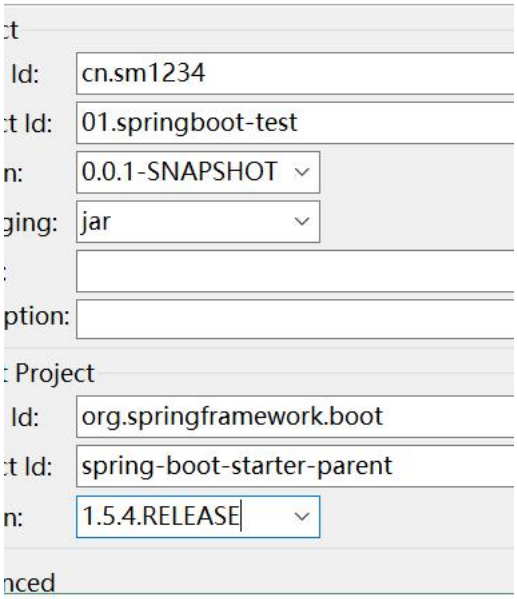
本章学习目标

- SpringBoot 单元测试
- 热部署
- SpringBoot 异常处理
- SpringBoot 表单验证
- 整合 Spring Data JPA
- SpringBoot 缓存实现
- SpringBoot 实现定时任务

1. SpringBoot 单元测试

1.1. 建立 Spring Boot 项目



pom.xml:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
```

```
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.4.RELEASE</version>
  </parent>
  <groupId>cn.sm1234</groupId>
  <artifactId>01.springboot-test</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <dependencies>
    <!-- spring 支持 -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
  </dependencies>

  <properties>
    <java.version>1.8</java.version>
  </properties>
</project>
```

1.2. 导入 test 的坐标

```
<!-- junit 测试支持 -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
    </dependency>
```

1.3. 编写 Dao 和 Service 类

Dao：

```
package cn.sm1234.dao;


import org.springframework.stereotype.Repository;


@Repository
public class UserDao {


    public void save(){
        System.out.println("UserDao.save()");
    }
}
```

Service：

```
package cn.sm1234.service;


import javax.annotation.Resource;


import org.springframework.stereotype.Service;


import cn.sm1234.dao.UserDao;


@Service
public class UserService {
    @Resource
    private UserDao userDao;
```

```
    public void save(){
        userDao.save();
    }
}
```

1.4. 编写 SpringBoot 启动器类

```
package cn.sm1234;


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


/**
 * SpringBoot 启动器类
 * @author lenovo
 *
 */
@SpringBootApplication
public class Application {


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

1.5. 编写测试类（重点）

```
package cn.sm1234.test;


import javax.annotation.Resource;


import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;


import cn.sm1234.Application;

import cn.sm1234.service.UserService;


/**

 * Spring Boot 测试类

 * @author lenovo

 *

 */

@RunWith(SpringJUnit4ClassRunner.class)  // @RunWith: 让 junit 和 Spring 环境进行整合

@SpringBootTest(classes={Application.class})     // @SpringBootTest: 该类是一个

SpringBoot 测试类，加载 SpringBoot 启动器类

// Spring:  @ContextConfiguation("classpath:applicationContext.xml")

public class UserServiceTest {


    @Resource

    private UserService userService;


    @Test
```

```
    public void testSave(){
        userService.save();
    }
}
```

2. SpringBoot 热部署

SpringBoot 热部署分为两种：

1）SpringLoader 插件

2）devtools 工具

2.1. 搭建环境

pom.xml:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.4.RELEASE</version>
    </parent>
    <groupId>cn.sm1234</groupId>
    <artifactId>02.springboot-springload</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <dependencies>
```

```
        <!-- web 依赖 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <!-- 导入 thymeleaf 坐标 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
    </dependencies>

    <properties>
        <java.version>1.8</java.version>
        <thymeleaf.version>3.0.2.RELEASE</thymeleaf.version>
        <thymeleaf-layout-dialect.version>2.0.4</thymeleaf-layout-dialect.version>
    </properties>
</project>
```

2.1.1. 编写 Controller

```
package cn.sm1234.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

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

```
    /**
     * 跳转到 list.jsp
     */
    @RequestMapping("/list")
    public String list(){
        System.out.println("run list...");
        return "list";
    }
}
```

2.1.2. 建立 list.html 页面

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>用户列表页面</title>
</head>
<body>
用户列表页面
</body>
</html>
```

2.1.3. 编写 SpringBoot 启动类

```
package cn.sm1234;
```

```
import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;



@SpringBootApplication

public class Application {



    public static void main(String[] args) {

        SpringApplication.run(Application.class, args);

    }

}
```

2.2. 使用 Springloader 进行热部署

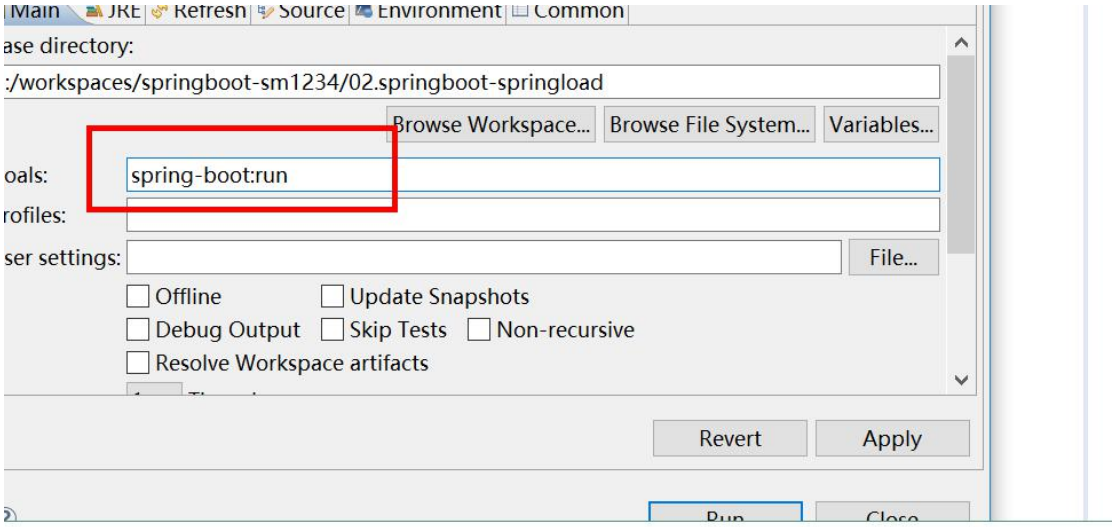
2.2.1. 方式一：Maven 插件使用方法

- 在 pom.xml 导入以下插件内容：

