SCC框架使用手册

V3.0.1

贾中进

# 目录

[目录 1](#_Toc27039833)

[1 架构 1](#_Toc27039834)

[2 目录结构 1](#_Toc27039835)

[3 支撑环境 2](#_Toc27039836)

[3.1 版本依赖 2](#_Toc27039837)

[3.2 Zipkin 2](#_Toc27039838)

[3.3 Turbine 3](#_Toc27039839)

[4 业务开发 3](#_Toc27039840)

[4.1 Model开发 3](#_Toc27039841)

[~~4.1.1~~ ~~JPA( 已遗弃)~~ 3](#_Toc27039842)

[4.1.2 Mybatis 3](#_Toc27039843)

[4.2 持久层ORM（数据库操作） 4](#_Toc27039844)

[~~4.2.1.1~~ ~~JPA（已遗弃）~~ 4](#_Toc27039845)

[4.2.1.2 Mybatis 6](#_Toc27039846)

[4.3 Service开发 8](#_Toc27039847)

[4.3.1 异常处理 8](#_Toc27039848)

[4.4 Controller开发 9](#_Toc27039849)

[4.4.1 HTTP请求类型 9](#_Toc27039850)

[4.4.2 异常处理 9](#_Toc27039851)

[4.4.3 同步请求 9](#_Toc27039852)

[4.4.4 异步请求 10](#_Toc27039853)

[5 组件集 10](#_Toc27039854)

[5.1 文件存储 10](#_Toc27039855)

[5.2 临时文件下载 11](#_Toc27039856)

[5.3 异步处理 12](#_Toc27039857)

[5.4 缓存 13](#_Toc27039858)

[5.4.1 内存缓存 13](#_Toc27039859)

[5.4.2 Ehcache缓存（建议不要使用） 14](#_Toc27039860)

[5.4.3 Redis缓存 14](#_Toc27039861)

[5.5 字典信息获取 15](#_Toc27039862)

[5.6 系统配置信息获取 15](#_Toc27039863)

[5.7 微服务间调用数据标准 16](#_Toc27039864)

[5.8 用户认证 16](#_Toc27039865)

[5.9 Mock 17](#_Toc27039866)

[5.10 路由 18](#_Toc27039867)

[5.11 用户权限认证/授权 18](#_Toc27039868)

[5.12 SHAPE缓存 19](#_Toc27039869)

[5.13 共享注册中心（DEBUG） 20](#_Toc27039870)

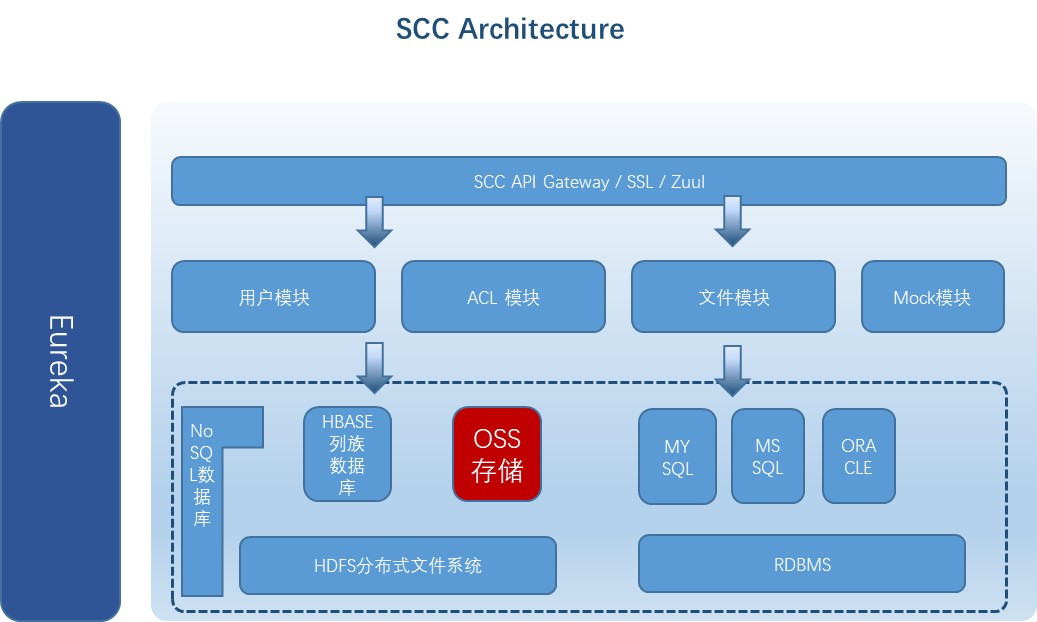
[5.14 用户、角色、资源、权限服务模块 21](#_Toc27039871)

[6 部署 21](#_Toc27039872)

[7 总结 21](#_Toc27039873)

# 架构

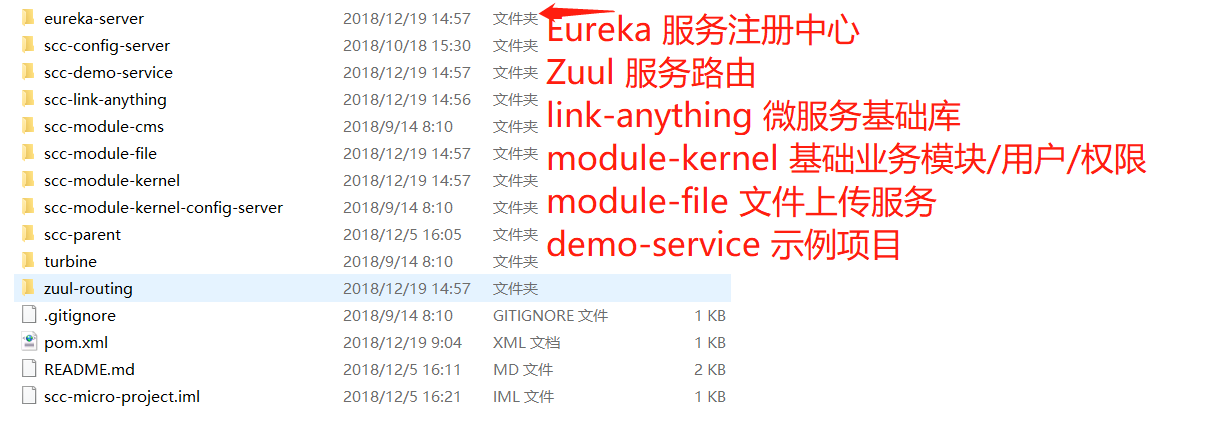
基于Spring Cloud构建，服务化、模块化、组件化、微服务化一些常见的功能。继承了微服务架构的伸缩性，扩展性。在一些模块组件的基础上，加上统一的代码开发规范，使项目开发过程更清晰、简单、直接、标准、健壮。



