ISC微服务

后台框架设计说明书

版本：1.4

编制人：王孝坤

目录

[ISC微服务 1](#_Toc6841524)

[后台框架设计说明书 1](#_Toc6841525)

[一、 微服务框架背景介绍 4](#_Toc6841526)

[二、 框架总设计说明 4](#_Toc6841527)

[1. 包图设计说明 4](#_Toc6841528)

[2. 类图设计说明 6](#_Toc6841529)

[三、 代码生成器插件 8](#_Toc6841530)

[1. 代码生成器使用说明 8](#_Toc6841531)

[2. Maven项目工程代码生成 8](#_Toc6841532)

[3. 框架代码生成 10](#_Toc6841533)

[4. 自定义代码的扩展方式说明 11](#_Toc6841534)

[四、 框架基本操作方法说明 12](#_Toc6841535)

[1. BasicController基础方法： 12](#_Toc6841536)

[2. BaseBiz基础方法 12](#_Toc6841537)

[3. BaseMapper接口定义 13](#_Toc6841538)

[4. \*\*\*-mapper.xml节点说明 13](#_Toc6841539)

[5. 精确查询和高级查询的说明 13](#_Toc6841540)

[五、 WebConfig使用说明 14](#_Toc6841541)

[1. WebMVC配置的应用场景说明 14](#_Toc6841542)

[2. 自定义拦截器逻辑 14](#_Toc6841543)

[3. 自定义消息转换器 16](#_Toc6841544)

[六、 动态数据源切换设计 16](#_Toc6841545)

[1. 动态数据源配置 16](#_Toc6841546)

[2. 主从库配置说明 17](#_Toc6841547)

[3. 多业务库配置说明 18](#_Toc6841548)

[4. 移植库配置说明 19](#_Toc6841549)

[5. 数据源切换原理和使用 19](#_Toc6841550)

[6. 注意事项说明 21](#_Toc6841551)

[7. 事务说明 22](#_Toc6841552)

[七、 统一返回值格式 24](#_Toc6841553)

[1. 统一返回值类型定义 24](#_Toc6841554)

[2. 统一返回格式的拦截封装 25](#_Toc6841555)

[3. 统一返回格式的应用 25](#_Toc6841556)

[八、 统一异常捕捉处理 26](#_Toc6841557)

[1. 实现原理 26](#_Toc6841558)

[2. 自定义异常处理 26](#_Toc6841559)

[3. 系统异常处理 27](#_Toc6841560)

[4. 调用方说明 27](#_Toc6841561)

[九、 断路器使用说明 27](#_Toc6841562)

[1. 断路器基础配置 27](#_Toc6841563)

[2. 断路器使用 27](#_Toc6841564)

[3. 断路器聚合监控 29](#_Toc6841565)

[十、 统一日志方式 31](#_Toc6841566)

[1. 文件日志 31](#_Toc6841567)

[2. 数据库服务日志 32](#_Toc6841568)

[十一、 时区转换和日期格式转换 34](#_Toc6841569)

[1. 应用场景说明 34](#_Toc6841570)

[2. 框架解决方案 34](#_Toc6841571)

[十二、 其他辅助开发项 35](#_Toc6841572)

[1. Redis初始化和使用 35](#_Toc6841573)

[2. Swagger配置和使用 36](#_Toc6841574)

[3. 关于springboot配置文件格式 37](#_Toc6841575)

[4. Controller和RestController的区别 38](#_Toc6841576)

[5. 配置文件配置项获取方式 38](#_Toc6841577)

[6. 初始化bean注入方式 38](#_Toc6841578)

[7. 关于项目划分jar包的思考 38](#_Toc6841579)

[十三、 版本修订历史记录 39](#_Toc6841580)

# 微服务框架背景介绍

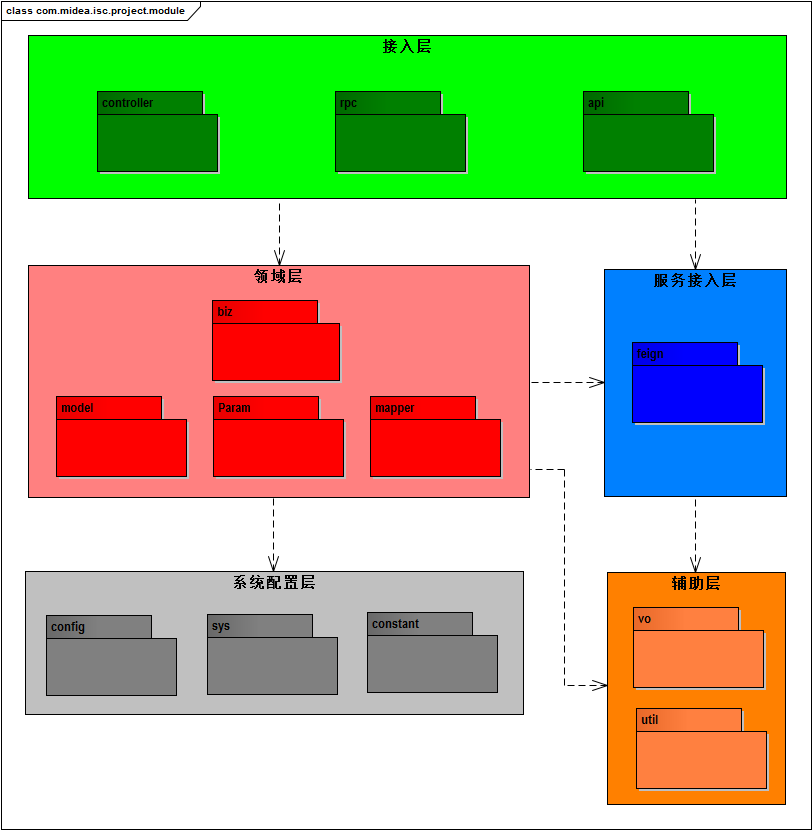
本文是基于ISC微服务架构设计基础上构建的ISC后台代码框架设计的使用说明书，使用

Springboot 2.0.6+springcloud Finchley.SR2框架并集成微服务架构中的配置中心和权限服务形成的一套快速开发框架。

PS:建议阅读本文档前先阅读ISC微服务架构设计说明书

# 框架总设计说明

## 包图设计说明



* 1. 包命名规范

包命名的统一前缀：com.midea.isc.project.module

作为微服务框架，一个项目可以分成多个服务模块，每个服务模块的工程是互相独立而业务上又是相关联的，固规定包名格式如上，例如销售项目(sale)划分仓储模块(wms)和订单模块(oms)则会独立生成两个项目工程，分别包名前缀为

com.midea.isc.sale.wms和com.midea.isc.sale.oms。

基于包名前缀再添加上述的约定包名形成具体包名，如：

com.midea.isc.sale.wms.controller和com.midea.isc.sale.wms.rpc等等。

当需要添加不在约束范围内的包名时，只要保持前缀一致即可,如：

com.midea.isc.sale.wms.siebel。

* 1. 包用途说明

**接入层：**

controller包：提供前端页面的后台请求实现，需要用户登录才能访问且由框架自动初始化入参的用户信息，服务间无法互调；

rpc包：提供服务间调用的请求实现，需要授权请求服务端后才能正常访问，swagger默认扫描的包；

api包：提供外围系统调用的请求实现，需要指定网关发送的请求才能访问，做不同的安全策略控制；

**领域层：**

biz包：主要完成业务逻辑实现

model包：dto用于在服务器与客户端之间或服务器与服务器之间进行数据传递

param包：dto用于入参的封装，继承自model，扩展了字段比较符的封装

mapper包：俗称dao层，用于定义数据仓储操作接口

**服务接入层：**

feign包：统一管理订阅其他服务提供的接口

**系统配置层：**

config包：用户初始化bean和系统基础配置，如数据库配置，WebMVC配置，spring bean注入等等

sys包：存放系统配置类、系统帮助类等

constant包：存放系统常量、全局静态变量、枚举值等

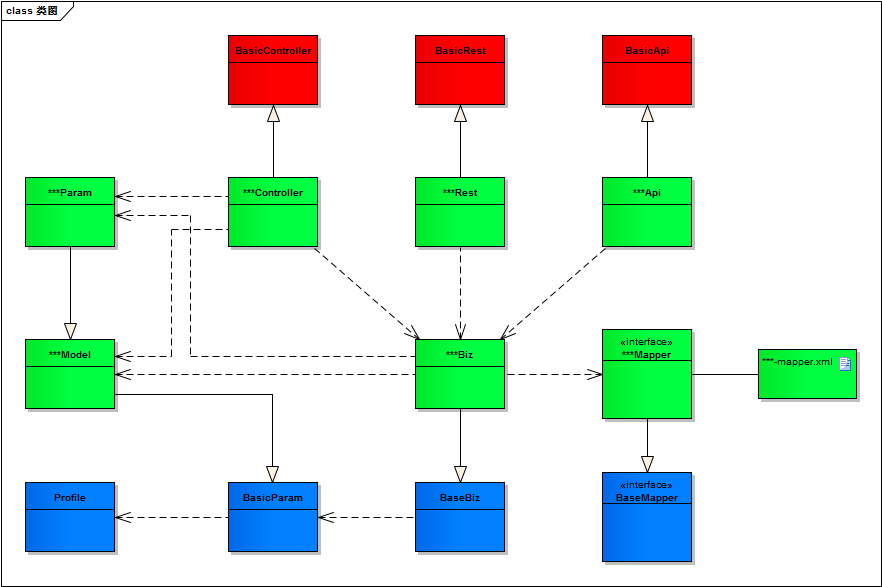
**辅助层：**

vo包：dto，用于扩展model和param之外的pojo类

util包：存放工具类的包

PS：代码生成器只会生成controller、biz、model、param、mapper等包，其他包根据项目实际需要添加

## 类图设计说明



1. 类图说明

红色类为common-web包提供的web基类

蓝色类为common包提供的通用基类和接口

绿色类为业务项目具体类，使用\*\*\*泛指具体类名

common-web包独立于common包主要是为剥离webmvc相关类的配置和基类，由于spring web开发存在多样性，例如JSP、springMVC、webfulx等，且这几个web框架存在互斥，同时存在引发冲突，固需要将web部分进行独立。

