

SpringBoot 高级整合篇

零. 前言-总结:

```
1  SpringBoot 高级
2
3  1. 缓存 (Spring缓存抽象) : @EnableCaching, @CacheConfig, @Caching, @Cacheable, @CachePut,
   @CacheEvict, 缓存原理, 自定义key的生成策略类
4  2. 整合redis 缓存: 使用docker (连接国内的镜像) 安装并开启redis, 使用redis的客户端测试连接远程redis
   服务器是否ok, 然后工程导入redis的dependency, 配置文件配置redis; 自定义redis的序列化规则类;
5      自定义RedisCacheManager, 使用自定义的序列化规则      ; 使用编码或者注解两种方式往redis缓存中放入
   数据
6
7  3. 消息中间件: JMS, AMQP, RabbitMQ
8      1. 异步处理
9      2. 应用解耦
10     3. 流量削峰
11 4. 整合RabbitMQ: docker pull registry.docker-cn.com/library/rabbitmq:3-management; docker run
   -d -p 5672:5672 -p 15672:15672 --name myrabbitmq c51d1c73d028
12     1. 进入到rabbitmq的管理界面, 添加三个不同类型的exchange (交换器) : direct, fanout, topic
13     2. 创建四个消息队列 (cris, cris.news, cris.emps, zc.news), 并且和交换器绑定, 测试交换器和队列之间
   通信规则
14     3. SpringBoot 工程整合RabbitMQ, 发送和取出rabbitmq服务器队列里消息 (可以自定义json类型的数据转
   换器, 这样发送到rabbitmq服务器的消息自动转换为json类型)
15     4. 模拟应用解耦情景, 使用rabbitmq的监听器注解 (这里需要开启@EnableRabbit注解)
16     5. 使用 编码的方式来创建exchange和queue, binding (绑定规则)
17
18 5. ElasticSearch: docker 下载elasticSearch并启动: docker run -e ES_JAVA_OPTS="-Xms256m -
   Xmx256m" -d -p 9200:9200 -p 9300:9300 --name ES01 0ba66712c1f9
19     - 基础概念: es集群, 索引 (google, apple), 类型 (Employee, product), 文档 (张三, 牙刷), id
20     - 使用restful格式的请求向elasticsearch服务器存储数据 (json) put请求, 获取数据 (get请求), 判断
   资源是否存在 (head请求), 删除数据 (delete请求), 更新数据 (put请求)
21     - 使用postman进行测试
22     - _search 搜索所有员工数据; 通过json格式的搜索信息进行强大的各种搜索 (重点就是查询表达式)
23     - springboot 整合elasticsearch
24     - 两种方式操作es: 1.jest 2. springdata
25     - jest客户端存储文档信息到es服务器; jest客户端通过查询表达式从es服务器获取数据
26     - 通过springboot 提供的ElasticSearchRepository 接口或者ElasticSearchTemplate 类 来进行数据
   的检索
27
28 6.SpringBoot 和任务管理
29     - 异步任务 (@EnableAsync, @Async) : springboot开启另外一个线程异步处理任务
30     - 定时任务: (@EnableScheduling, @Scheduled) : 定时表达式
31     - 邮件任务: 导入starter, 使用JavaMailSenderImpl 发送一封简单的邮件; 发送一封复杂的邮件 (引入
   html代码和附件)
32
33 7.SpringBoot 与安全
34     - Shiro和Spring security
35     - 搭建thymeleaf 环境和整合页面资源
36     - 使用spring security 完成用户认证和授权 (引入starter, 编写spring security的配置类); 自定义
```

授权规则，自定义用户认证（用户名和密码，roles）

- 完成用户注销功能
- 整合thymeleaf 引擎模板为spring security 框架准备的jar 包（使用高级功能：判断用户身份显示页面信息和对应的权限模块）
- 记住我和定制登录页（难点，重点）

8. SpringBoot 和分布式（绝对重点）

- Docker 下载zookeeper和启动 (docker run --name zk1 -p 2181:2181 --restart always -d 56d414270ae3)
- SpringBoot 整合Dubbo（重点）；空工程里面创建两个模块（买票和用户模块）；彼此之间通过Dubbo 通信；服务提供模块引入dubbo和zookeeper，然后配置文件配置dubbo相关信息，最后发布出去
- 消费者模块同样引入dubbo和zookeeper，同时引入服务者的接口（@Reference）进行服务的远程调用
- 关于版本问题（SpringBoot2.x和1.5.x版本有些还是差别比较大，比如SpringBoot2.x底层使用的是Spring5.x版本，我们再写springMVC的拦截器的时候就需要将静态资源放行，而在SpringBoot1.5.x版本中就不需要，并且通过dubbo注册服务到zookeeper中心两个大版本之间也有区别，这里使用1.5.x版本可以完成dubbo和zookeeper的整合，但是2.x版本就需要另外设置了）

9. SpringBoot 和 SpringCloud（一站式分布解决方案）：

- 配置和启动Eureka 服务注册中心（配置文件和@EnableEurekaServer 开启服务）
- 完成服务提供模块的配置和书写，并且使用不同的端口通过maven的package命令打包多个jar包，通过java -jar 的cmd窗口运行不同的服务模块的jar包
- 完成消费者模块（配置文件的书写和@EnableDiscoveryClient, @LoadBalanced , RestTemplate)

10. SpringBoot 开发热部署（Devtools）

- 导入响应的启动器即可（并且针对java代码和html代码均有效，只需要ctrl+f9 即可，炒鸡方便，eclipse则是直接ctrl+s 保存即可）

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
</dependency>
```

11. SpringBoot 监控管理（应用监控和管理功能）

- 引入actuator启动器
- 配置文件进行授权（management.security.enabled=false）
- 监控端点测试
- 定制端点信息
- health 端点监控应用组件的健康状态（自定义组件健康状态监控器类）

一. SpringBoot 及其缓存抽象

1. 引入对应的starter(web,cache,mybatis,mysql,druid)

```
1 <dependencies>
2 <dependency>
3 <groupId>org.springframework.boot</groupId>
4 <artifactId>spring-boot-starter-cache</artifactId>
5 </dependency>
6 <dependency>
```

```

7         <groupId>org.springframework.boot</groupId>
8         <artifactId>spring-boot-starter-web</artifactId>
9     </dependency>
10    <dependency>
11        <groupId>org.mybatis.spring.boot</groupId>
12        <artifactId>mybatis-spring-boot-starter</artifactId>
13        <version>1.3.2</version>
14    </dependency>
15
16    <dependency>
17        <groupId>mysql</groupId>
18        <artifactId>mysql-connector-java</artifactId>
19        <scope>runtime</scope>
20    </dependency>
21    <dependency>
22        <groupId>org.springframework.boot</groupId>
23        <artifactId>spring-boot-starter-test</artifactId>
24        <scope>test</scope>
25    </dependency>
26    <!-- https://mvnrepository.com/artifact/com.alibaba/druid -->
27    <dependency>
28        <groupId>com.alibaba</groupId>
29        <artifactId>druid</artifactId>
30        <version>1.1.6</version>
31    </dependency>
32 </dependencies>

```

2. application.yml

```

1  spring:
2    datasource:
3    # 数据源基本配置
4    username: root
5    password: 123456
6    driver-class-name: com.mysql.jdbc.Driver
7    url: jdbc:mysql://120.78.138.11:3306/spring_cache?useSSL=false
8    type: com.alibaba.druid.pool.DruidDataSource
9    # 数据源其他配置
10   initialSize: 5
11   minIdle: 5
12   maxActive: 20
13   maxWait: 60000
14   timeBetweenEvictionRunsMillis: 60000
15   minEvictableIdleTimeMillis: 300000
16   validationQuery: SELECT 1 FROM DUAL
17   testWhileIdle: true
18   testOnBorrow: false
19   testOnReturn: false
20   poolPreparedStatements: true
21   # 配置监控统计拦截的filters, 去掉后监控界面sql无法统计, 'wall'用于防火墙
22   filters: stat,wall,log4j
23
24   maxPoolPreparedStatementPerConnectionSize: 20

```

```

24     useGlobalDataSourceStat: true
25     connectionProperties: druid.stat.mergeSql=true;druid.stat.slowSqlMillis=500
26     redis:
27         host: 120.78.138.11
28
29
30     # 直接将mybatis的配置写在总配置文件中（开启驼峰命名规则）
31     mybatis:
32         configuration:
33             map-underscore-to-camel-case: true
34
35     logging:
36         level: debug
37
38     # 如果使用mybatis的配置文件来配置mybatis以及sql映射文件，就必须指定文件的位置
39     #mybatis:
40     #     config-location: classpath:/mybatis/mybatis-config.xml
41     #mapper-locations: classpath:/mybatis/mapper/*.xml
42
43     logging.level.com.cris.springboot.mapper=debug # 这行配置写在application.properties

```

3. 配置druid 数据源

```

1  /**
2   * @ClassName DruidConfig
3   * @Description 配置Druid 数据源信息，方便监控
4   * @Author zc-cris
5   * @Version 1.0
6   */
7  @Configuration
8  public class DruidConfig {
9      @ConfigurationProperties(prefix = "spring.datasource")
10     @Bean
11     public DataSource druid() {
12         return new DruidDataSource();
13     }
14
15     //配置Druid的监控
16     //1、配置一个管理后台的Servlet
17     @Bean
18     public ServletRegistrationBean statViewServlet() {
19         ServletRegistrationBean bean = new ServletRegistrationBean(new StatViewServlet(),
20             "/druid/*");
21         Map<String, String> initParams = new HashMap<>();
22
23         initParams.put("loginUsername", "admin");
24         initParams.put("loginPassword", "123456");
25         initParams.put("allow", ""); //默认就是允许所有访问
26         //         initParams.put("deny", "");
27
28         bean.setInitParameters(initParams);
29
30         return bean;
31     }

```

