解@MqPubsub为广播者监听者模式

注解@MqSafeProCon为事务安全的生产者消费者模式

监听者实例：

@MqListener(mqMode=MqMode.ProCon,topicType=Topic.CHART,topicName="test") **public** **class** SMSCheckerListener **implements** MQReceiver {

**private** **static** Logger *logger* = LoggerFactory.*getLogger*(SMSCheckerListener.**class**);

@Override

**public** **void** listen(Message message) {

*logger*.info("+++++++++++++++++MessageContent:" + message.getContent());

//具体逻辑

}

}

注解mqMode值设置消息模式：

PubSub//广播者监听者模式

ProCon//生产者消费者模式

SafeProCon//事务安全的生产者消费者模式

topicType设置话题类型

topicName设置具体话题

## 框架提供的部分方法说明



### http连接及回调

HttpConnection httpConnection = ApplicationContext.*CTX*.getHttpConnectionFactory()

.getHttpClient(HttpConnectionEnum.***MULTION***);

获取http请求连接，若线程为异步线程，则使用该httpConnection发送get请求时，将使用异步发送，如果请求地址有在sys\_baseconf.properties的pipelineOriginList里配置过，则发送带回调的get请求将使用pipeline异步发送。

注：同步方法及异步post方法不支持回调方式请求。

http请求的异步回调方式样例：

@Transaction(DBConnectionType.***WEAK\_WRITE***)

**public** **class** CallBackTest **implements** HttpCallBack {

@Override

**public** **void** completed(HttpResponseEntity httpResponseEntity) {

}

@Override

**public** **void** failed(Exception ex) {

}

@Override

**public** **void** cancelled() {

}

}

调用方式：

HttpConnection httpConnection = ApplicationContext.*CTX*.getHttpConnectionFactory()

.getHttpClient(HttpConnectionEnum.***MULTION***);

httpConnection.executeGet("http://www.xxx.com", **null**, **new** CallBackTest());

## 白盒测试开发



### 说明

fwwbt.properties配置说明：

xxxUrl=http://127.0.0.1:8080/xxx/cip

scanPath=com.xxx.test.job

encryptkey=asdfghjkqwertyui

如有其他版本，可自行添加

scanPath为测试任务的包扫描路径，建议放在com.xxx.test.job里。

encryptkey为全局秘钥。

启动方法在net.jahhan.test. StartTest

一次测试任务可包含多次测试请求

### 实例

测试任务例子：

@JobAnnocation(test = **true**)

**public** **class** CouponsUseJob **implements** Job {

**private** **final** Logger logger = LoggerFactory.*getLogger*(CouponsUseJob.**class**);

**public** **void** run() {

ApplicationContext.*CTX*.getThreadLocalManager().openThreadLocal(

**new** InvocationContext());//此句为必须放在第一句

**try** {

Map<String, Object> loginResult=(**new** Login()).request("12345674567", "闽A 110");

} **catch** (Exception e) {

logger.error("job error:", e);

}

}

}

其中注解中的test为true的时候，测试任务才会运行

com.xxx.test.request为放单次请求的包

请求例子如下：

**public** **class** Login **extends** RequestSender {

**public** Login() {

setDescription("登陆接口");

setRequestSecrityType(CryptEnum.***LOGIN***);

setResponseSecrityType(CryptEnum.***LOGIN***);

setSign(CryptEnum.***SIGN***);

}

**public** Map<String, Object> request(String telephone, String car\_no) {

Map<String, Object> contentMap = **new** HashMap<>();

contentMap.put("telephone", telephone);

contentMap.put("car\_no", car\_no);

**return** actRequestSend("merchant", "fw\_test\_login", 4, "1.2.0",

contentMap);

}

}

setDescription("登陆接口");//接口描述

setRequestSecrityType(CryptEnum.***LOGIN***); //请求加密模式

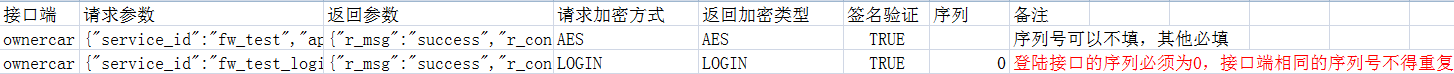
setResponseSecrityType(CryptEnum.***LOGIN***); //响应加密模式

