springboot简介和springboot的基本用法

1、springboot简介

1.1、Spring的缺点

虽然Spring的组件代码是轻量级的，但它的配置却是重量级的。一开始，Spring用XML配置，而且是很多XML配置。Spring 2.5引入了基于注解的组件扫描，这消除了大量针对应用程序自身组件的显式XML配置。Spring 3.0引入了基于Java的配置，这是一种类型安全的可重构配置方式，可以代替XML。

所有这些配置都代表了开发时的损耗。因为在思考Spring特性配置和解决业务问题之间需要进行思维切换，所以编写配置挤占了编写应用程序逻辑的时间。和所有框架一样，Spring实用，但与此同时它要求的回报也不少。

除此之外，项目的依赖管理也是一件耗时耗力的事情。在环境搭建时，需要分析要导入哪些库的坐标，而且还需要分析导入与之有依赖关系的其他库的坐标，一旦选错了依赖的版本，随之而来的不兼容问题就会严重阻碍项目的开发进度。 1.jsp中要写很多代码、控制器过于灵活,缺少一个公用控制器 2.Spring不支持分布式,这也是EJB仍然在用的原因之一。

正因为spring在使用过程中有很多缺点，所以springboot就应运而生了

1.2、什么是SpringBoot

Spring Boot 是所有基于 Spring 开发的项目的起点。Spring Boot 的设计是为了让你尽可能快的跑起来 Spring 应用程序并且尽可能减少你的配置文件。简单来说就是SpringBoot其实不是什么新的框架，它默认配置了很多框架的使用方式，就像maven整合了所有的jar包，spring boot整合了所有的框架。

Spring Boot 是用来简化 Spring 的搭建和开发过程的全新框架。Spring Boot 去除了大量的 xml 配置文件，简化了复杂的依赖管理，配合各种 starter 使用，基本上可以做到自动化配置。Spring 可以做的事情，现在用 Spring boot 都可以做。

1.3、SpringBoot四个主要特性

1、SpringBoot Starter：他将常用的依赖分组进行了整合，将其合并到一个依赖中，这样就可以一次性添加到项目的Maven或Gradle构建中；

2、自动配置：SpringBoot的自动配置特性利用了Spring4对条件化配置的支持，合理地推测应用所需的bean并自动化配置他们；

3、命令行接口：（Command-line-interface, CLI）：SpringBoot的CLI发挥了Groovy编程语言的优势，并结合自动配置进一步简化Spring应用的开发；

4、Actuatir：它为SpringBoot应用的所有特性构建一个小型的应用程序。但首先，我们快速了解每项特性，更好的体验他们如何简化Spring编程模型。

1.4、SpringBoot的优点

1、创建项目非常快，几秒钟就可以搭建完成； 2、快速集成插件，Spring Boot 提供了入门依赖添加，用于快速的集成各种框架； 3、内置了 Web 容器(嵌入的Tomcat)，运行更加方便； 4、提供了很多监控的指标； 5、可以完全使用代码的方式进行开发，而不需要 XML配置； 6、简化Maven 配置 7、自动配置Spring

1.5、SpringBoot的核心功能

1、起步依赖 本质上是一个Maven项目对象模型（Project Object Model，POM），定义了对其他库的传递依赖，这些东西加在一起支持某项功能。简单的说，起步依赖就是将具备某种功能的坐标打包到一起，并提供一些默认的功能。

2、SpringBoot的自动配置是一个运行时（更准确地说，是应用程序启动时）的过程，考虑了众多因素，才决定Spring配置应该用哪个，不该用哪个。该过程是Spring自动完成的。Spring可以自动配置一个bean和其他协作bean之间的关系。

2、springboot的使用

2.1、springboot工程搭建

一个springboot工程的搭建其实很简单

1、导入必要的jar包，如：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.3.1.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
```

Web启动器

```
<!-- web 启动器 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <!--排除Tomcat启动器，如果用jetty需要排除，如果要打包(war)部署到服务器需要排除内置Tomcat -->
    <!--            <exclusions>
                    <exclusion>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-tomcat</artifactId>
                    </exclusion>
                </exclusions>-->
</dependency>
```

2、启动类

```
@SpringBootApplication
@MapperScan("com.gp.wy.dao")
@ServletComponentScan(basePackages = {"com.gp.wy"})
@EnableConfigurationProperties(DruidDataSourceProperty.class)
public class SpringbootTest extends SpringBootServletInitializer {

    /*
    * 1、要完成Spring容器的启动
    * 2、把项目部署到tomcat
```

```
     * */
    public static void main(String[] args) {
        ConfigurableApplicationContext applicationContext =
SpringApplication.run(SpringbootTest.class,
                args);
    }

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {
        return builder.sources(SpringBootTest.class);
    }
}
```

注解解释：

@SpringBootApplication(scanBasePackages = {**"com.gp.wy"**}) 扫描包下面的注解并把类实例化加入到spring容器中

@MapperScan(**"com.gp.wy.dao"**) 扫描该包，把dao包下的接口生成代理实例

@ServletComponentScan(basePackages = {**"com.gp.wy"**})

扫描@*WebFilter***，@WebListener，@WebServlet

注解，把Servlet、Filter、Listener加入到spring容器中、

@EnableConfigurationProperties(DruidConfig.**class**) 开启配置文件读取功能，DruidConfig类必须要有

@ConfigurationProperties(prefix = **"spring.druid"**,ignoreInvalidFields = **true**)

注解，该注解会读取默认的配置文件application.properties配置，然后会读取**spring.druid**作为前缀的配置属性。

2.2、springboot整合Servlet、Filter、Listener

有的时候需要在springboot工程里面加入Servlet、Filter和Listener，这时候只需要在启动类上加入：

@ServletComponentScan(basePackages = {"com.gp.wy"})注解，然后在Servlet、Filter、Listener上面加入相应注解即可。如：