1. 基类和基接口用途说明
   * 1. BasicController：Controller基类，提供基础单表的增删改查服务的封装，支持高级搜索和多语言，需要用户登录才可访问
     2. BasicRest：Service基类，即继承此类代表提供的方法都是用于给其他服务间调用的，需要相应服务授权才可访问
     3. BasicApi：Api基类，即继承此类代表提供的方法都是用于给外围系统调用的，需要经过相应的token校验和安全策略管理才可访问
     4. Profile：用户信息POJO类，已登录用户信息的传递介质，包括以下信息：

登录名、全名、国家、时区、登录语言、默认币种、当前账套、所属事业部、所属组织、登录信息(令牌、应用、支持语言、过期时间等)、业务系统扩展信息(此字段为json对象，存储结构由业务系统自己设计，不建议存储大量数据)

* + 1. BasicParam：入参基类，引入Profile用于获取登录用户信息。包含分页属性\_\_page, \_\_pagesize，针对MySQL生成相应的limitClause语句。包含排序属性orderFields，为hashmap类型，key为字段名，value为asc或desc
    2. BaseBiz：业务逻辑基类，泛型编程，具体逻辑类、入参、出参由泛型参数传入，提供单表的基础增删改查操作，并根据操作类型自动区分主从库操作。针对insert和update操作，提供setWho和setWhoForUpdateInfo操作，用于自动赋值表设计的固定字段创建人创建时间，最后更新人更新时间等，子表可直接调用
    3. BaseMppaer：仓储操作基接口，定义基础的增删改查操作接口

1. 类关系说明
2. BasicParam引用Profile存放登录用户，作为入参的基础部分
3. \*\*\*Model继承自BasicParam，\*\*\*Param继承自\*\*\*Model

此设计主要考虑减少冗余代码，降低sonar扫描产生的坏味道，model作为出参可视为入参的一个子集固param继承model，而由于类的继承是单一继承，固入参基类BasicParam只能由\*\*\*Model继承

而这个设计导致另外一个问题，即出参和入参都包含了BasicParam，从前端角度来说出参包含BasicParam是多余的

解决方案是通过控制BasicBaram的序列化和反序列化，实现出参序列化到前端的时候不包含BasicParam，而入参反序列化到后端的时候包含BasicParam。

get方法添加@JsonIgnore标记忽略序列化，set方法添加@JsonProperty标记支持反序列化

1. \*\*\*Controller继承BasicController基类，同时通过引用\*\*\*Model、\*\*\*Param、\*\*\*Biz传入具体参数实现具体业务单表的增删改查服务。\*\*\*Rest和\*\*\*Api继承和引用关系类似，但不提供基础服务项
2. \*\*\*Biz继承BaseBiz基类，同时通过引用\*\*\*Model、\*\*\*Param、\*\*\*Mapper传入具体参数实现具体业务单表的增删改查操作

此处应该注意\*\*\*Mapper为引用关系非接口实现关系，mapper为dao层通过mybaits这个ORM框架实现数据库操作，\*\*\*Biz通过引用mapper实现数据库操作，而逻辑处理在这里不使用多余的接口编程，直接定义方法使用。

1. BaseBiz引用BasicParam是泛型编程的时候可通过入参P的父类来获取登录用户的信息
2. \*\*\*Mapper继承BaseMapper来包含代码生成器默认生成的基础数据库增删改查操作，\*\*\*Mapper提供的方法需要和\*\*\*-mapper.xml一一映射
3. 其他说明项
   1. 对于表设计的通用字段要求，这样BaseBiz的基础操作才能成功自动设置，不按照规范命名系统不会报错但需要自己赋值相关字段：

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名(Java) | 列名(数据库) | 数据类型 | 用途 |
| createdBy | created\_by | int | 创建人Id |
| createdByName | created\_by\_name | varchar(50) | 创建人登录名 |
| creationDate | creation\_date | datetime | 创建时间 |
| lastUpdateBy | last\_updated\_by | int | 最后更新人Id |
| lastUpdateByName | last\_updated\_by\_name | varchar(50) | 最后更新人登录名 |
| lastUpdateDate | last\_update\_date | datetime | 最后更新时间 |

* 1. createdBy和lastUpdateBy为老框架保留字段，存放common\_user表中的主键字段，新框架更偏向于存放登录用户名(即MIP账号)，所以添加扩展字段createdByName和lastUpdateByName来保存MIP账号。但是基于新表建议按如下定义表字段，框架代码会自动判断createdBy和lastUpdateBy的数据类型来进行赋值：

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名(Java) | 列名(数据库) | 数据类型 | 用途 |
| createdBy | created\_by | varchar(50) | 创建人登录名 |
| creationDate | creation\_date | datetime | 创建时间 |
| lastUpdateBy | last\_updated\_by | varchar(50) | 最后更新人登录名 |
| lastUpdateDate | last\_update\_date | datetime | 最后更新时间 |

* 1. BaseBiz包含setWho和setWhoForUpdateInfo两个方法，访问控制为protected，子类可访问，入参为继承自BasicParam且已初始化profile的\*\*\*Param才能正常赋值固定字段。setWho主要是新建数据时使用，setWhoForUpdateInfo主要是更新数据时使用，这两个方法在老框架中会同步设置创建时间和最后更新时间，但是这种做法不太合理，这样做需要确保所有节点服务器时区一致时间也一致，新框架通过sql语句获取数据库当前时间来设置，同时新框架将最后更新时间字段用作update时的乐观锁替代版本号的字段，所以在sql执行前不应该被改变，这个修改要求开发人员自写的insert和update的sql语句，要注意创建时间和更新时间的取值设置。

# 代码生成器插件

本章节只讲解代码生成器如何使用，而代码生成器的设计实现原理将在其他文档中讲解。

## 代码生成器使用说明

ISC微服务框架的代码生成器已经实现maven插件化开发，并默认配置到所有项目工程的pom文件中，目前支持maven工程生成和框架代码生成，后续考虑集成前端vue脚手架的生成。与以往的代码生成器相比，maven插件化实现了代码生成和maven项目工程结构的无缝集成，并约束了包和类的命名规范，同时在美的maven私服中进行版本管理和升级维护，确保各个项目可以即时获取最新版本，同样有助于开发者管理和发布新版本代码生成器。pom文件插件配置节点如下图所示：



## Maven项目工程代码生成

用于生成maven项目工程的脚手架和基础类，项目首次初始化联系

[xiaokun.wang@midea.com](mailto:xiaokun.wang@midea.com)进行初次生成。

* + 1. 配置项说明(configuration支持的子节点)

|  |  |  |  |
| --- | --- | --- | --- |
| 配置项 | 是否必填 | 默认值 | 说明 |
| project | 否 | 空字符串 | 项目名称 |
| application | 是 | admin | 应用名称也可理解为模块名称 |
| administrtor | 否 | administrtor | 应用管理员MIP账号 |
| basedir | 是 | 无 | 工程脚手架存放目录 |
| port | 是 | 无 | Springboot应用启动的端口 |
| configdir | 是 | 无 | Springcloud配置中心目录，即配置文件存放目录 |

* + 1. 运行maven工程代码生成

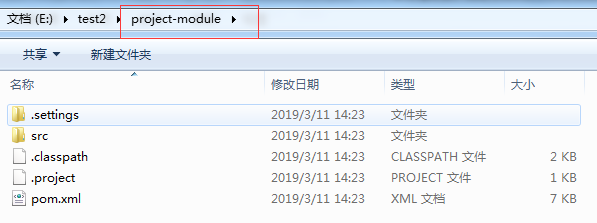
主要通过运行maven命令cg:project进行代码的生成，运行方式可选如下：

1. 在pom文件所在目录打开cmd窗口，然后输入cg:project
2. 通过IDE运行maven命令cg:project

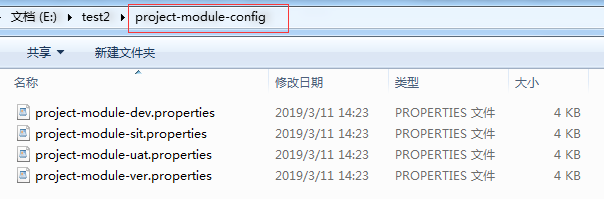
项目首次初始化由框架组负责生成，可使用main函数方式完成首次生成。

* + 1. 生成的脚手架目录和基础类介绍

1. 总共生成两个目录project-module和project-module-config(project和module会替换成具体项目名和模块名，参照配置项)，一个存放maven工程一个存放配置文件，如图所示：

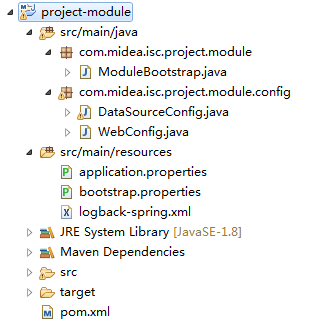


maven工程



配置文件

1. 其中配置文件需要由开发负责人上传到配置中心相应的目录(参照配置项说明的configdir)。
2. maven工程脚手架主要包括如下图所示的配置类和配置文件：



1. ModuleBootstrap：springboot启动类，添加了如下标记用于框架的初始化

|  |  |
| --- | --- |
| 标记名 | 用途 |
| @SpringBootApplication | Springboot启动类标记，exclude默认的数据库配置，采用自定义的数据库配置 |
| @EnableFeignClients | 扫描指定包，注入订阅服务的Bean |
| @EnableScheduling | 启动定时任务，框架初始用于定时获取访问权限信息 |
| @EnableTransactionManagement | 启用事务管理 |
| @EnableAuthClient | 初始化和启用微服务架构的权限管理 |
| @Import | 导入公共包的配置类，目前默认导入redis配置、服务日志配置、swagger配置、数据源切面代理配置、统一数据格式配置、统一异常处理配置，具体使用见后续章节详解 |
| @RemoteApplicationEventScan | 消息总线相关事件的包扫描 |

