

# 1. 学习SpringBoot整合常用的框架

我们在学习项目一时用到了很多框架比如shiro、Dubbo、Quartz、rabbitMQ，每使用一个框架时都需要一些xml配置文件，比较麻烦，而SpringBoot在整合这些框架时就非常方便而且不用出现任何xml文件，接下来见证一下吧！

## 1.1 整合Thymeleaf

### 1.1.1 需求分析

现在需要把用户信息在list.html页面上展示，目前有两种思路

- 思路1：直接在浏览器上访问 list.html页面，进入页面后异步加载商品数据，渲染页面
- 思路2：将请求交给tomcat处理，在服务端完成查询数据，跳转到list.html页面

两种方式都行，现在我们选择思路2，但是现在的页面是静态的HTML页面，静态页面中没有request域、不能使用>标签，那怎么把数据显示到静态页面上呢，答案是：可以使用Thymeleaf。Thymeleaf是一种模板引擎，今天的资料中有提供Thymeleaf的学习。

下面我们就用Thymeleaf把数据显示到静态页面上

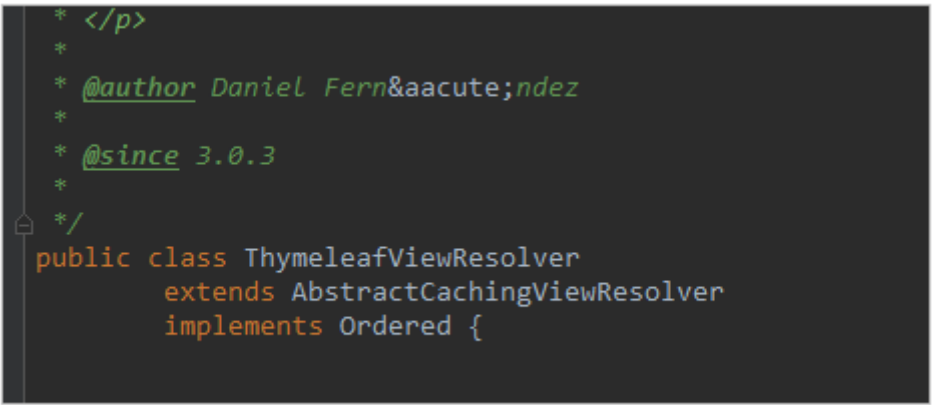
### 1.1.2 环境准备

在昨天的springboot-demo项目中修改：

第一步：添加Thymeleaf的启动器

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

不需要做任何配置，启动器已经帮我们把Thymeleaf的视图器配置完成：



```
* </p>
*
* @author Daniel Fernandez
*
* @since 3.0.3
*
*/
public class ThymeleafViewResolver
    extends AbstractCachingViewResolver
    implements Ordered {
```

而且，还配置了模板文件（html）的位置，与jsp类似的前缀+ 视图名 + 后缀风格：

```
* @author Kozuki Shimizu
* @since 1.2.0
*/
@ConfigurationProperties(prefix = "spring.thymeleaf")
public class ThymeleafProperties {

    private static final Charset DEFAULT_ENCODING = StandardCharsets.UTF_8;

    public static final String DEFAULT_PREFIX = "classpath:/templates/";

    public static final String DEFAULT_SUFFIX = ".html";
}
```

- 默认前缀: `classpath:/templates/`
- 默认后缀: `.html`

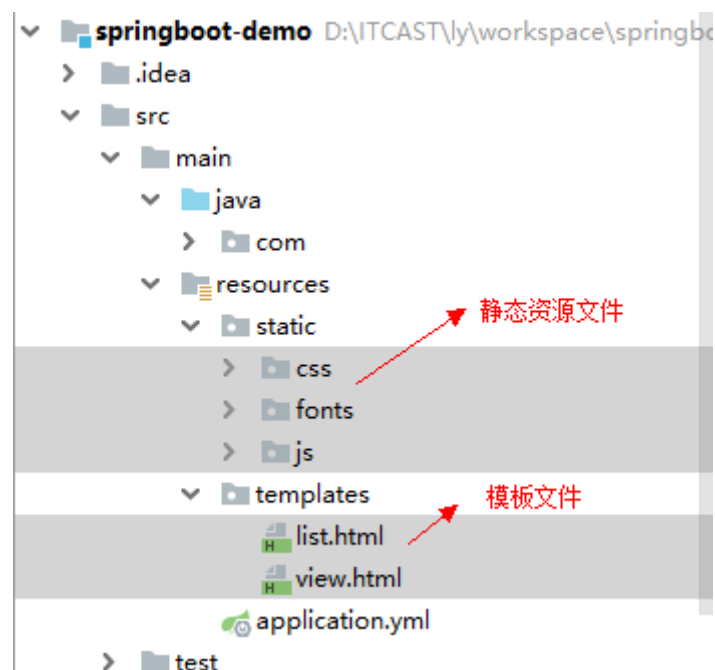
所以如果我们返回视图: `list`, 会指向到 `classpath:/templates/list.html`

第二步: 把今天资料中准备好的静态资源, 直接放入到项目的resources下的templates和static文件夹下

资源文件 -> 静态资源文件 -> day02\_SpringBoot-高级篇 -> 资料 -> static

名称	修改日期	类型	大小
css	2019/12/6 11:33	文件夹	
fonts	2019/12/6 11:33	文件夹	
js	2019/12/6 11:33	文件夹	
list.html	2019/12/6 11:31	Chrome HTML D...	2 KB
view.html	2019/12/6 11:32	Chrome HTML D...	2 KB

如下结构:



### 1.1.3 实现功能

第一步：在Controller、service中添加查询所有用户的方法

我们重新创建一个正式的用户Controller代码，因为要跳转到页面，所以就不用RestController注解了

UserController代码如下：

```
package com.leyou.controller;

import com.leyou.pojo.User;
import com.leyou.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;

import java.util.List;

@Controller
@RequestMapping("/user")
public class UserController {

    @Autowired
    private UserService userService;

    @GetMapping("/{id}")
    public String user(@PathVariable("id") Long id, Model model){
        User user = userService.findById(id);
        model.addAttribute("user",user);
        return "view";
    }

    @GetMapping("/findAll")
    public String findAll( Model model){
        List<User> userList = userService.findAll();
        model.addAttribute("userList",userList);
        return "list";
    }
}
```

service实现类添加的代码：

```
@Override
public List<User> findAll() {
    return userMapper.selectAll(); //直接使用通用mapper提供的方法
}
```

第二步：查看静态页面

**注意**，把html的名称空间，改成：

```
xmlns:th="http://www.thymeleaf.org"
```

```
view.html x list.html x
1 <!DOCTYPE html>
2 <!-- 网页使用的语言 -->
3 <html lang="zh-CN" xmlns:th="http://www.thymeleaf.org">
4 <head>
5 <!-- 指定字符集 -->
```

第三步：测试

<http://localhost/user/findAll>

员工信息列表					
编号	姓名	用户名	年龄	备注信息	操作
1	张三	zhangsan1	25	张三同学在学习Java	<a href="#">查看</a>
2	李四	lisi11	21	李四同学在学习Java	<a href="#">查看</a>
3	王五	wangwu	22	王五同学在学习php	<a href="#">查看</a>
4	张伟	zhangwei	20	张伟同学在学习播音学Java	<a href="#">查看</a>
5	李娜	lina	28	李娜同学在学习播音学Java	<a href="#">查看</a>

点击查看按钮

员工查看页面	
姓名：张三 用户名：zhangsan1 年龄：25 性别：男 备注：张三同学在学习Java	<a href="#">返回列表</a>

搞定！

## 1.2 SpringBoot整合rabbitMQ

### 1.2.1 需求分析

创建两个springboot项目。一个用来向RabbitMQ中发送消息，一个从RabbitMQ中获取消息

### 1.2.2 环境准备

第一步：准备RabbitMQ

浏览器访问：<http://127.0.0.1:15672> 使用guest/guest登录，在控制创建

虚拟主机: /leyou

添加用户并设置密码: leyou/leyou

给用户赋虚拟主机的权限:使用"leyou"账户操作"/leyou"虚拟主机、

Overview Connections Channels Exchanges Queues **Admin**

### Users

▼ All users

Filter:  ☐ Regex ?