```
@WebServlet(urlPatterns = "/jack/*")
public class JackServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
```

```
@WebFilter(urlPatterns = "/*",filterName = "myFilter")
public class MyFilter implements Filter {
    @Override
    public void doFilter(ServletRequest request, ServletResponse
        System.out.println("--------MyFilter----------");
        chain.doFilter(request,response);
    }
}
```

```java
@WebListener
public class MyListener implements ServletContextListener {

    @Override
    public void contextDestroyed(ServletContextEvent contextEvent) {
        System.out.println("contextDestroyed"); }
    @Override
    public void contextInitialized(ServletContextEvent contextEvent) {
        System.out.println("contextInitialized"); }
}
```

2.3、springboot整合druid

Druid是一个非常优秀的连接池，非常好的管理了数据库连接，可以实时监控数据库连接对象和应用程序的数据库操作记录，如何在springboot中整合druid呢？可以分为如下几个步骤

1、jar包导入

```xml
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.0.26</version>
</dependency>
```

2、druid配置

数据库连接配置加载

```java
@Data
@Configuration
@ConfigurationProperties(prefix = "spring.druid",ignoreInvalidFields = true)
public class DruidConfig {

    private String driverClassName;
    private String jdbcUrl;
    private String jdbcUrl1;
    private String username;
    private String password;
    private int maxActive;
    private int minIdle;
    private int initialSize;
    private Long timeBetweenEvictionRunsMillis;
    private Long minEvictableIdleTimeMillis;
    private String validationQuery;
    private boolean testWhileIdle;
    private boolean testOnBorrow;
    private boolean testOnReturn;
    private boolean poolPreparedStatements;
    private Integer maxPoolPreparedStatementPerConnectionSize;
    private String filters;
    private String connectionProperties;
```

读取application.properties配置文件中以spring.druid作为前缀的配置属性值。

数据源对象创建并加入到spring容器中

```
@Bean(destroyMethod = "close",initMethod = "init")
public DataSource getDs1(){
    DruidDataSource druidDataSource = new DruidDataSource();
    druidDataSource.setDriverClassName(driverClassName);
    druidDataSource.setUrl(jdbcUrl);
    druidDataSource.setUsername(username);
    druidDataSource.setPassword(password);
    druidDataSource.setMaxActive(maxActive);
    druidDataSource.setInitialSize(initialSize);
    druidDataSource.setTimeBetweenConnectErrorMillis(timeBetweenEvictionRunsMillis);
    druidDataSource.setMinEvictableIdleTimeMillis(minEvictableIdleTimeMillis);
    druidDataSource.setValidationQuery(validationQuery);
    druidDataSource.setTestWhileIdle(testWhileIdle);
    druidDataSource.setTestOnBorrow(testOnBorrow);
    druidDataSource.setTestOnReturn(testOnReturn);
    druidDataSource.setPoolPreparedStatements(poolPreparedStatements);
    druidDataSource.setMaxPoolPreparedStatementPerConnectionSize(maxPoolPreparedStatementPerConnectionSize);

    try {
        druidDataSource.setFilters(filters);
    } catch (SQLException e) {
        e.printStackTrace();
    }

    return druidDataSource;
}
```

3、Druid数据监控配置

```
@Bean
public ServletRegistrationBean druidStateViewServlet(){
    ServletRegistrationBean servletRegistrationBean = new ServletRegistrationBean(new StatViewServlet(), ...urlMa
    //初始化参数initParams
    //添加白名单
    servletRegistrationBean.addInitParameter( name: "allow", value: "");
    //添加ip黑名单
    servletRegistrationBean.addInitParameter( name: "deny", value: "192.168.16.111");
    //登录查看信息的账号密码
    servletRegistrationBean.addInitParameter( name: "loginUsername", value: "admin");
    servletRegistrationBean.addInitParameter( name: "loginPassword", value: "123");
    //是否能够重置数据
    servletRegistrationBean.addInitParameter( name: "resetEnable", value: "false");
    return servletRegistrationBean;
}

/**
 * 过滤不需要监控的后缀
 * @return
 */
@Bean
public FilterRegistrationBean druidStatFilter(){
    FilterRegistrationBean filterRegistrationBean = new FilterRegistrationBean(new WebStatFilter());
    //添加过滤规则
    filterRegistrationBean.addUrlPatterns("/*");
    //添加不需要忽略的格式信息
    filterRegistrationBean.addInitParameter( name: "exclusions", value: "*.js,*.gif,*.jpg,*.png,*.css,*.ico,/druid/
    return filterRegistrationBean;
}
```

2.4、springboot整合mybatis

Mybatis是一个非常优秀的ORM框架，公司里面用得也比较多，下面我们来在springboot中整合一下mybatis框架，可以分为如下几个步骤：

1、jar包导入