```
<!-- 导入 springloader 插件 -->

    <build>

        <plugins>

            <plugin>

                <groupId>org.springframework.boot</groupId>

                <artifactId>spring-boot-maven-plugin</artifactId>

                <dependencies>

                    <dependency>

                        <groupId>org.springframework</groupId>

                        <artifactId>springloaded</artifactId>

                        <version>1.2.5.RELEASE</version>

                    </dependency>

                </dependencies>
```

```
            </plugin>
        </plugins>
    </build>
```

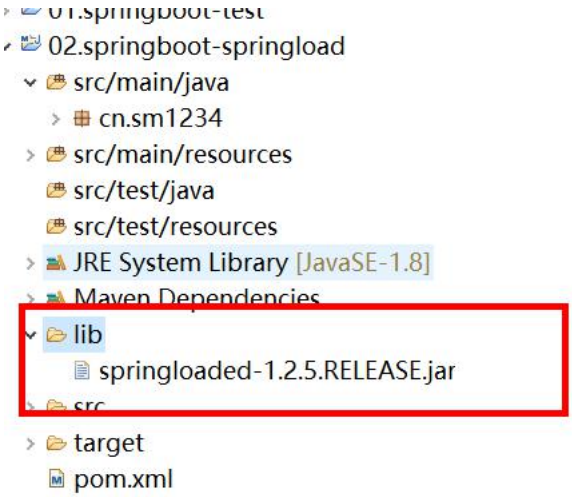
- 使用 spring-boot:run 的 maven 命令进行启动



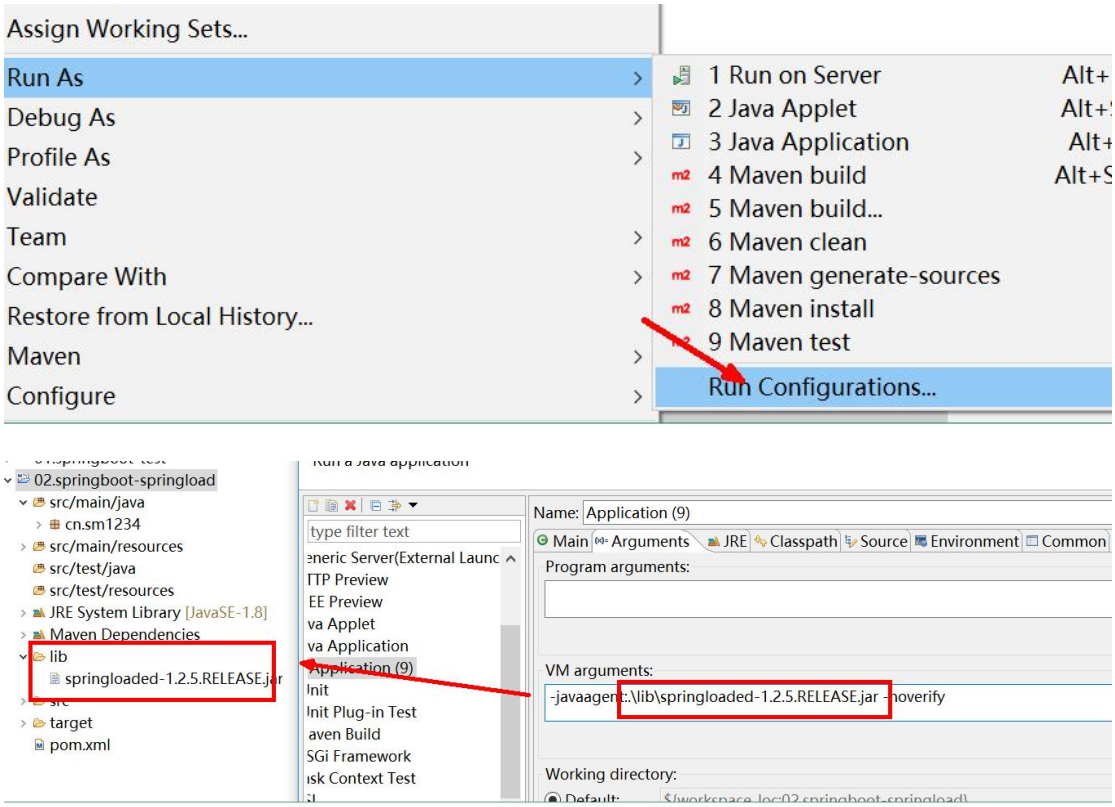
注意：这种方式的缺点是程序启动后，在系统后台开启进程，而且需要手动杀死。

2.2.2. 方式二：直接使用 jar 包

2.2.2.1. 下载 springloader 的 jar 包，导入项目



2.2.2.2. 运行程序，加入 JVM 参数




```
-javaagent:.\lib\springloaded-1.2.5.RELEASE.jar -noverify
```

2.3. devtools 工具

2.3.1. 导入 devtools 的坐标

```
<!-- 导入 devtools 的坐标 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <optional>true</optional>
            <scope>true</scope>
        </dependency>
```

2.3.2. 启动 SpringBoot 启动类测试

3. SpringBoot 异常处理

SpringBoot 的异常处理分为 5 种情况：

1）自定义 error 的错误页面

2）@ExceptionHandler 注解方法

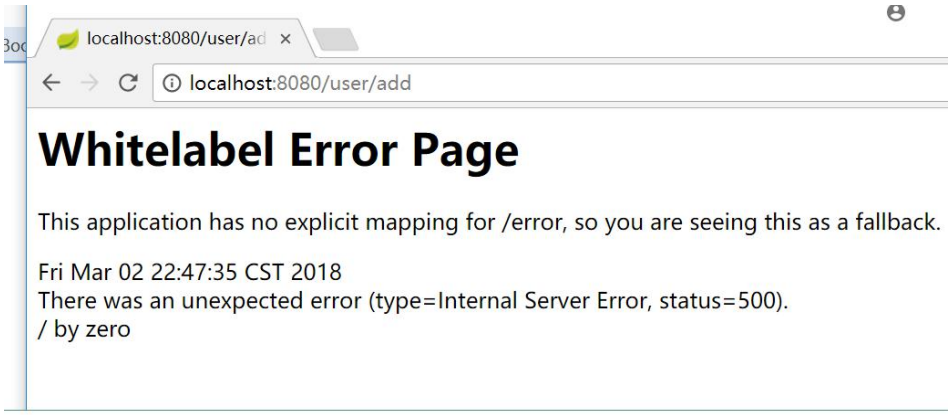
3）@ControllerAdvice+@ExceptionHandler

4）配置 SimpleMappingExceptionResolver 类

5）自定义 HandlerExceptionResovler 类

3.1. 自定义 error 的错误页面

我们要知道的是 SpringBoot 应用默认已经提供一套错误处理机制：把所有后台错误统一交给 error 请求，然后跳转到了本身自己的错误提示页面。



这时，如果需要修改提示页面，我们可以在 templates 目录直接建立 error.html 页面即可。

```
<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>自定义错误页面</title>

</head>
```

```
<body>

<h3>错误页面</h3>

<div th:text="${exception}"></div>

</body>

</html>
```

3.2. @ExceptionHandler 注解方法

3.2.1. Controller 自定义不同的@ExceptionHandler 注解方法