setSign(CryptEnum.***SIGN***); //签名校验

### excel配置方式测试

com.xxx.test.job.ExcelActTestJob方法，可读取act\_test.xls配置文件进行接口测试。

配置实例如下：



# 部署说明



## tomcat部署附加说明

使用框架需要在tomcat的lib包添加下面4个jar包：

commons-pool2-2.3.jar

jedis-2.7.2.jar

base-session-1.0.0-SNAPSHOT.jar

tomcat-juli-8.0.28.jar

在tomcat的conf路径里的content.xml里添加如下配置：

<Valve className=*"net.jahhan.session.RedisSessionHandlerValve"* />

<Manager className=*"net.jahhan.session.RedisSessionManager"*

host=*"192.168.1.158"* port=*"6372"* database=*"0"* password="*Passw0rd*" maxInactiveInterval=*"60"* />

server.xml里修改cookieid名称：

<Context docBase=*"deploy-managerAction"* path=*"/manager"* reloadable=*"true"* source=*"org.eclipse.jst.jee.server:FrameWorkManagerAction"* sessionCookieName=*"XXXID"*/></Host>

maven打包为war文件。

部署到服务器需根据情况修改相应配置文件。

## redis启动附加说明

redis启动后需加入如下脚本：

|  |  |
| --- | --- |
| 用途 | 加载脚本 |
| 如果map表为空，则设置一个键值对 | SCRIPT LOAD "if redis.call('hlen',KEYS[1])==0 then return redis.call('hset',KEYS[1],KEYS[2],KEYS[3]) else return 0 end" |
| 如果值相等，则删除数据 | SCRIPT LOAD "if redis.call('get',KEYS[1])==KEYS[2] then return redis.call('del',KEYS[1]) else return 0 end" |
| 如果list为空，则设置一个值 | SCRIPT LOAD "if redis.call('llen',KEYS[1])==0 then return redis.call('lpush',KEYS[1],KEYS[2]) else return 0 end" |
| 如果旧值相等，则设置一个新值 | SCRIPT LOAD "if redis.call('get',KEYS[1])==KEYS[2] then return redis.call('set',KEYS[1],KEYS[3]) else return 0 end" |
| 返回列表所有值 | SCRIPT LOAD "if redis.call('LLEN',KEYS[1])>0 then return redis.call('LRANGE',KEYS[1],0,redis.call('LLEN',KEYS[1])-1) else return nil end" |
| 如果健的值大于给定值，则减少 | SCRIPT LOAD "local tmp=tonumber(redis.call('GET',KEYS[1])) if tmp>tonumber(KEYS[2]) then return redis.call('DECRBY',KEYS[1],KEYS[3]) else return tmp end" |
| 如果健的值小于给定值，则增加 | SCRIPT LOAD "local tmp=tonumber(redis.call('GET',KEYS[1])) if tmp<tonumber(KEYS[2]) then return redis.call('INCRBY',KEYS[1],KEYS[3]) else return tmp end" |

# 系统接口说明



## fw\_getDomainIp获取服务ip

调用方式：

{"service\_id":" fw\_getDomainIp","app\_type":4,"ver\_name":"1.2.0" ,"content": {" domainName": " ownercar"}}

参数说明：

domainName要获取的ip名称

## fw\_consumeGetMessage获取接口响应时间信息

调用方式：

{"service\_id":"fw\_consumeGetMessage","app\_type":4,"ver\_name":"1.2.0" ,"content": {"day": 20151028}}

参数说明：

day：日期

act：接口名称

statistics：统计天数

以上参数选传一个即可

# 配置及参数说明



## 系统错误码

|  |  |
| --- | --- |
| 错误码说明 | 错误码 |
| 成功，返回跳转页面 | -1 |
| 成功 | 0 |
| 未知错误 | 999 |
| 系统启动错误 | 998 |
| 编码时错误 | 997 |
| 业务级错误 | 996 |
| Session无效或者过期 | 900 |
| Session验证不通过 | 901 |
| 文件上传失败 | 902 |
| 数据库错误 | 903 |
| 服务器维护中 | 904 |
| Redis错误 | 905 |
| 参数错误 | 906 |
| 权限不足 | 908 |
| 页面不存在 | 909 |
| 接口已经失效或者不存在 | 910 |
| 获取认证KEY失败 | 911 |
| 已连接同类型websocket | 912 |
| 用户已登陆 | 913 |
| 系统配置错误 | 914 |
| 报文格式不正确 | 915 |
| 报文为空 | 916 |
| 加密失败 | 917 |
| 解密失败 | 918 |
| 签名验证失败 | 919 |
| 响应失败 | 920 |
| 快速响应失败 | 921 |
| 未支持错误 | 922 |