```
<!-- 把mybatis的启动器引入 -->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>RELEASE</version>
</dependency>
```

2、application.properties配置

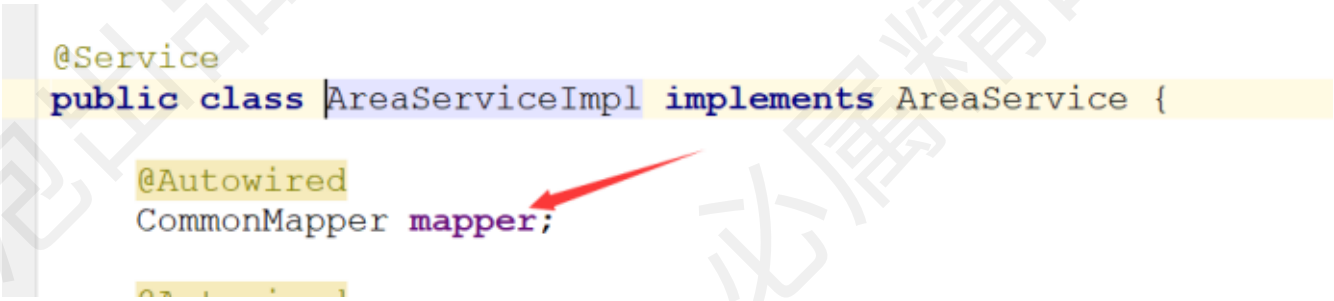
```
#把该包下的bean生成别名

mybatis.typeAliasesPackage=com.gp.wy.bean

 #Mybatis会把这个路径下的xml解析出来建立接口的映射关系
mybatis.mapperLocations=classpath:com/gp/wy/xml/\*Mapper.xml
```

3、启动类添加扫描注解

```
@MapperScan("com.gp.wy.dao")
```

4、业务使用

该注解会扫描dao包下的接口，把接口生成代理对象并加入到spring容器中，在业务代码里面就可以按照类型注入了。如图：



2.5、springboot整合JPA

JPA也是一个非常优秀的ORM框架，用起来也非常简单

1、jar包导入

```
<!-- Spring Boot JPA -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

2、application.properties配置

自动创建表

spring.jpa.hibernate.ddl-auto:**update**打印sql语句

spring.jpa.show-sql:**true**

3、实例bean

```java
@Data
@Entity
@Table(name = "xx_student")
public class Student {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column(name = "name")
    private String name;

    @Column(name = "cardNum")
    private String cardNum;
}
```

当启动的时候会自动生成xx_student表

4、dao配置

```java
/*
* Student操作对象
* Integer主键类型
*
* */
public interface StudentDao extends JpaRepository<Student,Integer> {

}
```

泛型中指定了操作的实体bean是哪一个，JpaRepository中定义了该实体的增删改查的所有方法，用studentDao对象调用即可

5、业务代码中使用

```java
@Service
public class StudentServiceImpl implements StudentService {

    @Autowired
    private StudentDao studentDao;

    @Override
    public List<Student> findAll() { return studentDao.findAll(); }

    @Override
    public Student findById(Integer id) {

        //
        Optional<Student> byId = studentDao.findById(id);

        if(byId.isPresent()) {
            return byId.get();
        }
        return null;
    }
```

2.6、springboot整合redis

redis在业务中也经常用到，那么我们看看springboot如何整合redis的

1、jar包导入

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

2、application.properties配置

```properties
# Redis数据库索引（默认为0）
spring.redis.database=0
# Redis服务器地址
spring.redis.host=192.168.67.139
# Redis服务器连接端口
spring.redis.port=6379
# Redis服务器连接密码（默认为空）
spring.redis.password=
# 连接池最大连接数（使用负值表示没有限制）
spring.redis.pool.max-active=8
# 连接池最大阻塞等待时间（使用负值表示没有限制）
spring.redis.pool.max-wait=-1
# 连接池中的最大空闲连接
spring.redis.pool.max-idle=8
# 连接池中的最小空闲连接
spring.redis.pool.min-idle=0
# 连接超时时间（毫秒）
spring.redis.timeout=0
```

3、把redis整合到spring的缓存体系中

CacheManager对象创建

```
//缓存管理器
@Bean
public CacheManager cacheManager(RedisConnectionFactory redisConnectionFactory) {
    RedisCacheConfiguration redisCacheConfiguration =
RedisCacheConfiguration.defaultCacheConfig()
            .entryTtl(Duration.ofHours(1)); // 设置缓存有效期一小时
    return RedisCacheManager
            .builder(RedisCacheWriter.nonLockingRedisCacheWriter(redisConnectionFactory))
            .cacheDefaults(redisCacheConfiguration).build();
}
```

创建redis连接对象

```
    @Bean
    public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory factory) {
        RedisTemplate<String, Object> template = new RedisTemplate<>();
        // 配置连接工厂
        template.setConnectionFactory(factory);
        //使用Jackson2JsonRedisSerializer来序列化和反序列化redis的value值（默认使用JDK的序列化方
式）
        Jackson2JsonRedisSerializer jacksonSeial = new
Jackson2JsonRedisSerializer(Object.class);
        ObjectMapper om = new ObjectMapper();
        // 指定要序列化的域，field,get和set,以及修饰符范围，ANY是都有包括private和public
        om.setVisibility(PropertyAccessor.ALL, JsonAutoDetect.Visibility.ANY);
        // 指定序列化输入的类型，类必须是非final修饰的，final修饰的类，比如String,Integer等会跑出异常
        om.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FINAL);
        jacksonSeial.setObjectMapper(om);
        // 值采用json序列化
        template.setValueSerializer(jacksonSeial);
        //使用StringRedisSerializer来序列化和反序列化redis的key值
        template.setKeySerializer(new StringRedisSerializer());
        // 设置hash key 和value序列化模式
        template.setHashKeySerializer(new StringRedisSerializer());
        template.setHashValueSerializer(jacksonSeial);
        template.afterPropertiesSet();
        return template;
    }
```