```
package cn.sm1234.controller;


import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.ExceptionHandler;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.servlet.ModelAndView;


@Controller

@RequestMapping("/user")

public class UserController {


    /**

     * 跳转到 list.jsp

     */

    @RequestMapping("/list")

    public String list() {

        System.out.println("run list1111111...");

        return "list";

    }
```

```
    @RequestMapping("/add")
    public String add() {
        // 模拟异常，运算异常
        int i = 100 / 0;
        return "list";
    }

    @RequestMapping("/update")
    public String update() {
        // 模拟异常，空指针异常
        String name = null;
        name.toLowerCase(); // 在这里发生空指针异常
        return "list";
    }

    // 处理 java.lang.ArithmeticException
    @ExceptionHandler(value = { java.lang.ArithmeticException.class })
    public ModelAndView handlerArithmeticException(Exception e) { // e:该对象包含错误信息

        // 设置错误信息
        ModelAndView mv = new ModelAndView();
        mv.addObject("exception", e.toString());
        mv.setViewName("error1");
        return mv;
    }

    // 处理 java.lang.NullPointerException
```

```
    @ExceptionHandler(value = { java.lang.NullPointerException.class })
    public ModelAndView handlerNullPointerException(Exception e) { // e:该对象包含错
误信息

        // 设置错误信息
        ModelAndView mv = new ModelAndView();
        mv.addObject("exception", e.toString());
        mv.setViewName("error2");
        return mv;
    }
}
```

3.3. @ControllerAdvice+@ExceptionHandler

3.3.1. 自定义全局异常处理类

```
package cn.sm1234.exception;

import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.servlet.ModelAndView;

@ControllerAdvice
public class GlobalExceptionHandler {

    // 处理 java.lang.NullPointerException
    @ExceptionHandler(value = { java.lang.NullPointerException.class })
    public ModelAndView handlerNullPointerException(Exception e) { // e:该对象包含错
```

```
误信息


        // 设置错误信息
        ModelAndView mv = new ModelAndView();
        mv.addObject("exception", e.toString());
        mv.setViewName("error2");
        return mv;
    }

    // 处理 java.lang.ArithmeticException
    @ExceptionHandler(value = { java.lang.ArithmeticException.class })
    public ModelAndView handlerArithmeticException(Exception e) { // e:该对象包含错误
信息


        // 设置错误信息
        ModelAndView mv = new ModelAndView();
        mv.addObject("exception", e.toString());
        mv.setViewName("error1");
        return mv;
    }
}
```

3.4. 配置 SimpleMappingExceptionResolver

3.4.1. 编写 SimpleMappingExceptionResolver 配置类

```
package cn.sm1234.exception;


import java.util.Properties;
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.handler.SimpleMappingExceptionResolver;

@Configuration
public class MySimpleMappingExceptionResolver {

    @Bean
    public SimpleMappingExceptionResolver getSimpleMappingExceptionResolver(){
        SimpleMappingExceptionResolver resolver = new
SimpleMappingExceptionResolver();

        Properties mappings = new Properties();

        /**
         * 参数一：异常类型，注意必要要全名
         * 参数二:视图的名称
         */
        mappings.put("java.lang.ArithmeticException", "error1");
        mappings.put("java.lang.NullPointerException", "error2");

        //设置异常映射信息
        //哪些异常要交给哪个错误页面显示
        resolver.setExceptionMappings(mappings);

        return resolver;
    }
}
```

3.5. 自定义 HandlerExceptionResovler 类

```
package cn.sm1234.exception;


import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;


import org.springframework.context.annotation.Configuration;

import org.springframework.web.servlet.HandlerExceptionResolver;

import org.springframework.web.servlet.ModelAndView;


@Configuration

public class MyHandlerExceptionResolver implements HandlerExceptionResolver {


    @Override

    public ModelAndView resolveException(HttpServletRequest arg0, HttpServletResponse

arg1, Object arg2,

            Exception e) {

        ModelAndView mv = new ModelAndView();

        //判断不同异常类型，做不同处理

        if(e instanceof ArithmeticException){

            mv.setViewName("error1");

        }


        if(e instanceof NullPointerException){

            mv.setViewName("error2");

        }
```

```
        mv.addObject("exception", e.toString());
        return mv;
    }

}
```

4. SpringBoot 表单验证

原理：利用 hibernate-validate 的注解实现的

4.1. User 类

```
package cn.sm1234.domain;

import org.hibernate.validator.constraints.NotBlank;

public class User {

    private Integer id;
    @NotBlank(message="用户名不能为空") // 非空
    private String name;
    @NotBlank(message="密码不能为空") // 非空
    private String password;
    private Integer age;
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
```

```
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public Integer getAge() {
        return age;
    }
    public void setAge(Integer age) {
        this.age = age;
    }
    @Override
    public String toString() {
        return "User [id=" + id + ", name=" + name + ", password=" + password + ", age="
+ age + "]";
    }

}
```

4.2. UserController

```
package cn.sm1234.controller;


import javax.validation.Valid;


import org.springframework.stereotype.Controller;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.RequestMapping;


import cn.sm1234.domain.User;


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


    /**
     * 跳转到 add.html
     * @return
     */
    @RequestMapping("toAdd")
    public String toAdd(){
        return "add";
    }


    /**
     * 用户添加
     * BindingResult: 用于封装验证对象（user）里面的验证结果
     */
```

```
    @RequestMapping("add")
    public String add(@Valid User user,BindingResult result){
        //如果存在验证错误
        if(result.hasErrors()){
            //返回 add.html
            return "add";
        }

        System.out.println("保存用户:"+user);
        return "succ";
    }
}
```

4.3. 添加页面 add.html

在添加页面回显错误信息：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>用户添加</title>
</head>
<body>
<h3>用户添加</h3>
<form action="/user/add" method="post">
用户名：<input type="text" name="name"/><font color="red"
th:errors="${user.name}"></font><br/>
密码：<input type="password" name="password"/><font color="red"
```

```
th:errors="${user.password}"></font><br/>
年龄：<input type="text" name="age"/><br/>
<input type="submit" value="保存"/>
</form>
</body>
</html>
```

这时发生错误：add.html 页面无法绑定 user 对象

解决办法：

```
    /**
     * 跳转到 add.html
     * @return
     */
    @RequestMapping("toAdd")
    public String toAdd(User user){
        return "add";
    }
```