# 目录结构

|  |  |  |  |
| --- | --- | --- | --- |
| 项 | 名称 | 版本 | 描述 |
| 库(lib) | lib | 3.0.1 | 基础代码库，所有的项目必须引入 |
| 模块(modules) | hadoop | 3.0.1 | Hadoop支撑 |
| http | 3.0.1 | Http Client实现的HTTP |
| tkdd | 3.0.1 | 简单任务流，跟踪、保证任务完成。 |
| \_cluster\_ | 3.0.1 | 分布式Master Slave |
| \_ny\_ | 3.0.1 | Netty Http Server/Client 实现 |
| Es | 3.0.1 | 全文检索 |
| HBase | 3.0.1 | HBase存储 |
| Kafka | 3.0.1 | 消息队列 |
| Zookeeper | 3.0.1 | 分布式协调 |
| Data streaming | 3.0.1 | 简单的数据流处理 |
| Mongo | 3.0.1 | 文档型NOSQL |
| Mybatis | 3.0.1 | ORM |
| Web | 3.0.1 | Web开发需要的常用功能组件 |
| 框架 | Web CPP | 3.0.1 | 包含用户、角色、资源、权限、菜单等功能 |
| Web SCC | 3.0.1 | 微服务基础框架，整合了用户、角色、资源、权限、菜单、文件服务等功能 |
| 产品 | OSS | 3.0.1 | 大数据存储、分析平台 |

**SCC**



* eureka-server 服务注册中心**（生产环境需要启动）**
* zuul-routing 服务路由**（生产环境需要启动）**
* scc-module-kerkel 基础核心服务**（生产环境需要启动）**
* scc-module-file文件上传服务**（生产环境需要启动）**
* scc-demo-service 微服务其他新服务参考配置/编码模板
* scc-link-anything 这是一个基础JAR包，只需要打包、发布就好了，不需要启动
* turbine 服务链路熔断监控服务， 使用了turbine stream ， hystrix dashboard

# 支撑环境

服务链路跟踪 Sleuth/Zipkin，服务熔断监控Turbine 等都需要Spring Cloud Stream 支持，当前通过stream(Kafka)减缓服务间网络通道流量。

## 版本依赖

|  |  |  |
| --- | --- | --- |
| 项 | 版本 | 备注 |
| Spring Cloud | 2.0.0.RELEASE |  |
| Spring Boot | 2.0.3.RELEASE |  |

## Zipkin

* kafka的topic为 zipkin-stream-topic
* java -jar zipkin-server-2.10.1-exec.jar --HTTP\_COLLECTOR\_ENABLED=false --KAFKA\_BOOTSTRAP\_SERVERS=one.3cgg.rec:9092 --KAFKA\_TOPIC=zipkin-stream-topic --STORAGE\_TYPE=mysql --MYSQL\_USER=root --MYSQL\_PASS=jiazhongjin

## Turbine

* kafka的topic为 turbine-stream-topic

# 业务开发

业务采用**模块化**的方式来开发，模块的包结构如下：

{project package}/{module name}/**controller**

{project package}/{module name}/**mapper**

{project package}/{module name}/**service**

{project package}/{module name}/**model**

1. **Controller** API的入口参数，返回数据定义，是消费者消费服务的协议，准则。
2. **Service** 关键性的服务业务，一般业务应当包含在一个独立的事务中。这些服务业务被 controller 消费，或者被其他服务业务消费。
3. **Mapper** 数据访问层接口定义
4. **Model** 与关系型数据库、NoSQL存储或者索引服务之间模型的映射。强烈建议使用简单的单实体即不包含实体之间的引用关系
5. **VO** API 消费者协定的数据结构

## Model开发

### ~~JPA( 已遗弃)~~

Model需要继承me.libme.module.spring.jpahibernate.\_m.JBaseModel，请使用JPA相关的映射注解标注实体表映射，常用的有 @Entity、@Table 、@Column，强烈建议实体不包含其他的实体，即不需要one-2-many or many-2-one 机制。

### Mybatis

Model需要继承me.libme.module.spring.mybatis.BaseModel

|  |
| --- |
| 1. **package** demo.libme.web.orm.user.model; 2. **import** me.libme.module.spring.mybatis.BaseModel; 3. /\*\* 4. \* Created by J on 2018/8/10. 5. \*/ 6. **public** **class** User **extends** BaseModel{ 8. **private** String userName; 10. **private** String password; 12. **private** String phone; |

## 持久层ORM（数据库操作）

**数据库所有的表设计必须带一个主键，建议用UUID**

#### ~~JPA（已遗弃）~~

数据库数据访问分为下面两种：单表操作，多表操作(查询)。

##### 单表操作

使用帮助类me.libme.module.spring.jpabean.SingleEntityManager，包含所有的增删改查。

###### 主键操作相关API

* me.libme.module.spring.jpabean.SingleEntityManager#saveOnly 持久化单个实体
* me.libme.module.spring.jpabean.SingleEntityManager#updateOnly 更新单个实体
* me.libme.module.spring.jpabean.SingleEntityManager#delete(java.lang.String, java.lang.Class<?>...) 根据主键删除单个实体，不会删除记录，仅仅把记录标记为删除

###### 复杂多条件查询API

如果你需要复杂的条件查询请使用下面的方法，me.libme.module.spring.jpabean.SingleEntityManager#singleEntityQuery2，示例如下

|  |
| --- |
| 1. **public** <T> T getParamCode(String code,String type,Class<T> clazz)  **throws** Exception{ 2. **return** paramCodeRepo.singleEntityQuery2() 3. .conditionDefault() 4. .equals("code",code) 5. .equals("type",type) 6. .ready() 7. .model(clazz); 8. } |