```

29     }
30
31
32     //2、配置一个web监控的filter
33     @Bean
34     public FilterRegistrationBean webStatFilter() {
35         FilterRegistrationBean bean = new FilterRegistrationBean();
36         bean.setFilter(new WebStatFilter());
37
38         Map<String, String> initParams = new HashMap<>();
39         initParams.put("exclusions", "*.js,*.css,/druid/*");
40
41         bean.setInitParameters(initParams);
42
43         bean.setUrlPatterns(Arrays.asList("/*"));
44
45         return bean;
46     }
47 }

```

4. javaBean

```

1 public class Employee implements Serializable{
2
3     private Integer id;
4     private String lastName;
5     private String email;
6     private Integer gender; //性别 1男 0女
7     private Integer dId;
8     ....
9 }

```

5. Mapper

```

1 // 这里使用注解版来标识mapper每个方法的执行sql, 实际开发中基本使用mapper映射文件
2 public interface EmployeeMapper {
3
4     // 插入数据完成后, 将自增长的id值又放入当前department 对象中
5     @Options(useGeneratedKeys = true, keyProperty = "id")
6     @Insert("insert into employee (lastName, email, gender, d_id) values ({lastName}, #
7     {email}, #{gender}, #{dId})")
8     public int saveEmp(Employee employee);
9
10    @Delete("delete from employee where id = #{id}")
11    public int removeEmpById(Integer id);
12
13    @Update("update employee set lastName = #{lastName}, email = #{email}, gender = #
14    {gender}, d_id = #{dId} where id = #{id}")
15    public int updateEmp(Employee employee);
16
17    @Select("select id, lastName, email, gender, d_id from employee where id = #{id}")
18
19 }

```

```

16     public Employee getEmpById(Integer id);
17
18     @Select("select id, lastName, email, gender, d_id from employee where lastName = #
{lastName}")
19     Employee getEmpByLastName(String lastName);
20 }
21

```

6. 测试druid 数据源

```

1  @RunWith(SpringRunner.class)
2  @SpringBootTest
3  public class SpringbootCacheApplicationTests {
4
5      @Autowired
6      EmployeeMapper employeeMapper;
7
8      // 测试能否从远程服务器的mysql中获取数据 (通过Druid数据源)
9      @Test
10     public void contextLoads() {
11         Employee emp = employeeMapper.getEmpById(1);
12         System.out.println(emp);
13     }
14 }

```

7. 主启动类

```

1  /**
2   * @Author zc-cris
3   * @Description 1. 创建数据库 2. 创建对应的javaBean 3. 创建数据源 (Druid) 4. 使用Mybatis (注解扫描) 4. 使用缓存
4   * ①. 开启基于注解的缓存 ②. 标注缓存注解即可(@Cacheable, @CacheEvict, @CachePut)
5   * @Param
6   * @return
7   */
8  @MapperScan("com.cris.springboot.mapper")
9  @SpringBootApplication
10 @EnableCaching
11 public class SpringbootCacheApplication {
12
13     public static void main(String[] args) {
14         SpringApplication.run(SpringbootCacheApplication.class, args);
15     }
16 }

```

8. service(重点)

```

1  /**
2   * @ClassName EmployeeService
3   * @Description TODO

```

```

4  * @Author zc-cris
5  * @Version 1.0
6  **/
7  // 对于缓存的一些通用配置可以使用@CacheConfig 注解完成，例如通用的cache组件等
8  @CacheConfig(cacheNames = "emp", cacheManager = "EmpCacheManager")
9  @Service
10 public class EmployeeService {
11
12     @Autowired
13     EmployeeMapper employeeMapper;
14
15     /**
16      * @return com.cris.springboot.bean.Employee
17      * @Author zc-cris
18      * @Description 将方法的运行结果放入缓存，下次使用相同的数据直接从缓存中取，不再调用方法
19      * - CacheManager 管理多个Cache 组件，对缓存的真正CRUD 操作在Cache组件中，每一个缓存组件都有一个唯一的名字
20      * - 默认会配置一个SimpleCacheConfiguration；给容器中注册了一个CacheManager：
21      ConcurrentMapCacheManager；
22      * 可以获取和创建ConcurrentMapCache 类型的缓存组件：将数据保存在ConcurrentMap 中
23      * <p>
24      * - 运行流程：@Cacheable
25      * 1. 方法运行前：先去查询Cache（缓存组件），按照cacheNames 指定的名字获取（CacheManager先获取到对应的缓存组件，第一次获取组件如果没有就自动创建）
26      * 2. 去Cache组件中查找缓存的内容，默认key就是方法的参数：
27      * key是按照某种策略生成的：默认使用SimpleKeyGenerator 生成key
28      * SimpleKeyGenerator生成key 的默认策略：
29      * 如果没有参数：key=new SimpleKey();
30      * 如果有一个参数：key=参数的值
31      * 如果有多个参数：key=new SimpleKey(params)
32      * 3. 没有查询到数据就调用目标方法（所以目标方法不一定会执行）
33      * 4. 将目标方法的返回值放入到缓存组件中
34      * <p>
35      * - 流程总结：1. 使用CacheManager【ConcurrentMapCacheManager】按照名字得到Cache组件
36      【ConcurrentMapCache】
37      * 2. key使用keyGenerator生成，默认是SimpleKeyGenerator（我们可以自定义）
38      * <p>
39      * - 核心属性：
40      * 1. cacheNames/value:指定缓存组件的名字
41      * 2. key: 缓存数据用的key，默认使用方法的参数值；可以使用SpEl；#id: 参数id的值；#a0 #p0
42      #root.args[0]
43      * 3. keyGenerator:key的生成器，可以自定义
44      * key/keyGenerator ：二选一
45      * 4. cacheManager: 指定缓存管理器；或者使用cacheResolver指定解析器，两者选一个即可
46      * 5. condition: 指定符合条件才缓存数据，condition = "#id > 0"
47      * 6. unless:否定缓存，unless 指定的条件为true，方法的返回值就不会被缓存，可以获取到结果进行判断
48      * unless = "#result == null"
49      * unless = "#a0 == 2":如果第一个参数的值为2，那么不缓存查询出来的数据
50      * 7. sync: 是否使用异步模式（默认为false）：如果为true，unless属性不支持
51      * @Param [id]
52      **/
53
54     // @Cacheable(cacheNames = {"emp"}, key = "#root.methodName + '[' + #id + ']'"

```

```

51     @Cacheable(cacheNames = {"emp"} /*, keyGenerator = "myKeyGenerator", condition = "#a0 >
1", unless = "#a0 == 2"*/)
52     public Employee getEmp(Integer id) {
53         System.out.println("查询" + id + "号员工...");
54         return employeeMapper.getEmpById(id);
55     }
56
57     /**
58      * @return com.cris.springboot.bean.Employee
59      * @Author zc-cris
60      * @Description @CachePut:先调用方法更新数据库数据, 然后更新缓存, 达到了同步更新缓存的目的,
        保证后面查询对应的数据都是缓存中最新的数据
61      * 运行时机: 和@Cacheable 不同, 一定要先调用目标方法; 然后将目标方法的返回结果缓存起来
62      * 注意: 缓存数据的key需要和@Cacheable 指定的key相同, 保证更新和查询的数据都是同一个key指定的
        数据
63      * key = "#employee.id":使用传入的employee对象的id
64      * key = "#result.id":使用返回后的employee对象的id
65      * 但是@Cacheable 中的key不能使用#result (因为是在目标方法执行前进行调用, 拿不到目标方法的返
        回值)
66      * @Param [employee]
67      */
68     @CachePut(value = "emp", key = "#result.id")
69     public Employee updateEmp(Employee employee) {
70         employeeMapper.updateEmp(employee);
71         return employee;
72     }
73
74     /**
75      * @return void
76      * @Author zc-cris
77      * @Description @CacheEvict 代表缓存清除
78      * key:指定要清除的缓存中的数据的key
79      * allEntries = true: 指定清除环村组件中的所有缓存; 默认为false
80      * beforeInvocation = false: 缓存的清除是否在目标方法执行之前进行
81      * 默认为false, 即缓存清除在方法执行后进行; 如果方法出现异常, 那么缓存清理失败
82      * 如果为true: 代表缓存清理是在目标方法执行前就进行了, 所以缓存肯定会清空
83      * @Param [id]
84      */
85     @CacheEvict(/*value = "emp", */key = "#id"/*, beforeInvocation = true, allEntries =
        true*/)
86     public void deleteEmp(Integer id) {
87         // 打印这句话代表数据已经被删除
88         System.out.println("deleteEmp" + id);
89         // employeeMapper.removeEmpById(id);
90         // int i = 10/0;
91     }
92
93     /**
94      * @return com.cris.springboot.bean.Employee
95      * @Author zc-cris
96      * @Description 多个缓存注解可以联合使用, 以适应更加复杂的缓存需求
97      * @Param [lastName]
98      */

```



```

99     @Caching(cacheable = {
100         @Cacheable(/*value = "emp",*/ key = "#lastName")
101     },
102     put = {
103         @CachePut(/*value = "emp",*/ key = "#result.id"),
104         @CachePut(/*value = "emp",*/ key = "#result.email")
105     })
106     public Employee getEmpByLastName(String lastName) {
107         return employeeMapper.getEmpByLastName(lastName);
108     }
109 }

```

9. controller

```

1  /**
2   * @ClassName EmployeeController
3   * @Description TODO
4   * @Author zc-cris
5   * @Version 1.0
6   **/
7  @RestController
8  public class EmployeeController {
9
10     @Autowired
11     EmployeeService employeeService;
12
13     @GetMapping("/emp/{id}")
14     public Employee getEmp(@PathVariable("id") Integer id) {
15         return employeeService.getEmp(id);
16     }
17
18     @GetMapping("/emp")
19     public Employee updateEmp(Employee employee) {
20         employeeService.updateEmp(employee);
21         return employee;
22     }
23
24     @GetMapping("/delEmp/{id}")
25     public void deleteEmp(@PathVariable("id") Integer id) {
26         employeeService.deleteEmp(id);
27     }
28
29     @GetMapping("/emp/lastName/{lastName}")
30     public Employee getEmpByLastName(@PathVariable("lastName") String lastName) {
31         return employeeService.getEmpByLastName(lastName);
32     }
33 }

```

10. 自定义缓存key生成器