4.4. 常用的表单验证注解

@NotBlank: 判断字符串是否为 null 或空字符串（去掉两边的空格）

@NotEmpty：判断字符串是否为 null 或空字符串

@Length：判断字符串长度（包括最小或最大）

@Min：判断数值类型的最小值

@Max：判断数值类型的的最大值

@Email： 判断邮箱格式是否合法 www@qq.com

```
public class User {
```

```
    private Integer id;

    //@NotBlank(message="用户名不能为空") // 非空

    @NotEmpty(message="用户名不能为空")

    private String name;

    @NotBlank(message="密码不能为空") // 非空

    @Length(min=4,max=10,message="密码必须在 4-10 位之间")

    private String password;

    @Min(value=0)

    private Integer age;

    @Email(message="邮箱不合法")

    private String email;
```

5. 整合 Spring Data JPA

5.1. Spring Data JPA 简介

Spring Data 是 Spring 旗下的一个持久态简化的框架。

而 Spring Data JPA 是 Spring Data 项目的一个模块。

Spring Data JPA 的作用：简化 Spring 项目的基于 JPA 的开发，简化后的效果就是持久态不需要编写实现类，只需要接口即可！

5.2. SpringBoot 整合 SpringDataJPA

5.2.1. 导入 SpringDataJPA 坐标

```
<!-- 导入 SpringDataJPA 的坐标 -->

        <dependency>

            <groupId>org.springframework.boot</groupId>
```

```
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <!-- 连接数据库驱动 -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
        </dependency>
        <!-- 连接池 -->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid</artifactId>
            <version>1.0.9</version>
        </dependency>
```

5.2.2. 配置连接

在 resources 目录下建立 application.properties.

```
spring.datasource.driverClassName=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/test
spring.datasource.username=root
spring.datasource.password=root

spring.datasource.type=com.alibaba.druid.pool.DruidDataSource

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

5.2.3. 编写实体类

```
@Entity
```

```
@Table(name="t_emp")
public class Emp {

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

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

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

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

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

5.2.4. 编写 Dao 接口（*）

```
package cn.sm1234.dao;

import org.springframework.data.jpa.repository.JpaRepository;

import cn.sm1234.domain.Emp;
/**
 * 参数一：需要映射的实体
```

```
 * 参数二：实体里面的 OID 的类型
 * @author lenovo
 *
 */
public interface EmpRepository extends JpaRepository<Emp, Integer>{
}
```

5.2.5. 编写测试代码

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

```
package cn.sm1234.test;

import javax.annotation.Resource;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import cn.sm1234.Application;
import cn.sm1234.dao.EmpRepository;
import cn.sm1234.domain.Emp;

@RunWith(SpringJUnit4ClassRunner.class)
```

```
@SpringBootTest(classes=Application.class)
public class EmpRepositoryTest {


    @Resource
    private EmpRepository empRepository;


    @Test
    public void testSave(){
        Emp emp = new Emp();
        emp.setName("张三");
        emp.setGender("男");
        emp.setTelephone("13422223333");
        emp.setAddress("广州天河");


        empRepository.save(emp);
    }
}
```

5.3. Spring Data JPA 提供的核心接口

1）Repository 接口

2）CrudRepository 接口

3）PagingAndSortingRepository 接口

4）JpaRepository 接口

5）JPASpecificationExecutor 接口

5.4. Repository 接口

1）提供基于方法名称命名查询

2）提供基于@Query 查询与修改

5.4.1. 提供基于方法名称命名查询

```
public interface EmpRepository extends Repository<Emp, Integer>{

}
```

```
public interface EmpRepository extends Repository<Emp, Integer>{


    //查询 name(驼峰式名称)
    public List<Emp> findByName(String name);
    public List<Emp> findByNameAndGender(String name,String gender);
    public List<Emp> findByTelephoneLike(String telphone);
}
```

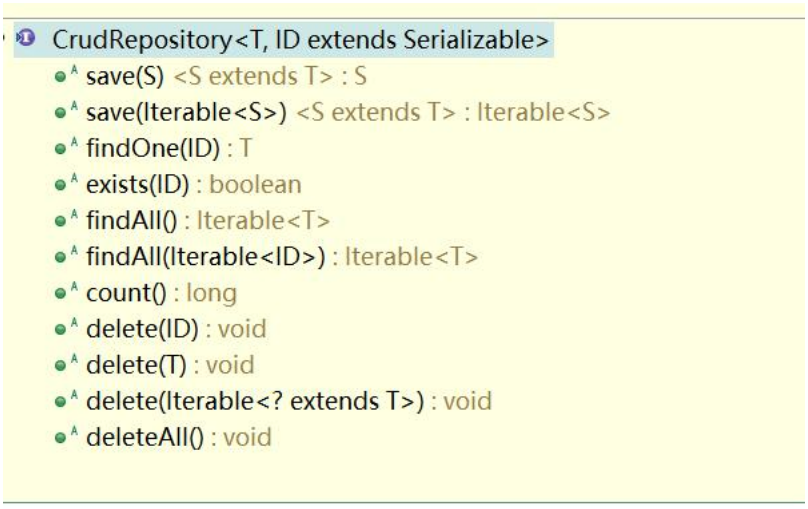
5.4.2. 提供基于@Query 查询与修改

```
    @Query("from Emp where name = ?")
    public List<Emp> queryName(String name);


    @Query(value="select * from t_emp where name = ?",nativeQuery=true)
    public List<Emp> queryName2(String name);


    @Query("update Emp set address = ? where id = ?")
    @Modifying // 进行修改操作
    public void updateAddressById(String address,Integer id);
```

5.5. CrudRepository 接口

CrudRepository 接口，作用是增删改查的基本操作。

注意：CrudRepository 接口继承了 Repository 接口



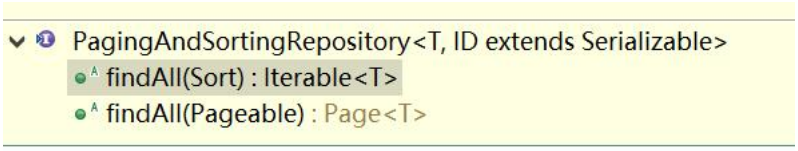

```
//添加
    @Test
    public void testSave(){
        Emp emp = new Emp();
        emp.setName("王五 666");
        emp.setGender("女");
        emp.setTelephone("1346666677777");
        emp.setAddress("广州番禺");

        empRepository.save(emp);
    }

    //修改
    @Test
    public void testUpdate(){
        Emp emp = new Emp();
        emp.setId(4);
```

```
        emp.setName("王五 6666");
        emp.setGender("女");
        emp.setTelephone("1346666677777");
        emp.setAddress("广州白云");

        empRepository.save(emp);
    }

    //查询所有数据
    @Test
    public void testFindAll(){
        //强制转换
        List<Emp> list = (List<Emp>)empRepository.findAll();
        for (Emp emp : list) {
            System.out.println(emp);
        }
    }
    //查询一个对象
    @Test
    public void testFineOne(){
        //强制转换
        Emp emp = empRepository.findOne(4);
        System.out.println(emp);
    }
    //删除一个对象
    @Test
    public void testDelete(){
        empRepository.delete(4);
    }
```