具体API请查看源码。

##### 多表操作

* me.libme.module.spring.jpahibernate.query2.JNativeQuery 基于Native SQL的查询
* me.libme.module.spring.jpahibernate.query2.JJPQLQuery 基于JPQL SQL的查询
* me.libme.module.spring.jpahibernate.query2.JNamedQuery 基于Named SQL查询

请使用从me.libme.module.webboot.fn.jpa.DataAccessSupport类继承的方法，来执行上面对应的操作

|  |
| --- |
| 1. **protected** JNativeQuery nativeQuery(){ 2. **return** **new** JNativeQuery(entityManager); 3. } 5. **protected** JJPQLQuery jpqlQuery(){ 6. **return** **new** JJPQLQuery(entityManager); 7. } 9. **protected** JNamedQuery namedQuery(){ 10. **return** **new** JNamedQuery(entityManager); 11. } |

###### JPQL Query

可以写JPQL查询语法，如何写JPQL请查询JPQL语法说明。下面是示例

~~当前还没有使用示例~~

使用方法请参考 Native Query, 仅仅把SQL语法换成 JPQL 就好了，其他都不变。

###### Native Query

可以写Native SQL 执行数据库查询, 示例

|  |
| --- |
| 1. **private** JQuery<?> buildRoleOnGroupQuery( 2. Map<String, JCondition.Condition> params){ 3. String jpql="select a.id as id" 4. + ", a.roleCode as roleCode" 5. + ", a.roleName as roleName " 6. + ", a.description as description " 7. + " from Role a" 8. + " , RoleGroup b " 9. + " , Group c " 10. + " where b.roleId=a.id and b.groupId=c.id and a.deleted='N' and c.deleted='N' and b.deleted='N' "; 11. JCondition.Condition condition=**null**; 12. **if**((condition=params.get("groupId"))!=**null**){ 13. jpql=jpql+ " and c.id "+condition.getOpe()+" :groupId"; 14. } 15. **return** jpqlQuery().setSql(jpql) 16. .setParams(params(params)); 17. } |

**注意： 返回实体采用（别名-属性【意味你的别名要和属性名一样，大小写敏感】）映射方式，请确认你的返回值列表都存在于你要返回的实体里面，否则会报错。**

###### Named Query

~~当前还没有使用示例~~

#### Mybatis

##### Mapper

Mapper必须定义为接口，且需要继承me.libme.module.spring.mybatis.IMybatisSingleEntityAccess，方法定义规则

* 分页方法最后一个参数必须为me.libme.kernel.\_c.\_m.Pageable，返回类型为me.libme.kernel.\_c.\_m.Page

|  |
| --- |
| 1. **package** me.libme.scc.usernew.mapper; 3. **import** me.libme.kernel.\_c.\_m.Page; 4. **import** me.libme.kernel.\_c.\_m.Pageable; 5. **import** me.libme.module.spring.mybatis.IMybatisSingleEntityAccess; 6. **import** me.libme.scc.usernew.model.User; 7. **import** org.apache.ibatis.annotations.Param; 9. **import** java.util.List; 11. /\*\* 12. \* Created by J on 2018/8/10. 13. \*/ 14. **public** **interface** UserNewMapper **extends** **IMybatisSingleEntityAccess<User,String>** {  17. **Page<User>** pageUsers(@Param("phone") String phone, **Pageable pageable**);  20. List<User> getUsers(@Param("phone")String phone); 22. User getUser(@Param("userName")String userName); 24. } |

##### XML

* saveOnly 保存单个实体
* updateOnly 更新单个实体
* delete 标记删除单个实体
* deletePermanently 永久删除单个实体
* getById 根据主键查询单个实体

|  |
| --- |
| 1. **<?xml** version="1.0" encoding="UTF-8" **?>** 2. <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd" **>** 3. **<mapper** namespace="me.libme.scc.usernew.mapper.UserNewMapper" **>** 5. **<resultMap** id="BaseResultMap" type="me.libme.scc.usernew.model.User" **>** 6. **<id** column="id" property="id" jdbcType="VARCHAR" **/>** 7. **<result** column="user\_name" property="userName" jdbcType="VARCHAR" **/>** 8. ... 9. **</resultMap>** 11. **<sql** id="Base\_Column\_List" **>** 12. id, user\_name, password, phone,CREATE\_ID, CREATE\_TIME,UPDATE\_ID,UPDATE\_TIME, DELETED, VERSION 13. **</sql>**  16. **<select** id="getById" resultMap="BaseResultMap" parameterType="java.lang.String" **>** 17. select 18. **<include** refid="Base\_Column\_List" **/>** 19. from t\_user 20. where id = #{id,jdbcType=VARCHAR} 21. **</select>**  24. **<delete** id="deletePermanently" parameterType="java.lang.String" **>** 25. delete from t\_user 26. where id = #{id,jdbcType=VARCHAR} 27. **</delete>** 29. **<insert** id="saveOnly"**>** 30. insert into t\_user 31. **<trim** prefix="(" suffix=")" suffixOverrides="," **>** 32. **<if** test="id != null" **>** 33. id, 34. **</if>** 35. ... 36. **</trim>** 37. **<trim** prefix="values (" suffix=")" suffixOverrides="," **>** 38. **<if** test="id != null" **>** 39. #{id,jdbcType=INTEGER}, 40. **</if>** 41. ... 42. **</trim>** 43. **</insert>** 45. **<update** id="updateOnly" **>** 46. update t\_user 47. **<set** **>** 48. **<if** test="userName != null" **>** 49. user\_name = #{userName,jdbcType=VARCHAR}, 50. **</if>** 51. ... 52. **</set>** 53. where id = #{id,jdbcType=VARCHAR} 54. **</update>** 56. **<update** id="delete" parameterType="java.lang.String" **>** 57. update t\_user  set DELETED = 'Y' 58. where id = #{id,jdbcType=VARCHAR} 59. **</update>** 61. **<select** id="getUsers" resultMap="BaseResultMap" parameterType="java.lang.String" **>** 62. select 63. **<include** refid="Base\_Column\_List" **/>** 64. from t\_user 65. where phone = #{phone,jdbcType=VARCHAR} 66. **</select>** 68. **<select** id="pageUsers" resultMap="BaseResultMap"**>** 69. select 70. **<include** refid="Base\_Column\_List" **/>** 71. from t\_user 72. where phone = #{phone,jdbcType=VARCHAR} 73. **</select>** 75. **<select** id="getUser" resultMap="BaseResultMap" parameterType="java.lang.String" **>** 76. select 77. **<include** refid="Base\_Column\_List" **/>** 78. from t\_user 79. where user\_name = #{userName,jdbcType=VARCHAR} 80. **</select>** 82. **</mapper>** |