1. DataSourceConfig：数据源配置类，详情请阅读后续章节：动态数据源切换设计
2. WebConfig：WebMVC基础配置类，详情阅读后续章节：WebConfig使用说明
3. bootstrap和application两个properties文件为springboot的基础配置类，bootstrap先于application加载，bootstrap主要配置端口号、注册中心地址、配置中心信息
4. logback-spring.xml，日志基础配置
5. pom.xml，maven基础配置以及依赖项和插件初始化

## 框架代码生成

用于生成基表或视图增删改查基础操作的框架代码，每个通过代码生成器生成的maven工程已经进行了初始配置，并有相关注释进行说明

1. 配置项说明(configuration支持的子节点)

|  |  |  |  |
| --- | --- | --- | --- |
| 配置项 | 是否必填 | 默认值 | 说明 |
| project | 否 | 空字符串 | 项目名称 |
| application | 是 | admin | 应用名称也可理解为模块名称 |
| sourceFolderPath | 是 | 无 | 源码的存放目录，在pom文件中通过${basedir}获取当前目录 |
| driverName | 是 | 无 | 数据库驱动，目前仅支持mysql和oracle |
| dbUrl | 是 | 无 | 数据库连接URL |
| dbName | 是 | 无 | 数据库名 |
| dbUser | 是 | 无 | 数据库登录用户名 |
| dbPwd | 是 | 无 | 用户密码 |
| mode | 是 | new | 生成模式new/update,new模式生成所有相关类，update模式代表只更新model/param/mapper.xml |
| tableNames | 是 | 无 | 需要生成的表名数组 |

1. 运行框架代码生成

主要通过运行maven命令cg:scheme进行代码的生成，运行方式可选如下：

1. 在pom文件所在目录打开cmd窗口，然后输入cg: scheme
2. 通过IDE运行maven命令cg: scheme
3. 生成代码介绍

生成的代码请参照类图设计说明的讲解

## 自定义代码的扩展方式说明

代码生成器只会生成controller、biz、model、param、mapper等类以及mapper对应的xml文件，mode为new时每次重新生成都会覆盖现有文件，所以要考虑自定义代码被覆盖的问题：

1. 其中controller、biz和mapper多次生成结果都是一致的，不会因为增减改字段而变化，固建议自定义代码直接在其中添加，每次重新生成需要将mode修改为update防止自定义代码被覆盖。
2. 而model、param和mapper对应的xml文件，会因为增减改字段而变化，固对这些文件的自定义代码不能直接在文件中添加，model和param自定义代码通过继承的方式扩展，mapper对应的xml文件则另起一个文件扩展数据库操作，命名规范为\*\*\*Ex-mapper.xml，使用相同的命名空间即可对应相同的mapper类。

# 框架基本操作方法说明

## BasicController基础方法：

|  |  |
| --- | --- |
| 方法名 | 说明 |
| query | 框架提供的查询方法，仅支持精确查找，保留是为了兼容旧项目的后台代码 |
| count | 对应query的行数统计方法 |
| insert | 基表的单记录新增操作 |
| update | 基表的单记录更新操作 |
| updateByOtsLock | 带乐观锁的基表单记录更新操作，默认必须有last\_update\_date字段，没有该字段则跟update方法无差别 |
| delete | 基表的记录删除操作，支持高级搜索的方式批量删除数据 |
| saveBulk | 批量操作，可同时包括批量新增、更新和删除操作 |
| find | 基表或视图的高级查询方法，支持精确查找和高级查找，高级查找表示支持根据数据类型提供不同比较符的查找方式，设计开发阶段要考虑清楚查找的性能再使用 |
| total | 对应高级查询的行数统计方法 |

## BaseBiz基础方法

与BasicController基本对应

|  |  |
| --- | --- |
| 方法名 | 说明 |
| selectOne | 基表或视图的精确查找并返回单一记录 |
| selectList | 对应Controller的query方法 |
| selectCount | 对应selectList的行数统计方法 |
| insert | 对应Controller的insert方法 |
| insertBulk | 批量新增方法，暂不支持多语言表和Oracle的批量 |
| update | 对应Controller的update方法 |
| updateByOtsLock | 对应Controller的updateByOtsLock方法 |
| updateBulk | 批量更新方法，暂不支持多语言表的批量 |
| delete | 对应Controller的delete方法 |
| deleteBulk | 批量删除方法 |
| saveBulk | 批量方法的统一调用，对应Controller的saveBulk方法 |
| find | 对应Controller的find方法 |
| total | 对应Controller的total方法 |
| setWho | 通过反射和入参基类BasicParam中的profile自动设置创建人和更新人相关字段 |
| setWhoForUpdateInfo | 通过反射和入参基类BasicParam中的profile自动设置更新人相关字段 |

而针对多语言表时需重写如下方法：

|  |  |
| --- | --- |
| 方法名 | 说明 |
| insert | 先调用insert完成主表的插入，再调用insertTL完成多语言表的插入，并启用事务管理 |
| update | 先调用update完成主表的更新，再调用updateTL完成多语言表的更新，如果updateTL操作返回0条，则调用copySourceLang完成新语言的初始化再调用updateTL，并启用事务管理 |
| updateByOtsLock | 先调用updateByOtsLock完成主表的版本判断和更新，成功后再调用updateTL完成多语言表的更新，如果updateTL操作返回0条，则调用copySourceLang完成新语言的初始化再调用updateTL，并启用事务管理 |

## BaseMapper接口定义

与BaseBiz一一对应，而针对多语言表时接口需添加如下方法定义：

|  |  |
| --- | --- |
| 方法名 | 说明 |
| insertTL | 完成多语言表的插入 |
| updateTL | 完成多语言表的更新 |
| copySourceLang | 完成多语言记录的初始化新增 |

## \*\*\*-mapper.xml节点说明

|  |  |
| --- | --- |
| 节点名 | 说明 |
| resultMap | 实现数据库列名和实体属性的映射关系 |
| orderClause | 实现排序语句的生成，将HashMap<String,String>自动转换成  order by t.\*\*\* asc,t.\*\*\* desc格式 |
| defaultWhereClause | 老框架精确查找过滤条件的生成 |
| advancedWhereClause | 高级查找过滤条件的生成 |

其他节点语句与BaseMapper中方法一一对应。

## 精确查询和高级查询的说明

精确查询的入参不需要比较符，由于param继承自model固只需要给model相应的字段赋值即可实现精确查找

高级查询的入参需要比较符，param针对model中的字段进行比较符字段的扩展，对于非日期字段采用字段名+Cond的方式命名比较符

类继承关系请参照类图设计说明

对于日期字段采用字段名+From和字段名+To来进行区间查询

比较符号说明：

|  |  |  |
| --- | --- | --- |
| 比较符 | 全称 | 说明 |
| eq | equal | 等于 |
| ne | not equal | 不等于 |
| me | multi equal | 多选匹配 |
| cn | contain | 模糊查询 |
| bw | begin with | 开始于 |
| ew | end with | 结束于 |
| gt | greater than | 大于 |
| ge | greater or equal | 大于等于 |
| lt | less than | 小于 |
| le | less or equal | 小于等于 |
| bt | between | 之间 |

各个数据类型支持的比较符说明

|  |  |
| --- | --- |
| 数据类型 | 支持的比较符 |
| String | eq,ne,me,cn,bw,ew |
| Number | eq,ne,gt,ge,lt,le |
| Date/Datetime | bt |

PS：以上所有文件及其中的方法和节点等都是由代码生成器生成作为基础操作供业务直接方法调用

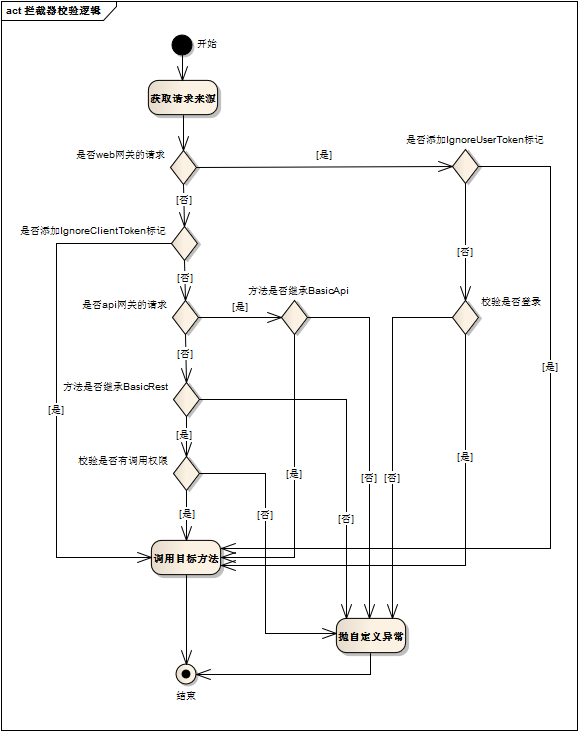
# WebConfig使用说明

## WebMVC配置的应用场景说明

配置类WebConfig实现WebMvcConfigurer 接口，WebMvcConfigurer配置类其实是Spring内部的一种配置方式，采用JavaBean的形式来代替传统的xml配置文件形式进行针对框架个性化定制。基于java-based方式的spring mvc配置。主要应用场景为：自定义拦截器配置、自定义资源映射、配置视图解析器、自定义消息转化器等。框架代码初始化目前主要使用自定义拦截器配置和自定义消息转换器配置。

## 自定义拦截器逻辑

框架默认添加ServiceAuthRestInterceptor拦截器，该拦截器位于common-web包中，在开发调试模式下(spring.cloud.config.profile:dev)拦截器只尝试获取登录用户信息如果成功则设置到当前线程本地变量中，不成功仅代表没有用户登录信息。集成环境下ServiceAuthRestInterceptor校验逻辑如图所示：