5.6. PagingAndSortingRepository 接口

PagingAndSortingRepository 接口，作用用于分页和排序查询。

注意：PagingAndSortingRepository 接口继承了 CrudRepository 接口




```
// 排序
    @Test
    public void testSort() {
        // 封装排序条件的对象
        Sort sort = new Sort(new Order(Direction.DESC, "id"));

        Iterable<Emp> list = empRepository.findAll(sort);
        for (Emp emp : list) {
            System.out.println(emp);
        }
    }

    // 分页
    @Test
    public void testPagealbe() {
        // Pageable：用于封装分页参数 。 当前页码和查询记录数（注意：当前页码从 0 开始的）
        Pageable pageable = new PageRequest(1, 2);
        // 封装排序条件的对象
        // Page: 用于封装分页查询后的结果
        Page<Emp> pageData = empRepository.findAll(pageable);

        System.out.println("总记录数：" + pageData.getTotalElements());
```

```
        List<Emp> content = pageData.getContent();
        for (Emp emp : content) {
            System.out.println(emp);
        }
        System.out.println("总页数: " + pageData.getTotalPages());
    }

    //排序+ 分页
    @Test
    public void testSortAndPagealbe() {
        // Pageable：用于封装分页参数 。 当前页码和查询记录数（注意：当前页码从 0 开始的）
        Sort sort = new Sort(new Order(Direction.DESC,"id"));

        Pageable pageable = new PageRequest(0, 2,sort);
        // 封装排序条件的对象
        // Page: 用于封装分页查询后的结果
        Page<Emp> pageData = empRepository.findAll(pageable);

        System.out.println("总记录数: " + pageData.getTotalElements());
        List<Emp> content = pageData.getContent();
        for (Emp emp : content) {
            System.out.println(emp);
        }
        System.out.println("总页数: " + pageData.getTotalPages());
    }
```

5.7. JpaRepository 接口

JpaRepository 接口，这个接口的作用主要继承 PagingAndSortingRepository 接口。

但这个接口还有额外的小功能，对之前接口的方法进行适配。

```
// 查询所有数据
@Test
public void testFindAllJpaRepository() {
    // 不需要强制转换
    List<Emp> list = empRepository.findAll();
    for (Emp emp : list) {
        System.out.println(emp);
    }
}
```

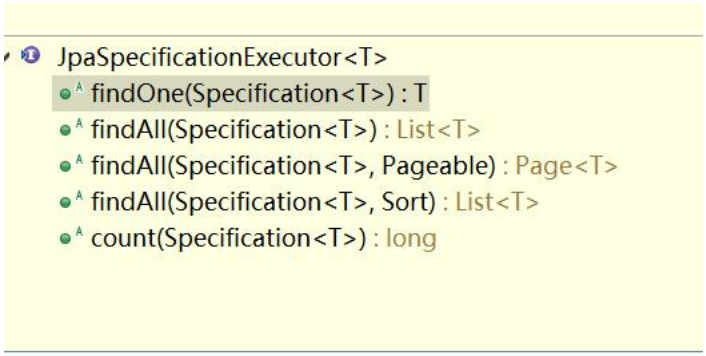
注意：我们在实际开发中通常我们的接口都是继承 JpaRepository 接口。

5.8. JpaSpecificationExecutor 接口

5JpaSpecificationExecutor 接口，作用是用于（组合）条件查询（条件+分页）。

注意：JpaSpecificationExecutor 接口是独立。

```
public interface EmpRepository extends JpaRepository<Emp,
Integer>,JpaSpecificationExecutor<Emp>{
```



5.8.1. 1 个条件

```
@Test
    public void testFindAllJPASpecificationExecutor() {
        /**
         * Specification:  用于封装条件数据的对象
         */
        Specification<Emp> spec = new Specification<Emp>() {
            //Predicate:该对象用于封装条件
            /**
             * Root<Emp> root :  根对象，用于查询对象的属性
             * CriteriaQuery<?> query:  执行普通的查询
             * CriteriaBuilder cb: 查询条件构造器，用于完成不同条件的查询
             */
            @Override
            public Predicate toPredicate(Root<Emp> root, CriteriaQuery<?> query,
CriteriaBuilder cb) {
                // where name = ?
                /**
                 * 参数一：查询的属性（需要使用 root 进行查询）
                 * 参数二：条件值
                 */
                Predicate pre = cb.equal(root.get("name"),"张三");

                return pre;
            }
        };

        List<Emp> list = empRepository.findAll(spec);
```

```
        for (Emp emp : list) {
            System.out.println(emp);
        }
    }
```

5.8.2. 多个条件

```
// 使用 JPASpecificationExecutor 接口的方法（多个条件）
    @Test
    public void testFindAllJPASpecificationExecutor2() {
        Specification<Emp> spec = new Specification<Emp>() {
            @Override
            public Predicate toPredicate(Root<Emp> root, CriteriaQuery<?> query,
CriteriaBuilder cb) {
                // where name = ? and gender = ?
                List<Predicate> preList = new ArrayList<Predicate>();

                preList.add( cb.equal(root.get("name"), "张三") );
                preList.add( cb.equal(root.get("gender"), "男") );

                Predicate[] preArray = new Predicate[preList.size()];
                return cb.and(preList.toArray(preArray));
            }
        };

        List<Emp> list = empRepository.findAll(spec);
        for (Emp emp : list) {
            System.out.println(emp);
        }
```

```
    }
```

5.9. 实现一对多关联

需求：员工　和 部门　是多对一。

5.9.1. 员工类

```
/**
 * 多方
 * @author lenovo
 *
 */
@Entity
@Table(name="t_emp")
public class Emp {

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

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

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

    @Column(name="telephone")
```

```
    private String telephone;


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


    //关联部门（1 方）
    @ManyToOne
    //@JoinColumn：维护外键字段
    @JoinColumn(name="dept_id")
    private Dept dept;
```

5.9.2. 部门类

```
/**
 * 一方
 * @author lenovo
 *
 */
@Entity
@Table(name="t_dept")
public class Dept {


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


    @Column(name="dept_name")
    private String deptName;
```

```
    //关联员工（多方）
    @OneToMany(mappedBy="dept")
    private Set<Emp> emps = new HashSet<Emp>();
```

5.9.3. 一对多关联操作

```
package cn.sm1234.test;


import javax.annotation.Resource;


import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;


import cn.sm1234.Application;
import cn.sm1234.dao.EmpRepository;
import cn.sm1234.domain.Dept;
import cn.sm1234.domain.Emp;


@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes=Application.class)
public class OneToManyTest {


    @Resource
    private EmpRepository empRepository;


    //添加操作
```

```
@Test
public void testSave(){
    //创建部门
    Dept dept = new Dept();
    dept.setDeptName("秘书部");

    //创建员工
    Emp emp = new Emp();
    emp.setName("小红");

    //关联
    dept.getEmps().add(emp);
    emp.setDept(dept);

    //保存数据
    empRepository.save(emp);
}

//查询
@Test
public void testFind(){
    //查询员工
    Emp emp = empRepository.findOne(9);

    //所在部门
    Dept dept = emp.getDept();
    System.out.println("员工"+emp.getName()+"的部门是：  "+dept.getDeptName());
}
```

```
}
```