4、在代码中的使用

```
@Test
public void redisTemplate() {
    redisTemplate.opsForValue().setIfAbsent("username","wy");
    System.out.println(redisTemplate.opsForValue().get("username"));
}
```

2.7、springboot整合mongodb

mongodb也是一款非常优秀的nosql数据库，springboot也提供了整合方案

1、jar包导入

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

2、application.properties配置

spring.data.mongodb.uri=mongodb://192.168.67.139:27017/aa_db

3、代码案例

```
@Service
public class MongoServiceImpl implements MongoService<User> {
    /*
    * 这个实例是从哪里来的？？
    * */
    @Autowired
    private MongoTemplate mongoTemplate;

    @PostConstruct
    public void test() {
        System.out.println(mongoTemplate);
    }

    @Override
    public String save(User obj) {
        mongoTemplate.save(obj);
        return "1";
    }

    @Override
    public List<User> findAll() {
        List<User> all = mongoTemplate.findAll(User.class);
        return all;
    }

    @Override
    public User getById(String id) {
        Query query = new Query(Criteria.where("_id").is(id));
        return mongoTemplate.findOne(query,User.class);
    }

    @Override
    public User getByName(String name) {
        Query query = new Query(Criteria.where("username").is(name));
        return mongoTemplate.findOne(query,User.class);
    }
```

```
    /*
    * update xx set() where xx =a
    * */
    @Override
    public String updateE(User user) {
        Query query = new Query(Criteria.where("_id").is(user.getId()));
        Update update = new Update().set("username", user.getUsername()).set("password",
user.getPassword());
        UpdateResult updateResult = mongoTemplate.updateFirst(query, update, User.class);
        return updateResult.toString();
    }

    @Override
    public String deleteE(User user) {
        DeleteResult remove = mongoTemplate.remove(user);
        return remove.toString();
    }

    @Override
    public String deleteById(String id) {
        User user = getById(id);
        String cacheResult = deleteE(user);
        return cacheResult;
    }

    @Override
    public List<User> findLikes(String reg) {
        Pattern pattern = Pattern.compile("^.*" + reg + ".*$" , Pattern.CASE_INSENSITIVE);
        Query query = Query.query(Criteria.where(reg).regex(pattern));
        List<User> users = mongoTemplate.find(query, User.class);
        return users;
    }
}
```

mongoTemplate对象是可以直接依赖注入进来的，由启动器创建了该类的对象。

4、测试案例

```
@Test
public void mongoTest() {
    User user = new User();
    user.setUsername("jack");
    user.setPassword("123");
    user.setId("87");
    mongoService.save(user);
    System.out.println(mongoService.findAll());
}
```

2.8、springboot整合JAX-RS规范

JAX-RS是JAVA EE6 引入的一个新技术。 JAX-RS即Java API for RESTful Web Services，是一个Java 编程语言的应用程序接口，支持按照表述性状态转移（REST）架构风格创建Web服务。JAX-RS使用了Java SE5引入的Java注解来简化Web服务的客户端和服务端的开发和部署。

其实就是类似于Servlet规范来接收用户请求的一个规范。

1、jar包导入

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jersey</artifactId>
</dependency>
```

2、代码配置

```
@Configuration
public class JerseyConfig {
    @Bean
    public ServletRegistrationBean jerseyServlet() {
        //手动注册servlet
        ServletRegistrationBean registrationBean = new ServletRegistrationBean(new
ServletContainer(), "/rest/*");

 registrationBean.addInitParameter(ServletProperties.JAXRS_APPLICATION_CLASS,JerseyResource
Config.class.getName());
        return registrationBean;
    }
}
```

手动注册了一个Servlet，拦截的路径是/rest/*

定义Jersey的资源加载类JerseyResourceConfig，如下

```
public class JerseyResourceConfig extends ResourceConfig {

    public JerseyResourceConfig() {

        register(RequestContextFilter.class);

        // 加载资源文件,这里直接扫描com.gp.wy.jersey下的所有api   JAX-RS
        packages("com.gp.wy.jersey");
    }
}
```

Packages方法是扫描包的路径，扫描jersey规范中相应的注解。如：

```
/*
* 可以接受http请求
* */
@Path("/jersey/")
public class JerseyController {

    @Path("{id}")
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public String hello(@PathParam("id") Long id) {
        return "hello";
    }
}
```

2.9、springboot整合swagger2

swagger2是一个可以根据接口定义自动生成接口API文档的框架

1、jar包导入

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.9.2</version>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.9.2</version>
</dependency>
```

2、代码配置

```
@Configuration
@EnableSwagger2
public class SwaggerConfig {
    @Bean
    public Docket createRestApi() {
        return new Docket(DocumentationType.SWAGGER_2)
                .pathMapping("/")
                .select()
                .apis(RequestHandlerSelectors.basePackage("com.gp.wy.controller"))
                .paths(PathSelectors.any())
                .build().apiInfo(new ApiInfoBuilder()
                        .title("xx公司API文档")
                        .description("xx公司API文档")
                        .version("9.0")
                        .contact(new Contact("wy", "blog.csdn.net", "aaa@gmail.com"))
                        .license("The Apache License")
                        .licenseUrl("http://www.baidu.com")
                        .build());
    }
```

```
}
```