Name	Tags	Can access virtual hosts	Has password
guest	administrator	/, /saas	*
saas		/saas	*

1、添加用户

▼ Add a user

Username:

Password:   (confirm)

Tags:

Set **Admin** | Monitoring | Polycymaker  
Management | Impersonator | None

**Add user**



Overview Connections Channels Exchanges Queues **Admin**

User **guest** Log out

### Virtual Hosts

▼ All virtual hosts

Filter:  ☐ Regex ?

2 items, page size up to 100

Overview		Messages			Network		Message rates		
Name	Users ?	State	Ready	Unacked	Total	From client	To client	publish	deliver / get
/	guest	running	NaN	NaN	NaN				
/saas	guest, saas	running	1	0	1				

2、添加虚拟主机

▼ Add a new virtual host

Name:

**Add virtual host**

**Users**  
**Virtual Hosts**  
Policies  
Limits  
Cluster



Overview Connections Channels Exchanges Queues **Admin**

### Users

▼ All users

Filter:  ☐ Regex ?

Name	Tags	Can access virtual hosts	Has password
guest	administrator	/, /leyou, /saas	*
leyou		No access	*
saas		/saas	*

3、点击leyou用户



Current permissions

... no permissions ...

Set permission

Virtual Host:

Configure regexp:

Write regexp:

Read regexp:

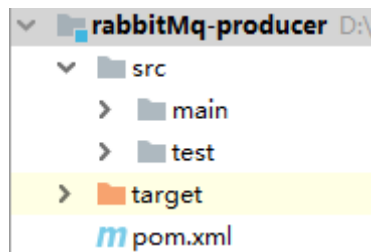
**Set permission**

▼ Topic permissions

4、给leyou用户分配虚拟主机/leyou

第二步:

1、创建消费提供者项目 rabbitMq-producer



## 2、项目中添加依赖

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.3.RELEASE</version>
</parent>

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

## 3、在resources目录下添加application.yml文件，里面内容如下

```
spring:
  rabbitmq:
    host: 127.0.0.1
    username: leyou
    password: leyou
    virtual-host: /leyou
```

## 4、在com.itcast下创建引导类

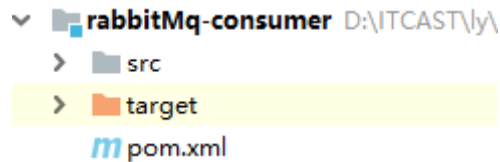
```
package com.leyou;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MqApplication {
    public static void main(String[] args) {
        SpringApplication.run(MqApplication.class, args);
    }
}
```

第三步：

## 1、创建消费提供者项目 rabbitMq-consumer



2、3、4和rabbitMq-producer的操作一模一样！

### 1.2.3 实现功能

springboot和junit整合发送消息

第一步：在**rabbitMq-producer**中的com.leyou.rabbitmq.producer包下添加RabbitMqProducer类，代码如下

```
package com.leyou.rabbitmq.producer;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest
//以上两个注解是在SpringBoot中加载Spring的IOC，使junit测试固定的两个注解
public class RabbitMqProducer {

    @Autowired
    private RabbitTemplate rabbitTemplate; //注入模板

    @Test
    public void testSend(){
        rabbitTemplate.convertAndSend("spring.test.exchange", "a.b",
            "SpringBoot整合rabbitMQ的一条消息");
    }

}
```

第二步：接收消息

在**rabbitMq-consumer**项目中的com.leyou.rabbitmq.consumer包下添加以下代码

```
package com.leyou.rabbitmq.consumer;

import org.springframework.amqp.core.ExchangeTypes;
import org.springframework.amqp.rabbit.annotation.Exchange;
import org.springframework.amqp.rabbit.annotation.Queue;
import org.springframework.amqp.rabbit.annotation.QueueBinding;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

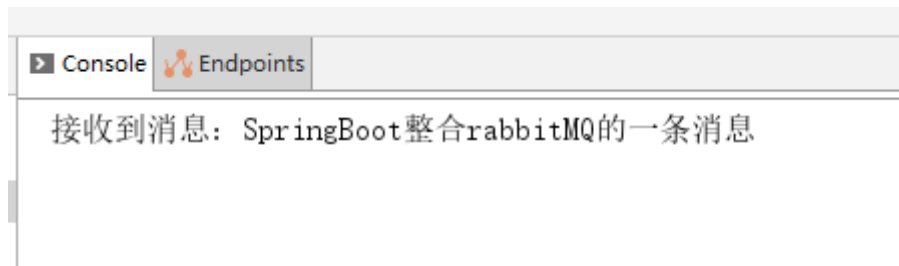
/**
 * 在方法上的注解上声明队列、交换机和队列交换机的绑定关系
 */
@Component
public class ConsumerListener {

    @RabbitListener(bindings = @QueueBinding(
```

```
        value = @Queue(value = "spring.test.queue", durable = "true"),
        exchange = @Exchange(value = "spring.test.exchange", type =
ExchangeTypes.TOPIC),
        key = "#.#"))
    public void myListen(String msg){
        System.out.println("接收到消息: " + msg);
    }
}
```

第三步：测试

- 1、先启动rabbitMq-consumer项目的引导类
- 2、在执行rabbitMq-producer项目中的junit方法



消费者端收到消息，搞定！

## 1.3 SpringBoot整合Redis

在互联网项目中经常把高频使用的数据会缓存起来，而redis是大多数项目优先考虑的一款非关系型数据库，

我们使用Redis都是采用的Jedis客户端，不过既然我们使用了SpringBoot，我就就使用Spring对Redis封装的套件，谁呢？是Spring Data Redis

Spring Data Redis，是Spring Data 家族的一部分。对Jedis客户端进行了封装，与spring进行了整合。可以非常方便的来实现redis的配置和操作。

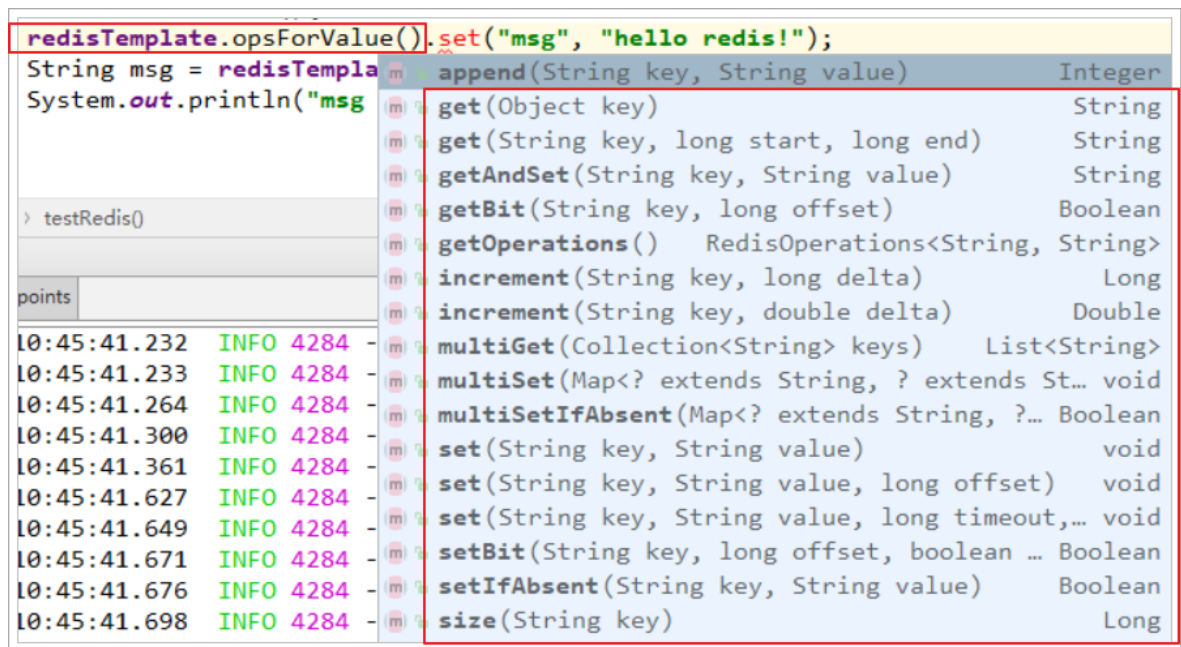
### 1.3.1.RedisTemplate基本操作

与以往学习的套件类似，Spring Data 为 Redis 提供了一个工具类：RedisTemplate。里面封装了对于Redis的五种数据结构的各种操作，包括：