5.10. 实现多对多关联

需求：用户 和 角色是多对多关联。

5.10.1. 用户类

```
/**
 * 用户
 * @author lenovo
 *
 */
@Entity
@Table(name="t_user")
public class User {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id")
    private Integer id;

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

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

    //关联角色
```

```
    @ManyToMany
    //@JoinTable: 映射中间表
    //  joinColumns: 当前表在中间表的外键字段
    @JoinTable(name="t_user_role",joinColumns=@JoinColumn(name="user_id"),inverseJoinColumns=@JoinColumn(name="role_id"))
    private Set<Role> roles = new HashSet<Role>();
```

5.10.2. 角色类

```
/**
 * 角色
 * @author lenovo
 *
 */
@Entity
@Table(name="t_role")
public class Role {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id")
    private Integer id;

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

    //关联用户
    @ManyToMany(mappedBy="roles")
    private Set<User> users = new HashSet<User>();
```

5.10.3. 实现多对多关联操作

```
/**
 * 多对多关联操作
 * @author lenovo
 *
 */
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes=Application.class)
public class ManyToManyTest {

    @Resource
    private UserRepository userRepository;

    //添加
    @Test
    public void testSave(){
        //创建用户
        User user = new User();
        user.setName("eric");
        user.setPassword("1234");

        //创建角色
        Role role = new Role();
        role.setName("超级管理员");

        Role role2 = new Role();
        role2.setName("普通管理员");
```

```
        //关联
        user.getRoles().add(role);
        user.getRoles().add(role2);
        role.getUsers().add(user);
        role2.getUsers().add(user);

        //保存数据
        userRepository.save(user);
    }

    //查询
    @Test
    public void testFind(){
        User user = userRepository.findOne(1);
        Set<Role> roles = user.getRoles();
        System.out.println("用户: "+user.getName()+"的角色为");
        for (Role role : roles) {
            System.out.println(role.getName());
        }
    }
}
```

6. SpringBoot 缓存实现

6.1. 整合 Ehcache

6.1.1. 导入缓存相关的坐标

```
<!-- 缓存坐标 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-cache</artifactId>
        </dependency>
        <!-- Ehcache 支持 -->
        <dependency>
            <groupId>net.sf.ehcache</groupId>
            <artifactId>ehcache</artifactId>
        </dependency>
```

6.1.2. 配置 ehcache.xml

该文件通常放在 resources 目录下

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../config/ehcache.xsd">

    <diskStore path="java.io.tmpdir"/>

    <!-- defaultCache:  默认配置 -->
    <defaultCache
            maxElementsInMemory="10000"
```

```
            eternal="false"
            timeToIdleSeconds="120"
            timeToLiveSeconds="120"
            maxElementsOnDisk="10000000"
            diskExpiryThreadIntervalSeconds="120"
            memoryStoreEvictionPolicy="LRU">
        <persistence strategy="localTempSwap"/>
    </defaultCache>

    <!-- 缓存名称为 emp 的配置 -->
    <cache name="emp"
            maxElementsInMemory="10000"
            eternal="false"
            timeToIdleSeconds="120"
            timeToLiveSeconds="120"
            maxElementsOnDisk="10000000"
            diskExpiryThreadIntervalSeconds="120"
            memoryStoreEvictionPolicy="LRU">
        <persistence strategy="localTempSwap"/>
    </cache>


</ehcache>
```

6.1.3. 配置 application.properties

```
spring.datasource.driverClassName=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/test
spring.datasource.username=root
```

```
spring.datasource.password=root


spring.datasource.type=com.alibaba.druid.pool.DruidDataSource


spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true


spring.cache.ehcache.config=ehcache.xml
```

6.1.4. 在启动类加上注解

```
package cn.sm1234;


import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cache.annotation.EnableCaching;


@SpringBootApplication

@EnableCaching // 启用缓存

public class Application {


    public static void main(String[] args) {

        SpringApplication.run(Application.class, args);

    }

}
```

6.1.5. 在指定的方法上使用缓存注解

```
@Override
    @Cacheable(value="emp")   //@Cacheable:把当前方法的返回值放入缓存,value 属性：缓存配置的名称
    public Emp findById(Integer id) {
        return EmpRepository.findOne(id);
    }
```

6.1.6. 测试代码

```
package cn.sm1234.test;

import javax.annotation.Resource;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import cn.sm1234.Application;
import cn.sm1234.service.EmpService;

@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes=Application.class)
public class EmpServiceTest {

    @Resource
    private EmpService empService;
```

```
    @Test
    public void testFindById(){
        //查询第一次
        System.out.println(empService.findById(1));
        //查询第二次
        System.out.println(empService.findById(1));
    }
}
```

注意：这时发生错误：

```
-03-06 10:49:36.818 ERROR 17828 --- [       emp.data] n.s.e.store.disk.DiskSt

.io.NotSerializableException: cn.sm1234.domain.Emp
    at java.io.ObjectOutputStream.writeObject0(Unknown Source) ~[na:1.8.0_31]
    at java.io.ObjectOutputStream.defaultWriteFields(Unknown Source) ~[na:1.8
    at java.io.ObjectOutputStream.defaultWriteObject(Unknown Source) ~[na:1.8
    at net.sf.ehcache.Element.writeObject(Element.java:875) ~[ehcache-2.10.4.
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) ~[na:1.8.0
```

原因：无法序列化 Emp 对象

解决办法：让 Emp 对象实现 Serializable 接口

6.2. @Cacheable 和@CacheEvict

@Cacheable：把方法返回值放入缓存

value：ehcache.xml 的缓存配置名称

```
<!-- 缓存名称为emp的配置 -->
<cache name="emp"
       maxElementsInMemory="10000"
       eternal="false"
       timeToIdleSeconds="120"
       timeToLiveSeconds="120"
       maxElementsOnDisk="10000000"
       diskExpiryThreadIntervalSeconds="120"
       memoryStoreEvictionPolicy="LRU">
    <persistence strategy="localTempSwap"/>
</cache>
```

key：给缓存值起个名称，只要该查询同一个名称的数据缓存都有效。

@CacheEvict：把数据从缓存清除

```
@Override
    @CacheEvict(value="emp",allEntries=true)
    public void save(Emp emp) {
        EmpRepository.save(emp);
    }
```

6.3. 整合 Spring Data Redis

Spring Data Redis 是属于 Spring Data 项目的模块。作用是用于简化 redis 技术开发。

6.3.1. 导入 spring data redis 坐标

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

6.3.2. 编写整合 redis 配置类（*）