```

1  /**

```

```

2  * @ClassName MyCacheConfig
3  * @Description TODO
4  * @Author zc-cris
5  * @Version 1.0
6  **/
7  // 自定义的缓存key生成策略类
8  @Configuration
9  public class MyCacheConfig {
10
11      @Bean("myKeyGenerator")
12      public KeyGenerator keyGenerator() {
13          return new KeyGenerator() {
14              @Override
15              public Object generate(Object target, Method method, Object... params) {
16                  return method.getName() + "[" + Arrays.asList(params).toString() + "];
17              }
18          };
19      }
20  }

```

二. SpringBoot 与 Redis整合

1. 整合redis 的dependency

```

1      <dependency>
2          <groupId>org.springframework.boot</groupId>
3          <artifactId>spring-boot-starter-data-redis</artifactId>
4      </dependency>

```

2. 主启动类

```

1  /**
2   * @Author zc-cris
3   * @Description 1. 创建数据库 2. 创建对应的javaBean 3. 创建数据源 (Druid) 4. 使用Mybatis (注解扫描) 4. 使用缓存
4   * ①. 开启基于注解的缓存 ②. 标注缓存注解即可 (@Cacheable, @CacheEvict, @CachePut)
5   * 默认使用的是ConcurrentMapCacheManager, 将数据保存到ConcurrentMap中
6   * 开发中常用的则是Redis, memcached, ehcache
7   * <p>
8   * - 整合redis作为缓存
9   * Redis是一个开源的, 内存中的数据结构存储系统, 可以用作数据库, 缓存和消息中间件
10  * 1. 使用docker 安装redis
11  * 2. 工程引入redis的starter
12  * 3. 配置redis
13  * 4. 测试redis
14  * 5. 测试缓存
15  * - 原理:
16  * - CacheManager-->Cache 缓存组件来给实际的缓存中存取数据

```

```

17  * - 引入了redis的starter以后，容器中保存的就是RedisManager 了
18  * - RedisManager 会帮我们创建RedisCache 缓存组件，而RedisCache 又是通过远程的redis服务器来缓存数
    据的
19  * - 默认保存数据k-v都是Object的时候，使用jdk的序列化来保存数据，如果需要将数据转换为json格式再保
    存？
20  * 1. 引入了redis的starter后，cacheManager 就变为了RedisCacheManager
21  * 2. 默认创建的RedisCacheManger 操作Redis的时候使用的是RedisTemplate<Object, Object>
22  * 3. RedisTemplate<Object, Object> 默认使用的是jdk的序列化机制
23  * 4. 自定义cacheManager
24  * @Param
25  * @return
26  **/
27 @MapperScan("com.cris.springboot.mapper")
28 @SpringBootApplication
29 @EnableCaching
30 public class SpringbootCacheApplication {
31
32     public static void main(String[] args) {
33         SpringApplication.run(SpringbootCacheApplication.class, args);
34     }
35 }

```

3. javaBean

```

1 public class Department implements Serializable {
2
3     private Integer id;
4     private String departmentName;
5     ...
6 }

```

4. 自定义Redis 配置(自定义的序列化规则器类RedisTemplate 和 RedisCacheManager)

```

1 /**
2  * @ClassName MyRedisConfig
3  * @Description 自定义Redis的序列化规则（以json格式保存我们的javaBean对象）
4  * @Author zc-cris
5  * @Version 1.0
6  **/
7 @Configuration
8 public class MyRedisConfig {
9
10
11     // 自定义redis的序列化规则
12     @Bean
13     public RedisTemplate<Object, Employee> employeeRedisTemplate(
14         RedisConnectionFactory redisConnectionFactory)
15         throws UnknownHostException {
16         RedisTemplate<Object, Employee> template = new RedisTemplate<Object, Employee>();

```

```

17     template.setConnectionFactory(redisConnectionFactory);
18     template.setDefaultSerializer(new Jackson2JsonRedisSerializer<Employee>
(Employee.class));
19     return template;
20 }
21
22
23 // 自定义redis的cacheManager, 将自定义的序列化规则器类传递进去
24 @Bean
25 @Primary // 如果有多个cacheManager的情况下, 一定要使用@Primary 指定默认的
cacheManager
26 public RedisCacheManager EmpCacheManager(RedisTemplate<Object, Employee>
employeeRedisTemplate) {
27     RedisCacheManager cacheManager = new RedisCacheManager(employeeRedisTemplate);
28     // 使用cache组件的cacheName作为前缀, 方便区分数据
29     cacheManager.setUsePrefix(true);
30     return cacheManager;
31 }
32
33 // 自定义redis的序列化规则
34 @Bean
35 public RedisTemplate<Object, Department> departmentRedisTemplate(
36     RedisConnectionFactory redisConnectionFactory)
37     throws UnknownHostException {
38     RedisTemplate<Object, Department> template = new RedisTemplate<Object, Department>
();
39     template.setConnectionFactory(redisConnectionFactory);
40     template.setDefaultSerializer(new Jackson2JsonRedisSerializer<Department>
(Department.class));
41     return template;
42 }
43
44 // 自定义redis的cacheManager, 将自定义的序列化规则器类传递进去
45 @Bean
46 public RedisCacheManager deptCacheManager(RedisTemplate<Object, Department>
departmentRedisTemplate) {
47     RedisCacheManager cacheManager = new RedisCacheManager(departmentRedisTemplate);
48     // 使用cache组件的cacheName作为前缀, 方便区分数据
49     cacheManager.setUsePrefix(true);
50     return cacheManager;
51 }
52 }

```

5. controller

```

1 /**
2  * @ClassName DepartmentController
3  * @Description TODO
4  * @Author zc-cris
5  * @Version 1.0
6  */
7 @RestController

```

```

8 public class DepartmentController {
9
10     @Autowired
11     DepartmentService departmentService;
12
13     @GetMapping("/dept/{id}")
14     public Department getDept(@PathVariable("id") Integer id) {
15         return departmentService.getDept(id);
16     }
17 }

```

6. mapper

```

1 public interface DepartmentMapper {
2
3     @Select("select * from department where id = #{id}")
4     public Department getDeptById(Integer id);
5 }

```

7. service(重点)

```

1 /**
2  * @ClassName DepartmentService
3  * @Description TODO
4  * @Author zc-cris
5  * @Version 1.0
6  */
7 @Service
8 public class DepartmentService {
9
10     @Autowired
11     DepartmentMapper departmentMapper;
12
13     @Qualifier("deptCacheManager")
14     @Autowired
15     CacheManager deptCacheManager;
16
17     /**
18      * @return com.cris.springboot.bean.Department
19      * @Author zc-cris
20      * @Description 缓存的数据可以存入redis; 但是第二次从缓存中查询却无法反序列化回来, 因为存入dept
21      的json格式的数据
22      * 但是默认使用RedisTemplate<Object, Employee> 序列化类来反序列化
23      * 解决方案: 为每个单独的需要序列化的javaBean对象创建CacheManager 和 cacheTemplate, service层
24      调用dao层方法的时候缓存注解需要注明指定的属性
25      * @Param [id]
26      */
27     // @Cacheable(cacheNames = "dept", cacheManager = "deptCacheManager")
28     // public Department getDept(Integer id){
29     //     return departmentMapper.getDeptById(id);
30     // }

```

```

29     public Department getDept(Integer id) {
30         System.out.println("查询" + id + "号部门");
31         Department dept = departmentMapper.getDeptById(id);
32         // 使用cacheManager 来获取指定的缓存组件
33         Cache cache = deptCacheManager.getCache("dept");
34         // 编码方式将数据放入到缓存组件中
35         cache.put("dept:1", dept);
36         return dept;
37     }
38 }

```

8. 测试

```

1     @Autowired
2     StringRedisTemplate stringRedisTemplate;    // 操作k-v都是字符串
3
4     @Autowired
5     RedisTemplate redisTemplate;    // 操作k-v都是对象的
6
7     @Autowired
8     RedisTemplate<Object, Employee> employeeRedisTemplate; // 自定义序列化规则的redisTemplate
9
10    /**
11     * @return void
12     * @Author zc-cris
13     * @Description Redis 常见的五大数据类型
14     * String, List, Set, Hash, ZSet (有序集合)
15     * @Param []
16     */
17    @Test
18    public void testRedis() {
19        //给远程redis服务器中保存数据
20        // stringRedisTemplate.opsForValue().append("msg", "hello");
21
22        // 从远程redis服务器读取数据
23        String msg = stringRedisTemplate.opsForValue().get("msg");
24        System.out.println(msg);
25
26        stringRedisTemplate.opsForList().leftPush("list", "1");
27        stringRedisTemplate.opsForList().leftPush("list", "2");
28
29    }
30
31    @Test
32    public void testRedis2() {
33        // 保存对象到redis服务器,默认使用jdk的序列化机制,也就是说将序列化后的数据保存到redis
34        // redisTemplate.opsForValue().set("emp-01", employeeMapper.getEmpById(1));
35        // 如果想将数据以json形式保存: 一般有两种方式: 1. 自己将对象转换为json; 2. 自定义redis的序列化规则 (推荐)
36
37        employeeRedisTemplate.opsForValue().set("emp-01", employeeMapper.getEmpById(1));
38    }

```

三. SpringBoot 和消息中间件 RabbitMQ

1. docker 整合 RabbitMQ

```
1 docker pull registry.docker-cn.com/library/rabbitmq:3-management
2 docker run -d -p 5672:5672 -p 15672:15672 --name myrabbitmq c51d1c73d028
```

2. 进入到rabbitMQ的管理界面，添加三个不同类型的exchange（交换器）：direct,fanout,topic

3. 创建四个消息队列（cris,cris.news,cris.emps,zc.news），并且和交换器绑定,测试交换器和队列之间通信规则

4. SpringBoot 工程整合RabbitMQ

① pom.xml