## Service开发

Service类定义需要加上@Service 和 @Transactional 注解。 方法定义要求

* 所有方法必须抛出java.lang.Exception
* 有分页功能的方法，最后一个参数必须是me.libme.kernel.\_c.\_m.SimplePageRequest

|  |
| --- |
| 1. **package** me.libme.scc.usernew.service; 3. **import** me.libme.kernel.\_c.\_m.Page; 4. **import** me.libme.kernel.\_c.\_m.SimplePageRequest; 5. **import** me.libme.kernel.\_c.util.UniqueUtils; 6. **import** me.libme.scc.usernew.mapper.UserNewMapper; 7. **import** me.libme.scc.usernew.model.User; 8. **import** org.springframework.beans.factory.annotation.Autowired; 9. **import** org.springframework.stereotype.Service; 10. **import** org.springframework.transaction.annotation.Transactional; 12. **import** java.util.List; 14. /\*\* 15. \* Created by J on 2018/8/10. 16. \*/ 17. **@Service** 18. **@Transactional** 19. **public** **class** UserNewService {  22. @Autowired 23. **private** UserNewMapper userNewMapper;//这里会报错，但是并不会影响 25. **public** Page<User> pageUsers(String phone, SimplePageRequest simplePageRequest) **throws** **Exception** { 26. Page<User> page= userNewMapper.pageUsers(phone,simplePageRequest); 27. **return** page; 28. } |

### 异常处理

方法签名没有抛出异常类型限制，因为异常关系到业务事务问题，建议都应当抛出运行时异常。

**注意：方法不应该隐藏运行异常，除非特殊需要，请慎重选择**

## Controller开发

Controller是定义消费者协议的类，方法定义要求

* 所有方法必须抛出java.lang.Exception
* 有分页功能的方法，最后一个参数必须是me.libme.module.webboot.SimplePageRequestVO
* 返回的数据对象必须是类似于POJO的（**禁止返回第三方JAR包中的类**，一般为JAVA8的基础数据类型，日期，BigDecimal，集合，自定义简单类等）
* 所有的返回类型必须是me.libme.kernel.\_c.\_m.ResponseModel

基于Spring MVC 开发，使用到一些简单的注解

* @Controller
* @RequestMapping ， **禁止使用PostMapping，PutMapping等快捷方式**
* @ResponseBody

此类方法定义的所有的返回数据都被封装成JSON返回。示例如下

|  |
| --- |
| 1. @ResponseBody 2. @RequestMapping(value = "/page/{phone}/{pageNum}/{pageSize}",method = RequestMethod.GET) 3. **public** ResponseModel pageUsers(@PathVariable("phone") String phone,@PathVariable("pageNum") **int** pageNum, @PathVariable("pageSize") **int** pageSize) **throws** Exception { 4. SimplePageRequestVO simplePageRequestVO=**new** SimplePageRequestVO(); 5. simplePageRequestVO.setPageNumber(pageNum); 6. simplePageRequestVO.setPageSize(pageSize); 7. SimplePageRequest simplePageRequest=**new** SimplePageRequest( 8. simplePageRequestVO.getPageNumber(),simplePageRequestVO.getPageSize()); 9. Page<User> object=userService.pageUsers(phone,simplePageRequest); 10. **return** ResponseModel.newSuccess(object); 11. } |

### HTTP请求类型

请确认一个API只对应一种HTTP请求类型，这意味着你必须在方法上添加HTTP类型注释，参考下面：

|  |
| --- |
| 1. @ResponseBody 2. @RequestMapping(value = "/page/{phone}/{pageNum}/{pageSize}",**method = RequestMethod.GET**) 3. **public** ResponseModel pageUsers(@PathVariable("phone") String phone,@PathVariable("pageNum") **int** pageNum, @PathVariable("pageSize") **int** pageSize) **throws** Exception { 4. } |

注意：虽然Spring 支持多类型映射，但业务层面的开发只能映射一种类型。建议基于项目的不同要求，简单的项目只使用GET,POST类型就好了。

### 异常处理

**类中定义的方法必须定义抛出异常类型为 Exception , 方法中不应该隐藏处理的异常，除非另有需要**

### 同步请求

与普通写法没有区别

### 异步请求

请使用类me.libme.module.webboot.\_c.SpringCallable 作为方法的返回值，示例如下（也可以参考me.libme.module.webboot.fn.file.FileEndpoint）

|  |
| --- |
| 1. @ResponseBody 2. @RequestMapping(path="/upload",method=RequestMethod.POST) 3. **public** **SpringCallable<ResponseModel>** upload(@RequestParam(name="name",required=**false**) String name, 4. @RequestParam("file") MultipartFile file) **throws** Exception { 6. **return** **new** SpringCallable<ResponseModel>() { 7. @Override 8. **public** ResponseModel doCall() **throws** Exception { 9. **if** (!file.isEmpty()) { 10. FileRequest fileRequest=**new** FileRequest(); 11. fileRequest.set\_i\_d(UniqueUtils.unique()); 12. fileRequest.setName(StringUtils.isNullOrEmpty(name) ? file.getOriginalFilename(): name); 13. fileRequest.setBytes(file.getBytes()); 14. List<FileResponse> fileResponses=fileStoreService.transfer(Arrays.asList(fileRequest)); 15. List<IAttachmentRecord> attachmentRecordVOs=**new** ArrayList<>(); 16. **for**(FileResponse fileResponse:fileResponses){ 17. IAttachmentRecord attachmentRecordVO=attachmentRecordListener.on(fileResponse); 18. **if**(attachmentRecordVO!=**null**) 19. attachmentRecordVOs.add( attachmentRecordVO); 20. } 21. **return** ResponseModel.newSuccess(attachmentRecordVOs); 22. } 23. **return** ResponseModel.newBysError("file is missing."); 24. } 25. }; 26. } |