校验逻辑说明如下：

1. 识别是否来自web网关的请求，代表前端页面请求，是则看步骤b，否则看步骤c
2. 获取类或者方法是否添加IgnoreUserToken标记，是则忽略用户登录校验跳过拦截器，否则通过token+app获取登录用户的信息Profile进行权限校验，校验通过则调用具体方法，不通过则抛出自定义异常
3. 获取类或者方法是否添加IgnoreClientToken标记，是则忽略服务调用权限校验跳过拦截器，否则判断是否来自api网关的请求，是则看步骤d，否则看步骤e
4. 判断服务请求方法对应的基类是否继承BasicApi，是则通过校验，否则抛自定义异常。api网关代表外围系统的调用，此时我们需要限定外围系统可访问的服务来增强服务的安全，类继承说明请参照类图设计说明章节
5. 判断服务请求方法对应的基类是否继承BasicRest：

如果继承自BasicRest则判断服务请求的来源是否有权限访问目标服务，服务间访问的授权可在管理员平台配置，有权限访问才能校验通过调用具体方法，没有权限则抛自定义异常。

如果不是继承自BasicRest则抛自定义异常。

服务间的鉴权是用来防止外抛的服务招到恶意调用或者盗用，导致数据的泄密和资源的浪费等。类继承说明请参照类图设计说明章节

## 自定义消息转换器

框架默认添加自定义消息转换器MyMappingJackson2HttpMessageConverter，重写extendMessageConverters方法，仅添加一个自定义的HttpMessageConverter ，不覆盖MVC默认注册的多个HttpMessageConverter。

Post请求(标记添加method = RequestMethod.POST，入参添加@RequestBody)才会触发消息转换器，同时只有入参继承自BasicParam该消息转换器才会生效。

MyMappingJackson2HttpMessageConverter主要完成以下功能：

1. 通过自定义的ObjectMapper添加自定义的日期格式序列化和反序列化器，

默认日期序列化器统一格式为yyyy-MM-dd HH:mm:ss

默认日期反序列化器支持以下格式：

yyyy-MM-dd HH:mm:ss和yyyy-MM-ddTHH:mm:ss

同时也支持自定义的日期格式

1. 通过当前线程变量获取已登录用户的profile初始化到BasicParam中，方便业务逻辑处理语句和SQL语句获取用户信息
2. 通过分页信息\_\_page和\_\_pagesize初始化limitClause，用于支持MySQL分页语句的生成

# 动态数据源切换设计

ISC微服务框架目前支持多库配置、主从库配置、移植库配置，初步实现跨库操作，读写分离，数据环境迁移等功能。

## 动态数据源配置

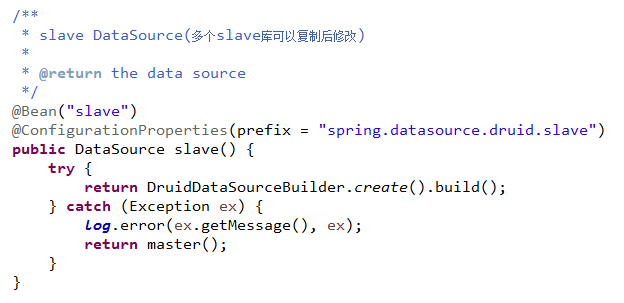
1. 取消springboot默认的数据库配置，只要在启动类的标记中添加如下配置即可：@SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })
2. 配置文件添加数据库相关配置，代码生成器已生成默认配置节点如下图所示：



1. DataSourceConfig配置类根据前缀读取不同数据库配置项形成动态数据源

## 主从库配置说明

1. 主库只能有一个如上图的主库配置章节
2. 从库可以有多个，可以通过复制上图从库配置章节然后将slave递增为slave1，slave2来添加从库配置
3. 修改DataSourceConfig来添加从库配置的读取，可复制默认slave配置如下图：



1. 修改@Bean(slave)递增为@Bean(slave1)，@Bean(slave2)，避免bean冲突，同时修改方法名为slave1(),slave2()
2. 修改prefix对应不同步骤b不同的配置节点的前缀
3. 最后修改动态数据源的方法，将新增的从库添加到数据源池中，如下图：



1. 注意key名的前缀DataSourceKey.slave.name()保持不变，后面数字继续递增
2. 从库的获取目前为简单的轮询机制

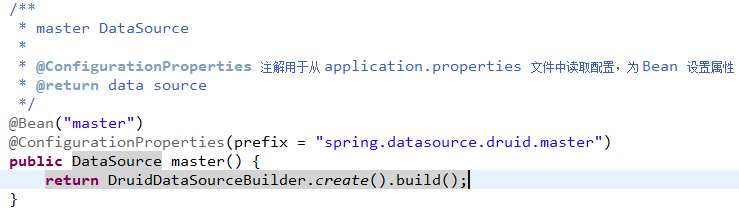
## 多业务库配置说明

考虑兼容旧框架系统涉及多业务库操作的问题，微服务框架也从框架设计层面支持多业务库，但从微服务架构设计思想的角度思考，不建议存在单个服务模块中出现多业务库的操作。本框架支持多库操作，但仅支持其中一个库的主从分离配置。多业务库配置说明如下：

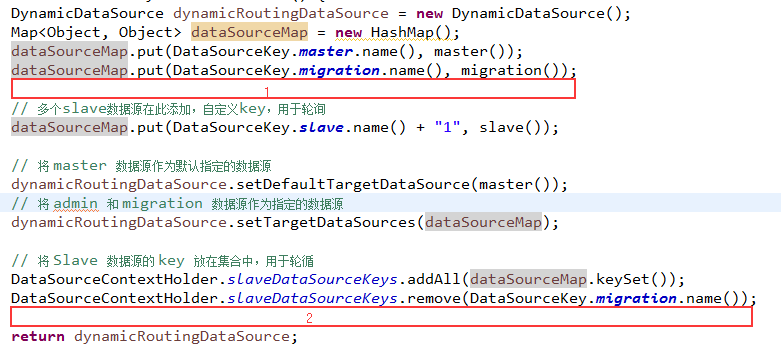
1. 复制配置文件中的主库配置章节，如下图：



1. 修改复制节点中的key名前缀中的master为相应业务库标记，例如：erp
2. 修改DataSourceConfig来添加业务库配置的读取，可复制master配置如下图：



1. 修改@Bean(master)为@Bean(erp)，修改方法名,例如erp();
2. 修改prefix对应不同步骤b不同的配置节点的前缀
3. 最后修改动态数据源的方法，将新增的业务库添加到数据源池中，如下图：



第一步添加业务库到数据源池中，例如：dataSourceMap.put(“erp”, erp());

第二步将业务库从从库中排除，例如：

DataSourceContextHolder.slaveDataSourceKeys.remove(“erp”);

## 移植库配置说明

本框架提供的移植库主要是基于环境数据迁移考虑，例如SIT配置数据迁移到UAT环境，UAT配置环境数据迁移到验证，以此类推。

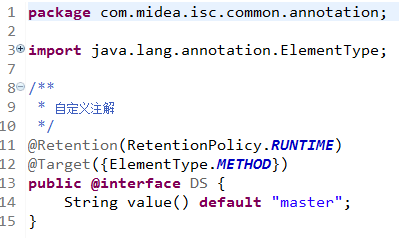
本框架仅支持单一移植库配置，不支持多个移植库的扩展，所以只需要修改配置文件中相关数据源的修改，其他都不用修改。如果不需要移植库则不配置即可。

## 数据源切换原理和使用

本章节首先简单介绍切库原理，然后介绍在实际代码中如何实现数据源的切换，包括主从切换、业务库切换、移植库切换，跨库操作使用说明以及框架基础操作默认使用的数据源。

切库原理：

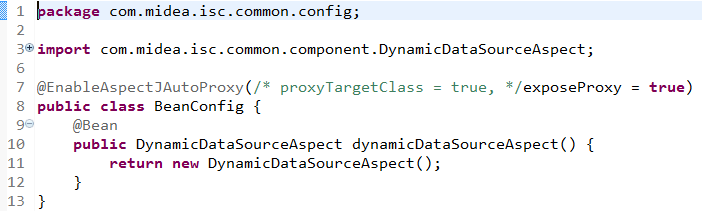
1. 切换基础原理：自定义注解 + AOP的方式实现数据源动态切换
2. Common包中已添加自定义注解DS，注意此注解只在Biz层使用，该注解的value为数据源的key名，默认为master：



1. 通过Spring AOP方式拦截添加了注解的方法，Common包已添加切面类的定义和注入，如下图所示：



切面类定义



切面类注入

1. DynamicDataSourceAspect类主要功能就是实现数据源的切换，根据拦截到的DS注解value值和@Before标记在方法执行前切入并设置当前线程的数据源，通过方法

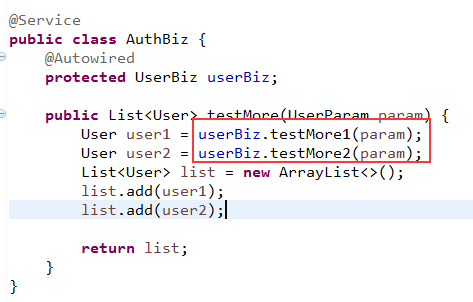
DataSourceContextHolder.setDB(dataSource)来切换数据源，value取值范围为mater,slave, migration以及自定义的业务库名例如erp，这里针对slave做特殊处理，slave不需要指定具体的slave名如slave1、slave2，获取从库数据源时通过方法DataSourceContextHolder.getSlaveDB()在从库池中轮询获取一个从库数据源。切面类还需要通过@After在方法执行完成后清空当前设置的数据源恢复默认值，不然将影响后续的执行语句。