```
1 <dependencies>
2 <dependency>
3 <groupId>org.springframework.boot</groupId>
4 <artifactId>spring-boot-starter-amqp</artifactId>
5 </dependency>
6 <dependency>
7 <groupId>org.springframework.boot</groupId>
8 <artifactId>spring-boot-starter-web</artifactId>
9 </dependency>
10
11 <dependency>
12 <groupId>org.springframework.boot</groupId>
13 <artifactId>spring-boot-starter-test</artifactId>
14 <scope>test</scope>
15 </dependency>
16 </dependencies>
```

② 自定义json类型的数据转换器

```
1 /**
2  * @ClassName MyRabbitConfig
3  * @Description TODO
4  * @Author zc-cris
5  * @Version 1.0
6  */
7 @Configuration
8 public class MyRabbitConfig {
```

```

9
10     @Bean
11     public MessageConverter messageConverter(){
12         return new Jackson2JsonMessageConverter();
13     }
14 }

```

③ javaBean

```

1  /**
2   * @ClassName User
3   * @Description TODO
4   * @Author zc-cris
5   * @Version 1.0
6   **/
7  public class User {
8      private String name;
9      private Integer age;
10     ...
11 }

```

④ service

```

1  /**
2   * @ClassName UserService
3   * @Description TODO
4   * @Author zc-cris
5   * @Version 1.0
6   **/
7  @Service
8  public class UserService {
9
10     @RabbitListener(queues = {"cris.news"})
11     public void receive(User user){
12         System.out.println(user);
13     }
14
15     @RabbitListener(queues = {"cris"})
16     public void receive2(Message message){
17         System.out.println(message.getBody()+"-----消息体"); //消息体
18         System.out.println(message.getMessageProperties()+"-----消息头"); //消息头
19     }
20 }

```

⑤ 主程序类

```

1  /**
2   * @Author zc-cris
3   * @Description 自动配置:
4   *     1. RabbitAutoConfiguration

```



```

5  * 2. 自动配置了连接工厂ConnectionFactory
6  * 3. RabbitProperties: 封装了RabbitMQ的配置信息
7  * 4. RabbitTemplate: 给RabbitMQ发送和接收消息的模板
8  * 5. AmqpAdmin: RabbitMQ的系统管理功能组件
9  * 6. @EnableRabbit + @RabbitListener 开启监听消息队列的消息, 如果消息发送到exchange然后被放入
   queue, 就会被
10 * 对应的监听方法获取到消息
11 * @Param
12 * @return
13 **/
14 @EnableRabbit
15 @SpringBootApplication
16 public class SpringbootAmqpRabbitmqApplication {
17
18     public static void main(String[] args) {
19         SpringApplication.run(SpringbootAmqpRabbitmqApplication.class, args);
20     }
21 }

```

⑥ 测试

```

1  @RunWith(SpringRunner.class)
2  @SpringBootTest
3  public class SpringbootAmqpRabbitmqApplicationTests {
4
5      @Autowired
6      RabbitTemplate rabbitTemplate;
7
8      @Autowired
9      AmqpAdmin amqpAdmin;
10
11     // 通过编码的方式创建exchange, queue, binding
12     @Test
13     public void test2(){
14         // amqpAdmin.declareExchange(new DirectExchange("amqpAdmin.exchange"));
15         // amqpAdmin.declareQueue(new Queue("amqpAdmin.queue", true));
16         // amqpAdmin.declareBinding(new Binding("amqpAdmin.queue",
17         Binding.DestinationType.QUEUE, "amqpAdmin.exchange", "cris521", null));
18         amqpAdmin.deleteQueue("amqpAdmin.queue");
19         amqpAdmin.deleteExchange("amqpAdmin.exchange");
20     }
21
22     /**
23     * @Author zc-cris
24     * @Description 使用springboost 整合rabbitmq的方式来操作rabbitmq消息中间件
25     * 1. 点对点(单播)
26     * @Param []
27     * @return void
28     **/
29     @Test
30     public void contextLoads() {

```

```

31 //      可以自定义message的消息体和消息头
32 //      rabbitTemplate.send(exchange, routeKey,message);
33
34
35 //      object默认作为消息体，只需要传入要发送的对象，自动序列化发送给rabbitmq 服务器
36 //      rabbitTemplate.convertAndSend(exchange, routeKey, Object);
37 Map<String, Object> map = new HashMap<>();
38 map.put("msg", "这个第一个通过springboot 发送过来的消息");
39 map.put("data", Arrays.asList("123", 321, true));
40 //      rabbitTemplate.convertAndSend("exchange.direct", "cris.news", map);
41 // 测试javaBean对象能否以json格式发送到服务器以及从服务器接收 (ok)
42 rabbitTemplate.convertAndSend("exchange.direct", "cris", new User("cris", 21));
43
44 }
45 // 取出rabbitmq 服务器消息队列里面的信息并且转换为java对象(消息队列里的消息：先进先出)
46 @Test
47 public void testReceive(){
48     Object o = rabbitTemplate.receiveAndConvert("cris");
49     System.out.println(o.getClass());    //class java.util.HashMap
50     System.out.println(o);    // {msg=这个第一个通过springboot 发送过来的消息, data=[123,
321, true]}
51 }
52
53 // 测试广播模式的交换器
54 @Test
55 public void test(){
56     rabbitTemplate.convertAndSend("exchange.fanout", "", new User("重庆吴亦凡", 25));
57 }
58 }

```

⑦ 配置文件（应该放在第一步的。。。）

```

1 spring:
2   rabbitmq:
3     host: 120.78.138.11
4     username: guest
5     password: guest
6 #   默认就是5672，可以不写
7     port: 5672
8 #   默认不写就是访问/
9 #   virtual-host: /

```

四. SpringBoot 整合 ElasticSearch

1. Docker 整合 ES


```
1 docker 下载elasticSearch并启动:  
2 docker run -e ES_JAVA_OPTS="-Xms256m -Xmx256m" -d -p 9200:9200 -p 9300:9300 --name ES01  
0ba66712c1f9
```

2. 使用postman 向ES 服务器发送REST形式的请求


- 存储数据 (json) put请求



- 获取数据 (get请求)




GET  http://120.78.138.11:9200/google/employee/1

Body Cookies Headers (3) Test Results

Pretty Raw Preview JSON 


```
1 {
2   "_index": "google",
3   "_type": "employee",
4   "_id": "1",
5   "_version": 1,
6   "found": true,
7   "_source": {
8     "first_name": "John",
9     "last_name": "Smith",
10    "age": 25,
11    "about": "I love to go rock climbing",
12    "interests": [
13      "sports",
14      "music"
15    ]
16  }
17 }
```

- 判断资源是否存在 (head请求)

HEAD  http://120.78.138.11:9200/google/employee/5 Params Send  Save 

Authorization Headers (2) Body Pre-request Script Tests Cookies Code

TYPE

Inherit auth from parent 

The authorization header will be automatically generated when you send the request. [Learn more about authorization](#)

This request is not inheriting any authorization helper at the moment. Save it in a collection to use the parent's authorization helper.

Body Cookies Headers (2) Test Results Status: 404 Not Found Time: 51 ms Size: 93 B

HEAD ▾

http://120.78.138.11:9200/google/employee/1

Params

Send ▾

Save ▾

Authorization

Headers (2)

Body

Pre-request Script

Tests

Cookies

Code

TYPE

Inherit auth from parent ▾

The authorization header will be automatically generated when you send the request. [Learn more about authorization](#)

This request is not inheriting any authorization helper at the moment. Save it in a collection to use the parent's authorization helper.

Body

Cookies

Headers (2)

Test Results

Status: 200 OK

Time: 88 ms

Size: 87 B

- 删除数据 (delete请求)

DELETE ▾

http://120.78.138.11:9200/google/employee/3

Params

Send ▾

Authorization

Headers (2)

Body ●

Pre-request Script

Tests

TYPE

Inherit auth from parent ▾

The authorization header will be automatically generated when you send the request. [Learn more about authorization](#)

This request is not inheriting any authorization helper at the moment. Save it in a collection to use the parent's authorization helper.

Body

Cookies

Headers (3)

Test Results

Status: 404 Not Found

Time: 143 ms

Pretty

Raw

Preview

JSON ▾

≡

```
1 {
2   "found": false,
3   "_index": "google",
4   "_type": "employee",
5   "_id": "3",
6   "_version": 1,
7   "result": "not_found",
8   "_shards": {
9     "total": 2,
10    "successful": 1,
11    "failed": 0
12  }
13 }
```

- 更新数据 (put请求)

PUT `http://120.78.138.11:9200/google/employee/2` Params Send

Authorization Headers (2) Body Pre-request Script Tests

form-data x-www-form-urlencoded raw binary JSON (application/json)

```

1 {
2   "first_name": "Jane",
3   "last_name": "Smith",
4   "age": 12,
5   "about": "I like to collect rock albums",
6   "interests": [ "music" ]
7 }

```

Body Cookies Headers (3) Test Results Status: 200 OK Time: 176 ms

Pretty Raw Preview JSON

```

1 {
2   "_index": "google",
3   "_type": "employee",
4   "_id": "2",
5   "_version": 3,
6   "result": "updated",
7   "_shards": {
8     "total": 2,
9     "successful": 1,
10    "failed": 0
11  },
12   "created": false
13 }

```

- `_search` 搜索员工数据；通过json格式的搜索信息进行强大的各种搜索（重点就是查询表达式）

POST `http://120.78.138.11:9200/google/employee/_search` Params Send

Authorization Headers (2) Body Pre-request Script Tests

form-data x-www-form-urlencoded raw binary JSON (application/json)

```

1 {
2   "query": {
3     "match": {
4       "last_name": "Smith"
5     }
6   }
7 }

```

Body Cookies Headers (3) Test Results Status: 200 OK

Pretty Raw Preview JSON