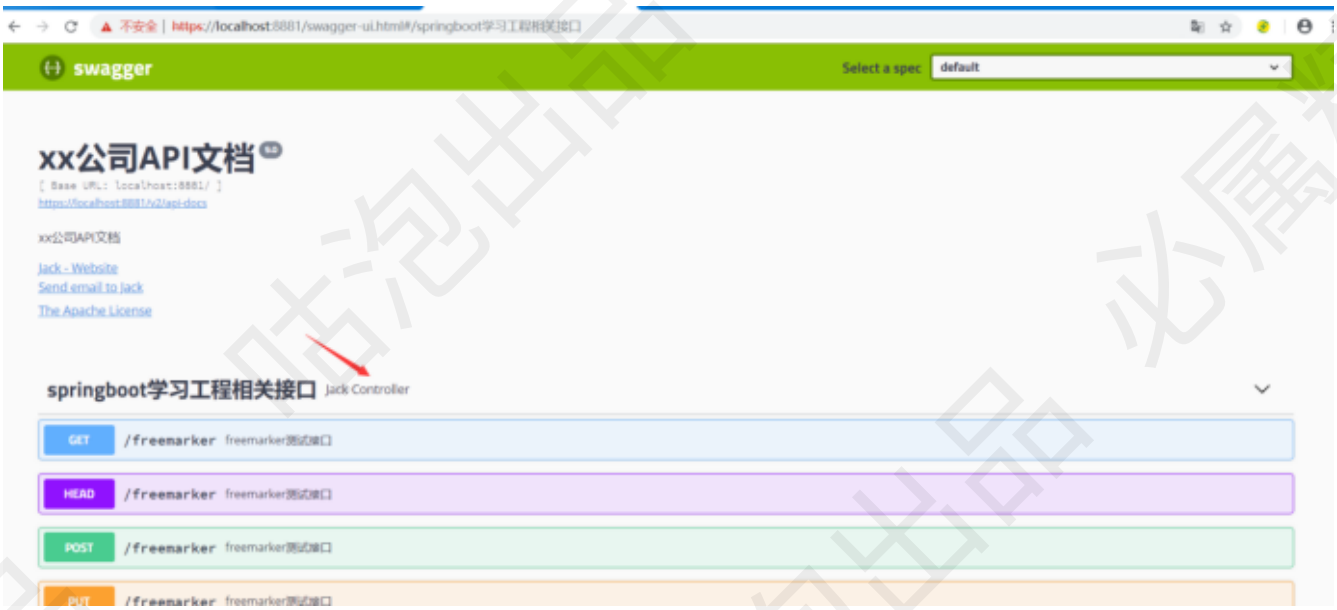
3、具体使用

```
@Controller
@Api(tags = "springboot学习工程相关接口")
public class WYController {

    private static final Logger logger = LoggerFactory.getLogger(JackController.class);

    @Autowired
    AreaService areaService;

    @Value("${application.field:default value jack}")
    private String zhuguangField = "";

    @ApiOperation("jsp测试接口")
    @ApiImplicitParams({
            @ApiImplicitParam(name = "username", value = "用户名", defaultValue = "jack"),
            @ApiImplicitParam(name = "address", value = "用户地址", defaultValue = "长沙")
    })
    @RequestMapping("/index")
    public ModelAndView index() {
        ModelAndView mv = new ModelAndView();
        mv.setViewName("index");
        mv.addObject("time", new Date());
        mv.addObject("message", zhuguangField);
        return mv;
    }

    @ApiOperation("freemarker测试接口")
    @ApiImplicitParams({
            @ApiImplicitParam(name = "map", value = "返回值")
    })
    @RequestMapping("/freemarker")
    public String freemarker(Model model) {

        List<User> list = new ArrayList<>();
        list.add(new User(1, "张飒飒", 25));
        list.add(new User(2, "李四四", 26));
        list.add(new User(3, "王五五", 23));
        list.add(new User(4, "赵六六", 24));

        // 需要一个Model对象
        model.addAttribute("list", list);
        return "user/user";
    }

    @ApiOperation("查询地区接口")
    @ApiImplicitParams({
            @ApiImplicitParam(name = "param", value = "地区编码")
    })
    @RequestMapping("/queryArea")
```

```
    public @ResponseBody String queryArea(String param) {
        List<ConsultConfigArea> areas = areaService.qryArea(new HashMap());
        for (ConsultConfigArea area : areas) {
            logger.info(area.getAreaCode() + "    " + area.getAreaName() + "    "
                    + area.getState());
        }
        return "OK";
    }

    @RequestMapping("/testDevTool")
    public @ResponseBody String testDevTool() {
        return "OK";
    }

    @RequestMapping("/testDevTool1")
    public @ResponseBody String testDevTool1() {
        return "OK";
    }

    @RequestMapping("/queryUser")
    public @ResponseBody String queryUser() {
        return "wuya";
    }
}
```