使用说明：

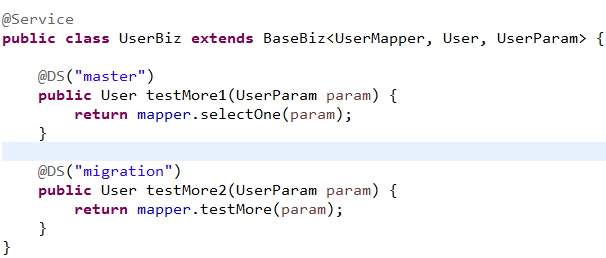
1. 主从切换、业务库切换、移植库切换，通过在Biz层的方法添加如下注解：

@DS("master"),@DS("slave"),@DS("migration"),@DS("erp")

1. 跨库操作，即在一个方法实现多库操作，这个时候需要独立一个方法来分别调用增加了DS标记的具体方法，如下图所示：

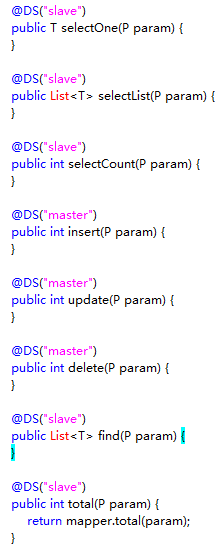


独立方法体



切库方法

1. 框架BaseBiz中的基础操作默认使用的数据源说明，如下图：



## 注意事项说明

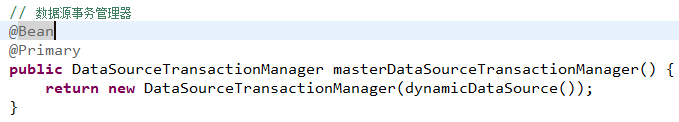
1. slave和migration配置项可以不配，但是配置项不可以删除，默认使用master作为数据源
2. 不配置就不能调用，否则报系统异常
3. 切面方法体内不可再通过切面的方式切换数据源
4. 主库也是从库池中的一个
5. 不加DS注解的方法统一默认使用master库进行操作

## 事务说明

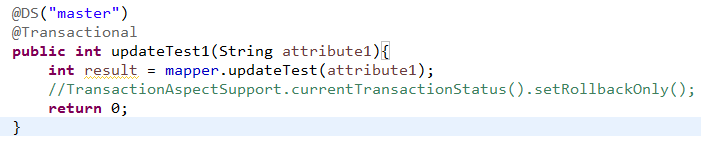
1. 独立业务库事务

首先启动事务需要在springboot的启动类添加@EnableTransactionManagement标记

其次在DataSourceConfig配置类中注入DataSourceTransactionManager事务管理器，框架生成的默认配置文件已经包含，注意数据源为动态数据源，添加@Primary标记：

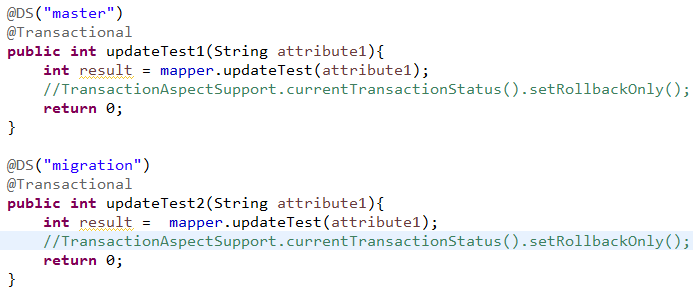


然后在需要使用的方法上添加@Transactional标记：



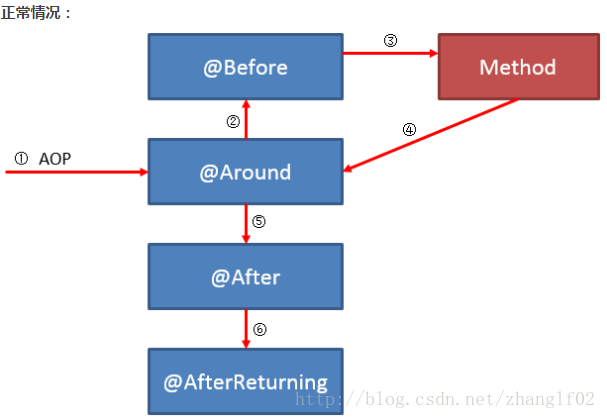
1. 添加事务后切库失败的问题分析和解决

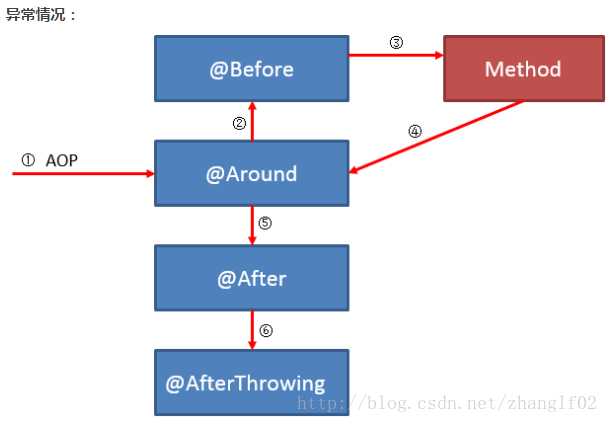
场景说明：一个biz层方法需要同时调用两个数据源，通过切面操作发现单独去调用每一个数据源可以灵活切换，后来涉及事务一个service调用两个数据源就发现动态数据源无法切换，如下图同时调用updateTest1和updateTest2，其实都使用的是master数据源。



分析原因和解决方案如下：

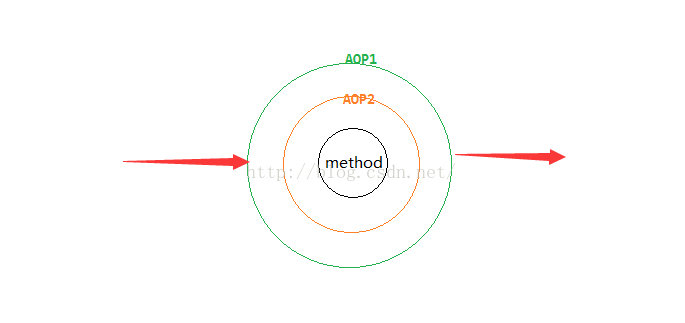
首先需要了解spring AOP的advice（通知）注解执行先后顺序：





解释：执行到核心业务方法或者类时，会先执行AOP。在aop的逻辑内，先走@Around注解的方法。然后是@Before注解的方法，然后这两个都通过了，走核心代码，核心代码走完，无论核心有没有返回值，都会走@After方法。然后如果程序无异常，正常返回就走@AfterReturn,有异常就走@AfterThrowing。

其次要了解多个AOP执行先后顺序：



spring aop就是一个同心圆，要执行的方法为圆心，最外层的order最小。从最外层按照AOP1、AOP2的顺序依次执行doAround方法，doBefore方法。然后执行method方法，最后按照AOP2、AOP1的顺序依次执行doAfter、doAfterReturn方法。也就是说对多个AOP来说，先before的，一定后after。

了解spring AOP的运作机制后再来分析我们的场景就知道原因：一个方法同时添加了两个标记@DS和@Transactional，DS为自定义标记，Transactional为spring标记，默认情况下spring标记的执行优先于自定义，此时Transactional先运行已经获取数据源连接，DS还没运行，故获取到的数据源为默认的master，即使后续运行了DS也没用，因为DS只是修改当前线程的变量并不是真正的切换数据源，只有创建连接的时候会根据这个线程变量指定数据源而已。

最终解决方案很简单：设置切换数据源的order值要比事务切面的order值小即可：



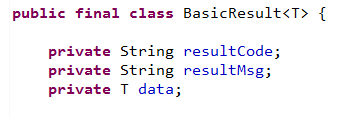
1. 跨库事务一致性

这里主要涉及跨库事务的研究，与分布式事务方案一同考虑，后续提供独立文档。

# 统一返回值格式

## 统一返回值类型定义

使用ISC微服务框架将统一所有请求返回数据的格式，包括正常返回、自定义异常返回、系统异常返回、拦截器校验失败返回等。统一使用Common包中BasicResult这个POJO类，定义如下：



resultCode：请求返回结果编码，成功统一返回ISC-000，系统异常统一返回ISC-999，其他自定义异常编码根据消息码自行定义

resultMsg：请求返回错误信息，主要两个用途，第一个在系统异常时用来存放异常堆栈信息，第二个在自定义异常时用于辅助返回提示信息或者提示信息的动态参数

data:用来存放正常返回时的处理结果，泛型编程

## 统一返回格式的拦截封装

统一返回格式不需要每一个具体服务自己封装返回结果，通过spring AOP的方式统一拦截服务请求的返回结果进行封装，具体实现存放与Common-web包中的ResultHandler类，实现原理为使用@ControllerAdvice和ResponseBodyAdvice对所有请求进行返回值的统一封装。

1. @ControllerAdvice：控制器增强，Spring4中定制用于选择控制器子集
2. ResponseBodyAdvice是spring4.1的新特性，其作用是在响应体写出之前做一些处理，例如改返回值、加密等。
3. 重写supports方法用于选择性的拦截，对于不需要拦截的方法只需要添加IgnoreResultHandler标记即可，当然也支持对于整个package不拦截封装，只要修改配置文件中的节点，如下图所示：



默认情况不拦截swagger的请求，不然swagger无法正常运行。

1. 特殊数据类型，例如byte[]需要转换成Byte[]，基本数据类型改为包装类
2. 逻辑说明：

BasicResult类型，不进行重复封装

String类型，我们要返回Json字符串，否则SpringBoot框架会转换出错

## 统一返回格式的应用

* + 1. 通过WebBean注入ResultHandler，然后在业务启动类import进去后即启用统一返回格式的拦截封装
  1. 服务提供方不需要自己封装返回值，按照实际业务返回需要的数据类型即可
  2. 前端调用后台服务，获取到的格式是BasicResult，要注意先判断返回码再获取data
  3. 服务间服务提供和消费也类似，提供方按照实际业务返回需要的数据类型即可，消费方获取到的格式是BasicResult，同样要注意先判断返回码再获取data