```

1 {
2   "took": 37,
3   "timed_out": false,
4   "_shards": {
5     "total": 5,
6     "successful": 5,
7     "skipped": 0,
8     "failed": 0
9   },
10  "hits": {
11    "total": 2,
12    "max_score": 0.2876821,
13    "hits": [
14      {
15        "_index": "google",
16        "_type": "employee",
17        "_id": "2",
18        "_score": 0.2876821
19      }
20    ]
21  }
22 }

```

3. SpringBoot 整合 ES 并使用jest 操作

① pom.xml

```

1 <dependencies>
2   <dependency>
3     <groupId>org.springframework.boot</groupId>
4     <artifactId>spring-boot-starter-data-elasticsearch</artifactId>
5   </dependency>

```

```

6      <dependency>
7          <groupId>io.searchbox</groupId>
8          <artifactId>jest</artifactId>
9          <version>5.3.3</version>
10     </dependency>
11     <dependency>
12         <groupId>org.springframework.boot</groupId>
13         <artifactId>spring-boot-starter-web</artifactId>
14     </dependency>
15
16     <dependency>
17         <groupId>org.springframework.boot</groupId>
18         <artifactId>spring-boot-starter-test</artifactId>
19         <scope>test</scope>
20     </dependency>
21 </dependencies>

```

② 主程序类

```

1  /*
2  * @Author zc-cris
3  * @Description SpringBoot 默认支持两种方式来操作ElasticSearch
4  * 1. Jest (默认不生效)
5  * 所以需要导入jest的工具包: (io.searchbox.client.JestClient)
6  * 2. SpringData ElasticSearch(有可能版本不对应, 要么升级springboot版本, 要么docker安装对应版本的
   ES)
7  * - Client 节点信息: clusterNodes, clusterName
8  * - ElasticSearchTemplate 操作es
9  * - 还可以编写一个ElasticSearchRepository 的子接口来操作es
10 * 3. 使用spring data 的方式来操作ES
11 * - ES5.6.9版本默认无法从外部访问9300端口, 2.xx 版本就可以, 暂时未解决。。。
12 * - ElasticSearchRepository 接口的子类可以根据方法名来进行数据的检索
13 * - 注入 ElasticSearchTemplate 可以根据不同的条件方法来检索数据
14 * @Param
15 * @return
16 **/
17 @SpringBootApplication
18 public class SpringbootElasticsearchApplication {
19
20     public static void main(String[] args) {
21         SpringApplication.run(SpringbootElasticsearchApplication.class, args);
22     }
23 }

```

③ javaBean

```

1  /**
2  * @ClassName Article
3  * @Description TODO
4  * @Author zc-cris
5  * @Version 1.0

```

```

6  /**/
7  public class Ariticle {
8
9      @JestId      // 指定文档的索引id属性
10     private Integer id;
11     private String author;
12     private String title;
13     private String content;
14     ...
15 }

```

④ application.yml

```

1  spring:
2      elasticsearch:
3          jest:
4              uris: http://120.78.138.11:9200

```

⑤ 测试

```

1  @RunWith(SpringRunner.class)
2  @SpringBootTest
3  public class SpringbootElasticsearchApplicationTests {
4
5      @Autowired
6      JestClient jestClient;
7
8      @Test
9      public void contextLoads() {
10
11          Article ariticle = new Ariticle();
12          ariticle.setId(1);
13          ariticle.setAuthor("张大帅");
14          ariticle.setTitle("屌丝程序员");
15          ariticle.setContent("逆袭白富美，走上人生巅峰~");
16
17          //构建索引
18          Index i = new Index.Builder(ariticle).index("cris").type("articles").build();
19
20          try {
21              // 执行
22              jestClient.execute(i);
23          } catch (IOException e) {
24              e.printStackTrace();
25          }
26
27      }
28
29      // 测试搜索
30      @Test
31      public void test() {

```



```

32      // 查询表达式
33      String string = "{\n" +
34          "      \"query\" : {\n" +
35          "          \"match\" : {\n" +
36          "              \"content\" : \"白富美\"\n" +
37          "          }\n" +
38          "      }\n" +
39          "    }";
40
41      // 构建搜索功能
42      Search build = new
Search.Builder(string).addIndex("cris").addType("articles").build();
43      try {
44          // 执行搜索功能
45          SearchResult execute = jestClient.execute(build);
46          System.out.println(execute.getJsonString());
47      } catch (IOException e) {
48          e.printStackTrace();
49      }
50  }
51 }
52

```

4. SpringBoot 使用ElasticSearchRepository 接口或者ElasticSearchTemplate 类操作ES

① javaBean

```

1  /**
2   * @ClassName Book
3   * @Description 如果使用sprig data 的方式来存储数据到ES，需要在javaBean上面使用@Document注解
4   * @Author zc-cris
5   * @Version 1.0
6   */
7  @Document(indexName = "cris", type = "books")
8  public class Book {
9      private Integer id;
10     private String name;
11     private String author;
12     ...
13 }

```

② application.properties

```

1  spring.data.elasticsearch.cluster-name=elasticsearch
2  spring.data.elasticsearch.cluster-nodes=120.78.138.11:9300

```

③ 自定义Repository

```

1 public interface BookRepository extends ElasticsearchRepository<Book, Integer> {
2 }

```

④ 测试

```

1 @Autowired
2 BookRepository bookRepository;
3
4 @Test
5 public void test03(){
6     Book book = new Book();
7     book.setId(1);
8     book.setName("神魔");
9     book.setAuthor("血红");
10    // System.out.println(bookRepository);
11    bookRepository.index(book);
12 }

```

五. SpringBoot 任务管理

1. pom.xml

```

1 <dependencies>
2     <dependency>
3         <groupId>org.springframework.boot</groupId>
4         <artifactId>spring-boot-starter-web</artifactId>
5     </dependency>
6     <dependency>
7         <groupId>org.springframework.boot</groupId>
8         <artifactId>spring-boot-starter-mail</artifactId>
9     </dependency>
10
11     <dependency>
12         <groupId>org.springframework.boot</groupId>
13         <artifactId>spring-boot-starter-test</artifactId>
14         <scope>test</scope>
15     </dependency>
16 </dependencies>

```

2. 主程序类

```
1 @EnableScheduling // 开启定时任务注解
2 @EnableAsync      // 开启异步注解
3 @SpringBootApplication
4 public class SpringbootTaskApplication {
5
6     public static void main(String[] args) {
7         SpringApplication.run(SpringbootTaskApplication.class, args);
8     }
9 }
```

3. 异步任务处理

```
1 /**
2  * @ClassName AsyncService
3  * @Description TODO
4  * @Author zc-cris
5  * @Version 1.0
6  */
7 @Service
8 public class AsyncService {
9
10     @Async
11     public void hello(){
12         try {
13             Thread.sleep(3000);
14         } catch (InterruptedException e) {
15             e.printStackTrace();
16         }
17         System.out.println("处理数据中...");
18     }
19 }
20
21 /**
22  * @ClassName AsyncController
23  * @Description TODO
24  * @Author zc-cris
25  * @Version 1.0
26  */
27 @RestController
28 public class AsyncController {
29
30     @Autowired
31     AsyncService asyncService;
32
33     @GetMapping("/hello")
34     public String hello(){
35         asyncService.hello();
36         return "success";
37     }
38 }
```

4. 定时任务处理

```

1  /**
2   * @ClassName ScheduleService
3   * @Description TODO
4   * @Author zc-cris
5   * @Version 1.0
6   */
7  @Service
8  public class ScheduleService {
9
10     /**
11      * @Author zc-cris
12      * @Description second,minute,hour,day of month,month,day of week
13      *          * * * * * MON-FRI （表示周一到周五的每一秒都执行这个定时任务）
14      *          【0 0/5 14,18 * * ?】:每天的14点和18点，整点开始每隔5分钟执行一次
15      *          【0 15 10 ? * 1-6】: 每个月的周一到周六的10点15分执行一次
16      *          【0 0 2 ? * 6L】:每个月的最后一个周六的2点执行一次
17      *          【0 0 2 LW * ?】:每个月的最后一个工作日的2点执行一次
18      *          【0 0 2-4 ? * 1#1】: 每个月的第一个周一的2点至4点每个整点执行一次
19      * @Param []
20      * @return void
21      */
22     //@@Scheduled(cron = "0 * * * * MON-FRI") // 周一到周五每分钟执行一次
23     //@@Scheduled(cron = "0,1,2,3,4 * * * * MON-FRI") // 周一到周五每分钟的每0,1,2,3,4 秒执行一
    次
24     //@@Scheduled(cron = "0-4 * * * * MON-FRI") // 同上
25     //@@Scheduled(cron = "0/4 * * * * MON-FRI") // 周一到周五每4秒执行一次
26     public void scheduledTask(){
27         System.out.println("定时任务执行中...");
28     }
29 }

```

4. 邮件任务处理

① application.yml

```

1  spring:
2    mail:
3      username: 990435014@qq.com
4      password: atyhijliedaqbcgg
5      host: smtp.qq.com
6      properties: {mail.smtp.ssl.enable: true}

```

② 邮件发送测试

```

1  @RunWith(SpringRunner.class)
2  @SpringBootTest
3  public class SpringbootTaskApplicationTests {
4

```