```
package cn.sm1234.config;


import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.connection.jedis.JedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.serializer.StringRedisSerializer;


import redis.clients.jedis.JedisPoolConfig;
```

```
/**
 * 整合 Spring Data Redis 的配置类
 * @author lenovo
 *
 */
@Configuration
public class RedisConfig {


    /**
     * 1.创建 JedisPoolConfig 对象：连接池参数
     * @return
     */
    @Bean
    public JedisPoolConfig getJedisPoolConfig(){
        JedisPoolConfig poolConfig = new JedisPoolConfig();
        //最大空闲数
        poolConfig.setMaxIdle(5);
        poolConfig.setMinIdle(3);
        //最大连接数
        poolConfig.setMaxTotal(10);
        return poolConfig;
    }


    /**
     * 2.创建 JedisConnectionFactory: 配置 redis 连接参数
     */
    @Bean
    public JedisConnectionFactory getJedisConnectionFactory(JedisPoolConfig
poolConfig){
```

```
        JedisConnectionFactory connectionFactory = new JedisConnectionFactory();


        //关联
        connectionFactory.setPoolConfig(poolConfig);


        //redis 主机地址
        connectionFactory.setHostName("localhost");
        //redis 端口
        connectionFactory.setPort(6379);
        //redis 数据库索引
        connectionFactory.setDatabase(1);


        return connectionFactory;
    }


    /**
     * 3.创建 RedisTemplate:  用于执行 redis 的操作方法
     */
    @Bean
    public RedisTemplate<String, Object> redisTemplate(JedisConnectionFactory
connectionFactory){
        RedisTemplate<String, Object> redisTemplate = new RedisTemplate<String,
Object>();


        //关联 connectionFactory
        redisTemplate.setConnectionFactory(connectionFactory);


        //设置 key 序列化
        redisTemplate.setKeySerializer(new StringRedisSerializer());
```

```
        //设置 value 序列化
        redisTemplate.setValueSerializer(new StringRedisSerializer());

        return redisTemplate;
    }

}
```

6.3.3. 编写测试方法存取数据

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes = Application.class)
public class RedisTest {

    @Resource
    private RedisTemplate<String, Object> redisTemplate;

    // 存入字符串
    @Test
    public void testSet() {
        redisTemplate.opsForValue().set("name", "sm1234@qq.com");
    }

    // 取出字符串
    @Test
    public void testGet() {
        String name = (String)redisTemplate.opsForValue().get("name");
        System.out.println(name);
```

```
    }
}
```

6.4. 抽取 redis 的参数

6.4.1. 编写 application.properties

```
spring.redis.host=localhost
spring.redis.port=6379
spring.redis.database=1


spring.redis.pool.max-idle=5
spring.redis.pool.min-idle=3
spring.redis.pool.max-active=10
```

6.5. 操作不同数据类型

6.5.1. 存取 JavaBean 类型

```
// 存入 JavaBean:JdkSerializationRedisSerializer
    @Test
    public void testSetJavaBean() {
        Emp emp = new Emp();
        emp.setName("张三");
        emp.setGender("男");
```

```
        //重新设置 value 序列化区
        redisTemplate.setValueSerializer(new JdkSerializationRedisSerializer());

        redisTemplate.opsForValue().set("emp", emp);
    }

    //取出 JavaBean
    @Test
    public void testGetJavaBean() {
        //重新设置 value 序列化区
        redisTemplate.setValueSerializer(new JdkSerializationRedisSerializer());

        Emp emp = (Emp)redisTemplate.opsForValue().get("emp");
        System.out.println(emp);
    }
```

6.5.2. 以 JSON 格式存储 JavaBean

```
    // 以 JSON 格式存入 JavaBean
    @Test
    public void testSetJavaBeanUserJson() {
        // 重新设置 value 序列化区
        redisTemplate.setValueSerializer(new
Jackson2JsonRedisSerializer<>(Emp.class));

        Emp emp = new Emp();
        emp.setName("李四");
        emp.setGender("男");
```

```
        redisTemplate.opsForValue().set("emp_json", emp);
    }

    // 以 JSON 格式取出 JavaBean
    @Test
    public void testGetJavaBeanUserJson() {
        // 重新设置 value 序列化区
        redisTemplate.setValueSerializer(new
Jackson2JsonRedisSerializer<>(Emp.class));

        Emp emp = (Emp) redisTemplate.opsForValue().get("emp_json");
        System.out.println(emp);
    }
```

7. SpringBoot 实现定时任务

7.1. @Scheduled 注解

7.1.1. 导入 spring-context-support 坐标

```
<!-- sping 对 schedule 的支持 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context-support</artifactId>
        </dependency>
```

7.1.2. 编写自定义任务调度类

```
package cn.sm1234.schedule;
```

```
import java.util.Date;


import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;


@Component
public class MySchedule {


    //每隔 3 秒执行一次该任务
    //@Scheduled:用于设置定时任务
    //conn 属性：cron 表达式（字符串格式）
    @Scheduled(cron="0/3 * * * * ?")
    public void test1(){
        System.out.println("执行任务："+new Date());
    }
}
```

7.1.3. 在启动类开启任务调用

```
package cn.sm1234;


import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.scheduling.annotation.EnableScheduling;


@SpringBootApplication
@EnableScheduling
public class Application {
```

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

7.2. cron 表达式语法

cron 表达式是一个字符串，分为 6 个或 7 个值（中间使用空格分隔）。

位置	时间域名	允许值	允许的特殊字符
1	秒	0-59	, - * /
2	分钟	0-59	, - * /
3	小时	0-23	, - * /
4	日	1-31	, - * / L W C
5	月	1-12	, - * /
6	星期	1-7	, - * ? / L C #
7	年(可选)	1970-2099	, - * /

Cron 表达式的时间字段除允许设置数值外，还可使用一些特殊的字符，提供列表、范围、通配符等功能，细说如下：

●星号(*)：可用在所有字段中，表示对应时间域的每一个时刻，例如，*在分钟字段时，表示“每分钟”；

●问号（?）：该字符只在日期和星期字段中使用，它通常指定为“无意义的值”，相当于点位符；

●减号(-)：表达一个范围，如在小时字段中使用“10-12”，则表示从 10 到 12 点，即 10,11,12；

●逗号(,)：表达一个列表值，如在星期字段中使用“MON,WED,FRI”，则表示星期一，星期三和星期五；

●斜杠(/)：x/y 表达一个等步长序列，x 为起始值，y 为增量步长值。如在分钟字段中使用 0/15，则表示为 0,15,30 和 45 秒，而 5/15 在分钟字段中表示 5,20,35,50，你也可以使用*/y，它等同于 0/y；

●L：该字符只在日期和星期字段中使用，代表“Last”的意思，但它在两个字段中意思不同。L 在日期字

段中，表示这个月份的最后一天，如一月的 31 号，非闰年二月的 28 号；如果 L 用在星期中，则表示星期六，等同于 7。但是，如果 L 出现在星期字段里，而且在前面有一个数值 X，则表示“这个月的最后 X 天”，例如，6L 表示该月的最后星期五；