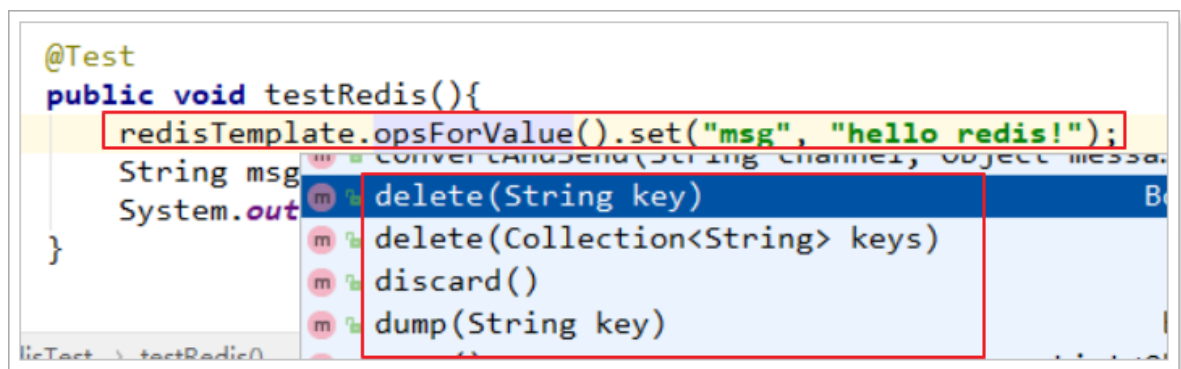
- redisTemplate.opsForValue()：操作字符串
- redisTemplate.opsForHash()：操作hash
- redisTemplate.opsForList()：操作list
- redisTemplate.opsForSet()：操作set
- redisTemplate.opsForZSet()：操作zset

例如我们对字符串操作比较熟悉的有：get、set等命令，这些方法都在 opsForValue()返回的对象中有：





其它一些通用命令，如del，可以通过redisTemplate.xx()来直接调用。

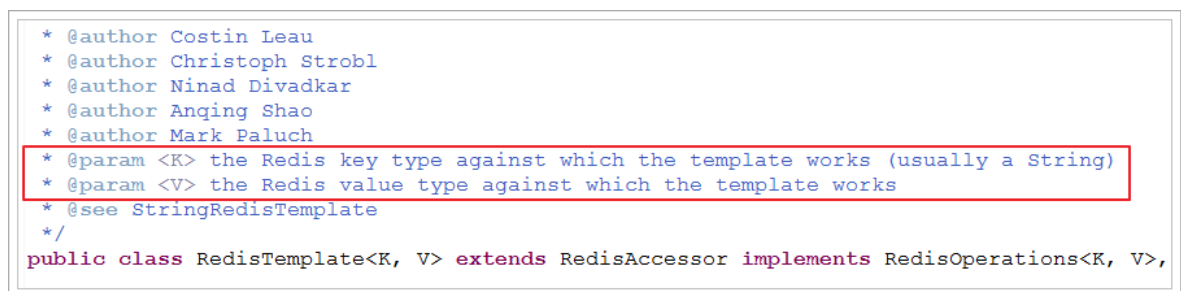


### 1.3.2.StringRedisTemplate

RedisTemplate在创建时，可以指定其泛型类型：

- K：代表key的数据类型
- V：代表value的数据类型

注意：这里的类型不是Redis中存储的数据类型，而是Java中的数据类型，RedisTemplate会自动将Java类型转为Redis支持的数据类型：字符串、字节、二二进制等等。



不过RedisTemplate默认会采用JDK自带的序列化（Serialize）来对对象进行转换。生成的数据十分庞大，因此一般我们都会指定key和value为String类型，这样就由我们自己把对象序列化为json字符串来存储即可。

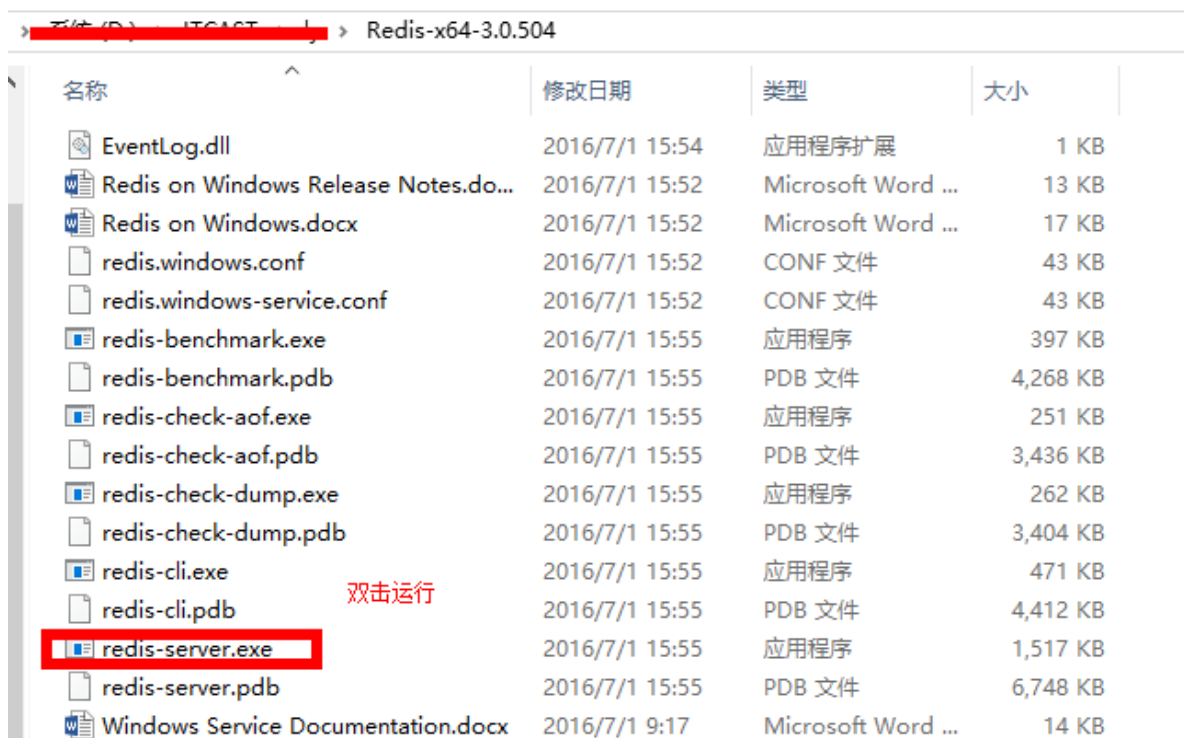
因为大部分情况下，我们都会使用key和value都为String的RedisTemplate，因此Spring就默认提供了这样一个实现：

```
* @author Costin Leau
public class StringRedisTemplate extends RedisTemplate<String, String> {

    /**
     * Constructs a new <code>StringRedisTemplate</code> instance. {@link #setCo
     * and {@link #afterPropertiesSet()} still need to be called.
     */
    public StringRedisTemplate() {
        RedisSerializer<String> stringSerializer = new StringRedisSerializer();
        setKeySerializer(stringSerializer);
        setValueSerializer(stringSerializer);
        setHashKeySerializer(stringSerializer);
        setHashValueSerializer(stringSerializer);
    }
}
```