# 组件集

## 文件存储

简单的文件存储类，在本地磁盘上存储，或者在任何一个磁盘映射上存储。

* me.libme.kernel.\_c.file.FileTransfer
* me.libme.kernel.\_c.file.SimpleDskTransfer
* **me.libme.module.webboot.fn.file.FileStoreService (主要使用这个类)**

|  |
| --- |
| 1. @Autowired 2. **private** **FileStoreService** fileStoreService; 3. @ResponseBody 4. @RequestMapping(path="/upload",method=RequestMethod.POST) 5. **public** **SpringCallable<ResponseModel>** upload(@RequestParam(name="name",required=**false**) String name, 6. @RequestParam("file") MultipartFile file) **throws** Exception { 8. **return** **new** SpringCallable<ResponseModel>() { 9. @Override 10. **public** ResponseModel doCall() **throws** Exception { 11. **if** (!file.isEmpty()) { 12. FileRequest fileRequest=**new** FileRequest(); 13. fileRequest.set\_i\_d(UniqueUtils.unique()); 14. fileRequest.setName(StringUtils.isNullOrEmpty(name) ? file.getOriginalFilename(): name); 15. fileRequest.setBytes(file.getBytes()); 16. List<FileResponse> fileResponses=fileStoreService.transfer(Arrays.asList(fileRequest)); 17. List<IAttachmentRecord> attachmentRecordVOs=**new** ArrayList<>(); 18. **for**(FileResponse fileResponse:fileResponses){ 19. IAttachmentRecord attachmentRecordVO=attachmentRecordListener.on(fileResponse); 20. **if**(attachmentRecordVO!=**null**) 21. attachmentRecordVOs.add( attachmentRecordVO); 22. } 23. **return** ResponseModel.newSuccess(attachmentRecordVOs); 24. } 25. **return** ResponseModel.newBysError("file is missing."); 26. } 27. }; 28. } |

你可以使用me.libme.module.webboot.fn.file.AttachmentRecordListener来处理文件上传完之后需要处理的业务，请参考上面的示例。

可以通过下面的属性来配置文件的处理行为等

|  |
| --- |
| 1. file: 2. endpoint-on: **true** 3. dsk-path: c:/file-store/   #simple file directory 4. host: http://localhost:8888/ #IP地址 指定文件存储的地址 |

## 临时文件下载

临时文件存储类，在本地磁盘上存储，或者在任何一个磁盘映射上存储。当前支持同步和异步存储方式。

* **me.libme.module.webboot.fn.file.DelayDownLoader(主要使用这个类)**
* me.libme.module.webboot.fn.file.TempFileTransfer

使用方式参考下面（me.libme.module.webboot.fn.file.\_TestTempFileController）

|  |
| --- |
| 1. /\*\* 2. \* Created by J on 2017/8/10. 3. \*/ 4. @Controller 5. @RequestMapping("/\_test\_temp\_file\_") 6. **public** **class** \_TestTempFileController { 8. @Autowired(required = **false**) 9. **private** **DelayDownLoader** delayDownLoader; 11. @Value("${spring.application.name}") 12. **private** String application; 14. @RequestMapping(path="/async",method= RequestMethod.GET) 15. @ResponseBody 16. **public** ResponseModel async() **throws** Exception { 17. UserTempFileResponse userTempFileResponse=**new** UserTempFileResponse(); 18. userTempFileResponse.set\_i\_d(UniqueUtils.sequence()); 19. userTempFileResponse.setApplication(application); 20. delayDownLoader.async(()->{ 21. List<UserTempFileRequest> userTempFileRequests=**new** ArrayList<>(); 22. UserTempFileRequest userTempFileRequest=**new** UserTempFileRequest(); 23. userTempFileRequest.setFileDesc("这是一个测试文档"); 24. userTempFileRequest.setName("测试文档.txt"); 25. userTempFileRequest.setRequestUserId(ServerSessionHolder.getSessionUser().getId()); 26. userTempFileRequest.setRequestUserName(ServerSessionHolder.getSessionUser().getUserName()); 27. userTempFileRequest.setTaskDesc("下载测试文档，哇哈哈"); 28. userTempFileRequest.set\_i\_d(userTempFileResponse.get\_i\_d()); 29. userTempFileRequest.setBytes("为测试准备的数据，哇哈哈".getBytes("utf-8")); 30. userTempFileRequests.add(userTempFileRequest); 31. **return**  userTempFileRequests; 33. }); 34. **return** ResponseModel.newSuccess(**new** UserTempFileResponse[]{userTempFileResponse}); 35. } |

## 异步处理

提供业务开发者线程相关的异步处理需求，尽量的隔离线程概念，减少线程操作下的资源协调问题。 当前一个异步调用以任务的形式提交给处理者，每个任务只有两种结果，成功处理，异常处理。请使用me.libme.kernel.\_c.async.TaskExecutor来处理，示例如下

|  |
| --- |
| 1. **private** **static** **TaskExecutor** executor; 3. **static** { 4. ThreadConfig threadConfig=**new** ThreadConfig(); 5. threadConfig.setName("test"); 6. threadConfig.setMaxCount(10); 7. threadConfig.setAliveCount(5); 8. threadConfig.setAliveTime(60); 9. SimpleThreadPoolExecutorFactory simpleThreadPoolExecutorFactory= 10. **new** SimpleThreadPoolExecutorFactory(threadConfig); 11. executor=**new** TaskExecutor(simpleThreadPoolExecutorFactory); 12. } 14. **public** **static** **void** a() **throws** Exception{ 15. executor.promise(**new** Task() { 16. @Override 17. **public** **void** run() { 18. LOGGER.info(Thread.currentThread().getName()+"..........call..."); 19. **try** { 20. TimeUnit.SECONDS.sleep(1l); 21. } **catch** (InterruptedException e) { 22. LOGGER.error(e.getMessage(),e); 23. } 24. } 25. }).then(**new** Call<Object,Object>() { 26. @Override 27. **public** Object call(Object input) { 28. LOGGER.info(Thread.currentThread().getName()+"..........then..."+input); 29. **return** **null**; 30. } 31. }).cat(**new** Catch() { 32. @Override 33. **public** **void** cat(Throwable error) { 34. LOGGER.info(Thread.currentThread().getName()+"..........catch..."); 35. LOGGER.error(error.getMessage(),error); 36. } 37. }); 39. } |