```

5      @Autowired
6      JavaMailSenderImpl javaMailSender;
7
8      // 测试发送一封简单的邮件
9      @Test
10     public void contextLoads() {
11         SimpleMailMessage message = new SimpleMailMessage();
12         // 邮件设置
13         message.setSubject("今晚开会");
14         message.setText("晚上8:00 到4楼开会");
15
16         message.setTo("17623887386@163.com");
17         message.setFrom("990435014@qq.com");
18
19         javaMailSender.send(message);
20
21     }
22
23     // 发送一封复杂的邮件
24     @Test
25     public void test() throws Exception{
26         // 创建一封复杂的邮件
27         MimeMessage message = javaMailSender.createMimeMessage();
28         // 是否要上传文件
29         MimeMessageHelper helper = new MimeMessageHelper(message, true);
30
31         // 邮件设置
32         helper.setSubject("通知-今晚开会");
33         helper.setText("<b style='color:red'>今晚9:00到天台好好聊聊, 兄弟</b>", true);
34         helper.setTo("17623887386@163.com");
35         helper.setFrom("990435014@qq.com");
36
37         // 上传附件
38         helper.addAttachment("1.jpg", new File("C:\\Users\\zc-cris\\Pictures\\FLAMING
MOUNTAIN.JPG"));
39         helper.addAttachment("2.jpg", new File("C:\\Users\\zc-cris\\Downloads\\明日花
\\1.jpg"));
40
41         // 发送邮件
42         javaMailSender.send(message);
43     }
44 }

```

六. SpringBoot 和SpringSecurity 整合

1. pom.xml

```

1      <dependencies>
2
3          <dependency>

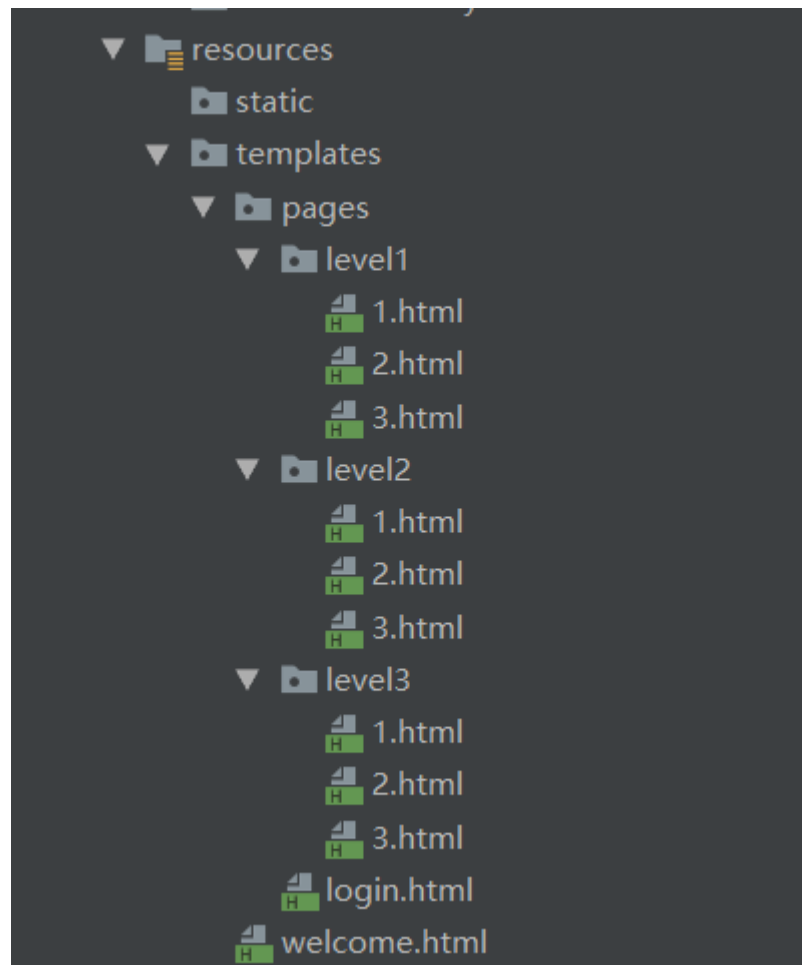
```

```
3         <groupId>org.thymeleaf.extras</groupId>
4         <artifactId>thymeleaf-extras-springsecurity4</artifactId>
5     </dependency>
6
7     <dependency>
8         <groupId>org.springframework.boot</groupId>
9         <artifactId>spring-boot-starter-thymeleaf</artifactId>
10    </dependency>
11    <dependency>
12        <groupId>org.springframework.boot</groupId>
13        <artifactId>spring-boot-starter-security</artifactId>
14    </dependency>
15    <dependency>
16        <groupId>org.springframework.boot</groupId>
17        <artifactId>spring-boot-starter-web</artifactId>
18    </dependency>
19
20    <dependency>
21        <groupId>org.springframework.boot</groupId>
22        <artifactId>spring-boot-starter-test</artifactId>
23        <scope>test</scope>
24    </dependency>
25 </dependencies>
```

2. application.properties

```
1 spring.thymeleaf.cache=false
```

3. 导入静态资源



4. 主程序类

```

1  /*
2  * @Author zc-cris
3  * @Description
4  *   1. 引入spring security 的启动器
5  *   2. 编写spring security 的配置类
6  * @Param
7  * @return
8  **/
9  @SpringBootApplication
10 public class SpringbootSecurityApplication {
11
12     public static void main(String[] args) {
13         SpringApplication.run(SpringbootSecurityApplication.class, args);
14     }
15 }

```

5. 自定义Spring Security 配置类

```

1  /**
2  * @ClassName MySecurityConfig
3  * @Description TODO

```

```

4  * @Author zc-cris
5  * @Version 1.0
6  **/
7  @EnableWebSecurity // 开启security注解
8  public class MySecurityConfig extends WebSecurityConfigurerAdapter {
9
10     // 自定义静态资源的权限规则
11     @Override
12     protected void configure(HttpSecurity http) throws Exception {
13         // super.configure(http);
14         // 定制请求的授权规则
15         http.authorizeRequests().antMatchers("/").permitAll()
16             .antMatchers("/level1/**").hasRole("VIP1")
17             .antMatchers("/level2/**").hasRole("VIP2")
18             .antMatchers("/level3/**").hasRole("VIP3");
19
20         // 开启自动配置的登录功能，即如果没有登录，没有权限就会来到默认的用户登录页面
21
22         http.formLogin().usernameParameter("user").passwordParameter("pwd").loginPage("/userlogin")
23         ; // 发送指定请求到指定的登录页面
24         // 1. 发送 /login请求来到默认的登录页面
25         // 2. 登录失败发送 /login?error 表示登录失败
26         // 3. 更多详情参考文档
27         // 4. 默认发送post形式的/login请求会交给spring security来处理登录验证；但是一旦修改了
28         loginPage方法，
29         // 即定制了loginPage，那么loginPage的post请求就是处理登录验证逻辑的请求（但是可以通过
30         loginProcessingUrl(String url)来修改请求路径
31
32         // 开启自动配置的注销功能
33         http.logout().logoutSuccessUrl("/"); // 修改注销成功后返回首页
34         // 1. 默认访问/logout 表示注销请求，同时清空session
35         // 2. 默认注销成功后返回到 /login?logout 页面（就是默认的登录页面）
36
37         // 开启默认的记住我功能；登录成功后，服务器将名为remember-me的cookie 发送给浏览器保存，以后
38         再访问服务器带上这个cookie即可，只要再
39         // 服务器找到对应的这条cookie记录就可以免登录；如果注销成功也会删除这个cookie
40         http.rememberMe().rememberMeParameter("remember"); // 自定义记住我的参数名字
41     }
42
43     // 用户登录以及授权功能
44     @Override
45     protected void configure(AuthenticationManagerBuilder auth) throws Exception {
46         // super.configure(auth);
47         /*实际开发中用户名和密码，角色信息都应该从内存中读取*/
48         auth.inMemoryAuthentication().withUser("cris").password("123456").roles("VIP1",
49 "VIP2")
50             .and()
51             .withUser("张三").password("123456").roles("VIP2", "VIP3");
52     }
53 }

```

6. controller


```
1 @Controller
2 public class KungfuController {
3     private final String PREFIX = "pages/";
4     /**
5      * 欢迎页
6      * @return
7      */
8     @GetMapping("/")
9     public String index() {
10         return "welcome";
11     }
12
13     /**
14      * 登录页
15      * @return
16      */
17     @GetMapping("/userlogin")
18     public String loginPage() {
19         return PREFIX+"login";
20     }
21
22
23     /**
24      * level1页面映射
25      * @param path
26      * @return
27      */
28     @GetMapping("/level1/{path}")
29     public String level1(@PathVariable("path")String path) {
30         return PREFIX+"level1/"+path;
31     }
32
33     /**
34      * level2页面映射
35      * @param path
36      * @return
37      */
38     @GetMapping("/level2/{path}")
39     public String level2(@PathVariable("path")String path) {
40         return PREFIX+"level2/"+path;
41     }
42
43     /**
44      * level3页面映射
45      * @param path
46      * @return
47      */
48     @GetMapping("/level3/{path}")
49     public String level3(@PathVariable("path")String path) {
50         return PREFIX+"level3/"+path;
51     }
52 }
```

7. 自定义登录页面

```

1 <!DOCTYPE html>
2 <html xmlns:th="http://www.thymeleaf.org">
3 <head>
4 <meta charset="UTF-8">
5 <title>Insert title here</title>
6 </head>
7 <body>
8     <h1 align="center">欢迎登陆武林秘籍管理系统</h1>
9     <hr>
10    <div align="center">
11        <form th:action="@{/userlogin}" method="post">
12            用户名:<input type="text" name="user"/><br>
13            密码:<input type="password" name="pwd"><br>
14            <input type="checkbox" name="remember">记住我<br>
15            <input type="submit" value="登陆">
16        </form>
17    </div>
18 </body>
19 </html>

```

8. 自定义欢迎页面（根据不同权限显示不同内容）