# 统一异常捕捉处理

之前处理工程异常，代码中最常见的就是try-catch-finally，有时一个try，多个catch，覆盖了核心业务逻辑。框架对后台异常进行了统一拦截处理，使业务代码能聚焦于逻辑的实现且减少重复代码出现，同时对于异常进行统一数据返回格式BasicResult的封装，结合消息码机制，实现前端界面和服务间互调的友好度。

## 实现原理

spring能够较好的处理这种问题，通过@ExceptionHandler和@ControllerAdvice能够集中异常，使异常处理与业务逻辑分离。

**@ControllerAdvice**：异常集中处理，更好的使业务逻辑与异常处理剥离开。

该注解作用对象为TYPE，包括类、接口和枚举等，在运行时有效，并且可以通过Spring扫描为bean组件。其可以包含由@ExceptionHandler、@InitBinder 和@ModelAttribute标注的方法，可以处理多个Controller类，这样所有控制器的异常可以在一个地方进行处理。

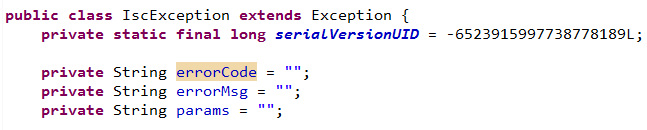
**@ExceptionHandler**：统一处理某一类异常，从而能够减少代码重复率和复杂度。

该注解作用对象为方法，并且在运行时有效，value()可以指定异常类。由该注解注释的方法可以具有灵活的输入参数：异常参数、请求或响应对象、Session对象、InputStream/Reader、OutputStream/Writer、Model等，方法返回值可以为：Model对象、Map对象、String对象、还有@ResponseBody、HttpEntity<?>或ResponseEntity<?>，以及void。

本框架在common-web包中提供了ExceptionsHandler类，并通过WebBean注入，实现异常的统一封装处理。

## 自定义异常处理

本框架的自定义异常统一使用Common包中已定义的IscException类，类定义如下：



errorCode为对应的消息码，支持对应提示信息的多语言配置，同时对于前端支持提示方式的配置，如消息类型(info/warn/error)、自动消失等。

errorMsg为错误辅助提示信息

params为提示信息的动态参数，例如某个消息码配置的消息内容为Hello,{} and {}!，此时params存放信息A,B，则最终提示信息为Hello,A and B。params不同参数值使用逗号分隔。

ExceptionsHandler中对自定义异常进行单独处理，首先文件日志会记录一个警告级别的日志，然后将IscException转化成统一的数据返回格式，errorCode赋值给BasicResult的resultCode属性，将IscException序列化成Json字符串赋值给BasicResult的resultMsg属性。

## 系统异常处理

除了自定义异常之外其他异常都将统一处理，首先文件日志会记录一个错误级别的日志，然后将Exception转化成统一的数据返回格式，系统异常resultCode默认为ISC-999，resultMsg存储异常的堆栈信息，其中堆栈信息已经进行Json特殊字符的转换处理。

## 调用方说明

* 1. 前端调用方

通过消息码方案+统一异常处理，实现前端服务请求结果的统一拦截处理，对于请求结果进行统一判断，resultCode为ISC-000为正常返回不作任何处理，如果为ISC-999为后台系统内部异常进行统一友好提示，堆栈信息默认隐藏需要时通过More方式查看，其他消息码为自定义消息码，根据后台配置进行展示。

* 1. 服务调用方

对于服务间的互调，从代码规范性来说，建议对返回结果优先进行resultCode的判断，如果为ISC-000才获取data中的内容，如果为其他消息码则进行日志记录再进行相关容错处理。

# 断路器使用说明

## 断路器基础配置

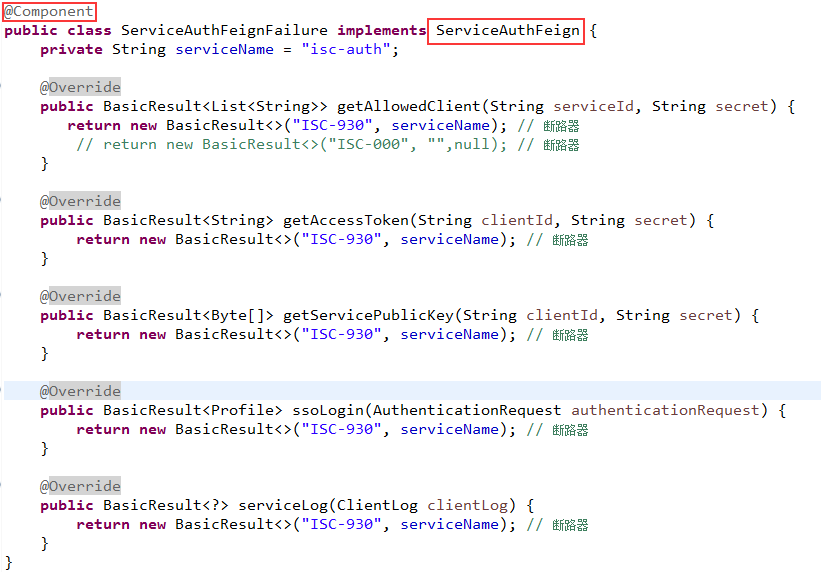
* 1. pom文件添加hystrix和actuator依赖
  2. application.properties配置文件添加feign.hystrix.enabled=true，Feign是自带断路器的，在D版本的Spring Cloud之后，它没有默认打开。需要在配置文件中配置打开它
  3. 启动类添加注解@EnableCircuitBreaker

## 断路器使用

1. 用@FeignClient注解添加fallback类， 该类必须实现@FeignClient修饰的接口。

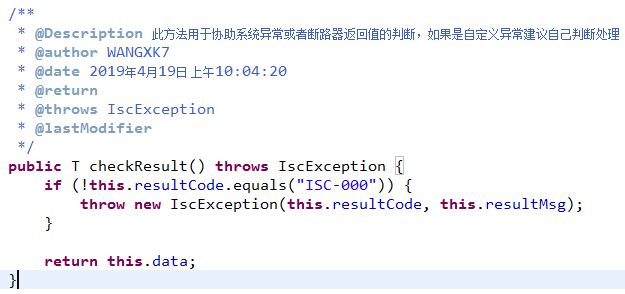


1. 创建错误处理类，必须实现被@FeignClient修饰的接口。注意添加@Component或者@Service注解，在Spring容器中生成一个Bean

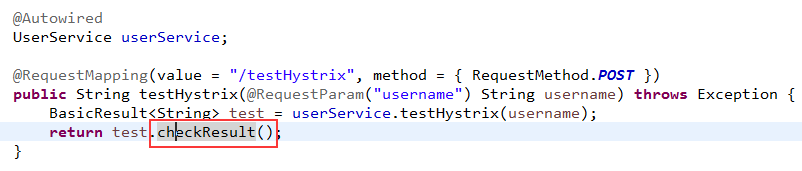


1. 统一断路器的返回方式

基于框架的统一数据返回格式，所有订阅方获取到的数据结构都为BasicResult，所以建议根据实际业务逻辑处理，如果可以忽略这个断路继续执行逻辑则返回ISC-000的消息码同时指定默认的返回值等，如果不可忽略则返回对应的错误码(例如框架的统一断路提示消息码为ISC-930)，BasicResult类提供返回值消息码判断方法checkResult，对非ISC-000的消息码统一抛自定义IscException中断程序逻辑。



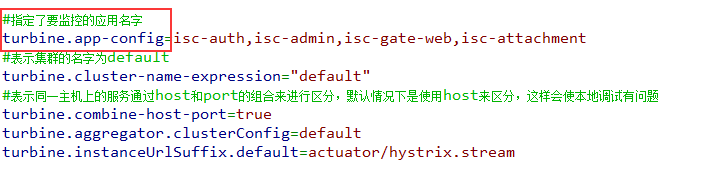
checkResult定义



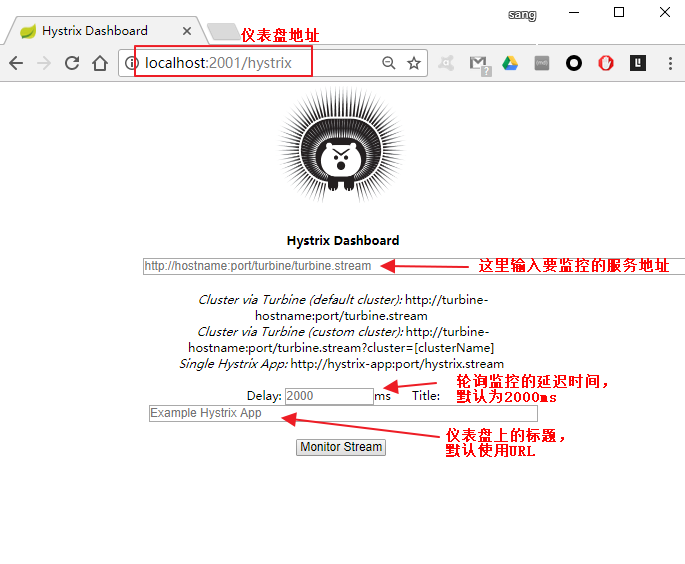
checkResult使用

## 断路器聚合监控

1. 由于各个服务实例没有添加断路器仪表盘无法查看断路器状态，固需要在断路器聚合监控平台添加各个服务实例的监控，对于有监控需求的项目通知turbine平台管理员添加配置即可