### 1.3.3.环境准备

第一步：把今天资料中提供的windows版本的redis软件，解压到一个没有中文没有空格的文件夹中运行



第二步：我们新建一个测试项目，添加Redis启动器的依赖、然后在项目中创建引导类、还有配置文件

#### 1、创建项目



#### 2、添加Redis启动器的依赖

```

<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.3.RELEASE</version>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-redis</artifactId>
    </dependency>
</dependencies>

```

3、在com.leyou包下添加引导类

```

package com.leyou;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class RedisApplication {
    public static void main(String[] args) {
        SpringApplication.run(RedisApplication.class, args);
    }
}

```

4、然后在配置文件中指定Redis地址：

在resources下添加application.yml文件，内容如下

```

spring:
  redis:
    host: 127.0.0.1
    port: 6379

```

## 1.3.4 测试RedisTemplate的使用

在com.leyou.redis下创建一个测试类RedisTest

然后就可以直接注入StringRedisTemplate对象了：

```

package com.leyou.test;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.data.redis.core.BoundHashOperations;
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.test.context.junit4.SpringRunner;

import java.util.Map;
import java.util.concurrent.TimeUnit;

```

```

@RunWith(SpringRunner.class)
@SpringBootTest
public class RedisTest {

    @Autowired
    private StringRedisTemplate redisTemplate;

    @Test
    public void testRedis() {
        // 存储数据
        this.redisTemplate.opsForValue().set("key1", "value1");
        // 获取数据
        String val = this.redisTemplate.opsForValue().get("key1");
        System.out.println("val = " + val);
    }

    @Test
    public void testRedis2() {
        // 存储数据，并指定剩余生命周期,5小时
        this.redisTemplate.opsForValue().set("key2", "value2",
            5, TimeUnit.HOURS);
    }

    @Test
    public void testHash(){
        BoundHashOperations<String, Object, Object> hashOps =
            this.redisTemplate.boundHashOps("user");
        // 操作hash数据
        hashOps.put("name", "jack");
        hashOps.put("age", "21");

        // 获取单个数据
        Object name = hashOps.get("name");
        System.out.println("name = " + name);

        // 获取所有数据
        Map<Object, Object> map = hashOps.entries();
        for (Map.Entry<Object, Object> me : map.entrySet()) {
            System.out.println(me.getKey() + " : " + me.getValue());
        }
    }
}

```

## 2. SpringBoot原理详细分析

以上的整合中大都是在项目中导入启动器就可以直接使用了，非常方便，这是什么原理呢，接下来我们从几个注解来剖析一下SpringBoot的原理

### 2.1 @Conditional说明

看他的中文意思：有条件的

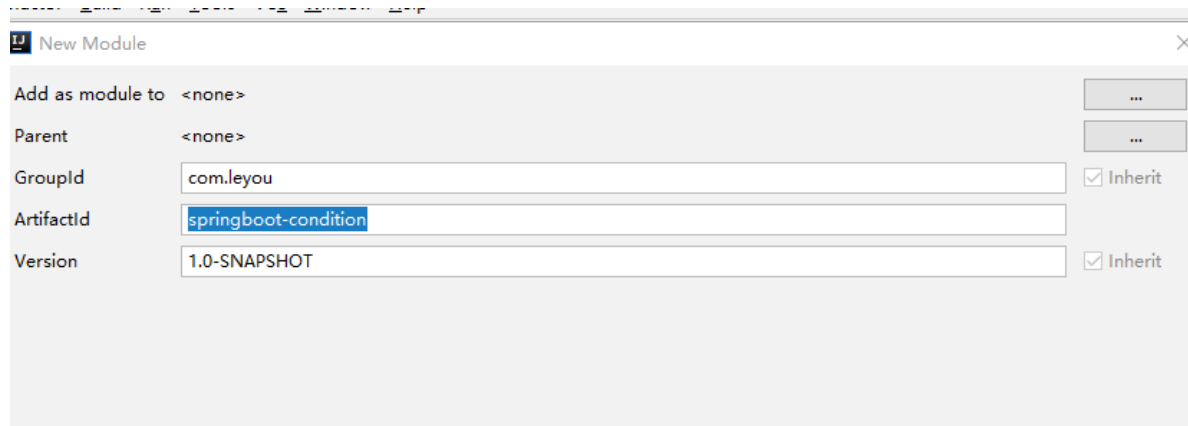
Condition 是在Spring 4.0 增加的条件判断功能，通过这个可以功能可以实现选择性的创建 Bean 操作。

SpringBoot是如何知道要创建哪个Bean的？比如SpringBoot是如何知道要创建RedisTemplate的？

接下来，创建一个项目具体演示@Conditional注解的用法：

## 2.1.1 环境准备

### 1、创建项目



### 2、导入依赖

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.3.RELEASE</version>
</parent>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <!--springboot项目基本的启动器-->
    <artifactId>spring-boot-starter</artifactId>
  </dependency>
</dependencies>
```

### 3、在com.leyou包下创建引导类

```
package com.leyou;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;

@SpringBootApplication
public class ConditionApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConditionApplication.class, args);
    }
}
```

### 4、在com.leyou.pojo中添加一个User类,空的就行

```
package com.leyou.pojo;

public class User {
}
```

5、在com.leyou.config下创建一个配置类，配置类中使用@Bean创建Bean对象

```
package com.leyou.config;

import com.leyou.pojo.User;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration //声明这是个配置类
public class UserConfig {
    @Bean //配置类中使用@Bean创建Bean对象
    public User user(){
        return new User();
    }
}
```

6、在引导类中获取User对象

```
@SpringBootApplication
public class ConditionApplication {
    public static void main(String[] args) {
        ConfigurableApplicationContext app =
        SpringApplication.run(ConditionApplication.class, args);
        User user = app.getBean(User.class);
        System.out.println(user);
    }
}
```

看控制台日志的输出，证明User对象已经获取了

The screenshot shows the Spring Boot console output. At the top, there's a ASCII art logo for Spring Boot. Below it, the text "Spring Boot :: (v2.1.3.RELEASE)" is displayed. Then, there are three log lines: "INFO 6048 --- [main] com.leyou.ConditionApplication". The final line, which is highlighted with a red box, is "com.leyou.pojo.User@928fbf", indicating that the User object has been successfully created and its memory address is printed.

现在要做一些限制

## 2.1.2 有选择性的创建对象

需求1：导入Jedis坐标后，创建User这个Bean，没导入，则不创建。

### 2.1.2.1 @Conditional使用说明



参考上图理解下面的步骤

① 在创建bean的位置添加@Conditional注解

```
@Configuration
public class UserConfig {
    @Bean
    @Conditional() //此时这里会报错的
    public User user(){
        return new User();
    }
}
```

② @Conditional注解的源码

```
package org.springframework.context.annotation;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Conditional {
    Class<? extends Condition>[] value();
}
```

③ 自定义一个实现了Condition接口的子类,需要实现一个matches方法，里面默认是直接返回false

```

package com.leyou.config;

import org.springframework.context.annotation.Condition;
import org.springframework.context.annotation.ConditionContext;
import org.springframework.core.type.AnnotatedTypeMetadata;

public class MyCondition implements Condition {
    @Override
    public boolean matches(ConditionContext conditionContext,
        AnnotatedTypeMetadata annotatedTypeMetadata) {
        return false;
    }
}

```

④ 在创建User的bean时指定MyCondition的字节码

```

@Configuration
public class UserConfig {
    @Bean
    @Conditional(MyCondition.class)
    public User user() {
        return new User();
    }
}

```

再运行引导类时发现User对象获取不到了

```

Exception in thread "main" org.springframework.beans.factory.NoSuchBeanDefinitionException: No qualifying bean of type 'com.leyou.pojo.User' available
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBean(DefaultListableBeanFactory.java:343)
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBean(DefaultListableBeanFactory.java:335)
    at org.springframework.context.support.AbstractApplicationContext.getBean(AbstractApplicationContext.java:1123)
    at com.leyou.ConditionApplication.main(ConditionApplication.java:12)

```

原因是因为自定义的MyCondition中的matches方法返回false导致的，那么如果返回true就能获取到了，那在回到我们的需求：导入Jedis坐标后，创建User这个Bean，没导入，则不创建，那么我们怎么知道Jedis的坐标有没有导入呢？其实很简单，看能不能加载到Jedis这个类就可判断了

### 2.1.2.2 修改Condition的条件

所以再修改MyCondition中的matches方法如下：