```

1 <!DOCTYPE html>
2 <html xmlns:th="http://www.thymeleaf.org"
3     xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity4">
4 <head>
5 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
6 <title>Insert title here</title>
7 </head>
8 <body>
9 <!--根据当前用户的登录情况，显示不同的内容-->
10 <h1 align="center">欢迎光临武林秘籍管理系统</h1>
11 <div sec:authorize="!isAuthenticated()">
12 <h2 align="center">游客您好，如果想查看武林秘籍 <a th:href="@{/userlogin}">请登录</a></h2>
13 </div>
14 <div sec:authorize="isAuthenticated()">
15     <h2><span sec:authentication="name"></span>,欢迎您! 您的角色有: <span
16 sec:authentication="principal.authorities"></span></h2>
17     <form th:action="@{/logout}" method="post">
18         <input type="submit" th:value="注销">
19     </form>
20 </div>
21 <hr>
22
23 <!--根据不同用户的roles，显示不同的页面数据-->
24 <div sec:authorize="hasRole('VIP1')">
25     <h3>普通武功秘籍</h3>
26     <ul>

```

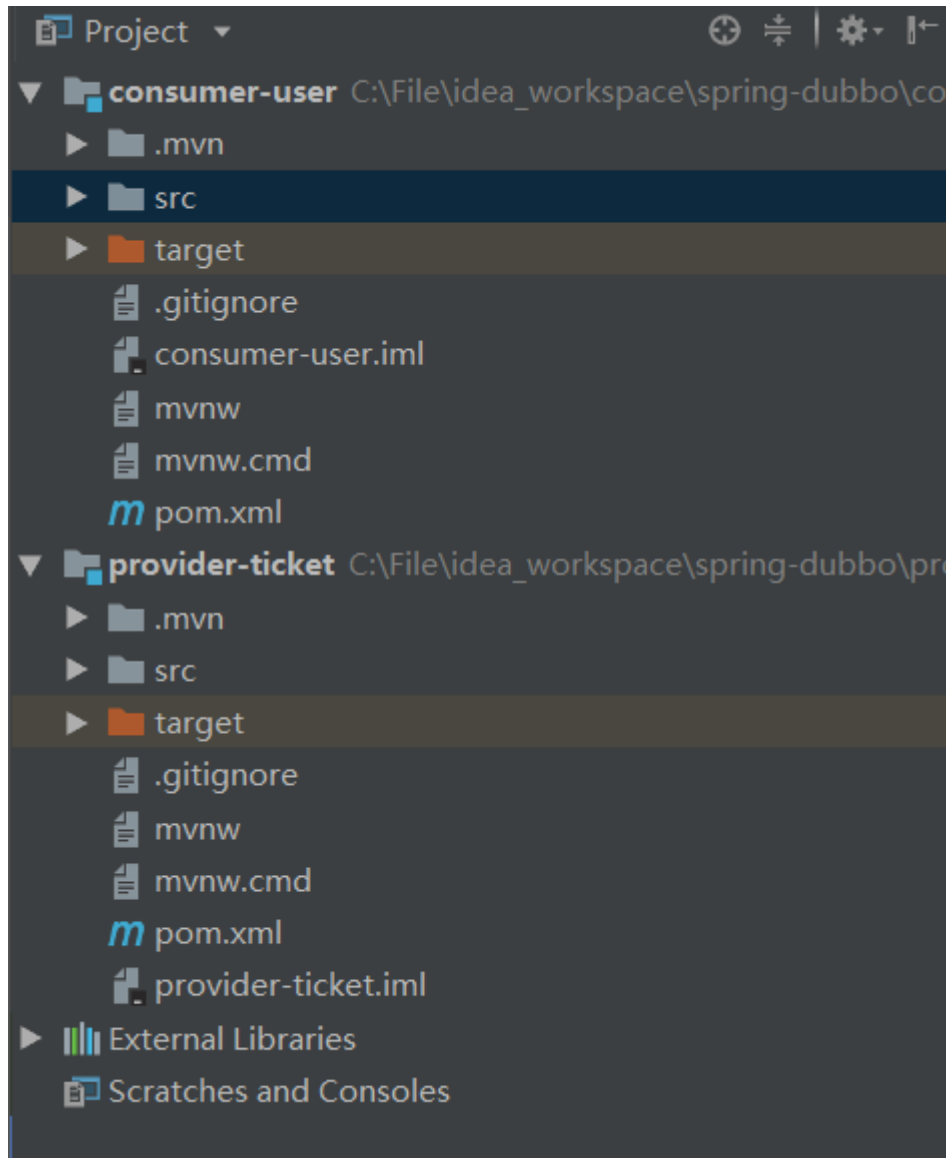
```
27         <li><a th:href="@{/level1/1}">罗汉拳</a></li>
28         <li><a th:href="@{/level1/2}">武当长拳</a></li>
29         <li><a th:href="@{/level1/3}">全真剑法</a></li>
30     </ul>
31 </div>
32
33 <div sec:authorize="hasRole('VIP2')">
34     <h3>高级武功秘籍</h3>
35     <ul>
36         <li><a th:href="@{/level2/1}">太极拳</a></li>
37         <li><a th:href="@{/level2/2}">七伤拳</a></li>
38         <li><a th:href="@{/level2/3}">梯云纵</a></li>
39     </ul>
40 </div>
41
42 <div sec:authorize="hasRole('VIP3')">
43     <h3>绝世武功秘籍</h3>
44     <ul>
45         <li><a th:href="@{/level3/1}">葵花宝典</a></li>
46         <li><a th:href="@{/level3/2}">龟派气功</a></li>
47         <li><a th:href="@{/level3/3}">独孤九剑</a></li>
48     </ul>
49 </div>
50 </body>
51 </html>
```

七. SpringBoot 分布式整合Dubbo, Zookeeper(绝对重点)

1. Docker 下载zookeeper和启动

```
1 docker run --name zk1 -p 2181:2181 --restart always -d 56d414270ae3
```

2. 构建空工程并创建对应的模块（服务提供模块和消费者模块）



3. SpringBoot 引入Dubbo, Zookeeper依赖

```
1  <dependencies>
2    <dependency>
3      <groupId>org.springframework.boot</groupId>
4      <artifactId>spring-boot-starter-web</artifactId>
5    </dependency>
6
7    <!--引入dubbo相关依赖-->
8    <dependency>
9      <groupId>com.alibaba.boot</groupId>
10     <artifactId>dubbo-spring-boot-starter</artifactId>
11     <version>0.1.0</version>
12   </dependency>
13
14   <!--导入zkclient 客户端-->
15   <dependency>
16     <groupId>com.github.sgroschupf</groupId>
17     <artifactId>zkclient</artifactId>
18     <version>0.1</version>
```

```

19         </dependency>
20
21         <dependency>
22             <groupId>org.springframework.boot</groupId>
23             <artifactId>spring-boot-starter-test</artifactId>
24             <scope>test</scope>
25         </dependency>
26     </dependencies>

```

4. 服务提供者模块

① application.properties

```

1 dubbo.application.name=provider-ticket
2 dubbo.registry.address=zookeeper://120.78.138.11:2181
3 dubbo.scan.base-packages=com.cris.ticket.service

```

② 主程序类

```

1  /*
2  * @Author zc-cris
3  * @Description 将服务注册到服务中心
4  *              1. 引入dubbo和zookeeper的相关依赖
5  *              2. 配置dubbo的扫描包和注册中心地址
6  *              3. 使用@Service (dubbo的注解) 发布服务
7  * @Param
8  * @return
9  */
10 @SpringBootApplication
11 public class ProviderTicketApplication {
12
13     public static void main(String[] args) {
14         SpringApplication.run(ProviderTicketApplication.class, args);
15     }
16 }

```

③ 服务组件的service

```

1  /**
2  * @ClassName TicketServiceImpl
3  * @Description TODO
4  * @Author zc-cris
5  * @Version 1.0
6  */
7  @Component
8  @Service // 使用dubbo 的@Service 注解将服务注册到zookeeper
9  public class TicketServiceImpl implements TicketService {
10     @Override
11     public String saleTicket() {
12         return "复仇者联盟三";
13     }
14 }

```

```

13     }
14 }

```

5. 消费者模块

① 引入相同的Zookeeper 和Dubbo 依赖

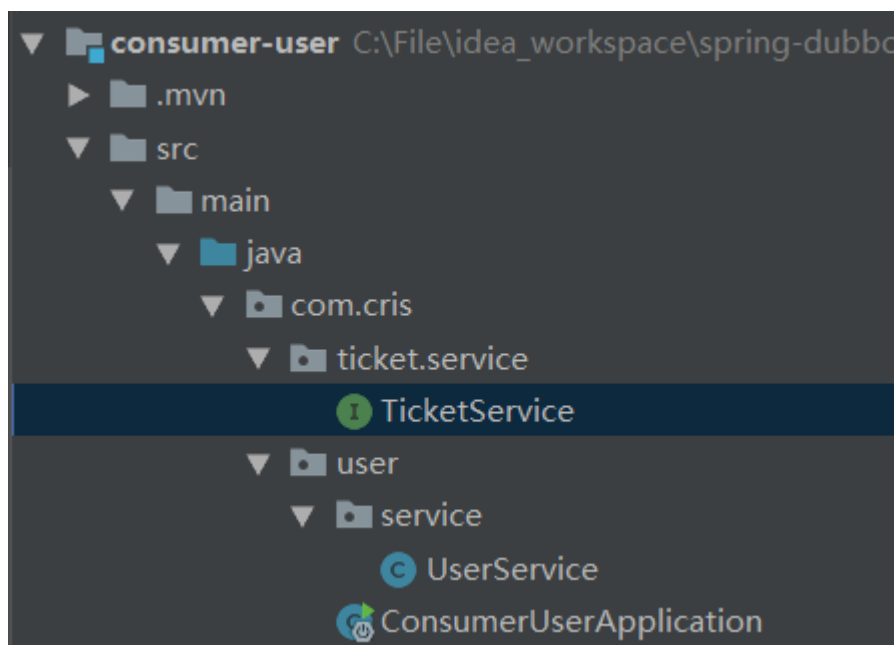
② service