使用下面属性配置线程池

|  |
| --- |
| 1. thread: 2. name: '' 3. alive-count: 3 4. max-count: 9 5. alive-time: 3600 # seconds |

或者手动初始化me.libme.kernel.\_c.async.TaskExecutor

|  |
| --- |
| 1. **private** **static** **TaskExecutor** executor; 3. **static** { 4. ThreadConfig threadConfig=**new** ThreadConfig(); 5. threadConfig.setName("test"); 6. threadConfig.setMaxCount(10); 7. threadConfig.setAliveCount(5); 8. threadConfig.setAliveTime(60); 9. SimpleThreadPoolExecutorFactory simpleThreadPoolExecutorFactory= 10. **new** SimpleThreadPoolExecutorFactory(threadConfig); 11. executor=**new** TaskExecutor(simpleThreadPoolExecutorFactory); 12. } |

## 缓存

所有的缓存都继承于me.libme.kernel.\_c.kv.CacheService, 提供简单的HASH 相关的缓存实现。

### 内存缓存

请使用me.libme.kernel.\_c.kv.MapCacheService 处理基于JVM的简单的hash-map 相关的缓存， 注意key的唯一性，示例如下

|  |
| --- |
| 1. **public** **class** MapCacheService<K,V> **implements** CacheService<K,V> { 3. **private** Map<K, Entry<K,V>> repo=**new** ConcurrentHashMap<>(); 5. **private** ScheduledExecutorService executorService=**null**; 7. **private** **final** **boolean** expired; 9. **public** MapCacheService() { 10. **this**(**false**); 11. } 13. **public** MapCacheService(**boolean** expired) { 14. **this**.expired=expired; 15. **if**(**this**.expired){ 16. executorService= Executors.newScheduledThreadPool(1); 17. { 18. executorService.scheduleAtFixedRate(()->{ 19. **for**(Map.Entry<K,Entry<K,V>> entryEntry : repo.entrySet()){ 20. Entry entry=entryEntry.getValue(); 21. **if**((System.currentTimeMillis()-entry.start)>entry.expired){ 22. repo.remove(entry.getKey()); 23. } 24. } 25. },10,500,TimeUnit.MILLISECONDS); 26. } 27. } 28. }  31. **public** **class** Entry<K,V>{ 32. **private** K key; 33. **private** V value; 34. } 35. ... 36. } |

### Ehcache缓存（建议不要使用）

基于Ehcache的缓存实现，示例如下

|  |
| --- |
| 1. **package** me.libme.module.webboot.fn.cache.ehcache; 3. **import** me.libme.module.webboot.fn.\_Cfg; 4. **import** org.springframework.beans.factory.annotation.Autowired; 5. **import** org.springframework.cache.Cache.ValueWrapper; 6. **import** org.springframework.cache.ehcache.EhCacheCacheManager; 8. **import** java.util.concurrent.TimeUnit; 10. **public** **class** \_EhCacheService **implements** EhCacheService<String,Object> { 12. **private** org.springframework.cache.Cache ehcache; 14. @Autowired 15. **private** \_Cfg cfg; 17. @Autowired 18. **public** **void** setEhcache(EhCacheCacheManager ehCacheCacheManager) { 19. **this**.ehcache = ehCacheCacheManager.getCache(cfg.getEhcache().getName()); 20. } |

配置

|  |
| --- |
| 1. ehcache: 2. name: dev |

Spring相关的配置

|  |
| --- |
| 1. spring: 2. cache: 3. ehcache: 4. config : classpath:ehcache.xml |

### Redis缓存

基于Redis的缓存实现，示例如下

|  |
| --- |
| 1. **package** me.libme.module.webboot.fn.cache.redis; 3. **import** me.libme.module.webboot.fn.\_Cfg; 4. **import** org.springframework.beans.factory.annotation.Autowired; 5. **import** org.springframework.data.redis.core.BoundValueOperations; 6. **import** org.springframework.data.redis.core.RedisTemplate; 8. **import** java.util.concurrent.TimeUnit; 10. **public** **class** \_RedisCacheService<K,V> **implements** RedisCacheService<K,V> { 12. **public** \_RedisCacheService(){ 13. System.out.println("--=-"); 14. } 16. @Autowired(required=**false**) 17. **private** RedisTemplate<K, V> redisTemplate ; 19. @Autowired 20. **private** \_Cfg cfg; 22. @Override 23. **public** V expire(K key, V object) { 24. **return** expire(key, object, cfg.getRedis().getExpiredTime(), TimeUnit.SECONDS); 25. } |

配置

|  |
| --- |
| 1. redis: 2. expired-time: 3600  #seconds 3. redis-off-proxy: **true** # 这个配置已经没有作用了，不用管，留着就好 |

Spring相关的配置

|  |
| --- |
| 1. spring: 2. redis: 3. host: redis.3cgg.rec 4. port: 6379 5. password: 123 |

## 字典信息获取

使用me.libme.module.webboot.fn.assist.codetable.CodeDictService获取字典信息，参考

|  |
| --- |
| 1. @Autowired 2. private CodeDictService codeDictService; 4. @ResponseBody 5. @RequestMapping(value = "/codetable",method = RequestMethod.GET) 6. public ResponseModel codetable(@RequestParam("code") String code,@RequestParam("type") String type)  { 7. Object value=codeDictService.val(code,type); 8. return ResponseModel.newSuccess(value); 9. } |

## 系统配置信息获取

使用me.libme.module.webboot.fn.assist.sysparam.SysParamDictService获取系统配置信息，参考

|  |
| --- |
| 1. @Autowired 2. private SysParamDictService sysParamDictService; 4. @ResponseBody 5. @RequestMapping(value = "/sysparam",method = RequestMethod.GET) 6. public ResponseModel sysparam(@RequestParam String name)  { 8. Object value=sysParamDictService.val(name); 10. return ResponseModel.newSuccess(value); 11. } |