```

public class MyCondition implements Condition {
    @Override
    public boolean matches(ConditionContext conditionContext,
        AnnotatedTypeMetadata annotatedTypeMetadata) {
        try {
            Class.forName("redis.clients.jedis.Jedis");
            return true;
        } catch (ClassNotFoundException e) {
            // 抛异常说明没有加载到Jedis这个类，就return false
            return false;
        }
    }
}

```

测试发现，导入Jedis的坐标就能获取User对象，不导入Jedis的坐标就不能获取User对象了

Jedis坐标如下，自己测试一下吧~

```

<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
</dependency>

```

### 2.1.3 @Conditional的高级玩法

需求：将类的判断定义为动态的。判断哪个字节码文件存在可以动态指定。

自定义条件的步骤

第一步：自定义一个注解@MyConditionalOnClass,

需要继承@Conditional注解，再自定义的注解中添加一个name属性

```

package com.leyou.config;

import org.springframework.context.annotation.Conditional;

import java.lang.annotation.*;

@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Conditional(MyCondition.class)
public @interface MyConditionalOnClass {
    String name();
}

```

第二步：修改刚才的MyCondition类：重写 matches 方法，在 matches 方法中进行逻辑判断，返回 boolean值。matches 方法两个参数：

context：上下文对象，可以获取属性值，获取类加载器，获取BeanFactory等。

metadata：元数据对象，用于获取注解属性。

```

public class MyCondition implements Condition {
    @Override

```

```

    public boolean matches(ConditionContext conditionContext,
        AnnotatedTypeMetadata annotatedTypeMetadata) {
        try {
            // 获取指定注解上的所有属性
            Map<String, Object> attributes =
                annotatedTypeMetadata.getAnnotationAttributes(MyConditionalOnClass.class.getName());
            // MyConditionalOnClass有一个name属性
            String className = (String)attributes.get("name");
            // className就是写在MyConditionalOnClass注解里的name值
            Class.forName(className);
            return true;
        } catch (ClassNotFoundException e) {
            // 抛异常说明没有加载到Jedis这个类，就return false
            return false;
        }
    }
}

```

第三步：在初始化Bean时，使用 @MyClassConditional(name="类的全限定名")注解  
在UserConfig中修改user()方法

```

@Bean
@MyConditionalOnClass(name="redis.clients.jedis.Jedis")
public User user(){
    return new User();
}

```

## 2.1.4 SpringBoot常用条件注解

ConditionalOnClass：判断环境中是否有对应字节码文件才初始化Bean，

和我们自定义的MyConditionalOnClass原理一模一样

ConditionalOnProperty：判断配置文件中是否有对应属性和值才初始化Bean

ConditionalOnMissingBean：判断环境中没有对应Bean才初始化Bean，和ConditionalOnClass相反

## 2.2 @Import说明

SpringBoot 工程是否可以直接获取第三方（非spring的）jar包中定义的Bean？

SpringBoot中提供了很多Enable开头的注解，这些注解都是用于动态启用某些功能的。而其底层原理是使用@Import注解导入一些配置类，实现Bean的动态加载。

比如@EnableAutoConfiguration注解

```

72     * @see AutoConfigureAfter
73     * @see SpringBootApplication
74     */
75     @Target(ElementType.TYPE)
76     @Retention(RetentionPolicy.RUNTIME)
77     @Documented
78     @Inherited
79     @AutoConfigurationPackage
80     @Import(AutoConfigurationImportSelector.class)
81     public @interface EnableAutoConfiguration {
82
83         String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";
84
85         /**
86          * Exclude specific auto-configuration classes such that they will never be applied.
87          * @return the classes to exclude

```

## 2.2.1 环境准备

说明

创建两个项目 第一个项目提供一个User的bean和配置类UserConfig 都放到cn.itcast包下

第二个项目com.leyou包下创建引导类，获取User对象

注意两个项目的包故意做成不一样的！

步骤

第一步：创建项目springboot-other

New Module

Add as module to	<none>	
Parent	<none>	
GroupId	cn.itcast	<input checked="" type="checkbox"/> Inherit
ArtifactId	springboot-other	
Version	1.0-SNAPSHOT	<input checked="" type="checkbox"/> Inherit

添加springboot最基础的启动器

```

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <version>2.1.3.RELEASE</version>
  </dependency>
</dependencies>

```

在cn.itcast.pojo包下创建实体类User

```
package cn.itcast.pojo;
public class User {
}
```

在cn.itcast.config包下创建配置类UserConfig

```
package cn.itcast.config;

import cn.itcast.pojo.User;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration //声明是配置类
public class UserConfig {
    @Bean //实例化对象
    public User user(){
        return new User();
    }
}
```

第二步：创建项目springboot-import

 New Module

Add as module to <none>

Parent <none>

GroupId com.leyou

ArtifactId springboot-import

Version 1.0-SNAPSHOT

添加springboot基本的启动器的依赖，并且添加springboot-other的依赖

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
        <version>2.1.3.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>cn.itcast</groupId>
        <artifactId>springboot-other</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>
</dependencies>
```

在com.leyou包下创建引导类获取springboot-other项目中的User对象

```
package com.leyou;

import cn.itcast.pojo.User;
import org.springframework.boot.SpringApplication;
```

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;

@SpringBootApplication
public class MyApplication {
    public static void main(String[] args) {
        //run方法后初始化容器
        ConfigurableApplicationContext app =
        SpringApplication.run(MyApplication.class, args);
        //    从容器中获取user对象
        User user = app.getBean(User.class);
        System.out.println(user); //验证是否能获取到user对象
    }
}
```

上面的代码肯定是获取不到的，因为@SpringBootApplication上的@ComponentScan只扫描了com.leyou,而第三方的对象在cn.itcast包下，所以是无法创建User对象的

我们可以在引导类上使用@ComponentScan (“cn.itcast”) 即可

```
@SpringBootApplication
@ComponentScan("cn.itcast")
public class MyApplication {
    public static void main(String[] args) {
        //run方法后初始化容器
        ConfigurableApplicationContext app = SpringApplication.run(MyApplication.class, args);
        //    从容器中获取user对象
        User user = app.getBean(User.class);
        System.out.println(user); //验证是否能获取到user对象
    }
}
```

但是在最项目时其实会用到很多非spring提供的包，都是用@ComponentScan扫描那么在引导类上会出现很多的，有没有其他方式呢？当然有，那就是@Import

## 2.2.2 @Import的使用

@Import的4种用法都可以把第三方包的类创建对象

第一种用法：引导类上 @Import(User.class),获取的时候需要使用类型方式

```
@SpringBootApplication
@Import(User.class)
public class MyApplication {
    public static void main(String[] args) {
        //run方法后初始化容器
        ConfigurableApplicationContext app = SpringApplication.run(MyApplication.class, args);
        //    从容器中获取user对象
        User user = app.getBean(User.class);
        System.out.println(user); //验证是否能获取到user对象
    }
}
```

第二种用法： @Import(UserConfig.class)

```

@SpringBootApplication
@Import(UserConfig.class)
public class MyApplication {
    public static void main(String[] args) {
        //run方法后初始化容器
        ConfigurableApplicationContext app = SpringApplication.run(MyApplication.class, args);
        // 从容器中获取user对象
        User user = app.getBean(User.class);
        System.out.println(user); //验证是否能获取到user对象
    }
}

```

还可以封装@Import

在springboot-other中创建一个@EnableUser注解，这个注解上添加@Import注解

```

package cn.itcast.config;

import org.springframework.context.annotation.Import;

import java.lang.annotation.*;

@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(UserConfig.class)
public @interface EnableUser {
}

```

在springboot-import项目的引导类中添加@EnableUser