4、ui界面

界面请求url：*https://localhost:8881/swagger-ui.html*



2.10、springboot整合Actuator监控管理

Actuator监控是一个用于监控springboot健康状况的工具，可以实时的工程的健康和调用情况

1、jar包导入

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

2、application.properties配置

actuator默认只开放两个接口分别是：

http://localhost:8881/actuator/health

http://localhost:8881/actuator/info

如果想开放监控的所有界面，需要在application.properties进行配置

#**默认只有**info/health

management.endpoints.web.exposure.include=*

这个监控不需要使用，是自动统计消息的。

1 /health/{component}/{instance} GET

报告程序的健康指标，这些数据由HealthIndicator实现类提供

2 /info GET

获取程序指定发布的信息，这些信息由配置文件中info打头的属性提供

3 /configprops GET

描述配置属性（包含默认值）如何注入到bean

4 /beans GET

描述程序中的bean，及之间的依赖关系

5 /env GET

获取全部环境属性

6 /env/{name} GET

根据名称获取指定的环境属性值

7 /mappings GET

描述全部的URI路径，及和控制器的映射关系

8 /metrics/{requiredMetricName} GET

统计程序的各种度量信息，如内存用量和请求数

9 /httptrace GET

提供基本的http请求跟踪信息，如请求头等

10 /threaddump GET

获取线程活动的快照

11 /conditions GET

提供自动配置报告，记录哪些自动配置通过，哪些没有通过

12 /loggers/{name} GET

查看日志配置信息

13 /auditevents GET

查看系统发布的事件信息

14 /caches/{cache} GET/DELETE

查看系统的缓存管理器，另可根据缓存管理器名称查询；另DELETE操作可清除缓存

15 /scheduledtasks GET

查看系统发布的定时任务信息

16 /features GET

查看Springcloud全家桶组件信息

17 /refresh POST

重启应用程序，慎用

18 /shutdown POST

关闭应用程序，慎用

2.11、springboot整合https

1、生成https证书

cd到jdk的bin目录，执行生成证书的指令：

keytool -genkey -alias spring -keypass 123456 -keyalg RSA -keysize 1024 -validity 365 -keystore E:/springboot.keystore -storepass 123456

genkey 表示要创建一个新的密钥。

alias 表示 keystore 的别名。

keyalg 表示使用的加密算法是 RSA，一种非对称加密算法。

keysize 表示密钥的长度。

keystore 表示生成的密钥存放位置。

validity 表示密钥的有效时间，单位为天。

正式开发过程中，需要申请正式的被浏览器信任的证书

2、application.properties配置

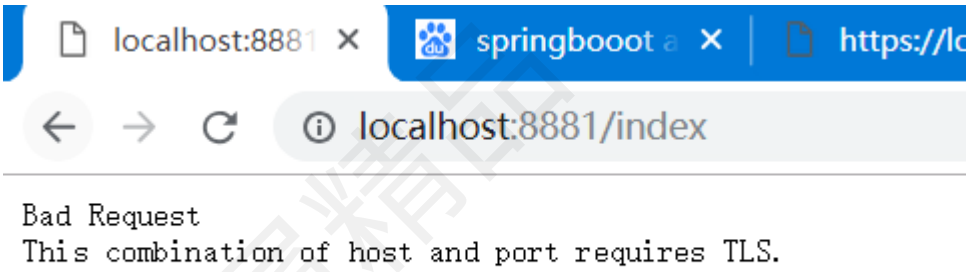
server.port=8881

server.ssl.key-password=123456

server.ssl.key-store=classpath:springboot.keystore

server.ssl.key-alias=spring

如果是http的方式访问则会报错



2.12、SpringBoot工程docker化

微服务项目用docker来部署和管理是非常方便和节省资源的，所有springboot项目也有docker话的需求，springboot项目docker是通过maven插件来实现的：

1、docker化插件

```
<plugin>
    <groupId>com.spotify</groupId>
    <artifactId>docker-maven-plugin</artifactId>
    <version>0.4.13</version>
    <configuration>
        <imageName>jack/micro</imageName>
        <dockerDirectory>${project.basedir}/src/main/docker</dockerDirectory>
        <resources>
            <resource>
                <targetPath>/</targetPath>
                <directory>${project.build.directory}</directory>
                <include>${project.build.finalName}.jar</include>
            </resource>
        </resources>
    </configuration>
</plugin>
```

2、Dockerfile文件

FROM docker.io/relateiq/oracle-java8

VOLUME /tmp

ADD springboot-web.jar app.jar

#RUN bash -c 'touch /app.jar'

ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]

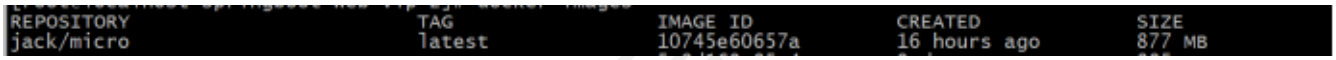
EXPOSE 8881

3、把工程上传到linux环境并打包生成工程镜像

在pom.xml文件目录下执行指令

mvn package docker:build

这个指令就会根据Dockerfile的内容来生成镜像



然后根据镜像来启动容器

docker run -ti -d -p 8881:8881 --name springboot jack/micro

查看容器启动日志

docker logs -f springboot

这样springboot工程镜像化就完成了。

2.13、SpringBoot整合rabbitmq

1、jar包导入

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

2、application配置

```
spring.rabbitmq.host=192.168.88.139
spring.rabbitmq.port=5672
spring.rabbitmq.username=admin
spring.rabbitmq.password=admin
```

3、代码配置类

```java
@Configuration
public class RabbitmqConfig {

    @Bean(name = "message")
    public Queue queueMessage() {
        /*
         * 这个步骤就是往rabbitmq的broker里面创建一个队列
         * */
        return new Queue( name: "jack.message");
    }
    /*
     * 创建交换器
     * */
    @Bean
    public TopicExchange exchange() {
        return new TopicExchange( name: "exchange.message");
    }

    @Bean
    Binding bindingExchangeMessage(@Qualifier("message") Queue queueMessage,
                                   TopicExchange exchange) {
        return BindingBuilder.bind(queueMessage).to(exchange).
                with( routingKey: "jack.message.routeKey");
    }
}
```