## 微服务间调用数据标准

微服务间通过JSON返回数据（me.libme.module.webboot.ResponseModel），使用me.libme.scc.kernel.web.Response能够快速反序列化简单的数据，参考

|  |
| --- |
| 1. String response=kernelService.getUsersByPage("3cgg",0,10); 2. LOGGER.info(response); 3. Page<? extends Map> page= Response.page(response, new TypeReference<List<HashMap>>() { 4. }); 5. LOGGER.info(JJSON.get().format(page)); 6. Page<User> userPage= Response.page(response, new TypeReference<List<User>>() { 7. }); 8. LOGGER.info(JJSON.get().format(userPage)); |

## 用户认证

使用基于JWT的简单用户认证机制，通过提供统一API接口来处理相关JWT业务，相关的类有

* me.libme.module.webboot.fn.jwt.JWTService
* me.libme.module.webboot.fn.\_controller.jwt.JWTValidationAspect
* me.libme.module.webboot.fn.jwt.JWTClaims

示例如下

|  |
| --- |
| 1. public class AclManagerController { 3. @Autowired 4. private UserManagerService userManagerService; 6. @Autowired 7. private SecurityService securityService;  10. @Autowired 11. private JWTService jwtService; 13. @Autowired 14. private JWTClaims jwtClaims; 16. @NoAuthentication 17. @NoAuthorization 18. @ResponseBody 19. @RequestMapping(path="/login",method= RequestMethod.POST) 20. public ResponseModel login(String userName,String password,HttpServletRequest request) throws Exception{ 22. boolean verify=userManagerService.verify(userName,password); 23. if(verify){ 24. UserRecord userRecord=userManagerService.getUserByName(userName); 26. SessionUser sessionUser=new SessionUser(); 27. sessionUser.setId(userRecord.getId()); 28. sessionUser.setUserName(userRecord.getUserName()); 29. sessionUser.setNatureName(userRecord.getNatureName()); 31. Map<String, Object> claims=jwtClaims.claims(sessionUser, null); 32. String token=jwtService.generateToken(claims); 33. return ResponseModel.newSuccess(token); 34. } 35. return ResponseModel.newBysError("Invalid User");  38. } |

你可以通过配置（cpp.jwt.expiration）来定义JWT的时效

|  |
| --- |
| 1. jwt: 2. secret: jiazhongjin 3. expiration: 3600000000 #milliseconds  ; long long time , only for developing 4. header: \_token |

## Mock

请使用注解me.libme.module.mock.Mock标注你需要返回的模型，当前支持mock单实体，mock实体列表，mock分页实体，参考源码了解更多，示例如下

|  |
| --- |
| 1. @RequestMapping(path="/one",method=RequestMethod.GET) 2. @ResponseBody 3. **@Mock(type=\_MockSessionUser.class)** 4. public ResponseModel one() { 5. \_MockSessionUser sessionUser = sessionUser(); 6. return ResponseModel.newSuccess(sessionUser); 7. } 9. @RequestMapping(path="/array",method=RequestMethod.GET) 10. @ResponseBody 11. **@Mock(type=\_MockSessionUser[].class)** 12. public ResponseModel array() { 13. \_MockSessionUser sessionUser = sessionUser(); 14. return ResponseModel.newSuccess(sessionUser); 15. } 17. @RequestMapping(path="/page",method=RequestMethod.GET) 18. @ResponseBody 19. **@Mock(type=\_MockSessionUser[].class , pageable=true)** 20. public ResponseModel page() { 21. \_MockSessionUser sessionUser = sessionUser(); 22. return ResponseModel.newSuccess(sessionUser); 23. } 25. @RequestMapping(path="/remote",method=RequestMethod.GET) 26. @ResponseBody 27. **@UriMock(uri = "sessionUser")** 28. public ResponseModel remote() { 29. \_MockSessionUser sessionUser = sessionUser(); 30. return ResponseModel.newSuccess(sessionUser); 31. } |

## 路由

当前采用Zuul作为微服务的路由组件。

## 用户权限认证/授权

权限在**路由模块（zuul）**中实现，参考配置文件application-urlmapping.yml，通过模块名来配置权限，格式为cpp.url.mapping.{模块名}.{角色名},路径配置正则表达式。如果需要过滤相应的URL，也可以配置exclude项（exclude为全局配置，应用于所有的模块）

模块名必须为在zuul路由中配置的名称，参考下面

|  |
| --- |
| 1. #直接配置URL路由的方式，用不上Ribbon，也用不上Hystrix，Zuul转发所使用的过滤器是 2. #org.springframework.cloud.netflix.zuul.filters.route.SimpleHostRoutingFilter ， 3. #在这个过滤器中，Zuul使用Apache HttpClient进行转发。??????????????????好像不太对 5. routes: 6. module-demo: 7. path: /scc/demo/\*\* 8. serviceId: module-demo 9. module-kernel: 10. path: /scc/kernel/\*\* 11. serviceId: module-kernel 12. module-cms: 13. path: /scc/cms/\*\* 14. serviceId: module-cms 15. module-file: 16. path: /scc/file/\*\* 17. serviceId: module-file |

**application-urlmapping.yml**

|  |
| --- |
| 1. cpp : 2. url : 3. mapping : 4. module-kernel : 5. { 6. ADMIN :  [/sysparam/getSysParamsByPage,/usermanager/getUsersByPage,.\*] 7. , 8. DEFAULT :  [/sysparam/getSysParamsByPage,/usermanager/getUsersByPage] 9. } 10. module-file : 11. { 12. ADMIN :  [/file/.\*,.\*] 13. , 14. DEFAULT :  [/file/.\*,.\*] 15. } 16. module-demo : 17. { 18. ADMIN :  [/newuser/.\*,.\*] 19. , 20. DEFAULT :  [/newuser/page,.\*] 21. } 22. module-exchange : 23. { 24. DEFAULT :  [/httpApi/.\*] 25. } 26. module-exchange-default : 27. { 28. DEFAULT :  [.\*] 29. } 30. exclude : 31. - ^/portal/.\* 32. - ^/codetablemanager/.\* 33. - ^/menumanager/.\* 34. - ^/sysctl/.\* 35. - swagger 36. - /v2/.\* 37. - /[\_]mock[\_]/.\* 38. - ^/aclmanager/login 39. - ^/[\_]refresh[\_]/.\* |