```

@SpringBootApplication
@EnableUser
public class MyApplication {
    public static void main(String[] args) {
        //run方法后初始化容器
        ConfigurableApplicationContext app = SpringApplication.run(MyApplication.class, args);
        // 从容器中获取user对象
        User user = app.getBean(User.class);
        System.out.println(user); //验证是否能获取到user对象
    }
}

```

第三种方式：@Import(ImportSelector的实现类),实现类实现接口的一个方法（最常用）

在springboot-other项目中创建一个ImportSelector的实现类

```

package cn.itcast.config;

import org.springframework.context.annotation.ImportSelector;
import org.springframework.core.type.AnnotationMetadata;

public class MyImportSelector implements ImportSelector {
    public String[] selectImports(AnnotationMetadata annotationMetadata) {
        return new String[]{"cn.itcast.pojo.User"};
    }
}

```

在springboot-import项目的引导类中添加@Import(MyImportSelector.class)

```

@SpringBootApplication
@Import(MyImportSelector.class)
public class MyApplication {
    public static void main(String[] args) {
        //run方法后初始化容器
        ConfigurableApplicationContext app = SpringApplication.run(MyApplication.class, args);
        // 从容器中获取user对象
        User user = app.getBean(User.class);
        System.out.println(user); //验证是否能获取到user对象
    }
}

```

第四种方式：@Import(ImportBeanDefinitionRegistrar的实现类)

和第三种方式差不多

在springboot-other项目中创建一个ImportBeanDefinitionRegistrar的实现类

```

package cn.itcast.config;

import cn.itcast.pojo.User;
import org.springframework.beans.factory.support.AbstractBeanDefinition;
import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.support.BeanDefinitionRegistry;
import org.springframework.context.annotation.ImportBeanDefinitionRegistrar;
import org.springframework.core.type.AnnotationMetadata;

public class MyImportBeanDefinitionRegistrar implements
ImportBeanDefinitionRegistrar {
    public void registerBeanDefinitions(AnnotationMetadata annotationMetadata,
        BeanDefinitionRegistry beanDefinitionRegistry) {
        AbstractBeanDefinition definition =
        BeanDefinitionBuilder.genericBeanDefinition(User.class).getBeanDefinition();
        beanDefinitionRegistry.registerBeanDefinition("user",definition);//把定义
        好的bean注册进去
    }
}

```

在springboot-import项目的引导类中添加@Import(MyImportBeanDefinitionRegistrar.class)

```

@SpringBootApplication
@Import(MyImportBeanDefinitionRegistrar.class)
public class MyApplication {
    public static void main(String[] args) {
        //run方法后初始化容器
        ConfigurableApplicationContext app = SpringApplication.run(MyApplication.class, args);
        // 从容器中获取user对象
        User user = app.getBean(User.class);
        System.out.println(user); //验证是否能获取到user对象
    }
}

```

## 2.3 小结

@EnableAutoConfiguration 注解内部使用 @Import(AutoConfigurationImportSelector.class)来加载配置类。

配置文件位置：META-INF/spring.factories，该配置文件中定义了大量的配置类，当 SpringBoot 应用启动时，会自动加载这些配置类，初始化Bean

并不是所有的Bean都会被初始化，在配置类中使用Condition来加载满足条件的Bean

## 3. 学习自定义启动器

有了上面的一些知识点，我们可以自定义一个发送邮件的启动器 email-spring-boot-starter 我们参考mybatis的启动器，因为mybatis是第三方的（不是spring提供的）

参考后步骤如下：

- ①创建 email-spring-boot-autoconfigure 模块
- ②创建 email-spring-boot-starter 模块,依赖 email-spring-boot-autoconfigure的模块
- ③在email-spring-boot-autoconfigure 模块中初始化EmailTempalte 的 Bean。并定义META-INF/spring.factories 文件
- ④在测试模块中引入自定义的email-spring-boot-starter 依赖，测试获取 EmailTempalte 的Bean，发送邮件。

详细步骤如下：

第一步：

创建email-spring-boot-autoconfigure 项目，添加springboot最基础的启动器、发邮件必须有的javax.mail依赖

然后把项目一中用到的发送邮件的工具类改造后放入到项目中的com.itheima.tempalte包下

1、创建项目



New Module	
Add as module to	<none>
Parent	<none>
GroupId	com.itheima
ArtifactId	email-spring-boot-autoconfigure
Version	1.0-SNAPSHOT

## 2、导出依赖

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <version>2.1.3.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>javax.mail</groupId>
    <artifactId>mail</artifactId>
    <version>1.4</version>
  </dependency>
</dependencies>
```

3、我们模仿spring，把MailUtil改名为EmailTemplate(显得高级^.^),因为spring中提供了好多的xxxTemplate

```
package com.itheima.template;

import javax.mail.Address;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;
import java.util.Properties;

public class EmailTemplate {

    private String host;
    private String from;
    private String password;

    public EmailTemplate(String host, String from, String password) {
        this.host = host;
        this.from = from;
        this.password = password;
    }

    /**
     *
     * @param to 收件人
     * @param subject 主题
     * @param content 内容
     * @throws Exception
     */
}
```

```

//实现邮件发送的方法
public void sendMsg(String to ,String subject ,String content) throws
Exception{
    Properties props = new Properties();
    props.setProperty("mail.smtp.host", host); //设置主机地址 smtp.qq.com
    smtp.sina.com
    props.setProperty("mail.smtp.auth", "true");//授权认证 代码客户端访问 必须设置
    为true 需要手机验证
    //2.产生一个用于邮件发送的Session对象
    Session session = Session.getInstance(props);
    //3.产生一个邮件的消息对象
    MimeMessage message = new MimeMessage(session);
    //4.设置消息的发送者
    Address fromAddr = new InternetAddress(from);
    message.setFrom(fromAddr);

    //5.设置消息的接收者
    Address toAddr = new InternetAddress(to);
    //TO 直接发送 CC抄送 BCC密送
    message.setRecipient(MimeMessage.RecipientType.TO, toAddr);

    //6.设置主题
    message.setSubject(subject);
    //7.设置正文
    message.setText(content);
    //8.准备发送，得到火箭
    Transport transport = session.getTransport("smtp");
    //9.设置火箭的发射目标
    transport.connect(host, from, password); //密码 授权密码!=登陆密码
    //10.发送
    transport.sendMessage(message, message.getAllRecipients());
    //11.关闭
    transport.close();
}
}

```

4、在创建EmailTemplate对象时一定要设置三个参数，这三个参数可以在配置文件中配置，所以我们专门创建一个EmailProperties类从配置文件中获取参数

```

package com.ithema.config;

import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties(prefix = "email") //前缀是email
public class EmailProperties {
    private String host;
    private String from;
    private String password;
    setter...getter...
}

```

5、准备一个配置类，创建EmailTemplate对象

```

package com.ithema.config;

```

```

import com.itheima.template.EmailTemplate;
import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;
import org.springframework.boot.context.properties.ConfigurationProperties;
import
org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import sun.security.action.PutAllAction;

import javax.mail.internet.MimeMessage;

@Configuration
@EnableConfigurationProperties(EmailProperties.class) //开启配置类
public class EmailAutoConfigure {
    @Bean
    @ConditionalOnClass(MimeMessage.class)
    public EmailTemplate emailTemplate(EmailProperties properties){ //开启后可以注
入到方法上使用
        return new
EmailTemplate(properties.getHost(),properties.getFrom(),properties.getPassword()
);
    }
}

```

第二步：创建email-spring-boot-starter, email-spring-boot-autoconfigure的模块

### 1、创建项目

 New Module

Add as module to	<none>
Parent	<none>
GroupId	com.itheima
ArtifactId	email-spring-boot-starter
Version	1.0-SNAPSHOT

### 2、添加依赖

```

<dependencies>
    <dependency>
        <groupId>com.itheima</groupId>
        <artifactId>email-spring-boot-autoconfigure</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>
</dependencies>