●W：该字符只能出现在日期字段里，是对前导日期的修饰，表示离该日期最近的工作日。例如 15W 表示离该月 15 号最近的工作日，如果该月 15 号是星期六，则匹配 14 号星期五；如果 15 日是星期日，则匹配 16 号星期一；如果 15 号是星期二，那结果就是 15 号星期二。但必须注意关联的匹配日期不能够跨月，如你指定 1W，如果 1 号是星期六，结果匹配的是 3 号星期一，而非上个月最后的那天。W 字符串只能指定单一日期，而不能指定日期范围；

●LW 组合：在日期字段可以组合使用 LW，它的意思是当月的最后一个工作日；

●井号(#)：该字符只能在星期字段中使用，表示当月某个工作日。如 6#3 表示当月的第三个星期五(6 表示星期五，#3 表示当前的第三个)，而 4#5 表示当月的第五个星期三，假设当月没有第五个星期三，忽略不触发；

● C：该字符只在日期和星期字段中使用，代表“Calendar”的意思。它的意思是计划所关联的日期，如果日期没有被关联，则相当于日历中所有日期。例如 5C 在日期字段中就相当于日历 5 日以后的第一天。1C 在星期字段中相当于星期日后的第一天。

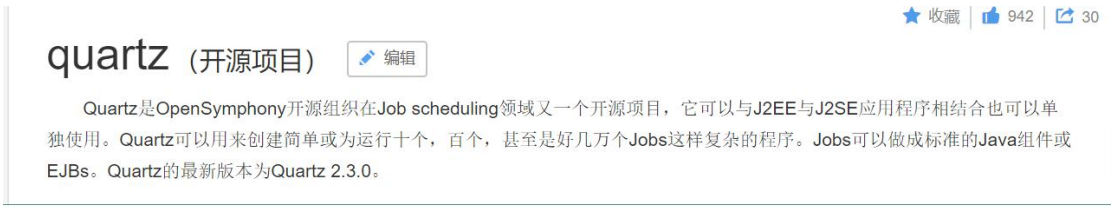
Cron 表达式对特殊字符的大小写不敏感，对代表星期的缩写英文大小写也不敏感。

例子：

```
@Scheduled(cron = "0 0 1 1 1 ?")//每年一月的一号的 1:00:00  执行一次

@Scheduled(cron = "0 0 1    1 1,6 ?") //一月和六月的一号的 1:00:00  执行一次

@Scheduled(cron = "0 0 1    1 1,4,7,10 ?") //每个季度的第一个月的一号的 1:00:00  执行一次

@Scheduled(cron = "0 0 1    1 * ?")//每月一号  1:00:00  执行一次

@Scheduled(cron = "0 0 1 ? * MON")    //每周周一  1:00:00  执行一次

@Scheduled(cron="0 0 1 * * *") //每天凌晨 1 点执行一次
```

7.3. 整合 Quartz

7.3.1. Quartz 框架简介



Quartz 应用思路：

1）Job - 任务 - 你要干什么？

2）Trigger - 触发器- 你什么时候干？

3）Scheduler - 任务调度- 你什么时候需要干什么？

7.3.2. Quartz 的 Java 应用（了解）

7.3.2.1. 导入 quartz 坐标

```
<!-- Quartz 支持 -->
        <dependency>
            <groupId>org.quartz-scheduler</groupId>
            <artifactId>quartz</artifactId>
            <version>2.2.1</version>
            <exclusions>
                <exclusion>
                    <artifactId>slf4j-api</artifactId>
                    <groupId>org.slf4j</groupId>
                </exclusion>
            </exclusions>
        </dependency>
```

7.3.2.2. 定义 Job 任务类

```
package cn.sm1234.quartz;


import java.util.Date;


import org.quartz.Job;
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;


/**
 * 自定义 Job 类
 * @author lenovo
 *
 */
public class MyJob implements Job{


    //任务被触发时被执行
    @Override
    public void execute(JobExecutionContext context) throws JobExecutionException {
        System.out.println("任务被执行:"+new Date());
    }


}
```

7.3.2.3. 编写代码创建任务调度程序

```
package cn.sm1234.quartz;
```

```
import org.quartz.JobBuilder;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.SimpleScheduleBuilder;
import org.quartz.Trigger;
import org.quartz.TriggerBuilder;
import org.quartz.impl.StdSchedulerFactory;

public class QuartzMain {

    public static void main(String[] args) throws Exception {
        //1.创建 Job 对象 - 你需要干什么?
        JobDetail job = JobBuilder.newJob(MyJob.class).build();

        //2.创建 Trigger 对象 - 你什么时候干?
        /**
         * 简单 trigger: 简单地重复
         * cron trigger: 按照 cron 表达式
         */
        Trigger trigger = TriggerBuilder
                            .newTrigger()

    .withSchedule(SimpleScheduleBuilder.repeatSecondlyForever())
                            .build();

        //3.创建 Scheduler
        Scheduler scheduler = StdSchedulerFactory.getDefaultScheduler();
```

```
        scheduler.scheduleJob(job, trigger);


        //4.启动 Scheduler
        scheduler.start();
    }
}
```

7.3.3. SpringBoot 整合 Quartz（*）

```
package cn.sm1234.quartz;


import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.scheduling.quartz.JobDetailFactoryBean;
import org.springframework.scheduling.quartz.SchedulerFactoryBean;
import org.springframework.scheduling.quartz.SimpleTriggerFactoryBean;


@Configuration
public class QuartzConfig {


    /**
     * 1.创建 Job 对象
     */
    @Bean
    public JobDetailFactoryBean getJobDetailFactoryBean(){
        JobDetailFactoryBean factory = new JobDetailFactoryBean();


        //关联我们定义 Job 类
```

```
        factory.setJobClass(MyJob.class);


        return factory;
    }


    /**
     * 2.创建 Trigger
     */
    @Bean
    public SimpleTriggerFactoryBean getSimpleTriggerFactoryBean(JobDetailFactoryBean
jobDetailFactoryBean){
        SimpleTriggerFactoryBean factory = new SimpleTriggerFactoryBean();


        //关联 JobDetail 对象
        factory.setJobDetail(jobDetailFactoryBean.getObject());


        //重复间隔时间（毫秒为单位）
        factory.setRepeatInterval(5000);


        //重复次数
        factory.setRepeatCount(4);


        return factory;
    }



    /**
     * 3.创建 Scheduler 对象
     */
```

```
    @Bean
    public SchedulerFactoryBean getSchedulerFactoryBean(SimpleTriggerFactoryBean
triggerFactoryBean){
        SchedulerFactoryBean factory = new SchedulerFactoryBean();

        //关联 trigger
        factory.setTriggers(triggerFactoryBean.getObject());

        return factory;
    }

}
```