```

1  /**
2   * @ClassName UserService
3   * @Description TODO
4   * @Author zc-cris
5   * @Version 1.0
6   */
7  @Service
8  public class UserService {
9
10     @Reference // 根据全类名来引用注册中心的服务提供者
11     TicketService ticketService;
12
13     public void buy(){
14         System.out.println("买到票了: "+ticketService.saleTicket());
15     }
16 }

```

③ 引入服务提供者的接口



④ application.yml

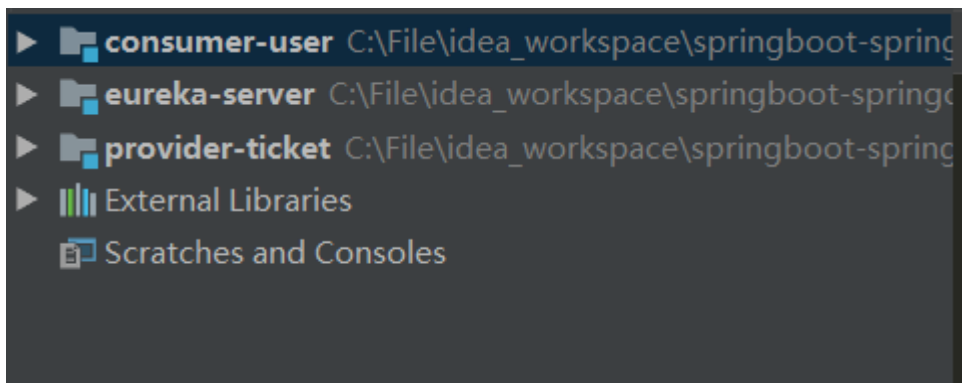
```
1 dubbo:
2   application:
3     name: consumer-user
4   registry:
5     address: zookeeper://120.78.138.11:2181
```

⑤ 测试

```
1 @RunWith(SpringRunner.class)
2 @SpringBootTest
3 public class ConsumerUserApplicationTests {
4
5     @Autowired
6     UserService userService;
7
8     @Test
9     public void contextLoads() {
10         userService.buy();
11     }
12 }
```

八. SpringBoot 整合 SpringCloud（绝对重点）

1. 创建一个空工程，引入eureka 服务中心模块，服务者模块，消费者模块



2. eureka 服务中心模块

① pom.xml（选择Eureka Server 启动器）

```

1 <dependencies>
2     <dependency>
3         <groupId>org.springframework.cloud</groupId>
4         <artifactId>spring-cloud-starter-eureka-server</artifactId>
5     </dependency>
6
7     <dependency>
8         <groupId>org.springframework.boot</groupId>
9         <artifactId>spring-boot-starter-test</artifactId>
10        <scope>test</scope>
11    </dependency>
12 </dependencies>

```

② application.yml

```

1 server:
2     port: 8761
3 eureka:
4     instance:
5         hostname: eureka-server # eureka实例的主机名
6     client:
7         register-with-eureka: false #不把自己注册到eureka上
8         fetch-registry: false # 不从eureka上获取服务的注册信息
9         service.url:
10            defaultZone: http://localhost:8761/eureka/ #我们自己的服务中心地址

```

③ 主程序类

```

1 /**
2  * @Author zc-cris
3  * @Description 注册中心
4  * 1. 配置Eureka注册中心的信息
5  * 2. @EnableEurekaServer
6  * @Param
7  * @return
8  */
9 @EnableEurekaServer
10 @SpringBootApplication
11 public class EurekaServerApplication {
12
13     public static void main(String[] args) {
14         SpringApplication.run(EurekaServerApplication.class, args);
15     }
16 }

```

3. 服务者模块

① pom.xml


```

1      <dependencies>
2          <dependency>
3              <groupId>org.springframework.cloud</groupId>
4              <artifactId>spring-cloud-starter-eureka</artifactId>
5          </dependency>
6
7          <dependency>
8              <groupId>org.springframework.boot</groupId>
9              <artifactId>spring-boot-starter-test</artifactId>
10             <scope>test</scope>
11          </dependency>
12      </dependencies>

```

② application.yml

```

1  server:
2      port: 8002
3  spring:
4      application:
5          name: provider-ticket
6  eureka:
7      instance:
8          prefer-ip-address: true #注册到服务中心使用ip进行注册
9      client:
10         service.url:
11             defaultZone: http://localhost:8761/eureka/ #我们自己的服务中心地址

```

③ service

```

1  /**
2   * @ClassName TicketService
3   * @Description TODO
4   * @Author zc-cris
5   * @Version 1.0
6   **/
7  @Service
8  public class TicketService {
9
10     public String getTicket(){
11         System.out.println("这是8002端口提供的服务");
12         return "《复仇者联盟3》";
13     }
14 }

```

④ controller

```

1  /**
2   * @ClassName TicketController
3   * @Description TODO

```

```

4  * @Author zc-cris
5  * @Version 1.0
6  **/
7  @RestController
8  public class TicketController {
9
10     @Autowired
11     TicketService ticketService;
12
13     @GetMapping("/ticket")
14     public String getTicket(){
15         return ticketService.getTicket();
16     }
17 }
18 }

```

4. 消费者模块

① pom.xml 同服务者模块（都是选择Eureka Discovery 启动器）

② application.yml

```

1  spring:
2      application:
3          name: consumer-user
4  server:
5      port: 8200
6  eureka:
7      instance:
8          prefer-ip-address: true #注册到服务中心使用ip进行注册
9      client:
10         service.url:
11             defaultZone: http://localhost:8761/eureka/ #我们自己的服务中心地址

```

③ 主程序类

```

1  @EnableDiscoveryClient //开启发现eureka注册中心的服务组件的功能
2  @SpringBootApplication
3  public class ConsumerUserApplication {
4
5      public static void main(String[] args) {
6          SpringApplication.run(ConsumerUserApplication.class, args);
7      }
8
9      @LoadBalanced //启动负载均衡机制
10     @Bean // 这个类帮助我们的消费者向服务中心发送http请求
11     public RestTemplate restTemplate(){
12         return new RestTemplate();
13     }

```

14 }

④ controller

```

1  /**
2   * @ClassName UserController
3   * @Description TODO
4   * @Author zc-cris
5   * @Version 1.0
6   */
7  @RestController
8  public class UserController {
9
10     @Autowired
11     RestTemplate restTemplate;
12
13     @GetMapping("/buy")
14     public String buyTicket(String name) {
15         String ticket = restTemplate.getForObject("http://PROVIDER-TICKET/ticket",
String.class);
16         return name + "买了票" + ticket;
17     }
18 }

```

5. 测试总结：

和 Dubbo 以及 Zookeeper 的分布式应用相比，SpringCloud 为我们提供了更加一致性的整体解决方案，通过 SpringCloud 为我们提供的五大神兽，我们可以以一种更加从容和优雅的方式搭建我们的分布式应用

SpringCloud 分布式开发五大常用组件

1. 服务发现——Netflix Eureka
2. 客户端负载均衡——Netflix Ribbon
3. 断路器——Netflix Hystrix
4. 服务网关——Netflix Zuul
5. 分布式配置——Spring Cloud Config

我们不再需要将服务者模块发布到远程的Zookeeper 注册中心服务器，不需要通过Dubbo 完成消费者模块和服务者模块之间的信息交互（导入服务者模块的接口到消费者模块）；SpringCloud 借助于强大的SpringBoot，将我们的应用完美的以微服务的架构呈现，并且提供一站式解决方案；每个模块之间的交互通过rest 请求进行交互；在一个庞大的整体模块中，每一个模块又分割的仅仅有条，这就是SpringCloud 给出的微服务架构最好解决方案。

九. SpringBoot 的热部署（Devtools）

1. 导入对应的启动器即可

```
1 <dependency>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-devtools</artifactId>
4     <scope>runtime</scope>
5 </dependency>
```

十. SpringBoot 监控管理

1. pom.xml

```
1 <dependencies>
2     <dependency>
3         <groupId>org.springframework.boot</groupId>
4         <artifactId>spring-boot-starter-actuator</artifactId>
5     </dependency>
6     <dependency>
7         <groupId>org.springframework.boot</groupId>
8         <artifactId>spring-boot-starter-data-redis</artifactId>
9     </dependency>
10    <dependency>
11        <groupId>org.springframework.boot</groupId>
12        <artifactId>spring-boot-starter-web</artifactId>
13    </dependency>
14
15    <dependency>
16        <groupId>org.springframework.boot</groupId>
17        <artifactId>spring-boot-devtools</artifactId>
18        <scope>runtime</scope>
19    </dependency>
20    <dependency>
21        <groupId>org.springframework.boot</groupId>
22        <artifactId>spring-boot-starter-test</artifactId>
23        <scope>test</scope>
24    </dependency>
25 </dependencies>
```

2. 配置文件(application.properties)定制端点信息

```
1 # 允许访问端点
2 management.security.enabled=false
3
4 info.app.name=hello
5 info.app.version=1.0.0
6
7 # 允许远程通过访问/shutdown 路径关闭应用, 仅开发使用
8 endpoints.shutdown.enabled=true
9
10 # 开启或关闭某个端点访问
```

```

10 #endpoints.beans.enabled=false
11 # 定制端点访问路径
12 #endpoints.dump.path=/du
13
14 # 关闭所有端点访问
15 #endpoints.enabled=false
16 #endpoints.beans.enabled=true
17
18 # 定制访问端点的根路径 (结合spring security 完成权限控制)
19 management.context-path=/manage
20
21 # (结合spring security 完成权限控制)
22 # 定制端点访问的端口号, -1表示关闭所有端点访问
23 management.port=8181
24
25 spring.redis.host=120.78.138.11

```

3. 自定义组件健康状态监控器类

```

1  /**
2   * @ClassName MyHealthIndicator
3   * @Description 自定义组件健康监控器必须实现HealthIndicator 接口, 并且命名必须是
   xxxHealthIndicator
4   * @Author zc-cris
5   * @Version 1.0
6   */
7  @Component
8  public class MyHealthIndicator implements HealthIndicator {
9
10     @Override
11     public Health health() {
12
13         // 自定义的检查方法
14         // 代表健康
15         // return Health.up().build();
16         return Health.down().withDetail("msg", "服务挂掉了! ").build();
17     }
18 }

```