```

第三步：在其他项目中导入email-spring-boot-starter的依赖进行测试

比如在springboot-import项目中

### 1、添加依赖

```
<dependency>
  <groupId>com.itheima</groupId>
  <artifactId>email-spring-boot-starter</artifactId>
  <version>1.0-SNAPSHOT</version>
</dependency>
```

2、添加配置文件application.yml,内容如下

```
email:
  host: smtp.sina.com
  from: 一个新浪的邮箱
  password: 密码
```

3、在引导类上测试

```
@SpringBootApplication
public class MyApplication {
    public static void main(String[] args) throws Exception{
        //run方法后初始化容器
        ConfigurableApplicationContext app =
        SpringApplication.run(MyApplication.class, args);
        // 从容器中获取EmailTemplate对象
        EmailTemplate emailTemplate = app.getBean(EmailTemplate.class);
        emailTemplate.sendMsg("接收邮件的一个邮箱地址","你看一下能不能收到","自定义了一个和springboot整合的项目");
    }
}
```

看收到的



自定义了一个和springboot整合的项目

## 4. SpringBoot监控（了解）

### 4.1 Actuator监控

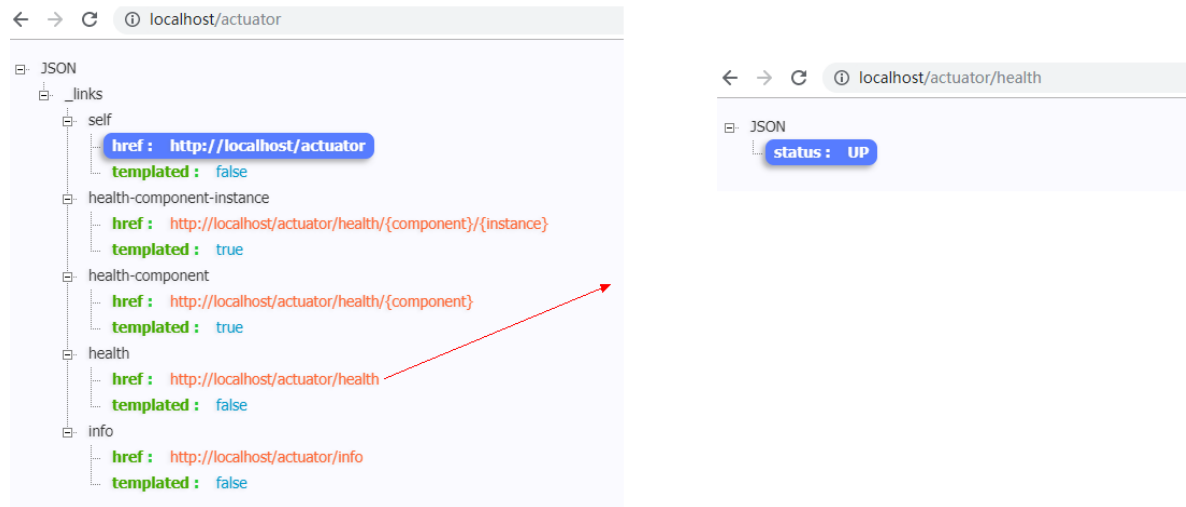
SpringBoot自带监控功能Actuator，可以帮助实现对程序内部运行情况监控，比如监控状况、Bean加载情况、配置属性、日志信息等。

在第一天的springboot-demo项目中演示

①导入依赖坐标

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

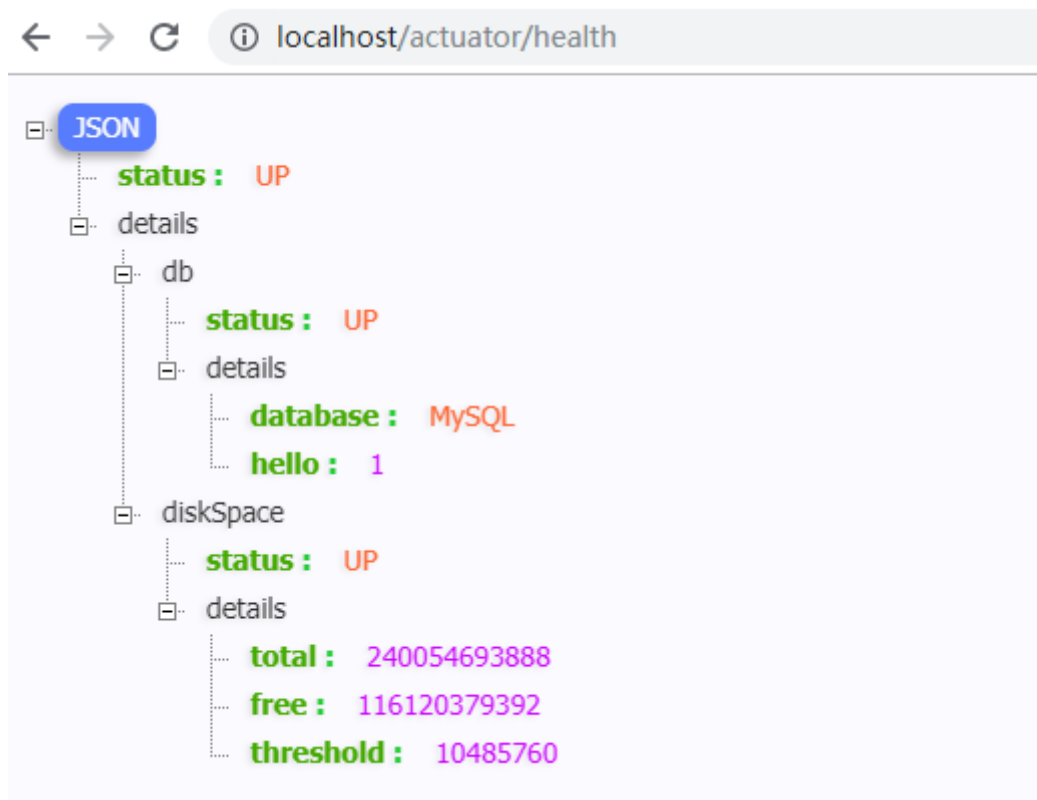
②访问<http://localhost/acruator> springboot-demo当时设置的端口号是80，可以省略的



如果想看更详细的health信息信息，在配置文件中添加以下代码

```
management:
  endpoint:
    health:
      show-details: always
```

重启项目后再访问



④ 如果想看更多的信息，在配置文件中添加以下代码

```
management:
  endpoint:
    health:
      show-details: always
  endpoints:
    web:
      exposure:
        include: "*"

```

重启后访问 <http://localhost/actuator>，发现多了很多东西

重点关注3个

<http://localhost/actuator/beans>

<http://localhost/actuator/env>

<http://localhost/actuator/mappings>

具体详细的解释：

路径	描述
/beans	描述应用程序上下文里全部的Bean，以及它们的关系
/env	获取全部环境属性
/env/{name}	根据名称获取特定的环境属性值
/health	报告应用程序的健康指标，这些值由HealthIndicator的实现类提供
/info	获取应用程序的定制信息，这些信息由info打头的属性提供
/mappings	描述全部的URI路径，以及它们和控制器(包含Actuator端点)的映射关系
/metrics	报告各种应用程序度量信息，比如内存用量和HTTP请求计数
/metrics/{name}	报告指定名称的应用程序度量值
/trace	提供基本的HTTP请求跟踪信息(时间戳、HTTP头等)

## 4.2 Spring Boot Admin

actuator的监控内容够详细，但是阅读性比较差，所以可以使用Spring Boot Admin提供一个可视化的界面查阅信息，Spring Boot Admin是一个开源社区项目，用于管理和监控SpringBoot应用程序。

Spring Boot Admin 有两个角色，客户端(Client)和服务端(Server)。

应用程序作为Spring Boot Admin Client向Spring Boot Admin Server注册

Spring Boot Admin Server 的界面将Boot Admin ClientActuatorEndpoint

**开发步骤如下：**

admin-server:

- ①创建 admin-server 模块
- ②导入依赖坐标 admin-starter-server
- ③在引导类上启用监控功能@EnableAdminServer

admin-client: 自己创建的项目就是所谓的client端

①创建 admin-client 模块

②导入依赖坐标 admin-starter-client

③配置相关信息: server地址等

④启动server和client服务, 访问server

**功能演示:**

第一步: 创建admin-server端

1、创建项目



New Module

Add as module to <none>

Parent <none>

GroupId com.itheima

ArtifactId springboot-admin-server

Version 1.0-SNAPSHOT

2、导入依赖

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.3.RELEASE</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.1.3.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-server</artifactId>
    <version>2.1.3</version>
  </dependency>
</dependencies>
```

3、创建引导类

```
package com.itheima;

import de.codecentric.boot.admin.server.config.EnableAdminServer;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
@EnableAdminServer //注意这里需要开启注解
public class SpringbootAdminServerApplication {
```

```
public static void main(String[] args) {  
    SpringApplication.run(SpringbootAdminServerApplication.class, args);  
}  
}
```

第二步：把springboot-demo项目作为client端

1、在springboot-demo项目中添加依赖

```
<dependency>  
  <groupId>de.codecentric</groupId>  
  <artifactId>spring-boot-admin-starter-client</artifactId>  
  <version>2.1.3</version>  
</dependency>
```

2、发布到server端,在application.yml中添加

spring.boot.admin.client.url=<http://localhost:8080>

```
server:  
  port: 80  
spring:  
  datasource:  
    url: jdbc:mysql://localhost:3306/springboot  
    username: root  
    password: root  
    driver-class-name: com.mysql.jdbc.Driver  
    type: com.zaxxer.hikari.HikariDataSource  
  boot:  
    admin:  
      client:  
        url: http://localhost:8080
```

第三步：两个项目启动（先启动server）

访问项目 <http://localhost:8080/#/applications> 查阅信息即可

## 5. SpringBoot项目的部署

SpringBoot 项目开发完毕后，支持两种方式部署到服务器：

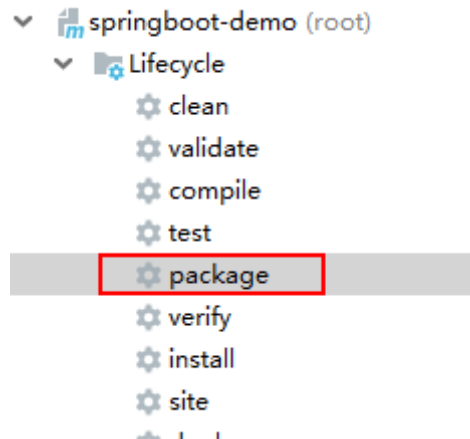
### 5.1 jar包(官方推荐)

第一步：在项目中添加一个插件



```
<build>
  <!-- 最后导出jar或war的名称-->
  <finalName>springboot-demo</finalName>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

第二步：执行package命令打出一个jar



第三步：直接使用java -jar 运行jar项目

把打出的jar放入到一个没有中文没有空格的位置(我直接放到了D盘根目录)执行java -jar

```
CA\Windows\System32\cmd.exe - java -jar springboot-demo.jar
Microsoft Windows [版本 10.0.17134.1130]
(c) 2018 Microsoft Corporation. 保留所有权利。

D:\>java -jar springboot-demo.jar

Spring
=====
:: Spring Boot ::      (v2.1.3.RELEASE)

2019-12-07 12:08:59.389 INFO 15756 --- [        main] com.leyou.Application : Starting Application v1.0-SNAPSHOT on DESKTOP-8V9MV76 with PID
6 (D:\springboot-demo.jar started by syl in D:\)
2019-12-07 12:08:59.393 INFO 15756 --- [        main] com.leyou.Application : No active profile set, falling back to default profiles: default
2019-12-07 12:09:00.655 INFO 15756 --- [        main] trationDelegate$BeanPostProcessorChecker : Bean 'org.springframework.transaction.annotation.ProxyTransaction
managementConfiguration' of type [org.springframework.transaction.annotation.ProxyTransactionManagementConfiguration$$EnhancerBySpringOGLIB$$e5154206] is not eligible
getting processed by all BeanPostProcessors (for example: not eligible for auto-proxying)
2019-12-07 12:09:01.467 INFO 15756 --- [        main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 80 (http)
2019-12-07 12:09:01.491 INFO 15756 --- [        main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2019-12-07 12:09:01.491 INFO 15756 --- [        main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.16]
2019-12-07 12:09:01.502 INFO 15756 --- [        main] o.a.catalina.core.AprLifecycleListener : The APR based Apache Tomcat Native library which allows optimal
formance in production environments was not found on the java.library.path: [D:\ITCAST\jdk1.8\bin;C:\WINDOWS\Sun\Java\bin;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\
em32;C:\WINDOWS.C:\WINDOWS\System32\Wbem;C:\WINDOWS\System32\WindowsPowerShell\v1.0\;D:\ITCAST\jdk1.8\bin;D:\softwareinstall\Git\cmd;D:\softwareinstall\TortoiseGit
.D\class101\apache-maven-3.5.2\bin;C:\WINDOWS\System32\OpenSSH\;D:\softwareinstall\nodejs\;D:\softwareinstall\SVNServer\bin;C:\Program Files\TortoiseSVN\bin;C:\Use
yl\AppData\Local\Microsoft\WindowsApps;C:\Users\syl\AppData\Roaming\npm;D:\softwareinstall\MicrosoftVSCode\bin;C:\Users\syl\AppData\Local\Pandoc\...]
2019-12-07 12:09:01.603 INFO 15756 --- [        main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2019-12-07 12:09:01.603 INFO 15756 --- [        main] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 2164 ms
2019-12-07 12:09:02.572 INFO 15756 --- [        main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService applicationTaskExecutor
2019-12-07 12:09:02.990 INFO 15756 --- [        main] t.m.m.autoconfigure.MapperCacheDisabler : Clear tk.mybatis.mapper.util.HsUtil CLASS CACHE cache.
2019-12-07 12:09:02.991 INFO 15756 --- [        main] t.m.m.autoconfigure.MapperCacheDisabler : Clear tk.mybatis.mapper.genid.GenIdUtil CACHE cache.
2019-12-07 12:09:02.992 INFO 15756 --- [        main] t.m.m.autoconfigure.MapperCacheDisabler : Clear tk.mybatis.mapper.version.VersionUtil CACHE cache.
2019-12-07 12:09:02.992 INFO 15756 --- [        main] t.m.m.autoconfigure.MapperCacheDisabler : Clear EntityHelper entityTableMap cache.
2019-12-07 12:09:03.001 INFO 15756 --- [        main] o.s.b.a.e.web.EndpointLinksResolver : Exposing 15 endpoint(s) beneath base path '/actuator'
2019-12-07 12:09:03.087 INFO 15756 --- [        main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 80 (http) with context path ''
2019-12-07 12:09:03.090 INFO 15756 --- [        main] com.leyou.Application : Started Application in 4.18 seconds (JVM running for 4.497)
```

成功！

## 5.2 war包

第一步：项目的打包方式改为war



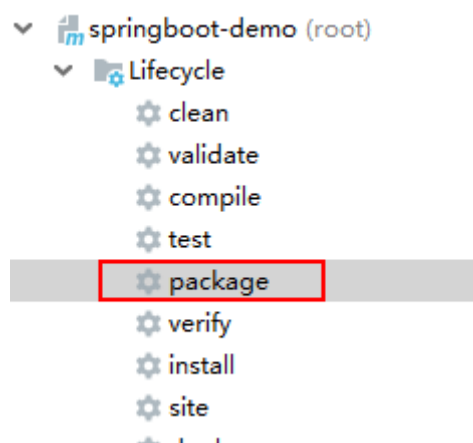
第二步：在引导类上修改代码

1、继承一个父类 SpringBootServletInitializer 2、添加一个方法 configure方法

```
@SpringBootApplication
@MapperScan("com.leyou.mapper")
public class Application extends SpringBootServletInitializer {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

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

第三步：执行package命令打出一个war包



第四步：放入到一个tomcat的webapps目录中运行即可