## 配置文件说明



### sys\_baseconf.properties

|  |  |
| --- | --- |
| Key值 | 说明 |
| uploadDir | 文件上传后缓存位置 |
| uploadURL | 文件上传后的url前缀 |
| jdbcFileName | 数据库连接文件名，默认为“jdbc” |
| recordTimeConsume | 是否开启接口耗时记录 |
| encryptkey.common | 全局秘钥 |
| companyName | 公司名，用于业务层识别 |
| htmlFilterInuse | 是否开启html过滤 |
| appType | 端类型 |
| noAuthorityUrl | 无权限返回页面，默认为“index.html” |
| sessionStrategy | session多端登陆策略，可选值范围：TICK（重登陆踢出策略）REJECT（重登陆拒绝策略）MULTI（允许重登陆），默认为MULTI |
| websocket.pathaccept | websocket接受的路径，用“，”隔开 |
| mq.actualize | 消息队列实现方式，目前只有redis方式，默认为redis |
| isDebug | 是否debug模式，默认为false |
| actSave | 是否保存接口（把接口信息存入t\_sys\_act表，接口信息变化时开启一次即可） |
| asyncTimeOut | 异步超时，默认为90000 |
| version | 业务代码对应框架版本，默认为“1.2” |
| authorityInuse | 是否开启权限管理，默认为false |
| allowAllOrigin | 是否跨域，默认为false |
| pipelineOriginList | 使用pipeline请求的域名列表，用“，”隔开 |
| cookieName | Cookie的名称 |
| cookiePath | Cookie的路径 |

### system.properties

system.properties里的内容在启动完成后将写入java运行参数中

|  |  |
| --- | --- |
| Key值 | 说明 |
| application.web | 运行实例的服务器地址，填内网ip |
| application.name | 运行实例的类别名称 |
| server.name | 运行实例的名称，用于同一台机子中同类别的运行实例的区分 |

### redis.properties

目前需开启5个库，库名分别是：common，cache，session，seq，mq

|  |  |
| --- | --- |
| 库名 | 说明 |
| common | 通用库 |
| cache | 数据库缓存库 |
| session | Session库 |
| seq | 数据库主键序列库 |
| mq | 消息队列库 |

每个库中都需配置4个参数

|  |  |
| --- | --- |
| 参数名 | 说明 |
| host | Redis的host ip地址 |
| port | 端口号 |
| database | 库号 |
| password | 密码 |

配置例子：

common.host=192.168.1.158

common.port=6370

common.database=0

common.password=Passw0rd

### domainIp.properties

域名转ip配置文件

### dfs\_client.conf

fastdfs配置文件

配置例子：

#seconds

connect\_timeout = 30

network\_timeout = 60

charset = ISO8859-1

group=group1

tracker\_server = 10.247.53.130:22122

# not used indeed

http.tracker\_http\_port = 80

http.anti\_steal\_token = no

http.secret\_key = FastDFS1234567890

tracker\_server为fastdfs的上传地址

### dubbo.properties

dubbo的配置文件

dubbo.annotation.package=net.jahhan.manager.action

为dubbo实现类及消费类的包名

dubbo.container=guice

为开启框架对dubbo的guice支持

dubbo.registry.address=zk://192.168.1.163:4180

dubbo.registry.client=curator2

为框架对zk连接方式的升级支持写法

dubbo.service.filter=default,dbTransaction

为框架对提供端的扩展，消费端不用配置

其他配置说明见dubbo的官网说明

<http://dubbo.io/User+Guide-zh.htm>

### jdbc.properties

数据库连接的配置文件

配置例子：

###############

jdbc.dbtype=mysql

jdbc.driver=com.mysql.jdbc.Driver

jdbc.checkoutTimeout=2000

jdbc.maxStatements=1000

jdbc.maxStatementsPerConnection=50

jdbc.maxIdleTime=60

jdbc.unreturnedConnectionTimeout=180

jdbc.initialPoolSize=20

jdbc.poolsizemin=10

jdbc.poolsizemax=100

jdbc.acquireIncrement=10

#########

read.jdbc.checkoutTimeout=2000

read.jdbc.maxStatements=1000

read.jdbc.maxStatementsPerConnection=50

read.jdbc.maxIdleTime=60

read.jdbc.unreturnedConnectionTimeout=180

read.jdbc.initialPoolSize=20

read.jdbc.poolsizemin=10

read.jdbc.poolsizemax=50

read.jdbc.acquireIncrement=10

#########

jdbc.url=jdbc:mysql://192.168.1.159:3306/qcw\_new?useUnicode=true&characterEncoding=utf-8

jdbc.username=

jdbc.password=

#########

read0.jdbc.url=jdbc:mysql://192.168.1.159:3306/qcw\_new?useUnicode=true&characterEncoding=utf-8

read0.jdbc.username=

read0.jdbc.password=

配置用#########隔开为4部分，第一部分为写库的配置，第二部分为读库配置，第三部分为写库的连接配置，第四部分为读库的连接配置，若有多个读库，可从read0,read1…依次配置。