## SHAPE缓存

Shape通过唯一编码标识缓存到redis服务器中，涉及到的类

* me.libme.scc.kernel.fn.shapecache.ShapeInfo
* me.libme.scc.kernel.fn.shapecache.ShapeCacheBridgeService
* me.libme.scc.kernel.fn.shapecache.ShapeInfoCache

|  |
| --- |
| 1. package me.libme.scc.kernel.fn.shapecache; 3. import me.libme.kernel.\_c.\_m.Page; 4. import me.libme.kernel.\_c.\_m.Pageable; 5. import me.libme.module.webboot.fn.loader.IName; 7. /\*\* 8. \* Created by J on 2018/12/7. 9. \*/ 10. public interface ShapeCacheBridgeService extends IName { 12. Page<? extends ShapeInfo> shapeInfo(Pageable pageable); 14. default Page<? extends ShapeInfo> shapeInfo(String param, Pageable pageable){throw new UnsupportedOperationException();} 16. } |

使用方法参考 ***me.libme.scc.demo.service.shape.RShape***

|  |
| --- |
| 1. package me.libme.scc.demo.service.shape; 3. /\*\* 4. \* 5. \* Created by J on 2018/12/9. 6. \*/ 7. @Component 8. public class RShape implements **ShapeCacheBridgeService** { 9. /\* 10. { 11. "me.libme.scc.demo.service.shape.RShape":{ 12. "name":"J" 13. } 14. } 15. \*/ 16. @Override 17. public Page<? extends ShapeInfo> shapeInfo(String param, Pageable pageable) { 18. if(pageable.pageNumber()==0){ 19. List<DemoShap> demoShaps=new ArrayList<>(); 20. DemoShap demoShape=new DemoShap(); 21. demoShape.setCode("123456"); 22. demoShape.setShape("123456-shape"); 23. demoShape.setUuid("123456"); 24. demoShaps.add(demoShape); 25. return PageUtil.wrap(demoShaps,1,pageable); 26. } 27. return null; 28. } 30. @Override 31. public Page<? extends ShapeInfo> shapeInfo(Pageable pageable) { 32. return shapeInfo(null,pageable); 33. } 34. } |

通过me.libme.scc.kernel.fn.shapecache.ShapeInfoCache获取Shape信息，参考

|  |
| --- |
| 1. package me.libme.scc.demo.service.shape; 3. /\*\* 4. \* Created by J on 2018/7/11. 5. \*/ 6. @RestController 7. public class ShapeController { 9. private static final Logger LOGGER= LoggerFactory.getLogger(ShapeController.class); 11. @Autowired 12. private **ShapeInfoCache** shapeInfoCache; 14. @RequestMapping(value = "/shape",method = RequestMethod.GET) 15. public ResponseModel shapeInfo(@RequestParam String code) { 16. ShapeInfo shapeInfo=shapeInfoCache.shapeInfo(code); 17. DemoShap demoShap=new DemoShap(); 18. demoShap.setUuid(shapeInfo.uuid()); 19. demoShap.setShape(shapeInfo.shape()); 20. demoShap.setCode(shapeInfo.code()); 21. return ResponseModel.newSuccess(demoShap); 22. }     28. } |

## 共享注册中心（DEBUG）

如果需要共享注册中心，且需要在本地Debug你的服务，你需要在application-extension.yml文件中配置你自己机器上启动的微服务，参考

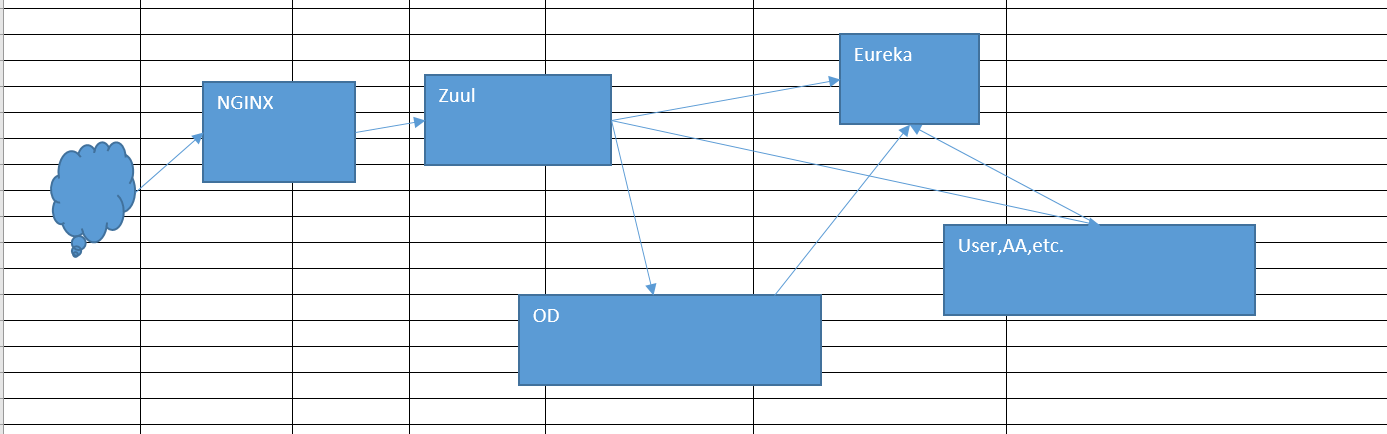
**application-extension.yml**

|  |
| --- |
| 1. #如果你期望本地的微服务响应请求，配置下面的Rule， 这个文件的改动不能提交到服务器上面，切记切记。。。。。。 2. #第一行是微服务的名称，如果多个微服务，复制重写一段，参考下面 4. module-kernel: 5. ribbon: 6. NFLoadBalancerRuleClassName: \_exclude\_in\_spring\_startup.SccZoneAvoidanceRule 7. # 8. #module-demo: 9. #  ribbon: 10. #    NFLoadBalaerRuleClassName: \_exclude\_in\_spring\_startup.SccZoneAvoidanceRule |

## 用户、角色、资源、权限服务模块

请参考包me.libme.webseed.\_b.\_core下的模块，包含用户，角色，资源，菜单，字典，系统参数等模块，参考下图

# 部署



# 总结

此框架是简单，清晰地服务化架构，剥离复杂多变的视图展示、请求上下文绑定，更专注于服务提供者的角色，适应于生产者-消费者开发模型。