4、消息发送和消费

消息发送

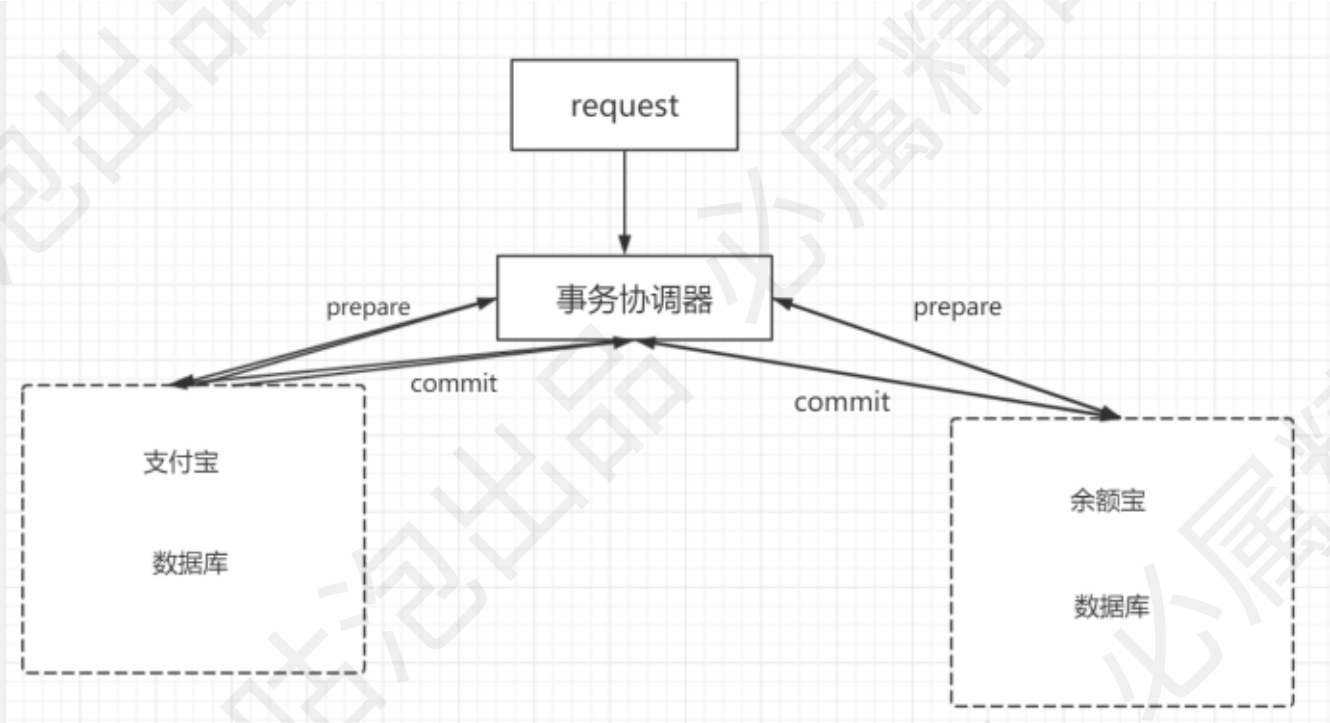
```java
@Slf4j
@Component
public class RabbitmqSender {

    @Autowired
    private AmqpTemplate amqpTemplate;

    public void sendMessage(String exchange,String roukekey,Message content) {
        try {
            log.info("============发送消息给余额宝=========" + content);
            amqpTemplate.convertAndSend(exchange, roukekey,
                    JSONObject.toJSONString(content));
        }catch (Exception e) {

        }
    }
}
```

消息消费

```java
@Slf4j
@Component
public class MessageListener {

    @Autowired
    OrderService orderService;

    @RabbitListener(queues = "jack.message.response")
    public void process(final String result) {
        log.info("==============接收到余额宝转账成功的应答消息==============" + result);
        orderService.updateMessage(result);
    }
}
```

2.14、springboot整合atomikos

Atomikos是一个基于XA协议的分布式事务解决管理框架，其核心思想就是两段提交，一般使用在在一个业务方法里面涉及到的对多个数据源的操作，要保证在这个业务方法操作中，保持对两个数据源的同时提交和同时回滚。如图所示：



Atomikos就是图中的事务协调器的角色，负责对两个数据源的管理，同时提交或同时回滚。

就是在真正提交之前有一个预提交的过程，就是检测两个数据源是否能够提交，如果有一个返回No，那么这个事务就不能提交。具体配置：

1、jar包导入

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-jta-atomikos</artifactId>
</dependency>
```

2、两个数据源连接信息配置

```
# Mysql 1
mysql.datasource.test1.url = jdbc:mysql://192.168.67.139:3307/zg?
useUnicode=true&characterEncoding=utf-8
mysql.datasource.test1.username = root
mysql.datasource.test1.password = 123456
mysql.datasource.test1.minPoolSize = 3
mysql.datasource.test1.maxPoolSize = 25
mysql.datasource.test1.maxLifetime = 20000
mysql.datasource.test1.borrowConnectionTimeout = 30
mysql.datasource.test1.loginTimeout = 30
mysql.datasource.test1.maintenanceInterval = 60
mysql.datasource.test1.maxIdleTime = 60

# Mysql 2
mysql.datasource.test2.url =jdbc:mysql://192.168.67.139:3306/zg?
useUnicode=true&characterEncoding=utf-8
mysql.datasource.test2.username =root
mysql.datasource.test2.password =123456
mysql.datasource.test2.minPoolSize = 3
mysql.datasource.test2.maxPoolSize = 25
mysql.datasource.test2.maxLifetime = 20000
mysql.datasource.test2.borrowConnectionTimeout = 30
mysql.datasource.test2.loginTimeout = 30
mysql.datasource.test2.maintenanceInterval = 60
mysql.datasource.test2.maxIdleTime = 60
```

3、数据源创建