1. 仪表盘监控配置说明



1. 仪表盘参数说明



# 统一日志方式

## 文件日志

1. 对于文件日志统一使用slf4j，优势说明如下：
2. slf4j为各种loging APIs提供一个简单统一的接口，slf4j只是一个日志标准并不是一种具体的日志系统，而是一个用户日志系统的facade，允许用户在部署最终应用时方便的变更其日志系统。
3. slf4j日志方法支持字符串格式化，传统的日志需要进行字符串的拼接，用字符串拼接的构造方式在debug disabled的情况下，字符串消息还是会被求值，存在类型转换和字符串连接的性能消耗。使用slf4j的格式化功能，这种用法不存在上面提到的缺点。slf4j使用自己的格式化语法{}，同时提供了适合不同参数个数的方法重载。例如：

logger.debug("Set {1,2} differs from {}", "3"); //output:Set {1,2} differs from 3

logger.debug("Set {1,2} differs from {{}}", "3"); //output:Set {1,2} differs from {3}

logger.debug("Set \\{} differs from {}", "3"); //output:Set {} differs from 3

logger.debug("File name is C:\\\\{}.", "file.zip"); //output:File name is C:\file.zip

1. slf4j使用方式：
   * 1. @Slf4j(推荐)

需要安装lombok插件，标记于类上可直接使用logger进行日志记录

* + 1. LoggerFactory获取

private static final Logger log = LoggerFactory.getLogger(this.class);

使用static修饰的属性是归这个类使用，也就是说不论这个类实例化多少个，大家用的都是同一个static属性。

1. 默认日志Logback：

默认情况下，Spring Boot会用Logback来记录日志，并用INFO级别输出到控制台。在运行应用程序和其他例子时，你应该已经看到很多INFO级别的日志了。spring-boot-starter其中包含了spring-boot-starter-logging，该依赖内容就是 Spring Boot 默认的日志框架 logback。Logback为log4j新一代日志框架，性能更优，同一开发团队。

1. 自定义日志配置：

Spring Boot官方推荐优先使用带有-spring的文件名作为你的日志配置（如使用logback-spring.xml，而不是logback.xml），命名为logback-spring.xml的日志配置文件，spring boot可以为它添加一些spring boot特有的配置项。默认的命名规则，并且放在 src/main/resources 下面即可。如果你即想完全掌控日志配置，但又不想用logback.xml作为Logback配置的名字，application.yml可以通过logging.config属性指定自定义的名字。

自定义配置文件logback-spring.xml这里不详细解释，具体可看项目中的配置文件，只针对几个重点进行说明：

1. 日志文件名称和存放路径从springboot的配置文件获取，不在此文件定义
2. 默认配置只保留最近90天的日志，日志文件的上限大小为500M，超过这个值就会删除旧的日志。可根据项目实际情况修改
3. 根据指定运行环境区分日志级别，根据配置文件的spring.profiles.active区分dev/sit/uat/ver/pro，springboot默认情况下打印info级别。默认添加如下配置针对DAO包进行全环境DEBUG日志设置输出SQL，对于其他包非pro下打印debug级别，pro环境下打印info级别以上，如下图：



## 数据库服务日志

服务日志是提供给服务调用行为的一种详细日志记录，并结构化到数据库中，主要用于记录请求的详细信息方便定位问题。由于记录到数据库中所以不建议所有的服务方法都添加服务日志，也不建议访问频率高的服务进行日志，服务日志耗费一定性能，所以慎重选择。目的主要是对于比较敏感的服务或者需要定位错误的服务进行日志。

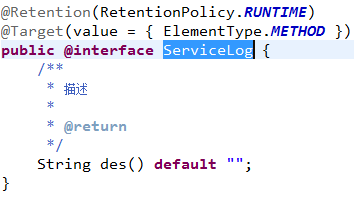
1. 日志表结构说明

|  |  |
| --- | --- |
| 字段列名 | 字段描述 |
| access\_id | 主键 |
| source\_service | 来源服务 |
| target\_service | 目标服务 |
| token | 来源服务令牌 |
| method | post/get/put/delete |
| uri | 服务请求详细地址 |
| ip\_address | 请求来源ip |
| request\_body | 请求体内容 |
| response\_body | 返回数据，只有方法正常返回时有值 |
| duration | 耗时 |
| exception | 异常信息，发生异常时记录堆栈信息 |
| access\_time | 访问时间 |

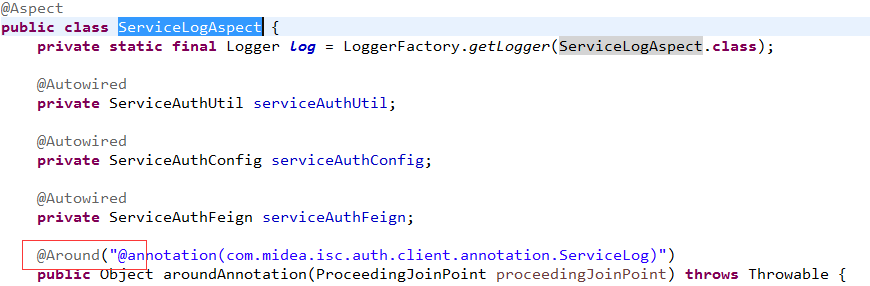
1. 实现原理

自定义注解 + AOP的方式实现服务日志的记录

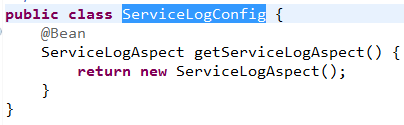
自定义注解ServiceLog，添加到需要日志的Controller中方法：



切面类ServiceLogAspect，使用@Around是可以同时在所拦截方法的前后执行一段逻辑：

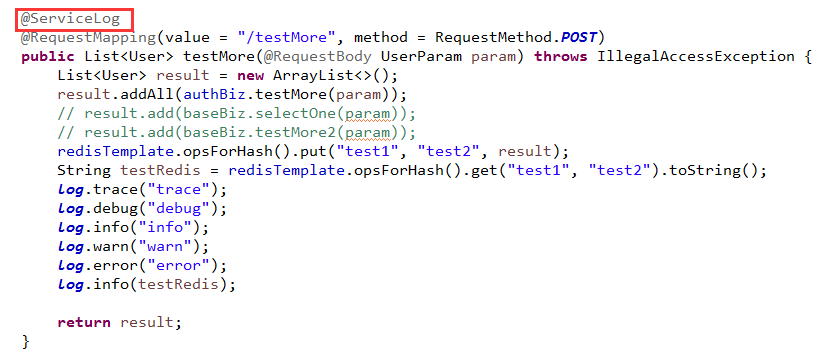


Bean注入配置类ServiceLogConfig，在项目的启动类中Import：



1. 使用方法

使用方法很简单，服务日志Bean的注入在代码生成的时候已经包含可直接使用，在需要日志的Controller方法前添加@ServiceLog注解即可，如下图：



# 时区转换和日期格式转换

国际系统常见的问题有不同时区的时间转换问题和不同国家/地区的日期显示格式问题。这两个问题都涉及时间的序列化和反序列化，统一考虑方案：

## 应用场景说明

* + - 1. 时区转换问题：对于考虑时区转换的时间字段，在数据库统一存放格林威治时间，系统间传递也是基于格林威治时间。这里需要注意部分字段的赋值由应用服务器设置如creationDate和lastUpdDate，此时应该设置应用服务器的时区为零时区，数据库字段如果使用数据库函数获取日期则需要设置数据库时区为零时区。

前后端时间传递的转换方案有两种：

1. 前后端时间传递的时区转换根据用户信息的时区进行换算
2. 前后端时间传递的时区转换根据客户端浏览器的时区进行换算

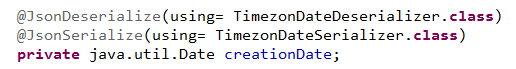
无论使用哪一种方案，只要前后端的标准一致即可。目前框架建议使用第一种，也提供了支持，显示的时间也是根据用户配置的所属时区来决定。如果使用第二种方案，则由前端支持，忽略框架的支持即可。

* + - 1. 日期格式问题：不同国家/地区的日期显示格式各有差异，数据库使用数据库支持的格式存放即可，系统间传递使用默认的yyyy-MM-dd HH:mm:ss格式，而前后端的传递则根据登录用户的所属国家/地区进行转换，当前尚未完成按国家在后台配置而是写死在常量中，目前支持的格式有：

|  |  |
| --- | --- |
| 国家/地区 | 日期格式 |
| CN | yyyy/MM/dd HH:mm:ss |
| JP | yyyy/MM/dd HH:mm:ss |
| US | MM/dd/yyyy HH:mm:ss |
| VN | MM/dd/yyyy HH:mm:ss |
| Default | yyyy/MM/dd HH:mm:ss |

## 框架解决方案

基于上述结论，框架提供了自定义的日期序列化(TimezonDateSerializer)和反序列化器(TimezonDateDeserializer)进行支持，使用方式简单如下图：



对需要进行时区换算和日期显示格式转换的属性添加两个标记。

TimezonDateSerializer：获取当前登录用户信息Profile，根据用户所属国家设置日期序列化格式(参照上表映射)，根据用户所属时区从GMT时区进行时间换算。当获取不到用户信息时，默认序列化格式为：yyyy/MM/dd HH:mm:ss，且不进行时区换算，直接输出数据库存储的时间。

TimezonDateDeserializer：获取当前登录用户信息Profile，前端传递过来的时间格式目前仅支持两种：yyyy-MM-dd HH:mm:ss和yyyy-MM-ddTHH:mm:ss，根据用户所属时区，将时间换算为GMT时间。当获取不到用户信息时，不进行时区的换算，保持原有传递过来的时间。

# 其他辅助开发项

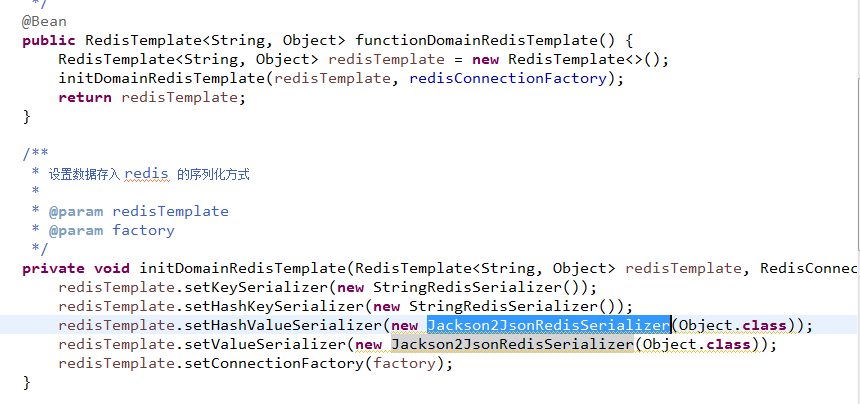
## Redis初始化和使用

1. Redis初始化配置和注入

本框架默认使用redis作为分布式缓存，springboot2.x版本使用redis需要添加依赖：spring-boot-starter-data-redis，代码生成器默认生成的maven工程已经完成redisTemplate的初始化和注入。

redisTemplate 默认的序列化方式为 jdkSerializeable, StringRedisTemplate的默认序列化方式为StringRedisSerializer可以通过手动配置, 将redisTemplate的序列化方式进行更改。

Common包的RedisConfig类完成redisTemplate注入和序列化方式的修改，代码如下：



从代码中可以看出key和hashkey使用StringRedisSerializer只允许字符串，传入其他类型会报错，而value和hashvalue使用Jackson2JsonRedisSerializer允许Jackson库将对象序列化为JSON字符串存储。优点是速度快，序列化后的字符串短小精悍，不需要实现Serializable接口。

JdkSerializationRedisSerializer: 使用JDK提供的序列化功能。 优点是反序列化时不需要提供类型信息(class)，但缺点是需要实现Serializable接口，还有序列化后的结果非常庞大，是JSON格式的5倍左右，这样就会消耗redis服务器的大量内存

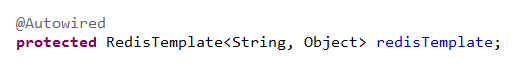
在项目工程的启动类中Import上图的RedisConfig类即完成了Redis的注入。

1. Redis的集群配置，配置文件已经连接国际系统的Redis测试集群，配置如图所示：



1. RedisTemplate使用

直接自动注入即可使用，如图所示：



## Swagger配置和使用

Swagger 是一个规范和完整的框架，用于生成、描述、调用和可视化 RESTful 风格的 Web 服务。总体目标是使客户端和文件系统作为服务器以同样的速度来更新。文件的方法，参数和模型紧密集成到服务器端的代码，允许API来始终保持同步。作用：1. 接口的文档在线自动生成2. 功能测试。

1. Swagger配置

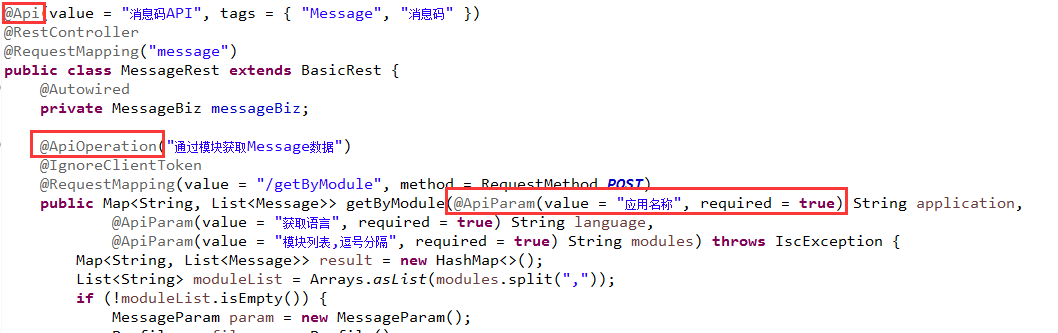
Swagger统一配置类SwaggerConfig存放在common-web中，主要完成扫描包的获取，标题设置，文档描述，版本管理，管理员配置等，在项目工程的启动类中Import配置类SwaggerConfig类即完成了Swagger的注入。对应的配置节点如下图：



1. Swagger使用

注意只能在配置文件中指定的包中使用Swagger的标记，其他包都不会被扫描。

使用方式如下图：

Swagger使用的注解及其说明：

- @Api()用于类；

表示标识这个类是swagger的资源

- @ApiOperation()用于方法；

表示一个http请求的操作

- @ApiParam()用于方法，参数，字段说明；

表示对参数的添加元数据（说明或是否必填等）

- @ApiModel()用于类

表示对类进行说明，用于参数用实体类接收

- @ApiModelProperty()用于方法，字段

表示对model属性的说明或者数据操作更改

- @ApiIgnore()用于类，方法，方法参数

表示这个方法或者类被忽略

- @ApiImplicitParam() 用于方法

表示单独的请求参数

- @ApiImplicitParams() 用于方法，包含多个 @ApiImplicitParam

更多Swagger的注解请自行百度。

1. Swagger相关扩展

Eureka注册中心集成Swagger，通过配置

eureka.instance.status-page-url=http://localhost:${server.port}/swagger-ui.html实现注册中心中实例状态可直接链接到该服务实例的Swagger文档。

统一返回格式封装需要忽略swagger的请求，否则将导致Swagger无法使用，配置文件已默认添加：result.hanlder.ignore=springfox.documentation.swagger

网关整合Swagger：通过路由遍历所有服务实现swagger文档的整合，实现网关对应的swagger可以查看所有服务实例的swagger，具体实现需要在网关进行扩展。

## 关于springboot配置文件格式

* + 1. yml格式文件

yaml语法：

1. 结构通过缩进来表示，连续的项目（如：数组元素、集合元素）通过减号“-”来表示，map结构里面的键值对（key/value）用冒号“:”来分割。yaml也有用来描述好几行相同结构数据的缩写语法，数组用“[]”包括起来，hash用“{}”来包括
2. 用#做注释
3. 用on、1、true来表示true，off、0、false来表示false

使用yaml的注意事项：

1. 结构通过缩进来表示，yaml不支持制表符tab缩进，请使用空格缩进
2. 如果参数是以空格开始或结束的字符串，应使用单引号把他包进来。如果一个字符串参数包含特殊字符，也要用单引号包起来
3. 每个冒号后面一定要有一个空格
4. 想要表示列表项，使用一个短横杠加一个空格
5. yaml中，空值可以用null或~表示
   * 1. properties格式文件

properties文件在Java中主要为配置文件，文件类型为：.properties，格式为文本文件，内容格式为"键=值"，格式相对简单，但开发过程中存在中文乱码问题。对于IDE使用Ecplise建议安装propedit插件，Idea则通过修改编码格式解决。

## Controller和RestController的区别

1. @RestController注解相当于@ResponseBody ＋ @Controller
2. 使用@RestController注解Controller，则Controller中的方法无法返回jsp页面，或者html，配置的视图解析器 InternalResourceViewResolver不起作用
3. 使用@Controller 注解，若返回json等内容到页面，则需要加@ResponseBody注解

## 配置文件配置项获取方式

在SpringBoot中读取XXX.properties等配置文件有许多方法，常见的方法有：@ConfigurationProperties方式、使用@Value注解方式、使用Environment、以及使用PropertiesLoaderUtils这四种方式。

1. @ConfigurationProperties注解将配置文件属性注入到配置对象类中，可指定前缀prefix，配置项不存在时，编译不通过
2. @Value("${propertyName}")注解指定具体配置项名称获取，配置项不存在时，编译不通过
3. 使用Environment的getProperty方法，配置项不存在是返回null
4. 通过注册监听器Listeners +PropertiesLoaderUtils的方式，这种方式不建议使用

## 初始化bean注入方式

在项目启动jar包内bean的初始化通过@Configuration标记类，@Bean标记方法即可实现Bean的注入

不同一个jar包的bean注入使用@Import注解通过导入的方式实现把实例加入springIOC容器中。

## 关于项目划分jar包的思考

1. API包(SDK)

API包一般设计用于存放接口和DTO供使用方引用，减少重复代码的编写和方便编译调试。但本框架不建议提供API包，原因如下：

1. 目前框架的统一拦截进行数据返回格式的封装，无需在每个方法封装返回格式，导致API定义的接口返回值和Controller实现不一致
2. 调用方将扫描API包注入所有的bean，耗费内存同时拖延了启动速度，实际使用中服务间互调应该保持在一定的接口数量，出现过多接口频繁调用就要考虑高内聚低耦合设计是否合理。虽然扫描方式除了包之外还可以细化到类，但如果细化到类则需要配置的具体类数量会太大
3. 同时要考虑feigh的断路器返回值，建议由各个调用方定义，而不是由提供方统一定义
4. 关于统一DTO，基于Http协议的restful的设计理念，无状态，无约束，灵活性绑定，DTO的共享可以减免重复定义，但是限定了调用方使用的灵活性，建议按需定义入参和出参，无需全盘接收(一个接口可能是提供给多方调用，DTO属性会超出某个具体调用方所需要的)
5. 使用API包(SDK) 来调用服务端接口的方式，使用起来确实比较方便，不看接口文档都可以直接编程了。但是要小心服务端接口的版本问题，基于restful可以解耦服务提供方和消费方，应该利用这个特性解耦
6. 接入层和底层的拆包

可以通过网关路由的方式解决前端、外围系统和服务互调负载不均衡的问题，当路由无法解决时才考虑拆包，普通项目一般不拆。

微服务工程本来就很细粒度了，无需在微服务工程里再分模块了，不然会增加复杂度。

# 版本修订历史记录

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 作者 | 内容 | 版本 |
| 2019-03-13 | 王孝坤 | ISC微服务后台框架设计说明 | V1.0 |
| 2019-03-27 | 王孝坤 | 添加时区换算和日期格式内容 | V1.1 |
| 2019-03-28 | 王孝坤 | 添加动态数据源的事务说明 | V1.2 |
| 2019-04-11 | 王孝坤 | 添加数据库批量操作的说明以及update乐观锁的实现方案 | V1.3 |
| 2019-04-22 | 王孝坤 | 添加断路器使用说明 | V1.4 |