```
@Configuration
@MapperScan(basePackages = "com.gp.wy.atomikos.db1.dao", sqlSessionFactoryRef =
"test1SqlSessionFactory",sqlSessionTemplateRef="test1SqlSessionTemplate")
public class Db1Config {

    @Autowired
    DBConfig1 testConfig;

    @Bean(name = "test1DataSource")
    public DataSource testDataSource() {
        MysqlXADataSource mysqlXaDataSource = new MysqlXADataSource();
        mysqlXaDataSource.setUrl(testConfig.getUrl());
        mysqlXaDataSource.setPinGlobalTxToPhysicalConnection(true);
        mysqlXaDataSource.setPassword(testConfig.getPassword());
        mysqlXaDataSource.setUser(testConfig.getUsername());
        mysqlXaDataSource.setPinGlobalTxToPhysicalConnection(true);

        AtomikosDataSourceBean xaDataSource = new AtomikosDataSourceBean();
        xaDataSource.setXaDataSource(mysqlXaDataSource);
        xaDataSource.setUniqueResourceName("test1DataSource");
        xaDataSource.setMinPoolSize(testConfig.getMinPoolSize());
        xaDataSource.setMaxPoolSize(testConfig.getMaxPoolSize());
        xaDataSource.setMaxLifetime(testConfig.getMaxLifetime());
        xaDataSource.setBorrowConnectionTimeout(testConfig.getBorrowConnectionTimeout());
        try {
```

```
            xaDataSource.setLoginTimeout(testConfig.getLoginTimeout());
        } catch (SQLException e) {
            e.printStackTrace();
        }
        xaDataSource.setMaintenanceInterval(testConfig.getMaintenanceInterval());
        xaDataSource.setMaxIdleTime(testConfig.getMaxIdleTime());
        xaDataSource.setTestQuery(testConfig.getTestQuery());
        return xaDataSource;
    }

    @Bean(name = "test1SqlSessionFactory")
    public SqlSessionFactory testSqlSessionFactory(@Qualifier("test1DataSource") DataSource
dataSource)
            throws Exception {
        SqlSessionFactoryBean bean = new SqlSessionFactoryBean();
        bean.setDataSource(dataSource);
        return bean.getObject();
    }

    @Bean(name = "test1SqlSessionTemplate")
    public SqlSessionTemplate testSqlSessionTemplate(
            @Qualifier("test1SqlSessionFactory") SqlSessionFactory sqlSessionFactory)
throws Exception {
        return new SqlSessionTemplate(sqlSessionFactory);
    }
}
```

```
@Configuration
@MapperScan(basePackages = "com.gp.wy.atomikos.db2.dao", sqlSessionFactoryRef =
"test2SqlSessionFactory",sqlSessionTemplateRef="test2SqlSessionTemplate")
public class Db2Config {

    @Autowired
    DBConfig2 testConfig;

    @Bean(name = "test2DataSource")
    public DataSource testDataSource() {
        MysqlXADataSource mysqlXaDataSource = new MysqlXADataSource();
        mysqlXaDataSource.setUrl(testConfig.getUrl());
        mysqlXaDataSource.setPinGlobalTxToPhysicalConnection(true);
        mysqlXaDataSource.setPassword(testConfig.getPassword());
        mysqlXaDataSource.setUser(testConfig.getUsername());
        mysqlXaDataSource.setPinGlobalTxToPhysicalConnection(true);

        AtomikosDataSourceBean xaDataSource = new AtomikosDataSourceBean();
        xaDataSource.setXaDataSource(mysqlXaDataSource);
        xaDataSource.setUniqueResourceName("test2DataSource");
        xaDataSource.setMinPoolSize(testConfig.getMinPoolSize());
        xaDataSource.setMaxPoolSize(testConfig.getMaxPoolSize());
        xaDataSource.setMaxLifetime(testConfig.getMaxLifetime());
        xaDataSource.setBorrowConnectionTimeout(testConfig.getBorrowConnectionTimeout());
        try {
            xaDataSource.setLoginTimeout(testConfig.getLoginTimeout());
```

```
        } catch (SQLException e) {
            e.printStackTrace();
        }
        xaDataSource.setMaintenanceInterval(testConfig.getMaintenanceInterval());
        xaDataSource.setMaxIdleTime(testConfig.getMaxIdleTime());
        xaDataSource.setTestQuery(testConfig.getTestQuery());
        return xaDataSource;
    }

    @Bean(name = "test2SqlSessionFactory")
    public SqlSessionFactory testSqlSessionFactory(@Qualifier("test2DataSource") DataSource
dataSource)
            throws Exception {
        SqlSessionFactoryBean bean = new SqlSessionFactoryBean();
        bean.setDataSource(dataSource);
        return bean.getObject();
    }

    @Bean(name = "test2SqlSessionTemplate")
    public SqlSessionTemplate testSqlSessionTemplate(
            @Qualifier("test2SqlSessionFactory") SqlSessionFactory sqlSessionFactory)
throws Exception {
        return new SqlSessionTemplate(sqlSessionFactory);
    }
}
```

4、具体代码案例

```
@Service
public class AreaServiceImpl implements AreaService {

    @Autowired
    private CommonMapper1 commonMapper1;

    @Autowired
    private CommonMapper2 commonMapper2;

    @Autowired
    TransactionManager transactionManager;


    @Transactional
    public int saveArea(ConsultConfigArea area) {
        int count = commonMapper1.addArea(area);
        int count1 = commonMapper2.addArea(area);
        return count;
    }
}
```